SANDIA REPORT
SAND2021-2961
Unclassified Unlimited Release
Printed March 16, 2021

Sandia National Laboratories

Sierra/SolidMechanics 5.0 User's Guide

SIERRA Solid Mechanics Team
Computational Solid Mechanics and Structural Dynamics Department
Engineering Sciences Center

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods





ABSTRACT

Sierra/SolidMechanics (Sierra/SM) is a Lagrangian, three-dimensional code for finite element analysis of solids and structures. It provides capabilities for explicit dynamic, implicit quasistatic and dynamic analyses. The explicit dynamics capabilities allow for the efficient and robust solution of models with extensive contact subjected to large, suddenly applied loads. For implicit problems, Sierra/SM uses a multi-level iterative solver, which enables it to effectively solve problems with large deformations, nonlinear material behavior, and contact. Sierra/SM has a versatile library of continuum and structural elements, and a large library of material models. The code is written for parallel computing environments enabling scalable solutions of extremely large problems for both implicit and explicit analyses. It is built on the SIERRA Framework, which facilitates coupling with other SIERRA mechanics codes. This document describes the functionality and input syntax for Sierra/SM.

ACKNOWLEDGMENTS

This document is the result of the collective effort of many individuals. The current core development team responsible for the Sierra/SM codes includes Frank N. Beckwith, Guy L. Bergel, Gabriel J. de Frias, Kevin L. Manktelow, Mark T. Merewether, Scott T. Miller, Matthew D. Mosby, Julia A. Plews, Vicki L. Porter, Timothy R. Shelton, Jesse D. Thomas, Benjamin C. Treweek, Michael R. Tupek, Michael G. Veilleux, and Ellen B. Wagman. This document is written and maintained by this team.

Many others have contributed to this document, either directly or indirectly. These include, but are not limited to Kenneth (Noel) Belcourt, Manoj K. Bhardwaj, Nicole L. Breivik, Arthur Brown, James V. Cox, Nathan K. Crane, David M. Day, J. Franklin Dempsey, James W. Foulk III, Scott Gampert, Brian D. Giffin, Steven P. Gomez, Jeffrey D. Gruda, Arne S. Gullerud, Jason D. Hales, Daniel C. Hammerand, Martin W. Heinstein, David M. Hensinger, Michael C. Hillman, Jason T. Ivey, Jacob Koester, Timothy D. Kostka, J. Richard Koteras, Brian T. Lester, Alex J. Lindblad, David J. Littlewood, Chi S. (David) Lo, Kevin N. Long, Edward Love, Kyran D. (Kim) Mish, Jakob T. Ostien, Kendall H. Pierson, Ramon Reyes, Rhonda K. Reinert, Nathaniel S. Roehrig, William M. Scherzinger, Gregory D. Sjaardema, Benjamin W. Spencer, Brian L. Stevens, Julia R. Walker, Gerald W. Wellman, Patrick G. Xavier, and Edouard Yreux.

SIERRA/SM 5.0 RELEASE NOTES

Following is a list of new features and syntax changes made to Sierra/SM since the 4.58 release.

A new capability to compute the max gap between mesh objects has been developed. See Section 9.3.2.2 for more details.

A shorter syntax is now available for specifying parameters for blocks. Instead of `BEGIN PARAMETERS FOR BLOCK BLOCK_1`, you may now say `BEGIN BLOCK BLOCK_1`.

Interaction specific output for `INTERACTION MASK`, `INTERACTION FORCE CONTACT`, `INTERACTION RELATIVE VELOCITY`, and `INTERACTION CONTACT STATUS` is now available. See Section 8.6.2.7 for more information.

Various new options have been added to the nonlocal averaging capability in user output, including a graph-distance search for averaging nodal variables. See Section 9.3.2.8 for more details.

A `RIGID BODY PROJECTOR` command has been added for use with the Belytschko-Leviathan shell which allows users to specify which rigid body modes to remove from the local force and velocity vectors. See Section 6.2.5 for more information.

A new `INITIAL TEMPERATURE` capability has been added to allow better interaction with material models that may evolve the temperature. See Section 7.10 for more information.

The old coordinate system syntax has been deprecated. Please see Section Section 2.1.9 for the new syntax.

`CONICAL` and `ELLIPSOIDAL` coordinate systems are now supported. See Section 2.1.9 for more details.

The line commands `INITIAL REBALANCE` and `PERIODIC REBALANCE` have been deprecated and replaced with the line command `REBALANCE TYPE`. See Section 6.8.1 for more information.

Equation of state (EoS) support has been added to the Total Lagrange elements, including the Composite Tet10.

The `COPY VARIABLE` command may now be given a specific time to copy the variable (as opposed to all times). See Section 7.3.5 for more details.

Generalized Alpha integration has been implemented for use in implicit dynamics. See Section 4.11.1.2 for more details.

The ability to apply a filter to global variables has been added to user output. See Section 9.3.2.7 for more details.

Deprecated features since the 4.58 release.

The compliant-joint cohesive model has been deprecated.

The line command `FORCE VALID ACME CONNECTIVITY` has been deprecated from element death.

The ability to initialize temperature through an initial condition block has been deprecated. Use the `INITIAL TEMPERATURE` instead. See Section 7.10 for more information.

The `T0` command in modular material models has been removed. The material models now rely on the temperature initialized in the `INITIAL TEMPERATURE` boundary condition.

Fixes, performance, and robustness improvements made to Sierra/SM since the 4.58 release.

An issue was fixed with the wave speed calculation in the Total Lagrange Hex element. Now, the wave speed depends on the current element density.

Point masses now work with implicit time integration.

An issue was fixed that involved tied contact on elements that have undergone element-to-particle conversion. Applying tied contact on the new particles created an unstable result. Now, the newly created particles will use a frictionless material model after conversion.

An issue was fixed with MPC's in current coordinates. When adding an MPC after a restart, the MPC was previously not initialized correctly.

An issue was fixed with the absolute search specified in global nodal transfers. Previously, the absolute search tolerance used was the one used in the first transfer block specified in the input file. This was the case even if the default absolute search tolerance was used in the first transfer block and an absolute search tolerance was specified later by another transfer block. Now, the max of the absolute search tolerances specified is used as the global nodal absolute search tolerance. See Section 11.2 for more details.

Performance has been improved for the variable interpolation capability. See Section 9.8 for more details.

Performance has been improved for the closest distance calculation. See Section 9.3.2 for more details.

Performance has been improved for the element nonlocal averaging capability. See Section 9.3.2 for more details.

Following is a list of new features and syntax changes made to Sierra/SM since the 4.56 release.

Initial mesh modification may now be turned off for subsequent restart runs. This can be controlled by the `APPLY AFTER RESTART` command. See Section 7.2 for more details.

Stabilization has been added to the 10-noded composite tetrahedron element to avoid displacement instabilities associated with volume-averaging. See Section 6.2.2 for more details.

A `LAYER THICKNESS` may now be specified for cohesive elements. This is useful for free floating cohesive elements that are MPC'd back to the rest of model. See Section 6.2.3 for more details.

A new user output capability to compute the closest and furthest distance between mesh objects has been developed. See Section 9.3.2.2 for more details.

The nodal nonlocal averaging capability can now be used with a distance weighting function. Additionally, the nonlocal domain can now be visualized via the `nonlocal_domains` global variable. See Section 9.3.2.8 for more details.

The syntax in contact, MPC's, Shell-Solid Joint, and periodic boundary conditions has changed. The new syntax uses `SIDE A / SIDE B` to specify individual interactions. A python script is available to help convert files to use the new syntax; please contact sierra-help if you would like a copy. See Sections 8.6, 7.15.9, 7.15.5, and 7.4.5.

Additional logfile output has been added for line welds and spot welds.

Various robustness improvements have been made to the preload capability. See Section 7.15.12 for more information on how to use the capability.

The variable `LAME_STATE_<model_name>` has been modified. It is now required to specify how many state variables the model has. For example, `LAME_STATE_DSA` is now `LAME_STATE_69_DSA`.

Various shape metrics are now available for output for the Tet10 elements. See Section 2.5 for a complete list of available shape metrics.

Deprecated features since the 4.56 release.

Some experimental solver options have been deprecated.

Fixes, performance, and robustness improvements made to Sierra/SM since the 4.56 release.

Layered shell elements can now be used in conjunction with element death.

Performance has been significantly improved for problems involving contact and mesh modification (such as element death).

Improved performance of element death.

Various other performance improvements have been made to Sierra/SM.

Various improvements have been made to the Composite Tet10 element, making it more robust in combination with other capabilities.

Various improvements have been made to the implicit solver.

The user output capability to perform an element average has been improved. The previous syntax `COMPUTE ELEMENT <elem_var> AS AVERAGE OF ELEMENT` is now the same as `COMPUTE ELEMENT <elem_var> AS VOLUME WEIGHTED AVERAGE OF ELEMENT`. A new un-weighted average is now available `COMPUTE ELEMENT <elem_var> AS UNWEIGHTED AVERAGE OF ELEMENT`, which is significantly faster than the volume weighted average (especially for multi-integration point elements). See Section 9.3.2.2 for more details

Contact with lofted beams and restart has been made more robust.

Fixed a number of issues with element death and particle conversion.

Fixed issues with restart and negative times.

The BT Shell element has been made more robust for problems involving large deformation and hourglass modes.

Following is a list of major documentation changes since the 4.56 release.

Explicit quasistatic mode has been moved from the User's Guide to the Capabilities In-Development Manual.

A new example problem has been added to the Example Problem's Manual for computing a global average over a specified sub domain. See the Example Problems Manual for more details.

Following is a list of new features and syntax changes made to Sierra/SM since the 4.54 release.

The line weld can now output weld forces and moments in the local coordinate system with the `line_weld_force_rst` and `line_weld_moment_rst` variables. See Sections 7.15.3.2 and 9.10.1.

New options are available to the user output `COPY` command for setting nodal variables to be the average, sum, max, or weighted sum of the values of connected elements. See Section 9.3.2.4.

A nonlocal averaging option has been added to user output. The averaged quantity may be a `GLOBAL` variable centered at a `POINT`, an `ELEMENT` variable weighted by volume, or a `NODAL` variable weighted by mass. In each case, the specified variable is averaged over a nonlocal domain with finite support. See Section 9.3.2.2.

Users can now specify a `CONTACT SHELL THICKNESS SCALE FACTOR` for lofted shell contact.

This option will compute the CONTACT SHELL THICKNESS by taking the shell thickness used for the element mechanics and multiplying that value by this scale factor. See Section 8.9.5.

A DENSITY SCALE FACTOR has been added to block parameters. This command scales the density of the material before applying it to the block density. Users may find this command useful if they wish to modify the density for a given block without duplicating the entire material model. See Section 6.1.7.1.

An analytic sphere option has been added for lofted particle contact as an alternative to lofted icosahedra. The option is currently available only for node-face contact. See Section 8.9.5.2.

A BEGIN PRELOAD command has been introduced to greatly simplify bolt preloading, reducing line count for preload syntax from $O(100)$ to $O(5)$. See Section 7.15.12.

Internal forces and reactions may now be specified in the SHEAR and NORMAL directions. These correspond to the tangential and normal reactions, respectively, with respect to the plane defining the internal cut section. See Section 9.3.2.2.

Added ability to use a user-defined DIRECTION with the internal reaction command in user output. See Section 9.3.2.2.

Added stiffness_hourglass_energy and viscous_hourglass_energy as new GLOBAL and ELEMENT variables for output. These variables are currently only available for uniform gradient hexahedra, fiber shells, and fiber membranes. See Section 9.10.1.

Added external_viscous_energy and internal_viscous_energy as new GLOBAL variables for output. See Section 9.10.1.

Added a new GLUED NORMAL contact definition that permits independent motion between interacting faces before initial contact. Upon contact, the surfaces become "glued" together, which constrains the faces such that no more relative normal or tangential motion can occur. See Section 8.5.8.

A user function may now be used with velocity damping in the VISCOUS DAMPING command block. See Section 7.15.5.2.

Green-Lagrange strain and related strain measures are now available for output. See Section 9.10.

Added a new Zapotec variable ZAP_LOST_MASS to account for mass lost due to attempted insertion/donation of inverted elements. See the Zapotec Manual for more details.

Added an IGNORE_ALL_GAPS option to Zapotec to zero the stress when the forward cell contains any Lagrangian material. This preserves the IGNORE_GAPS behavior from previously released versions. See the Zapotec Manual for more details.

COPY VARIABLE has been added for nodal scalar and vector quantities between dissimilar meshes for TIME STEP INITIALIZATION, BEGIN PRESCRIBED FORCE, and BEGIN TRACTION. See Section 7.3.5 for more information on copy variable.

ITERATION PRINT has been added to the FULL TANGENT PRECONDITIONER. See section 4.3 for more information on the Full Tangent Preconditioner.

A new stabilization for the composite tetrahedral element is now available to better control

displacement instabilities. See Section 6.2.2 for more information.

A new Conforming Reproducing Kernel (CRK) method has been added with some preliminary capabilities. The method is still under active development. See the Chapter 18 of the Capabilities-In-Development manual for more information.

Deprecated features since the 4.54 release.

MPC specification using `BEGIN MPC` is now deprecated and has been replaced with `BEGIN TIED MPC` and `BEGIN EQUATION MPC`. See Sections 7.15.8 and 7.15.9.

The Forte product coupling Alegra and Presto has been deprecated.

The input command `THERMAL STRAIN FUNCTION` has been deprecated. Instead, use the `THERMAL LOG STRAIN FUNCTION` or `THERMAL ENGINEERING STRAIN FUNCTION` commands. See Section 5.1.4 for more details.

The line command `CONSTRAINT ENFORCEMENT = OPTION` has been deprecated from the `FULL TANGENT PRECONDITIONER`. See section 4.3 for more information on the Full Tangent Preconditioner.

The `ELEMENT JACOBIAN` command has been removed from the Total Lagrange section because it was unnecessary. See the In-Development manual for more information on the Total Lagrange section, or Section 6.2.2 for more information about the Composite Tet10 Total Lagrange element.

The `INITIAL DEFORMATION GRADIENT` command has been removed from the Total Lagrange section because it was unnecessary. See the In-Development manual for more information on the Total Lagrange section, or Section 6.2.2 for more information about the Composite Tet10 Total Lagrange element.

Using `BEGIN ANALYTIC PLANE` or `BEGIN ANALYTIC CYLINDER` with Dash contact has been deprecated. The `BEGIN ANALYTIC OBJECT` syntax should be used instead. See Section 8.8 for more detail about contact with analytic objects.

Fixes, performance, and robustness improvements made to Sierra/SM since the 4.54 release.

A bug was fixed in MPCs that caused non-deterministic iteration behavior. This behavior would arise when resolving chained MPCs. See section 7.15.8.1 for more information on chained MPCs.

Improvements were made to transfers to prevent nodal transfers between blocks which did not share nodes, e.g., between two blocks in contact. Syntax changes were introduced to specify whether nodal transfers should interpolate block-by-block, over the blocks defined in the same transfer, or over all of the blocks in the analysis. See Section 11.2.

A segmentation fault in adaptive time stepping was identified and fixed. See Section 8.8 for more information on adaptive time stepping.

The code now properly errors if both the ARTIFICIAL ENGINEERING STRAIN and THERMAL STRAIN are specified within the same material definition. Specifying both results in an ambiguous strain and caused an MPI failure which is now prevented. See Sections 5.1.3 and 5.1.4 for more detail on artificial and thermal strains.

Fixed transfer logic in coupled thermal-mechanical problems which led to non-convergent behavior in implicit. See the Sierra Code Coupling Module: Arpeggio manual for more details on coupled thermal-mechanical problems.

Fixed issue where mesh attributes were not properly initialized during a restarted analysis. This caused anomalous behavior with rigid bodies during restart.

Fixed an issue where field variables were not properly copied to restart files, resulting in zero-fields and causing excessive slowdowns. See Section 8.8 for more information about restart.

The LOFTING FACTOR MESH VARIABLE value was not retained during a restarted analysis. Introduced fix to properly reinitialize the value on restart. See Sections 6.2.5 and 6.2.7 for more details about the LOFTING FACTOR MESH VARIABLE command.

Added error if an active enclosing block and an embedded block are contiguously meshed in a solver submodel. See Section 7.15.10 for more details about the submodel capability.

Fixed issue where rigid bodies were ignored if the same name (including empty strings) was assigned to multiple rigid bodies. See Section 6.3.1 for more information on rigid bodies.

Improved formatting of columns in logfile when long block names are present.

Fixed crash when the command OUTPUT MESH = EXPOSED SURFACE was specified within BEGIN RESULTS OUTPUT. See Section 9.3.1.6 for more details.

The error message for line welds indicating no shells were found within the search tolerance of the beam now specify the beam block where the error originated. See Section 9.3.1.6 for more information on line welds.

Fixed bug in constraint enforcement between MPCs and Dash contact.

The solid_angle calculation for Tet10 elements has been fixed. See Section 2.5 for more information on the solid_angle and other distortion metrics.

Element death has been fixed when MATERIAL CRITERION = ON was specified for a material model that did not have a material criterion. See Section 2.5 for more details about material death criterion.

Artificial strain can now be set with any subset of R, S, or T functions. See Section 2.5 for more details on Artificial Strain.

Elements are now correctly killed at initialization after transfer (to account for elements that have inverted after the transfer). See Section 2.5 for more details on inter-procedural transfers.

Contact area is now computed correctly for the Composite Tet10 element. Section 6.2.2 for more information on the Composite Tet.

LINE SEARCH = BACKTRACK now correctly works with volume exceptions allowing a timestep

cutback. See Section 4.2.3 for more details on the line search command.

A bug was fixed with overlap removal and Total Lagrange elements. See Section 8.9.4 for more details on overlap removal.

A bug was fixed in variable interpolation that created potentially polluted results. See Section 9.8 for more details on variable interpolation.

Modified the `IGNORE_GAPS` option in Zapotec. `IGNORE_GAPS` now sets the stress state to zero when the forward cell contains a Lagrangian material different than itself. See the Zapotec Manual for more details.

Following is a list of new features and syntax changes made to Sierra/SM since the 4.52 release.

A `NONLOCAL AVERAGE` user output command has been added to create global, nodal, or element variables that are averages of nodal or element variables inside a spherical domain. See Section 9.3.2.2 for more details.

`LAGRANGE ACCEPTABLE RELATIVE GAP` and `LAGRANGE ACCEPTABLE GAP` options have been added to the control contact solver to ensure that the penetration or gap is sufficiently removed during each load step. A default `LAGRANGE ACCEPTABLE RELATIVE GAP` is now used for implicit contact. See Section 4.5.1.3 for more details.

Convergence metrics for augmented Lagrange control contact are now stored as global variables. See Table 9-3 for the list of variables.

A `STRESS MEASURE` option has been added for the mean-quadrature eight-noded hexahedral element with `TOTAL_LAGRANGE` strain incrementation and `HYPERELASTIC` hourglass formulation. The default configuration of the stress has been changed to `CAUCHY` for best compatibility with material models. See Section 6.2.1 for more details.

The ability to control the start time of the fragment identification algorithm has been added to the `IDENTIFY FRAGMENTS` capability (see Section 9.12).

A `LOFTING FACTOR MESH VARIABLE` command has been added to the shell and membrane element sections (Sections 6.2.5 and 6.2.7, respectively) so that lofting factors can be set from any element field stored on the input mesh.

Deprecated features since the 4.52 release.

The `S ROTATION FUNCTION`, `S ROTATION SCALE FACTOR`, `T ROTATION FUNCTION`, and `T ROTATION SCALE FACTOR` commands for the line weld capability (Section 7.15.3.2) are deprecated and scheduled for removal on or after the 4.56 release, because these options over-constrained the weld and conflicted with translational constraints.

The node-based tetrahedron remeshing capability has been removed.

Fixes, performance, and robustness improvements made to Sierra/SM since the 4.52 release.

The time to create volumetric multi-point constraint's (MPCs) has been reduced from hours to seconds for typical use cases. Volumetric MPCs are described in Section 7.15.8.

The robustness of the line weld capability (Section 7.15.3.2) has been improved through development driven by unit and verification testing. Output variables have been added for the line weld forces and moments.

The `INTERPOLATION` inter-procedural transfer has been fixed to correctly handle the `BLOCK BY BLOCK` option. Inter-procedural transfers are described in Section 11.2.

An issue was fixed in the `NODAL FIELD DEPENDENT FRICTION MODEL` (Section 8.5.13) so now it changes between sliding and glued contact immediately when the `contact_transition` field signals the change.

The node-based time step now considers the stiffness contributions of all spot welds connected to nodes; previously, only one spot weld would contribute.

The accuracy of nodal-to-element variable interpolations has been improved for mean-quadrature eight-noded hexahedron elements, because an error in the interpolation shape functions has been fixed. This improves accuracy of interpolation and L_2-projection transfers (Section 11.2) and user output that interpolates from nodes to elements (Section 9.3.2.2).

A warning has been added to notify the user that sidesets cannot be used for contact lofting (Section 8.9.5). The contact surfaces used in lofting must be defined by blocks.

Dynamic coordinate systems (defined by nodes of node sets) previously remained in the reference configuration in implicit quasi-statics and implicit dynamics, but they now update correctly. Dynamic (and static) coordinate systems are described in Section 2.1.9.

Following is a list of new features and syntax changes made to Sierra/SM since the 4.50 release.

Multi-Point Constraints (MPCs, Section 7.15.8) now support higher-order tetrahedral element topologies such as the 10-noded composite tet.

The composite tetrahedral element supports viscous damping (Section 7.15.5.2).

Users may now control the amount of Sierra log file output from element death (see Section 6.5.4.7).

The user output `EXTRAPOLATE` command supports specifying different projection types via the `EXTRAPOLATE PROJECTION TYPE` command. See Section 9.3.2.5 for more details.

Deprecated features since the 4.50 release.

The Fortissimo product is no longer supported.

Volume repulsion (undocumented feature) is deprecated and scheduled for removal on or after the 4.54 release.

The `ENHANCED_STRAIN` element formulation is deprecated and scheduled for removal on or after the 4.54 release.

Fixes, performance, and robustness improvements made to Sierra/SM since the 4.50 release.

The Dash interface material capability has been fixed for the Tvergaard Hutchinson and Thouless Parmigiani surface models. See Section 8.6.2.5 for more details.

Spot welds now evaluate weld failure only at the converged state of an implicit analysis load step rather than at each nonlinear iteration.

Some parallel issues in extrapolating variables on a block-by-block basis were addressed.

Corrected initialization of data filtering (Section 9.3.2.7) with restart and results output.

CONTENTS

1. Introduction **33**
- 1.1. Document Overview . . . 34
- 1.2. Running The Code . . . 36
- 1.3. Overall Input Structure . . . 37
- 1.4. Conventions for Command Descriptions . . . 40
  - 1.4.1. Key Words . . . 40
  - 1.4.2. User-Specified Input . . . 40
  - 1.4.3. Optional Input . . . 41
  - 1.4.4. Default Values . . . 41
  - 1.4.5. Multiple Options for Values . . . 41
  - 1.4.6. Known Issues and Warnings . . . 42
  - 1.4.7. Commands Applicable to Only Explicit or Implicit Analyses . . . 43
- 1.5. Style Guidelines . . . 44
  - 1.5.1. Comments . . . 44
  - 1.5.2. Continuation Lines . . . 44
  - 1.5.3. Capital and Lower Case Letters . . . 44
  - 1.5.4. Commas and Tabs . . . 44
  - 1.5.5. Blank Spaces . . . 45
  - 1.5.6. General Format of the Command Lines . . . 45
  - 1.5.7. Delimiters . . . 46
  - 1.5.8. Order of Commands . . . 46
  - 1.5.9. Abbreviated END Specifications . . . 46
  - 1.5.10. Indentation . . . 47
  - 1.5.11. Aprepro Text Processing . . . 47
  - 1.5.12. Including Files . . . 48
- 1.6. Exodus II Database Naming Conventions . . . 49
- 1.7. Major Scope Definitions for an Input File . . . 50
- 1.8. Input/Output Files . . . 52
- 1.9. Obtaining Support . . . 54
  - 1.9.1. Helpful Information to Include when Requesting Support . . . 54

References . . . 55

2. General Commands **57**
- 2.1. SIERRA Scope . . . 57
  - 2.1.1. SIERRA Command Block . . . 57
  - 2.1.2. Title . . . 57
  - 2.1.3. Restart Control . . . 58
  - 2.1.4. User Subroutine Identification . . . 59

2.1.5. XY Functions . . . 59
2.1.6. General Functions . . . 69
2.1.7. Axes, Directions, and Points . . . 71
2.1.8. Orientation . . . 73
2.1.9. Coordinate System Block Command . . . 77
2.1.10. Define Coordinate System Line Command . . . 82
2.2. Procedure and Region . . . 85
2.2.1. Procedure . . . 86
2.2.2. Time Control . . . 86
2.2.3. Region . . . 87
2.3. Use Finite Element Model . . . 89
2.4. Mesh Quality . . . 90
2.5. Element Distortion Metrics . . . 91
2.6. Activation/Deactivation of Functionality . . . 94
2.7. Error Recovery . . . 95
2.8. Manual Job Control . . . 96
2.9. Solution Termination . . . 98
2.9.1. Solution Termination Examples . . . 100

3. Explicit Dynamic Time Step Control 101
3.1. Procedure Time Control . . . 102
3.1.1. Command Blocks for Time Control and Time Stepping . . . 104
3.1.2. Initial Time Step . . . 106
3.1.3. Time Step Scale Factor . . . 106
3.1.4. Time Step Increase Factor . . . 106
3.1.5. Step Interval . . . 107
3.1.6. Print Banner Interval for Solution Control . . . 108
3.1.7. Time Control Example . . . 108
3.2. Other Critical Time Step Methods . . . 110
3.2.1. Lanczos Method . . . 111
3.2.2. Power Method . . . 120
3.2.3. Node-Based Method . . . 127
3.3. Mass Scaling . . . 129
3.3.1. What is Mass Scaling? . . . 129
3.3.2. Mass Scaling Command Block . . . 129
3.3.3. Node Set Commands . . . 130
References . . . 133

4. Implicit Solver and Time Step Control 134
4.1. Multilevel Solver . . . 135
4.1.1. Common Multilevel Solver Commands for All Controls . . . 138
4.2. Conjugate Gradient Solver . . . 141
4.2.1. Convergence Commands . . . 143
4.2.2. Preconditioner Commands . . . 147
4.2.3. Line Search Command . . . 150

4.2.4. CG Algorithm Commands . . . 151
4.2.5. Diagnostic Output Commands . . . 153
4.2.6. CG Input Example . . . 154
4.3. Full Tangent Preconditioner . . . 155
4.3.1. Solver Selection Commands . . . 157
4.3.2. Matrix Formation Commands . . . 157
4.3.3. Reset and Iteration Commands . . . 159
4.3.4. Smoothing Iterations . . . 161
4.3.5. Fall-Back Strategy Commands . . . 163
4.3.6. CG with Full Tangent Preconditioner Input Example . . . 163
4.4. FETI Equation Solver . . . 165
4.4.1. Convergence Commands . . . 166
4.4.2. Memory Usage Commands . . . 167
4.4.3. Solver Commands . . . 168
4.4.4. CG with Full Tangent Preconditioner and FETI Input Example . . . 169
4.4.5. Printing Stiffness Matrix . . . 169
4.5. Control Contact . . . 171
4.5.1. Control Contact Commands . . . 172
4.5.2. Augmented Lagrange Control Contact . . . 177
4.5.3. Kinematic Control Contact . . . 179
4.5.4. CG with Control Contact Input Example . . . 181
4.6. Control Stiffness . . . 182
4.6.1. Convergence Commands . . . 183
4.7. Control Failure . . . 186
4.7.1. Convergence Command . . . 187
4.7.2. Level Command . . . 187
4.7.3. Other Commands . . . 187
4.7.4. Control Failure Input Example . . . 187
4.8. Control Damped Solve . . . 189
4.8.1. Convergence Command . . . 190
4.8.2. Level Command . . . 191
4.8.3. Control Damped Solve Artificial Spring Commands . . . 191
4.8.4. Other Commands . . . 192
4.8.5. Control Damped Solve Input Example . . . 192
4.9. Predictors . . . 193
4.9.1. Loadstep Predictor . . . 193
4.9.2. Level Predictor . . . 195
4.9.3. Loadstep Predictor Input Example . . . 195
4.10. Implicit Time Step Control . . . 196
4.10.1. Command Blocks for Time Control and Time Stepping . . . 196
4.10.2. Adaptive Time Stepping . . . 199
4.10.3. Time Control Example . . . 203
4.11. Implicit Dynamic Time Integration . . . 205
4.11.1. Implicit Dynamics . . . 205
4.12. Deprecated Commands . . . 207

References 208

5. Material Models **209**
5.1. General Material Commands 212
5.1.1. Density Command 212
5.1.2. Biot's Coefficient Command 212
5.1.3. General Artificial Strains 213
5.1.4. Thermal Strain Behavior 214
5.1.5. Elastic Constants 217
5.1.6. The Isotropic/Kinematic hardening BETA parameter 219
5.1.7. Coordinate Systems 220
5.2. Material Models 223
5.2.1. Elastic Model 223
5.2.2. Thermoelastic Model 225
5.2.3. Neo-Hookean Model 226
5.2.4. Elastic Fracture Model 228
5.2.5. Elastic-Plastic Model 230
5.2.6. Elastic-Plastic Power-Law Hardening Model 234
5.2.7. Ductile Fracture Model 237
5.2.8. Multilinear EP Hardening Model 240
5.2.9. Multilinear EP Hardening Model with Failure 244
5.2.10. Elastic-Plastic Hardening Model with Failure 249
5.2.11. Thermoelastic-Plastic Model 250
5.2.12. Thermoelastic-Plastic Model with Failure 252
5.2.13. Hosford Plasticity Model 254
5.2.14. Hill Plasticity Model 263
5.2.15. Barlat Plasticity Model 274
5.2.16. Johnson–Cook Model 286
5.2.17. BCJ Model 289
5.2.18. BCJ_MEM Model 292
5.2.19. Power Law Creep Model 301
5.2.20. Viscoplastic Model 303
5.2.21. Munson-Dawson Viscoplastic Model 308
5.2.22. Soil and Crushable Foam Model 312
5.2.23. Foam Plasticity Model 315
5.2.24. Low Density Foam Model 319
5.2.25. Hyperfoam Model 321
5.2.26. Elastic Three-Dimensional Orthotropic Model 323
5.2.27. Wire Mesh Model 327
5.2.28. Orthotropic Crush Model 329
5.2.29. Orthotropic Rate Model 333
5.2.30. Plane Stress Rate Plasticity 336
5.2.31. Incompressible Solid Model 343
5.2.32. Universal Polymer Model 346
5.2.33. Mooney-Rivlin Model 352

5.2.34. Stiff Elastic . . . 355
5.2.35. Swanson Model . . . 357
5.2.36. Viscoelastic Swanson Model . . . 360
5.2.37. Additional Material Models in Development . . . 365
5.3. Cohesive Zone Material Models . . . 366
5.3.1. Traction Decay . . . 366
5.3.2. Tvergaard Hutchinson . . . 367
5.3.3. Thouless Parmigiani . . . 369
5.3.4. Functional Interface Traction . . . 371
5.3.5. Alves-Roehl . . . 372
5.4. VUMAT . . . 373
5.4.1. Other Guidelines for VUMAT Use . . . 374
References . . . 378

6. Elements **382**
6.1. Finite Element Model . . . 383
6.1.1. Identification of Mesh File . . . 386
6.1.2. Alias . . . 387
6.1.3. Omit Block . . . 387
6.1.4. Time Scale Factor . . . 388
6.1.5. Component Separator Character . . . 388
6.1.6. DECOMPOSITION METHOD . . . 389
6.1.7. Element Block Parameters . . . 389
6.1.8. Element Block Default Parameters . . . 398
6.2. Element Sections . . . 401
6.2.1. Solid Section . . . 404
6.2.2. Total Lagrange Section: Composite Tet . . . 408
6.2.3. Cohesive Section . . . 409
6.2.4. Localization Section . . . 410
6.2.5. Shell Section . . . 414
6.2.6. Layered Shell Section . . . 423
6.2.7. Membrane Section . . . 425
6.2.8. Beam Section . . . 428
6.2.9. Truss Section . . . 438
6.2.10. Spring Section . . . 440
6.2.11. Damper Section . . . 441
6.2.12. Point Mass Section . . . 442
6.2.13. Particle Section . . . 444
6.2.14. Superelement Section . . . 449
6.2.15. User Mount Section . . . 453
6.2.16. Cylindrical Joint . . . 465
6.3. Element-like Functionality . . . 469
6.3.1. Rigid Body . . . 469
6.3.2. Output . . . 474

6.4. Mass Property Calculations . . . 475
6.4.1. Block Set Commands . . . 475
6.4.2. Structure Command . . . 476
6.5. Element Death . . . 477
6.5.1. Block Set Commands . . . 479
6.5.2. Criterion Commands . . . 480
6.5.3. Evaluation Commands . . . 485
6.5.4. Miscellaneous Option Commands . . . 485
6.5.5. Cohesive Zone Setup Commands . . . 492
6.5.6. Example . . . 492
6.5.7. Element Death Visualization . . . 492
6.6. Particle Embedding . . . 493
6.7. Explicitly Computing Derived Quantities . . . 495
6.8. Mesh Rebalancing . . . 495
6.8.1. Rebalance . . . 495
6.8.2. Zoltan Parameters . . . 498
References . . . 500

7. Boundary Conditions and Initial Conditions 502
7.1. General Boundary Condition Concepts . . . 503
7.1.1. Mesh-Entity Assignment Commands . . . 503
7.1.2. Methods for Specifying Boundary Conditions . . . 505
7.2. Initial Mesh Modification . . . 507
7.3. Initial Variable Assignment . . . 509
7.3.1. Mesh-Entity Set Commands . . . 510
7.3.2. Variable Identification Commands . . . 511
7.3.3. Specification Command . . . 511
7.3.4. Weibull Probability Distribution Commands . . . 512
7.3.5. External Mesh Database Commands . . . 513
7.3.6. User Subroutine Commands . . . 516
7.3.7. Distance Function Commands . . . 517
7.3.8. Additional Command . . . 517
7.3.9. Specialized Initialization Methods . . . 517
7.3.10. Examples . . . 518
7.4. Kinematic Boundary Conditions . . . 519
7.4.1. Fixed Displacement . . . 519
7.4.2. Prescribed Displacement . . . 521
7.4.3. Prescribed Velocity . . . 526
7.4.4. Prescribed Acceleration . . . 530
7.4.5. Periodic Boundary Condition . . . 534
7.4.6. Fixed Rotation . . . 536
7.4.7. Prescribed Rotation . . . 538
7.4.8. Prescribed Rotational Velocity . . . 543
7.4.9. Reference Axis Rotation . . . 548
7.4.10. Subroutine Usage for Kinematic Boundary Conditions . . . 551

7.5. Initial Velocity Conditions . . . 553
7.5.1. Node Set Commands . . . 554
7.5.2. Direction Specification Commands . . . 555
7.5.3. Angular Velocity Specification Commands . . . 555
7.5.4. User Subroutine Commands . . . 556
7.6. Force Boundary Conditions . . . 557
7.6.1. Pressure . . . 557
7.6.2. Traction . . . 566
7.6.3. Prescribed Force . . . 572
7.6.4. Prescribed Moment . . . 577
7.7. Distributed Force and Moment . . . 582
7.8. Gravity . . . 586
7.9. Centripetal Force . . . 588
7.10. Initial Temperature . . . 589
7.10.1. Block Set Commands . . . 590
7.10.2. Specification Commands . . . 590
7.11. Prescribed Temperature . . . 591
7.11.1. Block Set Commands . . . 592
7.11.2. Specification Command . . . 592
7.11.3. User Subroutine Commands . . . 593
7.11.4. External Mesh Database Commands . . . 593
7.11.5. Coupled Analysis Commands . . . 594
7.11.6. Additional Commands . . . 595
7.12. Pore Pressure . . . 596
7.12.1. Block Set Commands . . . 597
7.12.2. Specification Command . . . 597
7.12.3. User Subroutine Commands . . . 597
7.12.4. External Mesh Database Commands . . . 598
7.12.5. Coupled Analysis Commands . . . 599
7.12.6. Additional Commands . . . 599
7.13. Fluid Pressure . . . 600
7.13.1. Surface Set Commands . . . 601
7.13.2. Specification Commands . . . 601
7.13.3. Additional Commands . . . 602
7.14. Inertia Relief . . . 604
7.14.1. Node Set Commands . . . 604
7.14.2. Component Command . . . 605
7.14.3. Additional Commands . . . 605
7.14.4. General Notes . . . 605
7.14.5. Example . . . 606
7.15. Specialized Boundary Conditions . . . 607
7.15.1. Cavity Expansion . . . 607
7.15.2. Silent Boundary . . . 609
7.15.3. Spot Weld . . . 610
7.15.4. Line Weld . . . 617

7.15.5. Shell Solid Joint . . . 619
7.15.6. Viscous Damping . . . 622
7.15.7. Artificial Strain . . . 625
7.15.8. Tied Multi-Point Constraints . . . 627
7.15.9. Equation Multi-Point Constraints . . . 630
7.15.10. Submodel . . . 632
7.15.11. RBE3 . . . 633
7.15.12. Bolt Preloading . . . 635
References . . . 642

8. Contact **643**
8.1. Overview of Contact Constraints . . . 643
8.2. Overview of Contact Enforcement . . . 645
8.3. Contact Definition Block . . . 646
8.4. Defining the Contact Surfaces . . . 649
8.4.1. Contact Surface Command Line . . . 649
8.4.2. Contact Node Set Command Line . . . 650
8.4.3. Contact Surface Command Block . . . 650
8.4.4. Skin All Blocks Command . . . 652
8.4.5. Faces and Nodes in Multiple Contact Surfaces . . . 653
8.4.6. Element Death and Surface Updates . . . 654
8.4.7. Block Skinning and Surface Definition Examples . . . 655
8.5. Friction Models . . . 661
8.5.1. Frictionless Model . . . 661
8.5.2. Constant Friction Model . . . 661
8.5.3. Velocity Dependent Coulomb Friction Model . . . 662
8.5.4. Variable Coefficient Coulomb Model . . . 662
8.5.5. Tied Model . . . 663
8.5.6. Inactive Friction Model . . . 663
8.5.7. Dynamic Tied Model . . . 664
8.5.8. Glued Model . . . 664
8.5.9. Glued Normal Model . . . 665
8.5.10. Cohesive Zone Model . . . 665
8.5.11. Hybrid Model . . . 667
8.5.12. Time Variant Model . . . 669
8.5.13. Nodal Field Dependent Model . . . 670
8.6. Definition of Interactions . . . 673
8.6.1. Interaction Defaults Block . . . 673
8.6.2. Specific Interactions . . . 676
8.6.3. Interaction Behavior for Particle Element Blocks . . . 683
8.6.4. Face/Face vs. Node/Face constraints . . . 684
8.7. Active Periods . . . 687
8.8. Contact with Analytic Objects . . . 688
8.8.1. Kinematic BCs with Analytic Objects . . . 688

8.9. Contact Algorithm Options . . . 689
8.9.1. Contact Library . . . 689
8.9.2. Dash Specific Options . . . 690
8.9.3. Enforcement . . . 693
8.9.4. Remove Initial Overlap . . . 693
8.9.5. Lofted Surface Options . . . 697
8.9.6. Search Options . . . 702
8.9.7. Enforcement Options . . . 704
8.10. Contact-Specific Outputs . . . 707
8.10.1. Contact Debugging . . . 707
8.11. Log File Contact Information . . . 709
8.12. Examples . . . 711
8.13. Common Contact Algorithmic Issues . . . 715
8.14. ACME Specific Commands . . . 716
8.14.1. ACME Specific Friction Models . . . 716
8.14.2. Kinematic Partition . . . 723
8.14.3. ACME automated search tolerances . . . 726
8.14.4. Frictional behavior without Friction Model . . . 726
8.14.5. Shell Overlap Removal Iterations . . . 727
8.14.6. Surface Normal Smoothing . . . 727
8.14.7. Angle for Multiple Interactions . . . 728
8.14.8. ACME Specific Lofted Geometry Options . . . 730
8.14.9. ACME Specific Debugging Options . . . 731
8.15. Analytic Rigid Surfaces . . . 732
8.16. Deprecated Options . . . 733
References . . . 734

9. Output **735**
9.1. Syntax for Requesting Variables . . . 736
9.1.1. Example 1 . . . 736
9.1.2. Example 2 . . . 737
9.1.3. Other command blocks . . . 738
9.1.4. Rigid Body Variables . . . 738
9.2. General Output Database Commands . . . 740
9.3. Results Output . . . 742
9.3.1. Exodus Results Output File . . . 742
9.3.2. User-Defined Output . . . 757
9.4. History Output . . . 780
9.4.1. Output Variables . . . 781
9.4.2. Outputting History Data on a Node Set . . . 784
9.4.3. Synchronize Output . . . 785
9.4.4. Use Output Scheduler . . . 785
9.4.5. Write History If System Error Encountered . . . 785
9.5. Heartbeat Output . . . 786
9.5.1. Output Variables . . . 788

9.5.2. Outputting Heartbeat Data on a Node Set . . . 791
9.5.3. Specifying Time Steps for Output . . . 791
9.5.4. Synchronize Output . . . 791
9.5.5. Use Output Scheduler . . . 792
9.5.6. Write Heartbeat On Signal . . . 792
9.5.7. Heartbeat Output Formatting Commands . . . 792
9.5.8. Monitor Output Events . . . 795
9.6. Restart Data . . . 797
9.6.1. Restart Options . . . 799
9.6.2. Component Separator Character . . . 805
9.6.3. Specifying Time Steps for Output . . . 806
9.6.4. Reducing Restart File Size . . . 806
9.6.5. Synchronize Output . . . 807
9.6.6. Use Output Scheduler . . . 807
9.6.7. Write Restart If System Error Encountered . . . 807
9.6.8. Restarting an Implicit/Explicit Analysis . . . 808
9.7. Output Scheduler . . . 810
9.7.1. Output Scheduler Command Block . . . 810
9.7.2. Example of Using the Output Scheduler . . . 811
9.8. Variable Interpolation . . . 812
9.8.1. Variable Interpolation Examples . . . 814
9.9. Global Output Options . . . 817
9.10. Variables . . . 818
9.10.1. Global, Nodal, Face, and Element Variables . . . 818
9.10.2. Variables for Material Models . . . 840
9.10.3. Energy Output Variables . . . 840
9.11. Solution Verification . . . 842
9.12. Identify Fragments . . . 844
9.13. Detect Breach . . . 846
References . . . 848

10. User Subroutines **849**
10.1. Programming . . . 852
10.1.1. Subroutine Interface . . . 853
10.1.2. Query Functions . . . 854
10.1.3. Miscellaneous Query Functions . . . 877
10.2. Command File . . . 880
10.2.1. Subroutine Identification . . . 880
10.2.2. User Subroutine Command Lines . . . 880
10.2.3. Time Step Initialization . . . 882
10.2.4. User Variables . . . 885
10.3. Compilation and Execution . . . 888
10.4. Examples . . . 889
10.4.1. Pressure as a Function of Space and Time . . . 889
10.4.2. Error Between a Computed and an Analytic Solution . . . 892

10.4.3. Transform Output Stresses to a Cylindrical Coordinate System 896
10.5. User Subroutine Library 902
10.5.1. aupst_preload_solver 902
10.5.2. aupst_torque_calc 907
10.5.3. aupst_trans_rot_velocity 908
10.5.4. aupst_cyl_transform 909
10.5.5. aupst_rec_transform 911
10.5.6. copy_data 912
10.5.7. trace 913
10.5.8. aupst_cavity_volume 914

11. Transfers **916**
11.1. SIERRA Transfers 916
11.2. Inter-procedural Transfers 917
11.2.1. Copying Data with Inter-procedural Transfers 919
11.2.2. Selecting Variables for Transfer 919
11.2.3. Additional Transfer Options 919
11.2.4. Interpolating Data with Interpolation Transfers 920
11.3. Automated Implicit/Explicit Hand-off 924
11.3.1. Known Issues and Limitations 925

A. Explicit Dynamic Example Problem **926**

Appendices **926**

B. Implicit Quasistatic Example Problem **935**

C. Consistent Units **943**

D. Constraint Enforcement Hierarchy **945**

E. Improving Explicit Dynamics Performance **947**
References 949

F. Troubleshooting Guide for Implicit Convergence **950**

G. User-Defined Model Initialization **952**

H. Other Scripts Maintained by Development Team **954**

I. Pre-defined User Options **955**

J. Known Issues **956**
J.1. Epic Material Model Issues 957
J.2. SPH Particle Method Time Step Stability Issues 958
J.3. Particle Creation Issues 959
J.4. Bad Element Shapes Yielding Invalid Contact Geometry 960

Index 961

LIST OF FIGURES

Figure 1-1. Input/output files 52

Figure 2-1. Piecewise linear and piecewise constant functions 62
Figure 2-2. Example Displacement Map 70
Figure 2-3. Adjacent shell elements with non-aligned local coordinate systems 74
Figure 2-4. Rectangular coordinate system 74
Figure 2-5. Z-Rectangular coordinate system. 75
Figure 2-6. Cylindrical coordinate system. 75
Figure 2-7. Spherical coordinate system. 76
Figure 2-8. Rotation about 1 76
Figure 2-9. Coordinate System Definition Vectors 79
Figure 2-10. Coordinate System Examples 80
Figure 2-11. Conical Coordinate System Definition at $\tilde{X}\tilde{Z}$ plane 81
Figure 2-12. Ellipsoidal Coordinate System
Using axis_stretching=(2.0, 1.1, 1.0) 81
Figure 2-13. Coordinate system behavior near the $\tilde{Z}$ axis. $\hat{R}$ is red, $\hat{S}$ is green, and $\hat{T}$ is blue. 83
Figure 2-14. Examples of elements with varying nodal Jacobians 91

Figure 4-1. Reaching convergence with controls and the multilevel solver. 136
Figure 4-2. Contact configuration at the beginning of the contact update. 180
Figure 4-3. Contact gap removal (after contact search). 180
Figure 4-4. Contact slip calculations. 181

Figure 5-1. Isotropic Hardening, Beta = 1.0 220
Figure 5-2. Kinematic Hardening, Beta = 0.0 221
Figure 5-3. Mixed Hardening, Beta = 0.5 221
Figure 5-4. Elastic-Plastic Theory: Example yield surface, f 231
Figure 5-5. EP Power Law Hardening Theory: Example Response 235
Figure 5-6. Multilinear EP Theory: Example Response 242
Figure 5-7. Hosford Plasticity Theory: Hosford yield surface with $a = 4,\ 8,\ 100$ 261
Figure 5-8. Hill Plasticity Theory: Hill yield surface 271
Figure 5-9. Barlat Plasticity Theory: Barlat yield surface 283
Figure 5-10. Orthotropic Crush Theory: Example Crush Curve 331

Figure 6-1. Association between command lines and command block. 393
Figure 6-2. The reference B_0, intermediate B_I, and current configuration B of the body. 411
Figure 6-3. Indexing convention for through-thickness integration points 418
Figure 6-4. Location of geometric plane of shell for various lofting factors. 419
Figure 6-5. Local rst coordinate system for a shell element. 420

Figure 6-6. Rotation of 30 degrees about the 1-axis (X'-axis). 421
Figure 6-7. Definition of shell R,S,T coordinate system 423
Figure 6-8. Layered Shell Example 425
Figure 6-9. Beam ROD Cross Section 430
Figure 6-10. Beam TUBE Cross Section 430
Figure 6-11. Beam BAR Cross Section 431
Figure 6-12. Beam BOX Cross Section 431
Figure 6-13. Beam I Cross Section 432
Figure 6-14. Beam L Cross Section 432
Figure 6-15. Beam C1 Cross Section 433
Figure 6-16. Beam C2 Cross Section 433
Figure 6-17. Beam T Cross Section 434
Figure 6-18. Beam T1 Cross Section 434
Figure 6-19. Beam Z Cross Section 435
Figure 6-20. Beam HAT Cross Section 435
Figure 6-21. Beam I2 Cross Section 436
Figure 6-22. Locations of REFERENCE AXIS command options 436
Figure 6-23. User Mount Coordinate System 455
Figure 6-24. Cylindrical Joint Axial Rotation 466
Figure 6-25. Cylindrical Joint Axial Translation 467
Figure 6-26. Cylindrical Joint Radial Rotation 467
Figure 6-27. Cylindrical Joint Radial Translation 467
Figure 6-28. Examples of how an element is converted and when it is not converted. 491

Figure 7-1. Result of a unit pressure load 565
Figure 7-2. Result of a unit traction load 571
Figure 7-3. Result of a unit prescribed force 577
Figure 7-4. Distributed Z Force Load Distribution 584
Figure 7-5. Distributed Z Moment Load Distribution 584
Figure 7-6. Distributed Load Example 585
Figure 7-7. Fluid Pressure Setup 601
Figure 7-8. Spot weld force–displacement curves. 611
Figure 7-9. Sign convention for spot weld normal displacements. 611
Figure 7-10. Spot weld normal and tangent. 612
Figure 7-11. Sign convention for spot weld normal displacements with `IGNORE INITIAL OFFSET = YES`. 614
Figure 7-12. Search tolerances for shell-solid joints. (a) input; (b) automatic 621
Figure 7-13. RBE3 test positions. 635
Figure 7-14. Anatomy of a typical preloaded bolted joint. Users are recommended to place the portion of the bolt shank to be preloaded in a separate element block. 637
Figure 7-15. Anatomy of a typical preloaded bolted joint. 640

Figure 8-1. Two blocks in contact. 643
Figure 8-2. Faces in Multiple Contact Surfaces: Example 1 654
Figure 8-3. Faces in Multiple Contact Surfaces: Example 2 655

Figure 8-4. Simple block skinning example 656
Figure 8-5. Block skinning with multiple blocks example 656
Figure 8-6. Block skinning advanced example 657
Figure 8-7. Contact surface definition examples 658
Figure 8-8. Hexahedral and shell contact surface definitions 659
Figure 8-9. Bricks sharing a cohesive zone and being pulled apart. 666
Figure 8-10. Displacement X vs. time for node 8 667
Figure 8-11. Cohesive force in x vs. time for node 8 668
Figure 8-12. Example usage of the hybrid friction model. 669
Figure 8-13. Node/Face Constraint example 685
Figure 8-14. Face/Face Constraint example 686
Figure 8-15. Node Normal Computation Options 692
Figure 8-16. Effect of overlap removal on model geometry. 694
Figure 8-17. Lofted beam geometry 699
Figure 8-18. Lofted particle geometry 700
Figure 8-19. Lofted shell geometry 701
Figure 8-20. Example lofted geometries produced by shell lofting 701
Figure 8-21. Illustration of normal and tangential tolerances 704
Figure 8-22. Problem with two blocks coming into contact 711
Figure 8-23. Problem with three blocks coming into contact. 713
Figure 8-24. State of bodies after contact and deformation. 714
Figure 8-25. Deformed contact geometry. 714
Figure 8-26. Illustration of kinematic partition values 725
Figure 8-27. Illustrations of multiple interactions at a node. 729

Figure 9-1. Pin-In-Hole Results for Max Normal Gap 769
Figure 9-2. Pin-In-Hole Results for Max Opposing Gap 769
Figure 9-3. Large Motion Results for Max Normal Gap 770
Figure 9-4. Large Motion Results for Max Opposing Gap 770
Figure 9-5. Default Cylindrical Transformation 775
Figure 9-6. Cylindrical System Transformation of Coordinates 775
Figure 9-7. Variable interpolation options. 816
Figure 9-8. Breach detection between surfaces 1 and 2. Dead elements in red. 846

Figure 10-1. Overview of components required to implement functionality. 852
Figure 10-2. Example of Preload Iteration 905
Figure 10-3. Anatomy of the Preload Iteration Cycle 906

Figure A-1. Mesh for example problem. 926
Figure A-2. Mesh with blue and green surfaces removed. 926

Figure B-1. Eraser schematic 935
Figure B-2. Complete eraser mesh 936

LIST OF TABLES

Table 4-1. Log File Convergence Markings 147

Table 5-1. State Variables for ELASTIC FRACTURE Model 229
Table 5-2. State Variables for ELASTIC PLASTIC Model 233
Table 5-3. State Variables for ELASTIC PLASTIC Model for Shells 233
Table 5-4. State Variables for EP POWER HARD Model 236
Table 5-5. State Variables for EP POWER HARD Model for Shells 236
Table 5-6. State Variables for DUCTILE FRACTURE Model 239
Table 5-7. State Variables for MULTILINEAR EP Model 243
Table 5-8. State Variables for MULTILINEAR EP Model for Shells 243
Table 5-9. State Variables for ML EP FAIL Model 247
Table 5-10. State Variables for ML EP FAIL Model for Shells 248
Table 5-11. State Variables for HOSFORD PLASTICITY Model (Section 5.2.13) 262
Table 5-12. State Variables for HILL PLASTICITY Model (Section 5.2.14) 273
Table 5-13. State Variables for BARLAT PLASTICITY Model (Section 5.2.15) 285
Table 5-14. State Variables for JOHNSON COOK Model 288
Table 5-15. State Variables for BCJ Model 291
Table 5-16. State Variables for BCJ_MEM Model 298
Table 5-17. Additional State Variables for BCJ_MEM Model When Modeling Recrystallization 299
Table 5-18. State Variables for POWER LAW CREEP Model 302
Table 5-19. State Variables for VISCOPLASTIC Model (Section 5.2.20) 307
Table 5-20. State Variables for MD_VISCOPLASTIC Model (Section 5.2.21) 311
Table 5-21. State Variables for SOIL FOAM Model 314
Table 5-22. State Variables for FOAM PLASTICITY Model 318
Table 5-23. State Variables for LOW DENSITY FOAM Model 320
Table 5-24. State Variables for WIRE MESH Model 328
Table 5-25. State Variables for ORTHOTROPIC CRUSH Model 332
Table 5-26. State Variables for ORTHOTROPIC RATE Model 335
Table 5-27. State Variables for PLANE STRESS RATE PLASTICITY Model (Section 5.2.30) 342
Table 5-28. State Variables for Universal Polymer Model 351
Table 5-29. State Variables for MOONEY RIVLIN Model 354
Table 5-30. State Variables for SWANSON Model 359
Table 5-31. State Variables for VISCOELASTIC SWANSON Model 364
Table 5-32. State Variables for TRACTION DECAY Surface Model 366
Table 5-33. State Variables for TVERGAARD HUTCHINSON Surface Model 368
Table 5-34. State Variables for THOULESS PARMIGIANI Surface Model 370

Table 5-35. VUMAT Input Arguments . . . 375
Table 5-36. VUMAT Output Arguments . . . 376
Table 5-37. Built-In State Variables for ABAQUS_VUMAT Model . . . 377

Table 6-1. Default moduli ratios for the various effective moduli models. . . . 396
Table 6-5. Variables For Cylindrical Joint . . . 468

Table 8-1. Output Variables Generated For Each Interaction Specific Output . . . 682

Table 9-1. Global Energy Variables . . . 819
Table 9-2. Other Global Variables For All Analyses . . . 821
Table 9-3. Global Variables for AL Control Contact . . . 822
Table 9-4. Global Variables for Rigid Bodies . . . 822
Table 9-5. Nodal Variables for All Analyses . . . 823
Table 9-6. Nodal Variables for Implicit Analyses . . . 823
Table 9-7. Nodal Variables for Shells and Beams . . . 824
Table 9-8. Nodal Variables for Spot Welds . . . 825
Table 9-9. Nodal Variables for Line Welds . . . 825
Table 9-10. Nodal Variables for MPCs . . . 826
Table 9-11. Nodal Variables for Contact . . . 827
Table 9-12. Element Variables for All Elements . . . 830
Table 9-13. Element Variables for Solid Elements . . . 832
Table 9-14. Element Variables for Membranes . . . 835
Table 9-15. Element Variables for Shells . . . 835
Table 9-16. Element Variables for Trusses . . . 836
Table 9-17. Element Variables for Cohesive Elements . . . 836
Table 9-18. Element Variables for Nonlinear Beams . . . 837
Table 9-19. Element Variables for Springs . . . 837
Table 9-20. Element Variables for Line Welds . . . 838
Table 9-21. Face Variables . . . 839

Table 10-1. Subroutine Input Parameters . . . 854
Table 10-2. Subroutine Output Parameters . . . 855
Table 10-3. aupst_get_real_param Arguments . . . 857
Table 10-4. aupst_get_integer_param Arguments . . . 858
Table 10-5. aupst_get_string_param Arguments . . . 859
Table 10-6. aupst_evaluate_function Arguments . . . 860
Table 10-7. aupst_get_time Argument . . . 860
Table 10-8. aupst_check_node_var Arguments . . . 862
Table 10-9. aupst_check_elem_var Arguments . . . 863
Table 10-10. aupst_get_node_var Arguments . . . 864
Table 10-11. aupst_get_elem_var Arguments . . . 865
Table 10-12. aupst_get_elem_var_offset Arguments . . . 866
Table 10-13. aupst_put_node_var Arguments . . . 867
Table 10-14. aupst_put_elem_var Arguments . . . 868
Table 10-15. aupst_put_elem_var_offset Arguments . . . 869

Table 10-16. aupst_check_global_var Arguments 871
Table 10-17. aupst_get_global_var Arguments 871
Table 10-18. aupst_put_global_var Arguments 872
Table 10-19. Topologies Used by Sierra/SM 873
Table 10-20. aupst_get_elem_topology Arguments 874
Table 10-21. aupst_get_elem_nodes Arguments 875
Table 10-22. aupst_get_face_topology Arguments 876
Table 10-23. aupst_get_face_nodes Arguments 877
Table 10-24. aupst_get_elem_centroid Arguments 878
Table 10-25. aupst_get_point Arguments 879
Table 10-26. aupst_get_proc_num Arguments 879
Table 10-27. Required aupst_preload_solver Parameters 904
Table 10-28. aupst_preload_solver Subroutine State Variables 907
Table 10-29. Required aupst_torque_calc Parameters 908
Table 10-30. Required aupst_trans_rot_velocity Parameters 909

Table C-1. Consistent Unit Sets 943

Table D-1. Constraint Enforcement Order 945

Table G-1. Required Variables For Consistent Model Initialization 953

Table I-1. Pre-defined functions, directions, and points that can be used in a SIERRA SM analysis 955

1. INTRODUCTION

Sierra/SolidMechanics (Sierra/SM) is a three-dimensional solid mechanics code with a versatile element library, nonlinear material models, large deformation capabilities, and contact. It is built on the SIERRA Framework [1, 2, 3, 4]. SIERRA provides a data management framework in a parallel computing environment that allows the addition of capabilities in a modular fashion. Contact capabilities are parallel and scalable.

This document provides information about the functionality in Sierra/SM and the command structure required to access this functionality in a user input file. This document is divided into chapters based primarily on functionality. For example, the command structure related to the use of various element types is grouped in one chapter; descriptions of material models are grouped in another chapter.

Sierra/SM provides both explicit transient dynamics and implicit quasistatics and dynamics capabilities. Both the explicit and implicit modules are highly scalable in a parallel computing environment. In the past, the explicit and implicit capabilities were provided by two separate codes, known as Presto and Adagio, respectively. These capabilities have been consolidated into a single code. The executable is named Adagio, but it provides the full suite of solid mechanics capabilities, for both implicit and explicit. The Presto executable has been disabled as a consequence of this consolidation.

Important references for both the implicit and explicit capabilities are given in the references section at the end of this chapter. Adagio was preceded by the codes JAC and JAS3D; JAC is described in [5]; JAS3D is described in [6]. Presto was preceded by the code Pronto3D. Pronto3D is described in [7] and [8]. Some of the fundamental nonlinear technologies used by Sierra/SM are described in [9, 10, 11]. Sierra/SM uses the Exodus II database. (Other options may be added in the future.) The Exodus II database format is described in [12]. The reference document for ACME [13] provides important information about contact. ACME is a third-party library for contact search and enforcement.

One of the key concepts for the command structure in the input file is a concept called *scope*. A detailed explanation of scope is provided in Section 1.3. Most of the command lines in Chapter 2 are related to a certain scope rather than to some particular functionality.

1.1. Document Overview

This document describes how to create an input file for Sierra/SM. Highlights of the document contents are as follows:

- Chapter 1 presents the overall structure of the input file, including conventions for the command descriptions, style guidelines for file preparation, and naming conventions for input files that reference the Exodus II database [12]. The chapter also gives an example of the general structure of an input file that employs the concept of scope.
- Chapter 2 explains some of the commands that are general to various applications based on the SIERRA Framework. These commands define scopes, functions, and coordinate systems, and they let you set up some of the main time control parameters (begin time, end time, time blocks) for your analysis. (Time control and time step control are discussed in more detail in Chapters 3 and 4.) Other capabilities documented in this chapter are available for calculating element distortion and for activating and deactivating functionality at different times throughout an analysis.
- Chapter 3 describes how to set the start time, end time, and time blocks for an explicit dynamic analysis. This chapter also discusses various options for controlling the critical time step for transient dynamics.
- Chapter 4 discusses the multilevel, nonlinear iterative solver used for implicit calculations. This chapter also describes how to set start time, end time, and time blocks for an implicit analysis.
- Chapter 5 describes material models that can be used in conjunction with the elements. Most of the material models can be used for both explicit and implicit analyses. Even though a material model can be used by both cases, it may be that the use of the material model is better suited for one type of analysis. For example, a material model set up to characterize behavior over a long time would be better suited for use in implicit analyses than in explicit analyses. In such cases, this will be noted. Chapter 5 also discusses the application of temperature to a mesh and the computation of thermal strains (isotropic and anisotropic).
- Chapter 6 lists the available elements and describes the commands used to access the various options for the elements. Most elements can be used for both explicit and implicit analyses. If that is not the case, a note is made to that effect.
  Chapter 6 also includes descriptions of the commands for mass property calculations, element death, and mesh rebalancing. “Element-like” capabilities and rigid bodies are also discussed in Chapter 6.
- Chapter 7 documents how to use kinematic boundary conditions, force boundary conditions, initial conditions, and specialized boundary conditions.
- Chapter 8 discusses how to define interactions of contact surfaces.
- Chapter 9 details the various options for obtaining output.
- Chapter 10 provides an overview of the user subroutine functionality.
- Chapter 11 documents how to perform transfers between procedures.
- Appendix A provides a sample input file for an explicit dynamic analysis of 16 lead spheres being crushed together inside a steel box. Appendix B provides a sample input file for an implicit, quasistatic analysis of an eraser being pulled across a surface. These problems

both emphasize large deformation and contact.

- The index allows you to find information about command blocks and command lines. In general, single-level entries identify the page where the command syntax appears, with discussion following soon thereafter—on the same page or on a subsequent page. Page ranges are not provided in this index. Some entries consist of two or more levels. Such entries are typically based on context, including such information as the command blocks in which a command line appears, the location of the discussion related to a particular command line, and tips on usage. The electronic version of this document contains hyperlinked entries from the page numbers listed in the index to the text in the body of the document.

1.2. Running The Code

The command to run a basic serial Sierra/SM analysis is:

```
sierra adagio -i sierra_input.i
```

The command to run a basic parallel Sierra/SM analysis is:

```
sierra -j 4 adagio -i sierra_input.i
```

The sierra command also optionally takes many more options to specify number of processors, queues to use, output log file names, etc. See the sierra command documentation for a full description of capabilities. The command to generate the sierra command documentation is:

```
sierra --help
```

The command to generate adagio specific options is:

```
sierra adagio --help
```

Note sierra script command line options have to be placed before the adagio command.

1.3. Overall Input Structure

Sierra/SM is one of many mechanics codes built on the SIERRA Framework. The SIERRA Framework provides the capability to perform multiphysics analyses by coupling together SIERRA codes appropriate for the mechanics of interest. Input files may be set up for analyses using only Sierra/SM, or they may be set up to couple Sierra/SM and one or more other SIERRA analysis codes. For example, you might run an implicit Sierra/SM analysis to compute a stress state, and then use the results of this analysis as initial conditions for an explicit analysis. For a Sierra multiphysics analysis, the commands for all physics modules used in the analysis will all be in the same input file. Therefore, the input for Sierra/SM reflects the fact that it could be part of a multiphysics analysis. Sierra/SM coupled with the Sierra package Aria is Arpeggio.

Sierra input file syntax uses ideas from programming languages. A multiphysics analysis is defined using *scope*, i.e., the extent of the subject matter that part of the input file deals with, to group similar commands. Examples of the Sierra input files are given in this section, section 1.7, and appendices A and B.

The broadest scope in the input file is the SIERRA scope. The SIERRA scope contains information that can be shared among different physics. Examples of physics information that can be shared are definitions of functions and materials. For example, in a coupled implicit/explicit Sierra/SM analysis, both the implicit and explicit components could reference functions to define time histories for boundary conditions or stress-strain curves. Material model definitions could also be shared in a similar manner.

A scope may contain another scope. Within a SIERRA scope the overall analysis from the start time to the end time is controlled by the `procedure` scope In a multiphysics analysis, a SIERRA scope may contain multiple procedures.

The `procedure` scope may contain a `region` scope. The region scope controls a single time step. Inside the procedure scope (but outside of the region scope) are commands that set the start time and the end time for the analysis.

Inside the region scope for Sierra/SM are definitions for boundary conditions and contact. In a multiphysics analysis, there would be more than one region. In a coupled implicit/explicit solid mechanics example, there would be both a Presto region and an Adagio region, each within its respective procedures. The definitions for boundary conditions and contact and the mesh specification for explicit dynamics would appear in the Presto region; the definitions for boundary conditions and contact and the mesh specification for the implicit analysis would appear in the Adagio region.

Scope is defined by a command block. The command blocks group together command lines or other command blocks that share a similar functionality. A command block will begin with an input line that has the word `BEGIN` and end with an input line that has the word `END`. The SIERRA scope, for example, is defined by a command block that begins with an input line of the following form:

```
BEGIN SIERRA my_problem
```

The two character strings `BEGIN` and `SIERRA` are the key words for this command block. An

input line defining a command block or a command line will have one or more key words. The string `my_problem` is a user-specified name for this SIERRA scope. The SIERRA scope is terminated by an input line of the following form:

```
END SIERRA my_problem
```

In the above input line, `END` and `SIERRA` are the key words to end this command block. SIERRA scope termination may be shortened to

```
END
```

as discussed in section 1.5.9. Similar input lines are used to define the procedure and region scopes. Boundary conditions are another example where a scope is defined. A particular instance of a boundary condition for a prescribed displacement boundary condition is defined with a command block. The command block for the boundary condition begins with an input line of the form:

```
BEGIN PRESCRIBED DISPLACEMENT
```

and ends with an input line of either of these forms:

```
END PRESCRIBED DISPLACEMENT

END
```

Command lines appear within the command blocks. The command lines typically have the form `keyword = value`, where `value` can be a real, an integer, or a string. In the previous example of the prescribed displacement boundary condition, there would be command lines inside the command block that are used to set various values. For example, the boundary condition might apply to all nodes in node set 10, in which case there would be a command line of the following forms:

```
NODE SET = nodelist_10

NODESET = nodelist_10
```

If the prescribed displacement were to be applied along a given component direction, there would be a command line of this form:

```
COMPONENT = X
```

The form above would specify that the prescribed displacement would be in the x-direction. Finally, if the displacement magnitude is described by a time history function with the name `my_function`, there would be a command line of this form:

```
FUNCTION = my_function
```

Functions are discussed in sections 2.1.5 and 2.1.6.

The command block for the boundary condition with the appropriate command lines would appear as follows:

```
BEGIN PRESCRIBED DISPLACEMENT
  NODE SET = nodelist_10
  COMPONENT = X
  FUNCTION = my_function
END PRESCRIBED DISPLACEMENT
```

It is possible to have a command line with the same key words appearing in different scopes. For example, we might have a command line identified by the word `TYPE` (introduced in section 2.1.5) in two or more different scopes. The command line would perform different functions based on the scope in which it appeared, and the associated value could be different in the two locations.

The input lines are read by a parser that searches for recognizable key words. If the key words in an input line are not in the list of key words used by Sierra/SM to describe command blocks and command lines, the parser will generate an error. A set of key words defining a command line or command block for Sierra/SM that is not in the correct scope will also cause a parser error. For example, the key words `STEP INTERVAL` define a valid command line in the scope of the `TIME CONTROL` command block. However, if this command line was to appear in the scope of one of the boundary conditions, it would not be in the proper scope, and the parser would generate an error. Once the parser has an input line with any recognizable key words in the proper scope, a method can be called that will handle the input line.

There is an initial parsing phase that checks only the parser syntax. If the parser encounters a command line it cannot parse within a certain scope, the parser will indicate it cannot recognize the command line and will list the various command lines that can appear within that scope. The initial parsing phase will catch errors such as the one described in the previous paragraph (a command line in the wrong scope). It will also catch misspelled key words.

The initial parsing catches some but not all types of errors. Some errors not detected in the initial parsing phase are caught afterwards and are handled one at a time. One such example is that initial parsing will not catch a command line value that is out of command line's specified range. A second example is a combination of command lines within a command block that is not allowed.

1.4. Conventions for Command Descriptions

The parts of an input file built into Sierra can be hard to distinguish from the user supplied parts. For example in section 1.3, the user supplied only the name `my_problem` for the `SIERRA` scope and the name `my_function` for a function. This section presents the syntax used in this document to describe input files.

A note about continuation characters. Many individual command lines discussed in the text appear on several text lines. In the text of this document, the continuation symbols that are used to continue lines in an actual input file (`\#` and `\$`, Section 1.5.2) are not used for those instances where the description of the command line appears on several text lines. The description of command lines will indicate all the key words, delimiters, and values that constitute a complete command line. As an example, the `DEFINE POINT` command line (Section 2.1.7) is presented in the text as follows:

```
DEFINE POINT <string>point_name WITH COORDINATES
   <real>value_1 <real>value_2 <real>value_3
```

If the `DEFINE POINT` command line were used as a command line in an input file and spread over two input lines, it would appear, with actual values, as follows:

```
DEFINE POINT center WITH COORDINATES \#
   10.0 144.0 296.0
```

In the above example, the `\#` symbol implies the first line is continued onto the second line.

1.4.1. Key Words

The key word or key words for a command are shown in uppercase letters. For actual input, you can use all uppercase letters for the key words, all lowercase letters for the key words, or some combination of uppercase and lowercase letters for the key words.

1.4.2. User-Specified Input

The input that you supply is typically shown in lowercase letters. (Occasionally, uppercase letters may be used for user input for purposes of clarity or in examples.) The user-supplied input may be a real number, an integer, a string, or a string list. For the command descriptions, a type appears before the user input. The type (real, integer, string, string list) description is enclosed by angle brackets, `<>`, and precedes the user-supplied input. For example:

```
<real>value
```

indicates that the quantity `value` is a real number. For the description of an input command, you would see:

```
FUNCTION = <string>function_name
```

Your input would be

```
FUNCTION = my_name
```

if you have specified a function name called `my_name`.

Valid user input consists of:

`<integer>`	Integer data is a single integer number.
`<real>`	Real data is a single real number. It may be formatted with the usual conventions, such as `1234.56` or `1.23456e+03`.
`<string>`	String data is a single string.
`<string list>`	A string list consists of multiple strings separated by white space, a comma, a tab, or white space combined with a comma or a tab.

1.4.3. Optional Input

Anything in an input line that is enclosed by square brackets, [ ], represents optional input within the line. This convention is not used to identify optional input lines. Any command line that is optional (in its entirety) will be described within the text.

1.4.4. Default Values

A value enclosed by parentheses, (), appearing after the user input denotes the default value. For example:

```
SCALE FACTOR = <real>scale_factor(1.0)
```

implies the default value for `scale_factor` is 1.0. Any value you specify will overwrite the default.

For your actual input file, you may omit a command line if you want to use the default value associated with the command line. For example, there is a `TIME STEP SCALE FACTOR` command line used to set one of the time control parameters; the parameter for this command line has a default value of 0.9. If you want to use the default value of 0.9 for this parameter, you do not have to include the `TIME STEP SCALE FACTOR` command line in the `TIME CONTROL` command block.

1.4.5. Multiple Options for Values

Quantities separated by the `|` symbol indicate that one and only one of the possible choices must be selected. For example:

```
EXPANSION RADIUS = <string>SPHERICAL|CYLINDRICAL
```

implies that expansion radius must be defined as `SPHERICAL` or `CYLINDRICAL`. One of the values must appear. This convention also applies to some of the command options within a begin/end block. For example:

```
SURFACE = <string>surface_name|
NODE SET|NODESET = <string>nodelist_name
```

in a command block specifies that either a surface or a node set must be specified. In the case of node sets, either the command `NODE SET` or `NODESET` may be used.

Quantities separated by the `/` symbol can appear in any combination, but any one quantity in the sequence can appear only once. For example,

```
COMPONENTS = <string>X/Y/Z
```

implies that components can equal any combination of X, Y, and Z. Any value (X or Y or Z) can appear at most once, and at least one value of X, Y, or Z must appear. Some examples of valid expressions in this case are as follows:

```
COMPONENTS = Z

COMPONENTS = Z X

COMPONENTS = Y X Z

COMPONENTS = Z Y X
```

An example of an invalid expression would be the following:

```
COMPONENTS = Y Y Z
```

1.4.6. Known Issues and Warnings

Where there are known issues with the code, these are documented in the following manner:



Known Issue: A description of the known issue with the code would be provided here.

Similarly, warnings regarding usage of code features that are not defective, but must be used with care because of their nature, are documented as follows:



Warning: A description of the warning related to the usage of a code feature would be provided here.

1.4.7. Commands Applicable to Only Explicit or Implicit Analyses

As mentioned previously, Sierra/SM has the ability to run either explicit dynamic analyses and implicit quasistatic and dynamic analyses. This guide documents all features of the code for either of these analysis types. In most cases, the features available for explicit analyses are also available for implicit analysis, and vice versa. However, there are some capabilities that are only available for one type of analysis. These are denoted in this document with the following icons in the margin.



Features and commands only available for explicit dynamics have this icon in the margin next to their documentation.



Features and commands only available for implicit dynamics and quasistatics have this icon in the margin next to their documentation.

In listings of command lines and blocks, smaller versions of these icons are used to denote commands only available for explicit or implicit analyses. These appear in the margin next to either an individual command line, or next to the opening line of a command block, as shown in these examples:




```
#Example explicit command line:
DEATH ON INVERSION = OFF|ON(OFF)
```



```
#Example implicit command line:
CAPTURE TOLERANCE = <real>cap_tol
```




```
#Example explicit command block:
BEGIN LINE WELD
  ...
END [LINE WELD]
```




```
#Example implicit command block:
BEGIN SOLVER
  ...
END [SOLVER]
```

1.5. Style Guidelines

This section gives information that will affect the overall organization and appearance of an input file. It also contains recommendations that will help construct input files that are readable and easy to proof.

1.5.1. Comments

A comment is anything between the `#` symbol or the `$` symbol and the end-of-line. If the first non-blank character in a line is a `#` or `$`, the entire line is a comment line. You can also place a `#` or `$` (preceded by a blank space) after the last character in an input line used to define a command block or command line.

1.5.2. Continuation Lines

An input line can be continued by placing a `\#` pair of characters (or `\$`) at the end of the line. The following line is then taken to be a continuation of the preceding line that was terminated by the `\#` or `\$`. Everything after the line-continuation pair of characters is discarded, including the end-of-line.

1.5.3. Capital and Lower Case Letters

Almost all the character strings in the input lines are case insensitive. For example, the `BEGIN SIERRA` key words could appear as:

```
BEGIN SIERRA

begin sierra

Begin Sierra
```

You could specify a `SIERRA` command block with:

```
BEGIN SIERRA BEAM
```

and terminate the command block with this input line:

```
END SIERRA beam
```

Case is important only for file name specifications. If you have defined a restart file with uppercase and lowercase letters and want to use this file for a restart, the file name you use to request this restart file must exactly match the original definition you chose.

1.5.4. Commas and Tabs

Commas and tabs in input lines are ignored.

1.5.5. Blank Spaces

We recommend that everything be separated by blank spaces. For example, a command line of the form

```
node set = nodelist_10
```

is recommended over the forms

```
node set= nodelist_10
```

```
node set =nodelist_10
```

Both of the above two lines are correct, but it is easier to check the first form (the equal sign surrounded by blank space) in a large input file.

The parser will accept the following line:

```
BEGIN SIERRABEAM
```

However, it is harder to check this line for the correct spelling of the key words and the intended SIERRA scope name than this line:

```
BEGIN SIERRA BEAM
```

It is possible to introduce hard-to-detect errors because of the way in which the blank spaces are handled by the command parser. Suppose you type

```
begin functions my_func
```

rather than the following correct form:

```
begin function my_func
```

For the incorrect form of this command line (in which `functions` is used rather than `function`), the parser will generate a string name of

```
s my_func
```

for the function name rather than the following expected name:

```
my_func
```

If you attempt to use a function named `my_func`, the parser will generate an error because the list of function names will include `s my_func` but not `my_func`.

1.5.6. General Format of the Command Lines

In general, command lines have the following form:

```
keyword = value
```

This pattern is not always followed, but it describes the vast majority of the command lines.

1.5.7. Delimiters

The delimiter used throughout this document is "=" (the equal sign). Typically, but not always, the = separates key words from input values in a command line. Consider this command line:

```
COMPONENTS = X
```

Here, the key word `COMPONENTS` is separated from its value, a string in this case, by the =. Some command lines do allow for other delimiters. The use of these alternate delimiters is not consistent, however, throughout the various command lines. (This lack of consistency has the potential for introducing errors in this document as well as in your input.) The = provides a strong visual cue for separating key words from values. By using the = as a delimiter, it is easier to proof your input file. It also makes it easier to do "cut and paste" operations. If you accidentally delete =, it is easier to detect than accidentally removing part of one of the other delimiters that could be used.

1.5.8. Order of Commands

Commands have no ordering requirements. Both the input sequence:

```
BEGIN PRESCRIBED DISPLACEMENT
  NODE SET = nodelist_10
  COMPONENT = X
  FUNCTION = my_function
END PRESCRIBED DISPLACEMENT
```

and the input sequence:

```
BEGIN PRESCRIBED DISPLACEMENT
  FUNCTION = my_function
  COMPONENT = X
  NODE SET = nodelist_10
END PRESCRIBED DISPLACEMENT
```

are valid, and they produce the same result. Remember, that command lines and command blocks must appear in the proper scope.

1.5.9. Abbreviated END Specifications

It is possible to terminate a command block without including the key word or key words that identify the block. You could define a specific instance of the prescribed displacement boundary condition with:

```
BEGIN PRESCRIBED DISPLACEMENT
```

and terminate it with:

```
END
```

as opposed to this specification:

```
END PRESCRIBED DISPLACEMENT
```

Both the short termination (`END` only) and the long termination (`END` followed by identification, or name, of the command block) are valid. It is recommended that the long termination be used for any command block that becomes large. The `RESULTS OUTPUT` command block described in later chapters can become fairly lengthy, so this is probably a good place to use the long termination. For most boundary conditions, the command block will typically consist of five lines. In such cases, the short termination can be used. Using the long termination for the larger command blocks will make it easier to proof your input files. If you use the long termination, the text following the `END` key word must exactly match the text following the `BEGIN` key word. You could not have `BEGIN PRESCRIBED DISPLACEMENT` paired with an `END PRESCRIBED DISPL` to define the beginning and ending of a command block.

1.5.10. Indentation

When constructing an input file, it is useful, but not required, to indent a scope that is nested inside another scope. Command lines within a command block should also be indented in relation to the lines defining the command block. This will make it easier to construct the input file with everything in the correct scope and with all the command blocks in the correct structure.

1.5.11. Aprepro Text Processing

Aprepro is used for input deck text preprocessing. Generally any text inside curly brackets `{ }` will be interpreted by aprepro. This include text inside of comments. See [14] for more information. The sierra script can run aprepro automatically on an input file by adding a command line option as shown in this example:

```
sierra -a adagio -i sierra_input.i
```

Note, if there is text in a file surrounded by brackets `{ }`, and you do not want it to be processed by aprepro, it can be "escaped," or turned off for that text block, with the '`\`' character. For example

```
# Include an external file with aprepro
#   {include("matProps.inp")}

# Turn aprepro off processing so that matProps is not included
#   \{include("matProps.inp")}
```

1.5.12. Including Files

External text files containing input commands can be included at any point in the Sierra/SM input file using the aprepro `include` command. This command can be used in any context in the input file. To use this command, use the command `include` command followed by the name of the file to be included. For example, the command:

```
{include("displacement_history.i")}
```

would include the `displacement_history.i` as if the contents of that file were placed in the position that it is included in the input file. The included file is contained in the standard echo of the input provided at the beginning of the log file. If requested, aprepro substitution will occur inside of the included text. Note, the user must use double quotes around the name of the text file. If single quotes are used, aprepro will not be able to find the file.

1.6. Exodus II Database Naming Conventions

When the mesh file has an Exodus II format, there are three basic conventions that apply to user input for various command lines. The conventions concern side sets (surfaces), node sets and blocks.

First, for a mesh file with the Exodus II format, the Exodus II side set is referenced as a surface. In SIERRA, a surface consists of element faces plus all the nodes and edges associated with these faces. A surface definition can be used not only to select a group of faces but also to select a group of edges or a group of nodes that are associated with those faces. In the case of boundary conditions, a surface definition can be used not only to apply boundary conditions that typically use surface specifications (pressure) but also to apply boundary conditions for what are called nodal boundary conditions (fixed displacement components). For nodal boundary conditions that use the surface specification, all the nodes associated with the faces on a specific surface will have this boundary condition applied to them. The specification for a surface identifier in the following chapters is `surface_name`. It typically has the form `surface_integerid`, where `integerid` is the integer identifier for the surface. If the side set identifier is 125, the value of `surface_name` would be `surface_125`. It is also possible to generate an alias for the side set[1] and use this for `surface_name`. If `surface_125` is aliased to `outer_skin`, then `surface_name` becomes `outer_skin` in the actual input line. It is also possible to name a surface in some mesh generation programs and that name can be used in the input file.

Second, for a mesh file with the Exodus II format, the Exodus II node set is still referenced as a node set. A node set can be used only for cases where a group of nodes needs to be defined. The specification for a node set identifier in the following chapters is `nodelist_name`. It typically has the form `nodelist_integerid`, where `integerid` is the integer identifier for the node set. If the node set number is 225, the value of `nodelist_name` would be `nodelist_225`. It is also possible to generate an alias for the node set and use this for `nodelist_name`. If `nodelist_225` is aliased to `inner_skin`, then `nodelist_name` becomes `inner_skin` in the actual input line. It is also possible to name a nodelist in some mesh generation programs and that name can be used in the input file.

Third, an element block is referenced as a block. The specification for an element block identifier in the following chapters is `block_name`. It typically has the form `block_integerid`, where `integerid` is the integer identifier for the block. If the element block number is 300, the value of `block_name` would be `block_300`. It is also possible to generate an alias for the block and use this for `block_name`. If `block_300` is aliased to `big_chunk`, then `block_name` becomes `big_chunk` in the actual input line. It is also possible to name an element block in some mesh generation programs and that name can be used in the input file.

A group of elements can also be used to select other mesh entities. In SIERRA, a block consists of elements plus all the faces, edges, and nodes associated with the elements. The block and surface concepts are similar in that both have associated derived quantities. Chapters 7 and 8 show how this concept of derived quantities is used in the input command structure.

[1] See the `ALIAS` command in Section 6.1.2

1.7. Major Scope Definitions for an Input File

The typical input file will have the structure shown in the two examples below. The major scopes—SIERRA, procedure, and region—are delineated with input lines for command blocks. Comment lines are included that indicate some of the key scopes that will appear within the major scopes. Note the indentation used to clarify scopes. An example of the structure an explicit dynamic (Presto) input file is shown:

Explicit

```
BEGIN SIERRA <string>some_name
  #
  # All command blocks and command lines in the SIERRA
  # scope appear here. The PROCEDURE PRESTO command
  # block is the beginning of the next scope.
  #
  # function definitions
  # material descriptions
  # description of mesh file
  #
  BEGIN PRESTO PROCEDURE <string>procedure_name
    #
    # time step control
    #
    BEGIN PRESTO REGION <string>region_name
      #
      # All command blocks and command lines in the
      # region scope appear here
      #
      # specification for output of result
      # specification for restart
      # boundary conditions
      # definition of contact
      #
    END [PRESTO REGION <string>region_name]
  END [PRESTO PROCEDURE <string>procedure_name]
END [SIERRA <string>some_name]
```

This is an example of an implicit (Adagio) input file:

Implicit

```
BEGIN SIERRA <string>some_name
  #
  # All command blocks and command lines in the SIERRA
  # scope appear here. The PROCEDURE ADAGIO command
  # block is the beginning of the next scope.
  #
  # function definitions
  # material descriptions
```

```
  # description of mesh file
  #
  BEGIN ADAGIO PROCEDURE <string>procedure_name
    #
    # time step control
    #
    BEGIN ADAGIO REGION <string>region_name
      #
      # All command blocks and command lines in the
      # region scope appear here
      #
      # solver commands
      # specification for output of result
      # specification for restart
      # boundary conditions
      # definition of contact
      #
    END [ADAGIO REGION <string>region_name]
  END [ADAGIO PROCEDURE <string>procedure_name]
END [SIERRA <string>some_name]
```

1.8. Input/Output Files

The primary user input to Sierra/SM is the input file introduced in this chapter. Throughout this document, we explain how to construct a valid input file. Be aware that Sierra/SM also processes many other types of input files and produces a variety of output files. These additional files are also discussed in this document where applicable. Figure 1-1 shows a simple schematic diagram of the various input and output files.

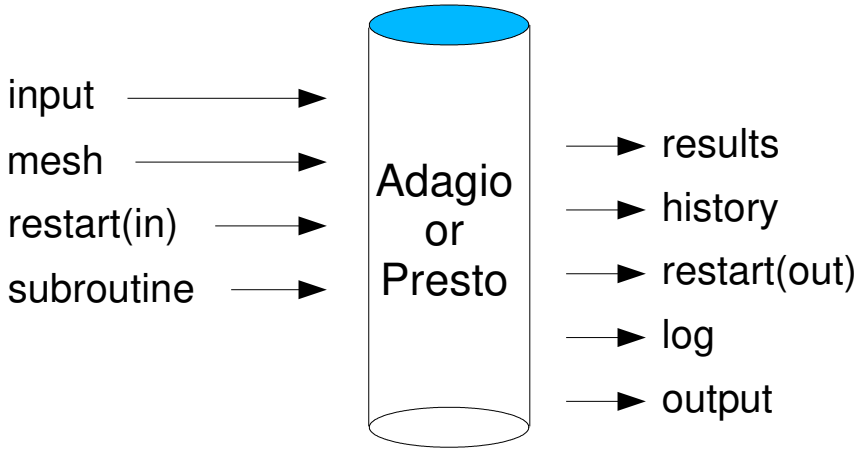


Figure 1-1. Input/output files

As shown in Figure 1-1, Sierra/SM uses the input file, mesh files, restart files, and user subroutine files. The input file, which is required, is a set of valid Sierra/SM command lines. Another required input is a mesh file, which provides a description of the finite element mesh for the object being analyzed. Restart and user subroutine files are optional inputs. The restart functionality lets you break an analysis from the start time to the termination time into a sequence of runs. The files generated by the restart functionality contain a complete state description for a problem at various analysis times, which we will refer to as restart times. You can restart Sierra/SM at any of these restart times because the complete state description is known (see Chapter 9). The user subroutine files let you build and incorporate specialized functionality into Sierra/SM (Chapter 10).

As also shown in Figure 1-1, Sierra/SM can generate a number of files. These include results files, history files, restart files, a log file, and an output file. Typically, only the log file and the output file are produced automatically. Generation of the other types of files is based on user settings in the input file for the particular output desired. Results files provide the values of global variables, element variables, and node variables at specified times (see Chapter 9). History files will also provide values of global variables, element variables, and node variables at specified times (see Chapter 9). History files are set up to provide a specific value at a specific node, for example, whereas results files provide a nodal value for large subsets of nodes or, more typically, all nodes. History files provide a more limited set of information than results files. As noted above, restart files can be generated at various analysis times. The log file contains a variety of information such as the Sierra/SM version number, a listing of the input file, initialization information, some model information (mass, critical time steps for element blocks, etc.), and information at various time steps. At every n^{th} step, where n is user selected, the log file gives the current analysis time; the

current time step; the kinetic, internal, and external energies; the error in the energy; and computing time information. You can monitor step information in the log file to gain information about how your analysis is progressing. The output file contains error information.

1.9. Obtaining Support

Support for all SIERRA Mechanics codes, including Sierra/SM, can be obtained by contacting the SIERRA Mechanics user support team by email at sierra-help@sandia.gov. Users on the Sandia Restricted Network may consult the Computational Simulation web pages (`http://compsim.sandia.gov/compsim/`) for "Getting Started" information and basic troubleshooting guidance of general Sierra issues.

1.9.1. Helpful Information to Include when Requesting Support

When making an inquiry to sierra-help, providing the following information will be helpful to get the most useful and timely response.

- A description of the problem, such as error message obtained or what behavior was observed versus what was expected.
- The input deck, log file, and mesh files associated with the run. If these files are under 4MB they can be sent directly via email to sierra-help. Alternatively the path to the directory with the run in question may be given. If the files related to the run are large or have access restrictions contact sierra-help to determine the best method to provide the user support team the relevant information about the run.
- Any information that may help the development team narrow down the problem. Such as which input deck command was added to an analysis that caused an error, if the problem ran in a previous version of the code, or if the same error was seen on other analyses.
- The priority of the request as it relates to your analysis. For example, is the problem a show-stopper for the analysis or is it merely an inconvenience? Is there a deadline looming that makes resolution of your issue a high priority?

References

[1] H.C. Edwards and J.R. Stewart. Sierra framework for developing advanced parallel mechanics applications. In L.T. Biegler, editor, *Large-Scale PDE-Constrained Optimization*, 2003.

[2] H.C. Edwards and J.R. Stewart. A framework approach for developing parallel adaptive multiphysics applications. *Finite Elements in Analysis and Design*, 40(12), 2004.

[3] H.C. Edwards and J.R. Stewart. SIERRA: A software environment for developing complex multi-physics applications. In K.J. Bathe, editor, *First MIT Conference on Computational Fluid and Solid Mechanics*, pages 1147–1150, Amsterdam, 2001. Elsevier.

[4] J.R. Koteras, A.S. Gullerud, V.L. Porter, W.M. Scherzinger, and K.H. Brown. PRESTO: Impact dynamics with scalable contact using the SIERRA framework. In K.J. Bathe, editor, *First MIT Conference on Computational Fluid and Solid Mechanics*, pages 294–296, Amsterdam, 2001. Elsevier.

[5] J.H. Biffle. JAC – a two-dimensional finite element computer program for the non-linear quasi-static response of solids with the conjugate gradient method. Technical Report SAND81-0998, Sandia National Laboratories, Albuquerque, NM, 1984. pdf.

[6] M.L. Blanford, M.W. Heinstein, and S.W. Key. JAS3D – a multi-strategy iterative code for solid mechanics analysis users' instructions, release 2.0, September 2001.

[7] L.M. Taylor and D.P. Flanagan. Pronto3D: A three-dimensional transient solid dynamics program. Technical Report SAND87-1912, Sandia National Laboratories, Albuquerque, NM, March 1989. pdf.

[8] S.W. Attaway, K.H. Brown, F.J. Mello, M.W. Heinstein, J.W. Swegle, J.A. Ratner, and R.I. Zadoks. Pronto3D user's instructions: A transient dynamic code for nonlinear structural. Technical Report SAND98-1361, Sandia National Laboratories, Albuquerque, NM, 1998. pdf.

[9] T.A. Laursen, S.W. Attaway, and R.I. Zadoks. SEACAS theory manuals: Part I. problem formulation in nonlinear solid mechanics. Technical Report SAND98-1760/1, Sandia National Laboratories, Albuquerque, NM, 1998. pdf.

[10] T.A. Laursen, S.W. Attaway, and R.I. Zadoks. SEACAS theory manuals: Part II. nonlinear continuum mechanics. Technical Report SAND98-1760/2, Sandia National Laboratories, Albuquerque, NM, 1998. pdf.

[11] T.A. Laursen, S.W. Attaway, and R.I. Zadoks. SEACAS theory manuals: Part III. finite element analysis in nonlinear solid mechanics. Technical Report SAND98-1760/3, Sandia National Laboratories, Albuquerque, NM, 1999. pdf.

[12] L.A. Schoof and V.R. Yarberry. EXODUS II: A finite element data model. Technical Report SAND92-2137, Sandia National Laboratories, Albuquerque, NM, September 1994. pdf. See also documentation available at EXODUS II Sourceforge page. link.

[13] K.H. Brown, R.M. Summers, M.W. Glass, A.S. Gullerud, M.W. Heinstein, and R.E. Jones. ACME: Algorithms for contact in a multiphysics environment, API version, 1.0. Technical Report SAND2001-3318, Sandia National Laboratories, Albuquerque, NM, 2001. pdf.

[14] G.D. Sjaardema. APREPRO: An algebraic preprocessor for parameterizing finite element analyses. Technical Report SAND92-2291, Sandia National Laboratories, Albuquerque, NM, 1999. pdf.

2. GENERAL COMMANDS

The commands described in this section appear in the SIERRA or procedure scope or control general functionality in Sierra/SM.

2.1. SIERRA Scope

These commands are used to set up some of the fundamentals of the Sierra/SM input. The commands are physics independent, or at least can be shared between physics. The commands lie in the SIERRA scope, not in the procedure or region scope.

2.1.1. SIERRA Command Block

```
BEGIN SIERRA <string>name
  #
  # All other command blocks and command lines
  # appear within the SIERRA scope defined by
  # begin/end sierra.
  #
END [SIERRA <string>name]
```

All input commands must occur within a `SIERRA` command block. The syntax for beginning the command block is:

```
BEGIN SIERRA <string>name
```

and for terminating the command block is as follows:

```
END [SIERRA <string>name]
```

In these input lines, `name` is a name for the `SIERRA` command block. All other commands for the analysis must be within this command block structure. The name for the `SIERRA` command block is often a descriptive name that identifies the analysis. The name is not currently used anywhere else in the file and is completely arbitrary.

2.1.2. Title

```
TITLE <string list>title
```

To permit a fuller description of the analysis, the input has a `TITLE` command line for the analysis, where `title` is a text description of the analysis. The title is transferred to the results file.

2.1.3. Restart Control

Restart allows a user to resume an analysis by initializing all of its fields from a specific time of a previous simulation. For more background information on restart, see Section 9.6.

The time at which to restart an analysis can be specified in either the Sierra scope or the `RESTART DATA` command block in the region scope. If the restart time is specified in the Sierra scope, a `RESTART DATA` command block is still required to specify input and output restart data filenames. When performing a multi-physics simulation involving coupling Sierra/SM to another Sierra code, it is beneficial to specify the restart time in the Sierra scope. However, if the analysis involves only Sierra/SM, it is sufficient to specify a restart time only in the `RESTART DATA` command block.

If using restart commands in the Sierra scope, one of two restart command lines (`RESTART` or `RESTART TIME`) are used. The details of these two commands are described below:

2.1.3.1. Restart Time

```
RESTART TIME = <real>restart_time
```

The `RESTART TIME` command line specifies a time from which to restart the simulation. This restart option will pick the restart time on the restart file that is closest to the user-specified time in the `RESTART TIME` command line. If the user specifies a restart time greater than the last time written to a restart file, the last time written to the restart file is picked as the restart time. If this command is specified, a restart file must exist. If the input restart database does not exist, the code will exit with an error. For details of this feature and a use case example, see Section9.6.1.3.

Using the `RESTART TIME` command requires the user to be more active in the management of the file names to prevent the overwriting of restart, results, and history files. To prevent the overwriting of existing restart files, an input restart file and an output restart file can be specified in the `RESTART DATA` command block. To prevent the overwriting of results and history files, the filenames must be changed in the respective output blocks. It is best to break the analysis into a sequence of 'restart runs' where each 'restart run' will consist of its own restart, results, and history files. For more information on overwriting of output files, see Chapter 9. If the user would prefer not to manage the overwriting of files, the automatic restart feature should be used. This feature is outlined below in Section 2.1.3.2.

2.1.3.2. Automatic Restart

```
RESTART = AUTOMATIC|AUTO
```

The AUTOMATIC or AUTO restart option automatically selects the last time written to the last restart file as the restart time. The automatic restart feature allows the user to restart runs with minimal changes to the input file. The only quantity that must be changed to move from one restart to another is the termination time. The RESTART command line manages the restart files to ensure any previous restart files are not overwritten. It also manages the results and history files to ensure any previous results or history files are not overwritten as well. For details of this feature and a use case example, see Section 9.6.1.2.

If the AUTOMATIC option is used and the restart database does not exist, the analysis will start from the initial time and not read any restart data. Note, the path to the restart file must be carefully defined when using the AUTOMATIC option. If there is a typo in the path or file name that prevents the correct restart database from being found, the analysis will start from the beginning of the first time period without error.

See Section 9.6 for a full discussion of implementing the restart capability.

2.1.4. User Subroutine Identification

```
USER SUBROUTINE FILE = <string>file_name
```

This command line is a part of a set of commands that are used to implement the user subroutine functionality. The string file_name identifies the name of the file that contains the FORTRAN code of one or more user subroutines.

To understand how this command line is used, see Chapter 10.

2.1.5. XY Functions

```
BEGIN FUNCTION <string>function_name
  TYPE = <string>CONSTANT|PIECEWISE LINEAR|PIECEWISE CONSTANT|
    ANALYTIC|PIECEWISE ANALYTIC|MULTICOLUMN PIECEWISE LINEAR|
    PIECEWISE MULTIVARIATE
  EXPRESSION VARIABLE: <var_name> = NODAL|NODAL_VECTOR|
    NODAL_TENSOR|NODAL_SYM_TENSOR|ELEMENT|ELEMENT_VECTOR|
    ELEMENT_TENSOR|ELEMENT_SYM_TENSOR|FACE|
    GLOBAL <var_source>
  ABSCISSA = <string>abscissa_label
  ORDINATE = <string>ordinate_label
  X SCALE = <real>x_scale(1.0)
  X OFFSET = <real>x_offset(0.0)
  Y SCALE = <real>y_scale(1.0)
  Y OFFSET = <real>y_offset(0.0)
  ABSCISSA SCALE = <real>x_scale(1.0)
  ABSCISSA OFFSET = <real>x_offset(0.0)
  ORDINATE SCALE = <real>y_scale(1.0)
  ORDINATE OFFSET = <real>y_offset(0.0)
  BEGIN VALUES
```

```
    <real>x_1   <real>y_1
    <real>x_2   <real>y_2
    ...
    <real>x_n   <real>y_n
  END [VALUES]
  BEGIN EXPRESSIONS
    <real>x_1   <string>analytic_expression_1
    <real>x_2   <string>analytic_expression_2
    ...
    <real>x_n   <string>analytic_expression_n
  END
  DATA FILE = <string>fileName
    [X FROM COLUMN <int>xCol Y FROM COLUMN <int>yCol]

  AT DISCONTINUITY EVALUATE TO <string>LEFT|RIGHT(LEFT)
  EVALUATE EXPRESSION = <string>analytic_expression1;
    analytic_expression2; ...
  DIFFERENTIATE EXPRESSION = <string>analytic_expression1;
    analytic_expression2; ...
  EVALUATE FROM <real> TO <real> BY <real>
  DEBUG = ON|OFF(OFF)
END [FUNCTION <string>function_name]
```

Many Sierra/SM features are driven by a user-defined description of the dependence of one variable on another. For instance, the prescribed displacement boundary condition requires the definition of a time-versus-displacement relation, and the thermal strain computations require the definition of a thermal-strain-versus-temperature relation. SIERRA provides a general method of defining these relations as functions using the `FUNCTION` command block, as shown above.

Many options exist for defining a general X versus Y function. More limited options exist for defining more complex functions of multiple variables. The more complex functions are described in Section 2.1.6.

There is no limit to the number of functions that can be defined. All function definitions must appear within the SIERRA scope.

A description of the various parts of the `FUNCTION` command block follows:

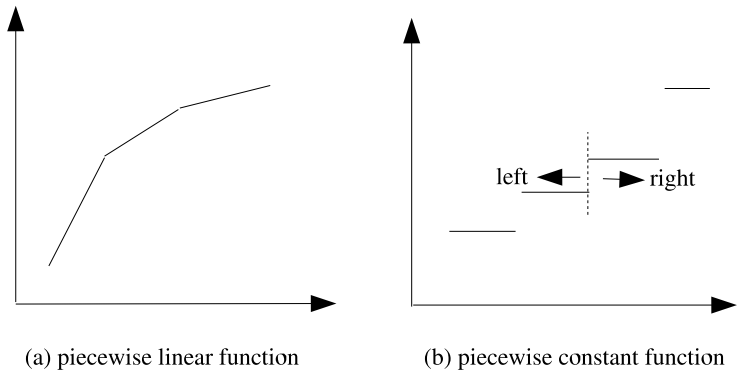
- The string `function_name` is a user-selected name for the function that is unique to the function definitions within the input file. This name is used to refer to this function in other locations in the input file.
- The `TYPE` command line has seven options to define the type of function. The value of this string can be `CONSTANT`, `PIECEWISE LINEAR`, `PIECEWISE CONSTANT`, `ANALYTIC`, `PIECEWISE ANALYTIC`, `MULTICOLUMN PIECEWISE LINEAR`, or `PIECEWISE MULTIVARIATE`.
- The `ABSCISSA` command line provides a descriptive label for the independent variable (x-axis) with the string `abscissa_label`. This command line is optional. (Note: Previous versions of SIERRA allowed the user to optionally add a scale factor and/or an offset via the `SCALE` and `OFFSET` keywords. These keywords are deprecated and their use will

generate a parse error. Any desired scale factors and offsets need to be specified via the X SCALE and/or X OFFSET command lines, respectively.)

- The ORDINATE command line provides a descriptive label for the dependent variable (y-axis) with the string ordinate_label. This command line is optional. (Note: Previous versions of SIERRA allowed the user to optionally add a scale factor and/or an offset via the SCALE and OFFSET keywords. These keywords are deprecated and their use will generate a parse error. Any desired scale factors and offsets need to be specified via the Y SCALE and/or Y OFFSET command lines, respectively.)
- The X SCALE command line sets the scale factor value for the abscissa, which has the following effect: $abscissa_{scaled,offset} = scale * (abscissa + offset)$. (Note: If no offset is specified via X OFFSET, $offset = 0.0$.) This command is not available for analytic or piecewise analytic functions. ABSCISSA SCALE is an alias for X SCALE.
- The X OFFSET command line sets the offset value for the abscissa, which has the following effect: $abscissa_{scaled,offset} = scale * (abscissa + offset)$. (Note: If no scale factor is specified via X SCALE, $scale = 1.0$.) This command is not available for analytic or piecewise analytic functions. ABSCISSA OFFSET is an alias for X OFFSET.
- The Y SCALE command line sets the scale factor value for the ordinate, which has the following effect: $ordinate_{scaled,offset} = scale * (ordinate + offset)$. (Note: If no offset is specified via Y OFFSET, $offset = 0.0$.) ORDINATE SCALE is an alias for Y SCALE.
- The Y OFFSET command line sets the offset value for the ordinate, which has the following effect: $ordinate_{scaled,offset} = scale * (ordinate + offset)$. (Note: If no scale factor is specified via Y SCALE, $scale = 1.0$.) ORDINATE OFFSET is an alias for Y OFFSET.
- The ABSCISSA SCALE command line is an alias for X SCALE.
- The ABSCISSA OFFSET command line is an alias for X OFFSET.
- The ORDINATE SCALE command line is an alias for Y SCALE.
- The ORDINATE OFFSET command line is an alias for Y OFFSET.
- The DEBUG command line prints functions to the log file if they were scaled and/or offset. This command line is optional and available for PIECEWISE LINEAR and PIECEWISE CONSTANT functions. The generated function name is the original function name concatenated with _with_scale_and_offset_applied. The generated function is a valid function and could be placed into the input file and used. The DEBUG command can also be used to print ANALYTIC and PIECEWISE ANALYTIC functions to the log file if they are evaluated using the command EVALUATE FROM <real> TO <real> BY <real>. The generated function name is the original function name concatenated with _as_piecewise_linear.
- The VALUES command block consists of the real value pairs (x_1,y_1) through (x_n, x_n), which describe the function. This command block must be used if the value on the TYPE command line is CONSTANT, PIECEWISE LINEAR, or PIECEWISE CONSTANT. For a CONSTANT function, only one value is needed. For a PIECEWISE LINEAR or PIECEWISE CONSTANT function, the values are (x, y) pairs of data that describe the function. The values are nested inside the VALUES command block.
- As an alternative to the VALUES command block, the X and Y data pair values can be read from an external file with the DATA FILE command. An example data file format is given below. All characters in a file line that follow the first '#' character in that line are considered as comments and ignored. The remainder of the file contains columns of Real

numbers. The columns can be separated by white space, commas, or some combination of white space and commas. Optionally, the columns of data to represent X and Y can be specified by the `X FROM COLUMN <int>xCol Y FROM COLUMN <int>yCol` command. If no optional column number is specified, it is assumed that column 1 represents the X data and column 2 represents the Y data. The data file option is valid for piecewise linear and piecewise constant functions.

```
#-----------------------------------------
# Time    |    Pressure   |   Temperature   |
#-----------------------------------------
   0.0         1.0e-3         1.5
   0.1         1.2e-3         1.7
   0.2         0.9e-3         2.9
   0.3         0.1e-3         9.0
```



(a) piecewise linear function (b) piecewise constant function

Figure 2-1. Piecewise linear and piecewise constant functions

- A `PIECEWISE LINEAR` function performs linear interpolations between the provided value pairs; a `PIECEWISE CONSTANT` function is constant valued between provided value pairs. Figure 2-1 (a) shows an example of a piecewise linear function, and Figure 2-1 (b) shows an example of a piecewise constant function.
- For functions that are based on tabular values, such as the `PIECEWISE LINEAR` and `PIECEWISE CONSTANT` functions, there is a possibility that the function may be evaluated for abscissa values that fall outside the range of the tabulated values. If the abscissa for which the function is to be evaluated is lower than the lowest tabulated abscissa, the function returns the ordinate corresponding to the smallest tabulated abscissa. Likewise, if the function is evaluated for an abscissa higher than the highest tabulated abscissa, the function returns the ordinate corresponding to the highest tabulated abscissa. For example, consider the following function:

```
begin function my_func
  type = piecewise linear
  begin values
    5.0  0.0
    10.0 50000.0
  end values
```

```
end function my_func
```

For values less than 5, this function returns 0, and for values greater than 10, it returns 50000.

- `PIECEWISE LINEAR` and `PIECEWISE CONSTANT` functions must be defined with either monotonically increasing x-values ($x_i >= x_{i-1}$) or monotonically decreasing x-values ($x_i <= x_{i-1}$). For example, the following function definition is invalid:

```
begin function my_func
  type = piecewise linear
  begin values
    0.0  0.0
    0.5  0.0
    1.0  1.0
    0.5  1.0
  end values
end function my_func
```

- For a piecewise constant function, a constant valued segment ends on the left hand side of an abscissa value and a new constant value segment begins on the right hand side of the same abscissa value. (This transition from one constant value to another is indicated by the dotted line in Figure 2-1 (b).) When the value of the function is to be evaluated at a discontinuity, where two potential values exist for the ordinate, the default behavior is to use the ordinate from the value pair that has the higher-valued abscissa. In other words, to use the value on the right hand side of the discontinuity. The `AT DISCONTINUITY EVALUATE TO` command line can be used to override this default behavior at an abscissa with two ordinate values. The command line can have a value of either `LEFT` or `RIGHT`. If `RIGHT` (the default) is specified, the ordinate value to the right of the abscissa is used; if `LEFT` is specified, the ordinate value to the left of the abscissa is used.
- The `PIECEWISE MULTIVARIATE` function is used with the Hydra Plasticity Material Model.
- The `MULTICOLUMN PIECEWISE LINEAR` function is not used in Sierra Solid Mechanics. It can be used in coupled analysis with Aria, Fuego, etc.
- The `EVALUATE EXPRESSION` command line consists of one or more user-supplied algebraic expressions. This command line must be used if the value on the `TYPE` command line is `ANALYTIC`. See the rules and options for composing algebraic expressions discussed below.
- The `DIFFERENTIATE EXPRESSION` command line consists of one or more user-supplied algebraic expressions. This command line must be used if the value on the `TYPE` command line is `ANALYTIC` and you need the derivative of the analytic function that you defined with `EVALUATE EXPRESSION`. This can happen if you are using an analytic function in a constitutive model. See the rules and options for composing algebraic expressions discussed below.

Importantly, a `FUNCTION` command block cannot contain both a `VALUES` command block and an `EVALUATE EXPRESSION` command line.

Rules and options for composing algebraic expressions. If you choose to use the `EVALUATE EXPRESSION` command line, you will need to write the algebraic expressions. The algebraic expressions are written using a C-like format. Each algebraic expression is terminated by a semicolon(;). The entire set of algebraic expressions, whether a single expression or several, is enclosed in a single set of double quotes(" ").

Expressions compute an output value as a function of input values. The input value of an expression is called an independent variable of the expression. An expression can use any name for the expression independent variable. The examples shown here use `x` for the independent variable.

Example: Return `sin(x)` as the value of the function.

```
begin function sinx
  type is analytic
  evaluate expression is "sin(x)"
end
```

Example: Return a piecewise linear interpolated ramp function. For x values less than 0.0 the function will return 0.0. For x values between 0.0 and 0.5 the function will return y values linearly interpolated between 0 and 100. For x values greater than 0.5 the function will return 100:

```
begin function pressure
  type is analytic
  evaluate expression is "   \#
    (x <= 0.0) ? (           \#
      0.0                    \#
    ) : (                    \#
      (x < 0.5) ? (          \#
        x*200.0              \#
      ) : (                  \#
       100.0                 \#
      )                      \#
    )                        \#
   "
end
```

Operators and Functions Available within Expressions The following functionality is currently implemented for the expressions:

Operators Valid arithmetic and Boolean operations are as follows:

Symbol	Operation
`+, -`	plus, minus
`*, /`	times, divide
`^`	power
`==, !=`	equivalent, not equivalent
`>, <`	greater/less than
`>=, <=`	greater/less than or equal to
`!`	not
`&, |, &&, ||`	and, or
`?, :`	ternary if, then, else

Parentheses `()`, represents standard mathematical meaning of operation ordering.

Component indexing `value[i]`, accesses the *i*th component of a multi-component variable such as a vector, tensor, etc. This index is one-based. For a vector `vec`,

```
vec_x = vec[1]
vec_y = vec[2]
vec_z = vec[3]
```

for a symmetric tensor `sym`,

```
sym_xx = sym[1]
sym_yy = sym[2]
sym_zz = sym[3]
sym_xy = sym[4]
sym_yz = sym[5]
sym_zx = sym[6]
```

and for a full tensor `full`,

```
full_xx = full[1]
full_yy = full[2]
full_zz = full[3]
full_xy = full[4]
full_yz = full[5]
full_zx = full[6]
full_yx = full[7]
full_zy = full[8]
full_xz = full[9]
```

Math functions Valid mathematical operations are as follows:

Function	Operation
`abs(x)`	absolute value of `x`
`mod(x,y)`	modulus of `x`|`y`
`min(x,y)`	minimum value of `x`, `y`
`max(x,y)`	maximum value of `x`, `y`
`sign(x)`	sign operator: -1 if `x` is negative, 1 if positive
`ipart(x)`	integer part of `x`
`fpart(x)`	fractional part of `x`

Power functions Valid functions for expressions containing exponents are as follows:

Function	Operation
`pow(x,y)`	x^y
`pow10(x)`	10^x
`sqrt(x)`	$\sqrt{x}$

Trigonometric functions Valid trigonometric operations are as follows:

Function	Operation
`acos(x)`	arccosine of `x`
`asin(x)`	arcsine of `x`
`asinh(x)`	inverse hyperbolic sin of `x`
`atan(x)`	arctangent of `x`
`atan2(y,x)`	arctangent of `y/x`, signs of `x` and `y`
`cos(x)`	cosine of `x`
`cosh(x)`	hyperbolic cosine of `x`
`sin(x)`	sine of `x`
`sinh(x)`	hyperbolic sine of `x`
`tan(x)`	tangent of `x`
`tanh(x)`	hyperbolic tangent of `x`

Logarithm functions Valid logarithmic operations are as follows:

Function	Operation
`log(x)` or `ln(x)`	natural logarithm of `x`
`log10(x)`	base-10 logarithm of `x`
`exp(x)`	e^x

Rounding functions Valid mathematical rounding operations are as follows:

Function	Operation
`ceil(x)`	smallest integer value not less than `x`
`floor(x)`	largest integer value not greater than `x`

Random functions Valid functions for generating random data are as follows:

Function	Operation
`random()`	random real number $0.0 \leq x < 1.0$
`random(x)`	seeds the random number generator
`time()`	elapsed time in seconds since January 1, 1970

Coordinate conversion functions Valid functions for converting to and from polar and Cartesian coordinate systems are as follows:

Function	Operation
`deg(x)`	converts `x` from radians to degrees
`rad(x)`	converts `x` from degrees to radians
`recttopolr(x,y)`	magnitude of vector (`x`, `y`)
`recttopola(x,y)`	angle of vector (`x`, `y`)
`poltorectx(r,th)`	x-coordinate of angle `th` at distance `r`
`poltorecty(r,th)`	y-coordinate of angle `th` at distance `r`

If, then, else syntax `A ? B : C`, where A is a Boolean statement such as `(x < 2.0)`, B is a statement (or set of statements) to be evaluated if A is true, and C is a statement (or set of statements) to be evaluated if A is false.

Other utilities Miscellaneous other utility functions are as follows:

- `cos_ramp(x, xstart, xend)` defines a cosine ramp loading function. For `x < xstart`, it returns 0.0. For `x > xend`, it returns 1.0. If `xstart <= x <= xend`, the function has the value `(1.0-cos(Pi*(x-xstart)/(xend-xstart)))/2.0`. The primary purpose for the cosine ramp is to provide smooth loading. If attempting to run a nearly quasistatic problem with the dynamic solver, a displacement boundary condition applied via the `cos_ramp` will generally give a smooth response for a given loading time.
- `cycloidal_ramp(x, xstart, xend)` defines a cycloidal ramp function. For `x < xstart` it returns 0.0. For `x > xend`, it returns 1.0. If `xstart <= x <= xend`, the function has the value `(x-xstart) /(xend-xstart)-1/(2*pi)*sin(2*pi/(xend-xstart)*(x-xstart))`. The primary purpose for the cycloidal front ramp is to provide smooth loading for prescribed displacements and velocities. This function has continuous acceleration derivatives at `xstart` and `xend`, providing a smoother response than the `cos_ramp` for prescribed displacement boundary conditions.
- `haversine_pulse(x, xstart, xend)` defines a haversine pulse function. For `x < xstart`, it returns 0.0. For `x > xend`, it returns 0.0. If `xstart <= x <= xend`, the function has the value `pow(sin(Pi*(x-xstart)/(xend-xstart)),2)`. Design shock loads for components are often specified as haversine acceleration pulses; this function is included to make it easier to apply this loading.

Constants. There are three predefined constants that may be used in an expression. These three constants are `e`, `pi`, and `two_pi`. Note that these constants are reserved variable names and thus cannot be redefined in an expression.

Math symbol	Expression	Value
e	`e`	2.7182818284...
π	`pi`	3.1415926535...
2π	`two_pi`	$2 * 3.1415926535\ldots$

Also, there are four predefined constant functions that can be used, `SIERRA_CONSTANT_FUNCTION_ZERO`, `SIERRA_CONSTANT_FUNCTION_ONE`, `SIERRA_LINEAR_RAMP_FUNCTION`, and `SIERRA_COS_RAMP_FUNCTION`. The first two functions are equivalent to defining functions with constant expressions "0.0" or "1.0", respectively. The third function is the equivalent to a linear ramp in time with a slope of 1.0, and the fourth function is identical to a `cos_ramp` that ramps from 0.0 at the start time of the simulation to 1.0 at the termination time. See Appendix I for a full list of pre-defined options that can be used in an input deck.

The `PIECEWISE ANALYTIC` function is a combination of several analytic functions that are evaluated over different ranges of x. As an example consider the following function. The function ramps up a load from time 0.0 to 1.0, holds that load constant from time 1.0 to 2.0, ramps the load back down to zero from time 2.0 to 3.0, and then holds the load constant again after time 3.0.

```
begin function force_ramp
  type is piecewise analytic
  begin expressions
    0.0 "(1.0-cos(pi*x))/2.0"
    1.0 "1.0"
    2.0 "(1.0+cos(pi*(x-2.0)))/2.0"
    3.0 "0.0"
  end
end
```

Note this function is equivalent to the single compound, `ANALYTIC` function below, though the piecewise version tends to be easier to read. The expression compound if then else statement below also demonstrates the recommended indentation for nested if statements. The statement could be written on a single line, but the indentation makes the logic of the evaluation statement more readable.

```
begin function force_ramp
  type is analytic
  evaluate expression = "            \#
    (x < 1.0) ? (                    \#
      (1.0-cos(pi*x))/2.0            \#
    ) : (                            \#
      (x < 2.0) ? (                  \#
        1.0                          \#
```

```
        ) : (                              \#
          (x < 3.0) ? (                    \#
            (1.0+cos(pi*(x-2.0)))/2.0      \#
          ) : (                            \#
            0.0                            \#
          )                                \#
        )                                  \#
      )                                    \#
    "
  end
```

2.1.6. General Functions

```
BEGIN FUNCTION <string>function_name
  TYPE = ANALYTIC|PIECEWISE ANALYTIC

  EXPRESSION VARIABLE: <var_name> = NODAL|NODAL_VECTOR|
     NODAL_TENSOR|NODAL_SYM_TENSOR|ELEMENT|ELEMENT_VECTOR|
     ELEMENT_TENSOR|ELEMENT_SYM_TENSOR|FACE|GLOBAL <var_source>

  BEGIN EXPRESSIONS
    <real>x_1   <string>analytic_expression_1
    <real>x_2   <string>analytic_expression_2
    ...
    <real>x_n   <string>analytic_expression_n
  END
  AT DISCONTINUITY EVALUATE TO <string>LEFT|RIGHT(LEFT)
  EVALUATE EXPRESSION = <string>analytic_expression1;
    analytic_expression2; ...
END [FUNCTION <string>function_name]
```

General functions can compute a global, element, or nodal scalar value that is a function of several input values. Each input value to the function is explicitly defined via an `EXPRESSION VARIABLE` command. The expression variable command defines variable for use in an expression or expression list. As an example consider a function to apply a combination of shearing and extension to a block. An example displacement map going from time zero to time one is shown in Figure 2-2. The equation for this displacement map is given by:

$$\Delta_x(x,y,t) = t*(0.5*x+0.5*y) \tag{2.1}$$

This displacement map can be applied by a combination of a general function and a displacement boundary condition by:

```
BEGIN FUNCTION delta_x
  TYPE = analytic
```

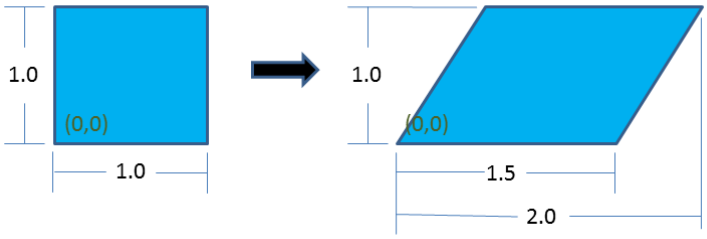


Figure 2-2. Example Displacement Map

```
  EXPRESSION variable: mx   = NODAL model_coordinates(x)
  EXPRESSION variable: my   = NODAL model_coordinates(y)
  EXPRESSION variable: time = GLOBAL time
  EVALUATE EXPRESSION = "time * (0.5*mx + 0.5*my)"
END

BEGIN PRESCRIBED DISPLACEMENT
  BLOCK = BLOCK_1
  COMPONENT = X
  FUNCTION = delta_x
END
```

The model_coordinates are the initial nodal coordinates. Time is the current analysis time. If a general function is being applied to a nodal boundary condition, such as displacement, function will be evaluated at each node. On a node the value of model_coordinates will be extracted and placed into the expression variables ‘mx’ and ‘my.’ The analysis time is placed into the expression variable ‘time.’ The expression is then evaluated producing a single value at the node, the nodal value is used to define the x displacement at that particular node.

A function used to define nodal quantities may reference NODAL or GLOBAL expression variables. A function used to define an element quantity may reference ELEMENT or GLOBAL expression variables. A function used to define a global quantity may only reference GLOBAL expression variables. Generally any quantity that would be available for output at nodes, at elements, or as a global variable may be used as an expression variable.

Expression variables may also accept vectors, tensors or symmetric tensors. A vector can define variables in terms of `x`, `y`, and `z`. For example,

```
EXPRESSION variable: A = element_vector velocity
```

would define `A_x`, `A_y`, and `A_z` for use in an expression. A symmetric tensor can define variables in terms of `xx`, `yy`, `zz`, `zy`, `yz`, and `zx`. For example,

```
EXPRESSION variable: A = element_sym_tensor unrotated_stress
```

would define `A_xx`, `A_yy`, `A_zz`, `A_xy`, `A_yz`, and `A_zx` for use in an expression. A tensor can define variables in terms of `xx`, `yy`, `zz`, `zy`, `yz`, `zx`, `yz`, `xz`, and `zy`.

```
EXPRESSION variable: A = element_tensor rotation
```

would define A_xx, A_yy, A_zz, A_xy, A_yz, A_zx, A_yx, A_zy, and A_xz. Particularly for tensors, this provides a clearer syntax for accessing the components of a multi-component variable.

Example of use:

```
# Define using named vector components
BEGIN FUNCTION velocity_vec
  TYPE = analytic
  EXPRESSION variable: V = nodal_vector velocity
  EVALUATE EXPRESSION = "sqrt((V_x)^2 + (V_y)^2 + (V_z)^2 )"
END

# Equivalent function using general variable indexing
BEGIN FUNCTION velocity_index
  TYPE = analytic
  EXPRESSION variable: V = nodal velocity
  EVALUATE EXPRESSION = "sqrt(V[1]^2 + V[2]^2 + V[3]^2 )"
END

...

BEGIN USER OUTPUT
  # speed1 and speed2 are equivalent, both are a nodal field that
  # is the local magnitude of the nodal velocity field
  compute nodal speed1 as function velocity_vec
  compute nodal speed2 as function velocity_index
END
```

2.1.7. Axes, Directions, and Points

```
DEFINE POINT <string>point_name WITH COORDINATES
  <real>value_1 <real>value_2 <real>value_3
DEFINE DIRECTION <string>direction_name WITH VECTOR
  <real>value_1 <real>value_2 <real>value_3
DEFINE AXIS <string>axis_name WITH POINT
  <string>point_1 POINT <string>point_2
DEFINE AXIS <string>axis_name WITH POINT
  <string>point DIRECTION <string>direction
```

Many Sierra/SM features require the definition of geometric entities. For instance, the prescribed displacement boundary condition requires a direction definition, and the cylindrical velocity initial condition requires an axis definition. Currently, Sierra/SM input permits the definition of points, directions, and axes. Definition of these geometric entities occurs in the SIERRA scope.

The DEFINE POINT command line is used to define a point:

```
DEFINE POINT <string>point_name WITH COORDINATES
  <real>value_1 <real>value_2 <real>value_3
```

where

- The string `point_name` is a name for this point. This name must be unique to all other points defined in the input file.
- The real values `value_1`, `value_2`, and `value_3` are the x, y, and z coordinates of the point.

The predefined point `SIERRA_POINT_ORIGIN` can be used to specify a point at the origin of the global coordinate system. Refer to Appendix I for a full list of predefined options that can be used in an input deck.

The `DEFINE DIRECTION` command line is used to define a direction:

```
DEFINE DIRECTION <string>direction_name WITH VECTOR
  <real>value_1 <real>value_2 <real>value_3
```

where

- The string `direction_name` is a name for this direction. This name must be unique to all other directions defined in the input file.
- The real values `value_1`, `value_2`, and `value_3` are the x, y, and z magnitudes of the direction vector.

The predefined directions `SIERRA_DIRECTION_X`, `SIERRA_DIRECTION_Y`, `SIERRA_DIRECTION_Z`, `SIERRA_DIRECTION_NEG_X`, `SIERRA_DIRECTION_NEG_Y`, and `SIERRA_DIRECTION_NEG_Z` can be used to specify the global axis directions +x, +y, +z, -x, -y, and -z, respectively. Refer to Appendix I for a full list of predefined options that can be used in an input deck.

There are two command lines that can be used to define an axis. The first `DEFINE AXIS` command line uses two points:

```
DEFINE AXIS <string>axis_name WITH POINT
  <string>point_1 POINT <string>point_2
```

where

- The string `axis_name` is a name for this axis. This name must be unique to all other axes defined in the input file.
- The strings `point_1` and `point_2` are the names for two points defined in the input file via a `DEFINE POINT` command line.

The second `DEFINE AXIS` command line uses a point and a direction:

```
DEFINE AXIS <string>axis_name WITH POINT
  <string>point DIRECTION <string>direction
```

where

- The string `axis_name` is a name for this axis. This name must be unique to all other axes defined in the input file.
- The string `point` is the name of a point defined in the input file via a `DEFINE POINT` command line.
- The string `direction` is the name of a direction defined in the input file via a `DEFINE DIRECTION` command line.

2.1.8. Orientation

```
BEGIN ORIENTATION <string>orientation_name
  SYSTEM = <string>RECTANGULAR|Z_RECTANGULAR|CYLINDRICAL|
    SPHERICAL(RECTANGULAR)
  #
  POINT A = <real>global_ax <real>global_ay <real>global_az
  POINT B = <real>global_bx <real>global_by <real>global_bz
  #
  ROTATION ABOUT <integer> 1|2|3(2) = <real>theta(0.0)
END [ORIENTATION <string>orientation_name]
```

The `ORIENTATION` command block is currently used in Sierra/SM to define a local *co-rotational* coordinate system at each shell element centroid for output of shell in-plane stresses and strains. Each `BEGIN SHELL SECTION` command block has an orientation that generates this local coordinate system for each element in the associated element blocks. The generated local coordinate system rotates with the shell element, thus making it *co-rotational*.

Stresses and strains for shell elements are computed in the shell element's local coordinate system, which typically varies from one element to the next, and can be seen in Figure 2-3. The local **R** axis of element 1 does not align with the local **R** axis of element 2. Thus without the use of an orientation to rotate the local coordinate system in to a consistent coordinate system, stress and strain results would be difficult to decipher. The use of an orientation generates a transformation matrix from a shell element's local **R**, **S**, **T** coordinate system to the final coordinate system, $\mathbf{R}'$, $\mathbf{S}'$, $\mathbf{T}'$, by use of orientation and element-specific basis vectors and a rotation about an axis, all defined below.

There are four orientation systems available in Sierra/SM to better align the stress and strain output of shell elements. These orientations are defined via the optional `SYSTEM` command line within the `ORIENTATION` command block and are: `RECTANGULAR`, `Z_RECTANGULAR`, `CYLINDRICAL` and `SPHERICAL` with the default being `RECTANGULAR`. If the `ORIENTATION` command line within a `SHELL SECTION` block is not used, the default orientation is a `RECTANGULAR` system with axes aligned with the global X,Y and Z axes.

Each of the four systems produces a unique set of bases vectors, $\mathbf{g}_1, \mathbf{g}_2, \mathbf{g}_3$, computed using the centroid of the shell element and the two required inputs, `POINT A` and `POINT B`. These four methods of computing $\mathbf{g}_1, \mathbf{g}_2, \mathbf{g}_3$ are described below.

- `RECTANGULAR`: `POINT A` defines a point, P_A, in the local coordinate system of the shell

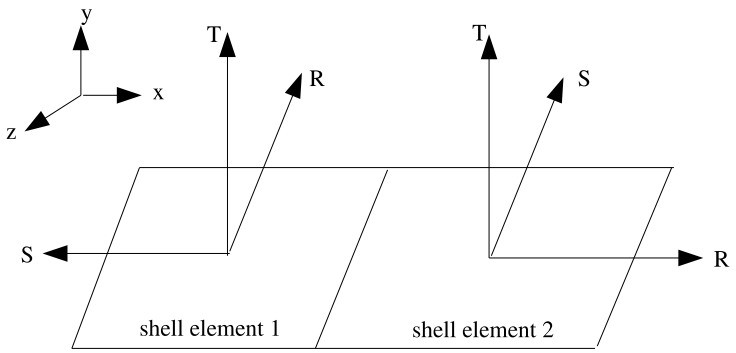


Figure 2-3. Adjacent shell elements with non-aligned local coordinate systems

element that lies in the direction of the first basis vector, $\mathbf{g}_1$, from the centroid, C with local coordinate of $(0,0,0)$, see Figure 2-4. Thus, the vector from the centroid of the element to point P_A is defined as $\mathbf{p}_A = P_A - C$. Similarly, a vector from the centroid to point P_B, defined via `POINT B`, can be computed as $\mathbf{p}_B = P_B - C$. Using these two vectors, the rectangular basis vectors are defined as: $\mathbf{g}_1 = \frac{\mathbf{p}_A}{\|\mathbf{p}_A\|}$, $\mathbf{g}_3 = \frac{\mathbf{g}_1 \times \mathbf{p}_B}{\|\mathbf{g}_1 \times \mathbf{p}_B\|}$, $\mathbf{g}_2 = \frac{\mathbf{g}_3 \times \mathbf{g}_1}{\|\mathbf{g}_3 \times \mathbf{g}_1\|}$.

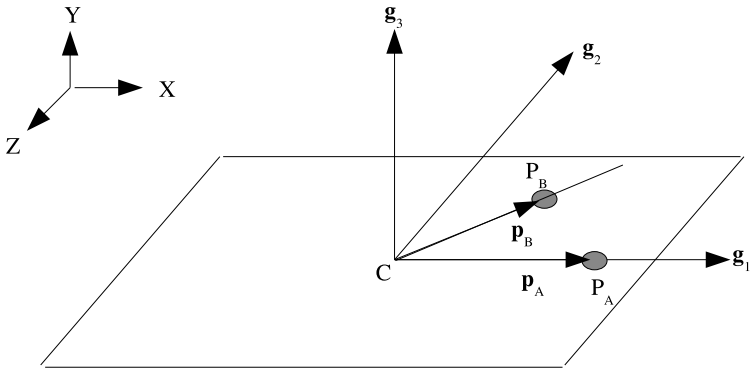


Figure 2-4. Rectangular coordinate system

- `Z_RECTANGULAR`: This set of basis vectors is defined similarly to the `RECTANGULAR` system, however point P_A now defines a point that lies in the direction of the third basis vector, $\mathbf{g}_3$ from the centroid of the element, see Figure 2-5. Using the same definition for $\mathbf{p}_A$ and $\mathbf{p}_B$ as above we can define the three basis vectors as: $\mathbf{g}_3 = \frac{\mathbf{p}_A}{\|\mathbf{p}_A\|}$, $\mathbf{g}_2 = \frac{\mathbf{g}_3 \times \mathbf{p}_B}{\|\mathbf{g}_3 \times \mathbf{p}_B\|}$, $\mathbf{g}_1 = \frac{\mathbf{g}_2 \times \mathbf{g}_3}{\|\mathbf{g}_2 \times \mathbf{g}_3\|}$.
- `CYLINDRICAL`: `POINT A`, P_A, and `POINT B`, P_B, define the direction of the third basis vector, $\mathbf{g}_3$ such that $\mathbf{g}_3 = \frac{P_B - P_A}{\|P_B - P_A\|}$, as seen in Figure 2-6. Defining a vector, $\mathbf{v}$, from P_A to the centroid of the element, C in the global coordinate system, the second basis vector, $\mathbf{g}_2$ is defined as: $\mathbf{g}_2 = \frac{\mathbf{v} \times \mathbf{g}_3}{\|\mathbf{v} \times \mathbf{g}_3\|}$. Therefore the first basis vector is defined as $\mathbf{g}_1 = \frac{\mathbf{g}_2 \times \mathbf{g}_3}{\|\mathbf{g}_2 \times \mathbf{g}_3\|}$.

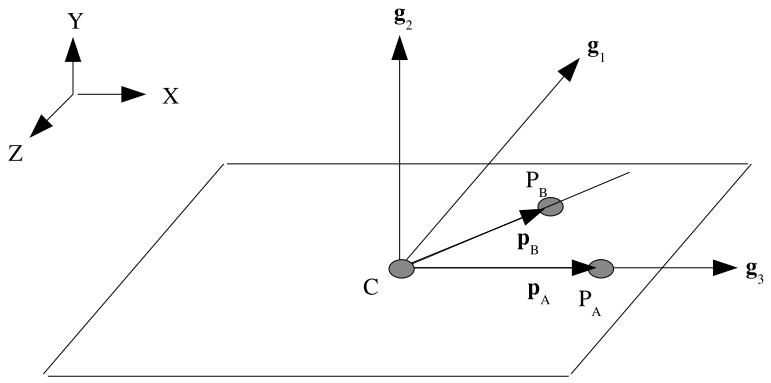


Figure 2-5. Z-Rectangular coordinate system.

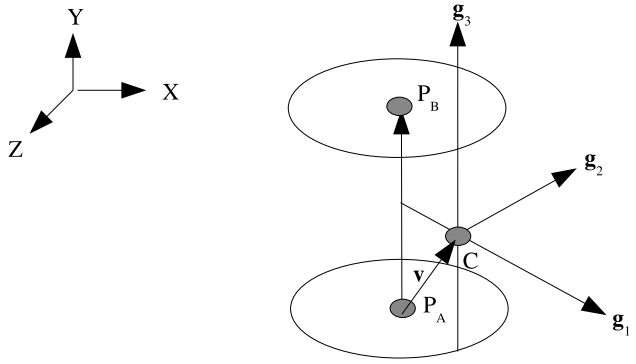


Figure 2-6. Cylindrical coordinate system.

- `SPHERICAL`: The point P_A, from the `POINT A` command line, defines the center of a sphere. The point P_B, from the `POINT B` command line, defines a polar axis for the sphere. (See Figure 2-7.) The first basis vector is defined by the vector that passes from P_A to the centroid, C in the global coordinate system, of the element: $\mathbf{g}_1 = \frac{C-P_A}{\|C-P_A\|}$. The third basis vector is parallel to the vector from P_A to P_B such that $\mathbf{g}_3 = \frac{P_B-P_A}{\|P_B-P_A\|}$. From these two vector, the second basis vector is defined as $\mathbf{g}_2 = \frac{\mathbf{g}_3\times\mathbf{g}_1}{\|\mathbf{g}_3\times\mathbf{g}_1\|}$. If $\mathbf{g}_1$ and $\mathbf{g}_3$ are collinear, the second basis vector is taken to be the normal of the element, $\mathbf{T}$, and the third basis is $\mathbf{g}_3 = \frac{\mathbf{g}_1\times\mathbf{g}_2}{\|\mathbf{g}_1\times\mathbf{g}_2\|}$.

It is possible, using the `ROTATION ABOUT` command line, to specify an angle and an axis to rotate the basis vectors, $\mathbf{g}_1$, $\mathbf{g}_2$ and $\mathbf{g}_3$ about to produce a second set of basis vectors, $\mathbf{g}'_1$, $\mathbf{g}'_2$ and $\mathbf{g}'_3$, that are then used to compute the final transformation matrix. The syntax for this command is as follows:

```
ROTATION ABOUT 1|2|3(2) = <real>theta(0.0)
```

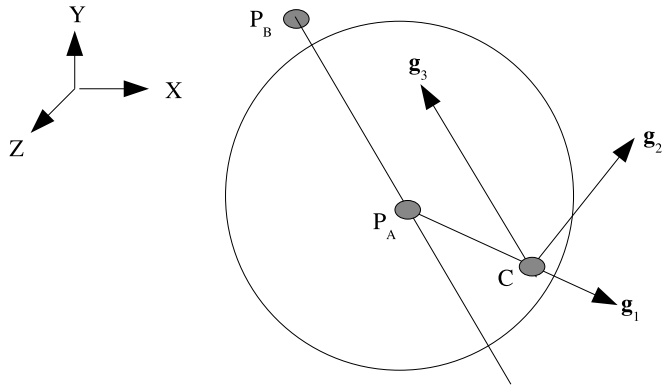


Figure 2-7. Spherical coordinate system.

Where the 1,2,3 refers to which basis vector the rotation will occur about. The default value is 2 with an angle theta of 0.0, thus implying a rotation of 0.0 radians about $\mathbf{g}_2$. How $\mathbf{g}'_1$, $\mathbf{g}'_2$ and $\mathbf{g}'_3$ are computed for each of the three cases is described below. In each case, the second order rotation tensor, $\mathbf{Q}$, is defined as:

$$\mathbf{Q} = \begin{bmatrix} 2(q_0^2+q_1^2)-1 & 2(q_1q_2-q_0q_3) & 2(q_1q_3-q_0q_3) \\ 2(q_1q_2+q_0q_3) & 2(q_o^2+q_2^2)-1 & 2(q_2q_3-q_0q_1) \\ 2(q_1q_3-q_0q_2) & 2(q_2q_3+q_0q_1) & 2(q_0^2+q_3^2)-1 \end{bmatrix}$$

Where: $q_0 = cos(\frac{\theta}{2})$, $q_1 = a_1 sin(\frac{\theta}{2})$, $q_2 = a_2 sin(\frac{\theta}{2})$, and $q_3 = a_3 sin(\frac{\theta}{2})$, where the vector $\mathbf{a} = [a_1, a_2, a_3]$ is the basis vector we are rotating about.

- Rotation about $\mathbf{g}_1$ (`ROTATION ABOUT 1`): The prime set of basis vectors are defined as: $\mathbf{g}'_1 = \mathbf{Q}\mathbf{g}_3$, $\mathbf{g}'_2 = \mathbf{Q}\mathbf{g}_2$, $\mathbf{g}'_3 = -\mathbf{g}_1$. Which can graphically be seen in Figure 2-8

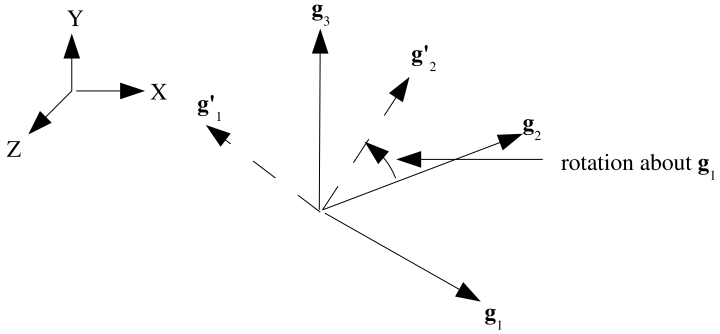


Figure 2-8. Rotation about 1

- Rotation about $\mathbf{g}_2$ (`ROTATION ABOUT 2`), default: The prime set of basis vectors are defined as: $\mathbf{g}'_1 = \mathbf{Q}\mathbf{g}_1$, $\mathbf{g}'_2 = \mathbf{Q}\mathbf{g}_3$, $\mathbf{g}'_3 = -\mathbf{g}_2$.
- Rotation about $\mathbf{g}_3$ (`ROTATION ABOUT 3`): The prime set of basis vectors are defined as: $\mathbf{g}'_1 = \mathbf{Q}\mathbf{g}_2$, $\mathbf{g}'_2 = \mathbf{Q}\mathbf{g}_1$, $\mathbf{g}'_3 = -\mathbf{g}_3$.

The next step is to project $\mathbf{g}'_1$ onto the shell face as follows: $\mathbf{R}' = \mathbf{g}'_1 - (\mathbf{g}'_1 \cdot \mathbf{T})\mathbf{T}$, which defines a rotated $\mathbf{R}$ vector. If $\mathbf{R}'$ is perpendicular to the face of the element, we then redefine $\mathbf{R}'$ to be: $\mathbf{R}' = \mathbf{g}'_2 - (\mathbf{g}'_2 \cdot \mathbf{T})\mathbf{T}$, the projection of $\mathbf{g}'_2$ onto the element face. Once we have $\mathbf{R}'$ we can define $\mathbf{S}'$ as $\frac{\mathbf{T}\times\mathbf{R}'}{\|\mathbf{T}\times\mathbf{R}'\|}$. Hence, the final transformation matrix taking local stress and strain (in the $\mathbf{R}$, $\mathbf{S}$ coordinate system) to the $\mathbf{R}'$, $\mathbf{S}'$ coordinate system is:

$$\mathbf{Q}' = \begin{bmatrix} \mathbf{R}\cdot\mathbf{R}' & \mathbf{R}\cdot\mathbf{S}' & 0 \\ \mathbf{S}\cdot\mathbf{R}' & \mathbf{S}\cdot\mathbf{S}' & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2.1.9. Coordinate System Block Command

The `RECTANGULAR|CYLINDRICAL|SPHERICAL|CONICAL|ELLIPSOIDAL COORDINATE SYSTEM` command block is used to define a local coordinate system for transforming a stress tensor from components in the global x, y, z coordinate system to the local system for output.

This command block cannot be used for in-plane stresses and strains of shell elements because the shell integration point stresses and strains are computed in a local element system that varies element to element and rotates with the element. For output of shell stresses, the `BEGIN ORIENTATION` command must be used as explained in Section 2.1.8.

The `COORDINATE SYSTEM` block is also used to define a local coordinate system on the elements of a block for use in a representative volume analysis. In these analyses, elements of each representative volume will be aligned with the local system defined on the parent element.

```
BEGIN RECTANGULAR COORDINATE SYSTEM <string>name
  ORIGIN = <real> <real> <real>
  Z POINT = <real> <real> <real>  # Point on ~Z axis
  XZ POINT = <real> <real> <real> # Point on ~X ~Z plane
  ORIGIN NODESET  = <nodelist>    # Single node nodeset
  Z POINT NODESET  = <nodelist>
  XZ POINT NODESET = <nodelist>
END

BEGIN CYLINDRICAL COORDINATE SYSTEM <string>name
  #... see inputs for RECTANGULAR COORDINATE SYSTEM
END

BEGIN SPHERICAL COORDINATE SYSTEM <string>name
  #... see inputs for RECTANGULAR COORDINATE SYSTEM
END
```

```
BEGIN CONICAL COORDINATE SYSTEM <string>name
  #... see inputs for RECTANGULAR COORDINATE SYSTEM
  ANGLE = <real>
END

BEGIN ELLIPSOIDAL COORDINATE SYSTEM <string>name
  #... see inputs for RECTANGULAR COORDINATE SYSTEM
  AXIS STRETCHING = <real> <real> <real> #angle in radians
END
```

The prefix to `COORDINATE SYSTEM` specifies the type of local coordinate system. Supported types are: `RECTANGULAR` (Cartesian), `CYLINDRICAL` (Polar), `SPHERICAL`, `CONICAL` (a cylindrical system with an aperture angle), and `ELLIPSOIDAL` (a spherical system stretched by `AXIS STRETCHING`).

A local element coordinate system is defined by a set of three control points. These points may be defined using absolute locations (`ORIGIN`, `Z POINT`, `XZ POINT`) or using nodelists in the mesh file (`ORIGIN NODESET`, `Z POINT NODESET`, `XZ POINT NODESET`) that contain exactly one node each. The control points define a local $\tilde{X}\tilde{Y}\tilde{Z}$ Cartesian coordinate system centered at the coordinate system origin. For cylindrical, spherical, conical, and ellipsoidal systems the orientation of the basis vectors at individual nodes and elements is spatially dependent. In this documetation, the local basis vectors at each node or element will be denoted as R, S, and T.

NOTE: When using nodesets instead of points, the defined coordinate system will be **dynamic** and will translate and rotate.

- `ORIGIN`: Required. Specifies the origin location for the new coordinate system. Replace with `ORIGIN NODESET` for a dynamic system.
- `Z POINT`: Required. Specifies a point on the $\tilde{Z}$ axis for the new coordinate system. Replace with `Z POINT NODESET` for a dynamic system. For `SPHERICAL`, this defines where the zenith (polar) angle ϕ is 0.
- `XZ POINT`: Required only for `RECTANGULAR` and `ELLIPSOIDAL`. Third point to specify the $\tilde{X}\tilde{Z}$-plane of the new system. Must not lie on the $\tilde{Z}$ axis. Replace with `XZ POINT NODESET` for a dynamic system. For `CYLINDRICAL` and `SPHERICAL`, this defines where the azimuthal angle θ is 0. `XZ POINT` also affects behavior at ill-defined points such as the origin of `SPHERICAL` systems, or along the $\tilde{Z}$ axis of `CYLINDRICAL` systems (see figure 2-13).
- `ANGLE` (`CONICAL COORDINATE SYSTEM` only): Required. Defines the angle in radians to rotate the axial direction away from the $\tilde{Z}$ axis (see figure 2-11).
- `AXIS STRETCHING` (`ELLIPSOIDAL COORDINATE SYSTEM` only): Required. Defines the aspect ratio to stretch a spherical system (see figure 2-12).

The three coordinate control points are illustrated in figure 2-9. The coordinate system types are depicted in figures 2-10 to 2-12.

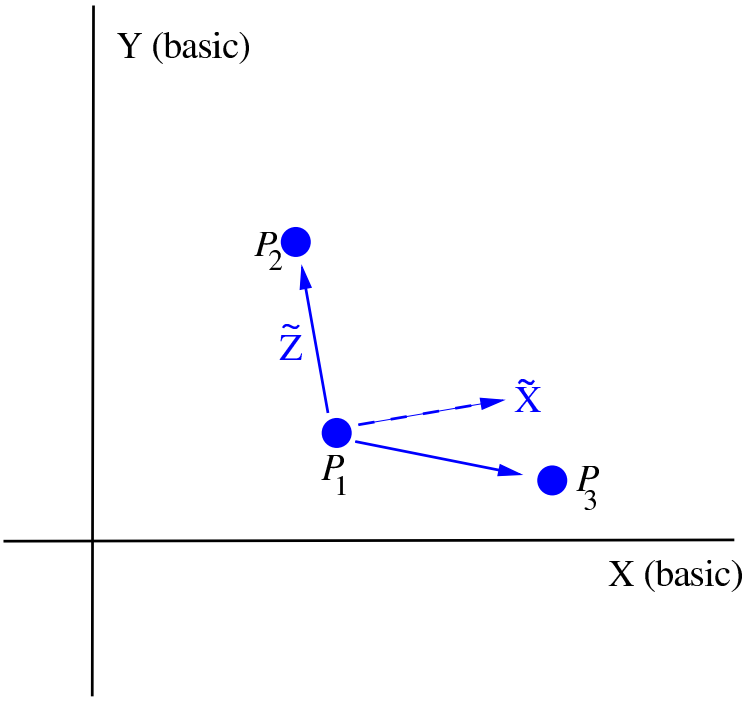


Figure 2-9. Coordinate System Definition Vectors

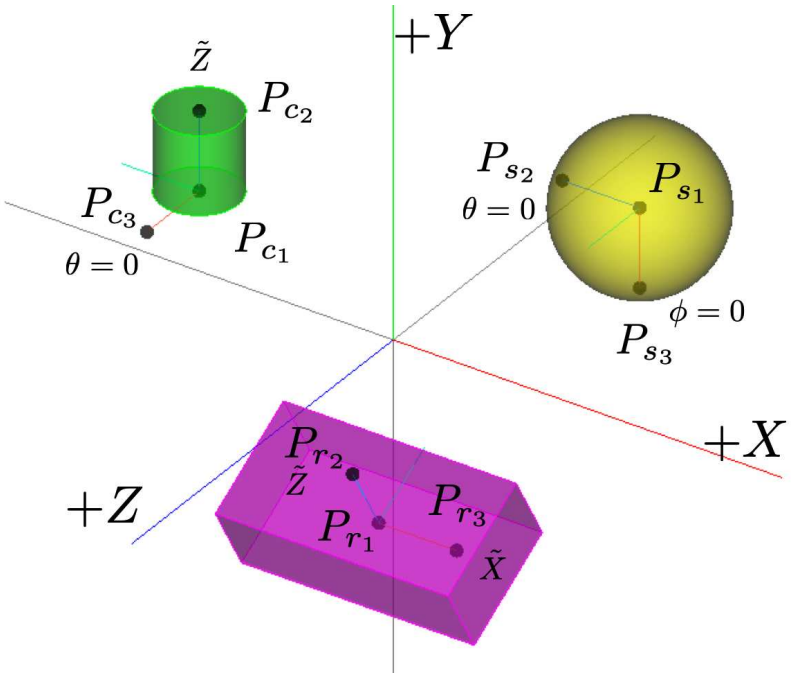


Figure 2-10. Three examples (rectangular, cylindrical, and spherical) of transformed coordinate systems are given. The center of the rectangular block $(6\times2\times3)$ is located at $(3,-1,5)$ and rotated 30 degrees about the X axis. The center of the cylinder (radius of 1, height of 2, and rotated 90 degrees about the X axis) is located at $(-5,3,0)$. The center of the sphere (radius of 2) is located at $(3,-1,5)$. The new coordinate systems are defined respectively of each of the geometries and are indicated by $\tilde{()}$ and the subscripts (r, c, and s).

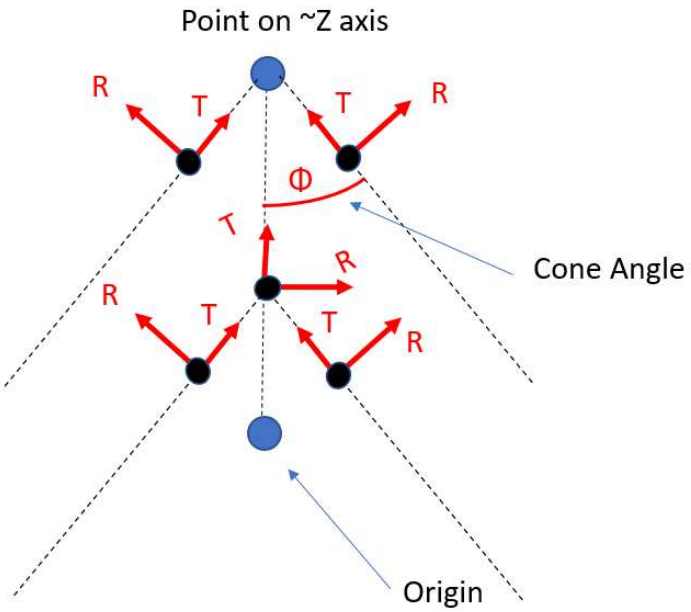


Figure 2-11. Conical Coordinate System Definition at $\tilde{X}\tilde{Z}$ plane

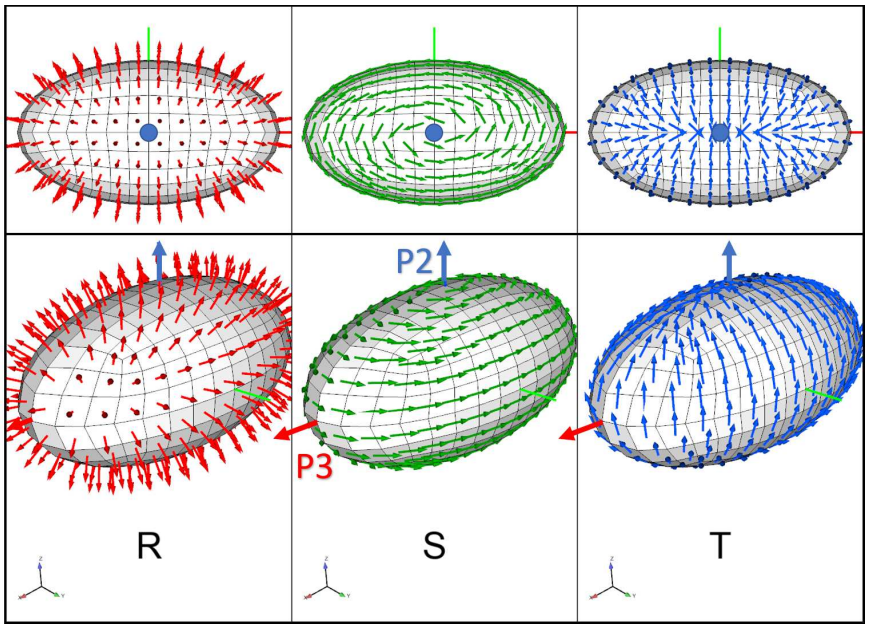


Figure 2-12. Ellipsoidal Coordinate System
Using axis_stretching=(2.0, 1.1, 1.0)

Examples of rectangular, cylindrical, and spherical coordinate systems are given in figure 2-10. In those examples, we wish to define coordinate systems for three geometries: a brick, a cylinder, and a sphere. The corresponding input deck is shown below.

```
#Example coordinate system input
begin rectangular coordinate system rc
  origin   3 -1 5
  z point  5 -1 5
  xz point 3 0.7321 6
end
begin cylindrical coordinate pinSystem
  origin -5 2 0
  z point -5 4 0
  xz point -5 2 2
end
coordinate spherical coordinate system ballLike
  origin 5 4 -2
  z point 3 4 -2
  xz point 5 2 -2
end
```

The cylindrical, conical, spherical, and ellipsoidal systems are ill-defined when sampled along the $\tilde{Z}$ axis. Doubly so for the spherical and ellipsoidal systems at the origin. At the origin, all coordinate systems fall back to a local Cartesian system defined by $\tilde{X}$, $\tilde{Y}$, and $\tilde{Z}$. Along the $\tilde{Z}$ axis, the cylindrical and conical systems also fall back to the Cartesian definition. Spherical and ellipsoidal systems on the $\tilde{Z}$ axis will instead maintain the first vector radially outward, the third vector along the $\tilde{X}$ axis, and the second vector as either the positive or negative $\tilde{Y}$ axis (generating a right handed system). This maintains the normal and tangential properties of the three vectors for spherical coordinates. See figure 2-13 for a visual representation of this behavior.

2.1.10. Define Coordinate System Line Command

```
DEFINE COORDINATE SYSTEM <string>coord_sys_name
  <string>coord_sys_type WITH POINT <string>point_1
  POINT <string>point_2 POINT <string>point_3
```

The line command `DEFINE COORDINATE SYSTEM` can also be used to define the axis directions of a local coordinate system to be located at nodes, element centroids, or element integration points. In this command

- The string `coord_sys_name` is a name for this coordinate system. This name must be unique to all other coordinate systems defined in the input file.
- The string `coord_sys_type` states the type of the coordinate system to be used. Three options are allowed for constructing a local coordinate system: `RECTANGULAR`, `CYLINDRICAL`, and `SPHERICAL`.
- The strings `point_1`, `point_2`, and `point_3` are the names for three points defined in the input file via `DEFINE POINT` command lines. These three points are used to define two of the coordinate system axes as described below for the different types of systems.

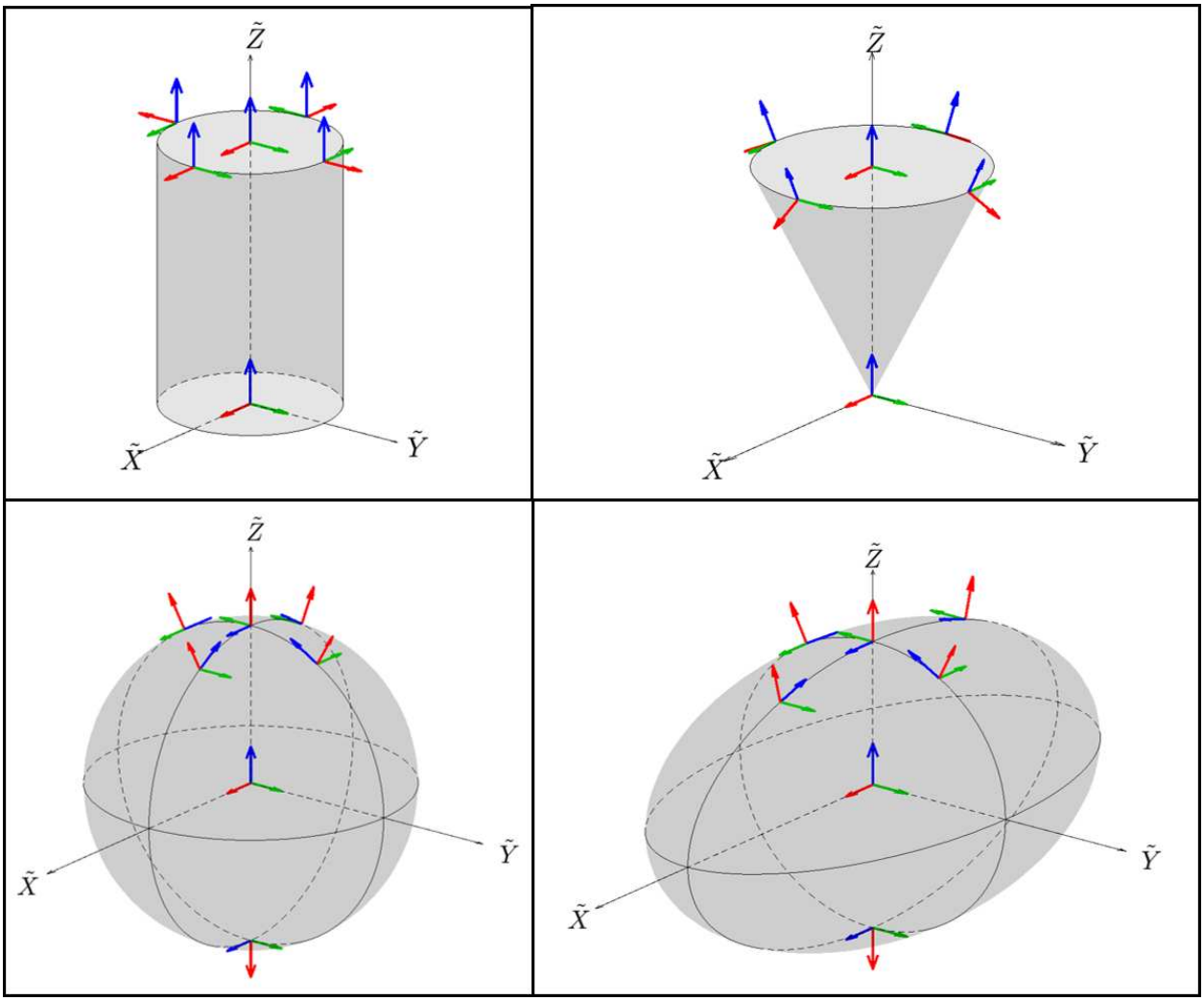


Figure 2-13. Coordinate system behavior near the $\tilde{Z}$ axis. $\hat{R}$ is red, $\hat{S}$ is green, and $\hat{T}$ is blue.

For a rectangular system, the local Z' axis is parallel to the vector from the `point_1` to `point_2`. The local X' axis is the component of the vector from `point_1` to `point_3` that is orthogonal to the Z' axis. Finally, the local Y' axis is obtained from the cross product of Z' and X'.

Likewise, for a cylindrical system, the local Z' axis is parallel to the vector from the `point_1` to `point_2` and defines the axis of the cylinder. The local X' axis is constructed as the vector normal to the cylindrical axis and passing through the location at which the local system is desired. This local point may be a node, an element centroid, or an element integration point. If this local point lies on the Z' axis, then the point `point_3` is instead used to define the X' axis. Finally, the Y' axis is obtained from the cross product of Z' and X'. Thus at the desired point not on the cylinder axis, the X' axis is through the cylinder thickness, Y' is tangent to the cylinder, and Z' is parallel to the cylinder axis.

For a spherical system, `point_1` specifies the center of a sphere. The local X' axis is constructed

as the vector from `point_1` through the location at which the local system is desired. This local point may be a node, an element centroid, or an element integration point. The local Z' axis is the component of the vector from the `point_1` to `point_2` that is normal to the X' axis. If this Z' is parallel the X' axis, then `point_3` is used instead. Finally, the Y' axis is obtained from the cross product of Z' and X'. Thus at a desired point, the X' axis is through the sphere thickness and the Y' and Z' axes lie in the tangent plane.

2.2. Procedure and Region

Explicit dynamics analyses use a Presto procedure and region. The Presto procedure scope is nested within the SIERRA scope, and the Presto region scope is nested within the procedure scope (see Section 1.3 for more information about scope). To create the scopes for the Presto procedure and Presto region for an explicit dynamics analysis, use the following commands:

```
BEGIN PRESTO PROCEDURE <string>procedure_name

  PRINT BANNER INTERVAL = <integer>print_banner_interval(MAX_INT)
  STOP AFTER INITIAL WRITE

  #
  # TIME CONTROL command block
  #
  BEGIN PRESTO REGION <string>region_name
    #
    # command blocks and command lines that appear in the
    # region scope
    #
  END [PRESTO REGION <string>region_name]
END [PRESTO PROCEDURE <string>procedure_name]
```

The `PRINT BANNER INTERVAL` line command sets the number of output lines to print before re-printing the column headings. The default value is set to the maximum allowed integer value. This means that the banner will be printed to the log file one time by default.

This command will be ignored in a solution control analysis that contains a presto region. For that case use the `PRINT BANNER INTERVAL FOR SOLUTION CONTROL` line command within a `TIME CONTROL` command block. See Section 3.1.6 for use of the `PRINT BANNER INTERVAL FOR SOLUTION CONTROL` line command.

The `STOP AFTER INITIAL WRITE` line command ends the procedure after initialization. This tool can be useful in an analysis with restart.

Implicit analyses follow a similar structure, but use an Adagio procedure and region. The following example shows the structure of an input file for an implicit analysis:

```
BEGIN ADAGIO PROCEDURE <string>procedure_name
  #
  # TIME CONTROL command block
  #
  BEGIN ADAGIO REGION <string>region_name
    #
    # command blocks and command lines that appear in the
    # region scope
    #
```

```
  END [ADAGIO REGION <string>region_name]
END [ADAGIO PROCEDURE <string>region_name]
```

For both implicit and explicit analyses, the `TIME CONTROL` command block appears within the `PRESTO PROCEDURE` or `ADAGIO PROCEDURE` command block but outside of the `PRESTO REGION` or `ADAGIO REGION` command block. These three command blocks (procedure, time control, and region) are discussed below.

Many command blocks and command lines fall within the region scope. These command blocks and command lines are described in other sections of this document.

2.2.1. Procedure

The analysis time, from the initial time to the termination time, is controlled within the procedure scope defined by the `PRESTO PROCEDURE` or `ADAGIO PROCEDURE` command block. For explicit dynamic analyses, the command block begins with an input line of the form:




```
BEGIN PRESTO PROCEDURE <string>procedure_name
```

and is terminated with an input line of the following form:

```
END [PRESTO PROCEDURE <string>procedure_name]
```

Implicit analyses use an Adagio procedure, which starts with the following line:




```
BEGIN ADAGIO PROCEDURE <string>procedure_name
```

and ends with the following line:

```
END [ADAGIO PROCEDURE <string>procedure_name]
```

The string `procedure_name` is the name for the procedure.

2.2.2. Time Control

Within the procedure scope, there is a `TIME CONTROL` command block. This command block lets the user set the initial time and the termination time for an analysis. This block also allows the user to control the size of the time step. The time step is controlled in different ways for explicit and implicit analyses, as explained below.

2.2.2.1. Explicit Dynamic Time Control

When Sierra/SM is used to run explicit transient dynamics analyses, it must run at a time step that is less than the critical time for the problem at any given instant. Typically, this global critical time step is based on a critical time step estimate calculated for each element. With the `TIME CONTROL` command block, the user can set an initial time step, scale the element-based time step estimate, and control the growth of the element-based estimate for the critical time step.

In addition to the element-based method for estimating the critical time step, Sierra/SM offers other methods for estimating the critical time step. One approach for estimating the critical time step is to calculate the maximum eigenvalue for the model. Two methods are implemented for calculating the maximum eigenvalue: the Lanczos method and the power method. A second approach for estimating the critical time step is to use a node-based method. The command blocks for implementing these various methods (maximum eigenvalue calculation and node-based) are in the region scope. There is also a mass-scaling technique that will influence the magnitude of the critical time step. If you use the mass-scaling technique, you must use the node-based method to obtain a critical time step estimate.

The estimation of the time step is a key part of any explicit Sierra/SM analysis. Time step determination and control is discussed in detail in Chapter 3 of this document. The `TIME CONTROL` command block with its associated command lines are described in detail in Chapter 3. Consult Chapter 3 to determine how to specify command lines associated with the `TIME CONTROL` command block and how the `TIME CONTROL` command block fits into the overall scheme for time step control in explicit transient dynamics. Also see Chapter 3 to learn about the other methods for estimating the critical time step and the mass-scaling technique.

2.2.2.2. Implicit Time Control

When running implicit dynamic or quasistatic analyses with Sierra/SM a solver is used to find a converged solution at every time step. The time steps can be arbitrarily large, although errors due to time discretization are larger with larger time steps. In addition, for nonlinear problems, it is often more difficult to obtain a converged solution with large time steps. The time step size is controlled by the user, although Sierra/SM has a capability to optionally modify the time step based on the effort required by the nonlinear solver.

The `TIME CONTROL` block contains the basic commands used to control the time step size. To use adaptive time stepping, an `ADAPTIVE TIME STEPPING` command block is placed in the Adagio region scope. In addition, for all implicit analyses, commands to control the solver are required in the Adagio region scope. The details of the commands related to time stepping and the nonlinear solver are documented in Chapter 4.

2.2.3. Region

Individual time steps are controlled within the region scope. For explicit analyses, the region scope is defined by a `PRESTO REGION` command block that begins with an input line of the form




```
BEGIN PRESTO REGION <string>region_name
```

and is terminated with an input line of the following form:

```
END [PRESTO REGION <string>region_name]
```

Implicit analyses define the region scope using a `ADAGIO REGION` command block that begins with the following line:




```
BEGIN ADAGIO REGION <string>region_name
```

and is terminated by the following line:

```
END [ADAGIO REGION <string>region_name]
```

The string `region_name` is the name for this region.

The region, as indicated previously, determines what happens at each time step. In the procedure, we set the begin time and end time for the analysis. Time is incremented in the region. It is in the region where we set information about what occurs at various time steps. The output of results, for example, is set by command blocks in the region. If we want results output at certain times or certain steps in the analysis, this information is set in command blocks in the region. The region also contains command blocks for the boundary conditions. A boundary condition can have a time-varying component. The region determines the value of the component for the current time step.

Two of the major types of command blocks, those for results output and boundary conditions, have already been mentioned. Other major types of command blocks in the region are those for restart control and contact. The region is also where the user selects the analysis model (finite element mesh). For implicit analyses, the command blocks for the solver are in the region. These blocks are discussed in Chapter 4.

The region makes use of information in the procedure and the SIERRA scope. For example, the specific element type used for an element block in the analysis model is defined in the SIERRA scope. This information about the element type is collected into an analysis model. The region then references this analysis model. As another example, the boundary condition command blocks can reference a function. The function will be defined in the SIERRA scope.

2.3. Use Finite Element Model

```
USE FINITE ELEMENT MODEL <string>model_name
```

The model specification occurs within the region scope. To specify the model (finite element mesh), use this command line. The string `model_name` must match a name used in a `FINITE ELEMENT MODEL` command block described in Section 6.1. If one of these command blocks uses the name `penetrator` in the command-block line and this is the model we wish to use in the region scope, then we would enter the command line as follows:

```
USE FINITE ELEMENT MODEL penetrator
```

2.4. Mesh Quality

```
BEGIN MESH QUALITY
  CHECK MESH TOPOLOGY = <string>ON|OFF(ON)
END [MESH QUALITY]
```

The BEGIN MESH QUALITY command block occurs within the region scope. This block will eventually offer a variety of options for diagnosing the quality and validity of the input mesh. Currently, the only option is CHECK MESH TOPOLOGY.

The CHECK MESH TOPOLOGY option checks for valid solid element connectivity and topology. In particular, it checks if any 3D bulk element is repeated twice in the mesh and if there are multiple elements which are (un-physically) attached to the same side of one face.

This check is on by default, any errors found trigger a fatal error and abort the run. The check can be manually turned off by included the command block with the OFF option.

2.5. Element Distortion Metrics

Sierra/SM can compute many element distortion metrics useful for assessing the quality of the solution as the mesh evolves over time. These metrics are computed as element variables, and can be used for output or as criteria for element death, just like any other element variable. The solution quality generally deteriorates as elements approach inversion, and if they do invert, the analysis aborts. The distortion metrics measure how close an element is to inversion. For all metrics a value of 1.0 is an ideal element and a value of 0.0 is a degenerate element. The following distortion metrics are available:

- `NODAL_JACOBIAN_RATIO` is currently available only for 8 node hexahedra and four node quads. This metric evaluates the Jacobian function at each node of each element, and then computes the nodal Jacobian ratio as the smallest nodal Jacobian divided by the largest nodal Jacobian. An element shaped like a parallelepiped with equal angles between all adjacent edges has a nodal Jacobian ratio of 1.0. A negative nodal Jacobian ratio indicates that the element is becoming either concave or locally inverted. See Figure 2-14 for examples of quadrilateral elements with positive, zero, and negative nodal Jacobian ratios. The element calculations on poorly shaped elements (those with negative nodal Jacobian ratios) will generally be less accurate than those on well shaped elements (those with positive nodal Jacobian ratios). In addition, contact surfaces may become tangled and non-physical if elements start to invert.

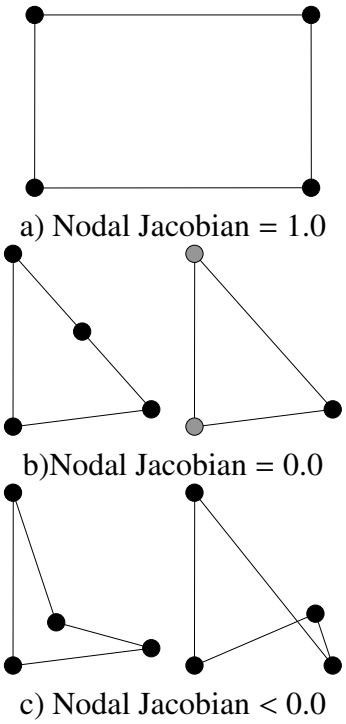


Figure 2-14. Examples of elements with varying nodal Jacobians

- `SCALED_JACOBIAN` is currently available only for 8 node hexahedra. This is the minimum value of the nodal jacobian normalized by the element volume. An element with right angle

edges will have a scaled jacobian of 1.0. A value of 0.0 indicates concavity or inversion. The SCALED_JACOBIAN is equivalent to the 'scaled jacobian' metric reported by cubit.

- ELEMENT_SHAPE computes a general shape metric for a variety of element topologies. Is currently defined for 8 node hexahedra, 4 node tetrahedra, 4 node quadrilaterals, and 3 node triangles. A value of 1.0 is optimal. A value of 0.0 represents a degenerate element. A negative value represents either a severely warped element or inverted element. For the 8 node hexahedron, the element shape is a scaled Jacobian equivalent to the 'shape' metric reported by cubit. For a tetrahedron or triangle, the shape is proportional to the element aspect ratio. For the 4 node quad element, the shape is proportional to the amount of warping of the quad. If a mesh contains elements with negative ELEMENT_SHAPE, problems could arise in the material or contact computations.
- ASPECT_RATIO is available only for tetrahedral elements. A perfect equilateral tetrahedron has an aspect ratio of 1.0. A degenerate zero-volume tetrahedron has an aspect ratio of zero. An inverted tetrahedron has a negative aspect ratio. A thin element can have a extremely small aspect ratio.
- INRADIUS For all tets but 10-node tetrahedrons, the radius of the smallest, fully contained sphere of the linear tet. For 10-node tetrahedrons, the mid-edge nodes are used to subdivide the tet into 12 linear sub-tets. The inradius is the smallest inradius of the 12 linear sub-tets multiplied by 2.3
- NORMALIZED_INRADIUS is available only for 10-node tetrahedron elements. This metric provides a measure of the deviation from an ideal equilateral tetrahedron and accounts for distortions which stem from both corner and mid-edge nodes. Inradii are calculated for the 12 sub-tetrahedra comprising the composite tetrahedron and normalized by the outer radius of the four nodes forming the parent, 4-node linear tetrahedron. Normalization factors scale the metric appropriately where values less than zero indicate an degenerate sub-tetrahedron and a value of 1.0 corresponds to a equilateral tetrahedron. The normalized_inradius is the minimum of the 12 normalized inradii.
- SOLID_ANGLE computes the minimum or maximum angle between edges of an element as compared to optimal angles. The optimal solid angle for tetrahedra and triangles is 60 degrees; for hexahedra and quadrilaterals, it is 90 degrees. This error metric is 1.0 for an element in which all angles are optimal. Severely distorted or twisted elements have values of this metric near 0.0, and it is negative for inverted elements. This metric is 0 for degenerate elements. The SOLID ANGLE metric functions with hexahedron, tetrahedron, triangle, and quadrilateral elements. Note that 10-node tetrahedral elements compute the solid angle using only the four vertices, so highly distorted mid-side nodes will not be represented with the solid_angle distortion metric.
- PERIMETER_RATIO measures the ratio of the deformed perimeter of an element to the undeformed perimeter of the element. This metric initially has a value of 1.0, and assumes values either greater than or lower than 1 as the mesh deforms. The PERIMETER_RATIO metric only works with triangle and quad elements.
- DIAGONAL_RATIO measures the ratio of the deformed maximum diagonal of an element to the undeformed maximum diagonal of the element. This metric initially has a value of 1.0, and assumes values either greater than or lower than 1 as the mesh deforms. The DIAGONAL_RATIO metric only works with hex and quad elements.

The results from any of these distortion metrics can be requested by specifying an `ELEMENT VARIABLES` command line (Section 9.3.1.4) in the `RESULTS OUTPUT` command block (Section 9.3.1) for each metric for which the results are of interest. For example, to request the `ASPECT_RATIO` metric to be output as an element variable named `aspect` in the results output file, the following syntax can be included in a `RESULTS OUTPUT` command block:

```
ELEMENT ASPECT_RATIO as aspect
```

The distortion metrics can also be used for element death (Section 6.5). For example, to kill all elements in which the value of the `SOLID_ANGLE` metric is less than 0.2, the following line can be used in an `ELEMENT DEATH` command block:

```
CRITERION IS ELEMENT VALUE OF SOLID_ANGLE < 0.2
```

For more information about the use of element variables for element death, see Section 6.5.2.2.

2.6. Activation/Deactivation of Functionality

```
ACTIVE PERIODS   = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

The ACTIVE PERIODS or INACTIVE PERIODS command line can be used to activate or deactivate functionality in the code at various points during an analysis. This functionality can include boundary conditions, element blocks, user subroutines, and contact definitions. Command blocks that support this capability are documented in the sections of this manual where they are described.

In the command line, the string list period_names is a list of the time periods defined in TIME STEPPING BLOCK command blocks (see Sections 3.1 and 4.10) during which the particular functionality is considered to be active. Each such period_name must match a name used in a TIME STEPPING BLOCK command block, e.g., time_block_name. Each defined time period runs from that period's start time to the next period's start time.

Only one of ACTIVE PERIODS or INACTIVE PERIODS can be used in any given command block. If the ACTIVE PERIODS command line is present, the functionality will be treated as active for all the named periods, and inactive for any time periods that are not listed. If the INACTIVE PERIODS command line is present, the functionality will be treated as inactive for all the named periods and active for any period not listed. If neither of these command lines is present, by default the functionality is active during all time periods.

2.7. Error Recovery

```
ERROR TOLERANCE = ON|OFF (ON)
```

This command controls how aggressively Sierra/SM will attempt to recover from a non-physical or ill-defined condition and continue with the analysis.

Under the default setting of `ON`, the code may invoke methods to keep the analysis moving forward despite the presence of non-physical or ambiguous conditions. For example, if the contact algorithm determines that a facet is potentially in two contact surfaces (an ambiguous condition), Sierra/SM will assign the facet to one of the surfaces and continue with the analysis. A second example pertains to SPH particles: if the deformation gradient of an SPH particle is inverted, the particle will be ignored rather than causing the analysis to terminate. Non-physical and ambiguous conditions will generally result in a warning message being printed.

If `ERROR TOLERANCE` is `OFF`, Sierra/SM will terminate an analysis in the presence of ambiguous or non-physical conditions. This may make it easier for an analyst to address non-fatal but potentially problematic issues.

2.8. Manual Job Control

The output of a running job can be controlled externally through the use of "shutdown" and "control" files. This mechanism allows for additional output of restart, results, history, and/or heartbeat data to be requested, as well as an optional graceful shutdown of the job.

A graceful shutdown is requested by inserting a "shutdown" file in the working directory of a running Sierra/SM job. The name of this file can be any of the following:

```
sierra.shutdown
adagio.shutdown
<base_name>.shutdown
```

If the `<base_name>.shutdown` variant is used, `<base_name>` is the base name of the input file. For example, if the input file name is `my_analysis.i`, then the shutdown file would be `my_analysis.shutdown`.

If the code detects the existence of a shutdown file, it will dump an output step to any open restart, results, history, or heartbeat file and then gracefully terminate the job. An entry will be written to the log file specifying that a shutdown file was detected and the file will be deleted. The contents of the file are not important; the shutdown file is only checked for existence. Note, the shutdown file will only be looked for at the end of a completed time step. Thus if the solve for a time step is taking an extremely long time the shutdown file will not be detected until that solve completes.

The control file capability is provided to give more control over output and execution than is possible with the shutdown file. The name of the control file is the same as that of the shutdown file except that the file name suffix is `.control` instead of `.shutdown`. The contents of the file consist of a single line, and are case insensitive. The syntax is:

```
DUMP [RESTART] [RESULTS] [HISTORY] [HEARTBEAT] [STEP] [TIME]
  [STOP|ABORT|SHUTDOWN|CONTINUE]
```

The optional strings `RESTART`, `RESULTS`, `HISTORY`, `HEARTBEAT`, `STEP`, and `TIME` are used to specify the type of output that is written before an action is taken. If any output blocks of the specified type were defined in the input file for the model, output will be written to the files. The `STEP` and `TIME` options result in the last step and time being written to the log file. Multiple output types may be requested. If no output type is requested, all types of output specified in the input file will be written.

In addition to controlling the type of output that occurs, the `.control` file also specifies whether the job should be terminated or allowed to continue. If `STOP`, `ABORT`, or `SHUTDOWN` is specified at the end of the line, the job will be gracefully terminated. If `CONTINUE` is specified or no option is specified, the job will continue.

Several examples of the contents of continue files are shown below. The following examples all result in output being written to all types of output file and the job continuing:

```
dump
```

```
dump continue
dump restart results history heartbeat step continue
```

In both of the following examples, the current step and time will be written to the log file, but no additional output will be written and the job will continue:

```
dump step
dump time continue
```

This example would result in a write to all output types and a graceful shutdown:

```
dump stop
```

In the following example, no output would be written to any files, but the current step and time would be written to the log file before a graceful shutdown:

```
dump step abort
```

An alternate abbreviated syntax is also supported. The abbreviated commands are shown below along with the full commands to which they are mapped:

```
sw1          dump restart shutdown
sw2          dump step continue
sw3          dump restart continue
sw4          dump results continue
```

If either the shutdown or control files are used, a message is output to the log file listing the name of that file, and for the control file, the contents of that file. The control or shutdown file is then deleted.

2.9. Solution Termination

Solution termination is the capability to stop the execution when a computed solution reaches some reference solution. This capability can be especially useful for users who want to terminate an analysis when a variable reaches a specific value, but they do not know a priori when this will occur. Solution termination may also be used in multi procedure or multi time period analysis to move on to the next solution procedure once some criteria is reached. The inputs are similar to those for solution verification (see Section 9.11).

```
BEGIN SOLUTION TERMINATION
  TERMINATE GLOBAL <string>gVar <|<=|=|>|>=
    <real>gVal [PLUS OR MINUS <real>Tol]
  TERMINATE GLOBAL <string>gVar <|<=|=|>|>=
    GLOBAL <string>gVar2 [PLUS OR MINUS <real>Tol]
  TERMINATE GLOBAL <string>gVar <|<=|=|>|>=
    FUNCTION <string>funcName
  TERMINATE NODAL <string>nVar
    AT NODE <integer>nodeNum <|<=|=|>|>=
    FUNCTION <string>funcName
  TERMINATE ELEMENT <string>eVar
    AT ELEMENT <integer>elemNum <|<=|=|>|>=
    <real>val

  TOLERANCE = <real>absTol
  RELATIVE TOLERANCE = <real>relTol
  TIME TOLERANCE = <real>timeTol

  SKIP TIMES = <real>time0 TO <real>time1
  ACTIVE PERIODS = <string list>periodNames
  INACTIVE PERIODS = <string list>periodNames

  COMPLETION FILE = <string>fileName

  TERMINATE TYPE = ENTIRE_RUN|CURRENT_PROCEDURE|
    CURRENT_TIME_PERIOD(ENTIRE_RUN)

  # write results, history, and restart files
  WRITE OUTPUT = OFF|ON(ON)
END
```

The various forms of the `TERMINATE GLOBAL`, `TERMINATE NODAL`, and `TERMINATE ELEMENT` commands check that the named variable has achieved some relation to some defined constant, time history function, or other global variable. Currently, the most thoroughly tested condition is for termination based on a global variable, such as kinetic energy, reaching a constant target value. However, it should be possible to terminate for any variables (including element and nodal), and to allow the target solution to be a user-defined, non-constant variable (e.g., defined

via an analytic function that depends on time or other field variables). The global variables used in solution termination can be created via a `USER OUTPUT` based command block.

Tolerances of how close a solution needs to be to terminate can be specified. The `TOLERANCE` and `PLUS OR MINUS` termination option both specify absolute tolerances. For example the command:

```
TERMINATE GLOBAL disp1 = 1.0 PLUS OR MINUS 0.1
```

will terminate the calculation at the first occurrence of disp1 being between 0.9 and 1.1.

The `RELATIVE TOLERANCE` command specifies an acceptable relative difference between values for termination. Termination occurs if:

$$\frac{|gVal - gVar|}{max(|gVal|, |gVar|)} < relTol \tag{2.2}$$

The `TIME TOLERANCE` is used with the time history function based termination commands. A termination will result if the `gVar` is within tolerance to any value of the function that would be computed at the current time plus or minus the `timeTol`.

The `SKIP TIMES` command is used to skip evaluation of whether to terminate over the defined time intervals. This enables solution-based termination to be active only during certain time intervals. Similarly solution termination can be made active only for one or more specified time periods defined in the time stepping block with use of the `ACTIVE PERIODS` and `INACTIVE_PERIODS` command lines described in Section 2.6 to activate and deactivate functionality during specific time blocks.

The `COMPLETION FILE` command causes solution termination to output a status file named `fileName` at successful completion of the code run. Existence of the status file indicates that the run was terminated successfully. If the termination condition is never reached, the code should run until the final time period.

The `TERMINATE TYPE` command defines what solution termination does when hitting the failure condition. The default `ENTIRE_RUN` option indicates that the entire run should exit when the termination condition is reached. The `CURRENT_PROCEDURE` option indicates that the current procedure will be stopped and the code will move on to the next defined procedure if there is one. If the subsequent procedure defines a start time the next procedure will start at that time. If the subsequent procedure does not define a start time the next procedure will start at the time the current procedure was terminated. The `CURRENT_TIME_PERIOD` option indicates that the current time period (as defined in the time stepping block) will be ended. The Sierra/SM will then move onto the next defined time period if there is one.

The `WRITE OUTPUT` option will write a new results, history, and restart files upon solution termination.



Warning: Multiple results will exist on the output files if output is already scheduled to be written and solution termination also triggers output to be written.

2.9.1. Solution Termination Examples

The following code is used to terminate a run when the global kinetic energy reaches some small value. This may be useful for ending an explicit analysis after a dynamic impact event has occurred and most of the dynamic energy of interest has been dissipated.

```
BEGIN SOLUTION TERMINATION
  TERMINATE GLOBAL kinetic_energy < 1.5e+4
END
```

The following example code could be used end an implicit run immediately before some fracture criteria (eqps) being reached. The fracture itself could then be evaluated in a subsequent explicit procedure.

```
BEGIN USER OUTPUT
  COMPUTE GLOBAL max_eqps AS MAX OF ELEMENT EQPS
  COMPUTE AT EVERY STEP
END

BEGIN SOLUTION TERMINATION
  TERMINATE GLOBAL max_eqps > 0.2
  TERMINATE TYPE = CURRENT_PROCEDURE
END
```



3. EXPLICIT DYNAMIC TIME STEP CONTROL

This chapter discusses time control for explicit dynamic Sierra/SM analyses. We begin with a broad overview of explicit dynamic time control and then describe the options that are available to users for time control. The commands and features documented in this chapter are applicable only for explicit dynamic analyses. For information on time stepping commands for implicit analyses, see Chapter 4.

The user initiates time control by setting a start time and a termination time for an analysis. The analysis is typically carried out with a large number of time steps, with each time step being much smaller than the analysis time. In explicit transient dynamics analyses, the time step must be less than a critical value. Sierra/SM has a number of methods for computing an estimate for the critical time step. These methods are discussed in detail in this chapter.

The primary time control uses a `TIME CONTROL` command block that appears in the procedure scope. Use of the `TIME CONTROL` command block gives the user, by default, access to an element-based method for estimating the critical time step. The user can access three other methods for estimating the critical time step by using specific command blocks that are placed in the region scope. These other methods tend to give better (larger) estimates for the critical time step. In addition, there is a technique for adjusting the time step known as mass scaling.

Section 3.1 describes the `TIME CONTROL` command block. In Section 3.2 we discuss the other methods for estimating the critical time step. One approach for improving this estimate is to compute the maximum eigenvalue for a problem. Two methods for computing the maximum eigenvalue are available: the Lanczos method and the power method. Section 3.2.1 discusses the Lanczos method; Section 3.2.1.5 describes the command block required to implement the Lanczos method. Section 3.2.2 discusses the power method; Section 3.2.2.5 describes the command block required to implement the power method. Another approach for improving the time step estimate relies on a node-based estimate. Section 3.2.3 discusses the node-based method; Section 3.2.3.1 describes the command block required to implement the node-based method. You should read the introductory material for the maximum eigenvalue calculation methods and the node-based method and understand this material thoroughly before you attempt to use these methods. Although these other methods give larger time step estimates than the element-based method, they may not result in a net reduction of central processor unit (CPU) time for an analysis unless they are used properly. In those sections dealing with these other methods, we discuss how to use these methods in a cost-effective manner. In Section 3.3 we describe the technique of mass scaling, which is a technique that artificially scales up nodal masses to allow larger time steps to be taken, and must be used with the node-based critical time step estimation method.

3.1. Procedure Time Control

As indicated previously, the primary time control uses a `TIME CONTROL` command block in the procedure scope. The user sets the start time and the termination time for the analysis in the `TIME CONTROL` command block. For reference purposes, the general layout of the command block is as follows:

```
BEGIN TIME CONTROL
  BEGIN TIME STEPPING BLOCK <string>time_block_name
    START TIME = <real>start_time_value
    BEGIN PARAMETERS FOR PRESTO REGION <string>region_name
      INITIAL TIME STEP = <real>initial_time_step_value
      TIME STEP SCALE FACTOR =
        <real>time_step_scale_factor(0.9)
      TIME STEP INCREASE FACTOR =
        <real>time_step_increase_factor(1.1)
      STEP INTERVAL = <integer>nsteps(100)
      USER TIME STEP = <real>time_step
      TIME STEP SELECTOR = AUTO
      PRINT BANNER INTERVAL FOR SOLUTION CONTROL =
        <int>print_banner_interval(MAX_INT)
    END [PARAMETERS FOR PRESTO REGION <string>region_name]
  END [TIME STEPPING BLOCK <string>time_block_name]
  # Second TIME STEPPING BLOCK command block
  # would be placed here, as applicable.
  #
  # Additional TIME STEPPING BLOCK command blocks
  # would be placed here, as applicable.
  #
  # Last TIME STEPPING BLOCK command block
  # would be placed here, as applicable.
  #
  TERMINATION TIME = <real>termination_time
END [TIME CONTROL]
```

The analysis time, as demonstrated above, can be subdivided into a number of time blocks, i.e., `TIME STEPPING BLOCK` command blocks. If the total analysis time is from time 0 to time T and there are three blocks, then the first block is defined from time 0 to time t_1, the second block is defined from time t_1 to time t_2, and the third block is defined from time t_2 to time T. (The times t_1 and t_2 are set by the user.) If we sum all the times for each block, the sum will be T. The different time periods defined by each block can be referenced so that we can turn certain functionality on or off throughout an analysis. For example, we may want to have a certain boundary condition turned off during our first time period and activated for the second time period. (Most analyses require only one block.) Use the `ACTIVE PERIODS` or `INACTIVE PERIODS` command lines described in Section 2.6 to activate and deactivate functionality during specific time blocks.

By default, Sierra/SM relies on the element-based critical time step estimate. At every time step, an element-based calculation is performed to determine a critical time step, based on an estimate of the Courant-Friedrichs-Lewy (CFL) condition that uses the effective material moduli. The effective material moduli can be calculated a number of ways, specified by the `EFFECTIVE MODULI MODEL` command line described in 6.1.7.7. You also have some direct control over the actual time step that is used by employing one of two techniques. We discuss these two techniques in the following paragraphs. The specific command lines for using these techniques are described in Section 3.1.1.

With the `INITIAL TIME STEP` line command you can set an initial time step that is smaller than the element-based critical time step estimate. Sierra/SM will start the analysis by using your initial time step value instead of the element-based critical time step estimate (as long as your value is less than the element-based critical time step estimate). You can then control the rate at which the time step increases from your initial value. This technique is employed by using the `TIME STEP INCREASE FACTOR` command line as follows:

- When setting `TIME STEP INCREASE FACTOR` to 1, the time step will never increase past the specified value of `INITIAL TIME STEP`. However, if the element-based critical time step is less than the value of the `INITIAL TIME STEP` at some point during the simulation, the time step will be decreased accordingly.
- Setting `TIME STEP INCREASE FACTOR` to a value greater than 1 causes the time step to grow (from the initial time step) at each time step until it reaches the value of the element-based critical time step estimate. From then on, the element-based critical time step estimate will essentially control the time step.

With the second technique, you can manipulate the element-based estimate with either a scale factor or a time step increase factor. This technique is employed by using the `TIME STEP SCALE FACTOR` command line or the `TIME STEP INCREASE FACTOR` command line as follows:

- The element-based estimate for the critical time step is usually smaller than some maximum theoretical value for your model. It may therefore be possible to scale the element-based critical time step estimate by some factor greater than 1. (Your scaled value must remain below the theoretical maximum limit, however. We discuss ways to obtain a critical time step close to the theoretical maximum in later sections of this chapter.)
- If there are stability problems with a particular problem, it may be necessary to scale the element-based estimate with a factor less than 1.
- You can also control the rate at which the time step can increase for an analysis. By specifying a time step increase factor, you can limit the increase in the size of the time step so that it does not increase too rapidly from one step to the next. For certain problems, the element-based critical time step estimate may increase rapidly from one step to the next. Limiting the increase in the size of the time step may enable some problems to run in a more stable fashion.

The `USER TIME STEP` command can be used to explicitly set the integration time step, overriding the critical time step computed by any time step estimation methods. This command can be used to force the analysis to integrate at any specified time step, even a super-critical one.

The TIME STEP SELECTOR = AUTO will cause Sierra/SM to select the most advantageous time step from between the standard element based and nodal based time step methods.



Warning: The TIME STEP SCALE FACTOR and USER TIME STEP commands both provide the ability to force the code to take a time step larger than the stability limit, and thus should be used with caution. If the time step specified with either of these commands is too large, the time integration will add energy to the system, causing the analysis to go unstable and ultimately fail.

Now that we have presented an overview of the functionality in the TIME CONTROL command block, we will discuss the actual command lines.

3.1.1. Command Blocks for Time Control and Time Stepping

```
BEGIN TIME CONTROL
  BEGIN TIME STEPPING BLOCK <string>time_block_name
    START TIME = <real>start_time_value
    BEGIN PARAMETERS FOR PRESTO REGION <string>region_name
      #
      # Time control parameters specific to PRESTO
      # are set in this command block.
      #
    END [PARAMETERS FOR PRESTO REGION <string>region_name]
  END [TIME STEPPING BLOCK <string>time_block_name]
  TERMINATION TIME = <real>termination_time
END [TIME CONTROL]
```

Time control resides in a TIME CONTROL command block. The command block begins with an input line of the form

```
BEGIN TIME CONTROL
```

and terminates with an input line of the following form:

```
END [TIME CONTROL]
```

Within the TIME CONTROL command block, a number of TIME STEPPING BLOCK command blocks can be defined. Each TIME STEPPING BLOCK command block contains the time at which the time stepping starts and a number of parameters that set time-related values for the analysis. Each TIME STEPPING BLOCK command block terminates at the start time of the following command block. The start times for the TIME STEPPING BLOCK command blocks must be in increasing order. Otherwise, an error will be generated. (The example in Section 3.1.7 shows the overall structure of the TIME CONTROL command block.)

In the above input lines, the values are as follows:

- The string time_block_name is a name for the TIME STEPPING BLOCK command block. This name must be unique to each command block of this type. The string

time_block_name can be referenced in an ACTIVE PERIODS or INACTIVE PERIODS command line to activate and deactivate functionality (see Section 2.6).

- The real value start_time_value is the start time for this TIME STEPPING BLOCK command block. Values set in this block apply from the start time for this block until the next start time or the termination time. The start time may be negative.
- The string region_name is the name of the Presto region affected by the parameters (see Section 2.2).

The termination time for the analysis is given by the following command line:

```
TERMINATION TIME = <real>termination_time
```

Here, termination_time is the time at which the analysis will stop. The TERMINATION TIME command line appears inside the TIME CONTROL command block but outside of any TIME STEPPING BLOCK command block.

The TERMINATION TIME command line can appear before the first TIME STEPPING BLOCK command block or after the last TIME STEPPING BLOCK command block. It is permissible to have TIME STEPPING BLOCK command blocks with start times after the termination time; in this case, those command blocks that have start times after the termination time are not executed. Only one TERMINATION TIME command line can appear in the TIME CONTROL command block. If more than one of these command lines appears, an error is generated.

Nested inside the TIME STEPPING BLOCK command block is a PARAMETERS FOR PRESTO REGION command block containing parameters that control the time stepping.

```
BEGIN PARAMETERS FOR PRESTO REGION <string>region_name
  INITIAL TIME STEP = <real>initial_time_step_value
  TIME STEP SCALE FACTOR = <real>time_step_scale_factor(0.9)
  TIME STEP INCREASE FACTOR =
    <real>time_step_increase_factor(1.1)
  STEP INTERVAL = <integer>nsteps(100)
  PRINT BANNER INTERVAL FOR SOLUTION CONTROL =
    <integer>print_banner_interval(MAX_INT)
END [PARAMETERS FOR PRESTO REGION <string>region_name]
```

These parameters are specific to a Presto analysis.
PRINT BANNER INTERVAL FOR SOLUTION CONTROL only applies to a Presto region included in a solution control analysis.

The command block begins with an input line of the form

```
BEGIN PARAMETERS FOR PRESTO REGION <string>region_name
```

and is terminated with an input line of the following form:

```
END [PARAMETERS FOR PRESTO REGION <string>region_name]
```

As noted previously, the string region_name is the name of the Presto region affected by the parameters. The command lines nested inside the PARAMETERS FOR PRESTO REGION

command block are described next. Some of these command lines will be ignored when either of the maximum-eigenvalue methods or the node-based method is used to estimate the critical time step. The discussions of the command blocks associated with these methods indicate whether these command lines are pertinent.

3.1.2. Initial Time Step

```
INITIAL TIME STEP = <real>initial_time_step_value
```

Presto computes a critical time step for the analysis and uses this value as the initial time step by default. To directly specify a smaller initial time step, use the `INITIAL TIME STEP` command line, where `initial_time_step_value` is the desired initial time step. This command line is only valid if it is in the first `TIME STEPPING BLOCK` command block in the problem.

The value specified for the initial time step will overwrite the calculated value for the critical time step if the specified initial time step is smaller than the critical time step. If you specify an initial time step that is larger than the critical time step, the time step is set to the value of the calculated critical time step.

3.1.3. Time Step Scale Factor

```
TIME STEP SCALE FACTOR = <real>time_step_scale_factor(0.9)
```

During the element computations, Presto computes a minimum time step required for stability of the computation (the critical time step). Using the `TIME STEP SCALE FACTOR` command line, you can provide a scale factor to modify the critical time step. A value greater than 1.0 for `time_step_scale_factor` will cause the time step to be greater than the computed critical time step, and thus the problem may become unstable. By default, the scale factor is 0.9.

3.1.4. Time Step Increase Factor

```
TIME STEP INCREASE FACTOR =
  <real>time_step_increase_factor(1.1)
```

During an analysis, the computed critical time step may change as elements deform, are killed, and so forth. By using the `TIME STEP INCREASE FACTOR` command line, you can limit the amount that the time step can increase between two adjacent time steps. The value `time_step_increase_factor` is a factor that multiplies the previous time step. The current time step can be no larger than the product of the previous time step and the scale factor.

An increase factor less than 1.0 will cause the time step to continuously decrease. The default value for this factor is 1.1, i.e., a time step cannot be more than 1.1 times the previous step.

3.1.5. Step Interval

```
STEP INTERVAL = <integer>nsteps(100)
```

Presto can output data about the current time step, the current internal and external energy, and the kinetic energy throughout an analysis to the log file. The `STEP INTERVAL` command line controls the frequency of this output, where `nsteps` is the number of time steps between output. The default value for `nsteps` is 100.

The output at any given step (read from left to right) is

- step number,
- time,
- time step,
- global element identifier for element controlling time step
- kinetic energy,
- internal energy,
- external energy,
- error in energy balance,
- hourglass energy
- CPU time, and
- wall clock time.

The time is at the current time, step n, and the time increment is the previous time step increment from step $n-1$ to step n.

The error in the energy balance is computed from the following relation:

```
energy balance error = (kinetic energy + internal energy
  - external energy) / external energy * 100
```

The above expression gives a percent error for the energy balance. An error in energy imbalance can indicate one of a number of potential issues:

- Time integration error. Each of the three kinetic, internal, and external energy terms are computed in different ways. Thus an imbalance in the terms could indicate an error in one or more of the terms due to a large time step.
- Something unphysical (but not necessarily incorrect) has occurred. For example during element death material and the associated energy blinks out of the model. The energy removed by such a process is not guaranteed to balance which gives some indication of the error numerical error introduced by element death. Similarly transferring results between non-matching meshes will tend to generate energy imbalance due to the inherit accuracy limitations of such transfers.
- Instability. A numeric instability (such as taking an unstable time step) will almost always manifest a large energy balance error.
- Energy accounting limitation. Additionally there are a few physical processes for which the energy is not fully accounted by Sierra/SM. One example is energy generating materials

such as explosives. If such materials are present the energy in the analysis may not balance, but this would not necessarily indicate anything wrong.

3.1.6. Print Banner Interval for Solution Control

```
PRINT BANNER INTERVAL FOR SOLUTION CONTROL =
  <integer>print_banner_interval(MAX_INT)
```

The `PRINT BANNER INTERVAL FOR SOLUTION CONTROL` line command sets the number of output lines to print before re-printing the column headings when a Presto region is used within a solution control analysis. The default value is set to the maximum allowed integer value and means that the column headings will be printed in the solution control log file only at the beginning of each advance of the presto region. A smaller value can be used to print the column headings more often when presto is subcycled within a solution control time step.

This command is only applicable when using solution control for a coupled analysis. It will be ignored in a stand-alone presto procedure. To specify the frequency of printing column headings for a presto procedure, see Section 2.2 for the description of the command `PRINT BANNER INTERVAL`.

3.1.7. Time Control Example

The following is a simple example of a `TIME CONTROL` command block:

```
BEGIN TIME CONTROL
  BEGIN TIME STEPPING BLOCK p1
    START TIME = 0.0
    BEGIN PARAMETERS FOR PRESTO REGION presto_region
      INITIAL TIME STEP = 1.0e-6
      STEP INTERVAL = 50
    END
  END
  BEGIN TIME STEPPING BLOCK p2
    START TIME = 0.5e-3
    BEGIN PARAMETERS FOR PRESTO REGION presto_region
      TIME STEP SCALE FACTOR = 0.9
      TIME STEP INCREASE FACTOR = 1.5
      STEP INTERVAL = 10
    END
  END
  TERMINATION TIME = 1.0e-3
END
```

The first `TIME STEPPING BLOCK`, `p1`, begins at time 0.0, the initial start time, and terminates at time 0.5×10^{-3}. The second `TIME STEPPING BLOCK`, `p2`, begins at time 0.5×10^{-3} and

terminates at time 1.0×10^{-3}, which is the time listed on the `TERMINATION TIME` command line. The `TIME STEPPING BLOCK` names `p1` and `p2` can be referenced by the `ACTIVE PERIODS` or `INACTIVE PERIODS` command lines as described in Section 2.6 to activate and deactivate functionality.

3.2. Other Critical Time Step Methods

Currently, there are four methods for calculating a critical time step for Presto. First, there is the traditional element-based method. We know that, in general, the element-based time step in Presto can give a fairly conservative estimate for the time step. Second, there is a node-based method for giving a critical time step estimate. Depending on the problem, the node-based method may or may not give an estimate for the critical time step that approaches the theoretical maximum value for a particular model. Although the node-based method can give a larger critical time step estimate than the element-based method, the node-based estimate may still be significantly lower than the maximum theoretical time step for a problem. Finally, there are two methods that use an estimate for the maximum eigenvalue to obtain an estimate for the critical time step. From the maximum eigenvalue, it is possible to derive the theoretical maximum critical time step for a problem via the formula

$$\Delta t_{crit} = 2.0/\sqrt{\theta_{max}}, \tag{3.1}$$

where Δt_{crit} is the critical time step and θ_{max} is the maximum eigenvalue. The two methods employed in Presto to give an estimate for the maximum eigenvalue are the Lanczos method and the power method. The Lanczos method can give an accurate estimate of the maximum eigenvalue for a problem using a very small number of Lanczos vectors compared to the total number of degrees of freedom in a problem. The power method uses a simple iterative process to obtain an estimate for the maximum eigenvalue. The power method is not as powerful a mathematical technique as the Lanczos method for obtaining a maximum eigenvalue estimate. However, it does present another viable option for obtaining a maximum eigenvalue estimate for certain problems and has therefore been included as another method for obtaining the maximum eigenvalue estimate. See [1, 2] for more information on the Lanczos method and other related methods in computational linear algebra.

The use of a critical time step from the node-based method or a method based on the maximum eigenvalue estimate is desirable because the larger critical time steps produced by these methods (compared to the element-based method) reduce CPU time. Both methods, however, are not cost effective if they are called every time step to give a critical time step estimate. The cost of doing one node-based estimate or one maximum eigenvalue estimate for the critical time step will not offset the cost benefit of the increase to the critical time step (compared to the element-based time step estimate) over a single time step. Hence, there must be some scheme for

- calling these methods only periodically throughout a calculation and
- maintaining a larger estimate (than the element-based estimate) for the critical time step in between these calls

if we are to gain a net benefit from the increase in the critical time step these methods can produce.

If you want to use the maximum eigenvalue estimate for calculating the critical time step and your problem has long periods with a relatively stable time step estimate, the Lanczos method for calculating the maximum eigenvalue will be the preferred method to use. If you have a problem for which the changes in the time step should be monitored more frequently, the power method will be better suited for this problem than the Lanczos method. The preferred use for each of

these two different eigenvalue methods will become obvious as you read the background material for these two methods in Section 3.2.1 and Section 3.2.2. Both the Lanczos method and the power method require the use of a variety of control parameters. For some of these control parameters, techniques have been devised to automatically calculate values for these parameters in what should be a robust and reliable method for a wide range of problems. Other parameters are defaulted to values that will optimize the use of a particular eigenvalue calculation method (Lanczos method or power method) with the type of problems that best suit the method that is employed. The default values for these control parameters may change as we gain more experience in using the Lanczos method and the power method.

Detailed discussions of the Lanczos method, the power method, and the node-based method follow. The implementation of the Lanczos method and the power method have many similarities.

3.2.1. Lanczos Method

The Lanczos method, as implemented in Presto, is outlined here so that certain aspects of it can be referenced in subsequent parts of this chapter. In the following set of equations, $\mathbf{K}_T$ is a tangent stiffness matrix, $\mathbf{M}$ is the mass matrix, and $\mathbf{r}_0$ is an arbitrary starting vector.

$$
\begin{aligned}
&\textit{Initialize}\\
&\quad \mathbf{q}_0 = \mathbf{0}\\
&\quad \beta = ([\mathbf{r}_0]^T \mathbf{M}^{-1} \mathbf{r}_0)^{1/2}\\
&\quad \mathbf{q}_1 = \mathbf{r}_0/\beta_1\\
&\quad \mathbf{p}_1 = \mathbf{M}^{-1}\mathbf{q}_1\\
&\textit{for } j = 1,n\\
&\quad \bar{\mathbf{r}}_j = \mathbf{K}_T \mathbf{p}_j\\
&\quad \hat{\mathbf{r}}_j = \bar{\mathbf{r}}_j - \mathbf{q}_{j-1}\beta_j\\
&\quad \alpha_j = [\mathbf{q}_j]^T \mathbf{M}^{-1} \hat{\mathbf{r}}_j = [\mathbf{p}_j]^T \hat{\mathbf{r}}_j\\
&\quad \mathbf{r}_j = \hat{\mathbf{r}}_j - q_j \alpha_j\\
&\quad \hat{\mathbf{p}}_j = \mathbf{M}^{-1}\mathbf{r}_j\\
&\quad \beta_{j+1} = ([\mathbf{r}_j]^T \mathbf{M}^{-1} \mathbf{r}_j)^{1/2} = ([\bar{\mathbf{p}}_j]^T \mathbf{r}_j)^{1/2}\\
&\quad \textit{if enough vectors, terminate loop}\\
&\quad \mathbf{q}_{j+1} = \mathbf{r}_j/\beta_{j+1}\\
&\quad \mathbf{p}_{j+1} = \bar{\mathbf{p}}_j/\beta_{j+1}\\
&\textit{end}
\end{aligned}
$$

The details for this form of the Lanczos method are described in Reference [3]. Notice that the Lanczos method is an iterative method. If we use the Lanczos method in a code like Presto to

compute the maximum eigenvalue for a particular finite element model, the number of iterations required to give a good estimate for the maximum eigenvalue will depend on the size of the finite element model (the number of nodes and elements), the types of elements in the model, and the material types and material properties used in the model.

The maximum eigenvalue for a particular finite element model gives us the largest estimate we can obtain for the critical time step. Hence, the time step estimate derived from the maximum eigenvalue is our "best" estimate for the critical time step. The estimate for the critical time step based on the maximum eigenvalue can be significantly larger than an element-based time step estimate. However, computing the critical time step (for a given time step) with the Lanczos method is more expensive than computing the critical time step with element-based calculations. Over one time step, it is not possible to recoup the cost of the Lanczos calculations with the increase in the size of the time step over the element-based estimate. Using the Lanczos method for estimating the critical time step in an explicit, transient dynamics code requires a methodology that effectively addresses the computational costs. The following sections outline a cost-effective approach to using the Lanczos method in an explicit, transient dynamics code.

3.2.1.1. Lanczos Method with Constant Time Steps

To explain how to use the Lanczos method in a cost-efficient manner, we must first establish the computational cost of using the Lanczos method. As indicated previously, computing the maximum eigenvalue for a finite element model requires some number of iterations (each iteration in the Lanczos method produces a Lanczos vector) to obtain a good estimate for the maximum eigenvalue. The cost of an iteration (Lanczos vector) is approximately the cost of an internal force calculation. Notice, in preceding equations for the Lanczos method, that the Lanczos method requires the product of the tangent stiffness matrix $\mathbf{K}_T$ with a vector $\mathbf{p}$. In Presto, we do not construct a tangent stiffness matrix. Instead, we provide the vector $\mathbf{p}$ for the internal force calculations. The internal force calculations give us the desired matrix $\times$ vector product of $\mathbf{K}_T\mathbf{p}$.

Over a given time step, the cost of an internal force calculation is the major computational cost. (This assumes no contact. The addition of contact introduces another computationally expensive process into a time step. For our initial discussion, we ignore the cost of contact.) To use the Lanczos method to get a critical time step estimate, one has to call the Lanczos method for some given time step. If the Lanczos method made n iterations to get a good estimate for the maximum eigenvalue, then the overall cost of the time step would be approximately $n+1$ times the cost of the internal force calculation. The cost of the time step would be the n internal force calculations for Lanczos and the actual internal force calculation to advance the time step. If n is 20 (a minimum for typical problems), the cost of the time step becomes 21 times the cost of the internal force calculation. The critical time step estimate based on the maximum eigenvalue would have to be at least 21 times greater than the element-based critical time step to recoup the cost of the maximum eigenvalue calculation. A typical value for the critical time step estimate based on the maximum value is more in the range of 1.1 to 2.0 times the element-based critical time step estimate. Obviously, the Lanczos method is too expensive to call at every time step for a critical time step estimate.

To explain how the Lanczos method can effectively be used in an explicit, transient dynamics code, we begin with a simple case study. In this case study, we compute the critical time step using the Lanczos method at some time step and then assume that this critical time step value remains constant for a subsequent number of time steps, n_L. We only call the Lanczos method once during the n_L time steps. (In reality, the critical time step in an explicit, transient dynamics code like Presto changes with each time step. We address this issue of the changing time step when we present the details for using the Lanczos method in a cost-effective manner.) For this case study, we also assume that the computational cost of an element-based estimate for the critical time step is part of the cost of an internal force calculation. The cost of the element-based time step estimate is a small part of the overall internal force calculations. Finally, for our initial discussion, we assume no contact. We address the issue of contact further in the discussion.

Assume that the Lanczos method computes a global estimate for the critical time step of Δt_L, which is the value to be used for n_L time steps. At the end of the n_L time steps, the analysis time for the code has been incremented by an amount ΔT, which is computed as

$$\Delta T = n_L \Delta t_L \,. \tag{3.2}$$

If the element-based estimate for the time step is Δt_e and the number of time steps required to increment the analysis time by ΔT is n_e, then, for the element-based time step, we have

$$\Delta T = n_e \Delta t_e \,. \tag{3.3}$$

Because the Lanczos estimate for the critical time step is larger than the element-based estimate, we know that $n_e > n_L$. Let us define the ratio r as

$$r = \Delta t_L / \Delta t_e = n_e / n_L \,. \tag{3.4}$$

The ratio r is greater than 1. Now that we have determined the relation between the number of steps required for a Lanczos-based critical time step estimate versus the element-based critical time step estimate to achieve the same analysis time increment, let us examine the computational costs for these two cases in terms of CPU time. Designate the CPU cost for a time step as Δt_{IF}. If the number of Lanczos vectors required to obtain the critical time step estimate is N_L, then the total computational cost of the Lanczos method and the n_L time steps is

$$n_L \Delta t_{IF} + N_L \Delta t_{IF} \,. \tag{3.5}$$

If we use the element-based method, the total computational cost is

$$n_e \Delta t_{IF} \,. \tag{3.6}$$

Recall that we have chosen n_L and n_e so that we have the same analysis time increment ΔT even though we have different critical time steps. Now, we must determine the point at which the computational cost for the Lanczos-based critical time step calculations is the same as the cost for the element-based critical time step calculations. This is the point at which

$$n_e \Delta t_{IF} = n_L \Delta t_{IF} + N_L \Delta t_{IF} \,. \tag{3.7}$$

If we rearrange the above equation to eliminate Δt_{IF} and make use of the ratio r, then we obtain

$$n_e = \frac{N_L}{1 - 1/r}. \tag{3.8}$$

Consider the case of $r = 1.25$ and $N_L = 20$. When $n_L = 80$ and $n_e = 100$, the above equations show that the calculations with the Lanczos-based critical time step and the calculations with the element-based time step give the same analysis time for the same computational expense. If we use the Lanczos-based critical time step Δt_L for more than eighty iterations, then the Lanczos-based approach becomes cost effective. Our above equations have established the "break-even" point at which it becomes cost effective to use the Lanczos method to reduce computational costs by overcoming the initial cost of the Lanczos calculations with the larger critical time step.

We can build on what we have done thus far to account for contact. Suppose that the computational cost of contact over a time step is some multiple m of the computational cost of the internal force calculation Δt_{IF}. Then the point at which the computational cost for the Lanczos-based calculations is the same as the computational cost for the element-based calculations is

$$(1 + m)n_L \Delta t_{IF} + N_L \Delta t_{IF} = (1 + m)n_e \Delta t_{IF}. \tag{3.9}$$

For the case with contact,

$$n_e = \frac{N_L}{(1 + m)(1 - 1/r)}. \tag{3.10}$$

Again, consider the case of $r = 1.25$ and $N_L = 20$. Assume the computational cost of contact calculations is the same as an internal force calculation ($m = 1$). For these values, the break-even point is $n_L = 40$ and $n_e = 50$. The added computational cost of the contact calculations results in reaching the break-even point with a smaller number of iterations when compared to the case with no contact.

The above derivations let us calculate a break-even point based on our assumptions of a constant critical time step. Considering that a typical analysis will run for tens of thousands of time steps, something on the order of 100 steps represents a reasonable number of steps to recoup the cost of the Lanczos calculations. Whether the cost of the Lanczos calculations can be recouped in something on the order of 100 calculations depends heavily upon N_L. If N_L is sufficiently small, we can recoup the cost of the Lanczos calculations in a reasonable number of steps.

Some computational studies indicate that N_L is in an acceptable range for many problems. The Lanczos method computes a good estimate for the maximum eigenvalue with a small number of Lanczos vectors, N_L, compared to the number of degrees of freedom in a problem. Some component studies show that, for a problem with between 250,000 and 350,000 degrees of freedom, one can obtain a good estimate for the maximum eigenvalue with only twenty Lanczos vectors. A large-scale study of a model involving 1.7 million nodes (5.1 million degrees of freedom) showed that only forty-five Lanczos vectors were required to obtain a good estimate of the maximum eigenvalue. These examples demonstrate that the number of Lanczos vectors required for a good maximum eigenvalue estimate is very small when compared to the number of degrees of freedom for a problem. When N_L is in the range of twenty to forty-five, Equations (3.7) and (3.10) show that, with an increase in the time step of about 1.2 to 1.25, we can recoup the cost of the Lanczos method in a reasonable number of time steps.

Now that we have determined we can recoup the cost of the Lanczos calculations in a reasonable number of time steps, let us look at the issue of reusing a Lanczos-based estimate in some manner.

3.2.1.2. Controls for Lanczos Method

As indicated in the above discussion, the Lanczos method can be used in a cost-effective manner in an explicit, transient dynamics code if a Lanczos calculation can be performed and the Lanczos-based estimate for the critical time step can be reused in some way over many subsequent time steps. This section presents an approach for reusing a Lanczos-based estimate over a number of time steps so that we maintain a critical time step estimate that is close to the theoretical maximum value in between the calls to the Lanczos method. The approach discussed here makes use of the element-based critical time step estimate at each time step.

We start our approach with a Lanczos calculation to determine the maximum eigenvalue. The Lanczos method converges to the maximum eigenvalue from below, which means that the method underestimates the maximum eigenvalue. Because the critical time step depends on the inverse of the maximum eigenvalue, we overestimate the critical time step. It is necessary, therefore, to scale back the critical time step estimate from the Lanczos method so that the calculations in the explicit time-stepping scheme do not become unstable. Our approach for determining a scaled-back value for the maximum critical time step makes use of the element-based time step estimate. Again, let Δt_L be the critical time step estimate from the Lanczos method and Δt_e be the critical time step estimate from the element-based calculations. The scaled-back estimate for the critical time step, Δt_s, is computed from the equation

$$\Delta t_s = \Delta t_e + f_s(\Delta t_L - \Delta t_e)\,, \tag{3.11}$$

where f_s is a scale factor. (A reasonable value for f_s ranges from 0.9 to 0.95 for our problems.) This value of f_s puts Δt_s close to and slightly less than a theoretical maximum critical time step.

Once Δt_s is determined, the ratio

$$t_r = \Delta t_s / \Delta t_e \tag{3.12}$$

is computed. This ratio is then used to scale subsequent element-based estimates for the critical time step. If $\Delta t_{e(n)}$ is the nth element-based critical time step after the time step where the Lanczos calculations are performed, then the nth time step after the Lanczos calculations, $\Delta t_{(n)}$, is

$$\Delta t_{(n)} = t_r \Delta t_{e(n)}\,. \tag{3.13}$$

The ratio t_r is used until the next call to the Lanczos method. The next call to the Lanczos method is controlled by one of two mechanisms. With the first mechanism, the user can set the frequency with which the Lanczos method is called. The user can set a parameter so that the Lanczos method is called only once every n time steps. This number remains fixed throughout an analysis. With the second mechanism, the user can control when the Lanczos method is called based on changes in the element-based critical time step. For this second mechanism, the change in the element-based critical time step estimate is tracked. Suppose the element-based critical time step

at the time the Lanczos method was called is Δt_e. At the nth step after the call to the Lanczos method, the element-based critical time step is $\Delta t_{e(n)}$. If the value

$$\Delta t_{lim} = \frac{|\Delta t_{e(n)} - \Delta t_e|}{\Delta t_e} \tag{3.14}$$

is greater than some limit set by the user, then the Lanczos method will be called. If there is a small, monotonic change in the element-based critical time step over a large number of time steps, this second mechanism will result in the Lanczos method being called. Or if there is a large, monotonic change in the element-based critical time step over a few time steps, the Lanczos method will also be called.

These two mechanisms for calling the Lanczos method can be used together. For example, suppose the second mechanism (the mechanism based on a change in the element-based time step) results in a call to the Lanczos method. This resets the counter for the first mechanism (the mechanism using a set number of time steps between calls to the Lanczos method).

This approach for reusing a Lanczos-based time step estimate has been implemented in Presto, and used for a few studies. One of the component studies, as indicated previously, used the same scale factor for $n_L = 1700$ iterations. The break-even point for this problem is $n_e = 45$ time steps (not accounting for contact, which was a part of the component modeling). For this particular problem, the extended use of the Lanczos estimate reduced the computational cost to 56% of what it would have been with the element-based time step.

Not all problems will lend themselves to reusing one Lanczos-based estimate for thousands of time steps. However, if it is possible to use the Lanczos-based estimate for two to three times the number of time steps required to reach the break-even point, we begin to see a noticeable reduction in the total number of time steps required for a problem.

3.2.1.3. Scale Factor for Lanczos Method

When the Lanczos method is called for a given time step, it must appear that the calculations are using the constant tangent stiffness matrix $\mathbf{K}_T$ for all iterations. As indicated previously, we use the internal force calculations to generate the product $\mathbf{K}_T\mathbf{p}_j$ (for the jth iteration) in the Lanczos calculations. Any vector $\mathbf{p}_j$, as calculated by the Lanczos method, may be such that it represents large-strain behavior and moves the internal force calculations into a nonlinear regime. It is necessary to scale the $\mathbf{p}_j$ vectors so that the internal force calculations are in a small-strain regime, which makes it appear that we are working with a constant tangent stiffness matrix. The vectors $\mathbf{p}_j$ must be scaled so that they represent velocities associated with small strain. When properly scaled vectors are sent to the internal force calculation, the internal force calculation effectively becomes a matrix $\times$ vector product with a constant tangent stiffness matrix for all iterations during a given call to the Lanczos method.

The scale factor for the $\mathbf{p}_j$ vectors, which we will designate as v_{sf}, must not be too small, as this will create round-off problems and give a bad estimate for the critical time step. If the scale factor is too large, we violate the above restriction of a constant tangent stiffness matrix.

There are two approaches for controlling the scale factor when the Lanczos method is used to compute the maximum eigenvalue. These approaches are discussed in Section 3.2.1.5.

3.2.1.4. Accuracy of Eigenvalue Estimate

Every time a new Lanczos vector is computed, we obtain an additional eigenvalue for our model and, in general, a better estimate for the maximum eigenvalue (and hence a better estimate for the critical time step). The Lanczos method can compute a good value for the maximum eigenvalue with a very small number of total computed eigenvalues compared to the number of degrees of freedom in a problem. There are examples, as previously indicated, of problems with 250,000 to 350,000 degrees of freedom where we have obtained a good estimate of the critical time step with twenty eigenvalues. In one problem with 5.1 million degrees of freedom, we obtained a good estimate of the critical time step with forty-five eigenvalues.

A user could, in theory, determine a reasonable number of eigenvalues necessary for obtaining a good estimate of the maximum eigenvalue based on the above information on model size and the number of eigenvalues required for a good maximum eigenvalue estimate. The user could test the validity of the choice of the number of eigenvalues by increasing the number of eigenvalues slightly and comparing the maximum eigenvalue estimate obtained with the larger number of eigenvalues to the original maximum eigenvalue estimate obtained with the smaller number of eigenvalues. If the change in the two maximum eigenvalue estimates (larger versus smaller total number of eigenvalues) is small, then the original estimate for the number of eigenvalues is reasonably accurate.

As an alternative to directly specifying the number of eigenvalues to be computed, a convergence tolerance could be set on the change in the magnitude of the maximum eigenvalue as additional eigenvalues (Lanczos vectors) are computed. Let θ_{max_n} be the maximum eigenvalue calculated corresponding to n eigenvalues (Lanczos vectors), and let θ_{max_n+1} be the maximum eigenvalue corresponding to $n+1$ eigenvalues (Lanczos vectors). The eigenvalues would be computed until

$$\frac{|\,\theta_{max\_n+1} - \theta_{max\_n}\,|}{\theta_{max\_n+1}} \tag{3.15}$$

is less than some tolerance. (We will now refer to the value of Equation (3.15) as the *convergence measure*.) If we calculate the convergence measure for a sequence of maximum eigenvalues computed by the Lanczos method, we will not necessarily see a monotonic decrease in the convergence measure for all time. Typically, the convergence measure will initially show a monotonically decreasing value for an increasing n, and then the convergence measure will show a monotonically increasing value for an increasing n. (The phase in which the convergence measure increases is usually very small, about one to two iterations.) After the phase in which the convergence measure monotonically increases, there is then a phase in which the convergence measure begins a long (over many iterations) monotonic decrease. This behavior (decrease of the convergence measure, slight increase, then long monotonic decrease) is typical of Krylov methods, of which the Lanczos method is one. Using the convergence measure to determine the number of eigenvalues to be computed for a problem is a reasonable option, but some care must be taken in setting the tolerance for the convergence measure.

Both of the techniques discussed above are offered as a way to set the maximum number of eigenvalues required to obtain an accurate eigenvalue estimate for a model. These techniques are discussed further in Section 3.2.1.5.

3.2.1.5. Lanczos Parameters Command Block

```
BEGIN LANCZOS PARAMETERS <string>lanczos_name
  STARTING VECTOR = <string>STRETCH_X|STRETCH_Y|STRETCH_Z|
    ISOTHERMAL(ISOTHERMAL)
  INCREASE OVER STEPS = <integer>incr_int(5)
  NUMBER EIGENVALUES = <integer>num_eig(100)
  EIGENVALUE CONVERGENCE TOLERANCE = <real>converge_tol(0.5e-3)
  SMALL STRAIN = <real>small_strain(1.0e-3)
  VECTOR SCALE = <real>vec_scale(1.0e-5)
  SCALE FACTOR = <real>time_scale(1.0)
  UPDATE ON TIME STEP CHANGE = <real>tstep_change(0.10)
  UPDATE STEP INTERVAL = <integer>step_int(500)
  FORCE LANCZOS TIMESTEP = <string>ON|OFF(OFF)
END [LANCZOS PARAMETERS <string>lanczos_name]
```

If you use the Lanczos method to compute a critical time step, there should be only one `LANCZOS PARAMETERS` command block, and it should appear in the region. If you have a `LANCZOS PARAMETERS` command block, you should not specify a `NODE BASED TIME STEP PARAMETERS` or a `POWER METHODS PARAMETERS` command block. If you use the Lanczos method to compute the critical time step, the time step increase factor (default or user-specified) will be used to control the increase in the time step estimate; the time step scale factor (default or user-specified) will be used to scale the final time step. These factors are specified in the `PARAMETERS FOR PRESTO REGION` portion of a `TIME STEPPING BLOCK` in the `TIME CONTROL` command block.

The Lanczos method requires some type of starting vector. This is determined by the `STARTING VECTOR` command line. The various options available for this command line will generate a displacement vector that stretches your model in the x-, y-, or z-direction, or in all three directions (x, y, and z) at once. (`STRETCH_X` stretches the model in the x-direction, `STRETCH_Y` stretches the model in the y-direction, `STRETCH_Z` stretches the model in the z-direction, and `ISOTHERMAL` stretches the model in all three directions at once.) The displacement vector then serves as a basis for generating the starting vector r_0 in the initialization phase of the Lanczos method. The Lanczos method appears to be fairly insensitive to the choice of a starting vector. The default starting vector is the `ISOTHERMAL` option. You may encounter cases where the use of one of the other options for a starting vector—`STRETCH_X`, `STRETCH_Y`, or `STRETCH_Z`—may result in a slight increase in accuracy for the critical time step estimate for a given number of eigenvalues. These other options (`STRETCH_X`, `STRETCH_Y`, and `STRETCH_Z`) are offered for these special cases.

The `INCREASE OVER STEPS` command line determines how many steps, i.e., `incr_int`, are used to transition from an element-based critical time step estimate to the Lanczos-based estimate at the beginning of an analysis. The user may want to increase from the element-based estimate to the Lanczos-based estimate over a number of time steps if the difference between these two estimates is large. The value of `incr_int` for this command line defaults to 5.

As indicated in Section 3.2.1.4, some number of eigenvalues must be computed by the Lanczos

method to obtain a good estimate for the maximum eigenvalue for a given finite element model. As one option, the user can specify the number of eigenvalues to be computed by the Lanczos method by using the `NUMBER EIGENVALUES` command line. The default value for `num_eig` in this command line is 100. As an alternative, the user can specify a tolerance on the convergence measure defined in Equation (3.15). The tolerance is set with the `EIGENVALUE CONVERGENCE TOLERANCE` command line. The default value for the convergence tolerance, `converge_tol`, is 0.5×10^{-3}. (This default value may change in the future as we gain more experience with the Lanczos method.)

By default, the convergence measure (with the default value of 0.5×10^{-3}) is used to determine the number of eigenvalues to be computed. If neither the `EIGENVALUE CONVERGENCE TOLERANCE` command line nor the `NUMBER EIGENVALUES` command line appears in the `LANCZOS PARAMETERS` command block, then the convergence measure will be used to determine the number of eigenvalues. If the `EIGENVALUE CONVERGENCE TOLERANCE` command line appears in the command block with a specified tolerance other than the default value, then the number of eigenvalues computed will be determined by the convergence measure and the user-specified tolerance. If the `NUMBER EIGENVALUES` command line appears in the `LANCZOS PARAMETERS` command block, then the user-defined number of eigenvalues will be used to set the number of eigenvalues computed by the Lanczos method.

The recommended technique for determining the number of eigenvalues is to use the convergence measure option with the default tolerance. If you use this technique, neither the `NUMBER EIGENVALUES` command line nor the `EIGENVALUE CONVERGENCE TOLERANCE` command line should be included in the `LANCZOS PARAMETERS` command block.

As indicated in Section 3.2.1.3, it is necessary to scale one of the vectors (with a scale factor v_{sf}) in the Lanczos calculations so that it appears we are always working with a constant tangent stiffness each time we call the Lanczos method. There are two approaches for determining the scale factor. With the first approach, the user can specify the value of a small strain for a particular model by using the `SMALL STRAIN` command line. (As long as the internal force calculation inside the Lanczos method is working with a vector that approximates small-strain behavior, it will appear that we are working with a constant tangent stiffness inside the Lanczos method.) The value for v_{sf} is then computed based on your specification for what constitutes a small strain in the model. (The default value for `small_strain` is 1.0×10^{-3}.) With the second approach for determining the scale factor, the user can specify a value for v_{sf} directly by using the `VECTOR SCALE` command line. The default value for v_{sf}, the `vec_scale` parameter in that command line, is 1.0×10^{-5}. Many tests have established that this value for the scale factor, 1.0×10^{-5}, works well for a range of models encountered at Sandia National Laboratories. If the user wants to use the option of directly specifying the scale factor, it is possible to determine whether a particular scale factor is suitable for a particular problem. Take a scale factor v_{sf}, plus values on either side of it, say, $0.9 \times v_{sf}$ and $1.1 \times v_{sf}$. If all three of these scale factor values produce almost the same estimate for the critical time step for a particular model, then the value for v_{sf} meets the criterion for an acceptable scale factor.

By default, the small-strain approach (with the default value of 1.0×10^{-3}) is used to determine a scale factor. If neither the `SMALL STRAIN` command line nor the `VECTOR SCALE` command line appears in the `LANCZOS PARAMETERS` command block, then the small-strain value will be used

to calculate the scale factor. If the `SMALL STRAIN` command line appears in the command block with any value other than the default value *and* the `VECTOR SCALE` command line does not appear in the command block, then the vector scale will be determined by the user-specified value for the small strain. If the `VECTOR SCALE` command line does appear in the `LANCZOS PARAMETERS` command block, then the vector scale will be determined by the user-specified value for the vector scale.

The recommended way of determining the scale factor is to use the small-strain approach with the default small-strain value. If you follow this advice, neither the `SMALL STRAIN` command line nor the `VECTOR SCALE` command line should be included in the `LANCZOS PARAMETERS` command block.

As indicated previously, the scale factor must not be too small, as this will create round-off problems and give a bad estimate for the critical time step. If the scale factor is too large, we violate the above restriction of a constant tangent stiffness matrix.

The `SCALE FACTOR` command line sets the factor f_s in Equation (3.11). The value for f_s, i.e., `time_scale`, is set to 1.0.

The `UPDATE ON TIME STEP CHANGE` command line sets the value for Δt_{lim} in Equation (3.14). If the change in the element-based critical time step estimate as given by Equation (3.14) exceeds the value for Δt_{lim}, then the Lanczos method is called for a new estimate for the critical time step. The default value for Δt_{lim}, i.e., `tstep_change`, is 0.10.

The `UPDATE STEP INTERVAL` command line sets the number of step intervals at which the Lanczos method is called. If `step_int` is set to 1000, the Lanczos method will be called every one thousand steps to compute an estimate for the critical time step. (The default value of `step_int` is 500.) This control mechanism interacts with the control established by the `UPDATE ON TIME STEP CHANGE` command line. Suppose we have set `step_int` to 1000, and we have computed 800 steps since the last call to the Lanczos method. If Δt_{lim} has been set to 0.15 and we exceed this value at step 800, then the change in the element-based time step will result in the Lanczos method being called. The counter for keeping track of the number of step intervals since the last Lanczos computation will be reset to zero. The next call to the Lanczos method will occur again in one thousand steps, unless we again exceed the change in the element-based time step.

3.2.2. Power Method

The power method is a simple iterative technique that gives an estimate for the maximum eigenvalue. Consider the following form of the eigenvalue problem, which is useful for our explicit transient dynamics problem.

$$\mathbf{K}_T \mathbf{x}_i - \theta_i \mathbf{M} \mathbf{x}_i = 0 \tag{3.16}$$

Reformulate the problem as

$$[\mathbf{M}]^{-1} \mathbf{K}_T \mathbf{x}_i - \theta_i \mathbf{x}_i = 0\,. \tag{3.17}$$

Denote the matrix product $\mathbf{M}^{-1}\mathbf{K}_T$ as the matrix operator $\mathbf{A}$. The eigenvalue problem is now

$$\mathbf{A}\mathbf{x}_i - \theta_i \mathbf{x}_i = 0\,. \tag{3.18}$$

The maximum eigenvalue for the problem in Equation (3.18) can be computed with the following iterative process:

$$
\begin{aligned}
&\textit{for } i = 1, n \\
&\quad \mathbf{x}_i = \mathbf{A}\mathbf{x}_{i-1} \\
&\quad \mathbf{x}_i = \mathbf{x}_i / \parallel \mathbf{x}_i \parallel \\
&\quad \theta_i = [\mathbf{x}_i]^T \mathbf{A}\mathbf{x}_i \\
&\textit{end}
\end{aligned}
$$

The value θ_i is the ith estimate for the maximum eigenvalue. As n increases, the power method should yield a more accurate estimate for the maximum eigenvalue. Similar to the Lanczos method, the power method must iterate at some given time step to obtain an estimate for the maximum eigenvalue at that time step. The details for the above form of the power method are described in [1].

Like the Lanczos method, the power method requires the use of an internal force calculation. The power method, however, is simpler than the Lanczos method. The power method requires that one vector and one scalar be computed. The Lanczos method requires the computation of five vectors and two scalars. Further, the power method yields a direct estimate for the maximum eigenvalue, whereas the Lanczos method requires that the maximum eigenvalue be extracted from a tridiagonal matrix.

If the sequence of eigenvalues for an analysis problem is

$$
| \lambda_1 | > | \lambda_2 | \geq \cdots \geq | \lambda_n | , \tag{3.19}
$$

then the rate of convergence to the maximum eigenvalue depends on the ratio

$$
\frac{| \lambda_1 |}{| \lambda_2 |} . \tag{3.20}
$$

Depending on the magnitudes of $| \lambda_1 |$ and $| \lambda_2 |$, the rate of convergence could be rather slow. The power method, for a given problem, may not converge to as accurate an estimate for the maximum eigenvalue as quickly as the Lanczos method. We should be careful, however, in how we interpret this convergence property for the power method. For many of our problems, we probably have a large number of eigenvalues clustered slightly below the maximum eigenvalue. The power method may give a reasonable estimate for the maximum eigenvalue in a relatively small number of iterations. The power method may converge quickly to some eigenvalue estimate that is slightly below the maximum eigenvalue. Convergence to the maximum eigenvalue, in terms of accuracy to many digits, might then be very slow. However, one would still have an adequate estimate for the maximum eigenvalue to set a value for the critical time step.

One of the advantages of the power method is that we can use the last value for $\mathbf{x}_i$ from a previous call to the power method as a starting vector for our current call to the power method and converge quickly (for some problems) to an accurate estimate for the maximum eigenvalue. Consider the following scenario. We make an initial call to the power method using some arbitrary $\mathbf{x}_0$ starting vector. We obtain a good estimate for the maximum eigenvalue after n iterations. After some number of time steps, we again call the power method and we use $\mathbf{x}_n$ from the previous call to the

power method as our current starting vector. Because we are solving nonlinear problems, the matrix operator, **A** , has most likely changed from the time when we made the initial call to the power method. If the changes in the matrix operator are small, then $\mathbf{x}_n$ should be an excellent choice for the starting vector, and we should converge quickly to a good estimate for the maximum eigenvalue. This behavior, quick convergence to a good eigenvalue estimate using a prior value of $\mathbf{x}_n$ as a starting vector, has been observed in some sample problems. We discuss this issue of restarting the power method using a previous $\mathbf{x}_n$ in more detail in Section 3.2.2.1.

Obtaining a cost-effective and user-friendly implementation of the power method requires addressing the same issues discussed above for an automated implementation of the Lanczos method. First, we must determine some scheme to call the power method on an intermittent basis so that it is cost effective. The control mechanism for calling the Lanczos method on an intermittent basis can also be used for the power method. Second, the **x** vector must be scaled appropriately before it is used in the internal force calculations. (For the Lanczos method, we must scale the vector **p**.) Finally, we need to terminate the power method. A tolerance on the value in Equation (3.15) can be set to terminate the method. (Setting a tolerance on Equation (3.15) is the same approach used for the Lanczos method.)

3.2.2.1. Power Method with Constant Time Steps

The cost of making one iteration for the power method is approximately the cost of an internal force calculation. As indicated previously, the cost of one iteration for the Lanczos method (the calculation of a Lanczos vector) is also approximately the cost of an internal force calculation. Therefore, the equations developed in Section 3.2.1.1 are also applicable for determining when the power method can be used in a cost-effective manner. Recall that the equation

$$n_e = \frac{N_L}{1 - 1/r} \tag{3.21}$$

in Section 3.2.1.1 gives the number of time steps for the break-even point at which it becomes cost effective for the case of no contact to use the Lanczos method to reduce computational costs by overcoming the initial cost of the Lanczos calculations with the larger critical time step. In the above equation, N_L is the number of Lanczos iterations, and r is the ratio of the Lanczos-based critical time to the element-based time step estimate. This equation is directly applicable to the power method because the cost of one iteration in the power method is similar to the cost of one iteration in the Lanczos method. (The variable N_L becomes the number of iterations for the power method.)

Now let us return to the scenario in Section 3.2.2 where we look at restarting the power method with the last value for $\mathbf{x}_i$ from a previous call to the power method. We make an initial call to the power method using some arbitrary $\mathbf{x}_0$ starting vector. We obtain a good estimate for the maximum eigenvalue after n iterations. Our estimate for the critical time step based on the maximum eigenvalue gives us a value of 1.25 for r. After some number of time steps, we again call the power method, and we use $\mathbf{x}_n$ from the previous call to the power method as our current starting vector. In our second call to the power method, the power method converges to a good estimate for the maximum eigenvalue in two iterations. The break-even point where we recoup

the cost of the power method (in our second call) is ten time steps. If the power method converges to a good estimate for the maximum eigenvalue in three iterations, then the break-even point where we recoup the cost of the power method (in our second call) is fifteen time steps. There are some problems where we can obtain a good estimate for the maximum eigenvalue in two or three iterations when we use the last value for $\mathbf{x}_n$ from a previous call to the power method as our starting vector for the current call to the power method. For the case where we can converge to a good estimate for the maximum eigenvalue in two or three iterations, we can easily call the power method once every forty or fifty time steps and use the power method to improve the time step estimate in a cost-effective manner. Over a time span of forty or fifty time steps, matrix operator $\mathbf{A}$, as defined in Equation (3.18) may change by only a small amount for a large class of problems. If matrix operator $\mathbf{A}$ changes by a small amount, then using the last value of $\mathbf{x}_n$ from a previous call to the power method provides an excellent starting vector and gives us quick convergence. This ability of the power method to converge quickly by using previous information makes it a viable scheme for calculating the maximum eigenvalue and producing a critical time step estimate.

For a problem with contact, the equations derived for the Lanczos method (with contact) in Section 3.2.1.1 are also applicable to the power method. For the power method and the Lanczos method, the added computational cost of the contact calculations results in reaching a break-even point with a smaller number of iterations when compared to a case where there is no contact.

3.2.2.2. Controls for Power Method

Using the power method is similar to using the Lanczos method in that we must be able to perform a power method calculation and reuse the power method estimate for the critical time step in some way over a number of subsequent time steps. The approach for reusing a power method estimate for the critical time step over a number of time steps is the same as the approach for reusing a Lanczos-based estimate over a number of time steps as described in Section 3.2.1.2. The approach discussed in Section 3.2.1.2 presents a way to reuse a maximum eigenvalue time step estimate over a number of time steps so that we maintain a critical time step estimate that is close to the theoretical maximum value in between the calls to the maximum eigenvalue calculations. The approach discussed in in Section 3.2.1.2 makes use of the element-based critical time step estimate at each time step.

3.2.2.3. Scale Factor for Power Method

When the power method is called for a given time step, it must appear that the calculations are using a constant matrix operator, $\mathbf{A}$, for all iterations. The $\mathbf{A}$ operator is the product of the inverse of the mass matrix $\mathbf{M}^{-1}$ and the tangent stiffness matrix $\mathbf{K}_T$ for the problem. The mass matrix is a constant. Therefore, we must assure that we are working with what is effectively a constant tangent stiffness matrix for all iterations during a call to the power method. This is the same requirement that we encounter in the Lanczos method (see Section 3.2.1.3). For the power method, we use the internal force calculations to generate the product $\mathbf{A}\mathbf{x}_i$ by first computing the internal forces using $\mathbf{x}_i$, which gives us the product $\mathbf{K}_T\mathbf{x}_i$, and then multiplying the internal force

results by the inverse of the mass matrix. Any vector $\mathbf{x}_i$, as calculated by the power method, may be such that it represents large-strain behavior and moves the internal force calculations into a nonlinear regime. It is necessary to scale the $\mathbf{x}_i$ vectors so that the internal force calculations are in a small-strain regime, which makes it appear that we are working with a constant tangent stiffness matrix. The vectors $\mathbf{x}_i$ must be scaled so that they represent velocities associated with small-strain calculations. When properly scaled vectors are sent to the internal force calculation, the internal force calculation becomes a matrix $\times$ vector product with a constant tangent stiffness matrix for all iterations during a given call to the power method.

The scale factor for the $\mathbf{x}_i$ vectors, which we will designate as v_{sf}, must not be too small, as this will create round-off problems and give a bad estimate for the critical time step. If the scale factor is too large, we violate the above restriction of a constant tangent stiffness matrix.

There are two approaches for controlling the scale factor when the power method is used to compute the maximum eigenvalue. These approaches are discussed in Section 3.2.2.5.

3.2.2.4. Accuracy of Eigenvalue Estimate

To use the power method efficiently, it is best not to specify a set number of iterations for every instance when the power method is called. Instead, some tolerance should be set on the convergence measure defined in Equation (3.15). The initial call to the power method may take a significant number of iterations to produce a good estimate for the maximum eigenvalue. On subsequent calls to the power method, the number of iterations to converge to a good estimate for the maximum eigenvalue should be small when we use previous values of $\mathbf{x}$ as a starting vector.

For the power method, we rely on the convergence measure defined by Equation (3.15) to control the number of iterations. The user can set an upper limit on the number of iterations for the power method. Control of the number of iterations for the power method is discussed in more detail in Section 3.2.2.5.

3.2.2.5. Power Method Parameters Command Block

```
BEGIN POWER METHOD PARAMETERS <string>powermethod_name
  STARTING VECTOR = <string>STRETCH_X|STRETCH_Y|STRETCH_Z|
    ISOTHERMAL(ISOTHERMAL)
  INCREASE OVER STEPS = <integer>incr_int(5)
  NUMBER ITERATIONS = <integer>num_iter(150)
  EIGENVALUE CONVERGENCE TOLERANCE =
    <real>converge_tol(0.5e-3)
  SMALL STRAIN = <real>small_strain(1.0e-3)
  VECTOR SCALE = <real>vec_scale(1.0e-5)
  SCALE FACTOR = <real>time_scale(0.9)
  UPDATE ON TIME STEP CHANGE = <real>tstep_change(0.10)
  UPDATE STEP INTERVAL = <integer>step_int(50)
END [POWER METHOD PARAMETERS <string>powermethod_name]
```

If you use the power method to compute a critical time step, there should be only one

POWER METHOD PARAMETERS command block, and it should appear in the region. If you have a POWER METHOD PARAMETERS command block, you should not specify a command block for the node-based method or the Lanczos method. If you use the power method to compute the critical time step, the time step increase factor (default or user-specified) set in the TIME CONTROL command block will be used to control the increase in the time step estimate; the time step scale factor (default or user-specified) set in the TIME CONTROL command block will not be used. These factors are specified in the PARAMETERS FOR PRESTO REGION portion of a TIME STEPPING BLOCK in the TIME CONTROL command block.

The power method requires some type of starting vector. This is determined by the STARTING VECTOR command line. The various options available in this command line will generate a displacement vector that stretches your model in the x-, y-, or z-direction, or in all three directions (x, y, and z) at once. (STRETCH_X stretches the model in the x-direction, STRETCH_Y stretches the model in the y-direction, STRETCH_Z stretches the model in the z-direction, and ISOTHERMAL stretches the model in all three directions at once.) The displacement vector then serves as a basis for generating the starting vector x_0 in the initialization phase of the Lanczos method. The default starting vector is the ISOTHERMAL option. You may encounter cases where use of one of the other options for a starting vector—STRETCH_X, STRETCH_Y, or STRETCH_Z—may result in a slight increase in accuracy for the critical time step estimate for a given number of iterations. These other options (STRETCH_X, STRETCH_Y, and STRETCH_Z) are offered for these special cases.

The INCREASE OVER STEPS command line determines how many steps, incr_int, are used to transition from an element-based critical time step estimate to the power method estimate at the beginning of an analysis. The user may want to increase from the element-based estimate to the power method estimate over a number of time steps if the difference between these two estimates is large. The value of incr_int defaults to 5.

As indicated in Section 3.2.2.4, some number of iterations are required by the power method to obtain a good estimate for the maximum eigenvalue for a given model. The number of iterations is controlled by the tolerance set for the convergence measure given in Equation (3.15). The tolerance is set with the EIGENVALUE CONVERGENCE TOLERANCE command line. The default value for the convergence tolerance, converge_tol is 0.5×10^{-3}. (This default value may change in the future as we gain more experience with the power method.)

The user can set an upper limit on the number of iterations for the power method by using the NUMBER ITERATIONS command line. If the number of iterations in the power method exceeds the value set by num_iter, then the power method is terminated. Otherwise, the iterative process in the power method is terminated by the tolerance set on the convergence measure. The default value for num_iter is 150.

As indicated in Section 3.2.2.3, it is necessary to scale the vectors $\mathbf{x}_i$ (with a scale factor v_{sf}) in the power method calculations so that it appears we are always working with a constant tangent stiffness each time we call the power method. There are two approaches for determining the scale factor. With the first approach, the user can specify the value of a small strain for a particular model by using the SMALL STRAIN command line. (As long as the internal force calculation inside the power method is working with a vector that approximates small-strain behavior, it will appear that we are working with a constant tangent stiffness inside the power method.) The value

for v_{sf} is then computed based on your specification for what constitutes a small strain in the model. (The default value for `small_strain` is 1.0×10^{-3}.) With the second approach for determining the scale factor, the user can specify a value for v_{sf} directly by using the `VECTOR SCALE` command line. The default value for v_{sf}, the `vec_scale` parameter in that command line, is 1.0×10^{-5}. A number of tests have established that this value for the scale factor, 1.0×10^{-5}, works well for a range of models encountered at Sandia National Laboratories. If the user wants to use the option of directly specifying the scale factor, it is possible to determine whether a particular scale factor is suitable for a particular problem. Take a scale factor v_{sf}, plus values on either side of it, say, $0.9 \times v_{sf}$ and $1.1 \times v_{sf}$. If all three of these scale factor values produce almost the same estimate for the critical time step for a particular model, then the value for v_{sf} meets the criterion for an acceptable scale factor.

By default, the small-strain approach (with the default value of 1.0×10^{-3}) is used to determine a scale factor. If neither the `SMALL STRAIN` command line nor the `VECTOR SCALE` command line appears in the `POWER METHOD PARAMETERS` command block, then the default small-strain value will be used to calculate the scale factor. If the `SMALL STRAIN` command appears in the command block with any value other than the default value *and* the `VECTOR SCALE` command line does not appear in the command block, then the vector scale will be determined by the user-specified value for the small strain. If the `VECTOR SCALE` command line does appear in the `POWER METHOD PARAMETERS` command block, then the vector scale will be determined by the user-specified value for the vector scale.

The recommended way of determining the scale factor is to use the small-strain approach with the default small-strain value. Thus, if you follow this advice, neither the `SMALL STRAIN` command line nor the `VECTOR SCALE` command line should be included in the `POWER METHOD PARAMETERS` command block.

As indicated previously, the scale factor must not be too small, as this will create round-off problems and give a bad estimate for the critical time step. If the scale factor is too large, we violate the above restriction of a constant tangent stiffness matrix.

The `SCALE FACTOR` command line sets the factor f_s in Equation (3.11). The value for f_s, i.e., `time_scale` is set to 0.9. More experience with the power method will help us determine whether a slightly larger value of f_s can be used as a default.

The `UPDATE ON TIME STEP CHANGE` command line sets the value for Δt_{lim} in Equation (3.14). If the change in the element-based critical time step estimate as given by Equation (3.14) exceeds the value for Δt_{lim}, then the power method is called for a new estimate for the critical time step. The default value for Δt_{lim}, i.e., `tstep_change`, is 0.10.

The `UPDATE STEP INTERVAL` command line sets the number of step intervals at which the power method is called. If `step_int` is set to 40, the power method will be called every forty steps to compute an estimate for the critical time step. (The default value of `step_int` is 50.) This control mechanism interacts with the control established by the `UPDATE ON TIME STEP CHANGE` command line. Suppose we have set `step_int` to 40, and we have computed thirty-five steps since the last call to the power method. If Δt_{lim} has been set to 0.15 and we exceed this value at step 35, then the change in the element-based time step will result in the power method being called. The counter for keeping track of the number of step

intervals since the last power method computation will be reset to zero. The next call to the power method will then occur again in forty steps, unless we again exceed the change in the element-based time step.

3.2.3. Node-Based Method

Now that we have developed schemes to make the Lanczos method and power method cost-effective tools for estimation of the critical time step, let us examine the node-based scheme.

The node-based method in Presto will give an estimate for the critical time step that is greater than or equal to the element-based estimate. It may or may not give an estimate for the time step that is close to the maximum value associated with the maximum eigenvalue for the problem. In general, we assume the node-based method will give us an estimate larger than the element-based estimate, but not significantly larger.

If the node-based scheme is used to determine the critical time step for a block of uniform elements (all the same material), then the estimate from the node-based method will be the same as that for the element-based estimate. The node-based estimate begins to diverge from (become larger than) the element-based estimate as the differences in aspect ratios of the elements attached to a node become larger. We can assume, therefore, that if the stiffest part of our structure has a relatively uniform mesh, the node-based method will not give a significantly larger estimate than the element-based method.

The node-based method costs only a fraction of an internal force calculation. However, the node-based method makes use of element time step estimates. Therefore, to use the node-based method, one must also do the element-based method. The cost of the node-based method is in addition to the element-based method. The modest increase in the critical step estimate from the node-based method is unlikely to offset the added cost of the node-based method.

To make the node-based method cost-effective, we can use the same methodology that is employed for the Lanczos method. Let t_b be the estimate for the critical time step from the node-based method. We define the ratio of the node-based estimate to the element-based estimate as

$$t_r = \frac{\Delta t_b}{\Delta t_e} . \tag{3.22}$$

This ratio is then used to scale subsequent element-based estimates for the critical time step. If $\Delta t_{e(n)}$ is the nth element-based critical time step after the time step where the node-based calculations are performed, then the nth time step after the node-based calculations, $\Delta t_{(n)}$, is

$$\Delta t_{(n)} = t_r \Delta t_{e(n)} . \tag{3.23}$$

The ratio t_r is used until the next call to the node-based method. The next call to the node-based method is controlled by one of two mechanisms described in the Lanczos discussion. The node-based method is called after a set number of times or a significant change in the element-based estimate for the critical time step.

3.2.3.1. Node-Based Parameters Command Block

```
BEGIN NODE BASED TIME STEP PARAMETERS <string>nbased_name
  INCREMENT INTERVAL = <integer>incr_int(0)
  STEP INTERVAL = <integer>step_int(1)
  TIME STEP LIMIT = <real>step_lim(0.10)
END [NODE BASED TIME STEP PARAMETERS <string>nbased_name]
```

If you use the node-based method to compute a critical time step, there should be only one `NODE BASED TIME STEP PARAMETERS` command block, and it should appear in the region. If you use the node-based method to compute the critical time step, you should set the time step scale factor to 1.0 and the time step increase factor to a large number (5.0 is an acceptable value for the time step increase factor). These factors are specified in the `PARAMETERS FOR PRESTO REGION` portion of a `TIME STEPPING BLOCK` in the `TIME CONTROL` command block. If you have a `NODE BASED TIME STEP PARAMETERS` command block, you should not specify a `LANCZOS PARAMETERS` command block.

The `INCREMENT INTERVAL` command line determines how many steps, `incr_int`, are used to transition from an element-based critical time step estimate to the node-based estimate at the beginning of an analysis. The user may want to increase from the element-based estimate to the node-based estimate over a number of time steps if the difference between these two estimates is large. The value of `incr_int` defaults to 0 so the node based time step is used immediately at the beginning of the analysis.

The `STEP INTERVAL` command line sets the number of step intervals at which the node-based method is called. If `step_int` is set to 1000, the node-based method will be called every one thousand steps to compute an estimate for the critical time step. (The default value of `step_int` is 1.) Increasing the step interval can improve performance, but depending on the problem could run the risk of sending the time step super-critical. This control mechanism interacts with the control established by the `TIME STEP LIMIT` command line. Suppose we have set `step_int` to 1000, and we have computed 800 steps since the last call to the node-based method. If Δt_{lim} has been set to 0.15 and we exceed this value at step 800, then the change in the element-based time step will result in the node-based method being called. The counter for keeping track of the number of step intervals since the last node-based computation will be reset to zero. The next call to the node-based method will occur again in one thousand steps, unless we again exceed the change in the element-based time step.

The `TIME STEP LIMIT` command line sets the value for Δt_{lim},

$$\Delta t_{lim} = \frac{|\Delta t_{e(n)} - \Delta t_e|}{\Delta t_e} \tag{3.24}$$

This is exactly the same as Equation (3.14) presented eleven pages earlier. If the change in the element-based critical time step estimate Δt_{lim}, exceeds the value for Δt_{lim}, then the node-based method is called for a new estimate for the critical time step. The default value for Δt_{lim}, i.e., `step_lim`, is 0.10.

3.3. Mass Scaling

3.3.1. What is Mass Scaling?

Mass scaling allows for arbitrarily increasing the mass of certain nodes to increase the global estimate of the critical time step; therefore, mass scaling is used in conjunction with a node-based time step. The nodes where the mass is increased must be associated with elements that control the global time step; increasing the mass at a node of an element effectively raises the critical time step estimate for that element.

Mass scaling only adjusts the *mass* at the nodes. It does not, however, modify the *density* value used in the element calculations, even though the net effect of mass scaling makes it appear as though it has. Mass scaling does not affect the load applied by gravity boundary conditions, centripetal force boundary conditions, or the mass properties reported in the log file.

Mass scaling can be useful in these circumstances, such as

- quasi-static or rigid-body motion: If the model or part of a model is undergoing what is nearly quasi-static or rigid-body motion, adding mass may have little effect on the end result.
- disparate element sizes: A model may contain groups of elements that are much smaller than others. For example, in screws or gears that are modeled in detail, elements in the screw threads or gear teeth could be much smaller than elements in other portions of the model. If the dynamic response of these small parts is unimportant relative to the rest of the model, adding mass to them might not affect the quantities of interest.
- increasing the critical time step for "unimportant" model components: In some problems, a relatively unimportant mesh feature may control the critical time step. Consider, for example, a car crash simulation in which the bumper is the first part to strike an object. The crumpling deformation could reduce the time step in some elements in the bumper, potentially controlling the global critical time step. At a later time in the analysis, however, the effect of the bumper on the overall crash dynamics may not be significant; thus, mass scaling could be applied to ensure that the noncritical bumper does not control the global time step.



Warning: Use of mass scaling *will* introduce unbounded and potentially unpredictable errors into the analysis and should thus be used with caution. The analyst must use discretion to determine whether mass scaling may be used in a way that does not distort the results of interest.

3.3.2. Mass Scaling Command Block

```
BEGIN MASS SCALING
  #
  # node set commands
  NODE SET|NODESET = <string list>nodelist_names
  SURFACE = <string list>surface_names
```

```
  BLOCK = <string list>block_names
  ELEMENT = <string list>element_names
  NODE = <string list>element_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list> nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  TARGET TIME STEP = <real>target_time_step
  ALLOWABLE MASS INCREASE RATIO = <real>mass_increase_ratio
  #
  # additional command
  ACTIVE PERIODS = <string list>periods
  INACTIVE PERIODS = <string list>periods

  UPDATE INTERVAL = INITIALIZATION_ONLY | EVERY_STEP | AUTOMATIC
    (AUTOMATIC)
END MASS SCALING
```

The `MASS SCALING` command block controls mass scaling for a specified set of nodes and appears in the adagio region scope. This command block contains one or more command lines to specify the node set. It also contains two command lines that determine how the actual mass scaling will be applied to the nodes in the node set. In addition to the command lines in the two command groups, there are two additional command lines: `ACTIVE PERIODS` and `INACTIVE PERIODS`. These command lines are used to activate or deactivate the mass scaling for specified time periods.

Multiple `MASS SCALING` command blocks can exist to apply different criteria to different portions of the mesh at different times. For any given set of `MASS SCALING` command blocks, mass will only be added to a node if doing so will allow increasing the global time step. The amount of artificial mass added to a node will vary in time as the mesh deforms and moves. The added mass computation is recomputed every time the node-based time step estimate is recomputed.

NOTE: Mass scaling *must* be used in conjunction with the node-based time step estimation method. Consult Section 3.2.3 for a description of the node-based method for estimating the critical time step.

Following are descriptions of the different command groups:

3.3.3. Node Set Commands

The `node set commands` portion of the `MASS SCALING` command block specifies the nodes associated with the boundary condition. This portion of the command block can include some combination of the following lines:

```
NODE SET|NODESET = <string list>nodelist_names
```

```
SURFACE = <string list>surface_names
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list> nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 7.1.1 for more information about the use of these command lines for creating a set of nodes for mass scaling. There must be at least one `NODE SET|NODESET`, `SURFACE`, `BLOCK`, or `INCLUDE ALL BLOCKS` command line in the command block.

3.3.3.1. Mass Scaling Commands

The `MASS SCALING` command block may contain either a

```
TARGET TIME STEP = <real>target_time_step
```

command line or an

```
ALLOWABLE MASS INCREASE RATIO = <real>mass_increase_ratio
```

command line, or both of these command lines can appear in the input block.

The `TARGET TIME STEP` command lines sets the maximum time step for a set of nodes. The parameter `target_time_step` is the maximum time step for all the nodes specified in the command block.

The `ALLOWABLE MASS INCREASE RATIO` command line sets an upper limit on the mass scaling at a node. The value specified for `mass_increase_ratio` limits the ratio of the mass at a node, as set by mass scaling, to the original mass at the node. (The original mass of the node is determined only by the element contributions.) This ratio must be a factor greater than or equal to 1. If m_s is the scaled mass at a node and m_0 is the original mass at the node due only to element contributions, then the ratio $m_s \div m_0$ will not exceed the value of `mass_increase_ratio`.

Mass scaling will add mass to nodes until the target time step is reached, the mass added to some node reaches the allowable mass increase ratio, or the current set of nodes no longer controls the global analysis time step.

The amount of mass added due to mass scaling is stored in the nodal variable `mass_scaling_added_mass`. This variable can be output and post-processed to determine how much mass is being added at a given time. See Section 9.3.1 regarding the output of nodal variables to a results file.

3.3.3.2. Additional Commands

The `ACTIVE PERIODS` or `INACTIVE PERIODS` command lines can optionally appear in the `MASS SCALING` command block:

```
ACTIVE PERIODS = <string list>periods

INACTIVE PERIODS = <string list>periods
```

These command lines determine when mass scaling is active. See Section 2.6 for more information about these command lines.

The `UPDATE INTERVAL` command defines how often mass scaling is updated. The default `AUTOMATIC` updates the mass scaling when the time step has changed significantly (greater than 5%) since the last mass update. Mass shifting can be expensive so this automatic option avoids changing the mass at too high of a frequency. The `EVERY_STEP` option recalculates the mass scaling every single step based off of the current model stiffness. Use of this option will ensure the model run at exactly the specified time step. The `INITIALIZATION_ONLY` option will calculate the mass scaling once at the first time step. The nodal mass scaling will not change after this point. This option can be useful to accept and bound how much mass will be added to the model once and not have the possibility of the mass changing later in time.

References

[1] G.H. Golub and C.F. Van Loan. *Matrix Computations, Third Edition*. The John Hopkins University Press, Baltimore, MD, 1996.

[2] L.N. Trefethen and D. Bau. *Numerical Linear Algebra*. SIAM, 1997.

[3] T.J.R. Hughes. *The Finite Element Method–Linear Static and Dynamic Finite Element Analysis*. Prentice-Hall, Inc, Englewood Cliffs, NJ, 1987.



4. IMPLICIT SOLVER AND TIME STEP CONTROL

This chapter discusses the commands used to control the solver, time stepping, and dynamics parameters for implicit calculations. The commands and features documented in this chapter are applicable only for quasistatic and implicit dynamic analyses. For information on time stepping commands for explicit dynamic analyses, see Chapter 3.

The solution strategy used in Sierra/SM is based on iterative solution techniques and builds on the capabilities pioneered in the JAS3D code [1]. Sierra/SM uses a nonlinear preconditioned conjugate gradient (CG) algorithm [2] to iteratively find a solution that satisfies equilibrium within a user-specified error tolerance at each load step.

The nonlinear CG algorithm iterates to a solution by computing a series of search direction vectors that are successively added to the trial velocity vector. In each iteration, the residual is multiplied by a preconditioning matrix to obtain a gradient direction. The gradient direction is then used to compute the next search direction in a way that ensures that each new search direction is orthogonal to the previous search directions. A line search is used to compute a scaling factor that when applied to the search direction results in a minimized residual.

The performance of the CG algorithm is highly dependent on the choice of preconditioner. Using the inverse of the full tangent stiffness matrix as a preconditioner minimizes the number of CG iterations required but can be computationally expensive. Alternatively, the inverse of a diagonalized stiffness matrix can be used. This requires much lower computational and memory resources, per iteration, but may require many more iterations.

Sierra/SM provides several preconditioner options that can be categorized in two general types. The first type is the nodal preconditioner. Several different variants are available including the diagonal preconditioners that were available in JAS3D. The second type is the full tangent preconditioner. This option is typically implemented through use of the parallel-scalable FETI linear solver.

Conditioning of a problem can also be improved by using a multilevel solver. Rather than directly solving the difficult, and potentially ill-posed, full problem with the CG solver, Sierra/SM forms a series of “model problems” and solves them with the core CG solver. Features of the model that make it difficult are either adjusted or held constant for a model problem. After each model problem is solved, the multilevel solver updates the quantities that were controlled. The process of forming model problems and updating is iterative, as with the core solver, and convergence must be achieved both in the core solver and in the multilevel controls.

Four types of controls are available for the multilevel solver. They are discussed at a high level in the next section.

This chapter discusses the commands for specifying solver options for both quasistatic and implicit dynamic problems. The CG solver can either be used alone or as the core solver within a

multilevel solver. Section 4.1 presents the characteristics and structure of the SOLVER command block, which contains all solver-related commands within the ADAGIO REGION command block. Section 4.2 discusses the CG command block, which controls the nonlinear preconditioned CG solver. An option in the CG command block is to use a full tangent preconditioner. The command block to control the behavior of this option is discussed in Section 4.3. The recommended linear solver for use with the full tangent preconditioner is FETI, whose command block is presented in Section 4.4. Section 4.9 describes the predictors, which are used to generate an initial trial solution for a load step or for a multilevel-solver model problem. Time stepping control for both quasistatic (where time serves as a pseudo-time load step parameter) and implicit dynamic problems is described in Section 4.10. Section 4.11 discusses techniques to perform implicit dynamic solutions. Finally, references are provided at the end of the chapter.

4.1. Multilevel Solver

Sierra/SM allows the conjugate gradient (CG) solver to be used either by itself or as the core solver within a multilevel solver. The multilevel solver improves the ability of the core CG solver to solve poorly conditioned problems by holding fixed some variables that would ordinarily be free to change in a full nonlinear iteration. With a multilevel solver CG iterations are performed while a variable is held fixed. After convergence is obtained with the controlled variable fixed, the control variable is updated, internal and external forces are re-computed, and a new residual calculation is performed. Moreover, the multilevel solver algorithms can be used in parallel and/or layered to treat multiple variables by assigning them to multiple controls. Convergence on any particular level requires that all levels leading up to that level be converged. Some Sierra/SM capabilities can be used only in conjunction with the appropriate multilevel solver. For example, a contact problem generally requires a control contact solver level, and a problem modeling material fracture with element death requires a control failure solver level.

As an example, consider the case illustrated in Figure 4-1 of a solver with two controls defined, one at a level above the other. Once all control variables have been initially established for a load step, CG iterations are performed until convergence is obtained on the core "model problem." At this point, the first variable controlled above the CG solve, also known as the "level 1" control, is adjusted to minimize its residual. This process is known as a "level 1 update." After the level 1 update, the model problem is solved again, and this process is repeated until the core solver and the variable controlled at level 1 have converged. Once a solution has been obtained at level 1, the variable controlled at level 2 is updated. After each level 2 update, the level 1 problem must be solved again. For convergence in the multilevel solver, all updates starting with the lowest level and proceeding to the highest level must be in equilibrium. Sierra/SM allows up to two control levels, and multiple controls can be defined at each level (though not with control failure, see Section 4.7).

Sierra/SM has four types of controls that can be used within the multilevel solver: control contact, control stiffness, control failure, and control damped solve.

Control contact is used to keep the set of nodes in contact constant during the solution of a model problem. Contact searches are performed during the multilevel solver updates. Control contact must be used for all problems that include a contact definition block, with the one exception of kinematic tied contact. If no control contact block is defined for problems that require one, a

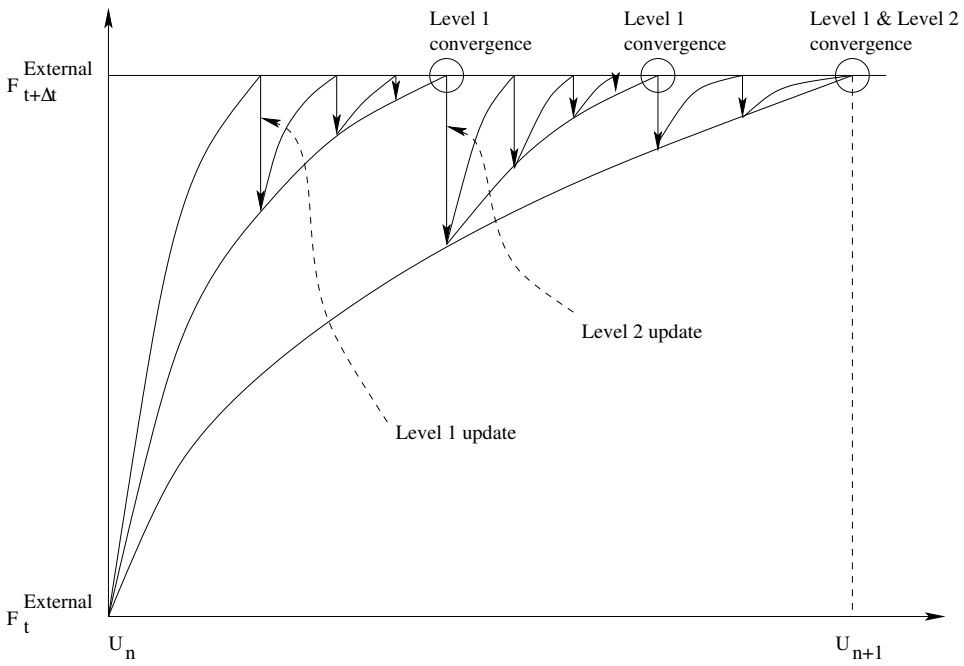


Figure 4-1. Reaching convergence with controls and the multilevel solver.

default control contact will be used. Control contact is not required for models that have only kinematic constraints, such as kinematic tied contact and MPCs. Control contact is documented in detail in Section 4.5.

Control stiffness is used to improve the conditioning of models that contain widely varying stiffness in different modes of material response. Such differences naturally exist for models with nearly incompressible materials where the bulk behavior is much stiffer than the shear behavior, for models with orthotropic and/or anisotropic materials that have much higher stiffness in preferred material directions, and for models containing several materials that are vastly different in their overall stiffness. Control stiffness works by scaling a given material response up (stiffening) or down (softening) to create a set of better conditioned model problems. For nearly incompressible materials, for instance, the bulk stress increments may be softened and/or the shear stress increments may be stiffened. For oriented materials, the stress increments in certain material directions are softened to create a material response that is more isotropic in nature. Finally, for the case where a material is much stiffer than the surrounding materials, both the bulk and shear stress increments are softened to create a response that has a stiffness much closer to that of the adjacent materials. Ultimately, as the control stiffness model problems progress, the true material response is recovered by building up stresses that correspond to the true material response. Special material models are required to use this capability, and Sierra/SM provides a number of material models that are designed to work with this capability. Control stiffness is documented in detail in Section 4.6.

Control failure is used to improve the conditioning of models that involve element death due to failure defined within a material model or to keep a consistent state for other failure criteria. Control failure used with material criterion element death limits the failure only to the single most

critical element during each control failure iteration. Iterations of control failure continue until no additional elements are marked for failure. Control failure is documented in detail in Section 4.7.

Control damped solve is used on problems that have initial rigid body modes or other problems that are inherently difficult to solve with a quasistatic implicit solver alone. Some examples of problems that may benefit from this control are problems that require contact to be seated before preload, implicit contact problems where rebound or bounce occur, difficult implicit contact problems in general, and problems with softening materials. Control damped solve is documented in detail in Section 4.8.

The following is the basic structure of the `SOLVER` command block:

```
BEGIN SOLVER

  # CG solver commands:

  BEGIN CG
    # ... see Sections 4.2, 4.3, and 4.4
  END [CG]

  # Multilevel control commands:

  BEGIN CONTROL CONTACT [<string>contact_name]
    # ... see Section 4.5
  END [CONTROL CONTACT <string>contact_name]

  BEGIN CONTROL STIFFNESS [<string>stiffness_name]
    # ... see Section 4.6
  END [CONTROL STIFFNESS <string>stiffness_name]

  BEGIN CONTROL FAILURE [<string>failure_name]
    # ... see Section 4.7
  END [CONTROL FAILURE <string>failure_name]

  BEGIN CONTROL DAMPED SOLVE [<string>ds_name]
    # ... see Section 4.8
  END [CONTROL DAMPED SOLVE <string>ds_name]

  # Predictor commands:

  BEGIN LOADSTEP PREDICTOR
    # ... see Section 4.9.1
  END [LOADSTEP PREDICTOR]

  LEVEL 1 PREDICTOR = <string>NONE|DEFAULT(DEFAULT)
  # see Section 4.9.2
```

```
END [SOLVER]
```

The SOLVER command block contains all solver-related commands, and must be present in the ADAGIO REGION command block. It is used regardless of whether the CG solver should be used alone or as a core solver in the multilevel solver. The CG command block (described in Section 4.2) is required for all models. If no multilevel controls are desired, the commands related to those controls are omitted from the SOLVER block.

In addition to the CG command block, the SOLVER command block contains command blocks that describe the controls used.

Aside from the command blocks for the core solver and the controls, the SOLVER block contains the LOADSTEP PREDICTOR command block, which is described in Section 4.9.1, and the LEVEL 1 PREDICTOR command line, which is described in Section 4.9.2.

Convergence tolerances for the CONTROL CONTACT, CONTROL STIFFNESS, and CONTROL DAMPED SOLVE solver levels are set within those command blocks while the tolerance of the core CG solver is set within the CG command block. Rather than using a convergence tolerance, the CONTROL FAILURE block is considered converged when no new elements are marked for failure during an iteration or when the maximum iterations is reached.

The commands within the CG command block are used in the same way whether the multilevel solver is used. It is, however, important to consider the role of tolerances within the multilevel solver. The convergence of the problem is controlled by the convergence of the highest solver level. For the multilevel solver to converge well, it is important for the core solver (or lower level of the solver, in the case of multiple levels) to reduce the residual by a certain amount during each model problem solution. For this reason, it is recommended that the solution tolerances be set an order of magnitude tighter for each lower level solver. The MINIMUM RESIDUAL IMPROVEMENT command in the CG command block is helpful for ensuring that the residual is reduced by the core solver during model problems, even though the residual might already be within the convergence tolerances (see Section 4.2.1).

4.1.1. Common Multilevel Solver Commands for All Controls

```
BEGIN SOLVER

  # CG solver commands:

  BEGIN CG

    # Parameters for CG
    # ... see Sections 4.2, 4.3, and 4.4

    # Diagnostic output commands, off by default:

    ITERATION PLOT = <integer>iter_plot
      [DURING <string list>period_names]
```

```
    ITERATION PLOT OUTPUT BLOCKS  = <string list>plot_blocks

  END [CG]

  #General Control commands:

  BEGIN CONTROL CONTACT|STIFFNESS|FAILURE|{DAMPED SOLVE} \#
    [<string>control_name]

    # ... see Sections 4.5, 4.6, 4.7, and 4.8

    # Level selection command:

    LEVEL = <integer>control_level

    # Diagnostic output commands, off by default:

    ITERATION PLOT = <integer>iter_plot
      [DURING <string list>period_names]
    ITERATION PLOT OUTPUT BLOCKS  = <string list>plot_blocks

  END [CONTROL CONTACT|STIFFNESS|FAILURE|{DAMPED SOLVE} \#
    [<string>control_name]]

END [SOLVER]
```

Displayed in this section are common commands to all controls that can be used for the multilevel solver. Other commands may be used for specific controls, but those commands are not listed in this section. For commands specific to `CONTROL CONTACT`, see section Sections 4.5. For commands specific to `CONTROL STIFFNESS`, see section 4.6. For commands specific to `CONTROL FAILURE`, see section 4.7. For commands specific to `CONTROL DAMPED SOLVE`, see section 4.8.

```
LEVEL = <integer>control_level(1)
```

The `LEVEL` command line specifies the level in the multilevel solver at which contact, stiffness, and/or failure is controlled. The `level` parameter can have a value of either 1 or 2, with 1 being the default for all controls.

This command is used when multiple controls (e.g., control contact and control stiffness) are active in the multilevel solver. It is permissible for multiple controls to exist at a given level. In order to have a control at level 2, another control at level 1 is required. If control failure is defined, it should be the only control at the outermost (highest numerical value) level.

```
ITERATION PLOT = <integer>iter_plot
  [DURING <string list>period_names]
ITERATION PLOT OUTPUT BLOCKS  = <string list>plot_blocks
```

The iteration plot command lines can be used to output diagnostic information for each of the controls. The `ITERATION PLOT` command line allows plots of the current state of the model to be written to the output database during the control updates at a frequency specified by the value of `iter_plot`. The plots are output with fictitious time steps bounded by the step limits. Every time such a plot is generated, a message is sent to the log file. By default no iteration plots are generated. This feature is useful for debugging but should be used with care to avoid excessive output.

The output produced by iteration plots is sent by default to every active results output file specified by a `RESULTS OUTPUT` command block. The `ITERATION PLOT OUTPUT BLOCKS` command line can optionally be used to supply a list of output command block names to which iteration plots should be written. Each of the output block names listed in `plot_blocks` must match the name of a `RESULTS OUTPUT` command block (see Section 9.3.1). The `ITERATION PLOT OUTPUT BLOCKS` command line can be useful for directing iteration plots to separate output files so that these plots are separated from the converged solution output. It can also be used to send output plots for different levels of the multilevel solver to different files.

4.2. Conjugate Gradient Solver

The core nonlinear preconditioned conjugate gradient solver is controlled through the `CG` command block. This must be nested within a `SOLVER` command block, regardless of whether multilevel controls are used or not.

```
BEGIN CG
  #
  # convergence commands
  # Default is not defined.
  TARGET RESIDUAL = <real>target_resid
    [DURING <string list>period_names]
  TARGET RELATIVE RESIDUAL = <real>target_rel_resid(1.0e-4)
    [DURING <string list>period_names]
  # Default is 10 times target residual
  # or 1e20 if a multilevel solver is defined
  # and control failure is not defined.
  ACCEPTABLE RESIDUAL = <real>accept_resid
    [DURING <string list>period_names]
  # Default is 10 times target relative residual
  # or 1e20 if a multilevel solver is defined
  # and control failure is not defined.
  ACCEPTABLE RELATIVE RESIDUAL = <real>accept_rel_resid
    [DURING <string list>period_names]
  # Default is not defined.
  REFERENCE = EXTERNAL|INTERNAL|BELYTSCHKO|RESIDUAL|ENERGY
    (EXTERNAL) [DURING <string list>period_names]
  MINIMUM REFERENCE = NONE|FRACTION_OF_INITIAL(NONE) <real>min_ref(0.0)
    [DURING <string list>period_names]
  RESIDUAL NORM TYPE = ALL|TRANSLATION|SCALE_RB_ROTATIONS
    (ALL) [DURING <string list>period_names]
  MINIMUM RESIDUAL IMPROVEMENT = <real>resid_improvement
    [DURING <string list>period_names]
  MINIMUM ITERATIONS = <integer>min_iter(0)
    [DURING <string list>period_names]
  RESIDUAL ROUNDOFF TOLERANCE = <real>(1.0e-15)
  #
  # default = 100 for tangent preconditioner.
  # default = max(NumNodes,1000) for all other preconditioners.
  MAXIMUM ITERATIONS = <integer>max_iter
    [DURING <string list>period_names]
  #
  # preconditioner commands
  PRECONDITIONER = BLOCK|BLOCK_INITIAL|DIAGONAL|
    ELASTIC|IDENTITY|PROBE|TANGENT(ELASTIC) [<real>scaling_factor]
  PRECONDITIONER ITERATION UPDATE = <integer>iter_update
  BALANCE PROBE = <integer>balance_probe(1)
```

```
    [DURING <string list>period_names]
  NODAL PROBE FACTOR = <real>probe_factor(1.0e-6)
    [DURING <string list>period_names]
  NODAL DIAGONAL SCALE = <real>nodal_diag_scale(0.0)
    [DURING <string list>period_names]
  NODAL DIAGONAL SHIFT = <real>nodal_diag_shift(0.0)
    [DURING <string list>period_names]
  BEGIN FULL TANGENT PRECONDITIONER
    # ... see Section 4.3
  END [FULL TANGENT PRECONDITIONER]
  #
  # line search command, default is secant
  LINE SEARCH ACTUAL|TANGENT [DURING <string list>period_names]
  LINE SEARCH SECANT [<real>scale_factor]
  LINE SEARCH BACKTRACK [<real>min_step(1.0e-3)
                         <real>reduction(0.95)
                         <real>init_step(1.0)
                         <enum>NONE|SECANT_INIT(NONE)
                         <real>dispinc(infinity)]
  LINE SEARCH TYPE MULTI = <real>num_domain(2)
    [DURING <string list>period_names]
  #
  # diagnostic output commands
  ITERATION PRINT = <integer>iprint(25)
    [DURING <string list>period_names]
  # off by default.
  ITERATION PLOT = <integer>iplot
    [DURING <string list>period_names]
  ITERATION PLOT OUTPUT BLOCKS = <string list>plot_blocks
  #
  # cg algorithm commands
  ITERATION RESET = <integer>iter_reset(infinity)
    [DURING <string list>period_names]
  ORTHOGONALITY MEASURE FOR RESET = <real>ortho_reset(0.5)
    [DURING <string list>period_names]
  RESET LIMITS <integer>iter_start <integer>iter_reset
    <real>reset_growth <real>reset_orthogonality
    [DURING <string list>period_names]
  BETA METHOD = FletcherReeves|PolakRibiere|Newton|
    PolakRibierePlus(PolakRibiere)
    [DURING <string list>period_names]
  # line search step length bounds
  MINIMUM STEP LENGTH = <real>min_step(-infinity)
    [DURING <string list>period_names]
  MAXIMUM STEP LENGTH = <real>max_step(infinity)
    [DURING <string list>period_names]
END [CG]
```

Sections 4.2.1 through 4.2.4 describe the components of the CG command block.

4.2.1. Convergence Commands

```
# Default is not defined.
TARGET RESIDUAL = <real>target_resid
  [DURING <string list>period_names]
TARGET RELATIVE RESIDUAL = <real>target_rel_resid(1.0e-4)
  [DURING <string list>period_names]
ACCEPTABLE RESIDUAL = <real>accept_resid(1.0e-3)
  [DURING <string list>period_names]
# Default is 10 times target relative residual
# or 1e20 if a multilevel solver is defined
# and control failure is not defined.
ACCEPTABLE RELATIVE RESIDUAL = <real>accept_rel_resid
  [DURING <string list>period_names]
# Default is not defined.
REFERENCE = EXTERNAL|INTERNAL|BELYTSCHKO|RESIDUAL|ENERGY(EXTERNAL)
  [DURING <string list>period_names]
MINIMUM REFERENCE = NONE|FRACTION_OF_INITIAL(NONE) <real>min_ref(0.0)
  [DURING <string list>period_names]
RESIDUAL NORM TYPE = ALL|TRANSLATION|SCALE_RB_ROTATIONS(ALL)
  [DURING <string list>period_names]
MINIMUM RESIDUAL IMPROVEMENT = <real>resid_improvement
  [DURING <string list>period_names]
MINIMUM ITERATIONS = <integer>min_iter(0)
  [DURING <string list>period_names]
# default = 100 for tangent preconditioner
# default = max(NumNodes,1000) for all other preconditioners.
MAXIMUM ITERATIONS = <integer>max_iter
  [DURING <string list>period_names]
RESIDUAL ROUNDOFF TOLERANCE = <real>1.0e-15)
  [DURING <string list>period_names]
```

The nonlinear preconditioned CG solver iterates to decrease the residual until the residual is deemed to be sufficiently small, based on user-specified convergence criteria. The command lines listed above are placed in the CG command block and used to control these convergence criteria.

Solver convergence is measured by computing the L^2 norm of the residual and comparing that residual norm with target convergence criteria specified by the user. Convergence can be monitored either directly in terms of the residual norm, or in terms of a relative residual, which is the residual norm divided by a reference quantity that is indicative of the current loading conditions on a model. Basing convergence on the relative residual is often helpful because doing so ensures that the convergence target is meaningful for the model. There are some situations, however, when it is better to check for convergence based on the absolute residual rather than on the relative residual. Checking the absolute residual is especially helpful when the model passes

through a stage during which there are no loads. The actual residual and the relative residual can be used for checking convergence at the same time.

All the convergence command lines in the CG command block can have different values for different time periods by using the DURING specification. This specification consists of the key word DURING followed by a list of time periods and is appended to the end of the command line. Each of the time periods included in the list must correspond to the name of a TIME STEPPING command block (i.e., time_block_name) specified in the TIME CONTROL command block. If the DURING specification is omitted from the command line, the value of the parameter in the command line is applicable for the entire analysis. This value can be overwritten for specific periods by repeating the same command line and using the DURING specification. In other words, multiple occurrences of the same command line can be included in the command block, where one without a DURING specification applies to many time periods and others with unique DURING specifications apply to specific time periods.

The TARGET RESIDUAL command line specifies the target convergence criterion in terms of the actual residual norm. The TARGET RELATIVE RESIDUAL command line specifies a convergence criterion in terms of the relative residual. If both command lines are included in the CG command block, the solver will accept the solution as converged if either the target residual or the target relative residual is below the specified values. Note that use of the term "target" in this discussion means "desired."

The solver also allows the user to define acceptable convergence criteria. If the solution has not converged to the specified targets before the maximum number of iterations is reached, the residual is checked against the acceptable convergence criteria. These criteria are specified via the ACCEPTABLE RESIDUAL and ACCEPTABLE RELATIVE RESIDUAL command lines. The concepts of acceptable residual and acceptable relative residual are the same as those used for the target limits, i.e., in the TARGET RESIDUAL and TARGET RELATIVE RESIDUAL command lines. If the solution has not met the target criteria but has met the acceptable criteria, the solution is allowed to proceed. The default value for each acceptable criterion command line is 10 times the value of its corresponding target criterion command line, e.g., if the value of target_resid was 1.0e-6, the value of accept_resid would be 1.0e-5.

When a multilevel solver is being used, the acceptable residual and acceptable relative residual are set to $1e20$ as long as control failure is not being used. This is because the multilevel controls ultimately check and determine convergence, and allowing the multilevel controls to continue iterating even when a model problem fails to converge can allow the overall problem to converge.

Solutions that meet only the acceptable criteria are noted in the log file. If the acceptable criteria are not met, the solution at that load step has failed to converge, and the solver exits. If adaptive time stepping, as discussed in Section 4.10.2, is active, a solution of the load step may be attempted with a smaller time step. Otherwise, the code will exit at this point with an error.

If relative residuals are given in the convergence criteria, the REFERENCE command line can be used to select the method for computing the reference load. This command line has several options: EXTERNAL, INTERNAL, BELYTSCHKO, RESIDUAL and ENERGY. When using the default option (EXTERNAL), the reference value is taken as the maximum of the L^2 norm of either the

current externally applied or reaction loads. If the model has force boundary conditions, the external force vector used for this norm is composed of the nodal forces resulting from those boundary conditions. If there are no prescribed forces, the reaction forces at all prescribed kinematic boundary conditions are used instead. The `INTERNAL` option uses the norm of the internal forces as the reference load. This option is helpful in situations where a structure has no externally applied boundary conditions, such as in the case of a thermally loaded structure. The `BELYTSCHKO` option uses the maximum of the norm of the external load, the norm of the reactions, the norm of the internal forces, and the norm of the inertial forces. The `RESIDUAL` option denotes that the residual from the initial residual should be used as the reference quantity. The initial residual is computed from the first iteration of the solver.

The `ENERGY` option to the `REFERENCE` command line is a modification of the `EXTERNAL` option where energy conjugate norms of both the residual and reference are used. The energy conjugate norm of the residual is given by $R \cdot K^{-1} \cdot R$ where K is the diagonal of the stiffness matrix. For the energy conjugate norm of the external forces, R is replaced by the external nodal forces. The primary advantage of the energy norm is that forces and moments are converted to consistent and compatible energy units.

The `MINIMUM REFERENCE` offers the ability to set a value that the reference cannot drop below. Using the option `NONE` will use the value of `min_ref` for the reference if the computed reference is smaller. The option `FRACTION_OF_INITIAL` will compute a minimum reference value that is a fraction `min_ref` of the reference at the first iteration of the loadstep. A M will appear by the reference value in the log file if the minimum reference was used. If the line command is not used the minimum reference is zero. Having a minimum reference can make the use of relative tolerances still possible in the cases that the reference value disappears as the problem is solved.

Convergence can be difficult in some problems with rigid bodies without any applied moments. In these cases, it has been found advantageous to scale the rotational residuals on the rigid bodies by an approximation of the moment arm of the rigid bodies about the reference point. This can be accomplished through use of the `RESIDUAL NORM TYPE` command with the keyword `SCALE_RB_ROTATIONS`. The other options are `ALL` and `TRANSLATION`. A value of `ALL` is the default and treats all values the same in both the residual norm and the reference norm. A value of `TRANSLATION` uses only the translational components of residual and forces in the norm computations. Though this command is also available for other levels in a multilevel solver, this command is only appropriate for the core CG solver of a Sierra/SM region and should not be used in either `RVE` or `Control Modes` regions.

The default `ALL` option means that the residual includes all translational and rotational degrees of freedom. Note, translational forces and rotational moments have different units. Thus the `ALL` option may artificially increase or decrease the contribution of rotational degrees of freedom to the total residual depending on the units used. For example a model with a length unit in millimeters may tend to increase the contribution of rotational degrees of freedom to the convergence check (due to the numerically large moment arm) whereas a model with a length unit of meters may tend to decrease the contribution of rotational degrees of freedom to the convergence check (due to a numerically small moment arm.) The `ALL` option is mathematically inconsistent, but remains the default for historical compatibility. Also in practice the inconsistency is generally irrelevant as translational and rotational degrees of freedom tend to

converge at the same rate so that convergence of both quantities is obtained simultaneously.

The TRANSLATION option to the RESIDUAL NORM TYPE command computes the residual norm over only translational degrees of freedom and ignores all rotational components. This option may be useful if the model is failing to converge rotational degrees of freedom as may occur on problems with very thick shells. The SCALE_RB_ROTATIONS option computes the residual over translational degrees of freedom plus scaled rigid body rotational degrees of freedom. The rigid body rotational degrees of freedom are divided by the rigid body characteristic size to produce consistent units for a unified residual norm. The SCALE_RB_ROTATIONS option excludes all other rotational degrees of freedom, such as those on shells, from the residual norm. The SCALE_RB_ROTATIONS option is recommended if a model contains large rigid bodies with applied external moment boundary conditions. Also note another option to avoid any issues encountered by residual unit incompatibility is to use the ENERGY option for the solver REFERENCE command.



Warning: The RESIDUAL NORM TYPE command should not be used in RVE or Control Modes regions.

The MINIMUM RESIDUAL IMPROVEMENT command line stipulates that the CG solver must reduce the initial residual by a specified amount to obtain convergence. If this command line is included in the CG command block, the improvement condition must be met in addition to the standard target criteria. The parameter resid_improvement is a real number between 0.0 and 1.0 that specifies the amount by which the residual must be improved as a fraction of the initial residual in the current model problem. Thus, to stipulate that the residual must be smaller than 10% of the initial residual, one would set the value of resid_improvement equal to 0.9. The MINIMUM RESIDUAL IMPROVEMENT command line is primarily useful in the context of the multilevel solver. The model problems presented to the core solver sometimes begin with low residuals and require few iterations to converge. This command line forces the core solver to further improve the residual, which can accelerate the convergence of the other solver levels.

The MAXIMUM ITERATIONS and MINIMUM ITERATIONS command lines are used to specify the maximum and minimum number of core nonlinear solver iterations, respectively. These limits apply for a load step if the CG solver is used in the stand-alone mode or for a model problem if the CG solver is used as the core solver in the multilevel solver. The default maximum number of iterations is set to $max(N_{nodes}, 1000)$ where N_{nodes} is the global number of nodes in the mesh. The linear CG algorithm is guaranteed to converge to the solution in a number of iterations equal to the number of equations in the matrix. However, since we are solving a nonlinear system of equations using a nonlinear CG algorithm, the number of iterations to convergence can be larger than the number of equations . The default value for the minimum number of iterations, min_iter, is zero. If a number greater than zero is specified, the solver will iterate at least that many times, regardless of whether the convergence criteria are met.

The RESIDUAL ROUNDOFF TOLERANCE command line specifies a tolerance used to compute an approximately zero residual. Due to finite precision arithmetic, the solver can only drop the residual so far before additional corrections to the nodal coordinate field cannot be numerically represented. The approximately zero residual is the residual that would be produced by a perturbation of the displacement by the smallest machine representable perturbation. The default

value for this command is the accuracy of double precision arithmetic (1.0e-15). This tolerance number can be set to zero to disable this feature or set higher to converge to larger, but still near machine limit, residuals.

The log file annotates which convergence criterion was met in each CG iteration using markings shown in Table 4-1.

Table 4-1. Log File Convergence Markings

Mark	Convergence Property
>M	Convergence is reached, but still have not iterated the minimum number of specified iterations
>A	Reached maximum iterations and still not converged
<T	Residual converged to target tolerance of either relative or absolute criteria
<A	Reached maximum iterations and converged to acceptable tolerance of either relative or absolute criteria
~0	Residual is converged due to being approximately zero to within machine precision

4.2.2. Preconditioner Commands

```
PRECONDITIONER = BLOCK|BLOCK_INITIAL|DIAGONAL|
  ELASTIC|IDENTITY|PROBE|TANGENT|(ELASTIC) [<real>scaling_factor]
PRECONDITIONER ITERATION UPDATE = <integer>iter_update
  [DURING <string list>period_names]
BALANCE PROBE = <integer>balance_probe(1)
  [DURING <string list>period_names]
NODAL PROBE FACTOR = <real>probe_factor(1.0e-6)
  [DURING <string list>period_names]
NODAL DIAGONAL SCALE = <real>nodal_diag_scale(0.0)
  [DURING <string list>period_names]
NODAL DIAGONAL SHIFT = <real>nodal_diag_shift(0.0)
  [DURING <string list>period_names]
BEGIN FULL TANGENT PRECONDITIONER
  # ... see Section 4.3
END [FULL TANGENT PRECONDITIONER]
```

These command lines are used to select the type of preconditioner used by the CG algorithm, as well as the method used to form the preconditioner for certain types of preconditioners. The preconditioner is applied to the residual vector to obtain a gradient direction, which in turn is used to find a search direction. The most effective preconditioner is one that closely approximates the effect of multiplying by the inverse of the tangent stiffness matrix.

There are two basic types of preconditioners available: nodal and full tangent. The nodal preconditioners provide only the diagonal terms of the full tangent stiffness matrix or

3-by-3-block diagonal matrices along the diagonal of the stiffness matrix. The diagonal nature of a nodal preconditioner allows it to be inverted inexpensively and also use very little memory. Nodal preconditioners are so termed because they only account for the stiffness at each node in isolation and ignore the coupling between nodes that is accounted for in the off-diagonal terms of a full stiffness matrix. The result of this approximation is that with a nodal preconditioner, in a single CG iteration, the equilibration of a residual at a node can only cause movement in nodes that are directly connected by an element to that node. Thus, as the model size increases, the number of iterations also increases. Nodal preconditioners require many iterations, but they are often efficient because the iterations are inexpensive, especially if the problem is "blocky" in nature.

The `PRECONDITIONER` command line allows the user to select a nodal preconditioner or the full tangent preconditioner with most defaults. As defined below, this command line has several options. Some options are synonyms for other options.

- The `BLOCK` and `ELASTIC` options, which are synonyms, specify the default preconditioner. These options provide a nodal preconditioner with 3-by-3-block matrices that are computed based on the elastic material properties. These matrices are updated at every model problem to account for the current geometry.
- The `BLOCK_INITIAL` option, which is a variant of the `BLOCK` preconditioner, forms the preconditioner at the beginning of the analysis but never recomputes it for efficiency.
- The `DIAGONAL` preconditioner is formed in the same way as the `BLOCK` preconditioner, but only uses the terms on the diagonal.
- The `PROBE` option forms a 3-by-3-block diagonal preconditioner by probing the stiffness of nodes rather than by using the elastic stiffness.
- The `IDENTITY` option uses an identity matrix for the preconditioner, meaning that the residual is used as the gradient direction. This option is only of academic interest and is included for completeness.
- The `TANGENT` option is a quick way of specifying the `FULL TANGENT PRECONDITIONER` with most defaults used. A small amount (1.0e-10) of tangent diagonal scaling is assigned by default when using `PRECONDITIONER = TANGENT`. See Section 4.3 for a further description of the full tangent preconditioner.

The `PRECONDITIONER ITERATION UPDATE` command line controls the frequency at which the full tangent and nodal probe preconditioner are updated. By default, the preconditioner is only updated at the first iteration of each model problem. If this command line is used, updates will occur at the specified frequency `iter_update`. If a model experiences highly nonlinear behavior during a model problem, it may be useful to use this command to cause more frequent preconditioner updates. Such problems may converge better when `iter_update` is set to about 10 for the full tangent case. Realize that frequent updates may reduce the number of CG iterations but increase the computational cost.

The `BALANCE PROBE` and `NODAL PROBE FACTOR` command lines control the behavior of the probing algorithm that is used to obtain the stiffness for the probe preconditioner (selected via the `PROBE` option in the `PRECONDITIONER` command line). For the probe preconditioner, the tangent

stiffness K,

$$K_{ij} = \frac{\partial F_i^{int}}{\partial x_j}, \tag{4.1}$$

where F^{int} is the internal force and x is the displacement, is approximated using finite differences. The nodal probe preconditioner only computes the 3-by-3-block diagonal version of the stiffness matrix. The `NODAL PROBE FACTOR` command line controls the probing distance, with the value of `probe_factor` defaulting to 1.0e-6. The probe factor controls the probing distance relative to element size. Smaller values may give a better tangent approximation, but these values could result in the generation of round-off errors.

The `BALANCE PROBE` command line selects the type of finite differencing scheme used to probe for the stiffness. The value of `balance_probe` can be set to 0, 1, or 2, as explained below.

- Setting `balance_probe` to 0, the default for the nodal probe preconditioner, causes the preconditioner to be formed by forward finite differencing. Probing occurs in only one direction, resulting in first-order error in the preconditioner. The stiffness computed when `balance_probe` is set to 0 appears as

  $$K_{ij} = \frac{F_i^{int}(x + \delta e_j) - F_i^{int}(x)}{\delta}, \tag{4.2}$$

  where δ is the probing distance, controlled by the `NODAL PROBE FACTOR` command line, and e_j is a unit vector in the jth equation direction.
- Setting `balance_probe` to 1, the default for the full tangent preconditioner, causes the preconditioner to be formed with central differencing. Probing at each element degree of freedom is performed in both positive and negative directions, leading to an estimate of the tangent stiffness with second-order error. This approach can be necessary in problems with material nonlinearity, where a probe that stretches rather than compresses the element can result in orders of magnitude differences in estimated stiffness. The stiffness computed when `balance_probe` is set to 1 appears as

  $$K_{ij} = \frac{F_i^{int}(x + \delta e_j) - F_i^{int}(x - \delta e_j)}{2\delta}. \tag{4.3}$$

- The value of `balance_probe` may also be set to 2, which allows the full tangent preconditioner to obtain central finite differencing with fourth-order error. This approach requires twice the work as central differencing but is more accurate. The stiffness computed when `balance_probe` is set to 2 appears as

  $$K_{ij} = \frac{-F_i^{int}(x + 2\delta e_j) + 8F^{int}(x + \delta e_j) - 8F^{int}(x - \delta e_j) + F_i^{int}(x - 2\delta e_j)}{12\delta}. \tag{4.4}$$

The `NODAL DIAGONAL SCALE` command line specifies a nodal diagonal scale factor, which is used to scale the diagonal entries of the nodal preconditioner. This is useful in some situations where a model may be softening due to material effects. The diagonal terms of the nodal preconditioner are scaled by (1.0 + `nodal_diagonal_scale`). The default value of `nodal_diagonal_scale` is 0.0, which means the diagonals are not scaled. A value of 0.1 scales up all the diagonal terms by 10 percent.

The NODAL DIAGONAL SHIFT command line specifies a nodal diagonal shift, which is summed into the diagonal entries of the nodal preconditioner. This is useful in some situations where a model may be softening due to material effects. The diagonal terms of the nodal preconditioner are shifted by (num_nodes*tangent_diagonal_shift). The default value of nodal_diagonal_shift is 0.0, which means the diagonals are not shifted.

The FULL TANGENT PRECONDITIONER command block enables the full tangent preconditioner and specifies options that apply specifically to that type of preconditioner. See Section 4.3 for a description of the options available. The FULL TANGENT PRECONDITIONER command block cannot be used with the PRECONDITIONER command line in a CG command block. The user must select only one type of preconditioner for an analysis.

4.2.3. Line Search Command

```
LINE SEARCH ACTUAL|TANGENT [DURING <string list>period_names]
LINE SEARCH SECANT [<real>scale_factor]
LINE SEARCH BACKTRACK [<real>min_step(1.0e-3)
                       <real>reduction(0.95)
                       <real>init_step(1.0)
                       <enum>NONE|SECANT_INIT(NONE)
                       <real>dispinc(infinity)]
LINE SEARCH TYPE MULTI = <real>num_domain(2)
  [DURING <string list>period_names]
```

During each CG iteration, after the search direction is computed, a line search is used to find an optimal scaling factor that when applied to the search vector will result in a minimized residual. The line search algorithm is controlled with the LINE SEARCH command line. The SECANT option is the default, and is used if the LINE SEARCH command line is not present.

- The ACTUAL line search is a single-step quadratic line search that is equivalent to the corresponding JAS3D line search option. The line search requires an additional evaluation for each iteration of the element internal forces and thus the material response. For a given search direction s, residual r, and velocity v, the step length α is computed as

$$\alpha = -\frac{s^T r(v)}{s^T (F_{int}(v+s) - F_{int}(v))}, \tag{4.5}$$

  where the residual is computed based on the internal force F_{int} and on the external force F_{ext} as $r = F_{int}(v) - F_{ext}(v)$.
- The default SECANT line search is a slight variation of the ACTUAL line search described above. Like the ACTUAL line search, this line search requires an additional internal force evaluation for each iteration. The step length α is computed as

$$\alpha = -\frac{s^T r(v)}{s^T (r(v+s) - r(v))}. \tag{4.6}$$

The optional `scale_factor` can be used with the secant line search to scale the search vector s in the equation above. If `scale_factor` is omitted, no scaling is done. If `scale_factor` is provided on the command line, s is scaled by the product of `scale_factor` and the dimension of the smallest element. Setting the scale factor to a small value can be helpful to avoid inverting elements during the evaluation of $r(v+s)$. The `SECANT` line search is recommended for problems in which the external force is highly nonlinear (e.g., a beam-bending problem with a pressure distribution).

- The `TANGENT` line search avoids the additional internal force calculation that is inherent in the other line search types by using an approximate tangent modulus computed in the last call to the material subroutine for each element. The step length α is computed as

$$\alpha = -\frac{s^T r(v)}{s^T K s}, \tag{4.7}$$

where K is the approximate tangent stiffness.



Warning: The `TANGENT` line search capability is not supported for any element formulation besides uniform gradient hex.

- The `BACKTRACK` line search uses a cutback search to find the optimal stepsize. The solver will first try the `init_step` stepsize. If that stepsize does not meet the desired `reduction` it will continue searching by halving the stepsize until the desired `reduction` in residual is met, 20 iterations, or the `min_stepsize` is reached. The optimal stepsize is selected from all those attempted. The `BACKTRACK` line search will tend to be more expensive than other line search methods but has potential to be more robust in some circumstances such as problems with very sharp nonlinearity. Optional enum `SECANT_INIT` will add the secant stepsize to the stepsizes attempted and will prefer it if successful. The `dispinc` will further limit the stepsize so that the displacement at any node is less than the value specified.
- The `MULTI` line search first subdivides the line search space into multiple domains and then applies the secant line search method on each domain. The best line search scaling factor found in any sub domain is ultimately used. The multi line search option may in some circumstances provide a more optimal or robust line search value than the secant, though at a significantly increased cost per iteration. The number of subdomains used is defined by the `sub_domain` option which defaults to two.

4.2.4. CG Algorithm Commands

```
ITERATION RESET = <integer>iter_reset(infinity)
  [DURING <string list>period_names]
ORTHOGONALITY MEASURE FOR RESET = <real>ortho_reset(0.5)
  [DURING <string list>period_names]
RESET LIMITS <integer>iter_start <integer>iter_reset
  <real>reset_growth <real>reset_orthogonality
  [DURING <string list>period_names]
BETA METHOD = FletcherReeves|PolakRibiere|Newton|
  PolakRibierePlus(PolakRibiere)
```

```
  [DURING <string list>period_names]
MINIMUM STEP LENGTH = <real>min_step(-infinity)
  [DURING <string list>period_names]
MAXIMUM STEP LENGTH = <real>max_step(infinity)
  [DURING <string list>period_names]
```

The behavior of the CG algorithm can be changed using the command lines listed above. The CG algorithm orthogonalizes the current search direction against the previous search direction at each iteration. During an iteration where a reset occurs, the orthogonalization is skipped, resulting in a search in the steepest descent direction. These command lines can be appended with the `DURING` specification, as discussed in Section 4.2.1.

The `ITERATION RESET` command line specifies that the CG algorithm be reset after every `iter_reset` iterations. The default value is set equal to the maximum integer, which defaults the conjugate gradient algorithm to not reset.

For the CG algorithm to work well, each new search direction is constructed orthogonal to the previous search directions. If the current search direction has a significant component in one of the previous search directions, error can be introduced that will not be corrected in future iterations, resulting in a lack of convergence. At every iteration, the relative lack of orthogonality between the current gradient direction and the previous search direction is computed. When these become far from orthogonal, a reset can be performed to ignore the previous search directions and use the current gradient direction as the next search direction, which is equivalent to setting the orthogonalization parameter β to zero for that iteration (see Equation 4.8). The `ORTHOGONALITY MEASURE FOR RESET` command line specifies that a reset will occur if the lack of orthogonality exceeds the value of `ortho_reset`. The default value of 0.5 results in few resets. Setting `ortho_reset` to a value as tight as 0.01 can result in many resets and potentially improved convergence on some problems.

The `RESET LIMITS` command line is also used to control CG resets. This command line is provided to achieve compatibility with JAS3D, and it behaves exactly the same as its JAS3D counterpart. The command line, however, should not be used when either the `ITERATION RESET` command line or the `ORTHOGONALITY MEASURE FOR RESET` command line is included in the `CG` command block. Up to four parameters can optionally be specified in the `RESET LIMITS` command line. The first, `iter_start`, sets the number of iterations to wait before looking for a minimum residual. The second, `iter_reset`, sets the number of iterations to allow between finding a minimum and restarting the iterative algorithm. The third, `reset_growth`, sets the amount of growth in the residual norm that would indicate divergence and thus trigger a reset. Finally, `reset_orthogonality` sets the relative lack of orthogonality between the current gradient direction and the previous search direction that would trigger a reset.

The `BETA METHOD` command line allows different formulas to be used in computing the orthogonalization parameter β in the CG algorithm,

$$s_k = g_k + \beta s_{k-1}, \tag{4.8}$$

where s_k is the current iteration's search direction, s_{k-1} is the previous iteration's search direction, and g_k is the current iteration's gradient direction.

In the below equations, r_k is the current iteration's residual vector, r_{k-1} is the previous iteration's residual vector, and g_{k-1} is the previous iteration's gradient vector. The `PolakRibierePlus` formula forces beta to be positive. A steepest descent direction is taken when beta becomes negative.

- For the `FletcherReeves` option the β scalar is computed as

$$\beta_k = \frac{r_k^T g_k}{r_{k-1}^T g_{k-1}}. \tag{4.9}$$

- For the `PolakRibiere` option (the default) β is computed as

$$\beta_k = \frac{r_k^T (g_k - g_{k-1})}{r_{k-1}^T g_{k-1}}. \tag{4.10}$$

- For the `PolakRibierePlus` option β is computed as

$$\beta_k = \max(\beta_k^{PR}, 0). \tag{4.11}$$

- For the `Newton` option the nonlinear conjugate gradient algorithm is transformed into a Quasi-Newton method without a line search. In order to best approximate a Newton solve, this method should be used with a full tangent preconditioner that is updated on every iteration.

The `MINIMUM STEP LENGTH` and `MAXIMUM STEP LENGTH` commands set the minimum step length and maximum step length that the CG algorithm can take along a search direction. The default values are set so that any positive or negative step length will be accepted. This option can be used to limit motion of the model along any search direction and can help improve convergence sometimes.

4.2.5. Diagnostic Output Commands

```
ITERATION PRINT = <integer>iter_print
  [DURING <string list>period_names]
ITERATION PLOT = <integer>iter_plot
  [DURING <string list>period_names]
ITERATION PLOT OUTPUT BLOCKS  = <string list>plot_blocks
```

The command lines listed above can be used to control the output of diagnostic information from the CG solver. The `ITERATION PRINT` and `ITERATION PLOT` command lines can be appended with the `DURING` specification, as discussed in Section 4.2.1.

The `ITERATION PRINT` command line controls the frequency at which the convergence information is printed to the log file. The default value of `iter_print` is 25 if a nodal preconditioner is used, and it is 1 if a full tangent preconditioner is used.

For a description of `ITERATION PLOT` and `ITERATION PLOT OUTPUT BLOCKS`, see section 4.1.1.

4.2.6. CG Input Example

```
begin solver
  Begin cg
    target relative residual = 1.0e-07
    Maximum Iterations = 5000
    preconditioner iteration update = 100
    preconditioner = elastic
  end
end
```

4.3. Full Tangent Preconditioner

In addition to the nodal preconditioners, Sierra/SM provides the capability to use a full tangent preconditioner in the CG algorithm. The full tangent preconditioner employs a scalable parallel linear solver to solve for the gradient direction using the full tangent stiffness matrix. Although the full tangent preconditioner is significantly more costly per CG iteration than the nodal preconditioners, it can solve problems in a smaller number of iterations. Full tangent preconditioners are especially effective in solving poorly conditioned problems that are often extremely difficult to solve using the CG algorithm with nodal preconditioners, such as those problems that involve the bending response of long, slender members.

To use a full tangent preconditioner generally two command blocks are added to the input file, one required and one optional. First, the `FULL TANGENT PRECONDITIONER` command block, described in this section, must be added to the `CG` command block. The `FULL TANGENT PRECONDITIONER` command block is used instead of the `PRECONDITIONER` command line in the `CG` command block. Second, a command block for a linear solver may be defined in the SIERRA scope, as described in Section 4.4. That linear solver command block may then be referenced from within the `FULL TANGENT PRECONDITIONER` command block, instructing the CG algorithm to use the specified linear solver. If no linear solver command block is given and/or no linear solver is referenced in the `FULL TANGENT PRECONDITIONER` command block, the FETI linear solver will be employed with default settings. The command block to enable the full tangent preconditioner is as follows:



Known Issue: Deactivation of element blocks (see Section 6.1.7.9) in conjunction with the full tangent preconditioner is available but not yet fully tested. Use caution when combining these capabilities.

```
BEGIN FULL TANGENT PRECONDITIONER
  #
  # solver selection commands
  LINEAR SOLVER = <string>linear_solver_name(FETI)
    [DURING <string list>period_names]
  NODAL PRECONDITIONER METHOD = ELASTIC|PROBE|DIAGONAL|BLOCK
    (ELASTIC)
  #
  # tangent matrix formation commands
  PROBE FACTOR = <real>probe_factor(1.0e-6)
    [DURING <string list>period_names]
  BALANCE PROBE = <integer>balance_probe(1)
    [DURING <string list>period_names]
  PENALTY FACTOR = <real>penalty_factor(100.0)
    [DURING <string list>period_names]
  # useful to improve ill-conditioned tangent matrix
  TANGENT DIAGONAL SCALE = <real>tangent_diag_scale(0.0)
    [DURING <string list>period_names]
  TANGENT DIAGONAL SHIFT = <real>tangent_diag_shift(0.0)
```

```
    [DURING <string list>period_names]
  # turn off for speed
  CONDITIONING = NO_CHECK|CHECK|AUTO_REGULARIZATION(CHECK)
    [DURING <string list>period_names]
  HANGING NODE FACTOR = <real>(1.0e-3)
    [DURING <string list>period_names]
  #
  # reset and iteration commands
  MAXIMUM RESETS FOR MODELPROBLEM = <integer>max_mp_resets
    (100000) [DURING <string list>period_names]
  MAXIMUM RESETS FOR LOADSTEP = <integer>max_ls_resets
    (100000) [DURING <string list>period_names]
  MAXIMUM ITERATIONS FOR MODELPROBLEM
    = <integer>max_mp_iter(100000)
    [DURING <string list>period_names]
  MAXIMUM ITERATIONS FOR LOADSTEP = <integer>max_ls_iter
    (100000) [DURING <string list>period_names]
  ITERATION UPDATE = <integer>iter_update
    [DURING <string list>period_names]
  ITERATION PRINT = <integer>iprint
  # freeze current preconditioner
  SMALL NUMBER OF ITERATIONS = <integer>small_num_iter
    [DURING <string list>period_names]
  SMOOTHING STEPS = <integer>smooth_steps(0)
    [DURING <string list>period_names]
  MINIMUM SMOOTHING ITERATIONS = <integer>min_smooth_iter(0)
    [DURING <string list>period_names]
  MAXIMUM SMOOTHING ITERATIONS = <integer>max_smooth_iter(0)
    [DURING <string list>period_names]
  TARGET SMOOTHING RESIDUAL = <integer>tgt_smooth_resid(0)
    [DURING <string list>period_names]
  TARGET SMOOTHING RELATIVE RESIDUAL
    = <integer>tgt_smooth_rel_resid(0)
    [DURING <string list>period_names]
  #
  # fall-back strategy commands
  STAGNATION THRESHOLD = <real>stagnation(1.0e-12)
    [DURING <string list>period_names]
  # triggers preconditioner switch or update
  MINIMUM CONVERGENCE RATE = <real>min_conv_rate(1.0e-4)
    [DURING <string list>period_names]
  # either switch to nodal preconditioner or update
  ADAPTIVE STRATEGY = SWITCH|UPDATE|NONE(SWITCH)
    [DURING <string list>period_names]


END [FULL TANGENT PRECONDITIONER]
```

The FULL TANGENT PRECONDITIONER command block contains all command lines related to the interaction with the linear solver. There are command lines to control selection of the linear solver, formation of the matrices passed into the linear solver, CG iteration strategies, updating strategies, and strategies for dealing with poor convergence. Reasonable defaults have been set for most problems. The command lines in this command block are described in Sections 4.3.1 through 4.3.5.

4.3.1. Solver Selection Commands

```
LINEAR SOLVER = <string>linear_solver_name
  [DURING <string list>period_names]
NODAL PRECONDITIONER METHOD = ELASTIC|PROBE|DIAGONAL|
  BLOCK(ELASTIC)
```

The command lines listed above are used in selecting a linear solver for use with the full tangent preconditioner and in selecting a nodal preconditioner. The LINEAR SOLVER command line specifies the name of the linear solver used to compute the action of the preconditioner by solving the linear system. If no linear solver is specified, Sierra/SM will default to using the FETI linear solver with its default settings. Linear solvers are defined in the SIERRA scope.

Although several linear-solver packages are available for use as preconditioners in Sierra/SM, we recommend that the FETI equation solver be used in production analyses. The FETI equation solver is actively maintained and tested by the Sierra/SM development team. The FETI EQUATION SOLVER command block is documented in Section 4.4. Other linear solvers may have compatibility problems with other code capabilities such as MPCs and contact.

An arbitrary number of equation solver command blocks can be included in the input file to define various sets of solver options. Each equation solver command block must be given a unique name, which is referenced in the linear_solver_name parameter of the LINEAR SOLVER command line. This name selects the specified linear solver as a preconditioner for the CG solver.

When Sierra/SM uses a full tangent preconditioner, it also forms a nodal preconditioner. This nodal preconditioner is used in contact calculations, as a fall-back preconditioner if the full tangent preconditioner does not perform well and optionally for performing smoothing iterations before use of the full tangent preconditioner.

The nodal preconditioner is selected with the NODAL PRECONDITIONER METHOD command line. Four options are available: ELASTIC, PROBE, DIAGONAL, and BLOCK. The ELASTIC option is the default. These options have the same meaning as the corresponding options given for the nodal preconditioner in the CG command block, as documented in Section 4.2.2.

4.3.2. Matrix Formation Commands

```
PROBE FACTOR = <real>probe_factor(1.0e-6)
  [DURING <string list>period_names]
BALANCE PROBE = <integer>balance_probe(1)
```

```
  [DURING <string list>period_names]
PENALTY FACTOR = <real>penalty_factor(100.0)
  [DURING <string list>period_names]
TANGENT DIAGONAL SCALE = <real>tangent_diag_scale(0.0)
  [DURING <string list>period_names]
TANGENT DIAGONAL SHIFT = <real>tangent_diag_shift(0.0)
  [DURING <string list>period_names]
CONDITIONING = NO_CHECK|CHECK|AUTO_REGULARIZATION(CHECK)
  [DURING <string list>period_names]
HANGING NODE FACTOR = <real>(1.0e-3)
  [DURING <string list>period_names]
```

The command lines listed above can be used to optionally control the way the stiffness matrix is formed. This matrix is formed by assembling contributions from all active elements. These element stiffness matrices are formed by finite differencing.

The `PROBE FACTOR` command line controls the finite difference probing distance relative to element size. The default value of `probe_factor` is 1.0e-6. Smaller values may give a better tangent approximation, but they may also generate round-off errors.

The `BALANCE PROBE` command line selects the type of finite-differencing scheme used to probe for the stiffness. The value of `balance_probe` can be set to 0, 1, or 2. Setting `balance_probe` to 0 causes the preconditioner to be formed by forward finite differencing. Probing occurs in only one direction, resulting in first-order truncation error in the preconditioner. Setting `balance_probe` to 1, the default for the full tangent preconditioner, causes the preconditioner to be formed with central differencing. Probing at each element degree of freedom is performed in both positive and negative directions, leading to a second-order estimate of the tangent stiffness. The value of `balance_probe` may also be set to 2, which allows the full tangent preconditioner to obtain central finite differencing with fourth-order error. This scheme requires twice the work as the central-differencing scheme but is more accurate. See Section 4.2.2 for a more detailed discussion of these options.

Constraints (i.e., contact, rigid body, and periodic boundary conditions) are enforced by conversion to penalty "elements" whose stiffnesses (controlled by the `PENALTY FACTOR` command line—see below) are contributed to the linear system directly. The errors caused by penalty compliance are corrected in each CG iteration, so the converged solution will not have errors due to penalty compliance.

The `PENALTY FACTOR` command line specifies the penalty stiffness. At each constraint, the diagonal tangent stiffness of the side B degree of freedom is multiplied by the value of the `penalty_factor` parameter. The default value for this parameter is 100.0, which is reasonable for most problems. Assigning a high value (greater than about 1.0e+8) can cause poor conditioning of the linear system and also cause the linear solver to perform poorly. Assigning a low value (less than about 10) could require more CG iterations due to the correction of errors resulting from penalty compliance.

The `TANGENT DIAGONAL SCALE` command line specifies a tangent matrix diagonal scale factor, which is used to scale the diagonal entries of the tangent matrix. This is useful in some situations

where a model does not have sufficient boundary conditions to remove all rigid body modes. This command has also been used as a last resort strategy to achieve convergence when all else fails. The diagonal terms of the tangent matrix are scaled by (1.0 + `tangent_diagonal_scale`). The default value of `tangent_diagonal_scale` is 0.0, which means the tangent matrix diagonals are not scaled. A value of 0.1 scales up all tangent matrix diagonal terms by 10 percent.

The `TANGENT DIAGONAL SHIFT` command line specifies a tangent matrix diagonal shift, which is summed into the diagonal entries of the tangent matrix. This is useful in some situations where a model does not have sufficient boundary conditions to remove all rigid body modes. This command has also been used as a last resort strategy to achieve convergence when all else fails. The diagonal terms of the tangent matrix are shifted by (`num_nodes*tangent_diagonal_shift`). The default value of `tangent_diagonal_shift` is 0.0, which means the tangent matrix diagonals are not shifted.

The `CONDITIONING` command line controls the linear system conditioning inside the FETI solver. It has no effect on any other linear system solver. By default, when used with the full tangent preconditioner FETI checks the conditioning of its coarse level matrix and reports a warning to the log file if it finds ill-conditioning (defined as a condition number greater than 1e5, where the condition number is defined as the ratio of the maximum eigenvalue to the minimum eigenvalue). This default behavior corresponds to the `CHECK` option. For performance optimization this check may be turned off using the `NO_CHECK` option. The third option, `AUTO_REGULARIZATION`, instructs FETI to add a small shift to its coarse matrix to regularize the linear system and artificially reduce the condition number. In the context of the CG solver, this will affect convergence but not the residual calculations. Coarse level matrix ill-conditioning often indicates an issue with the problem, such as the presence of zero energy rigid body modes either inherently or temporarily as a result of a CG iteration, and `AUTO_REGULARIZATION` may improve CG convergence in the presence of ill-conditioning. A basic power method is used to compute the highest eigenvalue, and a shifted power method is used to compute the lowest eigenvalue. This capability should be viewed both as a diagnostic tool and a robustness aide.

The `HANGING NODE FACTOR` command is used to apply a nodal stiffness to any nodes that may exist on a processor but are not attached to any element on that processor. This configuration can occur with parallel contact, spot welds, MPCs, and other non-local joints and boundary conditions. This artificial stiffness is used to avoid singularities in the processor local stiffness matrix. The value used is a scale factor on the diagonal nodal stiffness.

4.3.3. Reset and Iteration Commands

```
MAXIMUM RESETS FOR MODELPROBLEM = <integer>max_mp_resets
  (100000) [DURING <string list>period_names]
MAXIMUM RESETS FOR LOADSTEP = <integer>max_ls_resets
  (100000) [DURING <string list>period_names]
MAXIMUM ITERATIONS FOR MODELPROBLEM = <integer>max_mp_iter
  (100000) [DURING <string list>period_names]
MAXIMUM ITERATIONS FOR LOADSTEP = <integer>max_ls_iter
  (100000) [DURING <string list>period_names]
```

```
ITERATION UPDATE = <integer>iter_update
  [DURING <string list>period_names]
ITERATION PRINT = <integer>iprint
SMALL NUMBER OF ITERATIONS = <integer>small_num_iter(0)
  [DURING <string list>period_names]
```

When a linear solver is used as a preconditioner, the efficiency of the solution can often be dramatically affected by the frequency at which the preconditioner is updated. The computational expense of updating the preconditioner, which involves forming and factorizing the stiffness matrix, is often much higher than the cost of applying the preconditioner, which involves only a back-solve, during a CG iteration. If the stiffness does not dramatically change from iteration to iteration, it is often much more efficient to re-use the preconditioner for many iterations, or even load steps, than it is to update it at every iteration.

The command lines listed above can be used to control when the preconditioner is updated. As Sierra/SM allows for the nodal preconditioner to be used in place of the full tangent preconditioner in some situations, these command lines also control this behavior. These command lines can be appended with the `DURING` specification, as discussed in Section 4.2.1.

The `MAXIMUM RESETS FOR MODELPROBLEM` command line limits the number of times that the preconditioner can be reset (updated) during a model problem. By default, there is no limit on the number of updates that can occur during a model problem. The `MAXIMUM RESETS FOR LOADSTEP` command line limits the updates that can occur during a load step. By default, there is also no limit on the number of updates that can occur during a load step.

The `MAXIMUM ITERATIONS FOR LOADSTEP` and `MAXIMUM ITERATIONS FOR MODELPROBLEM` command lines can be used to limit the number of CG iterations performed using the full tangent preconditioner per model problem or load step, respectively. The values of the parameters `max_mp_iter` and `max_ls_iter` are both unlimited by default. If one of these iteration limits is reached, the CG solver will continue to iterate using the fall-back nodal preconditioner until either the CG solver iteration limit is reached or convergence is achieved. It can be useful to limit the iterations taken by the full tangent preconditioner. If the full tangent preconditioner is performing poorly, the nodal preconditioner may work better sometimes.

The `ITERATION UPDATE` command line controls the frequency at which the full tangent preconditioner is updated. By default, it is only updated at the first iteration of each model problem. If this command line is used, the preconditioner will be updated at the frequency specified by `iter_update`. If a model experiences highly nonlinear behavior over the course of a model problem, it may be useful to use this command line to cause more frequent preconditioner updates. Such problems may converge better when `iter_update` is set to about 10. Realize that frequent updates may reduce the number of CG iterations but dramatically increase the computational cost.

The `ITERATION PRINT` command line controls the frequency at which the convergence information is printed to the log file while applying the full tangent preconditioner. The default behavior is inherited from the same command in the containing `CG` block, which defaults to

printing every iteration when the full tangent preconditioner is in use. Alternatively, when a large number of conjugate gradient smoothing iterations are being used in conjunction with the full tangent preconditioner it can be convenient to set `ITERATION PRINT` to a larger number in the `CG` block while also setting `ITERATION PRINT` to a smaller number in the full tangent preconditioner. Iteration information will also be output during a full tangent preconditioner update.

It can often be efficient to update the preconditioner infrequently. The `SMALL NUMBER OF ITERATIONS` command line is used to "freeze" the preconditioner, or prevent updates, until a model problem requires more iterations than the number specified in `small_num_iter`. The default value of `small_num_iter` is 0, meaning that the preconditioner is never re-used for the next model problem unless the `SMALL NUMBER OF ITERATIONS` command line is used. It is often beneficial to set `small_num_iter` to 10. This can result in the re-use of the preconditioner over many load steps. When this command line is used, the CG solver may converge in a small number of iterations immediately following a preconditioner update; however, over the course of several load steps, the iteration counts slowly increase because the preconditioner is out of date. At a certain point, the iteration count exceeds `small_num_iter`, and the preconditioner is reset. This solution strategy is efficient because it allows re-use of the preconditioner while it is still relatively effective.



Warning: The `SMALL NUMBER OF ITERATIONS` command is not guaranteed to be compatible with linear solvers other than FETI. For instance, the Aztec solver fails to find the underlying preconditioner when it is used in an aria and an adagio region. A possible workaround is to use a unique linear solver in each region.

4.3.4. Smoothing Iterations

```
SMOOTHING STEPS = <integer>smooth_steps(0)
  [DURING <string list>period_names]
MINIMUM SMOOTHING ITERATIONS = <integer>min_smooth_iter(0)
  [DURING <string list>period_names]
MAXIMUM SMOOTHING ITERATIONS = <integer>max_smooth_iter(0)
  [DURING <string list>period_names]
TARGET SMOOTHING RESIDUAL = <integer>tgt_smooth_resid(0)
  [DURING <string list>period_names]
TARGET SMOOTHING RELATIVE RESIDUAL =
  <integer>tgt_smooth_rel_resid(0)
  [DURING <string list>period_names]
AUTOMATIC SMOOTHING FACTOR = <real>auto_smooth
AUTOMATIC SMOOTHING RELATIVE RESIDUAL FACTOR =
  <real>rel_auto_smooth
AUTOMATIC SMOOTHING RESIDUAL FACTOR = <real>abs_auto_smooth
```

Sometimes it is beneficial to perform many CG iterations using the nodal preconditioner before applying the full tangent preconditioner. The iterations with the nodal preconditioner are used as

a “smoother” to put the model in a better state for the full tangent preconditioner. The option SMOOTHING STEPS will delay the use of the full tangent preconditioner until the specified number of model problems have been solved, in a multilevel solver a contact iteration is a model problem solve. Smoothing iterations are controlled using the MINIMUM SMOOTHING ITERATIONS, MAXIMUM SMOOTHING ITERATIONS, TARGET SMOOTHING RESIDUAL, and TARGET SMOOTHING RELATIVE RESIDUAL command lines. By default, no smoothing iterations are taken. Using these commands, the user can specify that smoothing iterations should be taken and control how many are taken.

The number of smoothing iterations taken can vary based on the convergence of the solver during those iterations. The smoothing iteration commands allow the user to control the desired convergence of the smoother before switching to the full tangent preconditioner. These commands behave the same way as the standard solver convergence commands, except that they apply only to the smoothing iterations. The standard convergence criteria still must be met to obtain a converged solution. The MINIMUM SMOOTHING ITERATIONS command controls the minimum smoothing iterations taken, and the MAXIMUM SMOOTHING ITERATIONS command controls the maximum smoothing iterations. The TARGET SMOOTHING RESIDUAL and TARGET SMOOTHING RELATIVE RESIDUAL control the residual and relative residual, respectively, that must be obtained in the smoothing iterations before switching to the full tangent preconditioner. Once tgt_smooth_resid has been reached, the solver switches from using the nodal preconditioner to the tangent preconditioner. If max_smooth_iter is reached before the target smoothing convergence is reached, the solver switches to using the full tangent preconditioner.

This feature can also be used in cases where the nodal preconditioner may be effective enough to achieve convergence with relatively few iterations, but occasionally the nodal preconditioner is not effective. If max_smooth_iter is set to a relatively high number, many load steps can be solved without ever using the full tangent preconditioner, but this preconditioner is used on the more difficult steps that require more iterations.

Another use of the smoothing iteration control commands is for switching between the nodal preconditioner and the full tangent preconditioner for certain time periods. These command lines, like all the other command lines discussed above, can be set on a period-specific basis by appending them with the DURING keyword, followed by a list of periods. To use the nodal preconditioner on some periods, the user can set max_smooth_iter to a high number for those periods and leave the default value of 0 for the periods in which the full tangent preconditioner is desired.

An alternative way the smoothing tolerances can be set is with the AUTOMATIC SMOOTHING FACTOR commands. The automatic smoothing factor commands automatically select an appropriate smoothing residual target based on the initial and target residuals.

The AUTOMATIC SMOOTHING RELATIVE RESIDUAL FACTOR effectively sets the TARGET SMOOTHING RELATIVE RESIDUAL r_S based off of the initial relative residual r_0, target relative residual r_T and input rel_auto_smooth scaling factor f using the following equation.

$$r_S = e^{f*(ln(r_T)-ln(r_0))+ln(r_0)} \tag{4.12}$$

Similarly The AUTOMATIC SMOOTHING RESIDUAL FACTOR effectively sets the TARGET SMOOTHING RESIDUAL R_S based off of the initial residual R_0, target residual R_T and input abs_auto_smooth scaling factor F using the following equation.

$$R_S = e^{F*(ln(R_T)-ln(R_0))+ln(R_0)} \tag{4.13}$$

Use of the AUTOMATIC SMOOTHING FACTOR command sets both the AUTOMATIC SMOOTHING RELATIVE RESIDUAL FACTOR and the AUTOMATIC SMOOTHING RESIDUAL FACTOR at the same time to the same value.

When using the AUTOMATIC SMOOTHING FACTOR command the scaling factor may be any number between zero and one though appropriate values are generally in the range of 0.1 to 0.5.

4.3.5. Fall-Back Strategy Commands

```
STAGNATION THRESHOLD = <real>stagnation(1.0e-12)
MINIMUM CONVERGENCE RATE = <real>min_conv_rate(1.0e-4)
ADAPTIVE STRATEGY = SWITCH|UPDATE|NONE(SWITCH)
```

Occasionally, the full tangent preconditioner may stagnate, failing to significantly reduce the residual from one CG iteration to the next. At each iteration, Sierra/SM checks for slow convergence, and attempts to remedy poor convergence if detected. The commands listed above can optionally be added to the FULL TANGENT PRECONDITIONER command block.

The STAGNATION THRESHOLD and MINIMUM CONVERGENCE RATE command lines are used to set the stagnation and min_conv_rate parameters, respectively. These parameters are used in a strategy that attempts to remedy slow convergence. The method for remedying slow convergence is controlled with the ADAPTIVE STRATEGY command line, which can have a value of SWITCH, UPDATE, or NONE. The default strategy is to SWITCH from the tangent preconditioner to the nodal preconditioner. The formula for computing the convergence rate is given by

$$abs((||R_{n-1}|| - ||R_n||)/||R_{n-1}||) \tag{4.14}$$

where R is the residual force and n is the iteration counter. In the SWITCH strategy, if the convergence rate is less than the value of min_conv_rate but greater than the value of stagnation, the CG solver will switch to using the fall-back nodal preconditioner. In the UPDATE strategy, the full tangent preconditioner would be updated instead. With either option SWITCH or UPDATE, if the convergence rate is below the value of stagnation, the solver will switch to the nodal preconditioner. The adaptive strategy can be turned off via the NONE option. When using this option, the iterations will continue with the existing preconditioner until an update or switch is performed by some other means.

4.3.6. CG with Full Tangent Preconditioner Input Example

```
begin solver
  Begin cg
```

```
    target relative residual = 1.0e-06
    Maximum Iterations       = 100
    # uses FETI as the default linear equation solver
    begin full tangent preconditioner
      tangent diagonal scale = 1e-6
    end
  end
end
```

4.4. FETI Equation Solver

FETI is a domain-decomposition-based parallel iterative linear solver that can be used to compute the action of the global tangent stiffness matrix for the nonlinear CG solver [3, 4]. FETI uses a direct solver on each domain and iteratively solves for Lagrange multiplier fields at the domain boundaries.

FETI also generates a coarse grid matrix from the global tangent stiffness matrix. The coarse matrix is used to accelerate convergence and distribute the error more evenly across the local mesh domains. The highest eigenvalue λ_1 and lowest eigenvalue λ_N of the coarse matrix are relatively inexpensive to calculate. These eigenvalues are then used to compute the condition number of the coarse matrix $\kappa = \lambda_1/\lambda_N$. If κ is excessively large, then a warning is printed to the log file, indicating that the problem is ill-posed.

Under typical usage, the FETI domains correspond to the portions of the model owned by each processor. If a model is run on a single processor, FETI behaves as a direct solver. Because large models are typically run using many processors, FETI uses significantly lower total computing resources than a direct solution method of the whole problem would require. The large number of small direct solutions may be performed in parallel.

Although a few other linear solvers are available for use with the full tangent preconditioner in the CG solver, it is recommended that FETI be used. FETI is actively maintained and tested by the Sierra/SM development team. FETI has been demonstrated as a robust parallel solver has been used successfully on a wide range of production analyses. The command block for the FETI equation solver is as follows:

```
BEGIN FETI EQUATION SOLVER <string>name
  #
  # convergence commands
  MAXIMUM ITERATIONS = <integer>max_iter(500)
  RESIDUAL NORM TOLERANCE = <real>residual_tol(1.0e-3)
  #
  # diagnostic commands, off by default.
  # "solver" turns on print statements from FETI
  # "matrix" prints the element matrices to the log file
  # "stats" prints time and iteration statistics
  # "export_matrices" exports the matrix to a MATLAB file
  # (in serial)
  PARAM-STRING "debugMask" VALUE <string>"solver"|"matrix"
  #
  # Hinge detection (bad decomposition topologies)
  # Use stk_balance to avoid bad decompositions
  PARAM-STRING "detect_bad_topology" VALUE <string>"true"
  #
  # Export cubit journal files with bad topologies for viz
  PARAM-STRING "debug_bad_topology" VALUE <string>"true"
  #
  # memory usage commands
```

```
  PARAM-STRING "precision" VALUE <string>"single"|"double"
    ("double")
  #
  # Lumped uses less memory, Dirichlet improves convergence
  PRECONDITIONING METHOD = NONE|LUMPED|DIRICHLET(DIRICHLET)
  MAXIMUM ORTHOGONALIZATION = <integer>max_orthog(1000)
  #
  # solver commands
  LOCAL SOLVER  = SKYLINE|SPARSE|ITERATIVE(SPARSE)
  COARSE SOLVER = SKYLINE|SPARSE|ITERATIVE(SPARSE)
  #
  # This command is only used in conjunction with
  # local solver = iterative.
  NUM LOCAL SUBDOMAINS = <integer>num_local_subdomains
  #
  # factorization tolerance for local domain matrix
  PARAM-REAL "local_rbm_tol" value <real>(1e-12)
  #
  # factorization tolerance for global coarse matrix
  PARAM-REAL "global_rbm_tol" value <real>(1e-9)
  #
  # linear residual stagnation tolerance
  PARAM-REAL "stagnation_tol" value <real>(1e-6)

END [FETI EQUATION SOLVER <string>name]
```

The command lines used to control FETI all reside in the `FETI EQUATION SOLVER` command block, where `name` identifies the particular command block. This command block, as with all equation solver command blocks, is placed in the SIERRA scope and referenced by `name` when it is used. Although many command lines are available to control the behavior of FETI, the default settings generally work well for the vast majority of problems. Thus, it is recommended that all default settings be used unless special behavior is desired because of the unique features of a specific model. The command lines in the `FETI EQUATION SOLVER` command block are described in Sections 4.4.1 through 4.4.3.

4.4.1. Convergence Commands

```
MAXIMUM ITERATIONS = <integer>max_iter(500)
RESIDUAL NORM TOLERANCE = <real>residual_tol(1.0e-3)
```

These command lines provide controls on the convergence of the FETI iterative solver and belong in the `FETI EQUATION SOLVER` command block.

The `MAXIMUM ITERATIONS` command line sets the maximum number of iterations allowed per FETI solve. The default value of the parameter `max_iter` is 500. A FETI solve occurs once per CG iteration when used with the full tangent preconditioner.

The RESIDUAL NORM TOLERANCE command line sets the convergence criterion for the FETI solver. The default value of the parameter residual_tol is 1.0e-3. The relative residual norm is compared against the linear residual $\|Ax - b\|$ divided by the norm of current residual, $\|b\|$ where b is the residual force, A is the tangent stiffness matrix, and x is the linear solution vector. If convergence is not reached before the iteration count exceeds max_iter, FETI will return the current approximate gradient direction to the CG solver and continue iterating. The code will not exit with an error. The default settings for both of these command lines are reasonable for most models and typically should not be modified.

4.4.2. Memory Usage Commands

```
PARAM-STRING "precision" VALUE <string>"single"|"double"
  ("double")
PRECONDITIONING METHOD = NONE|LUMPED|DIRICHLET(DIRICHLET)
MAXIMUM ORTHOGONALIZATION = <integer>max_orthog(1000)
```

The command lines listed above can be placed in the FETI EQUATION SOLVER command block to enable optional memory-saving features of the FETI solver. These features will adversely impact the solver effectiveness to some degree, but they can be useful if the memory requirements of a model exceed the capacity of the machine on which the model is run. Before using these features, it is important to consider that on a distributed memory cluster, spreading the model out over more processors can reduce the memory requirements on each processor. For this reason, it is often better to use more processors rather than use the options described here.

FETI has the option of using either single or double precision for storage of internal variables. The default behavior is to use double precision, and this is typically recommended. To select single precision, the user would specify the command line as follows: PARAM-STRING "precision" VALUE "single". Using single-precision variables within FETI can reduce the memory requirements by roughly a factor of two. Use of single precision may, however, degrade the performance of the solver, requiring more iterations within FETI or more CG iterations. Using single precision in FETI does not affect the Sierra/SM data structures, which are always double precision, and therefore does not adversely affect solution ultimate accuracy.

The PRECONDITIONING METHOD command line selects the preconditioning method that is used internally within FETI. The default option, DIRICHLET, typically results in the best convergence rate. The LUMPED option uses less memory, but it usually results in more iterations within FETI. This option should only be used if there are constraints on memory usage. The NONE option uses no preconditioner and is included only for completeness. This option is not of practical interest.

Like the CG solver, the FETI equation solver stores a set of search directions to ensure that the search direction used in each iteration is orthogonal to previous search directions. The number of search directions stored is controlled with the MAXIMUM ORTHOGONALIZATION command line. The default value of 1000 for max_orthog provides optimal convergence for linear solves that take less than 1000 iterations. Setting max_orthog to a lower number can decrease memory usage however, as with the other options discussed above, may require more iterations for

convergence.

4.4.3. Solver Commands

```
LOCAL SOLVER  = SKYLINE|SPARSE|ITERATIVE(SPARSE)
COARSE SOLVER = SKYLINE|SPARSE|ITERATIVE(SPARSE)
NUM LOCAL SUBDOMAINS = <integer>num_local_subdomains
PARAM-REAL "local_rbm_tol" value <real>(1e-12)
PARAM-REAL "global_rbm_tol" value <real>(1e-9)
```

These command lines control the type of solver used by FETI for solving the linear system that arises from the coarse grid and for the local solution mesh linear system. The `LOCAL SOLVER` command line is used to select the solver for the local subdomains. Similarly, the solver for the coarse grid can be selected with the `COARSE SOLVER` command lines. Both of these command lines allow the same three options: `SPARSE`, `SKYLINE`, and `ITERATIVE`, where `SPARSE` is the typical default. The coarse solver default is set based on the number of processors being used in the problem. For processor counts less than 3, the `SKYLINE` solver is the default coarse grid solver. For processor counts greater than 3 and less than 2000, the `SPARSE` solver is the default coarse grid solver. For processor counts greater than 2000, the `ITERATIVE` coarse grid solver is the default.

The FETI algorithm uses these direct solvers to factorize the local matrix and coarse grid matrices before iterating on the solution. These factorization algorithms use tolerances to detect singularities in the matrix. The default values for `local_rbm_tol` and `global_rbm_tol` are recommended in most situations.

- The `SPARSE` option uses a sparse matrix storage direct solver. The sparse matrix is factored into an $A = LDL^T$ decomposition. FETI takes advantage of equation orderings to reduce matrix fill-in. The default equation ordering is done by calling the METIS sparse matrix ordering algorithm. This solver is recommended for both speed and memory efficiency.
- The `SKYLINE` option uses a skyline (profile) matrix storage direct solver. The skyline matrix is factored into an $A = LDL^T$ decomposition. This method is robust, detects rigid-body modes effectively, and includes the reverse Cuthill-McKee (RCM) and Sloan equation orderings. This solver uses more memory than does the sparse solver.
- The `ITERATIVE` option implements a multiple domain FETI algorithm for the local solver or the coarse solver. The number of domains for the local solver is either computed based on internal heuristics or input by the user. The number of domains for the coarse solver is defaulted to 2 which means that the iterative coarse solver will run on 2 processors. This option is recommended in special circumstances where memory becomes an issue with the sparse direct solver or the problem is being run on greater than 2000 processors.

The `NUM LOCAL SUBDOMAINS` command line sets the number of local subdomains for the `ITERATIVE` local solver. This command line is not used for the other local solvers. As the number of local subdomains increases, the size of the local subdomains decreases, thus reducing the memory requirements and the time it takes for the local matrix factorizations. As the number of subdomains increases, the size of the coarse grid increases, thus requiring increased

factorization time and memory. A good rule of thumb is to set the number of local subdomains to the total number of elements in the mesh divided by 400. For a serial run, if the `NUM LOCAL SUBDOMAINS` command line is not specified, FETI sets the number of domains to the greater of 2 or the number of elements divided by 400. For parallel runs, FETI uses the same number of domains as that used by Sierra/SM.

4.4.4. CG with Full Tangent Preconditioner and FETI Input Example

The following example defines a FETI solver for use in a CG preconditioner. The linear solver `my_feti` references a FETI solver with a non-default parameter set. This example sets the preconditioning method to the lumped method to reduce memory usage.

```
...
    Begin solver
      Begin cg
        target relative residual = 1.0e-09
        Maximum Iterations       = 100
        begin full tangent preconditioner
          linear solver = my_feti
        end
      end
    end
  end adagio region adagio
end adagio procedure Agio_Procedure

begin feti equation solver my_feti
  preconditioning method = lumped
end
```

4.4.5. Printing Stiffness Matrix

The following example shows how to print the element stiffness matrices of your model. This option will produce a large amount of output, so it is not advised for large meshes or runs with a high numbers of time steps.

```
BEGIN FETI EQUATION SOLVER feti
  #
  # diagnostic commands, off by default.
  # "solver" turns on print statements from FETI
  # "matrix" prints the element matrices to the log file
  # "export_matrices" exports the matrix to a MATLAB file (in serial)
  PARAM-STRING "debugMask" VALUE "solver"

  # This line sets the number of decimal precision
```

```
  # for the stiffness matrix output:
  param-int "print_precision" value 15
END
```

When using this command, you will need to run your problem in serial (1 processor) with a full tangent preconditioner. Remember to specify this `FETI` solver using the `LINEAR SOLVER` option in the `FULL TANGENT PRECONDITIONER` command block. The individual element matrices will be printed to the log file in a format that is readable by Matlab.

4.5. Control Contact

The multilevel solution control scheme used for contact is called control contact. Control contact alleviates some of the challenges associated with the nonlinear effects of contact by incrementally enforcing the contact nonlinearity outside the core solver. The specific steps taken during a control contact iteration depends on whether augmented Lagrange or kinematic enforcement is being used. Generally, the interactions (node-face or face-face pairs) that are being enforced are only changed once at the start of each control contact iteration and then held fixed during the core CG solve.

For Augmented Lagrange contact, see section 4.5.2 for a detailed description.

The command block for control contact is:

```
BEGIN CONTROL CONTACT
  #
  # Convergence commands
  TARGET RESIDUAL = <real>target_resid
    [DURING <string list>period_names]
  TARGET RELATIVE RESIDUAL = <real>target_rel_resid(1.0e-3)
    [DURING <string list>period_names]
  TARGET RELATIVE CONTACT RESIDUAL =
    <real>target_rel_cont_resid
    [DURING <string list>period_names]
  ACCEPTABLE RESIDUAL = <real>accept_resid
    [DURING <string list>period_names]
  ACCEPTABLE RELATIVE RESIDUAL = <real>accept_rel_resid(1.0e-2)
    [DURING <string list>period_names]
  ACCEPTABLE RELATIVE CONTACT RESIDUAL =
    <real>accept_rel_cont_resid(1.0e-1)
    [DURING <string list>period_names]
  MINIMUM ITERATIONS  = <integer>min_iter(0)
    [DURING <string list>period_names]
  MAXIMUM ITERATIONS  = <integer>max_iter(100)
    [DURING <string list>period_names]
  #
  # level selection command
  LEVEL = <integer>control_contact_level(1)
  #
  # Augmented Lagrange enforcement commands used with
  # enforcement = al defined in the contact definition
  #
  # Gap Enforcement Options
  LAGRANGE ACCEPTABLE RELATIVE GAP = <real>(0.90)
    [DURING <string list>period_names]
  LAGRANGE ACCEPTABLE GAP = <real>(-1)
    [DURING <string list>period_names]
  #
```

```
  # Adaptive Penalty Options
  LAGRANGE INITIALIZE = NONE|MULTIPLIER|PENALTY|BOTH(MULTIPLIER)
    [DURING <string list>period_names]
  LAGRANGE ADAPTIVE PENALTY = OFF|SEPARATE|UNIFORM(OFF)
    [DURING <string list>period_names]
  LAGRANGE ADAPTIVE PENALTY GROWTH FACTOR = <real>(1.5)
    [DURING <string list>period_names]
  LAGRANGE ADAPTIVE PENALTY REDUCTION FACTOR = <real>(2.0/3.0)
    [DURING <string list>period_names]
  LAGRANGE ADAPTIVE PENALTY THRESHOLD = <real>(0.25)
    [DURING <string list>period_names]
  LAGRANGE MAXIMUM PENALTY MULTIPLIER = <real>(100.0)
    [DURING <string list>period_names]
  LAGRANGE MULTIPLIER PREDICTION SCALE FACTOR =<real>(0.85)
    [DURING <string list>period_names]
  #
  # Lagrange Tolerance Options
  LAGRANGE TOLERANCE = <real>(0.0)
    [DURING <string list>period_names]
  #
  # Miscellaneous Options
  LAGRANGE MAXIMUM UPDATES = <integer>(infinity)
    [DURING <string list>period_names]
  LAGRANGE NODAL STIFFNESS MULTIPLIER = <real>(0.0)
    [DURING <string list>period_names]
  #
  # Line search commands
  MAXIMUM LINE SEARCH STEP = <real>(2.0)
    [DURING <string list>period_names]
  MINIMUM LINE SEARCH STEP = <real>(0.01)
    [DURING <string list>period_names]
  #
  # Diagnostic output commands, off by default.
  ITERATION PLOT = <integer>iter_plot
    [DURING <string list>period_names]
  ITERATION PLOT OUTPUT BLOCKS = <string list>plot_blocks
END [CONTROL CONTACT]
```

For a problem containing a contact definition, control contact will be created by default if not defined in the input file. The default control contact will attempt to solve to a relative residual tolerance of 1.0e-3 and will exist at level 1 of the solver. If non-default or advanced options are desired the control contact block must be defined directly.

4.5.1. Control Contact Commands

The line commands within the `CONTROL CONTACT` command block are used to control convergence during the contact updates, select the level for the contact control within the

multilevel solver, and output diagnostic information. These commands are described in detail in sections Section 4.5.1.1 and 4.1.1.

4.5.1.1. Convergence Commands

The command lines listed in this section specify convergence criteria for the contact updates.

```
TARGET RESIDUAL = <real>target_resid
  [DURING <string list>period_names]
TARGET RELATIVE RESIDUAL =
  <real>target_rel_resid(1.0e-3)
  [DURING <string list>period_names]
TARGET RELATIVE CONTACT RESIDUAL = <real>target_rel_cont_resid(1.0e-3)
  [DURING <string list>period_names]
ACCEPTABLE RESIDUAL = <real>accept_resid
  [DURING <string list>period_names]
ACCEPTABLE RELATIVE RESIDUAL = <real>accept_rel_resid(1.0e-2)
  [DURING <string list>period_names]
ACCEPTABLE RELATIVE CONTACT RESIDUAL =
  <real>accept_rel_cont_resid(1.0e-1)
  [DURING <string list>period_names]
MINIMUM ITERATIONS = <integer>min_iter(0)
  [DURING <string list>period_names]
MAXIMUM ITERATIONS = <integer>max_iter(100)
  [DURING <string list>period_names]
```

Solver convergence is monitored for control contact in the same way as for the core CG solver. With the exception of two additional command lines used for controlling the residual due to contact, these command lines are the same as those for the core CG solver (described in Section 4.2.1). Here, however, these command lines are applied to the contact control level.

Contact convergence is evaluated by computing the L^2 norm of the residual forces and comparing that with a target convergence criteria specified by the user in the control contact block. The relative residual is computed by dividing the residual by a reference quantity that is indicative of the current loading conditions on the model. Note the residual has units of force (or energy) while the relative residual is a dimensionless quantity where one or larger is an unconverged solution and the value moves towards zero for increasingly exact convergence.

The `TARGET RESIDUAL` command specifies the target convergence residual in terms of the actual residual norm. The `TARGET RELATIVE RESIDUAL` command specifies a convergence criterion in terms of the relative residual. If both command lines are specified, the multilevel solver will accept the contact solution as converged if either the target residual or the target relative residual is below the specified values.

The `TARGET RELATIVE CONTACT RESIDUAL` command line specifies a relative residual tolerance for just the nodes in contact. When this command is used this criterion must be satisfied in addition to the target residual or relative residual for all nodes. The default value for the target

relative contact residual is the same value as the target relative residual or 10 times the target residual if the target relative residual has not been specified. As the contact residual is the residual on the subset of nodes of the model currently in contact the contact residual will always be strictly lower than the standard residual. For the TARGET RELATIVE CONTACT RESIDUAL to have any effect the value must be set below the TARGET RELATIVE RESIDUAL. A small TARGET RELATIVE CONTACT RESIDUAL command can be used to give improved confidence in the accuracy of the solution in the contact zone.

Acceptable convergence criteria can also be used for contact convergence. If the solution has not converged to the specified target before the maximum number of iterations is reached, the residual is checked against the acceptable convergence criteria. These criteria are specified via the ACCEPTABLE RESIDUAL, ACCEPTABLE RELATIVE RESIDUAL, and ACCEPTABLE RELATIVE CONTACT RESIDUAL command lines. The concepts of acceptable residual, relative residual, and relative contact residual are the same as those used for the targets. If the solution has not met the target criteria but has met the acceptable criteria, the solution is allowed to proceed. The defaults for each of these acceptable criteria are 10 times the corresponding target criteria defaults.

Control contact will use the same reference type (EXTERNAL, BELYTSCHKO, ENERGY, etc.) as the underlying CG solver block.

The MAXIMUM ITERATIONS and MINIMUM ITERATIONS command lines specify the maximum and minimum number of contact updates, respectively. The default minimum number of iterations, min_iter, is 0. If a number greater than 0 is specified, the multilevel solver will update contact at least that many times, regardless of whether the convergence criteria have been met.

For an explanation of commands common to all controls, see section 4.1.1.

4.5.1.2. Tips on Setting Control Contact and CG tolerances

Generally the target residual in the CG block should be set to about an order of magnitude below the target residual in the CONTACT block, while the acceptable residual in the CG block should be set to a very large number. The acceptable should be large enough to assure solution acceptance for any remotely reasonable result.

For each control contact step, a set of active contact constraints is held fixed. The core CG solver subsequently solves the analysis problem subject to a linearized version of those active contact constraints. Occasionally control contact will pick a set of constraints that cannot be solved; for example, creating two constraints that conflict such that the solution is over determined. When this happens, the CG solver will never be able to drop the residual of the over-constrained system. In this case, contact needs to try again to pick a better constraint set. When CG hits the maximum number of iterations and meets the high acceptable tolerance, it will return to the contact solver. Though the residual is still high, it usually has enough information to at least help contact select an improved constraint set. Note, when using control contact the actual true tolerance of the solution is the control contact residual. The CG residual is used for the intermediate model problems. Thus, if control contact eventually converges to a reasonably small residual, the

solution itself is correct even if some of the intermediate CG steps only converged on a high acceptable residual.

4.5.1.3. Augmented Lagrange Gap Enforcement Options

```
# Gap Enforcement Options
LAGRANGE ACCEPTABLE RELATIVE GAP = <real>(0.90)
  [DURING <string list>period_names]
LAGRANGE ACCEPTABLE GAP = <real>(-1)
  [DURING <string list>period_names]
```

The `LAGRANGE ACCEPTABLE RELATIVE GAP` and `LAGRANGE ACCEPTABLE GAP` can be used to ensure that the penetration or gap is sufficiently removed before moving on to the next loadstep. Augmented Lagrange enforcement applies equal and opposite forces with each control contact iteration. These options force contact iterations to continue until both residual balance and a sufficient amount of penetration or gap is removed.

The `LAGRANGE ACCEPTABLE RELATIVE GAP` by default is set to 0.90. The control contact solver level will continue iterating until the largest contact relative gap value is less than or equal to the specified value. The relative gap is the gap value divided by the respective interaction tolerance and is unitless. Failing to converge contact to a relative gap below 1.0 risks missing contact on the subsequent loadstep.

The `LAGRANGE ACCEPTABLE GAP` by default is not specified. The control contact solver level will keep iterating until the largest contact gap value is less than or equal to the specified value. The gap value specified is the actual largest gap value and has the units of length. This command is most appropriately used when model dimensions are well understood, manual normal tolerances have been set, and a known gap tolerance is desired.

4.5.1.4. Augmented Lagrange Initialization Commands

```
LAGRANGE INITIALIZE = NONE|MULTIPLIER|PENALTY|BOTH(MULTIPLIER)
  [DURING <string list>period_names]
```

The `LAGRANGE INITIALIZE` command is used to control how Lagrange multipliers are set at the beginning of a load step. The default value of `MULTIPLIER` specifies that the Lagrange multipliers will be initialized to some factor of the previous step Lagrange multipliers. By default this multiplication factor is `0.85` and can be changed with the `LAGRANGE MULTIPLIER PREDICTION SCALE FACTOR` command.

The `NONE` option turns off initialization of the Lagrange multipliers, setting them to zero. This option may aid convergence in some very nonlinear analyses where the next load step is in a completely different contact state than the previous load step. The `PENALTY` option initializes Lagrange multipliers based off of material stiffness and gap. The `BOTH` initializes Lagrange multipliers based off of both the old multiplier and the material stiffness and gap in an attempt to

achieve a more efficient prediction. It is generally recommended that only the MULTIPLIER or NONE options be used as the penalty based initialization is experimental at this time.

4.5.1.5. Augmented Lagrange Adaptive Penalty Commands

```
# Adaptive Penalty Options
LAGRANGE ADAPTIVE PENALTY = OFF|SEPARATE|UNIFORM(OFF)
  [DURING <string list>period_names]
LAGRANGE ADAPTIVE PENALTY GROWTH FACTOR = <real>(1.5)
  [DURING <string list>period_names]
LAGRANGE ADAPTIVE PENALTY REDUCTION FACTOR = <real>(2.0/3.0)
  [DURING <string list>period_names]
LAGRANGE ADAPTIVE PENALTY THRESHOLD = <real>(0.25)
  [DURING <string list>period_names]
LAGRANGE MAXIMUM PENALTY MULTIPLIER = <real>(100.0)
  [DURING <string list>period_names]
LAGRANGE MULTIPLIER PREDICTION SCALE FACTOR =<real>(0.85)
  [DURING <string list>period_names]
```

The command lines listed above can be used to control the augmented Lagrange contact adaptive penalty algorithm. Augmented Lagrange contact enforcement is activated when `enforcement = al` is set in the contact definition block.

The LAGRANGE ADAPTIVE PENALTY command indicates how the penalty multiplier will be adjusted as the problem progresses. The option SEPARATE scales the penalty differently for each contact interaction. The option UNIFORM scales all interactions equally. The default option is OFF.

The LAGRANGE ADAPTIVE PENALTY GROWTH FACTOR command sets the adaptive penalty growth factor, while the LAGRANGE ADAPTIVE PENALTY REDUCTION FACTOR command sets the adaptive penalty reduction factor. The LAGRANGE ADAPTIVE PENALTY THRESHOLD sets the adaptive penalty threshold. The LAGRANGE MAXIMUM PENALTY MULTIPLIER sets the maximum penalty multiplier, which is the upper bound on the penalty multiplier when running augmented Lagrange contact.

In augmented Lagrange contact, if the current gap magnitude is greater than the previous gap magnitude, the penalty multiplier is reduced by multiplying it by the penalty reduction factor. If the current gap magnitude is between the previous gap magnitude and the previous gap magnitude times the penalty threshold, the penalty multiplier is increased by multiplying it by the penalty growth factor. The lower bound for the penalty multiplier is 1.0 and the upper bound is the maximum penalty multiplier.

The LAGRANGE MULTIPLIER PREDICTION SCALE FACTOR sets a scaling factor on the Lagrange multipliers when moving from one load step to the next. Generally the converged Lagrange multipliers at a new load step will be close to the multipliers obtained in the previous load step. When predicting the next configuration, a default scaling factor of 0.85 is applied to the previous Lagrange multipliers to reduce the danger of overshooting the contact solution. A

value closer to 1.0 may accelerate convergence but may also introduce issues by overshooting the true contact solution.

4.5.1.6. Augmented Lagrange Tolerance

```
# Tolerance Options
LAGRANGE TOLERANCE = <real>(0.0)
  [DURING <string list>period_names]
```

The LAGRANGE TOLERANCE can be used to set the Lagrange multiplier tolerance for updating. Once the change to the Lagrange multiplier is less than the tolerance, the Lagrange multiplier will no longer be updated. The default is 0.0, which always updates the Lagrange multiplier.

4.5.1.7. Augmented Lagrange Miscellaneous Options

```
# Miscellaneous Options
LAGRANGE MAXIMUM UPDATES = <integer>(infinity)
  [DURING <string list>period_names]
  [DURING <string list>period_names]
LAGRANGE NODAL STIFFNESS MULTIPLIER = <real>(0.0)
  [DURING <string list>period_names]
```

The LAGRANGE MAXIMUM UPDATES command sets the maximum number of augmented Lagrange updates performed during a load step. By default, the Lagrange multiplier is always updated. Setting this to zero makes contact a pure penalty formulation.

The LAGRANGE NODAL STIFFNESS MULTIPLIER sets a scale factor on the amount of contact stiffness included in the nodal preconditioner. The default value is 0.0, which means no contact stiffness is present in the nodal preconditioner. A value of 1.0 distributes the full contact stiffness to the contact nodes and a value much greater than 1.0 effectively pins (or fixes) the contact nodes in the nodal preconditioner.

4.5.2. Augmented Lagrange Control Contact

For augmented Lagrange enforcement, a search is only performed once in the predicted configuration of a new load step. It is crucial that the predicted configuration be reasonable. Experience has shown that no prediction is often best, so it is recommended that the load step predictor be turned off when using augmented Lagrange enforcement for contact. This can be accomplished by setting the scale factor predictor to zero (See Section 4.9.1).

The contact search provides a set of all possible interactions. In the log file example in Section 4.5.2.1, the listed NUM INTERACTIONS is the number of possible interactions that were found in the contact search. Each interaction is a node-face or face-face pair, depending on the constraint formulations used in the contact definition.

Interactions can exist in one of three states: RELEASED, CAPTURED, or DUBIOUS. Tied interactions are determined once during the search and can only be CAPTURED or RELEASED. Other interactions start in a DUBIOUS or RELEASED state after the search. To reduce chatter, these interactions can only move to CAPTURED or RELEASED by moving through the DUBIOUS state.

The DUBIOUS state serves as an intermediate state. During a model problem, interactions in the DUBIOUS state are enforced with a nonlinear penalty kernel during the model problem, while interactions in the CAPTURED state use a linear penalty kernel. This is why the first model problem is often the most difficult. The logic to move interactions between states is based on the gapping history combined with the value of the accumulated Lagrange multiplier.

During a contact update, the Lagrange multiplier is augmented before the state determination by adding the penalty kernel to the Lagrange multiplier. This update causes an additional residual imbalance from what was solved for in the model problem. If this update is sufficiently small, you will converge at the control contact level and will not return to level 0 to solve the residual imbalance. The maximum relative change in the value of the Lagrange multiplier is listed in the log file as RELATIVE LMULT CHANGE. Because the convergence is determined exclusively on the residual balance, it is sometimes desirable to set a LAGRANGE TOLERANCE so that if the change in the Lagrange multiplier is small enough, the update is not done. This preserves the residual balance determined in the model problem, assuming there is no change in the interaction states.

As contact iterations progress, the values of MAX GAP, MAX RELATIVE GAP and RELATIVE LMULT CHANGE should decrease and the set of interactions should stop changing (signified by the output line NO ACTIVE SET CHANGE). Successive iterations should report a RELATIVE RESIDUAL for control contact closer to the RELATIVE RESIDUAL reported in the model problem until convergence is reached.

Many of the log file quantities described here are also stored as global variables (see Table 9-3).

4.5.2.1. Augmented Lagrange Control Contact Log File Example

```
================================================================================================================
Begin load step =    37 Solution period Adagio_Procedure_time_control_1 is  14.8% complete
        Old Time       Time Step        New Time       Stop Time     CPU Time(s)    Wall Time(s)
      7.4000e-03      2.0000e-04      7.6000e-03      5.0000e-02      3.8784e+00      4.0469e+00
----------------------------------------------------------------------------------------------------------------
                                        MP                   RELATIVE     EXTERNAL
                                      ITER     RESIDUAL     RESIDUAL    REFERENCE      ENERGY DISPLACEMENT
----------------------------------------------------------------------------------------------------------------
                                         0    9.358e-03    6.885e-02    1.359e-01           -            -
                                        16    1.814e-06    1.595e-05<T  1.137e-01   4.634e-17    8.057e-08
----------------------------------------------------------------------------------------------------------------
MAX GAP               = 2.779e-05 PREVIOUS = 0.000e+00
MAX RELATIVE GAP      = 6.175e-03 PREVIOUS = 0.000e+00
NUM INTERACTIONS      = 60
RELEASED INTERACTIONS = 0
CAPTURED INTERACTIONS = 60
DUBIOUS INTERACTIONS  = 0
RELATIVE LMULT CHANGE = 1.465e-01
ACTIVE SET CHANGE
----------------------------------------------------------------------------------------------------------------
CONTACT ITERATION =            0
ABSOLUTE RESIDUAL =    3.444e-03
RELATIVE RESIDUAL =    2.914e-02
----------------------------------------------------------------------------------------------------------------

... ( Skipped Contact Iteration 1 - 9 for readability ) ...
```

```
------------------------------------------------------------------------------------------------------
                                        MP                  RELATIVE     EXTERNAL
                                      ITER     RESIDUAL     RESIDUAL    REFERENCE      ENERGY DISPLACEMENT
------------------------------------------------------------------------------------------------------
                                         0    8.104e-05    6.810e-04    1.190e-01           -            -
                                         5    1.295e-06    1.088e-05<T  1.190e-01   9.540e-19    2.180e-08
------------------------------------------------------------------------------------------------------
MAX GAP               = 1.136e-06 PREVIOUS = 1.129e-06
MAX RELATIVE GAP      = 1.514e-04 PREVIOUS = 1.506e-04
NUM INTERACTIONS      = 60
RELEASED INTERACTIONS = 4
CAPTURED INTERACTIONS = 56
DUBIOUS INTERACTIONS  = 0
RELATIVE LMULT CHANGE = 7.380e-03
NO ACTIVE SET CHANGE
------------------------------------------------------------------------------------------------------
CONTACT ITERATION =           10
ABSOLUTE RESIDUAL =    5.179e-05
RELATIVE RESIDUAL =    4.352e-04<T
-------------------------------------------------------------------------------
   LEVEL 1                                                  ITERATIONS          TOTAL
   CONTACT     MODEL PROB      LOAD STEP        AVERAGE          TOTAL         CPU(s)
 CONVERGED             11            114             10           4069      4.825e+00
```

4.5.3. Kinematic Control Contact

For kinematic enforcement, a search is performed each contact iteration to provide a set of nodes in contact. This constraint set is held constant during the subsequent model problem. For frictional contact, side B nodes are fixed to side A faces during the CG iterations. For frictionless contact, the side B nodes are fixed in the normal direction but are allowed to slide during the CG iterations. After model problem convergence, the constraint set is updated to reflect the changing contact conditions. This update gives rise to a force imbalance to be solved in another model problem.

Changing or updating the constraint set is called a contact update. Multiple contact updates are typically required before equilibrium is achieved. The contact update includes a search, a gap removal, an equilibrium query, and a slip calculation if equilibrium is not satisfied.

New constraints are detected in the search phase based on the current deformed configuration of the model. If a node is found to penetrate a side A surface, the node is added to the set of constraints.

The gap removal phase involves creating or destroying constraints and calculating push-back vectors for side B nodes. A penetration may be removed at once (the default behavior), or it may be incrementally removed over a number of model problems. For interactions where contacting surfaces are free to separate, a side B node constraint continues to exist as long as there is a compressive force between the side B node and the side A surface. The constraint changes during the gap removal phase to reflect changes in the shape and orientation of the side A surface. Constraints are destroyed when a tensile force exceeding a specified tolerance (see Section 8.6.2.7) exists at the contact interface.

During the slip portion of the contact update, a residual force is calculated at each node and resolved into normal and tangential components. This force reflects changes in residual due to gap removal that occurred earlier in the update. For frictional contact, the friction coefficient is used to determine the tangential load capacity. If the tangential loads exceed this capacity, nodes

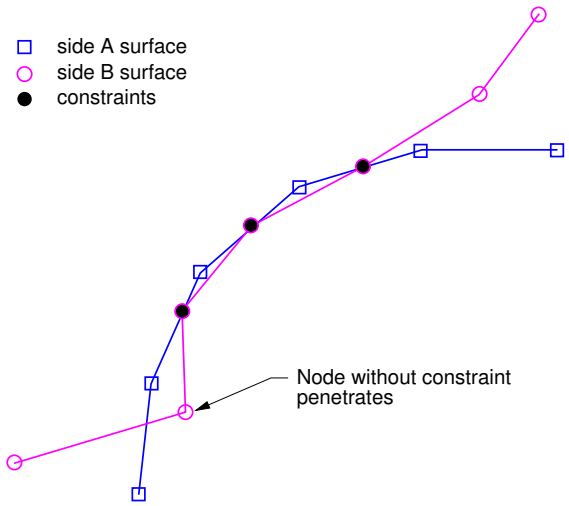


Figure 4-2. Contact configuration at the beginning of the contact update.

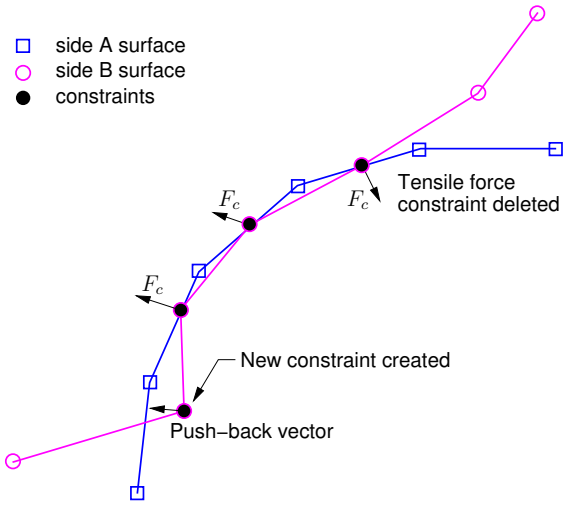


Figure 4-3. Contact gap removal (after contact search).

will slip.

The procedure for kinematic control contact is illustrated in the following sequence of figures. It is assumed in these figures that control contact is the level 1 solver, so the model problems are directly solved by the core solver. In Figure 4-2, the model problem has three constraints that are enforced during iterations of the core solver. However, during solution of the model problem an additional node penetrates the side A surface.

Once the model problem has converged, a contact update is performed. During the search, one node if found to have penetrated the surface, resulting in a new constraint for this node. A push-back vector is calculated for this node to remove the penetration. In the gap removal portion of the update (Figure 4-3), one of the constraints is found to have accumulated a tensile force large enough that surfaces should separate, indicating that this constraint should be eliminated.

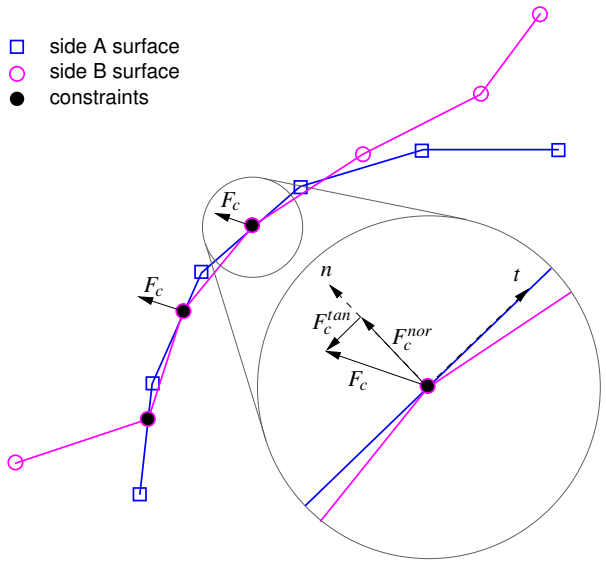


Figure 4-4. Contact slip calculations.

In the second phase of the contact update, slip is allowed to occur along the contact interface. After the gaps are removed, the external and internal force vectors are recomputed, and forces are partitioned along normal and tangential directions of the side A surface. This process is illustrated in Figure 4-4.

4.5.4. CG with Control Contact Input Example

```
begin solver
  begin control contact
    target relative residual = 1.0e-4
    maximum iterations       = 100
  end
  begin cg
    acceptable relative residual = 1.0e10
    target relative residual = 1.0e-6
    maximum iterations       = 100
    iteration print          = 25
    begin full tangent preconditioner
      minimum smoothing iterations = 25
    end
  end
end
```

4.6. Control Stiffness

Control stiffness is an augmented Lagrange iterative solution strategy for solving mechanics models that have *widely* varying stiffness in various modes of material response. For instance, nearly incompressible materials have a bulk/volumetric response that is much stiffer than the corresponding shear/deviatoric response. A similar case arises for materials that are orthotropic or anisotropic in nature and exhibit widely varying stiffness in the principal material directions.

Using control stiffness allows part of the constitutive response of a material to be softened or stiffened to give a tangent response that results in model problems that can be solved more easily by the core solver. This is accomplished by scaling a chosen stress increment up or down in a given model problem and adding it to a saved reference stress to yield a scaled stress that is used for equilibrium evaluations. The reference stress accumulates the model problem stress increments so that the true material response for a time step is obtained after solving a sequence of model problems.

This capability is highly specialized to certain use cases and is not recommended for most problems. It is also only available for use with certain specialized material models. Use of this capability should be considered only as a last resort when convergence cannot be achieved in any other way. Potential users of this feature are advised to first contact the Sierra/SM development team for advice on how to effectively use this capability.

The command block for control stiffness is as follows:

```
BEGIN CONTROL STIFFNESS [<string>stiffness_name]
  #
  # convergence commands
  TARGET <string>RESIDUAL|PRESSURE INCREMENT|
    SDEV INCREMENT|STRESS INCREMENT
    = <real>target [DURING <string list>period_names]
  TARGET RELATIVE <string>RESIDUAL|PRESSURE INCREMENT|
    SDEV INCREMENT|STRESS INCREMENT|
    STRAIN INCREMENT
    = <real>target_rel [DURING <string list>period_names]
  ACCEPTABLE <string>RESIDUAL|PRESSURE INCREMENT|
    SDEV INCREMENT|STRESS INCREMENT
    = <real>accept [DURING <string list>period_names]
  ACCEPTABLE RELATIVE <string>RESIDUAL|PRESSURE INCREMENT|
    SDEV INCREMENT|STRESS INCREMENT|
    STRAIN INCREMENT
    = <real>accept_rel [DURING <string list>period_names]
  REFERENCE = EXTERNAL|INTERNAL|BELYTSCHKO|RESIDUAL|ENERGY(EXTERNAL)
    [DURING <string list>period_names]
  MINIMUM ITERATIONS  = <integer>min_iter(0)
    [DURING <string list>period_names]
  MAXIMUM ITERATIONS  = <integer>max_iter
    [DURING <string list>period_names]
  #
```

```
  # level selection command
  LEVEL = <integer>stiffness_level
  #
  # diagnostic output commands, off by default.
  ITERATION PLOT = <integer>iter_plot
    [DURING <string list>period_names]
  ITERATION PLOT OUTPUT BLOCKS  = <string list>plot_blocks
END [CONTROL STIFFNESS <string>stiffness_name]
```

To enable control stiffness, a `CONTROL STIFFNESS` command block must exist in the `SOLVER` command block. The command lines within the `CONTROL STIFFNESS` command block are used to control convergence during the control stiffness updates, select the level for control stiffness within the multilevel solver, and output diagnostic information. These commands are described in detail in sections Section 4.6.1 and 4.1.1.

4.6.1. Convergence Commands

The command lines listed in this section are placed in the `CONTROL STIFFNESS` command block to control convergence criteria for stiffness control within the multilevel solver.

```
TARGET <string>RESIDUAL|PRESSURE INCREMENT|
  SDEV INCREMENT|STRESS INCREMENT
  = <real>target [DURING <string list>period_names]
TARGET RELATIVE <string>RESIDUAL|PRESSURE INCREMENT|
  SDEV INCREMENT|STRESS INCREMENT|STRAIN INCREMENT
  = <real>target_rel [DURING <string list>period_names]
ACCEPTABLE <string>RESIDUAL|PRESSURE INCREMENT|
  SDEV INCREMENT|STRESS INCREMENT
  = <real>accept [DURING <string list>period_names]
ACCEPTABLE RELATIVE <string>RESIDUAL|PRESSURE INCREMENT|
  SDEV INCREMENT|STRESS INCREMENT|STRAIN INCREMENT
  = <real>accept_rel [DURING <string list>period_names]
REFERENCE = EXTERNAL|INTERNAL|BELYTSCHKO|RESIDUAL|ENERGY(EXTERNAL)
  [DURING <string list>period_names]
MINIMUM ITERATIONS = <integer>min_iter(0)
  [DURING <string list>period_names]
MAXIMUM ITERATIONS = <integer>max_iter
  [DURING <string list>period_names]
```

Convergence for problems using control stiffness requires that two criteria be met. The first criterion is that force equilibrium must be achieved. The second criterion is that the scaled stress used for equilibrium must be close enough to the unscaled stress that is determined from the unmodified material model.

The command lines listed above for the equilibrium evaluation are similar to those used for the core CG solver and have similar meaning. Here, the command lines are applied as part of

determining convergence for the control stiffness series of model problems. Some of the command lines have multiple options, e.g., the command line beginning with TARGET has five options. This practice has been used for the command lines beginning with TARGET, TARGET RELATIVE. ACCEPTABLE, and ACCEPTABLE RELATIVE. That all of these command lines can be appended with the DURING specification, as discussed in Section 4.2.1.

Convergence of equilibrium using scaled stresses is measured by computing the L^2 norm of the residual and comparing that residual norm with target convergence criteria specified by the user. Convergence can be monitored either directly in terms of the residual norm, or in terms of a relative residual, which is the residual norm divided by a reference quantity that is indicative of the current loading conditions on a model. Either the residual, the relative residual, or both of these can be used for checking convergence at the same time. The TARGET RESIDUAL command line specifies the target convergence criterion in terms of the actual residual norm. The TARGET RELATIVE RESIDUAL command line specifies a convergence criterion in terms of the relative residual. If both absolute and relative criteria are specified for the residual calculation, equilibrium is accepted if either criterion is met.

The multilevel solver also allows acceptable convergence criteria to be input for residual convergence. If the solution has not converged to the specified targets before the maximum number of iterations is reached, the residual is checked against the acceptable convergence criteria. These criteria are specified via the ACCEPTABLE RESIDUAL and ACCEPTABLE RELATIVE RESIDUAL command lines. The concepts of absolute and relative values are the same here as discussed for the target limits. When a multilevel solver is being used, the level 0 (cg) acceptable residual and relative residual defaults are overridden and set to $1e20$. This choice was made to improve the overall robustness of the multilevel solver when faced with difficult contact problems. As long as the targets in the controls (contact, stiffness, failure, or damped solve) are met, then the solution is converged and allowed to continue time stepping.

If relative residuals are given in the convergence criteria, the REFERENCE command line can be used to select the method for computing the reference load. This command line has several options: EXTERNAL, INTERNAL, BELYTSCHKO, RESIDUAL and ENERGY. When using the default option (EXTERNAL), the reference value is taken as the maximum of the L^2 norm of either the current externally applied or reaction loads. The INTERNAL option uses the norm of the internal forces as the reference load. The BELYTSCHKO option uses the maximum of the norm of the external load, the norm of the reactions, the norm of the internal forces, and the norm of the inertial forces. The RESIDUAL option denotes that the residual from the initial residual should be used as the reference quantity. The ENERGY residual is the energy conjugate norm of the residual and reference vectors. These criteria are further explained in Section 4.2.1.

For control stiffness, unlike other controls, the convergence of the series of model problems to obtain the final solution for a time step is also based on other criteria that measure how far apart the scaled and unscaled stress responses are from each other. Either a direct error measure of the difference of the scaled and unscaled stress is used or a relative strain error computed using the difference between the scaled and unscaled stress and a representative modulus is employed.

There are many variants of the TARGET command that can be used for the direct stress differences. The TARGET PRESSURE INCREMENT and TARGET SDEV INCREMENT commands are used to specify convergence for nearly incompressible materials based on the pressure and

deviatoric stress increments, respectively. Also, `TARGET STRESS INCREMENT` is used to base convergence on the stress increment for materials which have their entire behavior softened as part of a control stiffness approach. If desired, the `RELATIVE` form of these convergence criteria can be used. If the `TARGET RELATIVE STRAIN INCREMENT` command is specified, the strain error measure between the scaled and unscaled stress will be computed and used.

Any combination of these criteria can be specified. If more than one is specified, all the criteria must be satisfied. However, each material block contributes either to the direct stress error measures or to the relative strain error measure. If a material block specifies a positive non-zero `REFERENCE STRAIN` in its definition, it will contribute to the relative strain error measure. Otherwise, it will contribute to the direct stress error measures. Finally, acceptable convergence criteria can be input for converging the difference between the scaled and unscaled stress. If the solution has not met the target criteria for equilibrium and the difference between the scaled and unscaled stresses but meets the acceptable criteria, it is allowed to proceed to the next time step.

The `MAXIMUM ITERATIONS` and `MINIMUM ITERATIONS` command lines specify the maximum and minimum number of control stiffness updates, respectively. The default minimum number of iterations, `min_iter`, is 0. If a number greater than 0 is specified for `min_iter`, the multilevel solver will update stiffness at least that many times, regardless of whether the convergence criteria have been met.

For an explanation of commands common to all controls, see section 4.1.1.

Separate from the iteration plot commands, control stiffness provides several other element variables for diagnostic output.

- bulk_strain_error
- shear_strain_error
- axial_strain_error
- anisotropic_strain_error

4.7. Control Failure

Control failure is used to improve the conditioning and accuracy of models that involve element death. To use control failure, an element death block (as defined in Section 6.5) and a multilevel solver control failure must both be defined.

When element death is used in the absence of control failure at the end of a load step all elements that meet the element death criteria are removed and the analysis moves on to the next step. There are two issues with this approach control failure will remedy. First death of one element may increase the load on remaining elements which should cause additional element failures. Second death of one element may relieve the load on another element, possibly putting that element below the failure criteria. Thus element death without control failure could either under predict or over predict the actual number of elements that should be removed in a given time step.

Control failure with material criterion limits the failure only to the single most critical element during each control failure iteration. With control failure when an element dies the model problem will be repeated with the element in a failed state in the presence of control failure. Iterations of the control failure continue until no additional elements are marked for failure.

The command block for control failure is as follows:

```
BEGIN CONTROL FAILURE [<string>failure_name]
  #
  #  convergence control command
  MAXIMUM ITERATIONS = <integer>max_iter
    [DURING <string list>period_names]
  #
  # level selection command
  LEVEL = <integer>failure_level
  #
  # diagnostic output commands, off by default.
  ITERATION PLOT = <integer>iter_plot
    [DURING <string list>period_names]
  ITERATION PLOT OUTPUT BLOCKS  = <string list>plot_blocks
END [CONTROL FAILURE <string>failure_name]
```

To enable control failure, a `CONTROL FAILURE` command block must exist in the `SOLVER` command block. The command lines within the `CONTROL FAILURE` command block are used to establish convergence criteria, select the level for control failure within the multilevel solver, and output diagnostic information. These commands are described in detail in sections Section 4.7.1 and 4.1.1.

Convergence of control failure is defined as having no new elements have been marked as failed in the previous model problem. Control failure will also complete if it reaches the `MAXIMUM ITERATIONS` number of model problem evaluations in a single load step.

When element death is not controlled by a material criterion, all elements that satisfy the specified criterion are marked at the end of the model problem.

When a material criterion is used, the material will set the variable "failure_flag" to 1.0 when the failure criterion is met. At the end of the model problem, control failure will set the "failure_flag" of the failed element with the largest "failure_measure" to 2.0, thereby marking it. For the marked element, at the beginning of the following model problem the material model will set the "failure_flag" to 3.0 as it begins decaying the material, while all other instances of "failure_flag" with a value of 1.0 are set back to 0.0. Control failure iterations continue until no elements are marked for failure (i.e. no elements have a "failure_flag" of 2.0) or the `MAXIMUM ITERATIONS` is reached. What this means is that only the most critical element is removed each model problem. This one-by-one element removal prevents potential over prediction of failure if several elements exceed the failure criterion on the same model problem.

4.7.1. Convergence Command

The only user input convergence criteria that is allowed for control failure is specification of maximum iterations.

```
MAXIMUM ITERATIONS = <integer>max_iter
   [DURING <string list>period_names]
```

Iterations for control failure will continue until no new elements are failing unless the user specified number of maximum iterations is reached. If a maximum number of iterations is specified in the input, control failure iterations will only continue until this number is reached. If the maximum number of iterations is reached, the analysis will terminate in an unconverged state and with an explanatory message in the log file. This command can be appended with the `DURING` specification, as discussed in Section 4.2.1. The unconverged state can also trigger adaptive time stepping if it is active.

4.7.2. Level Command

When used in a multilevel solver the `CONTROL FAILURE` command block should be processed at the highest solver level and should be the only control acting at that level. For more details on the `LEVEL` command see Section 4.1.1.

4.7.3. Other Commands

For an explanation of commands related to iteration plot, see section 4.1.1.

4.7.4. Control Failure Input Example

The following example shows the setup for a general multilevel solver involving control contact and control failure. Each load step first a multilevel control contact model problem is solved. Control failure then removes the most failed element based off of the material failure criterion.

Control contact again solves the model problem with that element removed. This process continues until no new elements fail. Due to the large number of iterative resolves required the accuracy provided by control failure for such a situation may come at a substantial run time cost.

```
begin element death
  include all blocks
  material criterion = on
end

Begin solver
  Begin control failure
    level = 2
  end control failure

  Begin control contact
    level = 1
    target relative residual = 1.0e-6
  end control contact

  Begin cg
    target relative residual = 1.0e-8
  end cg
End
```

4.8. Control Damped Solve

Control damped solve is used to aid implicit quasistatic runs in convergence. With implicit quasistatic problems, it is sometimes very difficult to achieve convergence with the cg solver alone (even when used in combination with the FETI equation solver). Some problems where convergence is difficult to achieve includes: problems with unconstrained rigid body motion, problems that contain very nonlinear physics, implicit contact problems that involve rebound or bounce, implicit contact problems where sliding occurs, and implicit fracture problems where failure introduces rigid body motion or softening materials.

At a high level, `CONTROL DAMPED SOLVE` places artificial springs to ground on nodes to allow the problem to converge. Once the springs are placed, `CONTROL DAMPED SOLVE` will iterate until the artificial spring force is below the user defined tolerance. To achieve convergence, the ground position of the springs is moved to the last predicted position at the control level. If the next predicted position is the same, the spring applies no force.

The command block for control damped solve is as follows:

```
BEGIN CONTROL DAMPED SOLVE [<string>damped solve_name]
  # Convergence commands
  TARGET SPRING FORCE = <real>target_spring_force
  TARGET RELATIVE SPRING FORCE = <real>target_rel_spring_force
  ACCEPTABLE SPRING FORCE = <real>accept_spring_force
  ACCEPTABLE RELATIVE SPRING FORCE = <real>accept_rel_spring_force
  REFERENCE = EXTERNAL|INTERNAL(EXTERNAL)
  MAXIMUM ITERATIONS = <integer>max_iter
  MINIMUM ITERATIONS = <integer>min_iter
  #
  # level selection command
  LEVEL = <integer>damped_solve_level
  #
  # artificial spring commands
  STIFFNESS FORM = SHIFTED|SCALED(SCALED)
  STIFFNESS FACTOR = <REAL>stiffness_factor
  FORCE DECAY = NONE|ITERFRAC|HALVED|TENTHED(HALVED)
  EVERY NTH NODE = <integer>nth_node(1)
  #
  # diagnostic output commands, off by default.
  ITERATION PLOT = <integer>iter_plot
    [DURING <string list>period_names]
  ITERATION PLOT OUTPUT BLOCKS  = <string list>plot_blocks
END [CONTROL DAMPED SOLVE <string>damped solve_name]
```

To enable control damped solve, a `CONTROL DAMPED SOLVE` command block must exist in the `SOLVER` command block. The command lines within the `CONTROL DAMPED SOLVE` command block are used to establish convergence criteria, select the level for control damped solve within the multilevel solver, specify characteristics of the artificial springs, and output diagnostic information. These commands are described in detail in sections Section 4.8.1 and 4.1.1.

4.8.1. Convergence Command

The command lines listed in this section specify convergence criteria for the control damped solve level.

```
TARGET SPRING FORCE = <real>target_spring_force
TARGET RELATIVE SPRING FORCE = <real>target_rel_spring_force
ACCEPTABLE SPRING FORCE = <real>accept_spring_force
ACCEPTABLE RELATIVE SPRING FORCE = <real>accept_rel_spring_force
REFERENCE = EXTERNAL|INTERNAL(EXTERNAL)
MAXIMUM ITERATIONS = <integer>max_iter
MINIMUM ITERATIONS = <integer>min_iter
```

Solver convergence is monitored for control damped solve by examining the force in the artificial springs. Convergence is evaluated by computing the L^2 norm of the spring forces and comparing that with a target convergence criteria specified by the user in the control damped solve block. The relative spring force is computed by dividing the spring force by a reference quantity that is indicative of the current loading conditions on the model. Note the spring force has units of force while the relative spring force is a dimensionless quantity where one or larger is an unconverged solution and the value moves towards zero for increasingly exact convergence. When using `CONTROL DAMPED SOLVE`, at least one of the following must be specified: `TARGET SPRING FORCE`, `TARGET RELATIVE SPRING FORCE`, `ACCEPTABLE SPRING FORCE`, or `ACCEPTABLE RELATIVE SPRING FORCE`.

The `TARGET SPRING FORCE` command specifies the target convergence force in terms of the actual spring force norm. The `TARGET RELATIVE SPRING FORCE` command specifies a convergence criterion in terms of the relative spring force norm. If both command lines are specified, the multilevel solver will accept the damped solve solution as converged if either the target spring force norm or the target relative spring force norm is below the specified values.

Acceptable convergence criteria can also be used for damped solve convergence. If the solution has not converged to the specified target before the maximum number of iterations is reached, the spring force is checked against the acceptable convergence criteria. These criteria are specified via the `ACCEPTABLE SPRING FORCE` and `ACCEPTABLE RELATIVE SPRING FORCE` command lines. The concepts of acceptable spring force and acceptable relative spring force are the same as those used for the targets. If the solution has not met the target criteria but has met the acceptable criteria, the solution is allowed to proceed. The defaults for each of these acceptable criteria are 10 times the corresponding target criteria defaults.

If relative spring forces are given in the convergence criteria, the `REFERENCE` command line can be used to select the method for computing the reference load. This command line has two options: `EXTERNAL` and `INTERNAL`. When using the default option (`EXTERNAL`), the reference value is taken as the maximum of the L^2 norm of either the current externally applied or reaction loads. The `INTERNAL` option uses the norm of the internal forces as the reference load. These criteria are further explained in Section 4.2.1.

The `MAXIMUM ITERATIONS` and `MINIMUM ITERATIONS` command lines specify the maximum and minimum number of damped solve updates, respectively. The default minimum number of

iterations, min_iter, is 0. If a number greater than 0 is specified, the multilevel solver will update the damped solve at least that many times, regardless of whether the convergence criteria have been met.

For an explanation of commands common to all controls, see section 4.1.1.

4.8.2. Level Command

When used in a multilevel solver the CONTROL DAMPED SOLVE command block is recommended to be at level 1 and be the only control acting at that level. This allows the artificial spring force to be minimized before considering other controls. For more details on the LEVEL command see Section 4.1.1.

4.8.3. Control Damped Solve Artificial Spring Commands

```
STIFFNESS FORM = SHIFTED|SCALED(SCALED)
STIFFNESS FACTOR = <REAL>stiffness_factor
FORCE DECAY = NONE|ITERFRAC|HALVED|TENTHED(HALVED)
EVERY NTH NODE = <integer>nth_node(1)
```

Commands directly relating to the artificial springs created by CONTROL DAMPED SOLVE are described in this section. STIFFNESS FORM specifies how the stiffness of the artificial springs is determined. If option SHIFTED is used all artificial springs added have the same stiffness. That spring's stiffness is the maximum value of the nodal stiffness from the element stiffness contributions then scaled by the STIFFNESS FACTOR. If the option SCALED is used the artificial springs stiffness is the node's nodal stiffness scaled by the STIFFNESS FACTOR.

The STIFFNESS FACTOR command specifies the scale factor for which to scale the spring stiffness. This scale factor can be between 0.0 and 1.0.

The FORCE DECAY command is used to decrease the stiffness used in the internal force routine (but not the preconditioner) with each control level iteration. The option NONE does nothing to the stiffness. The option ITERFRAC decreases the stiffness by 1 over the current iteration number. The option HALVED decreases the stiffness by half with each iteration. The option TENTHED decreases the stiffness by a factor of 10 with each iteration. The current FORCE DECAY FACTOR is echoed in the log file.

The EVERY NTH NODE command specifies the nodes for which the artificial springs are applied to. If EVERY NTH NODE = 1, springs will be applied to every node in the model. If EVERY NTH NODE = 5, springs will be applied to every 5th node in the model and then perturbed by 1 every CONTROL DAMPED SOLVE iteration.

Currently, there is not an automated method of deciding which nodes need springs. More advanced options for spring placement are currently in-development.

When a CONTROL DAMPED SOLVE block is included in the input deck, an integer field named Control_NodeMask is created. A node with an integer value of Control_NodeMask greater

than 0 will have a spring added to ground. This field is what Sierra/SMuses to determine which nodes should have artificial springs. If desired, a user can set this field manually either by using a script like `exodus.py` or by setting the field with an initial condition block. To keep the field at the user defined value set `EVERY NTH NODE = 0` to avoid the internals from resetting the field.

4.8.4. Other Commands

For an explanation of commands related to iteration plot, see section 4.1.1.

4.8.5. Control Damped Solve Input Example

The following example shows the setup for a general multilevel solver involving control contact and control damped solve. Each load step first a multilevel control damped solve model problem is solved. Once the control damped solve force converges to the specified value, control contact is then solved at the next control level. Control damped solve again solves the model problem with the contact forces included, and the process continues until both levels converge to their specified criteria.

```
Begin solver

  Begin cg
    target relative residual = 1.0
  end cg

  Begin control damped solve
    level = 1
    target relative spring force = 1.0e-3
    stiffness shift factor = 1.0e-2
    maximum iterations = 100
  end control damped solve

  Begin control contact
    level = 2
    target relative residual = 1.0e-6
  end control contact
End
```

4.9. Predictors

Predictors are an important component of the nonlinear implicit solution strategy in Sierra/SM. A predictor is used to generate an initial trial solution for a load step or for a multilevel solver model problem. The goal of the predictor is to generate a trial solution that is as close as possible to the actual converged solution. A good prediction that gives a trial solution close to the actual solution can dramatically reduce the number of iterations required for convergence.

There are two types of predictors used in Adagio: the load step predictor and the level 1 predictor. The load step predictor, documented in Section 4.9.1, estimates a solution at the beginning of a new load step, and is applicable to all types of models. The level 1 predictor estimates the solution at the beginning of a new level 1 model problem in the multilevel solver. This predictor type, documented in Section 4.9.2, is only applicable for the multilevel solver.

4.9.1. Loadstep Predictor

```
BEGIN LOADSTEP PREDICTOR
  TYPE = <string>SCALE_FACTOR|SECANT|
    EXTERNAL|EXTERNAL_FIRST(SECANT)
  SCALE FACTOR = <real>scale_factor(1.0)
    [<real>first_scale_factor]
    [DURING <string list>period_names]
  SLIP SCALE FACTOR = <real>slip_factor(1.0)
    [DURING <string list>period_names]
END [LOADSTEP PREDICTOR]
```

The `LOADSTEP PREDICTOR` command block controls the behavior of the predictor that is used to predict the solution at the beginning of a new load step. This command block is placed in the `SOLVER` scope.

4.9.1.1. Predictor Type

```
TYPE = <string>SCALE_FACTOR|SECANT|
  EXTERNAL|EXTERNAL_FIRST(SECANT)
```

There are multiple types of load step predictors available in Adagio that are suitable for different types of analysis. The predictor type is selected using the `TYPE` command line in the `LOADSTEP PREDICTOR` command block.

- The scale factor predictor is selected with the `SCALE_FACTOR` option. This type of predictor extrapolates from the solution of the previous time step using the velocity field from that step, multiplied by a scale factor, to obtain a trial solution. The actual scale factor that is applied can be controlled with the `SCALE FACTOR` command line, which also appears in the `LOADSTEP PREDICTOR` command block. This type of predictor works well with models that have smoothly varying loads.

- The secant predictor is selected with the SECANT option, which is useful for scenarios in which the loading may not be monotonic. This type of predictor uses a line search to compute an optimal scaling factor for use in extrapolating from the previous solution. This type of predictor is the default, and provides good performance without requiring user intervention to select appropriate scale factors.
- The EXTERNAL predictor is experimental and should not generally be used. This option is discussed in more detail in the in development manual.

4.9.1.2. Scale Factor

```
SCALE FACTOR = <real>scale_factor(1.0)
  [<real>first_scale_factor]
  [DURING <string list>period_names]
```

The SCALE FACTOR command controls the behavior of the SCALE_FACTOR predictor type. The velocity field of the previous step, multiplied by the value of scale_factor, is used to extrapolate from the previous solution to obtain a trial solution. The default value of 1.0 provides good performance on models that experience smooth loading.

Setting the value of scale_factor to 0.0 effectively turns off the predictor. Note, if an analysis has trouble reaching convergence turning off the predictor with a zero scale factor may be helpful. This is particularly true of analysis with contact or other potentially time discontinuous boundary conditions. Extrapolating the solution between steps with different boundary conditions may overshoot the solution in a way from which is difficult to recover.

The optional first_scale_factor parameter is used as the scale factor for the first step of a time period. If first_scale_factor is omitted, the default behavior is to use scale_factor for all time steps. Models are often subjected to loading that is mostly monotonic, but with discontinuities at the beginning of a new period. Using 1.0 and 0.0 for the values of scale_factor and first_scale_factor, respectively, often gives good performance in such cases.

The SCALE FACTOR command line can be appended with the DURING specification, as discussed in Section 4.2.1. This allows for the scale factor to be specified per time period.

4.9.1.3. Slip Scale Factor

```
SLIP SCALE FACTOR = <real>slip_factor(1.0)
  [DURING <string list>period_names]
```

The SLIP SCALE FACTOR only has an effect in a analyses with contact that use the kinematic contact enforcement option, see Section 8.9.3 for details.

The prediction of slip on side B nodes that are currently active in sliding contact can be controlled separately from other nodes in the model by using the SLIP SCALE FACTOR command line. The parameter slip_factor controls the scaling of the predicted slip on these nodes. If

slip_factor is left at its default value of 1.0, contact slip will be predicted using the same scale factor used for the rest of the model. If the value of slip_factor is set to 0.0, the predicted solution will not allow any sliding contact nodes to slip. This command line applies to both the SCALE_FACTOR and the SECANT types of predictors.

The SLIP SCALE FACTOR command line can be appended with the DURING specification, as discussed in Section 4.2.1.

4.9.2. Level Predictor

```
LEVEL 1 PREDICTOR = <string>NONE|DEFAULT(DEFAULT)
```

The LEVEL 1 PREDICTOR command line controls the level 1 predictor. This command line appears in the SOLVER command block and is only applicable if the multilevel solver is used. The level 1 predictor estimates the solution for the next model problem based on the solution of the previous model problem. The default behavior is to enable this predictor selectively based on what solver controls are used (indicated by DEFAULT). The predictor can be turned off in all cases with the NONE.

4.9.3. Loadstep Predictor Input Example

The below example includes use of the predictor. The example predictor definition would be appropriate for an analysis with discontinuous loading during time period p1 followed by a smoother loading during time period p2. Turning off the predictor in time period p1 prevents overshooting solutions during sharp changes in loading. Turning back on the predictor in time period p2 allows acceleration of solution in the more uniformly loaded time period.

```
begin solver
  begin loadstep predictor
    type = secant
    scale factor = 0 0 during p1
    scale factor = 1 0 during p2
  end
  Begin cg
    target relative residual = 1.0e-09
    Maximum Iterations       = 100
  end
end
```

4.10. Implicit Time Step Control

Time stepping in Sierra/SMis controlled by using a `TIME CONTROL` command block in the procedure scope. This command block controls the start time, the termination time, and the method by which time is advanced during the analysis. The analysis time can optionally be subdivided into a number of time periods. If the analysis is from time 0 to time T and it is split into three periods, the first period is defined from time 0 to time t_1, the second period is defined from time t_1 to time t_2, and the third period is defined from time t_2 to time T. (The times t_1 and t_2 are set in the input deck.) The sum of the times for the three periods is T.

There are many reasons for splitting an analysis into multiple time periods. Boundary conditions may be turned on or off in different time periods. Different solver parameters may also be used in different time periods.

Time stepping can be uniform during a solution period, or it can vary during a solution period by using a function to control the time step as a function of analysis time. With implicit time stepping an automatic time stepping scheme can also be used to automatically take larger time steps at points in the analysis when the solution is relatively easy and take smaller time steps when it becomes more difficult to solve a time step.

A description of the `TIME CONTROL` command block follows in Section 4.10.1. Section 4.10.2 describes the `ADAPTIVE TIME STEPPING` command block, which allows adaptation of the size of the time step based on solution difficulty. A simple example of a `TIME CONTROL` command block is presented in Section 4.10.3.

4.10.1. Command Blocks for Time Control and Time Stepping

```
BEGIN TIME CONTROL
  BEGIN TIME STEPPING BLOCK <string>time_block_name
    START TIME = <real>start_time_value
    BEGIN PARAMETERS FOR ADAGIO REGION <string>region_name
      #
      # Time control parameters specific to ADAGIO
      # are set in this command block.
      #
    END [PARAMETERS FOR ADAGIO REGION <string>region_name]
  END [TIME STEPPING BLOCK <string>time_block_name]
  TERMINATION TIME = <real>termination_time
END [TIME CONTROL]
```

Sierra/SM time control resides in a `TIME CONTROL` command block. The command block begins with the input line

```
BEGIN TIME CONTROL
```

and terminates an input line of the following form:

```
END [TIME CONTROL]
```

Time control is defined at the procedure scope. If a procedure has multiple regions the time control for each region is defined in a single procedure time control block.

An arbitrary number of `TIME STEPPING BLOCK` command blocks can be present to define individual time stepping periods within the `TIME CONTROL` command block. Each `TIME STEPPING BLOCK` command block contains the time at which the time period starts and a number of time steps for time stepping within the period. Each time period terminates at the start time of the following time period. The start times for the `TIME STEPPING BLOCK` command blocks must appear in increasing time. The example in Section 4.10.3 shows the overall structure of the `TIME CONTROL` command block.

In the above input lines, the parameters are as follows:

- The string `time_block_name` is a name for the time period. Every time period must have a unique name. These names are referenced to control solution parameters or to activate/deactivate functionality.
- The real value `start_time_value` is the start time for this `TIME STEPPING BLOCK` time period. A time period goes from its start time until the start time of the next period or the termination time. The start time may be negative.
- The string `region_name` is the name of the region affected by the parameters (see Section 2.2).

The final termination time for the analysis is given by `TERMINATION TIME` command. The input `termination_time` is the time at which the analysis will stop. The `TERMINATION TIME` command line appears inside the `TIME CONTROL` command block but outside of any `TIME STEPPING BLOCK` command block.

The `TERMINATION TIME` command line can appear before the first `TIME STEPPING BLOCK` command block or after the last `TIME STEPPING BLOCK` command block. It is permissible to have `TIME STEPPING BLOCK` command blocks with start times greater than the termination time; but these time periods will not executed and will have no effect on the analysis. Only one `TERMINATION TIME` command line can appear in the `TIME CONTROL` command block. If more than one of these command lines appears, an error is generated.

Nested inside the `TIME STEPPING BLOCK` command block is a `PARAMETERS FOR ADAGIO REGION` command block containing parameters that control the time stepping.

```
BEGIN PARAMETERS FOR ADAGIO REGION <string>region_name
  TIME INCREMENT = <real>time_increment_value
  NUMBER OF TIME STEPS = <integer>nsteps
  TIME INCREMENT FUNCTION = <string>time_function
END [PARAMETERS FOR ADAGIO REGION <string>region_name]
```

The parameters in the `BEGIN PARAMETERS FOR ADAGIO REGION` block are specific to an implicit analysis. Note, similar parameters in a `BEGIN PARAMETERS FOR PRESTO REGION` block are specific to an explicit analysis 3.

The command block begins with an input line of the form

```
BEGIN PARAMETERS FOR ADAGIO REGION <string>region_name
```

and is terminated with an input line of the following form:

```
END [PARAMETERS FOR ADAGIO REGION <string>region_name]
```

As noted previously, the string `region_name` is the name of the region to which the parameters apply. The command lines nested inside the `PARAMETERS FOR ADAGIO REGION` command block are used to control the way time stepping occurs, and are described in Sections 4.10.1.1 through 4.10.1.3. Only one of these command lines (`TIME INCREMENT`, `NUMBER OF TIME STEPS`, or `TIME INCREMENT FUNCTION`) may be used in a given time period.

4.10.1.1. Time Increment

```
TIME INCREMENT = <real>time_increment_value
```

The `TIME INCREMENT` command line directly specifies the size of a uniform time step used during a time period. The size of the time step is specified with the parameter `time_increment_value`.

4.10.1.2. Number of Time Steps

```
NUMBER OF TIME STEPS = <integer>nsteps
```

The `NUMBER OF TIME STEPS` command line specifies the number of uniform time steps used during a time period. The number of time steps is specified with the parameter `nsteps`. If this command line is used, the time step size is computed by dividing the length of the time period by the specified number of time steps.

4.10.1.3. Time Increment Function

```
TIME INCREMENT FUNCTION = <string>time_function
```

The `TIME INCREMENT FUNCTION` command line allows the time step to be specified as a function of analysis time. The function used to control the time step is referenced by the parameter `time_function`. The actual function is defined within a `FUNCTION` command block within the SIERRA scope. It is often convenient to use a function of the `PIECEWISE CONSTANT` type (see Section 2.1.5) with this command line. Using a function to control time incrementation allows for the time increment to be conveniently switched many times during an analysis without creating a new time block every time the time step size needs to be changed.

4.10.2. Adaptive Time Stepping

```
BEGIN ADAPTIVE TIME STEPPING
  METHOD = <string>SOLVER|SOLVER_AVERAGE|MATERIAL|USER(SOLVER)
    [DURING <string list>period_names]
  TARGET ITERATIONS = <integer>target_iter(0)
    [DURING <string list>period_names]
  ITERATION WINDOW = <integer>iter_window(0)
    [DURING <string list>period_names]
  CUTBACK FACTOR = <real>cutback_factor(0.5)
    [DURING <string list>period_names]
  GROWTH FACTOR = <real>growth_factor(1.5)
    [DURING <string list>period_names]
  MAXIMUM FAILURE CUTBACKS = <integer>max_cutbacks(5)
    [DURING <string list>period_names]
  MAXIMUM MULTIPLIER = <real>max_multiplier(infinity)
    [DURING <string list>period_names]
  MINIMUM MULTIPLIER = <real>min_multiplier(1.0e-8)
    [DURING <string list>period_names]
  RESET AT NEW PERIOD = TRUE|FALSE(TRUE)
    [DURING <string list>period_names]
  ACTIVE PERIODS = <string list>period_names
  USER CRITERIA GLOBAL <string>var_name  <|<=|=|>=|> <real>cmp_val
END [ADAPTIVE TIME STEPPING]
```

An Adagio region has the capability to adapt the time step size during an analysis based on either solution difficulty or on feedback from material models. This capability is enabled by inserting an `ADAPTIVE TIME STEPPING` command block in the region scope. If this adaptive command block is not present in the Adagio region, the time stepping specified in the `TIME CONTROL` command block will be used exclusively. In addition to adjusting the size of time steps as the analysis proceeds the adaptive time stepping algorithm can attempt to solve a load step by using a smaller time step if a solution fails to converge. Only a single adaptive stepping time block can be used in a region.

The adaptive time stepping algorithm works by computing a multiplier that is applied to the time step specified in the `TIME CONTROL` command block. Thus, even if adaptive time stepping is used the base time step must be specified for all loading periods in the `TIME CONTROL` command block. For the initial load step in each solution period the Adagio region always uses the prescribed time step. In subsequent steps or after convergence failure the adaptive time stepping algorithm computes a factor that is multiplied by the prescribed time step to adjust the time step.

The command lines nested within the `ADAPTIVE TIME STEPPING` command are used to control the behavior of this feature and are described in Sections 4.10.2.1 through 4.10.2.10.

Many of these command lines can optionally be set for a specific time period. To set a parameter for a specific time period the `DURING` keyword followed by a list of applicable period names to a command line may be used (see Section 4.2.1). Multiple instances of these command lines can

exist to set the parameter differently for different time periods. If a command line is not appended with the DURING specification, it is used to set the default behavior for all time periods. Setting a default in this way does not preclude setting a period-specific value for that parameter with another instance of the same command line.

4.10.2.1. Method

```
METHOD = <string>SOLVER|SOLVER_AVERAGE|MATERIAL|USER(SOLVER)
  [DURING <string list>period_names]
```

The METHOD command line is used to select the adaptive time stepping algorithm. The default SOLVER method adapts the time step based on the difficulty of the solution over a time step. Solution difficulty is determined based on the number of iterations required to achieve convergence during a time step. If the SOLVER method is used, the TARGET ITERATIONS command line can be used to specify the target iterations used by this method to adapt the time step.

The SOLVER_AVERAGE method can be used when the problem has a multi level solver control such as control contact. With the SOLVER_AVERAGE option the difficulty is determined as the average number of iterations the solver takes per model problem instead of the accumulated number of iterations taken per time step.

The MATERIAL method uses the material models to inform an appropriate step size. A material may provide a maximum time step to prevent too much nonlinearity over a step. Note, only a handful of materials will define time steps and this option is primarily relevant to materials that have time dependent creep or other rate effects. The material suggested time step multiplier will still be bounded by the adaptive time stepping block input parameters for MAXIMUM MULTIPLIER, MINIMUM MULTIPLIER, GROWTH FACTOR, and CUTBACK FACTOR.

The USER method must be paired with a USER CRITERIA GLOBAL, see Section 4.10.2.11. When the criteria is satisfied a time step cutback is forced using the CUTBACK FACTOR, a convergence failure is mimicked and the current solution is thrown away. The time step is still bounded by the adaptive time stepping block input parameters but the GROWTH FACTOR is unused.



Warning: The USER method does not support the optional DURING .

4.10.2.2. Target Iterations

```
TARGET ITERATIONS = <integer>target_iter(0)
  [DURING <string list>period_names]
```

The solver method for adaptive time stepping adjusts the time step to achieve a target level of difficulty. The level of difficulty is determined by the total number of core solver iterations

required to solve for a time step. The TARGET ITERATIONS command specifies a target number of iterations per time step. This command line is optional.

The specified value of target_iter is used in conjunction with the value of iter_window (specified with the ITERATION WINDOW command documented in Section 4.10.2.3) to control the next step size. If the number of iterations for the previous step is greater than (target_iter-iter_window), and less than (target_iter+iter_window), the step size is kept the same. If the number of iterations is greater than (target_iter+iter_window), the step size is decreased, and if it is less than (target_iter-iter_window), the step size is increased.

If the target iterations is undefined or zero solver will not adjust the time step based on iteration count. The solver will cutback the time step if an element inverts, the solver fails to converge, or a material model suggests a smaller time step.

4.10.2.3. Iteration Window

```
ITERATION WINDOW = <integer>iter_window
  [DURING <string list>period_names]
```

The ITERATION WINDOW command specifies the size of the window for the iteration count used in the solver type of adaptive time stepping. See Section 4.10.2.2 for an explanation of how this is used in conjunction with the specified target iterations to adaptively choose the time step size. The default value of iter_window is the value specified in the TARGET ITERATIONS command line divided by 5.

4.10.2.4. Cutback Factor

```
CUTBACK FACTOR = <real>cutback_factor(0.5)
  [DURING <string list>period_names]
```

The CUTBACK FACTOR command line controls the amount by which the next time step is reduced if the solution is difficult. The parameter cutback_factor specifies the multiplier that is applied to the current time step with each cutback. The default value for the CUTBACK FACTOR is 0.5, the factor must be between zero and one. The cutback factor parameter also controls the amount that the current time step is cut back if a solution fails. The CUTBACK FACTOR command line can be appended with the DURING specification, as discussed in Section 4.2.1.

4.10.2.5. Growth Factor

```
GROWTH FACTOR = <real>growth_factor(1.5)
  [DURING <string list>period_names]
```

The GROWTH FACTOR command line controls the amount by which the next time step is increased if the solution is easy. It requires the user to set TARGET ITERATIONS to determine the level of

difficulty in the solution, as described in 4.10.2.2. The parameter `growth_factor`, which has a default value of 1.5, specifies the multiplier (a 50 percent increase) that is applied to the current time step each time the time step is increased. The `GROWTH FACTOR` command line can be appended with the `DURING` specification, as discussed in Section 4.2.1.

4.10.2.6. Maximum Failure Cutbacks

```
MAXIMUM FAILURE CUTBACKS = <integer>max_cutbacks(5)
  [DURING <string list>period_names]
```

The `MAXIMUM FAILURE CUTBACKS` command line controls how many cutbacks to a time step can be taken if a solution fails to converge. The parameter `max_cutbacks` has a default value of 5 if adaptive time stepping is enabled. If an `ADAPTIVE TIME STEPPING` command block is not present, the code will exit rather than attempting to cut back after an unsuccessful solution. The `MAXIMUM FAILURE CUTBACKS` command line can be appended with the `DURING` specification, as discussed in Section 4.2.1.

4.10.2.7. Maximum Multiplier

```
MAXIMUM MULTIPLIER = <real>max_multiplier(infinity)
  [DURING <string list>period_names]
```

Time steps are adaptively adjusted by updating a multiplier that is applied to the base time step specified in the `TIME CONTROL` command block. By default, there are no limits on the size of this multiplier. The `MAXIMUM MULTIPLIER` command line can be used to limit the growth of the time step. The parameter `max_multiplier` specifies the upper bound for the multiplier. The largest possible time step in an analysis is the product of `max_multiplier` and the baseline time step size. This baseline size is specified in the `TIME CONTROL` command block. The `MAXIMUM MULTIPLIER` command line can be appended with the `DURING` specification, as discussed in Section 4.2.1.

4.10.2.8. Minimum Multiplier

```
MINIMUM MULTIPLIER = <real>min_multiplier(1.0e-8)
  [DURING <string list>period_names]
```

Time steps are adaptively adjusted by updating a multiplier that is applied to the base time step specified in the `TIME CONTROL` command block. By default, the `MINIMUM MULTIPLIER` is set to $1.0e-8$. The `MINIMUM MULTIPLIER` command line can be used to limit the shrinkage of the time step. The parameter `min_multiplier` specifies the lower bound for the multiplier. The smallest possible time step in an analysis is the product of `min_multiplier` and the baseline time step size. The `MINIMUM MULTIPLIER` command line can be appended with the `DURING` specification, as discussed in Section 4.2.1.

4.10.2.9. Reset at New Period

```
RESET AT NEW PERIOD = TRUE|FALSE(TRUE)
  [DURING <string list>period_names]
```

This command controls whether the time step is reset to the user-specified base time step at the beginning of a new time period. If this parameter is set to `TRUE`, which is the default value, the time step is reset. If this parameter is set to `FALSE`, the adaptive time stepping is maintained from the previous period. If the base time step is changed in the new period, the multiplier is scaled to maintain the time step originally calculated for the next step. The multiplier is scaled by the ratio: time increment of the previous period/time increment of the new period. If the resulting multiplier is outside of the specified range, it will be adjusted to stay within that range and the time step will change. In summary, the first time step of the new period will be equal to the base time step of the previous period times the scaled multiplier. The value of this parameter can optionally be specified per time period with the `DURING` option (see Section 4.2.1). If adaptive time stepping is inactive for a given period, the time step will always be reset even if it is requested that the adaptive time step not be reset.

4.10.2.10. Active Periods

```
ACTIVE PERIODS = <string list>period_names
```

If the `ADAPTIVE TIME STEPPING` command block is present in the Adagio region, it is active for all periods by default. Adaptive time stepping can be activated for specific time periods with the `ACTIVE PERIODS` command line. See Section 2.6 for more information about this optional command line.

4.10.2.11. User Criteria Global

```
USER CRITERIA GLOBAL <string>var_name  <|<=|=|>=|> <real>cmp_val
```

The `USER CRITERIA GLOBAL` command is for paring with method `USER`. At the end of a solve, the global variable `var_name` is compared to the `cmp_val` with the specified comparison type. If the comparison is true a cutback to the timestep is forced. The `CUTBACK FACTOR` is used to reduce the time step while respecting bounds on the timestep defined by `MINIMUM MULTIPLIER` and `MAXIMUM FAILURE CUTBACKS`.

4.10.3. Time Control Example

The following is a simple example of a `TIME CONTROL` command block:

```
BEGIN TIME CONTROL
  BEGIN TIME STEPPING BLOCK p1
```

```
    START TIME = 0.0
    BEGIN PARAMETERS FOR ADAGIO REGION adagio_region
      NUMBER OF TIME STEPS = 10
    END
  END
  BEGIN TIME STEPPING BLOCK p2
    START TIME = 1.0
    BEGIN PARAMETERS FOR ADAGIO REGION adagio_region
      TIME INCREMENT = 0.2
    END
  END
  TERMINATION TIME = 2.0
END
```

The first `TIME STEPPING BLOCK`, `p1`, begins at time 0.0, the initial start time, and terminates at time 1.0. The first time period will take ten time steps each of time 0.1 to reach time 1.0.

The second `TIME STEPPING BLOCK`, `p2`, begins at time 1.0 and terminates at time 2.0, the time listed on the `TERMINATION TIME` command line. The second time period will take five time steps each of time 0.2 to reach the final end time of 2.0.

The `TIME STEPPING BLOCK` names `p1` and `p2` can be referenced elsewhere to activate boundary conditions or control the solver in different ways during the two analysis time periods.

4.11. Implicit Dynamic Time Integration

Sierra/SM has the ability to perform implicit solution of both quasistatic and dynamic problems. In quasistatic calculations, the solution for static equilibrium is obtained at each step, ignoring the inertial mass and acceleration terms of the equations of motion. In dynamic problems, the inertial terms are included.

4.11.1. Implicit Dynamics

```
BEGIN IMPLICIT DYNAMICS
  ACTIVE PERIODS = <string list>period_names
  USE HHT|GENERALIZED ALPHA INTEGRATION
  ALPHA = <real>alpha(-0.1) [DURING <string list>period_names]
  BETA = <real>beta(0.3025) [DURING <string list>period_names]
  GAMMA = <real>beta(0.6) [DURING <string list>period_names]
  CONTACT TIMESTEP = ON|OFF(ON)
    [DURING <string list>period_names]
END [IMPLICIT DYNAMICS]
```

The `IMPLICIT DYNAMICS` command block enables implicit dynamics and contains commands to control the behavior of implicit time integration. This command block is placed in the `ADAGIO REGION` command block. Formerly, it was necessary to create a new procedure for switching between implicit dynamics and quasistatics in an analysis. Now, these different analysis types can all be done within a single region, and the command lines in the `IMPLICIT DYNAMICS` command block can be used to enable or disable implicit dynamics for specific time periods within a single region. Sections 4.11.1.1 through 4.11.1.4 describe the command lines in this command block.

4.11.1.1. Active Periods

```
ACTIVE PERIODS = <string list>period_names
```

If the `IMPLICIT DYNAMICS` command block is present in the Adagio region, it is active for all periods by default. Implicit dynamics can be activated for specific time periods with the `ACTIVE PERIODS` command line. See Section 2.6 for more information about this optional command line.

4.11.1.2. Use HHT|Generalized Alpha Integration

```
USE HHT|GENERALIZED ALPHA INTEGRATION
```

Adagio supports the HHT and generalized alpha algorithms for implicit time integration. Implementation follows the theory outlined in the Sierra/SD- Theory Manual.

4.11.1.3. HHT Parameters

```
ALPHA = <real>alpha(-0.1) [DURING <string list>period_names]
BETA = <real>beta(0.3025) [DURING <string list>period_names]
GAMMA = <real>gamma(0.6) [DURING <string list>period_names]
```

HHT integration parameters may be modified via the ALPHA, BETA, and GAMMA command lines. These parameters can all be specified for the entire analysis, or they can vary by solution period by appending them with the optional DURING specification (see Section 4.2.1).

- The ALPHA command line specifies α, which is the dissipation factor applied to the internal force vector. The value of alpha controls numerical damping and must always be less than or equal to zero. Its default value of -0.1 results in slight numerical damping of modes with frequencies higher than can be resolved with a given time step. To maintain second-order accuracy, the value of alpha should not be less than $-\frac{1}{3}$.
- The BETA command line specifies the stability parameter β for time integrators in the Newmark family. The default value of beta is 0.3025. It should have a value of $0.25(1-\alpha)^2$ to maintain second-order accuracy.
- The GAMMA command line specifies the dissipation factor γ for Newmark time integrators. The default value of gamma is 0.6. It should have a value of $0.5-\alpha$ to maintain second-order accuracy.

The HHT time integrator [5] is used to integrate the semi-discrete equations of motion in time. The behavior of the HHT integrator is controlled with three parameters: α, β, and γ. With proper selection of these parameters, the HHT integrator is unconditionally stable. If $\alpha = 0$, the HHT integrator reduces to the Newmark method [6], for which unconditional stability requires $2\beta \geq \gamma \geq 1/2$. The trapezoidal rule is recovered if $\alpha = 0$, $\beta = 0.25$, and $\gamma = 0.5$. For a detailed discussion of the theory of implicit time integrators, see [7].

4.11.1.4. Implicit Dynamic Timestep Cutback for Contact

```
CONTACT TIMESTEP = ON|OFF(ON)
  [DURING <string list>period_names]
```

When running an implicit dynamics simulation, errors can sometimes occur when too large of a time step is taken for solving a problem involving contact. For example blocks move completely through each other during a single time step contact may be missed or solved inaccurately. The CONTACT TIMESTEP command line specifies whether the time step can be cut back for better integration of contact impact. The default is ON which allows for the time step to be cut. This parameter can also vary by period by appending the command lines with the optional DURING specification (see Section 4.2.1).

4.12. Deprecated Commands

The command `NUMBER OF SMOOTHING ITERATIONS` in the full tangent preconditioner block is deprecated. The command `MINIMUM SMOOTHING ITERATIONS` is a direct replacement and should be used instead.

References

[1] M.L. Blanford, M.W. Heinstein, and S.W. Key. JAS3D – a multi-strategy iterative code for solid mechanics analysis users' instructions, release 2.0, September 2001.

[2] R. Fletcher and C.M. Reeves. Function minimization by conjugate gradients. *The Computer Journal*, 7:149–153, 1964.

[3] C. Farhat, M. Lesoinne, and K. Pierson. A scalable dual-primal domain decomposition method. *Numerical Linear Algebra with Applications*, 7:687–714, 2000.

[4] C. Farhat, M. Lesoinne, P. LeTallec, K. Pierson, and D. Rixen. FETI-DP: A dual-primal unified FETI method – part i: A faster alternative to the two-level FETI method. *International Journal for Numerical Methods in Engineering*, 50:1523–1544, 2001.

[5] H.M. Hilber, T.J.R. Hughes, and R.L. Talor. Improved numerical dissipation for time integration algorithms in structural dynamics. *Earthquake Engineering and Structural Dynamics*, 5:283–292, 1977.

[6] N.M. Newmark. A method of computation for structural dynamics. *Journal of Engineering Mechanics, ASCE*, 85(EM3):67–94, 1959.

[7] T.J.R. Hughes. *The Finite Element Method–Linear Static and Dynamic Finite Element Analysis*. Prentice-Hall, Inc, Englewood Cliffs, NJ, 1987.

5. MATERIAL MODELS

This chapter describes material models that can be used in conjunction with the elements in Sierra/SM. Most of the material models can be used in either implicit or explicit analyses. Even though a material model can be used by both implicit and explicit, usage of the model may be better suited for the type of problems solved implicit verses explicit. For example, a material model that was built to characterize behavior over a long time would be better suited for use in Adagio/Implicit than in Presto/Explicit.

The material models described in this chapter are, in general, applicable to solid elements. Structural elements, such as shells and beams, have a more limited set of material models. Chapter 6 describes the element library, including which material models are available for the various elements. The introduction to Chapter 6 summarizes all the element types available in Sierra/SM. For each element type, a list of available material models is provided.

State variable output is available for many materials. See Section 9.10.2 to learn how to output the state variables for materials or to the specific material model sections for the list of computed state variables for that material model.

Most material models used by Sierra/SM are defined by the LAMÉ material library [1].

General Model Form. `MATERIAL` command blocks appear in the SIERRA scope in the general form shown below.

```
BEGIN MATERIAL <string>mat_name
  #
  # Command lines and command blocks for material
  # models appear in this scope.
  #
END [MATERIAL <string>mat_name]
```

`MATERIAL` command blocks are physics independent in the sense that the information in them can be shared by more than one application. The command block begins with the line:

```
BEGIN MATERIAL <string>mat_name
```

and terminates with the line:

```
END [MATERIAL <string>mat_name]
```

Here, the string mat_name is a user-specified name for the command block. This name is typically descriptive of the material being modeled, e.g., aluminum_t6061.

Within a MATERIAL command block, there will usually be at least one BEGIN PARAMETERS FOR MODEL command block as well as other general material command lines used to describe particular material models. The general material command lines, if present, are listed first, followed by one or more material-model command blocks. The general material command lines may be used to specify the density of the material, the Biot's coefficient, and the application of temperatures and thermal strains. Each material-model command block follows the naming convention of PARAMETERS FOR MODEL model_name, where model_name identifies a particular material model, such as elastic, elastic-plastic, or orthotropic crush. Each such command block contains all the parameters needed to describe a particular material model.

As noted above, more than one material-model command block can appear within a MATERIAL command block. For example, a MATERIAL command block called steel could have two nested material-model command blocks. One of the material-model command blocks would provide an elastic model for steel and the other an elastic-plastic model for steel. The general form of a MATERIAL command block would be as follows:

```
BEGIN MATERIAL <string>mat_name
  #
  # General material command lines
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_coefficient_value
  #
  THERMAL {LOG|ENGINEERING} STRAIN FUNCTION =
    <string>thermal_strain_function
  # or all three of the following
  THERMAL {LOG|ENGINEERING} STRAIN X FUNCTION =
    <string>thermal_strain_x_function
  THERMAL {LOG|ENGINEERING} STRAIN Y FUNCTION =
    <string>thermal_strain_y_function
  THERMAL {LOG|ENGINEERING} STRAIN Z FUNCTION =
    <string>thermal_strain_z_function
  THERMAL {LOG|ENGINEERING} STRAIN DIRECTION
    <string>dir_name FUNCTION = <string>funcName
  #
  ARTIFICIAL ENGINEERING STRAIN FUNCTION =
    <string>artificial_strain_function
  # or all three of the following
  ARTIFICIAL ENGINEERING STRAIN X FUNCTION =
    <string>artificial_strain_x_function
  ARTIFICIAL ENGINEERING STRAIN Y FUNCTION =
    <string>artificial_strain_y_function
  ARTIFICIAL ENGINEERING STRAIN Z FUNCTION =
    <string>artificial_strain_z_function
  ARTIFICIAL ENGINEERING STRAIN DIRECTION
    <string>dir_name FUNCTION = <string>funcName
```

```
  BEGIN PARAMETERS FOR MODEL <string>model_name1
    #
    # Parameters for material model model_name1
  END PARAMETERS FOR MODEL <string>model_name1
  #
  BEGIN PARAMETERS FOR MODEL <string> model_name2
    #
    # Parameters for material model model_name2
  END PARAMETERS FOR MODEL <string> model_name2
  #
  # Additional model command blocks if required
  #
END [MATERIAL <string>mat_name]
```

In the above general form for a `MATERIAL` command block, the string `model_name1` could be `ELASTIC` and the string `model_name2` could be `ORTHOTROPIC CRUSH`. Typically, however, only one material model would be desired for a given block, and the `MATERIAL` command block would have only one `PARAMETERS FOR MODEL` command block. A particular material model may only appear once within a given `MATERIAL` command block.

Although multiple material models can be defined for one material within a `MATERIAL` command block, only one material model is actually used for a given element block during an analysis. The ability to define multiple constitutive models for one material is provided as a convenience to enable easily switching between models. The material name and the model name are both referenced when material models are assigned to element blocks within the `FINITE ELEMENT MODEL` command block, which is described in Section 6.1.

This chapter is organized according to the general form for the `MATERIAL` command block. Section 5.1 discusses the `DENSITY` command line, the `BIOTS COEFFICIENT` command line, and the command lines used for thermal strains, and also explains how temperatures and thermal strains are applied. Section 5.2 describes each of the material models that are used by Sierra/SM. References applicable to Sierra/SM are listed at the end of the chapter.

As indicated in the introductory material, not all the material models are applicable to all element types. As one example, there are the material models available for use with cohesive zone elements or with Dash contact, as detailed in Section 5.3. As another example, there is a one-dimensional elastic material model that is used for a truss element but is not applicable to solid elements such as hexahedra or tetrahedra. For this particular example, the specific material-model usage is hidden from the user. If the user specifies a linear elastic material model for a truss, the one-dimensional elastic material model is used. If the user specifies a linear elastic material model for a hexahedron, a full three-dimensional elastic material model is used. As another example, the energy-dependent material models in Presto can only be used for solid elements such as hexahedra and tetrahedra. (Chapter 6 indicates what material models are available for which element formulations.)

Nonlinear material models have state variables. See Section 9.10.2 for more information on how

to output the state variables for the nonlinear material models.

Brief descriptions of the material models are presented in this chapter. More detailed descriptions can be found in the references for most material models.

5.1. General Material Commands

A MATERIAL command block may include command lines applicable to all the material models specified within the command block. These command lines related to density, to Biot's coefficient, and the thermal strain behavior are discussed, respectively, in Section 5.1.1, Section 5.1.2, and Section 5.1.4. Additionally several material models share similar behavior for certain commands such as elastic constants and the beta parameter discussed in Section 5.1.5 and Section 5.1.6.

5.1.1. Density Command

```
DENSITY = <real>density_value
```

This command line specifies the density of the material. The units of the input parameter density_value are mass per unit volume.

A MATERIAL command block can contain one or more PARAMETERS FOR MODEL command blocks. The specified density_value will be used with all models described in these PARAMETERS FOR MODEL command blocks.

5.1.2. Biot's Coefficient Command

```
BIOTS COEFFICIENT = <real>biots_value
```

This command line specifies the Biot's coefficient of the material. Biot's coefficient is used with the pore pressure capability (commonly called an effective stress model). See Section 7.12 for more information on pore pressure. If not given, the value defaults to 1.0 which is a common value for many rock properties. This parameter is unitless. The pore pressure capability is implemented as the following effective stress model

$$\sigma_{effective} = \sigma - \alpha P \tag{5.1}$$

where P is the pressure and α is Biot's coefficient.

A MATERIAL command block can contain one or more PARAMETERS FOR MODEL command blocks. The specified biots_value will be used with all models described in these PARAMETERS FOR MODEL command blocks.

5.1.3. General Artificial Strains

Sierra/SM allows a user to apply an artificial strain to a material. Artificial strains may be applied via the Artificial Strain commands or via the Thermal Strain commands (5.1.4). General artificial strains may be useful for fitting parts together for a preload analysis or modeling a variety of other physics that could cause material expansion or contraction (such as radiation induced expansion, material desiccation, etc.)

Artificial strains may also be defined via a boundary condition. See Section 7.15.7.

The artificial strain commands are placed in the `MATERIAL` command block.

```
ARTIFICIAL ENGINEERING STRAIN FUNCTION =
  <string>artificial_strain_function
#
# Or all three of the following
#
ARTIFICIAL ENGINEERING STRAIN X FUNCTION =
  <string>artificial_strain_x_function
ARTIFICIAL ENGINEERING STRAIN Y FUNCTION =
  <string>artificial_strain_y_function
ARTIFICIAL ENGINEERING STRAIN Z FUNCTION =
  <string>artificial_strain_z_function
#
# Or one to three of the following
#
ARTIFICIAL ENGINEERING STRAIN DIRECTION
 <string>dir_name FUNCTION = <string>funcName
```

The `ARTIFICIAL ENGINEERING STRAIN FUNCTION` defines an isotropic strain.

The X, Y, and Z functions may define anisotropic strain. Note, the X, Y, and Z directions correspond to the material coordinate system. For most solid element materials, the material coordinate system is aligned with the global X, Y, and Z axes. For orthotropic solid materials, the X, Y, and Z thermal strains are applied in the material orthotropic R, S, and T coordinate system (which may vary element to element). For two dimensional shell and membrane elements, the X and Y directions are in the plane of the shell, and Z is the through thickness direction. Z strains generally have no effect on shells. See Section 6.2.5 for a discussion of the shell *rst* coordinate system. For one dimensional beam or truss elements, the X direction is along the length of the beam, and the Y and Z lie in the orthogonal directions. Y and Z strains generally have no effect on beams or trusses.

The `STRAIN DIRECTION` function defines strain in an arbitrary direction (note, still in the material coordinate system). Up to three orthogonal direction functions may be used.

All the artificial strain functions define the material engineering strain with respect to time. If a material is using artificial strain, it must define either one isotropic function or all three anisotropic functions.

It is highly important to note that there is zero artificial strain applied at the reference (original) model configuration. Thus artificial strains are calculated as the difference in strain function from the current time back to the reference time (analysis start time). For clarity, it is recommended that the artificial strain function be zero at the start time of the analysis. Artificial strain is a component of Non-Material strain; consequently, it is a strain that does not produce stress from the material response. The output variable names for artificial strain can be found in Table 9-13, Table 9-14, Table 9-15, Table 9-16, or Table 9-18 depending on the element type it is applied to.

5.1.4. Thermal Strain Behavior

Isotropic and orthotropic thermal strains may be defined for use by material models listed in a `MATERIAL` command block. Section 5.1.4.1 describes the command lines that are used to define the thermal strain behavior. These command lines must be used in conjunction with other command blocks outside of a `MATERIAL` command block for the calculations of thermal strains to be activated. Section 5.1.4.2 explains the process for activating thermal strains.

5.1.4.1. Defining Thermal Strains

```
THERMAL {LOG|ENGINEERING} STRAIN FUNCTION =
  <string>thermal_strain_function
#
# or all three of the following
#
THERMAL {LOG|ENGINEERING} STRAIN X FUNCTION =
  <string>thermal_strain_x_function
THERMAL {LOG|ENGINEERING} STRAIN Y FUNCTION =
  <string>thermal_strain_y_function
THERMAL {LOG|ENGINEERING} STRAIN Z FUNCTION =
  <string>thermal_strain_z_function
#
# Or one to three of the following
#
THERMAL {LOG|ENGINEERING} STRAIN DIRECTION
 <string>dir_name FUNCTION = <string>funcName
```

A `MATERIAL` command block may include command lines that define thermal strain behavior. It is possible to specify either an isotropic thermal-strain field using the command line `STRAIN FUNCTION` or an orthotropic thermal-strain field using the command lines `STRAIN X FUNCTION`, `STRAIN Y FUNCTION`, and `STRAIN Z FUNCTION`. For any of these command lines, the user supplies a thermal strain function (via a `FUNCTION` command block), which defines the thermal strain as a function of temperature. Thermal strain is a component of Non-Material strain; consequently, it is a strain that does not produce stress from the material response. The computed thermal strain is then subtracted from the strain passed to the material model.

Thermal strains may be specified in either engineering or log strain values. Note, the older version of these commands `THERMAL STRAIN FUNCTION =` specified log strain. Thermal strain data is often provided in engineering strain units, though the native strain measure in side of Adagio/Presto is log strain. At small strain values, log strain and engineering strain are nearly identical.

The `THERMAL STRAIN FUNCTION` defines an isotropic strain.

The X, Y, and Z functions define anisotropic strain. Note, the X, Y, and Z directions correspond to the material coordinate system. For most solid element materials, the material coordinate system is aligned with the global X, Y, and Z axes. For orthotropic solid materials, the X, Y, and Z thermal strains are applied in the material orthotropic R, S, and T coordinate system (which may vary element to element). For two dimensional shell and membrane elements, the X and Y directions are in the plane of the shell, and Z is the through thickness direction. Z strains generally have no effect on shells. See Section 6.2.5 for a discussion of the shell *rst* coordinate system. For one dimensional beam or truss elements, the X direction is along the length of the beam, and the Y and Z lie in the orthogonal directions. Y and Z strains generally have no effect on beams or trusses.

The `STRAIN DIRECTION` function defines strain in an arbitrary direction (note, still in the material coordinate system). Up to three orthogonal direction functions may be used.

If an isotropic thermal strain field is to be applied, the `THERMAL {LOG|ENGINEERING} STRAIN FUNCTION` command line is placed in the `MATERIAL` command block outside of the specifications of any material models. Such placement is necessary because the isotropic thermal strain is a general material property, not a property that is specific to any particular constitutive model such as `ELASTIC` or `ELASTIC-PLASTIC`. The input value of `thermal_strain_function` is the name of the function that defines thermal strain as a function of temperature for the material model described in this particular `MATERIAL` command block. The function is defined within the SIERRA scope using a `FUNCTION` command block. For more information on how to set the input to compute thermal strains and how to apply temperatures, see Section 5.1.4.2.

The specification of an orthotropic thermal strain field requires that all three of the `THERMAL STRAIN X FUNCTION`, `THERMAL STRAIN Y FUNCTION`, and `THERMAL STRAIN Z FUNCTION` command lines be placed in the `MATERIAL` command block. All three command lines must be provided, even when there is no thermal strain in one or more directions. The values of `thermal_strain_x_function`, `thermal_strain_y_function`, and `thermal_strain_z_function` are the names of the functions for thermal strains in the *X*-direction, the *Y*-direction, and the *Z*-direction, respectively. These functions are defined within the SIERRA scope using `FUNCTION` command blocks. To specify that there should be no thermal strain in a given direction, use a function that always evaluates to zero for that direction.

If one the `THERMAL {LOG|ENGINEERING} STRAIN FUNCTION` command lines are used for one of the several of the material models, as discussed in Section 5.1.4.2. The model external artificial strain and the model internal artificial strain will be multiplied together. The use of a thermal strain is identified in the descriptions of the material models in Section 5.2 by the notation "`thermal strain option`."

If both artificial and thermal strain are defined simultaneously the two strain components are additive.

The output variable names for thermal strain can be found in Table 9-12.

5.1.4.2. Activating Thermal Strains

Sierra/SM has the capability to compute thermal strains on three-dimensional continuum, two-dimensional (shell, membrane), and one-dimensional (beam, truss) elements. Three inputs are required to activate thermal strains:

- One or more thermal strain functions (strain as a function of temperature) must be defined. Each thermal strain function is defined with a `FUNCTION` command block. (This function is the standard function definition that appears in the SIERRA scope.) The thermal strain function gives the total thermal strain associated with a given temperature. It is the change in thermal strain with the change in temperature that gives rise to thermal stresses in a body.
- The material models used by blocks that experience thermal strain must have their thermal strain behavior defined. The command lines for defining isotropic and orthotropic thermal strain are described in Section 5.1.4.1. Materials with isotropic thermal strain use the `THERMAL {LOG|ENGINEERING} STRAIN FUNCTION` command line, while those with orthotropic thermal strain must define thermal strain in all three directions using the `THERMAL {LOG|ENGINEERING} STRAIN X FUNCTION`, `THERMAL {LOG|ENGINEERING} STRAIN Y FUNCTION`, and `THERMAL {LOG|ENGINEERING} STRAIN Z FUNCTION` command lines. These inputs can be used with all material models with the exception of the following: elastic three-dimensional orthotropic, Mooney-Rivlin, NLVE three-dimensional orthotropic, Swanson, and viscoelastic Swanson. These models require their own model-specific inputs to define thermal strain and must not use these standard commands. Information for defining thermal strains is provided in the individual descriptions of these models in Section 5.2.
- A temperature field must be applied to the affected blocks. The boundary condition command block to specify the application of temperatures is `PRESCRIBED TEMPERATURE`. A description of the `PRESCRIBED TEMPERATURE` command block is given in Section 7.11.

An applied temperature is prescribed at the nodes, but thermal strain is computed based on element temperature. Element temperatures are obtained by averaging the temperatures of the nodes connected to a given element.

It is important to note that there is zero thermal strain applied at the reference temperature. The thermal strain reference temperature is stored in the element field named 'thermal_strain_reference_temperature' and may vary from element to element. Unless otherwise specified, the thermal strain reference temperature for each element is given by the initial temperature for that element, i.e., the model without other loads is in a stress-free state at the initial temperature. If desired, a different stress-free reference temperature can be used by prescribing the reference temperature with the `INITIAL CONDITION` command block as described in Section 7.3.

A second critical note is that thermal strain is applied based the change in temperature over a step. First the temperature is computed at the beginning of the step and the end of the step. Next the thermal strain is computed at these beginning and end of the step temperatures. The thermal strain increment is computed by the difference of these beginning and end of step strains. Finally this strain increment is applied to the material. Due to this thermal strain enforcement process the thermal strain is affected only by the slope of the thermal strain function. A vertical offset of the thermal strain function has no effect. Additionally if two discontinuous temperature functions are applied in different time periods thermal strain will not see the temperature jump between these two functions.

5.1.4.3. Thermal Strain Example

Example Consider the following thermal strain function:

```
BEGIN FUNCTION thermStrain
  TYPE IS PIECEWISE LINEAR
  BEGIN VALUES
    500  -0.04
    1000  0.01
    1500  0.03
  END
END
```

Let an element have an initial temperature of 1000 and a thermal strain reference temperature of 1000. Initially the element has zero applied thermal strain as it is at the strain-free reference temperature. If the element heats to temperature 1500, the thermal strain applied to the element will be 0.02 (the thermal strain at 1500 minus the thermal strain at the reference temperature 1000). If the element cools to 500, the thermal strain applied to the element will be -0.05 (the thermal strain at 500 minus the thermal strain at the reference temperature 1000).

5.1.5. Elastic Constants

```
YOUNGS MODULUS = <real> E
POISSONS RATIO = <real> ν
SHEAR MODULUS  = <real> G
BULK MODULUS   = <real> K
LAMBDA         = <real> λ
TWO MU         = <real> 2μ
```

Many material models contain input for basic elastic constants. These elastic constants may define the material response before yield, the elastic unloading response, or the approximate elastic stiffness of the material model. There exist several inter-related ways to define the elastic material properties. Six separate elastic constants may potentially be set `YOUNGS MODULUS` (E), `POISSONS RATIO` (ν), `BULK MODULUS` (K), `SHEAR MODULUS` (G), `LAMBDA` (λ), and `TWO MU` (2μ).

Generally elastic constants may either be set within the `BEGIN MATERIAL` command block or the `BEGIN PARAMETERS FOR MODEL ....` command block. Constants defined in the `BEGIN MATERIAL` block apply to all models in the material block. Constants defined in the `BEGIN PARAMETERS FOR MODEL ....` block apply to just that specific material model.

The parsed elastic constants are echoed to the log file when reading the material to verify correct setup. For example:

```
  MATERIAL ELASTIC CONSTANT:steel_ELASTIC

  E           Nu          G           K           Lambda
  7.43e+11    0.34        2.77e+11    7.74e+11    5.89e+11
```

Note, the elastic constants printed at this point in the log are based off of the raw values of the global elastic constants such as `YOUNGS MODULUS`. The printed constants are not scaled for temperature dependence or any other external scaling factors. The proper element-by-element scaling will ultimately be used by the element mechanics and can be viewed by plotting the element variables `dilmod` (Dilatational Modulus) and `shrmod` (Shear Modulus.)

Examples:

```
#
#  Define elastic constants for a specific material model
#
BEGIN MATERIAL steel
  BEGIN PARAMETERS FOR MODEL ELASTIC
    YOUNGS MODULUS = 200e+9
    POISSONS RATIO = 0.3
  END
END

#
#  Define elastic constants for all associated material models
#
BEGIN MATERIAL aluminum
  YOUNGS MODULUS = 40e+9
  POISSONS RATIO = 0.35

  BEGIN PARAMETERS FOR MODEL ELASTIC
    ....
  END
  BEGIN PARAMETERS FOR MODEL ELASTIC_PLASTIC
    ....
  END
END
```

The elastic properties are defined as follows:

- `YOUNGS MODULUS` describes the tensile elasticity, how much an object strains when it is pulled with opposing forces. Young's modulus must be a non-negative number.
- `POISSONS RATIO` describes how much a material expands laterally in unconstrained compression. An incompressible material has a Poisson's ratio of `0.5`. The valid range for Poisson's ratio if from -1 to 0.5.
- `BULK MODULUS` describes the volumetric elasticity, how much an object shrinks under isotropic compression. Bulk modulus must be a non-negative number.
- `SHEAR MODULUS` describes the resistance of a solid to shear forces. A freely flowing fluid would have zero shear modulus. Shear modulus must be a non-negative number.
- `LAMBDA` is the first Lame elastic constant. Lambda can have any value though is generally positive for real materials.
- `TWO MU` is twice the shear modulus.

The elastic constants are interrelated and defining any two of these constants (with the exception of `TWO MU` and `SHEAR MODULUS`) fully defines the elastic properties of the material. Generally exactly two of these constants should be set for each material model. Note, `TWO MU` and `SHEAR MODULUS` define the same constant (`TWO MU` is just twice the `SHEAR MODULUS`). Thus `TWO MU` and `SHEAR MODULUS` should not be defined at the same time.

Given Young's Modulus E and Poisson's ratio (ν) the other four elastic constants would be set as follows:

$$\texttt{BULK MODULUS:}\quad K = \frac{E}{3(1-2\nu)} \tag{5.2}$$

$$\texttt{SHEAR MODULUS:}\quad G = \frac{E}{2(1+\nu)} \tag{5.3}$$

$$\texttt{LAMBDA:}\quad \lambda = \frac{E\nu}{(1+\nu)(1-2\nu)} \tag{5.4}$$

$$\texttt{TWO MU:}\quad 2\mu = \frac{E}{1+\nu} \tag{5.5}$$

5.1.6. The Isotropic/Kinematic hardening BETA parameter

```
BETA = <real>betaVal(1.0)
```

Several plasticity model variants make use of the Isotropic/Kinematic hardening factor `BETA`. In a three-dimensional framework, hardening is the law that governs how the yield surface evolves in stress space. If the yield surface grows uniformly in stress space, the hardening is called isotropic hardening. When `BETA` is 1.0 the model uses only isotropic hardening. If the yield surface does not grow in stress space but rather moves, the hardening is called kinematic hardening. When `BETA` is 0.0 the model uses only kinematic hardening. Values of `BETA` between 0.0 and 1.0 define

a mixture of isotropic and kinematic hardening. Note, the input BETA has other meanings in some models, so see the specific material model documentation for details.

The mechanics of isotropic hardening are demonstrated in Figure 5-1.

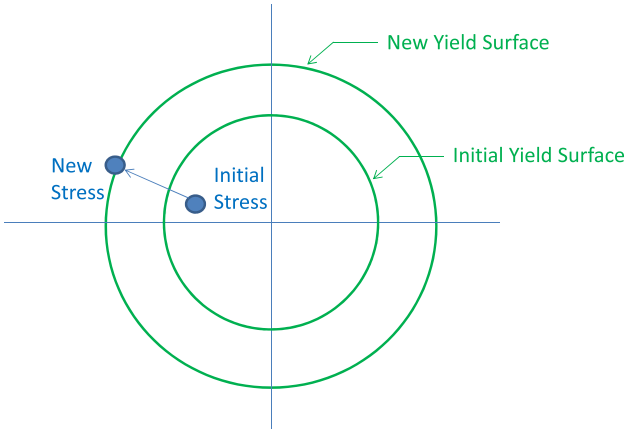


Figure 5-1. Isotropic Hardening, Beta = 1.0

The mechanics of kinematic hardening are demonstrated in Figure 5-2. If using pure kinematic hardening, then irrespective of the post yield hardening behavior the yield surface will never grow larger than the initial yield stress. Thus when using pure kinematic hardening the particular shape of the post yield hardening curve generally has no effect.

The mechanics of a mixed 0.5 BETA parameter is demonstrated in Figure 5-3.

5.1.7. Coordinate Systems

```
COORDINATE SYSTEM              = <string> coordinate_system_name
DIRECTION FOR ROTATION         = <real> 1|2|3
ALPHA                          = <real> α1 (degrees)
SECOND DIRECTION FOR ROTATION  = <real> 1|2|3
SECOND ALPHA                   = <real> α2 (degrees)
```

Several anisotropic materials can be tied to an element specific coordinate frame. By default the material R, S, and T coordinate frame lines up with the global X, Y, and Z coordinate system. A different material R, S, and T coordinate frame can be defined with the above commands, for example to define material properties that vary in the hoop and radial directions of a cylinder.

Note, in addition to the coordinate system commands listed here, some material models have other model specific coordinate system options.

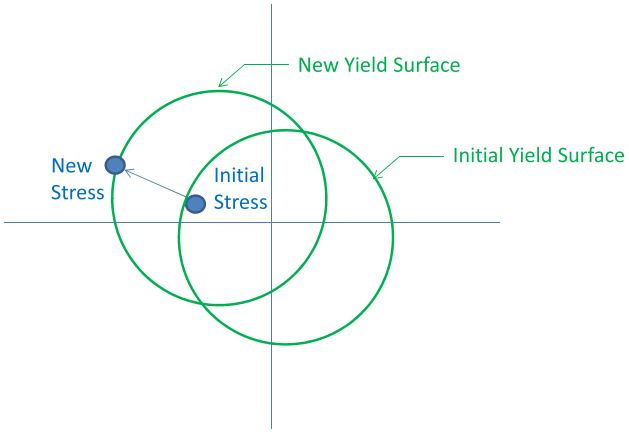


Figure 5-2. Kinematic Hardening, Beta = 0.0

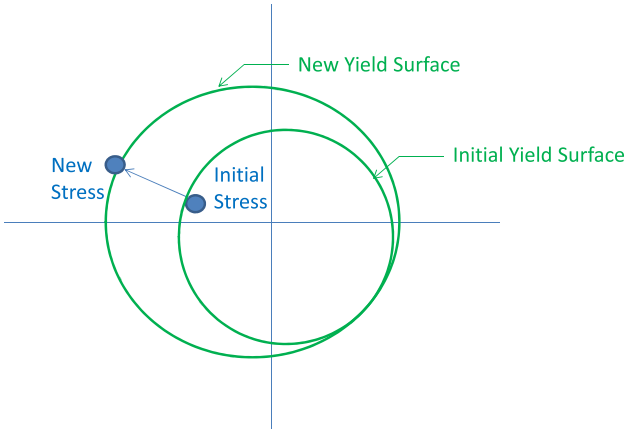


Figure 5-3. Mixed Hardening, Beta = 0.5

The specification of the principal material directions begins with the selection of a user-specified coordinate system given by the `COORDINATE SYSTEM` command line. The `coordinate_system_name` references a coordinate system defined by a separate `COORDINATE SYSTEM` command blocks (see Section 2.1.9). `COORDINATE SYSTEM` defines a

local R, S, and T frame for each element based on the element centroid. This initial coordinate system can then be rotated twice to give the final material directions at each material point.

The first rotation of the initial coordinate system is defined using the DIRECTION FOR ROTATION and ALPHA command lines. The axis for rotation of the initial coordinate system is specified by the DIRECTION FOR ROTATION command line, 1 corresponds to the initial coordinate system local R axis, 2 corresponds to the initial coordinate system local S axis, and 3 corresponds to the initial coordinate system local T axis. The right hand rule angle of rotation about this axis is given by ALPHA. This rotation yields an intermediate coordinate system.

A secondary rotation of the intermediate coordinate system may be defined using the SECOND DIRECTION FOR ROTATION and SECOND ALPHA command lines. The axis for rotation of the intermediate coordinate system is specified by the SECOND DIRECTION FOR ROTATION command line, 1 corresponds to the intermediate coordinate system local R axis, 2 corresponds to the intermediate coordinate system local S axis, and 3 corresponds to the intermediate coordinate system local T axis. The right hand rule angle of rotation about this axis is given by SECOND ALPHA. The resulting coordinate system gives the final R, S, and T coordinate frame for the material directions.

5.2. Material Models

This section contains descriptions of the general fully supported materials available in Sierra/SM. Additional models relevant to shock problems are described in the separate Sierra/SM 5.0 User's Guide: Addendum for Shock Capabilities. Also, additional experimental, in-development, and specialty models may be found in the Sierra/SM 5.0 Capabilities in Development manual.

5.2.1. Elastic Model

```
BEGIN PARAMETERS FOR MODEL ELASTIC
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
END [PARAMETERS FOR MODEL ELASTIC]
```

The elastic model is a hypoelastic extension of isotropic, small-strain, linear elasticity [2] [3] [4]. The stress-strain response for an isotropic, elastic material is

$$\sigma_{ij} = \lambda \delta_{ij} \varepsilon_{kk} + 2\mu \varepsilon_{ij} \tag{5.6}$$

where the Lamé constants, λ and μ, are given by

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)} \quad ; \quad \mu = \frac{E}{2(1+\nu)} \tag{5.7}$$

This model is extended to a finite-deformation, hypoelastic model by first making it a rate equation. Then the stress rate is replaced with an objective stress rate and the strain rate is replaced with the rate of deformation. This gives us

$$\overset{\circ}{\sigma}_{ij} = \lambda \delta_{ij} D_{kk} + 2\mu D_{ij} \tag{5.8}$$

The stress rate is arbitrary, as long as it is objective. Two objective stress rates are commonly used: the Jaumann rate and the Green-McInnis rate. For problems with fixed principal axes of deformation, these two rates give the same answers. For problems where the principal axes of deformation rotate during the deformation, the two rates can give different answers. Generally speaking there is no reason to pick one objective rate over another. Sierra/SM uses the Green-McInnis rate.

The fourth-order elastic moduli are used in many constitutive models. There are many equivalent representations for the elastic moduli. In index notation we present the following three representations

$$\overset{\circ}{\sigma}_{ij} = \mathbb{C}_{ijkl} D_{kl}$$

$$\mathbb{C}_{ijkl} = \frac{E}{1+\nu}\left[\frac{\nu}{1-2\nu}\delta_{ij}\delta_{kl} + \frac{1}{2}\left(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}\right)\right] \tag{5.9}$$

$$\mathbb{C}_{ijkl} = \lambda\delta_{ij}\delta_{kl} + \mu\left(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}\right) \tag{5.10}$$

$$\mathbb{C}_{ijkl} = K\delta_{ij}\delta_{kl} + \mu\left(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk} - \frac{2}{3}\delta_{ij}\delta_{kl}\right) \tag{5.11}$$

where K is the elastic bulk modulus and is given by

$$K = \frac{E}{3(1-2\nu)} \tag{5.12}$$

See Section 5.1.5 for more information on elastic constants input.

There are no output variables available for the elastic model. For information about the elastic model, consult [5].

5.2.2. Thermoelastic Model

```
BEGIN PARAMETERS FOR MODEL THERMOELASTIC
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Thermoelastic functions
  #
  YOUNGS MODULUS FUNCTION = <string> ym_function_name
  POISSONS RATIO FUNCTION = <string> pr_function_name
END [PARAMETERS FOR MODEL THERMOELASTIC]
```

The thermoelastic material model is used to describe a temperature-dependent linear elastic material. This is a hypoelastic model generally valid for small strains.

In the above command blocks:

- See Section 5.1.5 for more information on elastic constants input.
- The `YOUNGS MODULUS FUNCTION` command line references the name of a function defined in a `FUNCTION` command line in the SIERRA scope. The function defines a scale factor on Young's modulus as a function of temperature.
- The `POISSONS RATIO FUNCTION` command line references the name of a function defined in a `FUNCTION` command line in the SIERRA scope. The function defines a scale factor on Poisson's ratio as a function of temperature.

For information about the thermoelastic model, consult [5]. Several plasticity models share the same underlying implementation. Output state variables available for this model are listed in Table 5-9. Note, depending on options used (temperature dependence of properties, failure, etc.) some of these state variables may not be computed.

5.2.3. Neo-Hookean Model

```
BEGIN PARAMETERS FOR MODEL NEO_HOOKEAN
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
END [PARAMETERS FOR MODEL NEO_HOOKEAN]
```

The neo-Hookean model is a hyperelastic generalization of isotropic, small-strain linear elasticity. The stress-strain response for the neo-Hookean model may be determined from a free energy function - in this case the strain energy density, W. The form of the strain energy density ([6]) is

$$W(C_{ij}) = \frac{1}{2}K\left[\frac{1}{2}\left(J^2-1\right)-\ln J\right]+\frac{1}{2}\mu\left(\bar{C}_{kk}-3\right) \quad , \tag{5.13}$$

where K and μ are the bulk and shear moduli, respectively. The deformation measure is given by C_{ij}, the components of the right Cauchy-Green tensor, where $C_{ij} = F_{ki}F_{kj}$. The determinant of the deformation gradient is given by J and is a measure of the volumetric part of the deformation. $\bar{C}_{ij}$ provides the isochoric part of the deformation and is given by

$$\bar{C}_{ij} = \bar{F}_{ki}\bar{F}_{kj}, \quad ; \quad \bar{F}_{ij} = J^{-1/3}F_{ij} \quad . \tag{5.14}$$

The second Piola-Kirchoff stress, with components S_{ij}, may be determined by taking a derivative of the strain energy density and the Cauchy stress may be found by mapping from the second Piola-Kirchoff stress. The components of the Cauchy stress are

$$\sigma_{ij} = \frac{1}{2}K\left(J-\frac{1}{J}\right)\delta_{ij}+J^{-5/3}\mu\left(B_{ij}-\frac{1}{3}B_{kk}\delta_{ij}\right) \quad , \tag{5.15}$$

where $B_{ij} = F_{ik}F_{jk}$, are the components of the left Cauchy-Green tensor and δ_{ij} is the Kronecker delta.

Linearizing (5.15) we recover small strain linear elasticity

$$\begin{aligned}\sigma_{ij} &= \left(K-\frac{2}{3}\mu\right)u_{k,k}\delta_{ij}+\mu\left(u_{i,j}+u_{j,i}\right) \\ &= \left(K-\frac{2}{3}\mu\right)\varepsilon_{kk}\delta_{ij}+2\mu\varepsilon_{ij} \quad .\end{aligned} \tag{5.16}$$

The neo-Hookean model is used for the recoverable (elastic) part for a number of inelastic, finite deformation constitutive models.

There are no output variables available for the neo-Hookean model.

5.2.4. Elastic Fracture Model

```
BEGIN PARAMETERS FOR MODEL ELASTIC_FRACTURE
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Fracture parameters
  #
  MAX STRESS                     = <real> σ_max
  CRITICAL CRACK OPENING STRAIN = <real> ε_ccos
END [PARAMETERS FOR MODEL ELASTIC_FRACTURE]
```

The elastic fracture model is linear elastic model with a simple failure model using a maximum-principal-stress failure criterion. Post failure, the stress decays isotropically based on additional strain parallel to the maximum principal stress. The critical crack opening strain, ε_{ccos} is the value of the component of strain over which the stress is decayed to zero. This strain parameter can be adjusted so that failure is mesh independent.

In the above command blocks:

- See Section 5.1.5 for more information on elastic constants input.
- The maximum principal stress at which failure occurs is defined with the `MAX STRESS` command line.
- The component of strain over which the stress decays to zero is defined with the `CRITICAL CRACK OPENING STRAIN` command line. This component of strain is aligned with the maximum-principal-stress direction at failure.

Output variables available for this model are listed in Table 5-1.

Table 5-1. State Variables for ELASTIC FRACTURE Model (Section 5.2.4)

Name	Variable Description
CRACK_OPENING_STRAIN	Critical value of opening strain
FAILURE_DIRECTION	Crack opening direction - vector
FAILURE_DIRECTION_X	Crack opening direction - x component
FAILURE_DIRECTION_Y	Crack opening direction - y component
FAILURE_DIRECTION_Z	Crack opening direction - z component
PRINCIPAL_STRESS	Value of maximum principal stress
MAX_RADIUS	Von Mises stress at failure
MAX_PRESS	Stress tensor pressure at failure
MAX_STRESS	Maximum principal stress (UQ)
CRITICAL_CRACK_ OPENING_STRAIN	Critical crack opening strain (UQ)

5.2.5. Elastic-Plastic Model

```
BEGIN PARAMETERS FOR MODEL ELASTIC_PLASTIC
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Hardening Behavior
  #
  YIELD STRESS      = <real> σ_y
  BETA              = <real> β (1.0)
  HARDENING MODULUS = <real> H'
END [PARAMETERS FOR MODEL ELASTIC_PLASTIC]
```

The elastic-plastic model is a hypoelastic, rate-independent linear hardening plasticity model. The rate form of the constitutive equation assumes an additive split of the rate of deformation into an elastic and plastic part

$$D_{ij} = D^{\mathrm{e}}_{ij} + D^{\mathrm{p}}_{ij} \tag{5.17}$$

The stress rate only depends on the elastic strain rate in the problem

$$\overset{\circ}{\sigma}_{ij} = \mathbb{C}_{ijkl} D^{\mathrm{e}}_{kl} \tag{5.18}$$

where $\mathbb{C}_{ijkl}$ are the components of the fourth-order, isotropic elasticity tensor.

The key to the model is finding the plastic rate of deformation. For associated flow the plastic rate of deformation is in a direction normal to the yield surface. The yield surface is given by

$$f\left(\sigma_{ij}, \alpha_{ij}, \bar{\varepsilon}^p\right) = \phi\left(\sigma_{ij}, \alpha_{ij}\right) - \bar{\sigma}\left(\bar{\varepsilon}^p\right) = 0 \tag{5.19}$$

where ϕ is the effective stress, α_{ij} are the components of the back stress (used with kinematic hardening), and $\bar{\sigma}$ is the hardening function which is a function of an internal state variable, the equivalent plastic strain $\bar{\varepsilon}^p$. An example of such a yield surface (plotted in the deviatoric π-plane) is presented below in Figure 5-4. The isotropy of the yield surface is clearly evident.

For the elastic plastic model a linear hardening law is assumed

$$\bar{\sigma} = \sigma_y + H' \bar{\varepsilon}^p \tag{5.20}$$

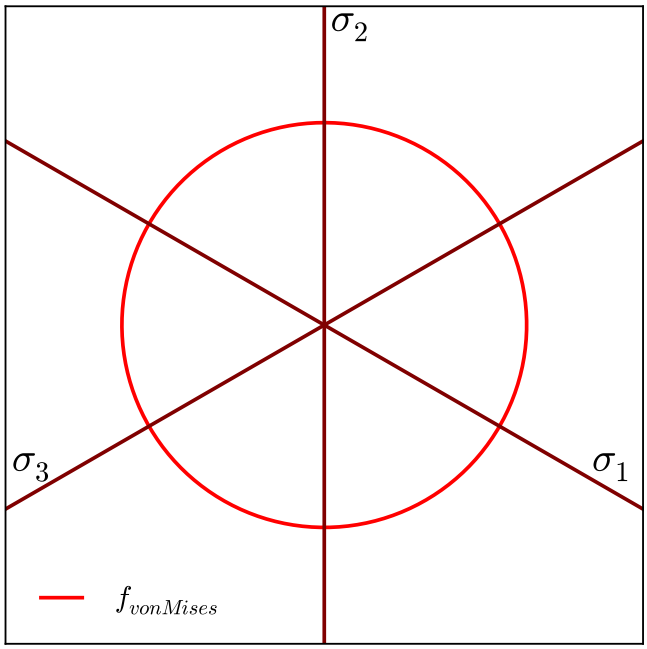


Figure 5-4. Example von Mises yield surface (J_2) used by the elastic-plastic model presented in the deviatoric π-plane. In this case the surface is plotted for $\alpha_{ij} = 0$ and $\bar{\varepsilon}^p = 0$.

where σ_y is the yield stress and H' is the hardening modulus.

If the stress state is such that $f < 0$, the the behavior of the material is elastic; if the stress state is such that $f = 0$ and $\dot{f} < 0$, i.e. the strain rate brings the stress inside the yield surface, then the behavior of the material is elastic; if the stress state is such that $f = 0$ and $\dot{f} > 0$, i.e. the strain rate brings the stress outside the yield surface, then plastic deformation occurs.

We assume associated flow in this model, which gives the plastic rate of deformation

$$D^{\mathrm{p}}_{ij} = \dot{\gamma}\frac{\partial \phi}{\partial \sigma_{ij}} \tag{5.21}$$

where $\dot{\gamma}$ is the consistency parameter. For the elastic-plastic model the yield surface is assumed to be a von Mises yield surface with a back stress tensor to denote the center of the yield surface. The effective stress for a von Mises yield surface is

$$\phi\left(\sigma_{ij}\right) = \sqrt{\frac{3}{2}\xi_{ij}\xi_{ij}} \quad ; \quad \xi_{ij} = s_{ij} - \alpha_{ij} \tag{5.22}$$

where s_{ij} are the components of the deviatoric stress tensor

$$s_{ij} = \sigma_{ij} - \frac{1}{3}\delta_{ij}\sigma_{kk} \tag{5.23}$$

and α_{ij} are the components of the back stress tensor, another internal state variable.

The equivalent plastic strain is found through equating the rate of plastic work

$$\dot{W}^p = \sigma_{ij} D^{\mathrm{p}}_{ij} = \bar{\sigma}\dot{\bar{\varepsilon}}^p \quad \rightarrow \quad \dot{\bar{\varepsilon}}^p = \dot{\gamma}$$
$$\bar{\varepsilon}^p = \int_0^t \dot{\gamma} dt \tag{5.24}$$

Finally, the model allows for kinematic hardening through the back stress. The back stress is a symmetric, deviatoric rank two tensor that evolves in the following manner

$$\dot{\alpha}_{ij} = \frac{2}{3}(1-\beta) H' D^{\mathrm{p}}_{ij} \tag{5.25}$$

The radius of the yield surface can be defined, $R = \sqrt{\xi_{ij}\xi_{ij}}$. The evolution of the radius of the yield surface is given by

$$\dot{R} = \sqrt{\frac{2}{3}}\beta H' \dot{\bar{\varepsilon}}^p \tag{5.26}$$

In (5.25) and (5.26) the parameter $\beta \in [0, 1]$ distributes the hardening between isotropic and kinematic hardening. If $\beta = 1$ the hardening is isotropic, if $\beta = 0$ the hardening is kinematic, and if β is between 0 and 1 the hardening is a combination of isotropic and kinematic.

In the above command blocks:

- See Section 5.1.5 for more information on elastic constants input. The elastic constants describe both the pre-yield behavior of the model and the slope of post yield unloading.
- The yield stress, the stress at which yield first initiates, is defined with the `YIELD STRESS` command line.
- The hardening modulus, the slope of the post yield hardening curve, is defined with the `HARDENING MODULUS` command line.
- The beta parameter defines if hardening is isotropic or kinematic. See Section 5.1.6 for details on the beta parameter.

Output variables available for this model are listed in Table 5-2 and Table 5-3. For information about the elastic-plastic model, consult [5].

Table 5-2. State Variables for ELASTIC PLASTIC Model (Section 5.2.5)

Name	Description
EQPS	equivalent plastic strain, $\bar{\varepsilon}^p$
RADIUS	radius of the yield surface, R
BACK_STRESS	back stress (symmetric tensor), α_{ij}

Table 5-3. State Variables for ELASTIC PLASTIC Model for Shells (Section 5.2.5)

Name	Description
EQPS	equivalent plastic strain, $\bar{\varepsilon}^p$
TENSILE_EQPS	equivalent plastic strain only accumulated when the material is in tension (trace of stress tensor is positive)
RADIUS	radius of the yield surface, R
BACK_STRESS	back stress (symmetric tensor), α_{ij}
ITERATIONS	radial return iterations
ERROR	error in plane stress iterations
PS_ITER	plane stress iterations
TSTRAIN	integrated thickness strain

5.2.6. Elastic-Plastic Power-Law Hardening Model

```
BEGIN PARAMETERS FOR MODEL EP_POWER_HARD
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Hardening behavior
  #
  YIELD STRESS       = <real> σ_y
  HARDENING CONSTANT = <real> A
  HARDENING EXPONENT = <real> n
  LUDERS STRAIN      = <real> ε_L
END [PARAMETERS FOR MODEL EP_POWER_HARD]
```

The elastic-plastic power law hardening model is a hypoelastic, rate-independent plasticity model with power law hardening [7]. The rate form of the constitutive equation assumes an additive split of the rate of deformation into an elastic and plastic part

$$D_{ij} = D^{\mathrm{e}}_{ij} + D^{\mathrm{p}}_{ij} \tag{5.27}$$

The stress rate only depends on the elastic strain rate in the problem

$$\overset{\circ}{\sigma}_{ij} = \mathbb{C}_{ijkl} D^{\mathrm{e}}_{kl} \tag{5.28}$$

where $\mathbb{C}_{ijkl}$ are the components of the fourth-order, isotropic elasticity tensor.

The key to integrating the model is finding the plastic rate of deformation. For associated flow the plastic rate of deformation is in a direction normal to the yield surface. The yield surface is given by

$$f\left(\sigma_{ij}, \bar{\varepsilon}^{p}\right) = \phi\left(\sigma_{ij}\right) - \bar{\sigma}\left(\bar{\varepsilon}^{p}\right) = 0 \tag{5.29}$$

where ϕ is the equivalent stress and $\bar{\sigma}$ is the hardening function which is a function of the equivalent plastic strain $\bar{\varepsilon}^{p}$. For this model the hardening function uses a power law

$$\bar{\sigma}\left(\bar{\varepsilon}^{p}\right) = \sigma_y + A\left\langle \bar{\varepsilon}^{p} - \varepsilon_L \right\rangle^{n} \tag{5.30}$$

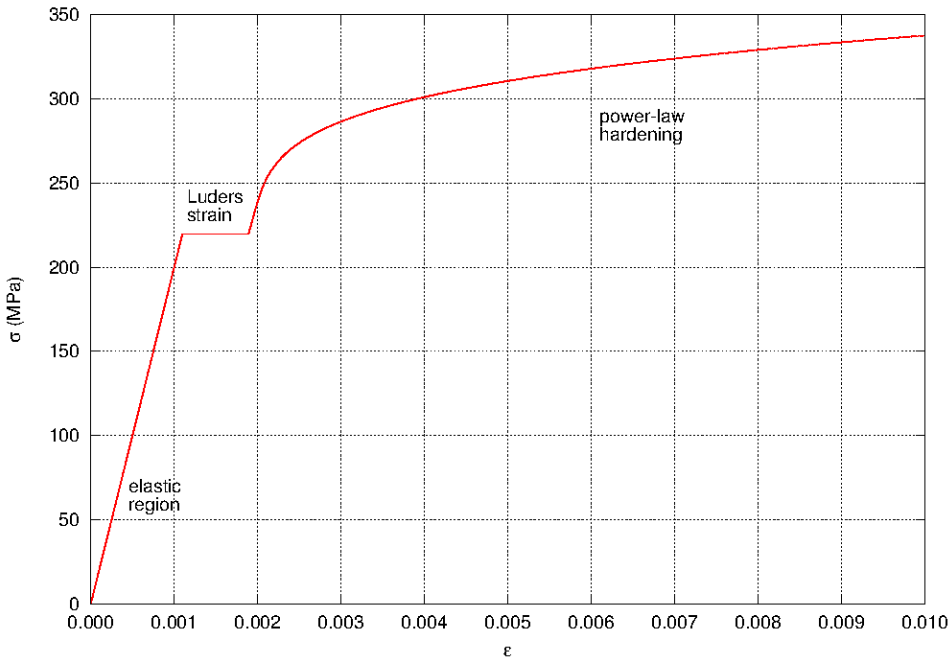


Figure 5-5. Typical stress-strain response for the power-law hardening model.

which is shown in Figure 5-5. The yield stress is σ_y, the hardening constant is A, the hardening exponent is n, and the Lüders strain is ε_L. The bracket $< \cdot >$ is the Macaulay bracket defined as

$$\langle x \rangle = \begin{cases} 0, & \text{if } x \leq 0 \\ x, & \text{if } x > 0. \end{cases} \tag{5.31}$$

By assuming associated plastic flow, the plastic rate of deformation can be written as

$$D_{ij}^{\mathrm{p}} = \dot{\gamma} \frac{\partial \phi}{\partial \sigma_{ij}}. \tag{5.32}$$

For this model the yield surface is chosen to be a von Mises yield surface, so

$$\phi\left(\sigma_{ij}\right) = \sqrt{\frac{3}{2} s_{ij} s_{ij}} \tag{5.33}$$

where s_{ij} are the components of the deviatoric stress

$$s_{ij} = \sigma_{ij} - \frac{1}{3} \delta_{ij} \sigma_{kk} \tag{5.34}$$

Unlike the elastic-plastic model 5.2.5, the power-law hardening model does not allow for kinematic hardening, so there is no back stress.

In the above command blocks:

- See Section 5.1.5 for more information on elastic constants input.
- The `YIELD STRESS` is the stress at which the plastic power law yielding and hardening model takes effect. See Figure 5-5.
- The `LUDERS STRAIN` defines a regime of zero hardening modulus prior to onset of the power law hardening. A small Luder band is seen in the hardening behavior or many metals. See Figure 5-5 for details.
- The `HARDENING CONSTANT` command line and `HARDENING EXPONENT` command define the power law hardening curve. Past the Luder strain the hardened yield surface radius is given by the `HARDENING CONSTANT` times plastic strain to the `HARDENING EXPONENT` power.

Output variables available for this model are listed in Table 5-4 and Table 5-5. For information about the elastic-plastic power-law hardening model, consult [5].

Table 5-4. State Variables for EP POWER HARD Model (Section 5.2.6)

Name	Description
`EQPS`	equivalent plastic strain, $\bar{\varepsilon}^p$
`TENSILE_EQPS`	equivalent plastic strain only accumulated when the material is in tension (trace of stress tensor is positive)
`RADIUS`	radius of yield surface, R
`ITERATIONS`	number of radial return iterations

Table 5-5. State Variables for EP POWER HARD Model for Shells (Section 5.2.6)

Name	Description
`EQPS`	equivalent plastic strain, $\bar{\varepsilon}^p$
`TENSILE_EQPS`	equivalent plastic strain only accumulated when the material is in tension (trace of stress tensor is positive)
`RADIUS`	radius of yield surface, R
`ITERATIONS`	number of radial return iterations
`ERROR`	error in plane stress iterations
`PS_ITER`	plane stress iterations

5.2.7. Ductile Fracture Model

```
BEGIN PARAMETERS FOR MODEL DUCTILE_FRACTURE
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Yield surface parameters
  #
  YIELD STRESS       = <real> σ_y
  HARDENING CONSTANT = <real> A
  HARDENING EXPONENT = <real> n
  LUDERS STRAIN      = <real> ε_L
  #
  # Failure parameters
  #
  CRITICAL TEARING PARAMETER    = <real> t_p^crit
  CRITICAL CRACK OPENING STRAIN = <real> ε_ccos
END [PARAMETERS FOR MODEL DUCTILE_FRACTURE]
```

The ductile fracture model is identical to the elastic-plastic power-law hardening model with the addition of a failure criterion and an isotropic decay of the stress to zero during the failure process within the constitutive model. To accomplish this task, the tearing parameter, t_p, proposed by Wellman [8] is introduced and the functional form as given as

$$t_p = \int_0^{\varepsilon} \left\langle \frac{2\sigma_{\max}}{3\left(\sigma_{\max} - \sigma_m\right)} \right\rangle^4 d\bar{\varepsilon}^p \tag{5.35}$$

where $\sigma_{\max}$ is the maximum principal stress, and σ_m is the mean stress. It can also be noted that the tearing parameter evolves during the plastic deformation regime as indicated by integrating over the effective plastic strain, $\bar{\varepsilon}^p$. The angle brackets denoting the Macaulay brackets, where

$$\langle x \rangle = \begin{cases} 0 & \text{if } x \le 0 \\ x & \text{if } x > 0 \end{cases}, \tag{5.36}$$

are used to ensure that the failure process occurs only with tensile stress states and prevent "damage healing". The failure process then initiates at a critical tearing parameter, t_p^{crit}, and the corresponding stress decay occurs over a strain interval corresponding to the *critical crack opening strain*, $\varepsilon_{\text{ccos}}$. Importantly, the $\varepsilon_{\text{ccos}}$ serves a dual role in that it may also be used to control

the energy dissipated during failure. With respect to the latter point, careful selection of the critical crack opening strain may be used to ensure consistent energy is dissipated through different meshes. This decay process is isotropic and linear with the current damage value being equivalent to the ratio of crack opening strain in the direction of the maximum principal stress to the critical value.

In the above command blocks:

- See Section 5.1.5 for more information on elastic constants input.
- The `YIELD STRESS` is the stress at which the plastic power load yielding and hardening model takes effect. See Figure 5-5.
- The `LUDERS STRAIN` defines a regime of zero hardening modulus prior to onset of the power law hardening. A small Luder band is seen in the hardening behavior or many metals. See Figure 5-5 for details.
- The `HARDENING CONSTANT` command line and `HARDENING EXPONENT` command define the power law hardening curve. Past the Luder strain the hardened yield surface radius is given by the `HARDENING CONSTANT` times plastic strain to the `HARDENING EXPONENT` power.
- `CRITICAL TEARING PARAMETER` defines the t_p value at which fracture and subsequent decay of stress will occur.
- When the model undergoes additionally strain after reaching the critical tearing parameter the stress in the model will decay to zero. The amount strain over which the stress decays to zero is defined with the `CRITICAL CRACK OPENING STRAIN` command line. The relevant opening strain is the component of strain that is aligned with the maximum-principal-stress direction at initial failure.

Output variables available for this model are listed in Table 5-6. For information about the ductile fracture material model, consult [8].

Table 5-6. State Variables for DUCTILE FRACTURE Model (Section 5.2.7)

Name	Description
EQPS	equivalent plastic strain, $\bar{\varepsilon}^p$
RADIUS	radius of yield surface, R
BACK_STRESS	back stress - tensor α_{ij}
TEARING_PARAMETER	Current value of the integrated tearing parameter
CRACK_OPENING_STRAIN	Current value of the crack opening strain. Will be zero prior to reaching the maximum tearing parameter.
FAILURE_DIRECTION	Crack opening direction (maximum principal stress direction at failure) - vector
DF_STRAIN_XX	XX component of current strain
DF_STRAIN_YY	YY component of current strain
DF_STRAIN_ZZ	ZZ component of current strain
DF_STRAIN_XY	XY component of current strain
DF_STRAIN_YZ	YZ component of current strain
DF_STRAIN_ZX	ZX component of current strain
MAX_RADIUS	Yield surface radius at failure
MAX_PRESS	Stress pressure norm at failure

5.2.8. Multilinear Elastic-Plastic Hardening Model

```
BEGIN PARAMETERS FOR MODEL MULTILINEAR_EP
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Hardening behavior
  #
  YIELD STRESS       = <real> σ_y
  BETA               = <real> β (1.0)
  HARDENING FUNCTION = <string> hardening_function_name
  #
  # Functions
  #
  YOUNGS MODULUS FUNCTION = <string> ym_function_name
  POISSONS RATIO FUNCTION = <string> pr_function_name
  YIELD STRESS FUNCTION   = <string> yield_stress_function_name
END [PARAMETERS FOR MODEL MULTILINEAR_EP]
```

The multilinear elastic-plastic model is a generalization of the standard rate independent plasticity models already presented - the linear and power law hardening models. However, rather than having a specific functional form, the multilinear hardening model allows the user to input a piecewise linear function for the hardening curve. The rate form of the constitutive equation assumes an additive split of the rate of deformation into an elastic and plastic part such that

$$D_{ij} = D^{\mathrm{e}}_{ij} + D^{\mathrm{p}}_{ij}. \tag{5.37}$$

The stress rate only depends on the elastic strain rate so that,

$$\overset{\circ}{\sigma}_{ij} = \mathbb{C}_{ijkl} D^{\mathrm{e}}_{kl}, \tag{5.38}$$

where $\mathbb{C}_{ijkl}$ are the components of the fourth-order, isotropic elasticity tensor.

The key to the model is finding the plastic rate of deformation. For associated flow, the plastic rate of deformation is in the direction normal to the yield surface. With a yield surface given by

$$\phi\left(\sigma_{ij}\right) - \bar{\sigma}\left(\bar{\varepsilon}^{p}\right) = 0 \tag{5.39}$$

then the plastic rate of deformation can be written as

$$D^{\mathrm{p}}_{ij} = \dot{\gamma}\frac{\partial \phi}{\partial \sigma_{ij}}. \tag{5.40}$$

For this model the yield surface is taken to be a von Mises yield surface, such that

$$\phi\left(\sigma_{ij}\right) = \sqrt{\frac{3}{2}s_{ij}s_{ij}} \tag{5.41}$$

where s_{ij} are the components of the deviatoric stress

$$s_{ij} = \sigma_{ij} - \frac{1}{3}\delta_{ij}\sigma_{kk}. \tag{5.42}$$

For simplicity it is easier to write (5.40) in terms of the normal to the yield surface

$$D^{\mathrm{p}}_{ij} = \dot{\gamma}N_{ij} \quad ; \quad N_{ij} = \frac{\partial \phi}{\partial \sigma_{ij}} \Big/ \left\| \frac{\partial \phi}{\partial \sigma_{ij}} \right\| \tag{5.43}$$

The model also incorporates temperature dependence in that the elastic properties and the yield stress can be functions of temperature. This is not as general as having the yield curves depend on temperature. For that behavior the thermoelastic-plastic model can be used.

An example stress vs. plastic strain hardening curve is shown in Figure 5-6. This curve was generated for a loading case of uniaxial strain. In this case, the effective stress is the same as the uniaxial. Therefore, for use with the multilinear elastic-plastic model this curve would simply have to be discretized and used as input.

In the above command blocks:

- See Section 5.1.5 for more information on elastic constants input.
- The beta parameter defines if hardening is isotropic or kinematic. See Section 5.1.6 for details on the beta parameter.
- `YIELD STRESS` defines the stress where plastic yielding first occurs.
- The `HARDENING FUNCTION` command line references the name of a function defined in a `FUNCTION` command line in the SIERRA scope. The function describes the hardening behavior of the material as stress versus equivalent plastic strain. The x values of the function should be values of equivalent plastic strain while the y values of the function can be either the increment of stress over the yield stress or the actual stress at the corresponding equivalent plastic strain. Note the hardening function can have its first point defined at (0,0), or at (0, `YIELD_STRESS`). Either function definition behaves the same as only the slope of the hardening function between two strains is used by the model.

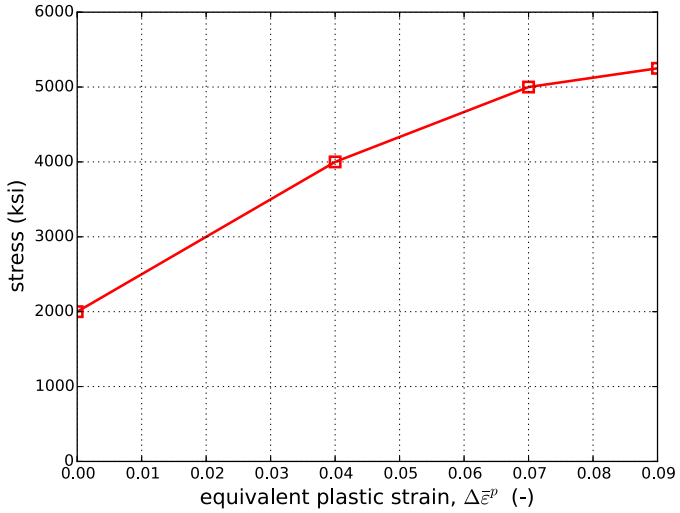


Figure 5-6. An example of a multilinear elastic-plastic stress-strain curve.

- The `YOUNGS MODULUS FUNCTION` command line references the name of a function defined in a `FUNCTION` command line in the SIERRA scope that describes a scale factor on Young's modulus as a function of temperature.
- The `POISSONS RATIO FUNCTION` command line references the name of a function defined in a `FUNCTION` command line in the SIERRA scope that describes a scale factor on Poisson's ratio as a function of temperature.
- The `YIELD STRESS FUNCTION` command line references the name of a function defined in a `FUNCTION` command line in the SIERRA scope that describes a scale factor on the yield stress as a function of temperature.

Output variables available for this model are listed in Table 5-7 and Table 5-8.

Table 5-7. State Variables for MULTILINEAR EP Model (Section 5.2.8)

Name	Description
EQPS	equivalent plastic strain
TENSILE_EQPS	equivalent plastic strain only accumulated when the material is in tension (trace of stress tensor is positive)
RADIUS	radius of yield surface
BACK_STRESS	back stress (symmetric tensor)
YOUNGS_MODULUS	the current Young's modulus as a function of temperature
POISSONS_RATIO	the current Poisson's ratio as a function of temperature
YIELD_STRESS	the current yield stress as a function of temperature
ITERATIONS	radial return iterations
YIELD_FLAG	inside (0) or on (1) the yield surface

Table 5-8. State Variables for MULTILINEAR EP Model for Shells (Section 5.2.8)

Name	Description
EQPS	equivalent plastic strain
TENSILE_EQPS	equivalent plastic strain only accumulated when the material is in tension (trace of stress tensor is positive)
RADIUS	radius of yield surface
BACK_STRESS	back stress (symmetric tensor)
YOUNGS_MODULUS	the current Young's modulus as a function of temperature
POISSONS_RATIO	the current Poisson's ratio as a function of temperature
YIELD_STRESS	the current yield stress as a function of temperature
ITERATIONS	radial return iterations
ERROR	error in plane stress iterations
PS_ITER	plane stress iterations

5.2.9. Multilinear Elastic-Plastic Hardening Model with Failure

```
BEGIN PARAMETERS FOR MODEL ML_EP_FAIL
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Hardening behavior
  #
  YIELD STRESS       = <real> σ_y
  BETA               = <real> β (1.0)
  HARDENING FUNCTION = <string> hardening_function_name
  #
  # Functions
  #
  YOUNGS MODULUS FUNCTION = <string> ym_function_name
  POISSONS RATIO FUNCTION = <string> pr_function_name
  YIELD STRESS FUNCTION   = <string> yield_stress_function_name
  #
  # Failure parameters
  #
  CRITICAL TEARING PARAMETER    = <real> t_p^crit
  CRITICAL CRACK OPENING STRAIN = <real> ε_ccos
  CRITICAL BIAXIALITY RATIO     = <real> critical_ratio(0.0)
  FAILURE EXPONENT              = <real> m (4.0)
END [PARAMETERS FOR MODEL ML_EP_FAIL]
```

Like the ductile fracture model, the multilinear elastic-plastic fail model is an extension of an existing plasticity model (multilinear elastic-plastic) to include a ductile failure criteria. Again, the tearing parameter criterion and failure propagation model of Wellman [8] is selected. Specifically, this approach uses a failure criterion (the tearing parameter, t_p) that is based on the history of the plastic strain and stress states. Most failure criteria for ductile failure involve some form of the stress triaxiality, or the ratio of the pressure and the effective (shear) stress. The tearing parameter, however, is slightly different in that it depends on the pressure and the maximum principal stress and is given as,

$$t_p = \int_0^{\varepsilon} \left\langle \frac{2\sigma_{\max}}{3\left(\sigma_{\max} - \sigma_m\right)} \right\rangle^m d\varepsilon_p, \tag{5.44}$$

with $\sigma_{\max}$ and σ_m being the maximum principal and mean stresses, respectively. The exponent m is typically taken to be 4 while the $\langle \cdot \rangle$ are Macaulay brackets defined as,

$$\langle x \rangle = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} , \tag{5.45}$$

and introduced so that failure only occurs and propagates under tensile stress states. Failure then initiates when the tearing parameter, t_p, reaches a critical value, t_p^{crit}. After this point, the stress decays (to 0) in a linear fashion according to the ratio of the crack opening strain in the maximum principal stress direction to its critical value, $\varepsilon_{\text{ccos}}$. Modification and control of this latter parameter is important as it may be used to ensure consistent energy is dissipated through different meshes.

In the above command blocks:

- See Section 5.1.5 for more information on elastic constants input.
- The beta parameter defines if hardening is isotropic or kinematic. See Section 5.1.6 for details on the beta parameter.
- `YIELD STRESS` defines the stress for onset of yielding and plasticity.
- The `HARDENING FUNCTION` command line references the name of a function defined in a `FUNCTION` command line in the SIERRA scope. The function describes the hardening behavior of the material as stress versus equivalent plastic strain. The x values of the function should be values of equivalent plastic strain while the y values of the function can be either the increment of stress over the yield stress or the actual stress at the corresponding equivalent plastic strain. Note the hardening function can have its first point defined at (0,0), or at (0, `YIELD_STRESS`). Either function definition behaves the same as only the slope of the hardening function between two strains is used by the model.
- The `YOUNGS MODULUS FUNCTION` command line references the name of a function defined in a `FUNCTION` command line in the SIERRA scope that describes a scale factor on Young's modulus as a function of temperature.
- The `POISSONS RATIO FUNCTION` command line references the name of a function defined in a `FUNCTION` command line in the SIERRA scope that describes a scale factor on Poisson's ratio as a function of temperature.
- The `YIELD STRESS FUNCTION` command line references the name of a function defined in a `FUNCTION` command line in the SIERRA scope that describes a scale factor on the yield stress as a function of temperature.
- `CRITICAL TEARING PARAMETER` defines the t_p value at which fracture and subsequent decay of stress will occur.
- When the model undergoes additionally strain after reaching the critical tearing parameter the stress in the model will decay to zero. The amount strain over which the stress decays to zero is defined with the `CRITICAL CRACK OPENING STRAIN` command line. The relevant opening strain is the component of strain that is aligned with the maximum-principal-stress direction at initial failure.
- The `CRITICAL BIAXIALITY RATIO` command line should only be used under highly specific conditions and with extreme caution. It is intended only for the special case where

the stress state is nearly biaxial, resulting in nearly identical principal strains. In this case, the eigenvector computation can give unreliable results for the direction vectors for the principal strains. If the ratio of the difference between two principal strains divided by their magnitude is less that the value specified by the `CRITICAL BIAXIALITY RATIO` command, the direction of the vector defining the crack opening strain will be given equal weight in each of the principal directions associated with those strains. The default value for the critical ratio is 0.0, which means that the principal directions will be accepted directly from the eigenvector computation. This command should only be used as a last resort if the loading is nearly biaxial and the default value has been demonstrated to lead to elements with high strains that are not failing long after reaching the critical tearing parameter.

- The `FAILURE EXPONENT` command line specifies the exponent on the tearing parameter, the *m* parameter in Equation 5.44. This exponent defaults to 4.0.

Output variables available for this model are listed in Table 5-9 and Table 5-10.

Table 5-9. State Variables for ML EP FAIL Model (Section 5.2.9)

Name	Variable Description
EQPS	Equivalent plastic strain
RADIUS	Radius of yield surface
BACK_STRESS	back stress - tensor
BACK_STRESS_XX	back stress - xx component
BACK_STRESS_YY	back stress - yy component
BACK_STRESS_ZZ	back stress - zz component
BACK_STRESS_XY	back stress - xy component
BACK_STRESS_YZ	back stress - yz component
BACK_STRESS_ZX	back stress - zx component
YOUNGS_MODULUS	Current Young's modulus as a function of temperature
POISSONS_RATIO	Current Poisson's ratio as a function of temperature
YIELD_STRESS	Current Yield stress as a function of temperature
TENSILE_EQPS	equivalent plastic strain only accumulated when the material is in tension (trace of stress tensor is positive)
ITERATIONS	radial return iterations
YIELD_FLAG	inside(0) or on(1) yield surface
TEARING_PARAMETER	Current integrated value of the tearing parameter. Zero until yield is reached
CRACK_OPENING_STRAIN	Current value of the crack opening strain. Zero until the critical tearing parameter is reached
FAILURE_DIRECTION	crack opening direction at failure - vector
FAILURE_DIRECTION_X	crack opening direction at failure - x component
FAILURE_DIRECTION_Y	crack opening direction at failure - y component
FAILURE_DIRECTION_Z	crack opening direction at failure - z component
MAX_RADIUS	maximum radius at initial failure
MAX_PRESSURE	maximum stress pressure norm at initial failure
CRITICAL_CRACK_ OPENING_STRAIN	
CRITICAL_TEARING_ PARAMETER	

Table 5-10. State Variables for ML EP FAIL Model for Shells(Section 5.2.9)

Name	Variable Description
EQPS	equivalent plastic strain
RADIUS	radius of yield surface
BACK_STRESS	back stress - tensor
BACK_STRESS_XX	back stress - xx component
BACK_STRESS_YY	back stress - yy component
BACK_STRESS_ZZ	back stress - zz component
BACK_STRESS_XY	back stress - xy component
BACK_STRESS_YZ	back stress - yz component
BACK_STRESS_ZX	back stress - zx component
YOUNGS_MODULUS	Current Young's modulus as a function of temperature
POISSONS_RATIO	Current Poisson's ratio as a function of temperature
YIELD_STRESS	Current Yield stress as a function of temperature
ITER	radial return iterations
ERROR	Error in plane stress iterations
PS_ITER	Plane stress iterations
TEARING_PARAMETER	Current integrated value of the tearing parameter. Zero until yield is reached
CRACK_OPENING_STRAIN	Current value of the crack opening strain. Zero until the critical tearing parameter is reached
FAILURE_DIRECTION	crack opening direction at failure - vector
FAILURE_DIRECTION_X	crack opening direction at failure - x component
FAILURE_DIRECTION_Y	crack opening direction at failure - y component
FAILURE_DIRECTION_Z	crack opening direction at failure - z component
RADIUS_MAX	maximum radius at initial failure
TENSILE_EQPS	equivalent plastic strain only accumulated when the material is in tension (trace of stress tensor is positive)

5.2.10. Elastic-Plastic Hardening Model with Failure

```
BEGIN PARAMETERS FOR MODEL ELASTIC_PLASTIC_FAIL
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Isotropic/Kinematic hardening factor
  #
  BETA         = <real> β (1.0)
  YIELD STRESS = <real> σ_y
  HARDENING FUNCTION      = <string>hardening_function_name
  POISSONS RATIO FUNCTION = <string>pr_function_name
  YIELD STRESS FUNCTION   = <string>yield_stress_function_name
  YOUNGS MODULUS FUNCTION = <string>ym_function_name
  #
  # Failure parameters
  #
  CRITICAL TEARING PARAMETER    = <real>crit_tearing
  CRITICAL CRACK OPENING STRAIN = <real>critical_strain
END [PARAMETERS FOR MODEL ELASTIC_PLASTIC_FAIL]
```

This model behaves identical to the `ML_EP_FAIL` (Section 5.2.9) material mode and only exists for legacy syntax compatibility purposes.

5.2.11. Thermoelastic-Plastic Model

```
BEGIN PARAMETERS FOR MODEL THERMOELASTIC_PLASTIC
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Isotropic/Kinematic hardening factor
  #
  BETA = <real>beta_parameter(1.0)
  YIELD STRESS = <real>yield_stress
  YOUNGS MODULUS FUNCTION = <string>ym_function_name
  POISSONS RATIO FUNCTION = <string>pr_function_name
  YIELD STRESS FUNCTION = <string>ys_function_name
  TEMPERATURES = <real(s)>temperature_values
  HARDENING FUNCTIONS = <string(s)>hardening_functions
END [PARAMETERS FOR MODEL THERMOELASTIC_PLASTIC]
```

The thermoelastic-plastic model is similar to the elastic-plastic model, but allows for temperature-dependent changes of the material properties. All elastic-plastic input parameters are still valid for the thermoelastic-plastic model, in addition to the temperature dependent functions.

In the above command blocks:

- See Section 5.1.5 for more information on elastic constants input.
- The beta parameter defines if hardening is isotropic or kinematic. See Section 5.1.6 for details on the beta parameter.
- The yield stress, the stress at which yield first initiates, is defined with the `YIELD STRESS` command line.
- The `YIELD STRESS FUNCTION` command line references the name of a function defined in a `FUNCTION` command line in the SIERRA scope that describes a scale factor on Yield Stress as a function of temperature.
- The `YOUNGS MODULUS FUNCTION` command line references the name of a function defined in a `FUNCTION` command line in the SIERRA scope that describes a scale factor on Young's modulus as a function of temperature.
- The `POISSONS RATIO FUNCTION` command line references the name of a function defined in a `FUNCTION` command line in the SIERRA scope that describes a scale factor on Poisson's ratio as a function of temperature.
- The `TEMPERATURES` command line specifies the temperatures that correspond to the hardening functions defined in the `HARDENING FUNCTIONS` command line. The number of

temperatures input here must match the number of hardening functions input.

- The `HARDENING FUNCTIONS` command line references the names of the functions defined in a `FUNCTION` command line in the SIERRA scope that describes the hardening behavior for the material as stress versus equivalent plastic strain at the temperatures defined in the `TEMPERATURES` command line. The number of hardening functions input here must match the number of temperatures input. If the material temperature exactly matches one of the input temperatures the hardening of the material will exactly follow the hardening curve associated with that temperature. If the material temperature is bracketed by two hardening curves the actual hardening curve taken will be linearly interpolated between the bracketing curves. If the material temperature is either above the highest defined temperature function or below the lowest defined temperature function the closest hardening curve will be used for the material hardening.

Several plasticity models share the same underlying implementation. Output state variables available for this model are listed in Table 5-9. Note, depending on options used (temperature dependence of properties, failure, etc.) some of these state variables may not be computed.

5.2.12. Thermoelastic-Plastic Model with Failure

```
BEGIN PARAMETERS FOR MODEL THERMOELASTIC_PLASTIC_FAIL
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Isotropic/Kinematic hardening factor
  #
  BETA = <real>beta_parameter(1.0)
  TEMPERATURES = <real(s)>temperature_values
  YIELD STRESS = <real>yield_stress
  YOUNGS MODULUS FUNCTION = <string>ym_function_name
  POISSONS RATIO FUNCTION = <string>pr_function_name
  YIELD STRESS FUNCTION = <string>ys_function_name
  HARDENING FUNCTIONS = <string(s)>hardening_functions
  CRITICAL CRACK OPENING STRAIN = <real>critical_strain
  CRITICAL TEARING PARAMETER = <real>crit_tearing
  CRITICAL CRACK OPENING STRAIN FUNCTION
     = <string>crack_open_strain_function_name
  CRITICAL TEARING PARAMETER FUNCTION
     = <string>tearing_parameter_function_name
END [PARAMETERS FOR MODEL THERMOELASTIC_PLASTIC_FAIL]
```

The thermoelastic-plastic model with failure is similar to the `ELASTIC_PLASTIC_FAIL` material model, but allows for additional temperature-dependent changes of the material.

In the above command blocks:

- See Section 5.1.5 for more information on elastic constants input.
- The beta parameter defines if hardening is isotropic or kinematic. See Section 5.1.6 for details on the beta parameter.
- `YIELD STRESS` defines the stress for onset of yielding and plasticity.
- The `TEMPERATURES` command line specifies the temperatures that correspond to the hardening functions defined in the `HARDENING FUNCTIONS` command line. The number of temperatures input here must match the number of hardening functions input.
- The `HARDENING FUNCTIONS` command line references the names of the functions defined in a `FUNCTION` command line in the SIERRA scope that describes the hardening behavior for the material as stress versus equivalent plastic strain at the temperatures defined in the `TEMPERATURES` command line. The number of hardening functions input here must match the number of temperatures input. If the material temperature exactly matches one of the input temperatures the hardening of the material will exactly follow the hardening curve

associated with that temperature. If the material temperature is bracketed by two hardening curves the actual hardening curve taken will be linearly interpolated between the bracketing curves. If the material temperature is either above the highest defined temperature function or below the lowest defined temperature function the closest hardening curve will be used for the material hardening.

- The `POISSONS RATIO FUNCTION` command line references the name of a function defined in a `FUNCTION` command line in the SIERRA scope that describes a scale factor on Poisson's ratio as a function of temperature.
- The `YIELD STRESS FUNCTION` command line references the name of a function defined in a `FUNCTION` command line in the SIERRA scope that describes a scale factor on the yield stress as a function of temperature.
- `CRITICAL TEARING PARAMETER` defines the t_p value at which fracture and subsequent decay of stress will occur.
- When the model undergoes additionally strain after reaching the critical tearing parameter the stress in the model will decay to zero. The amount strain over which the stress decays to zero is defined with the `CRITICAL CRACK OPENING STRAIN` command line. The relevant opening strain is the component of strain that is aligned with the maximum-principal-stress direction at initial failure.
- The `CRITICAL CRACK OPENING STRAIN FUNCTION` command line references the name of a function defined in a `FUNCTION` command line in the SIERRA scope that describes a scale factor on the crack opening strain as a function of temperature.
- The `CRITICAL TEARING PARAMETER FUNCTION` command line references the name of a function defined in a `FUNCTION` command line in the SIERRA scope that describes a scale factor on the tearing parameter as a function of temperature.

Several plasticity models share the same underlying implementation. Output state variables available for this model are listed in Table 5-9. Note, depending on options used (temperature dependence of properties, failure, etc.) some of these state variables may not be computed.

5.2.13. Hosford Plasticity Model

```
BEGIN PARAMETERS FOR MODEL HOSFORD_PLASTICITY
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Yield surface parameters
  #
  YIELD STRESS = <real> σ_y
  A            = <real> a (1.0)
  #
  # Hardening model
  #
  HARDENING MODEL = LINEAR | POWER_LAW | VOCE | USER_DEFINED |
    FLOW_STRESS | DECOUPLED_FLOW_STRESS | JOHNSON_COOK |
    POWER_LAW_BREAKDOWN
  #
  # Linear hardening
  #
  HARDENING MODULUS = <real> H'
  #
  # Power-law hardening
  #
  HARDENING CONSTANT = <real> A
  HARDENING EXPONENT = <real> n (0.5)
  LUDERS STRAIN      = <real> ε_L (0.0)
  #
  # Voce hardening
  #
  HARDENING MODULUS       = <real> A
  EXPONENTIAL COEFFICIENT = <real> n
  #
  # Johnson-Cook hardening
  #
  HARDENING FUNCTION = <string>hardening_function_name
  RATE CONSTANT      = <real> C
  REFERENCE RATE     = <real> ε̇_0
  #
  # Power law breakdown hardening
  #
```

```
HARDENING FUNCTION = <string>hardening_function_name
RATE COEFFICIENT   = <real> g
RATE EXPONENT      = <real> m
#
# User defined hardening
#
HARDENING FUNCTION = <string>hardening_function_name
#
#
# Following Commands Pertain to Flow_Stress Hardening Model
#
#    -  Isotropic Hardening model
#
ISOTROPIC HARDENING MODEL = LINEAR | POWER_LAW | VOCE |
                            USER_DEFINED
#
# Specifications for Linear, Power-law, and Voce same as above
#
# User defined hardening
#
ISOTROPIC HARDENING FUNCTION = <string>iso_hardening_fun_name
#
#     - Rate dependence
#
RATE MULTIPLIER = JOHNSON_COOK | POWER_LAW_BREAKDOWN |
   USER_DEFINED | RATE_INDEPENDENT (RATE_INDEPENDENT)
#
# Specifications for Johnson-Cook, Power-law-breakdown
#     same as before EXCEPT no need to specify a
#     hardening function
#
# User defined rate multiplier
#
RATE MULTIPLIER FUNCTION = <string> rate_mult_function_name
#
#     - Temperature dependence
#
TEMPERATURE MULTIPLIER = JOHNSON_COOK | USER_DEFINED |
   TEMPERATURE_INDEPENDENT (TEMPERATURE_INDEPENDENT)
#
#  Johnson-Cook temperature dependence
#
MELTING TEMPERATURE   = <real> θ_melt
REFERENCE TEMPERATURE = <real> θ_ref
TEMPERATURE EXPONENT  = <real> M
#
#  User-defined temperature dependence
```

```
 TEMPERATURE MULTIPLIER FUNCTION = <string>temp_mult_function_name
#
 #
 # Following Commands Pertain to Decoupled_Flow_Stress Hardening Model
 #
 #    -  Isotropic Hardening model
 #
 ISOTROPIC HARDENING MODEL = LINEAR | POWER_LAW | VOCE | USER_DEFINED
 #
 # Specifications for Linear, Power-law, and Voce same as above
 #
 # User defined hardening
 #
 ISOTROPIC HARDENING FUNCTION = <string>isotropic_hardening_function_name
 #
 #     - Rate dependence
 #
 YIELD RATE MULTIPLIER = JOHNSON_COOK | POWER_LAW_BREAKDOWN |
             USER_DEFINED | RATE_INDEPENDENT (RATE_INDEPENDENT)
 #
 # Specifications for Johnson-Cook, Power-law-breakdown same as before
 #     EXCEPT no need to specify a hardening function
 #     AND should be preceded by YIELD
 #
 #     As an example for Johnson-Cook yield rate dependence,
 #
 YIELD RATE CONSTANT  = <real> C^y
 YIELD REFERENCE RATE = <real> ε̇_0^y
 #
 # User defined rate multiplier
 #
 YIELD RATE MULTIPLIER FUNCTION = <string>yield_rate_mult_function_name
 #
 HARDENING_RATE MULTIPLIER = JOHNSON_COOK | POWER_LAW_BREAKDOWN |
             USER_DEFINED | RATE_INDEPENDENT (RATE_INDEPENDENT)
 #
 #     Syntax same as for yield parameters but with a HARDENING prefix
 #
 #     - Temperature dependence
 #
 YIELD TEMPERATURE MULTIPLIER = JOHNSON_COOK | USER_DEFINED |
           TEMPERATURE_INDEPENDENT (TEMPERATURE_INDEPENDENT)
 #
 #  Johnson-Cook temperature dependence
 #
 YIELD MELTING TEMPERATURE   = <real> θ_melt^y
 YIELD REFERENCE TEMPERATURE = <real> θ_ref^y
```

```
  YIELD TEMPERATURE EXPONENT  = <real> M^Y
  #
  #  User-defined temperature dependence
  YIELD TEMPERATURE MULTIPLIER FUNCTION = <string>yield_temp_mult_fun_name
  #
  HARDENING TEMPERATURE MULTIPLIER = JOHNSON_COOK | USER_DEFINED |
            TEMPERATURE_INDEPENDENT (TEMPERATURE_INDEPENDENT)
  #
  #   Syntax for hardening constants same as for yield but
  #            with HARDENING prefix
#
  #
  # Optional Failure Definitions
  #    Following only need to be defined if intend to use failure model
  #
  FAILURE MODEL = TEARING_PARAMETER | JOHNSON_COOK_FAILURE | WILKINS
                | MODULAR_FAILURE | MODULAR_BCJ_FAILURE

  CRITICAL FAILURE PARAMETER = <real> d_crit
  #
  # TEARING_PARAMETER Failure model definitions
  #
  TEARING PARAMETER EXPONENT  = m
  #
  # JOHNSON_COOK_FAILURE Failure model definitions
  #
  JOHNSON COOK D1       = <real> D1
  JOHNSON COOK D2       = <real> D2
  JOHNSON COOK D3       = <real> D3
  JOHNSON COOK D4       = <real> D4
  JOHNSON COOK D5       = <real> D5
  #
  #Following Johnson-Cook parameters can only be defined once.  As such, only
  #  needed if not previously defined via Johnson-Cook multipliers
  #  w/ flow-stress hardening.  Does need to be defined
  #  w/ Decoupled Flow Stress
  #
  REFERENCE RATE        = <real> ε̇0
  REFERENCE TEMPERATURE = <real> T_ref
  MELTING TEMPERATURE   = <real> T_melt
  #
  # WILKINS Failure model definitions
  #
  WILKINS ALPHA    = <real> α
  WILKINS BETA     = <real> β
  WILKINS PRESSURE = <real> B
  #
```

```
#  MODULAR_FAILURE Failure model definitions
#
PRESSURE MULTIPLIER         = PRESSURE_INDEPENDENT | WILKINS
                              | USER_DEFINED (PRESSURE_INDEPENDENT)
LODE ANGLE MULTIPLIER       = LODE_ANGLE_INDEPENDENT |
                              WILKINS (LODE_ANGLE_INDEPENDENT)
TRIAXIALITY MULTIPLIER      = TRIAXIALITY_INDEPENDENT | JOHNSON_COOK
                              | USER_DEFINED (TRIAXIALITY_INDEPENDENT)
RATE FAIL MULTIPLIER        = RATE_INDEPENDENT | JOHNSON_COOK
                              | USER_DEFINED (RATE_INDEPENDENT)
TEMPERATURE FAIL MULTIPLIER = TEMPERATURE_INDEPENDENT | JOHNSON_COOK
                              | USER_DEFINED (TEMPERATURE_INDEPENDENT)
#
# Individual multiplier definitions
#
PRESSURE MULTIPLIER = WILKINS
WILKINS ALPHA       = <real> α
WILKINS PRESSURE    = <real> B
#
PRESSURE MULTIPLIER = USER_DEFINED
PRESSURE MULTIPLIER FUNCTION = <string> pressure_multiplier_fun_name
#
LODE ANGLE MULTIPLIER = WILKINS
WILKINS BETA          = <real> β
#
TRIAXIALITY MULTIPLIER = JOHNSON_COOK
JOHNSON COOK D1        = <real> D1
JOHNSON COOK D2        = <real> D2
JOHNSON COOK D3        = <real> D3
#
TRIAXIALITY MULTIPLIER = USER_DEFINED
TRIAXIALITY MULTIPLIER FUNCTION = <string> triaxiality_multiplier_fun_name
#
RATE FAIL MULTIPLIER = JOHNSON_COOK
JOHNSON COOK D4      = <real> D4
# REFERENCE RATE should only be added if not previously defined
REFERENCE RATE       = <real> ε̇0
#
RATE FAIL MULTIPLIER = USER_DEFINED
RATE FAIL MULTIPLIER FUNCTION = <string> rate_fail_multiplier_fun_name
#
TEMPERATURE FAIL MULTIPLIER = JOHNSON_COOK
JOHNSON COOK D5             = <real> D5
# JC Temperatures should only be defined if not previously given
REFERENCE TEMPERATURE       = <real> Tref
MELTING TEMPERATURE         = <real> Tmelt
#
```

```
TEMPERATURE FAIL MULTIPLIER          = USER_DEFINED
TEMPERATURE FAIL MULTIPLIER FUNCTION = <string> temp_multiplier_fun_name
#
# MODULAR BCJ_FAILURE Failure model definitions
#
INITIAL DAMAGE    = <real> φ_{t=0}
INITIAL VOID SIZE = <real> υ_υ 0
DAMAGE BETA       = <real> β (0.5)
GROWTH MODEL      = COCKS_ASHBY | NO_GROWTH (NO_GROWTH)
NUCLEATION MODEL  = HORSTEMEYER_GOKHALE | CHU_NEEDLEMAN_STRAIN
                    | NO_NUCLEATION (NO_NUCLEATION)
#
GROWTH RATE FAIL MULTIPLIER        = JOHNSON_COOK | USER_DEFINED
                                     | RATE_INDEPENDENT
                                     (RATE_INDEPENDENT)
GROWTH TEMPERATURE FAIL MULTIPLIER = JOHNSON_COOK | USER_DEFINED
                                     | TEMPERATURE_INDEPENDENT
                                     (TEMPERATURE_INDEPENDENT)
#
NUCLEATION RATE FAIL MULTIPLIER        = JOHNSON_COOK | USER_DEFINED
                                         | RATE_INDEPENDENT
                                         (RATE_INDEPENDENT)
NUCLEATION TEMPERATURE FAIL MULTIPLIER = JOHNSON_COOK | USER_DEFINED
                                         | TEMPERATURE_INDEPENDENT
                                         (TEMPERATURE_INDEPENDENT)
#
# Definitions for individual growth and nucleation models
#
GROWTH MODEL = COCKS_ASHBY
DAMAGE EXPONENT = <real> m (0.5)
#
NUCLEATION MODEL      = HORSTEMEYER_GOKHALE
NUCLEATION PARAMETER1 = <real> N_1 (0.0)
NUCLEATION PARAMETER2 = <real> N_2 (0.0)
NUCLEATION PARAMETER3 = <real> N_3 (0.0)
#
NUCLEATION MODEL          = CHU_NEEDLEMAN_STRAIN
NUCLEATION AMPLITUDE      = <real> ψ
MEAN NUCLEATION STRAIN    = <real> ε_N
NUCLEATION STRAIN STD DEV = <real> s
#
# Definitions for rate and temperature fail multiplier
#   Note: only showing definitions for growth.
#    Nucleation terms are the same just with NUCLEATION instead
#    of GROWTH
#
GROWTH RATE FAIL MULTIPLIER = JOHNSON_COOK
```

```
  GROWTH JOHNSON COOK D4       = <real> D_4^g
  GROWTH REFERENCE RATE        = <real> ε̇_0^g
  #
  GROWTH RATE FAIL MULTIPLIER          = USER_DEFINED
  GROWTH RATE FAIL MULTIPLIER FUNCTION = <string> growth_rate_fail_mult_func
  #
  GROWTH TEMPERATURE FAIL MULTIPLIER = JOHNSON_COOK
  GROWTH JOHNSON COOK D5             = <real> D_5^g
  GROWTH REFERENCE TEMPERATURE       = <real> T_ref^g
  GROWTH MELTING TEMPERATURE         = <real> T_melt^g
  #
  GROWTH TEMPERATURE FAIL MULTIPLIER          = USER_DEFINED
  GROWTH TEMPERATURE FAIL MULTIPLIER FUNCTION = <string> temp_fail_mult_func
  #
  #
  # Optional Adiabatic Heating/Thermal Softening  Definitions
  #    Following only need to be defined if intend to use failure model
  #
  THERMAL SOFTENING MODEL = ADIABATIC | COUPLED
  #
  SPECIFIC HEAT = <real> c_p #not needed for COUPLED
  BETA_TQ       = <real> β^TQ
  T0            = <real> T_{t=0}
END [PARAMETERS FOR MODEL HOSFORD_PLASTICITY]
```

Like other elastic-plastic models in Lamé, the Hosford plasticity model is a rate-independent hypoelastic formulation. Unlike the Hill and other more complex plasticity models, it is isotropic. In a similar fashion to those models, the total rate of deformation is additively decomposed into an elastic and plastic part such that

$$D_{ij} = D^{\mathrm{e}}_{ij} + D^{\mathrm{p}}_{ij}. \tag{5.46}$$

The objective stress rate, depending only on the elastic deformation, may then be written as,

$$\overset{\circ}{\sigma}_{ij} = \mathbb{C}_{ijkl} D^{\mathrm{e}}_{kl}. \tag{5.47}$$

The Hosford plasticity model utilizes a yield surface first put forth by W. F. Hosford in the 1970's [9] that is isotropic but non-quadratic. This specific form was proposed due to experimental observations of biaxial stretching in which neither the Tresca or J_2 yield surfaces could describe the results. In contrast to many of the yield surfaces proposed for similar purposes, only two parameters are utilized. Even with these limited terms, the developed model is quite versatile and can be reduced to von Mises or Tresca conditions as well as capturing responses in between. This yield surface is given as,

$$f\left(\sigma_{ij}, \bar{\varepsilon}^p\right) = \phi\left(\sigma_{ij}\right) - \bar{\sigma}\left(\bar{\varepsilon}^p\right) = 0, \tag{5.48}$$

in which $\phi\left(\sigma_{ij}\right)$ is the Hosford effective stress and $\bar{\sigma}(\bar{\varepsilon}^p)$ is the current yield stress that may depend on rate and/or temperature. The Hosford effective stress is a non-quadratic function of the principal stresses ($\sigma_i,\ \ i = 1,2,3$) and is given as

$$\phi\left(\sigma_{ij}\right) = \left[\frac{|\sigma_1 - \sigma_2|^a + |\sigma_2 - \sigma_3|^a + |\sigma_1 - \sigma_3|^a}{2}\right]^{1/a} \tag{5.49}$$

in which a is the yield surface exponent. Interestingly, if $a = 2$ or 4 the yield surface reduces to that of a J_2 von Mises surface while $a = 1$ or as $a \to \infty$ produces a Tresca like shape. If the value of a is above 4 the yield surface takes a position between the Tresca and J_2 limits. Typical values are $a = 6$ or $a = 8$ for *bcc* and *fcc* metals, respectively [10]. To highlight this variability the yield surface

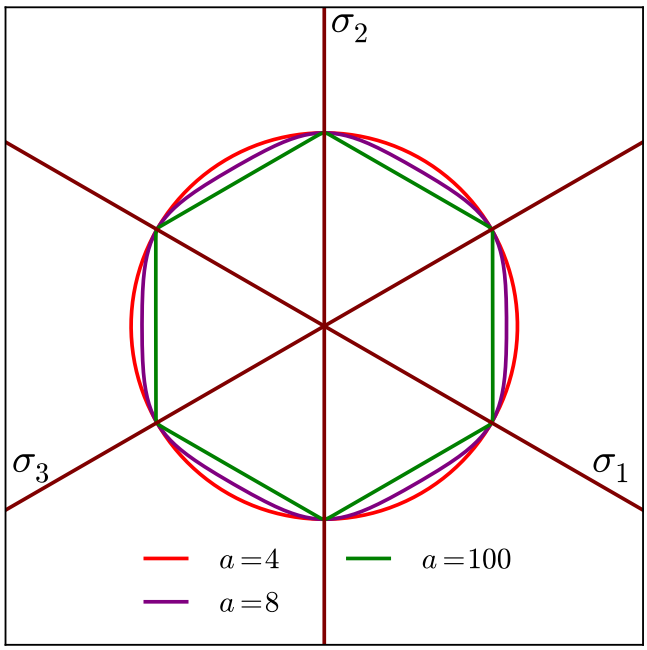


Figure 5-7. Example Hosford yield surfaces, $f\left(\sigma_{ij}, \bar{\varepsilon}^p = 0; a\right)$, presented in the deviatoric π-plane. The presented surfaces correspond to the different yield exponents $a = 4, 8,$ and 100.

For the hardening function, $\bar{\sigma}(\bar{\varepsilon}^p)$, a variety of forms including linear, power law, or a more general user defined function may be used.

An associated flow rule is utilized such that the plastic rate of deformation is normal to the yield surface and is given by,

$$\dot{D}^{\mathrm{p}}_{ij} = \dot{\gamma}\frac{\partial \phi}{\partial \sigma_{ij}}, \tag{5.50}$$

where $\dot{\gamma}$ is the consistency multiplier enforcing $f = 0$ during plastic deformation. Given the form of f, it can also be shown that $\dot{\gamma} = \dot{\bar{\varepsilon}}^p$.

For details on the plasticity model, please see [11]. Additional details on failure models and adiabatic heating capabilities may be found in [12, 13] and [14], respectively.

In the command blocks that define the Hosford plasticity model:

- See Section 5.1.5 for more information on elastic constants input.
- The reference nominal yield stress, $\bar{\sigma}$, is defined with the `YIELD STRESS` command line.
- The yield surface exponent, a, is defined with the `A` command line.
- The type of hardening law is defined with the `HARDENING MODEL` command line, other hardening commands then define the specific shape of that hardening curve.
- The hardening modulus for a linear hardening model is defined with the `HARDENING MODULUS` command line.
- The hardening constant for a power law hardening model is defined with the `HARDENING CONSTANT` command line.
- The hardening exponent for a power law hardening model is defined with the `HARDENING EXPONENT` command line.
- The Lüders strain for a power law hardening model is defined with the `LUDERS STRAIN` command line.
- The hardening function for a user defined hardening model is defined with the `HARDENING FUNCTION` command line.
- The shape of the spline for the spline based hardening is defined by the `CUBIC SPLINE TYPE`, `CARDINAL PARAMETER`, `KNOT EQPS`, and `KNOT STRESS` command lines.

Output variables available for this model are listed in Table 5-11.

Table 5-11. State Variables for HOSFORD PLASTICITY Model (Section 5.2.13)

Name	Description
`EQPS`	equivalent plastic strain, $\bar{\varepsilon}^p$
`EQDOT`	equivalent plastic strain rate, $\dot{\bar{\varepsilon}}^p$
`SEFF`	effective stress, ϕ
`TENSILE_EQPS`	tensile equivalent plastic strain, $\bar{\varepsilon}_t^p$
`DAMAGE`	damage, ϕ
`VOID_COUNT`	void count, η
`VOID_SIZE`	void size, υ
`DAMAGE_DOT`	damage rate, $\dot{\phi}$
`VOID_COUNT_DOT`	void count rate, $\dot{\eta}$
`PLASTIC_WORK_HEAT_RATE`	plastic work heat rate, $\dot{Q}^p$

5.2.14. Hill Plasticity Model

```
BEGIN PARAMETERS FOR MODEL HILL_PLASTICITY
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Material coordinates system definition
  #
  COORDINATE SYSTEM             = <string> coordinate_system_name
  DIRECTION FOR ROTATION        = <real> 1|2|3
  ALPHA                         = <real> α1 (degrees)
  SECOND DIRECTION FOR ROTATION = <real> 1|2|3
  SECOND ALPHA                  = <real> α2 (degrees)
  #
  # Yield surface parameters
  #
  YIELD STRESS = <real>  σy
  R11 = <real> R11 (1.0)
  R22 = <real> R22 (1.0)
  R33 = <real> R33 (1.0)
  R12 = <real> R12 (1.0)
  R23 = <real> R23 (1.0)
  R31 = <real> R31 (1.0)
  #
  # Hardening model
  #
  HARDENING MODEL = LINEAR | POWER_LAW | VOCE | USER_DEFINED |
    FLOW_STRESS | DECOUPLED_FLOW_STRESS | JOHNSON_COOK |
    POWER_LAW_BREAKDOWN
  #
  # Linear hardening
  #
  HARDENING MODULUS = <real> H'
  #
  # Power-law hardening
  #
  HARDENING CONSTANT = <real> A
  HARDENING EXPONENT = <real> n (0.5)
  LUDERS STRAIN      = <real> εL (0.0)
  #
```

```
# Voce hardening
#
HARDENING MODULUS       = <real> A
EXPONENTIAL COEFFICIENT = <real> n
#
# Johnson-Cook hardening
#
HARDENING FUNCTION = <string>hardening_function_name
RATE CONSTANT      = <real> C
REFERENCE RATE     = <real> ε̇0
#
# Power law breakdown hardening
#
HARDENING FUNCTION = <string>hardening_function_name
RATE COEFFICIENT   = <real> g
RATE EXPONENT      = <real> m
#
# User defined hardening
#
HARDENING FUNCTION = <string>hardening_function_name
#
#
# Following Commands Pertain to Flow_Stress Hardening Model
#
#    -  Isotropic Hardening model
#
ISOTROPIC HARDENING MODEL = LINEAR | POWER_LAW | VOCE |
                            USER_DEFINED
#
# Specifications for Linear, Power-law, and Voce same as above
#
# User defined hardening
#
ISOTROPIC HARDENING FUNCTION = <string>iso_hardening_fun_name
#
#     - Rate dependence
#
RATE MULTIPLIER = JOHNSON_COOK | POWER_LAW_BREAKDOWN |
   USER_DEFINED | RATE_INDEPENDENT (RATE_INDEPENDENT)
#
# Specifications for Johnson-Cook, Power-law-breakdown
#     same as before EXCEPT no need to specify a
#     hardening function
#
# User defined rate multiplier
#
RATE MULTIPLIER FUNCTION = <string> rate_mult_function_name
```

```
#
#     - Temperature dependence
#
TEMPERATURE MULTIPLIER = JOHNSON_COOK | USER_DEFINED |
   TEMPERATURE_INDEPENDENT (TEMPERATURE_INDEPENDENT)
#
#  Johnson-Cook temperature dependence
#
MELTING TEMPERATURE   = <real> θ_melt
REFERENCE TEMPERATURE = <real> θ_ref
TEMPERATURE EXPONENT  = <real> M
#
#  User-defined temperature dependence
TEMPERATURE MULTIPLIER FUNCTION = <string>temp_mult_function_name
#
#
# Following Commands Pertain to Decoupled_Flow_Stress Hardening Model
#
#    -  Isotropic Hardening model
#
ISOTROPIC HARDENING MODEL = LINEAR | POWER_LAW | VOCE | USER_DEFINED
#
# Specifications for Linear, Power-law, and Voce same as above
#
# User defined hardening
#
ISOTROPIC HARDENING FUNCTION = <string>isotropic_hardening_function_name
#
#     - Rate dependence
#
YIELD RATE MULTIPLIER = JOHNSON_COOK | POWER_LAW_BREAKDOWN |
            USER_DEFINED | RATE_INDEPENDENT (RATE_INDEPENDENT)
#
# Specifications for Johnson-Cook, Power-law-breakdown same as before
#     EXCEPT no need to specify a hardening function
#     AND should be preceded by YIELD
#
#     As an example for Johnson-Cook yield rate dependence,
#
YIELD RATE CONSTANT  = <real> C^y
YIELD REFERENCE RATE = <real> ε̇_0^y
#
# User defined rate multiplier
#
YIELD RATE MULTIPLIER FUNCTION = <string>yield_rate_mult_function_name
#
HARDENING_RATE MULTIPLIER = JOHNSON_COOK | POWER_LAW_BREAKDOWN |
```

```
              USER_DEFINED | RATE_INDEPENDENT (RATE_INDEPENDENT)
  #
  #     Syntax same as for yield parameters but with a HARDENING prefix
  #
  #     - Temperature dependence
  #
  YIELD TEMPERATURE MULTIPLIER = JOHNSON_COOK | USER_DEFINED |
           TEMPERATURE_INDEPENDENT (TEMPERATURE_INDEPENDENT)
  #
  #  Johnson-Cook temperature dependence
  #
  YIELD MELTING TEMPERATURE   = <real> θ^y_melt
  YIELD REFERENCE TEMPERATURE = <real> θ^y_ref
  YIELD TEMPERATURE EXPONENT  = <real> M^y
  #
  #  User-defined temperature dependence
  YIELD TEMPERATURE MULTIPLIER FUNCTION = <string>yield_temp_mult_fun_name
  #
  HARDENING TEMPERATURE MULTIPLIER = JOHNSON_COOK | USER_DEFINED |
           TEMPERATURE_INDEPENDENT (TEMPERATURE_INDEPENDENT)
  #
  #   Syntax for hardening constants same as for yield but
  #            with HARDENING prefix
#
  #
  # Optional Failure Definitions
  #    Following only need to be defined if intend to use failure model
  #
  FAILURE MODEL = TEARING_PARAMETER | JOHNSON_COOK_FAILURE | WILKINS
                  | MODULAR_FAILURE | MODULAR_BCJ_FAILURE

  CRITICAL FAILURE PARAMETER = <real> d_crit
  #
  # TEARING_PARAMETER Failure model definitions
  #
  TEARING PARAMETER EXPONENT  = m
  #
  # JOHNSON_COOK_FAILURE Failure model definitions
  #
  JOHNSON COOK D1       = <real> D1
  JOHNSON COOK D2       = <real> D2
  JOHNSON COOK D3       = <real> D3
  JOHNSON COOK D4       = <real> D4
  JOHNSON COOK D5       = <real> D5
  #
  #Following Johnson-Cook parameters can only be defined once.  As such, only
  #  needed if not previously defined via Johnson-Cook multipliers
```

```
#  w/ flow-stress hardening.  Does need to be defined
#  w/ Decoupled Flow Stress
#
REFERENCE RATE        = <real> ε̇0
REFERENCE TEMPERATURE = <real> T_ref
MELTING TEMPERATURE   = <real> T_melt
#
# WILKINS Failure model definitions
#
WILKINS ALPHA    = <real> α
WILKINS BETA     = <real> β
WILKINS PRESSURE = <real> B
#
#  MODULAR_FAILURE Failure model definitions
#
PRESSURE MULTIPLIER         = PRESSURE_INDEPENDENT | WILKINS
                              | USER_DEFINED (PRESSURE_INDEPENDENT)
LODE ANGLE MULTIPLIER       = LODE_ANGLE_INDEPENDENT |
                              WILKINS (LODE_ANGLE_INDEPENDENT)
TRIAXIALITY MULTIPLIER      = TRIAXIALITY_INDEPENDENT | JOHNSON_COOK
                              | USER_DEFINED (TRIAXIALITY_INDEPENDENT)
RATE FAIL MULTIPLIER        = RATE_INDEPENDENT | JOHNSON_COOK
                              | USER_DEFINED (RATE_INDEPENDENT)
TEMPERATURE FAIL MULTIPLIER = TEMPERATURE_INDEPENDENT | JOHNSON_COOK
                              | USER_DEFINED (TEMPERATURE_INDEPENDENT)
#
# Individual multiplier definitions
#
PRESSURE MULTIPLIER = WILKINS
WILKINS ALPHA       = <real> α
WILKINS PRESSURE    = <real> B
#
PRESSURE MULTIPLIER = USER_DEFINED
PRESSURE MULTIPLIER FUNCTION = <string> pressure_multiplier_fun_name
#
LODE ANGLE MULTIPLIER = WILKINS
WILKINS BETA          = <real> β
#
TRIAXIALITY MULTIPLIER = JOHNSON_COOK
JOHNSON COOK D1        = <real> D1
JOHNSON COOK D2        = <real> D2
JOHNSON COOK D3        = <real> D3
#
TRIAXIALITY MULTIPLIER = USER_DEFINED
TRIAXIALITY MULTIPLIER FUNCTION = <string> triaxiality_multiplier_fun_name
#
RATE FAIL MULTIPLIER = JOHNSON_COOK
```

```
JOHNSON COOK D4      = <real> D4
# REFERENCE RATE should only be added if not previously defined
REFERENCE RATE       = <real> ε̇0
#
RATE FAIL MULTIPLIER = USER_DEFINED
RATE FAIL MULTIPLIER FUNCTION = <string> rate_fail_multiplier_fun_name
#
TEMPERATURE FAIL MULTIPLIER = JOHNSON_COOK
JOHNSON COOK D5             = <real> D5
# JC Temperatures should only be defined if not previously given
REFERENCE TEMPERATURE       = <real> T_ref
MELTING TEMPERATURE         = <real> T_melt
#
TEMPERATURE FAIL MULTIPLIER          = USER_DEFINED
TEMPERATURE FAIL MULTIPLIER FUNCTION = <string> temp_multiplier_fun_name
#
# MODULAR BCJ_FAILURE Failure model definitions
#
INITIAL DAMAGE    = <real> φ_{t=0}
INITIAL VOID SIZE = <real> υ_υ 0
DAMAGE BETA       = <real> β (0.5)
GROWTH MODEL      = COCKS_ASHBY | NO_GROWTH (NO_GROWTH)
NUCLEATION MODEL  = HORSTEMEYER_GOKHALE | CHU_NEEDLEMAN_STRAIN
                    | NO_NUCLEATION (NO_NUCLEATION)
#
GROWTH RATE FAIL MULTIPLIER        = JOHNSON_COOK | USER_DEFINED
                                     | RATE_INDEPENDENT
                                     (RATE_INDEPENDENT)
GROWTH TEMPERATURE FAIL MULTIPLIER = JOHNSON_COOK | USER_DEFINED
                                     | TEMPERATURE_INDEPENDENT
                                     (TEMPERATURE_INDEPENDENT)
#
NUCLEATION RATE FAIL MULTIPLIER        = JOHNSON_COOK | USER_DEFINED
                                         | RATE_INDEPENDENT
                                         (RATE_INDEPENDENT)
NUCLEATION TEMPERATURE FAIL MULTIPLIER = JOHNSON_COOK | USER_DEFINED
                                         | TEMPERATURE_INDEPENDENT
                                         (TEMPERATURE_INDEPENDENT)
#
# Definitions for individual growth and nucleation models
#
GROWTH MODEL = COCKS_ASHBY
DAMAGE EXPONENT = <real> m (0.5)
#
NUCLEATION MODEL      = HORSTEMEYER_GOKHALE
NUCLEATION PARAMETER1 = <real> N1 (0.0)
NUCLEATION PARAMETER2 = <real> N2 (0.0)
```

```
  NUCLEATION PARAMETER3 = <real> N3 (0.0)
  #
  NUCLEATION MODEL          = CHU_NEEDLEMAN_STRAIN
  NUCLEATION AMPLITUDE      = <real> ψ
  MEAN NUCLEATION STRAIN    = <real> εN
  NUCLEATION STRAIN STD DEV = <real> s
  #
  # Definitions for rate and temperature fail multiplier
  #   Note: only showing definitions for growth.
  #    Nucleation terms are the same just with NUCLEATION instead
  #    of GROWTH
  #
  GROWTH RATE FAIL MULTIPLIER = JOHNSON_COOK
  GROWTH JOHNSON COOK D4      = <real> D4g
  GROWTH REFERENCE RATE       = <real> ε̇0g
  #
  GROWTH RATE FAIL MULTIPLIER          = USER_DEFINED
  GROWTH RATE FAIL MULTIPLIER FUNCTION = <string> growth_rate_fail_mult_func
  #
  GROWTH TEMPERATURE FAIL MULTIPLIER = JOHNSON_COOK
  GROWTH JOHNSON COOK D5             = <real> D5g
  GROWTH REFERENCE TEMPERATURE       = <real> Trefg
  GROWTH MELTING TEMPERATURE         = <real> Tmeltg
  #
  GROWTH TEMPERATURE FAIL MULTIPLIER          = USER_DEFINED
  GROWTH TEMPERATURE FAIL MULTIPLIER FUNCTION = <string> temp_fail_mult_func
  #
  #
  # Optional Adiabatic Heating/Thermal Softening  Definitions
  #    Following only need to be defined if intend to use failure model
  #
  THERMAL SOFTENING MODEL = ADIABATIC | COUPLED
  #
  SPECIFIC HEAT = <real> cp #not needed for COUPLED
  BETA_TQ       = <real> βTQ
  T0            = <real> T(t=0)
END [PARAMETERS FOR MODEL HILL_PLASTICITY]
```

The Hill plasticity model is similar to other plasticity models except that it is not isotropic. It is a hypoelastic, rate-independent plasticity model. The rate form of the equation assumes an additive split of the rate of deformation into an elastic and plastic part

$$D_{ij} = D^{\mathrm{e}}_{ij} + D^{\mathrm{p}}_{ij} \tag{5.51}$$

The stress rate only depends on the elastic rate of deformation

$$\overset{\circ}{\sigma}_{ij} = \mathbb{C}_{ijkl} D^{e}_{kl} \tag{5.52}$$

where $\mathbb{C}_{ijkl}$ are the components of the fourth-order, isotropic elasticity tensor.

The Hill plasticity model has an orthotropic yield surface that assumes orthogonal principal material directions. An example of this yield surface is presented below in Figure 5-8 along with examples of two isotropic surfaces – the von Mises (J_2) and Hosford (with $a = 8$). The various surface parameters correspond to 2090-T3 aluminum and the specific Hill strengths are found in [11]. By comparing the Hill surface to the two isotropic surfaces, the impact of the anisotropy is clear. Additionally, substantial differences to the normals of the yield surfaces at points of intersection highlight the impact of the yield function selection on the resulting flow directions.

Like other plasticity models, the Hill yield surface, f, is written,

$$f\left(\sigma_{ij}, \bar{\varepsilon}^{p}\right) = \phi\left(\sigma_{ij}\right) - \bar{\sigma}\left(\bar{\varepsilon}^{p}\right) = 0, \tag{5.53}$$

with ϕ being the effective stress and $\bar{\sigma}$ is the current yield stress that may be dependent on rate and/or temperature. The Hill effective stress is essentially an orthotropic extension of the von Mises function. After accounting for plastic incompressibility and related constraints, there are six individual yield stresses: σ^{y}_{11}, σ^{y}_{22}, σ^{y}_{33},τ^{y}_{12}, τ^{y}_{23}, and τ^{y}_{31}. These yield stresses correspond to 3 normal and 3 shear yield stresses. Written in terms of the components, the effective stress has the form,

$$\begin{aligned} \phi^2\left(\sigma_{ij}\right) \quad &= F(\hat{\sigma}_{22} - \hat{\sigma}_{33})^2 + G(\hat{\sigma}_{33} - \hat{\sigma}_{11})^2 + H(\hat{\sigma}_{11} - \hat{\sigma}_{22})^2 \\ &\quad + 2L\hat{\sigma}^2_{23} + 2M\hat{\sigma}^2_{31} + 2N\hat{\sigma}^2_{12}. \end{aligned} \tag{5.54}$$

The coefficients F, G, H, L, M, and N were introduced by Hill. In terms of the yield stresses they are:

$$\begin{aligned} F &= \frac{(\bar{\sigma})^2}{2}\left[\frac{1}{\left(\sigma^{y}_{22}\right)^2} + \frac{1}{\left(\sigma^{y}_{33}\right)^2} - \frac{1}{\left(\sigma^{y}_{11}\right)^2}\right] \quad ; \quad L = \frac{(\bar{\sigma})^2}{2}\left[\frac{1}{\left(\tau^{y}_{23}\right)^2}\right] \\ G &= \frac{(\bar{\sigma})^2}{2}\left[\frac{1}{\left(\sigma^{y}_{33}\right)^2} + \frac{1}{\left(\sigma^{y}_{11}\right)^2} - \frac{1}{\left(\sigma^{y}_{22}\right)^2}\right] \quad ; \quad M = \frac{(\bar{\sigma})^2}{2}\left[\frac{1}{\left(\tau^{y}_{31}\right)^2}\right] \\ H &= \frac{(\bar{\sigma})^2}{2}\left[\frac{1}{\left(\sigma^{y}_{11}\right)^2} + \frac{1}{\left(\sigma^{y}_{22}\right)^2} - \frac{1}{\left(\sigma^{y}_{33}\right)^2}\right] \quad ; \quad N = \frac{(\bar{\sigma})^2}{2}\left[\frac{1}{\left(\tau^{y}_{12}\right)^2}\right]. \end{aligned} \tag{5.55}$$

where $\bar{\sigma}$ is a reference yield stress.

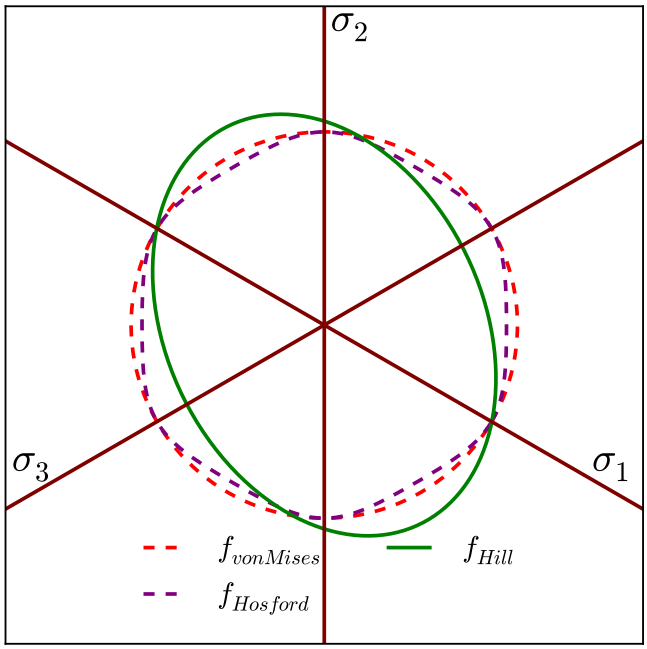


Figure 5-8. Example anisotropic Hill yield surface, $f_{Hill}\left(\sigma_{ij}, \bar{\varepsilon}^p = 0\right)$, presented in the deviatoric π-plane fit to 2090-T3 aluminum. Comparison von Mises (J_2) and Hosford (with $a = 8$) surfaces are also presented.

Rather than input the six independent yield stresses, the ratios of the yield stresses to some reference yield stress are generally used as input. These ratios are

$$R_{11} = \frac{\sigma^y_{11}}{\bar{\sigma}} \quad ; \quad R_{12} = \sqrt{3}\frac{\tau^y_{12}}{\bar{\sigma}}$$

$$R_{22} = \frac{\sigma^y_{22}}{\bar{\sigma}} \quad ; \quad R_{23} = \sqrt{3}\frac{\tau^y_{23}}{\bar{\sigma}} \tag{5.56}$$

$$R_{33} = \frac{\sigma^y_{33}}{\bar{\sigma}} \quad ; \quad R_{31} = \sqrt{3}\frac{\tau^y_{31}}{\bar{\sigma}}.$$

These ratios are set up so that if $R_{ij} = 1$ then the yield surface is isotropic.

The orientation of the principal material axes with respect to the global Cartesian axes may be specified by the user. First, a rectangular or cylindrical reference coordinate system is defined. *Spherical coordinate systems are not currently implemented for the Hill model.* The material coordinate system can then be defined through two successive rotations about axes in the reference rectangular or cylindrical coordinate system. In the case of the cylindrical coordinate system this allows the principal material axes to vary point-wise in a given element block. Refer to Section 5.1.7 for details on the definition of material coordinate systems.

The plastic rate of deformation, as with the isotropic models, assumes associated flow

$$D_{ij}^{\mathrm{p}} = \dot{\gamma}\frac{\partial \phi}{\partial \sigma_{ij}}. \tag{5.57}$$

Given the form for ϕ, the consistency parameter, $\dot{\gamma}$ is equal to the rate of the equivalent plastic strain, $\dot{\bar{\varepsilon}}^{p}$.

For more information about the Hill plasticity model, consult [15]. Additional discussion on options for failure models and adiabatic heating may be found in [12, 13] and [14], respectively.

In the command blocks that define the Hill plasticity model:

- See Section 5.1.5 for more information on elastic constants input.
- See Section 5.1.7 for more information on material coordinates system definition commands.
- The reference nominal yield stress, $\bar{\sigma}$, is defined with the `YIELD STRESS` command line.
- The ratio of the normal yield stress in the $\bar{\mathbf{e}}_1\bar{\mathbf{e}}_1$ material direction is defined with the `R11` command line. The default is 1.0.
- The ratio of the normal yield stress in the $\bar{\mathbf{e}}_2\bar{\mathbf{e}}_2$ material direction is defined with the `R22` command line. The default is 1.0.
- The ratio of the normal yield stress in the $\bar{\mathbf{e}}_3\bar{\mathbf{e}}_3$ material direction is defined with the `R33` command line. The default is 1.0.
- The ratio of the shear yield stress in the $\bar{\mathbf{e}}_1\bar{\mathbf{e}}_2$ material direction is defined with the `R12` command line. The default is 1.0.
- The ratio of the shear yield stress in the $\bar{\mathbf{e}}_2\bar{\mathbf{e}}_3$ material direction is defined with the `R23` command line. The default is 1.0.
- The ratio of the shear yield stress in the $\bar{\mathbf{e}}_3\bar{\mathbf{e}}_1$ material direction is defined with the `R31` command line. The default is 1.0.
- The type of hardening law is defined with the `HARDENING MODEL` command line, other hardening commands then define the specific shape of that hardening curve.
- The hardening modulus for a linear hardening model is defined with the `HARDENING MODULUS` command line.
- The hardening constant for a power law hardening model is defined with the `HARDENING CONSTANT` command line.
- The hardening exponent for a power law hardening model is defined with the `HARDENING EXPONENT` command line.
- The Lüders strain for a power law hardening model is defined with the `LUDERS STRAIN` command line.
- The hardening function for a user defined hardening model is defined with the `HARDENING FUNCTION` command line.

- The shape of the spline for the spline based hardening is defined by the `CUBIC SPLINE TYPE`, `CARDINAL PARAMETER`, `KNOT EQPS`, and `KNOT STRESS` command lines.

Output variables available for this model are listed in Table 5-12.

Table 5-12. State Variables for HILL PLASTICITY Model (Section 5.2.14)

Name	Description
`EQPS`	equivalent plastic strain, $\bar{\varepsilon}^p$
`EQDOT`	equivalent plastic strain rate, $\dot{\bar{\varepsilon}}^p$
`SEFF`	effective stress, ϕ
`TENSILE_EQPS`	tensile equivalent plastic strain, $\bar{\varepsilon}_t^p$
`DAMAGE`	damage, ϕ
`VOID_COUNT`	void count, η
`VOID_SIZE`	void size, υ
`DAMAGE_DOT`	damage rate, $\dot{\phi}$
`VOID_COUNT_DOT`	void count rate, $\dot{\eta}$
`PLASTIC_WORK_HEAT_RATE`	plastic work heat rate, $\dot{Q}^p$

5.2.15. Barlat Plasticity Model

```
BEGIN PARAMETERS FOR MODEL BARLAT_PLASTICITY
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Material coordinates system definition
  #
  COORDINATE SYSTEM             = <string> coordinate_system_name
  DIRECTION FOR ROTATION        = <real> 1|2|3
  ALPHA                         = <real> α1 (degrees)
  SECOND DIRECTION FOR ROTATION = <real> 1|2|3
  SECOND ALPHA                  = <real> α2 (degrees)
  #
  # Yield surface parameters
  #
  YIELD STRESS = <real> σy
  A            = <real> a (4.0)
  CP12         = <real> c'12  (1.0)
  CP13         = <real> c'13  (1.0)
  CP21         = <real> c'21  (1.0)
  CP23         = <real> c'23  (1.0)
  CP31         = <real> c'31  (1.0)
  CP32         = <real> c'32  (1.0)
  CP44         = <real> c'44  (1.0)
  CP55         = <real> c'55  (1.0)
  CP66         = <real> c'66  (1.0)
  CPP12        = <real> c''12  (1.0)
  CPP13        = <real> c''13  (1.0)
  CPP21        = <real> c''21  (1.0)
  CPP23        = <real> c''23  (1.0)
  CPP31        = <real> c''21  (1.0)
  CPP32        = <real> c''32  (1.0)
  CPP44        = <real> c''44  (1.0)
  CPP55        = <real> c''55  (1.0)
  CPP66        = <real> c''66  (1.0)
  #
  # Hardening model
  #
  HARDENING MODEL = LINEAR | POWER_LAW | VOCE | USER_DEFINED |
```

```
  FLOW_STRESS | DECOUPLED_FLOW_STRESS | JOHNSON_COOK |
  POWER_LAW_BREAKDOWN
#
# Linear hardening
#
HARDENING MODULUS = <real> H'
#
# Power-law hardening
#
HARDENING CONSTANT = <real> A
HARDENING EXPONENT = <real> n (0.5)
LUDERS STRAIN      = <real> ε_L (0.0)
#
# Voce hardening
#
HARDENING MODULUS       = <real> A
EXPONENTIAL COEFFICIENT = <real> n
#
# Johnson-Cook hardening
#
HARDENING FUNCTION = <string>hardening_function_name
RATE CONSTANT      = <real> C
REFERENCE RATE     = <real> ε̇_0
#
# Power law breakdown hardening
#
HARDENING FUNCTION = <string>hardening_function_name
RATE COEFFICIENT   = <real> g
RATE EXPONENT      = <real> m
#
# User defined hardening
#
HARDENING FUNCTION = <string>hardening_function_name
#
#
# Following Commands Pertain to Flow_Stress Hardening Model
#
#    -  Isotropic Hardening model
#
ISOTROPIC HARDENING MODEL = LINEAR | POWER_LAW | VOCE |
                            USER_DEFINED
#
# Specifications for Linear, Power-law, and Voce same as above
#
# User defined hardening
#
ISOTROPIC HARDENING FUNCTION = <string>iso_hardening_fun_name
```

```
#
#     - Rate dependence
#
RATE MULTIPLIER = JOHNSON_COOK | POWER_LAW_BREAKDOWN |
   USER_DEFINED | RATE_INDEPENDENT (RATE_INDEPENDENT)
#
# Specifications for Johnson-Cook, Power-law-breakdown
#     same as before EXCEPT no need to specify a
#     hardening function
#
# User defined rate multiplier
#
RATE MULTIPLIER FUNCTION = <string> rate_mult_function_name
#
#     - Temperature dependence
#
TEMPERATURE MULTIPLIER = JOHNSON_COOK | USER_DEFINED |
   TEMPERATURE_INDEPENDENT (TEMPERATURE_INDEPENDENT)
#
#  Johnson-Cook temperature dependence
#
MELTING TEMPERATURE   = <real> θ_melt
REFERENCE TEMPERATURE = <real> θ_ref
TEMPERATURE EXPONENT  = <real> M
#
#  User-defined temperature dependence
TEMPERATURE MULTIPLIER FUNCTION = <string>temp_mult_function_name
#
#
# Following Commands Pertain to Decoupled_Flow_Stress Hardening Model
#
#    -  Isotropic Hardening model
#
ISOTROPIC HARDENING MODEL = LINEAR | POWER_LAW | VOCE | USER_DEFINED
#
# Specifications for Linear, Power-law, and Voce same as above
#
# User defined hardening
#
ISOTROPIC HARDENING FUNCTION = <string>isotropic_hardening_function_name
#
#     - Rate dependence
#
YIELD RATE MULTIPLIER = JOHNSON_COOK | POWER_LAW_BREAKDOWN |
            USER_DEFINED | RATE_INDEPENDENT (RATE_INDEPENDENT)
#
# Specifications for Johnson-Cook, Power-law-breakdown same as before
```

```
  #     EXCEPT no need to specify a hardening function
  #     AND should be preceded by YIELD
  #
  #     As an example for Johnson-Cook yield rate dependence,
  #
  YIELD RATE CONSTANT  = <real> C^y
  YIELD REFERENCE RATE = <real> ε̇_0^y
  #
  # User defined rate multiplier
  #
  YIELD RATE MULTIPLIER FUNCTION = <string>yield_rate_mult_function_name
  #
  HARDENING_RATE MULTIPLIER = JOHNSON_COOK | POWER_LAW_BREAKDOWN |
              USER_DEFINED | RATE_INDEPENDENT (RATE_INDEPENDENT)
  #
  #     Syntax same as for yield parameters but with a HARDENING prefix
  #
  #     - Temperature dependence
  #
  YIELD TEMPERATURE MULTIPLIER = JOHNSON_COOK | USER_DEFINED |
            TEMPERATURE_INDEPENDENT (TEMPERATURE_INDEPENDENT)
  #
  #  Johnson-Cook temperature dependence
  #
  YIELD MELTING TEMPERATURE   = <real> θ_melt^y
  YIELD REFERENCE TEMPERATURE = <real> θ_ref^y
  YIELD TEMPERATURE EXPONENT  = <real> M^y
  #
  #  User-defined temperature dependence
  YIELD TEMPERATURE MULTIPLIER FUNCTION = <string>yield_temp_mult_fun_name
  #
  HARDENING TEMPERATURE MULTIPLIER = JOHNSON_COOK | USER_DEFINED |
            TEMPERATURE_INDEPENDENT (TEMPERATURE_INDEPENDENT)
  #
  #   Syntax for hardening constants same as for yield but
  #            with HARDENING prefix
#
  #
  # Optional Failure Definitions
  #    Following only need to be defined if intend to use failure model
  #
  FAILURE MODEL = TEARING_PARAMETER | JOHNSON_COOK_FAILURE | WILKINS
                | MODULAR_FAILURE | MODULAR_BCJ_FAILURE

  CRITICAL FAILURE PARAMETER = <real> d_crit
  #
  # TEARING_PARAMETER Failure model definitions
```

```
#
TEARING PARAMETER EXPONENT  = m
#
# JOHNSON_COOK_FAILURE Failure model definitions
#
JOHNSON COOK D1       = <real> D1
JOHNSON COOK D2       = <real> D2
JOHNSON COOK D3       = <real> D3
JOHNSON COOK D4       = <real> D4
JOHNSON COOK D5       = <real> D5
#
#Following Johnson-Cook parameters can only be defined once.  As such, only
#  needed if not previously defined via Johnson-Cook multipliers
#  w/ flow-stress hardening.  Does need to be defined
#  w/ Decoupled Flow Stress
#
REFERENCE RATE        = <real> ε̇0
REFERENCE TEMPERATURE = <real> T_ref
MELTING TEMPERATURE   = <real> T_melt
#
# WILKINS Failure model definitions
#
WILKINS ALPHA    = <real> α
WILKINS BETA     = <real> β
WILKINS PRESSURE = <real> B
#
#  MODULAR_FAILURE Failure model definitions
#
PRESSURE MULTIPLIER         = PRESSURE_INDEPENDENT | WILKINS
                              | USER_DEFINED (PRESSURE_INDEPENDENT)
LODE ANGLE MULTIPLIER       = LODE_ANGLE_INDEPENDENT |
                              WILKINS (LODE_ANGLE_INDEPENDENT)
TRIAXIALITY MULTIPLIER      = TRIAXIALITY_INDEPENDENT | JOHNSON_COOK
                              | USER_DEFINED (TRIAXIALITY_INDEPENDENT)
RATE FAIL MULTIPLIER        = RATE_INDEPENDENT | JOHNSON_COOK
                              | USER_DEFINED (RATE_INDEPENDENT)
TEMPERATURE FAIL MULTIPLIER = TEMPERATURE_INDEPENDENT | JOHNSON_COOK
                              | USER_DEFINED (TEMPERATURE_INDEPENDENT)
#
# Individual multiplier definitions
#
PRESSURE MULTIPLIER = WILKINS
WILKINS ALPHA       = <real> α
WILKINS PRESSURE    = <real> B
#
PRESSURE MULTIPLIER = USER_DEFINED
PRESSURE MULTIPLIER FUNCTION = <string> pressure_multiplier_fun_name
```

```
#
LODE ANGLE MULTIPLIER = WILKINS
WILKINS BETA          = <real> β
#
TRIAXIALITY MULTIPLIER = JOHNSON_COOK
JOHNSON COOK D1        = <real> D1
JOHNSON COOK D2        = <real> D2
JOHNSON COOK D3        = <real> D3
#
TRIAXIALITY MULTIPLIER = USER_DEFINED
TRIAXIALITY MULTIPLIER FUNCTION = <string> triaxiality_multiplier_fun_name
#
RATE FAIL MULTIPLIER = JOHNSON_COOK
JOHNSON COOK D4      = <real> D4
# REFERENCE RATE should only be added if not previously defined
REFERENCE RATE       = <real> ε̇0
#
RATE FAIL MULTIPLIER = USER_DEFINED
RATE FAIL MULTIPLIER FUNCTION = <string> rate_fail_multiplier_fun_name
#
TEMPERATURE FAIL MULTIPLIER = JOHNSON_COOK
JOHNSON COOK D5             = <real> D5
# JC Temperatures should only be defined if not previously given
REFERENCE TEMPERATURE       = <real> Tref
MELTING TEMPERATURE         = <real> Tmelt
#
TEMPERATURE FAIL MULTIPLIER          = USER_DEFINED
TEMPERATURE FAIL MULTIPLIER FUNCTION = <string> temp_multiplier_fun_name
#
# MODULAR BCJ_FAILURE Failure model definitions
#
INITIAL DAMAGE    = <real> φt=0
INITIAL VOID SIZE = <real> υυ 0
DAMAGE BETA       = <real> β (0.5)
GROWTH MODEL      = COCKS_ASHBY | NO_GROWTH (NO_GROWTH)
NUCLEATION MODEL  = HORSTEMEYER_GOKHALE | CHU_NEEDLEMAN_STRAIN
                    | NO_NUCLEATION (NO_NUCLEATION)
#
GROWTH RATE FAIL MULTIPLIER        = JOHNSON_COOK | USER_DEFINED
                                     | RATE_INDEPENDENT
                                     (RATE_INDEPENDENT)
GROWTH TEMPERATURE FAIL MULTIPLIER = JOHNSON_COOK | USER_DEFINED
                                     | TEMPERATURE_INDEPENDENT
                                     (TEMPERATURE_INDEPENDENT)
#
NUCLEATION RATE FAIL MULTIPLIER        = JOHNSON_COOK | USER_DEFINED
                                         | RATE_INDEPENDENT
```

```
                                          (RATE_INDEPENDENT)
NUCLEATION TEMPERATURE FAIL MULTIPLIER = JOHNSON_COOK | USER_DEFINED
                                          | TEMPERATURE_INDEPENDENT
                                          (TEMPERATURE_INDEPENDENT)
#
# Definitions for individual growth and nucleation models
#
GROWTH MODEL = COCKS_ASHBY
DAMAGE EXPONENT = <real> m (0.5)
#
NUCLEATION MODEL      = HORSTEMEYER_GOKHALE
NUCLEATION PARAMETER1 = <real> N_1 (0.0)
NUCLEATION PARAMETER2 = <real> N_2 (0.0)
NUCLEATION PARAMETER3 = <real> N_3 (0.0)
#
NUCLEATION MODEL          = CHU_NEEDLEMAN_STRAIN
NUCLEATION AMPLITUDE      = <real> ψ
MEAN NUCLEATION STRAIN    = <real> ε_N
NUCLEATION STRAIN STD DEV = <real> s
#
# Definitions for rate and temperature fail multiplier
#   Note: only showing definitions for growth.
#    Nucleation terms are the same just with NUCLEATION instead
#    of GROWTH
#
GROWTH RATE FAIL MULTIPLIER = JOHNSON_COOK
GROWTH JOHNSON COOK D4      = <real> D_4^g
GROWTH REFERENCE RATE       = <real> ε̇_0^g
#
GROWTH RATE FAIL MULTIPLIER          = USER_DEFINED
GROWTH RATE FAIL MULTIPLIER FUNCTION = <string> growth_rate_fail_mult_func
#
GROWTH TEMPERATURE FAIL MULTIPLIER = JOHNSON_COOK
GROWTH JOHNSON COOK D5             = <real> D_5^g
GROWTH REFERENCE TEMPERATURE       = <real> T_ref^g
GROWTH MELTING TEMPERATURE         = <real> T_melt^g
#
GROWTH TEMPERATURE FAIL MULTIPLIER          = USER_DEFINED
GROWTH TEMPERATURE FAIL MULTIPLIER FUNCTION = <string> temp_fail_mult_func
#
#
# Optional Adiabatic Heating/Thermal Softening  Definitions
#    Following only need to be defined if intend to use failure model
#
THERMAL SOFTENING MODEL = ADIABATIC | COUPLED
#
SPECIFIC HEAT = <real> c_p #not needed for COUPLED
```

```
  BETA_TQ        = <real> β^TQ
  T0             = <real> T_{t=0}
END [PARAMETERS FOR MODEL BARLAT_PLASTICITY]
```

The Barlat plasticity model is a hypoelastic, rate-independent plasticity model. The underlying yield surface is both anisotropic and non-quadratic [16]. With respect to the former, linear transformations of the deviatoric stress are used to capture texture and anisotropy effects. The rate form of this model assumes an additive split of the rate of deformation into an elastic and plastic part

$$D_{ij} = D^{\mathrm{e}}_{ij} + D^{\mathrm{p}}_{ij}. \tag{5.58}$$

The stress rate only depends on the elastic rate of deformation

$$\overset{\circ}{\sigma}_{ij} = \mathbb{C}_{ijkl} D^{\mathrm{e}}_{kl} \tag{5.59}$$

where $\mathbb{C}_{ijkl}$ are the components of the fourth-order, isotropic elasticity tensor.

To describe anisotropy in the yield-behavior, two linear transformation tensors, C'_{ijkl} and C''_{ijkl}, are introduced such that,

$$s'_{ij} = C'_{ijkl} s_{kl} \qquad ; \qquad s''_{ij} = C''_{ijkl} s_{kl}, \tag{5.60}$$

with s_{ij} being the deviatoric stress tensor ($s_{ij} = \sigma_{ij} - 1/3\sigma_{kk}\delta_{ij}$) and s'_{ij} and s''_{ij} being transformed stresses. Two transformations are used to capture both the anisotropy of the yield surface and flow rule. In Voigt notation the two transformation tensors are given as,

$$\left[C'\right] = \begin{bmatrix} 0 & -c'_{12} & -c'_{13} & 0 & 0 & 0 \\ -c'_{21} & 0 & -c'_{23} & 0 & 0 & 0 \\ -c'_{31} & -c'_{32} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & c'_{44} & 0 & 0 \\ 0 & 0 & 0 & 0 & c'_{55} & 0 \\ 0 & 0 & 0 & 0 & 0 & c'_{66} \end{bmatrix} \tag{5.61}$$

$$\left[C''\right] = \begin{bmatrix} 0 & -c''_{12} & -c''_{13} & 0 & 0 & 0 \\ -c''_{21} & 0 & -c''_{23} & 0 & 0 & 0 \\ -c''_{31} & -c''_{32} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & c''_{44} & 0 & 0 \\ 0 & 0 & 0 & 0 & c''_{55} & 0 \\ 0 & 0 & 0 & 0 & 0 & c''_{66} \end{bmatrix}. \tag{5.62}$$

Alternatively, the transformed stresses may be written in terms of the Cauchy stress tensor as,

$$s'_{ij} = L'_{ijkl}\sigma_{kl} \qquad ; \qquad s''_{ij} = L''_{ijkl}\sigma_{kl}, \tag{5.63}$$

where $L'_{ijkl} = C'_{ijmn}II_{mnkl}$ and $L''_{ijkl} = C''_{ijmn}II_{mnkl}$. In this case, II_{ijkl} is the symmetric deviatoric projection tensor and takes the form of,

$$II_{ijkl} = \frac{1}{2}\left(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}\right) - \frac{1}{3}\delta_{ij}\delta_{kl}. \tag{5.64}$$

In reduced form,

$$\left[L'\right] = \frac{1}{3}\begin{bmatrix} c'_{12}+c'_{13} & -2c'_{12}+c'_{13} & c'_{12}-2c'_{13} & 0 & 0 & 0 \\ -2c'_{21}+c'_{23} & c'_{21}+c'_{23} & c'_{21}-2c'_{23} & 0 & 0 & 0 \\ -2c'_{31}+c'_{32} & c'_{31}-2c'_{32} & c'_{31}+c'_{32} & 0 & 0 & 0 \\ 0 & 0 & 0 & 3c'_{44} & 0 & 0 \\ 0 & 0 & 0 & 0 & 3c'_{55} & 0 \\ 0 & 0 & 0 & 0 & 0 & 3c'_{66} \end{bmatrix}, \tag{5.65}$$

and an analogous expression may be written for L''_{ijkl}.

The yield surface, f, is given as,

$$f\left(\sigma_{ij}, \bar{\varepsilon}^p\right) = \phi\left(\sigma_{ij}\right) - \bar{\sigma}\left(\bar{\varepsilon}^p\right) = 0, \tag{5.66}$$

in which $\phi\left(\sigma_{ij}\right)$ is the effective stress and $\bar{\sigma}(\bar{\varepsilon}^p)$ is the current yield stress that may depend on rate and/or temperature. The effective stress is written in terms of the principal transformed stresses (s'_i and s''_i, respectively) and the yield surface exponent, a, as,

$$\begin{aligned} \phi\left(\sigma_{ij}\right) = \Bigg\{ \frac{1}{4}\Big[& |s'_1 - s''_1|^a + |s'_1 - s''_2|^a + |s'_1 - s''_3|^a \\ & + |s'_2 - s''_1|^a + |s'_2 - s''_2|^a + |s'_2 - s''_3|^a \\ & + |s'_3 - s''_1|^a + |s'_3 - s''_2|^a + |s'_3 - s''_3|^a \Big] \Bigg\}^{1/a}. \end{aligned} \tag{5.67}$$

An example of such a yield surface is given in Figure 5-9 along with examples of previously presented (von Mises, Hosford, Hill) surfaces. The presented Barlat surface corresponds to that of 2090-T3 aluminum first characterized by Barlat *et al.* [16]. In Figure 5-9, both the anisotropy and non-quadratic nature of the yield surface is evident leading to differing strengths and flow directions at various stresses from any of the other models.

The orientation of the principal material axes with respect to the global Cartesian axes may be specified by the user. First, a rectangular or cylindrical reference coordinate system is defined.

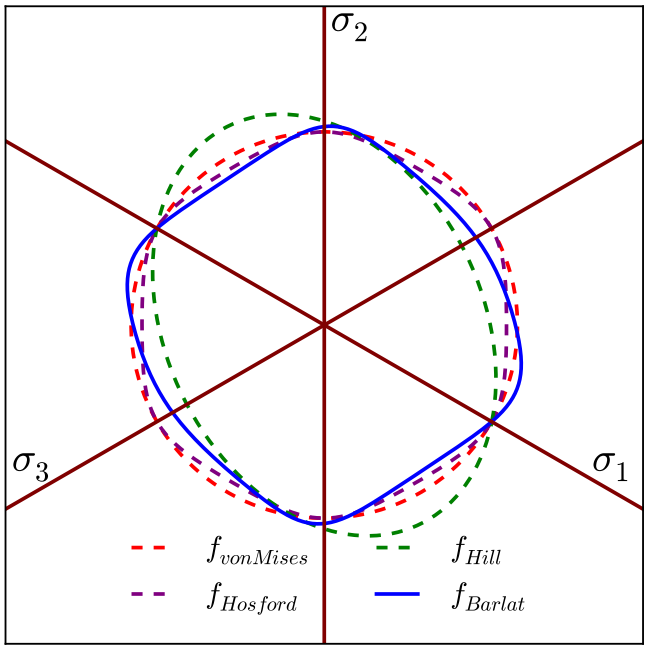


Figure 5-9. Example Barlat yield surface, $f_{Barlat}\left(\sigma_{ij}, \bar{\varepsilon}^p = 0\right)$, of 2090-T3 aluminum presented in the deviatoric π-plane. Comparison von Mises (J_2), Hosford (with $a = 8$), and Hill surfaces are also presented for comparison.

Spherical coordinate systems are not currently implemented for the Barlat model. The material coordinate system can then be defined through two successive rotations about axes in the reference rectangular or cylindrical coordinate system. In the case of the cylindrical coordinate system this allows the principal material axes to vary point-wise in a given element block. Refer to Section 5.1.7 for details on the definition of material coordinate systems.

The plastic rate of deformation, as with the isotropic models, assumes associated flow

$$D^{\mathrm{p}}_{ij} = \dot{\gamma}\frac{\partial \phi}{\partial \sigma_{ij}}, \tag{5.68}$$

in which $\dot{\gamma}$ is the consistency multiplier. Given the form for ϕ, $\dot{\gamma}$ is equal to the rate of the equivalent plastic strain, $\dot{\bar{\varepsilon}}^p$. As the yield surface is cast in transformed stress space, determining the flow direction in Cartesian space may be done via the chain rule (details may be found in [11]) leading to an expression of the form,

$$\frac{\partial \phi}{\partial \sigma_{ij}} = \sum_{k=1}^{3}\left(\frac{\partial \phi}{\partial s'_k}\frac{\partial s'_k}{\partial s'_{mn}}L'_{mnij} + \frac{\partial \phi}{\partial s''_k}\frac{\partial s''_k}{\partial s''_{mn}}L''_{mnij}\right). \tag{5.69}$$

For more information about the Barlat plasticity model, consult [16, 11]. Additional discussion on options for failure models and adiabatic heating may be found in [12, 13] and [14], respectively.

In the command blocks that define the Barlat plasticity model:

- See Section 5.1.5 for more information on elastic constants input.
- See Section 5.1.7 for more information on material coordinates system definition commands.
- The reference nominal yield stress, $\bar{\sigma}$, is defined with the `YIELD STRESS` command line.
- The exponent for the yield surface description, a, is defined with the `A` command line.
- The transformation coefficient, c'_{12}, is defined with the `CP12` command line. It is defaulted to 1.0.
- The transformation coefficient, c'_{13}, is defined with the `CP13` command line. It is defaulted to 1.0.
- The transformation coefficient, c'_{21}, is defined with the `CP21` command line. It is defaulted to 1.0.
- The transformation coefficient, c'_{23}, is defined with the `CP23` command line. It is defaulted to 1.0.
- The transformation coefficient, c'_{31}, is defined with the `CP31` command line. It is defaulted to 1.0.
- The transformation coefficient, c'_{32}, is defined with the `CP32` command line. It is defaulted to 1.0.
- The transformation coefficient, c'_{44}, is defined with the `CP44` command line. It is defaulted to 1.0.
- The transformation coefficient, c'_{55}, is defined with the `CP55` command line. It is defaulted to 1.0.
- The transformation coefficient, c'_{66}, is defined with the `CP66` command line. It is defaulted to 1.0.
- The transformation coefficient, c''_{12}, is defined with the `CPP12` command line. It is defaulted to 1.0.
- The transformation coefficient, c''_{13}, is defined with the `CPP13` command line. It is defaulted to 1.0.
- The transformation coefficient, c''_{21}, is defined with the `CPP21` command line. It is defaulted to 1.0.
- The transformation coefficient, c''_{23}, is defined with the `CPP23` command line. It is defaulted to 1.0.
- The transformation coefficient, c''_{31}, is defined with the `CPP31` command line. It is defaulted to 1.0.
- The transformation coefficient, c''_{32}, is defined with the `CPP32` command line. It is defaulted to 1.0.
- The transformation coefficient, c''_{44}, is defined with the `CPP44` command line. It is defaulted

to 1.0.

- The transformation coefficient, c''_{55}, is defined with the `CPP55` command line. It is defaulted to 1.0.
- The transformation coefficient, c''_{66}, is defined with the `CPP66` command line. It is defaulted to 1.0.
- The type of hardening law is defined with the `HARDENING MODEL` command line, other hardening commands then define the specific shape of that hardening curve.
- The hardening modulus for a linear hardening model is defined with the `HARDENING MODULUS` command line.
- The hardening constant for a power law hardening model is defined with the `HARDENING CONSTANT` command line.
- The hardening exponent for a power law hardening model is defined with the `HARDENING EXPONENT` command line.
- The Lüders strain for a power law hardening model is defined with the `LUDERS STRAIN` command line.
- The hardening function for a user defined hardening model is defined with the `HARDENING FUNCTION` command line.
- The shape of the spline for the spline based hardening is defined by the `CUBIC SPLINE TYPE`, `CARDINAL PARAMETER`, `KNOT EQPS`, and `KNOT STRESS` command lines.

Output variables available for this model are listed in Table 5-13.

Table 5-13. State Variables for BARLAT PLASTICITY Model (Section 5.2.15)

Name	Description
`EQPS`	equivalent plastic strain, $\bar{\varepsilon}^p$
`EQDOT`	equivalent plastic strain rate, $\dot{\bar{\varepsilon}}^p$
`SEFF`	effective stress, ϕ
`TENSILE_EQPS`	tensile equivalent plastic strain, $\bar{\varepsilon}_t^p$
`DAMAGE`	damage, ϕ
`VOID_COUNT`	void count, η
`VOID_SIZE`	void size, υ
`DAMAGE_DOT`	damage rate, $\dot{\phi}$
`VOID_COUNT_DOT`	void count rate, $\dot{\eta}$
`PLASTIC_WORK_HEAT_RATE`	plastic work heat rate, $\dot{Q}^p$

5.2.16. Johnson–Cook Model

```
BEGIN PARAMETERS FOR MODEL JOHNSON_COOK
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Yield surface parameters
  #
  YIELD STRESS = <real> σ_y
  HARDENING CONSTANT = <real>hardening_constant
  HARDENING EXPONENT = <real>hardening_exponent
  RHOCV = <real>rho_cv
  RATE CONSTANT = <real>rate_constant
  REFERENCE RATE = <real>reference_rate(0.001)
  D1 = <real>d1(0.0)
  D2 = <real>d2(0.0)
  D3 = <real>d3(0.0)
  D4 = <real>d4(0.0)
  D5 = <real>d5(0.0)
  EDOT_REF = <real>edot_ref(0.0)
  #
  # Temperature softening commands
  #
  BETA = <real>beta(0.95)
  THERMAL EXPONENT = <real>thermal_exponent
  REFERENCE TEMPERATURE = <real>reference_temperature
  MELT TEMPERATURE = <real>melt_temperature
  INITIAL TEMPERATURE = <real>init_temperature
  FORMULATION = <int>form(0)
END [PARAMETERS FOR MODEL JOHNSON_COOK]
```

The Johnson–Cook model is used to model materials—typically metals—undergoing plastic deformation at high rates and finite strains. The hardening function is rate- and temperature-dependent and is defined by

$$\sigma = \left[\sigma_y + B\left(\bar{\varepsilon}_p\right)^N\right]\left[1 + C\left\langle \ln \dot{\varepsilon}^* \right\rangle\right]\left[1 - T^{*M}\right], \tag{5.70}$$

where σ_y is the yield stress, B is the hardening constant, $\bar{\varepsilon}_p$ is the equivalent plastic strain, N is the hardening exponent, and C is the rate constant. The non-dimensional effective total strain rate $\dot{\varepsilon}^*$ is given by

$$\dot{\varepsilon}^* = \dot{\varepsilon}/\dot{\varepsilon}_0, \tag{5.71}$$

where $\dot{\varepsilon}_0$ is a reference strain rate defined by the input `EDOT_REF` material parameter. Note if `EDOT_REF` is undefined, then $\dot{\varepsilon}^*$ is set to the larger of either the maximum strain rate the material has encountered or the `REFERENCE RATE` input parameter. If `EDOT_REF` is defined, then the `REFERENCE RATE` input is not used by the model and has no effect. The Macaulay brackets in (5.70) ensure that σ is equal to the static flow stress $\sigma_s = \left[\sigma_y + B\left(\bar{\varepsilon}_p\right)^N\right]\left[1-\theta^{*M}\right]$ when $\dot{\varepsilon} < \dot{\varepsilon}_0$.

Parameter T^* is the homologous temperature and is defined as

$$T^* = (T - T_{ref})/(T_{melt} - T_{ref}), \tag{5.72}$$

where T is the current temperature, T_{ref} is the reference temperature, and T_{melt} is the melt temperature. In the case where $M \leq 0$, $(1-(T^*)^M) = 1$.

Note that the temperature used internal to the Johnson–Cook model is *not* the standard model "temperature" field. Instead, the material temperature is defined via the `INITIAL TEMPERATURE` command. Plastic work internal to the model results in adiabatic heating and raises the material temperature. The resulting change in temperature is computed according to

$$\Delta T = \frac{\beta}{\rho C_v}\sigma_y \Delta\bar{\varepsilon}_p, \tag{5.73}$$

where ρ is the material density, C_v is the specific heat, and β is the fraction of plastic work that is converted to heat, where $0 \leq \beta \leq 1$. By default, $\beta = 0.95$ though it may be set to a different value with the `BETA` command.

The Johnson–Cook model also encapsulates a failure criterion; the damage model [17] has a failure strain given by

$$\varepsilon^f = \left(D_1 + D_2 \exp\left(D_3\sigma^*\right)\right)\left(1 + D_4 \ln\dot{\varepsilon}^*\right)\left(1 + D_5 T^*\right) \tag{5.74}$$

where D_1, D_2, D_3, D_4, and D_5 are material parameters, σ^* is the triaxiality, $\sigma^* = \sigma_m/\bar{\sigma}$, $\dot{\varepsilon}^*$ is the dimensionless strain rate, and T^* is the homologous temperature, both defined above. Damage is accumulated over time:

$$D = \int_0^t \frac{\dot{\bar{\varepsilon}}^p}{\varepsilon^f} dt. \tag{5.75}$$

When $D = 1$, the material has failed. By default, fracture behavior is not active. For more information about the Johnson–Cook material model, consult [18, 17].

5.2.16.1. Command lines

Valid command lines for a command block defining a Johnson–Cook model are as follows:

- Elastic constants: Consult Section 5.1.5 for information on elastic constants input.
- `YIELD STRESS` defines σ_y, the stress for onset of yield and plasticity.
- `HARDENING CONSTANT` defines B.
- `HARDENING EXPONENT` defines N.
- `RHOCV` defines ρC_v, the volumetric heat capacity.

- `INITIAL TEMPERATURE` defines the material initial temperature. Note that the Johnson–Cook material model temperature is *not* linked to the standard model "temperature" field that is set from `BEGIN PRESCRIBED TEMPERATURE` command blocks.
- `THERMAL EXPONENT` defines the thermal exponent M.
- `REFERENCE TEMPERATURE` defines the reference temperature T_{ref}.
- `MELT TEMPERATURE` defines the melt temperature T_{melt}.
- `REFERENCE RATE` defines the reference strain rate, $\dot{\varepsilon}_0$. The default value is 0.001 s^{-1}.
- `BETA` defines the fraction of plastic work that is converted to heat, β. The default is 0.95.
- `D1`, `D2`, `D3`, `D4`, and `D5` define fracture coefficients D_1, D_2, D_3, D_4, and D_5, respectively. The default value for all coefficients is 0.0.
- `FORMULATION` controls the strain rate source term. A value of `0` triggers the use of the total strain rate, while `1` uses the plastic strain rate. The plastic strain rate may be stabler since it varies monotonically and at a much lower frequency than the total strain rate.

State variables for this model are listed in Table 5-14.

Table 5-14. State Variables for JOHNSON COOK Model (Section 5.2.16)

Name	Variable Description
`RADIUS`	radius of yield surface
`EQPS`	equivalent plastic strain
`THETA`	temperature
`EQDOT`	effective total strain rate
`ITER`	
`EFAIL`	failure strain, ε^f
`DAMAGE`	damage, D
`YIELD_STRESS`	yield stress



Warning: Strongly rate-dependent models may fare poorly in implicit quasistatic solution. In implicit analyses the rate term used to evaluate the current load step is the rate seen by the model in the previous load step. This may cause the solution to oscillate between high- and low-rate equilibrium states from step to step.

5.2.17. BCJ Model

```
BEGIN PARAMETERS FOR MODEL BCJ
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  #
  #
  C1 = <real>c1
  C2 = <real>c2
  C3 = <real>c3
  C4 = <real>c4
  C5 = <real>c5
  C6 = <real>c6
  C7 = <real>c7
  C8 = <real>c8
  C9 = <real>c9
  C10 = <real>c10
  C11 = <real>c11
  C12 = <real>c12
  C13 = <real>c13
  C14 = <real>c14
  C15 = <real>c15
  C16 = <real>c16
  C17 = <real>c17
  C18 = <real>c18
  C19 = <real>c19
  C20 = <real>c20
  DAMAGE EXPONENT = <real>damage_exponent
  INITIAL ALPHA_XX = <real>alpha_xx
  INITIAL ALPHA_YY = <real>alpha_yy
  INITIAL ALPHA_ZZ = <real>alpha_zz
  INITIAL ALPHA_XY = <real>alpha_xy
  INITIAL ALPHA_YZ = <real>alpha_yz
  INITIAL ALPHA_XZ = <real>alpha_xz
  INITIAL KAPPA = <real>initial_kappa
  INITIAL DAMAGE = <real>initial_damage
  YOUNGS MODULUS FUNCTION = <string>ym_function_name
  POISSONS RATIO FUNCTION = <string>pr_function_name
  SPECIFIC HEAT = <real>specific_heat
```

```
  THETA OPT = <integer>theta_opt
  FACTOR = <real>factor
  RHO    = <real>rho
  TEMP0  = <real>temp0
END [PARAMETERS FOR MODEL BCJ]
```

The BCJ plasticity model is a state-variable model for describing the finite deformation behavior of metals. It uses a multiplicative decomposition of the deformation gradient into elastic, volumetric plastic, and deviatoric parts. The model considers the natural configuration defined by this decomposition and its associated thermodynamics. The model incorporates strain rate and temperature sensitivity, and damage, through a yield-surface approach in which state variables follow a hardening-minus-recovery format.

Because the BCJ model has such an extensive list of parameters, we will not present the usual synopsis of parameter names with command lines. As with most other material models, the `thermal strain option` is used to define thermal strains. See Section 5.1.4.1 and Section 5.1.4.2 for further information on defining and activating thermal strains. In addition, only two of the five elastic constants are required. The user should consult [19, 20, 21] for a description of the various parameters. The parameters for the `SPECIFIC HEAT`, `THETA OPT`, `FACTOR`, `RHO`, and `TEMP0` command lines are used to accommodate changes to the model for heat generation due to plastic dissipation. For coupled solid/thermal calculations, the plastic dissipation rate is stored as a state variable and passed to a thermal code as a heat source term. For uncoupled calculations, temperature is stored as a state variable, and temperature increases due to plastic dissipation are calculated within the material model.

If temperature is provided from an external source, `theta_opt` is set to 0. If the temperature is calculated by the BCJ model, `theta_opt` is set to 1.

Output variables available for this model are listed in Table 5-15.



Warning: Strongly rate-dependent models may fare poorly in implicit quasistatic solution. In implicit analyses the rate term used to evaluate the current load step is the rate seen by the model in the previous load step. This may cause the solution to oscillate between high- and low-rate equilibrium states from step to step.

Table 5-15. State Variables for BCJ Model

Name	Variable Description
BACK_STRESS_XX	back stress tensor - xx component
BACK_STRESS_YY	back stress tensor - yy component
BACK_STRESS_ZZ	back stress tensor - zz component
BACK_STRESS_XY	back stress tensor - xy component
BACK_STRESS_YZ	back stress tensor - yz component
BACK_STRESS_ZX	back stress tensor - zx component
KAPPA	hardening scalar
DAMAGE	damage term
DAMAGE_RATE	rate of change of damage term
EQPS	equivalent plastic strain
THETA	temperature for adiabatic heating
HEAT	rate of heating due to plastic dissipation
YM	Young's modulus
PR	Poisson's ratio

5.2.18. BCJ_MEM Model

```
BEGIN PARAMETERS FOR MODEL BCJ_MEM
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ

  YOUNGS MODULUS FUNCTION = <string> youngs_modulus_scale_factor
  POISSONS RATIO FUNCTION = <string> poissons_ratio_scale_factor
  #
  # Initial yield as function of temperature
  #
  RATE INDEPENDENT YIELD SHEAR COEFFICIENT       = <real> Y_μ
  RATE INDEPENDENT YIELD CONSTANT                = <real> Y_0
  RATE INDEPENDENT YIELD TEMPERATURE DEPENDENCE  = <real> Y_1
  RATE INDEPENDENT YIELD TEMPERATURE DEPENDENCE2 = <real> Y_2
  RATE INDEPENDENT YIELD TEMPERATURE DEPENDENCE3 = <real> Y_3
  RATE INDEPENDENT YIELD TEMPERATURE DEPENDENCE4 = <real> Y_4
  #
  # Flow rule rate dependence
  #
  FLOW RULE COEFFICIENT CONSTANT               = <real> f_1
  FLOW RULE COEFFICIENT TEMPERATURE DEPENDENCE = <real> f_2
  FLOW RULE EXPONENT CONSTANT                  = <real> n_1
  FLOW RULE EXPONENT TEMPERATURE DEPENDENCE    = <real> n_2
  #
  # Isotropic hardening
  #
  ISOTROPIC HARDENING SHEAR COEFFICIENT                = <real> H_μ
  ISOTROPIC HARDENING CONSTANT                         = <real> H_1
  ISOTROPIC HARDENING TEMPERATURE DEPENDENCE           = <real> H_2
  ISOTROPIC DYNAMIC RECOVERY CONSTANT                  = <real> R_d1
  ISOTROPIC DYNAMIC RECOVERY TEMPERATURE DEPENDENCE    = <real> R_d2
  ISOTROPIC STATIC RECOVERY CONSTANT                   = <real> R_s1
  ISOTROPIC STATIC RECOVERY TEMPERATURE DEPENDENCE     = <real> R_s2
  ISOTROPIC STATIC RECOVERY SINH CONSTANT              = <real> Q_1
  ISOTROPIC STATIC RECOVERY SINH TEMPERATURE DEPENDENCE = <real> Q_2
  #
  # Kinematic hardening
  #
  BACK STRESS HARDENING CONSTANT = <real> h_1
```

```
BACK STRESS HARDENING TEMPERATURE DEPENDENCE = <real> h_2
BACK STRESS DYNAMIC RECOVERY CONSTANT = <real> r_d1
BACK STRESS DYNAMIC RECOVERY TEMPERATURE DEPENDENCE =
  <real> r_d2
#
# heat due to plastic dissipation
TEMPERATURE OPTION = <real>
PLASTIC DISSIPATION FACTOR = <real> β
DENSITY FOR PLASTIC DISSIPATION CALCULATIONS = <real> ρ
SPECIFIC HEAT FOR PLASTIC DISSIPATION CALCULATIONS =
  <real> c_p
INITIAL TEMPERATURE FOR UNCOUPLED ADIABATIC HEATING =
  <real> θ_o
#
# damage evolution
DAMAGE EXPONENT                                  = <real> m
VOLUMETRIC EFFECTS OF DAMAGE OPTION              = <real>
PRESSURE INTEGRATION OPTION                      = <real>
IMPLICIT DAMAGE SOLVER NUMBER OF ITERATIONS      = <real>
IMPLICIT DAMAGE SOLVER RESIDUAL TOLERANCE        = <real>
MAXIMUM CHANGE IN DAMAGE ALLOWED PER TIMESTEP = <real>
#
# misorientation variable
MISORIENTATION VARIABLE HARDENING CONSTANT = <real> h_ζ
MISORIENTATION VARIABLE HARDENING EXPONENT = <real> r
#
# integration options
SEMI IMPLICIT PLASTIC STRAIN SOLVER NUMBER OF ITERATIONS =
  <real>
SEMI IMPLICIT PLASTIC STRAIN SOLVER RESIDUAL TOLERANCE =
  <real>
#
# initial values for state variables
#
INITIAL KAPPA    = <real> κ_o
INITIAL DAMAGE   = <real> φ_o
INITIAL ZETA     = <real> ζ_o
INITIAL ALPHA_XX = <real> α_xx_o
INITIAL ALPHA_XY = <real> α_xy_o
INITIAL ALPHA_XZ = <real> α_xz_o
INITIAL ALPHA_YY = <real> α_yy_o
INITIAL ALPHA_YZ = <real> α_yz_o
INITIAL ALPHA_ZZ = <real> α_zz_o
#
#  Melting related variables
#  Optional temperature at which material turns into a fluid and
#  basic properties of that fluid
```

```
  #
  MELT TEMPERATURE = <real>melt_temp
  FLUID VISCOSITY  = <real>fluid_viscosity
  FLUID BULK       = <real>fluid_bulk
END [PARAMETERS FOR MODEL BCJ_MEM]
```

The BCJ_MEM material model, like BCJ, is a rate and temperature-dependent elasto-viscoplasticity model with isotropic damage. The two models have slightly different forms for the flow rule and static recovery expressions. In addition, the BCJ_MEM model includes the effects of recrystallization and grain growth [22]. Currently, the temperature must be prescribed in the input deck for these models to work. This will be changed in the future.

In its full form, the BCJ_MEM model has considerable complexity, but the material parameters in the list above are all optional, with the exception of two elastic constants and a yield strength. For example, in its simplest form, BCJ_MEM can be used to model elastic, perfectly-plastic materials by specifying the following:

```
BEGIN PARAMETERS FOR MODEL BCJ_MEM
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  RATE INDEPENDENT YIELD CONSTANT = <real> Y
END [PARAMETERS FOR MODEL BCJ_MEM]
```

To model an elastic-plastic material with linear hardening, the line

```
  ISOTROPIC HARDENING CONSTANT = <real> H
```

would need to be added to specify the hardening rate. The more general form includes rate and temperature effects in the yield strength. BCJ_MEM is a state variable model that tracks history dependence through the use of internal state variables. Thus if the temperature and rate are not constant during loading, the yield stress is not a simple function of rate, temperature, and plastic strain. For the simplified case of uniaxial tension with no kinematic hardening, the following equations must be integrated to determine the current yield stress:

$$\dot{\sigma} = E(\dot{\epsilon} - \dot{\epsilon}_p), \tag{5.76}$$

$$\dot{\epsilon}_p = f(\theta)\sinh^{n(\theta)}\left\langle \frac{\sigma}{\kappa + Y(\theta)} - 1 \right\rangle, \tag{5.77}$$

$$\dot{\kappa} = \kappa\frac{\dot{\mu}}{\mu} + [H(\theta) - R_d(\theta)\kappa]\,\dot{\epsilon}_p - R_s(\theta)\kappa\sinh\left(Q(\theta)\kappa\right), \tag{5.78}$$

where θ is the temperature and κ represents the isotropic hardening variable, which evolves according to a hardening minus dynamic recovery form originally proposed by Kocks and Mecking [23], with a static recovery term developed by Nes [24]. The flow rule can be inverted to obtain the yield stress:

$$\sigma_y = [Y + \kappa]\left\{1 + \sinh^{-1}\left[\left(\frac{\dot{\epsilon}^p}{f}\right)^{1/n}\right]\right\}. \tag{5.79}$$

To illustrate the evolution of the yield stress for the simple case of constant strain rate, constant temperature loading with no stage IV hardening and no static recovery, the equation for the isotropic hardening variable reduces to

$$\dot{\kappa} = [H(\theta) - R_d(\theta)\kappa]\,\dot{\epsilon}_p \tag{5.80}$$

which can be integrated analytically; the flow rule can be inverted to get the following relation for yield strength:

$$\sigma_y = \left[Y + \frac{H}{R_d}\left(1 - e^{-R_d \epsilon_p}\right)\right]\left\{1 + \sinh^{-1}\left[\left(\frac{\dot{\epsilon}^p}{f}\right)^{1/n}\right]\right\} \tag{5.81}$$

The expression in brackets represents the sum of the rate-independent yield strength and the hardening due to plastic strain; the expression in braces provides the rate dependence of the yield strength. This equation can be compared to Equation (5.70) of the Johnson Cook model.

The temperature dependence of the initial yield strength is of the form

$$Y(\theta) = \frac{Y_o}{Y_4 + e^{-Y_1/\theta}}\frac{1}{2}\left[1 + \tanh\left(Y_2\left(Y_3 - \theta\right)\right)\right], \tag{5.82}$$

where Y_i are material constants.

The yield can also be specified as a simple constant times the shear modulus,

$$Y(\theta) = Y_\mu \mu(\theta) \tag{5.83}$$

where the shear modulus can depend on temperature through user-defined scale factors:

```
YOUNGS MODULUS FUNCTION = <string> youngs_modulus_scale_factor
POISSONS RATIO FUNCTION = <string> poissons_ratio_scale_factor
```

Similarly, the temperature dependence of the hardening parameter in the evolution equation for the isotropic hardening variable can be specified in one of two ways:

$$H(\theta) = H_1 - H_2\theta \tag{5.84}$$

or

$$H(\theta) = H_\mu \mu(\theta) \tag{5.85}$$

The other coefficients in the evolution equation for the isotropic hardening variable have an Arrhenius-type temperature dependence, as shown here for the dynamic recovery parameter:

$$R_d(\theta) = R_{d1} e^{-R_{d2}/\theta} \tag{5.86}$$

Rate effects in the flow rule: The flow rule coefficient has an Arrhenius temperature dependence:

$$f(\theta) = f_1 e^{-f_2/\theta}. \tag{5.87}$$

The exponent in the flow rule has the following temperature dependence:

$$n(\theta) = n_1 + \frac{n_2}{\theta}. \tag{5.88}$$

The rate dependence of the model can make the system of equations stiff. The default integration for plastic strain uses the radial return method. A semi-implicit integration scheme that is implicit in plastic strain can be selected by setting the
`SEMI IMPLICIT PLASTIC STRAIN SOLVER NUMBER OF ITERATIONS` to whatever value is desired. If the residual does not drop below the tolerance specified by
`SEMI IMPLICIT PLASTIC STRAIN SOLVER RESIDUAL TOLERANCE`, then the radial return value for plastic strain is used instead, and a message is sent to the output file to this effect. It is recommended to always use the semi-implicit plastic strain solver when running implicit analyses since it will improve convergence. For explicit, the errors incurred by not doing the semi-implicit integration are not significant enough to warrant the time cost of implicit iterations.

Stage IV hardening: Stage IV hardening is modeled through the misorientation variable, , which loosely represents the geometrically necessary dislocation density and evolves according to the relation proposed by Kok et al. [25]. The misorientation variable, when used, adds an additional hardening term to the evolution equation for the isotropic hardening variable:

$$\dot{\kappa} = \kappa\frac{\dot{\mu}}{\mu} + \left[H(\theta)\left(1 + \frac{\zeta}{\kappa}\right) - R_d(\theta)\kappa\right]\dot{\epsilon}_p - R_s(\theta)\kappa\sinh\left(Q(\theta)\kappa\right) \tag{5.89}$$

The parameters in the evolution equation for the misorientation variable are independent of temperature, since they represent geometrically necessary dislocations:

$$\frac{d}{dt}\left(\frac{\zeta}{\mu}\right) = h_\zeta\left(\frac{\zeta}{\mu}\right)^{1-1/r}\dot{\epsilon}_p \tag{5.90}$$

Heat generation due to plastic dissipation: Heat generation due to plastic dissipation can be modeled in BCJ_MEM in two ways: for coupled solid/thermal calculations
(`TEMPERATURE OPTION = 0`), the plastic dissipation rate is stored as a state variable (AHEAT) and passed to a thermal code as a heat source term; for uncoupled calculations
(`TEMPERATURE OPTION = 1`), temperature is stored as a state variable (THETA), and temperature evolution due to adiabatic heating is calculated within the material model. The heat generated, AHEAT, is calculated as $\beta\sigma\dot{\epsilon}_p$. For uncoupled adiabatic heating, the temperature, THETA, evolves as

$$\dot{\theta} = \frac{\beta}{\rho c_p}\sigma\dot{\epsilon}_p \tag{5.91}$$

where β is the fraction of plastic work that is dissipated as heat, ρ is the density, and c_p is the specific heat.

Isotropic damage : Isotropic damage is included in the BCJ_MEM through a void growth equation proposed by Cocks and Ashby [26]:

$$\dot{\phi} = \sqrt{\frac{2}{3}}\dot{\epsilon}_p \frac{1-(1-\phi)^{m+1}}{(1-\phi)^m} \sinh\left[\frac{2(2m-1)}{2m+1}\frac{< p >}{\sigma_{vm}}\right], \tag{5.92}$$

where σ_{vm} is the von Mises stress, p is the hydrostatic stress, "$< \cdot >$" are Macaulay brackets, and m is the `DAMAGE EXPONENT`. When damage is included in the model, the yield stress decreases with damage according to

$$\sigma_y = (1-\phi)(Y+\kappa)\left\{1+\sinh^{-1}\left[\left(\frac{\dot{\epsilon}^p}{f}\right)^{1/n}\right]\right\} \tag{5.93}$$

We assume Young's modulus and the shear modulus decrease as damage increases, according to $E(\theta,\phi) = E(\theta)(1-\phi)$ and $\mu(\theta,\phi) = \mu(\theta)(1-\phi)$. It then follows that the bulk modulus must vary in the same way:

$$K(\theta,\phi) = k(\theta)(1-\phi) \tag{5.94}$$

Due to the effects of damage, the pressure evolution equation becomes:

$$\dot{p} = \left[\frac{\dot{k}(\theta)}{k(\theta)} - \frac{\dot{\phi}}{1-\phi}\right] p + 3D_{ave}k(\theta)(1-\phi) - k(\theta)\dot{\phi} \tag{5.95}$$

In this equation, the last term is due to volumetric expansion of the material as voids grow. This term can lead to some non-physical responses under certain loading conditions, so its inclusion is optional in the BCJ_MEM material model and is controlled by the `VOLUMETRIC EFFECTS OF DAMAGE OPTION` parameter. If this optional parameter is 0 (default), then the volumetric term is not included; if the value is 1, then the volumetric term is active. The integration of the pressure equation can be selected by setting the value of `PRESSURE INTEGRATION OPTION` to either 0 for implicit (Backward Euler) or 1 for implicit midpoint method.

The integration algorithm for damage solves the damage equation analytically using the midpoint value for triaxiality. The default value for `IMPLICIT DAMAGE SOLVER NUMBER OF ITERATIONS` is 50, but can be set to whatever value is desired. If the residual does not drop below the tolerance specified by `IMPLICIT DAMAGE SOLVER RESIDUAL TOLERANCE`, then the value for damage from the last iteration attempt is used instead, and a message is sent to the output file to this effect.

Modeling recrystallization: Recrystallization is a process by which the microstructure in a worked material is softened by the nucleation and growth of a new set of grains with relatively low dislocation density. The method used by BCJ_MEM to model recrystallization is fully documented in Reference [22], so it will not be repeated here. Additional output variables when modeling recrystallization are listed in Table 5-17

Kinematic hardening: The yield surface in stress space can either grow (or contract) through the isotropic hardening variable or translate through evolution of the back stress, or kinematic hardening, variable. The back stress evolves according to the equation

$$\mathring{\alpha} = h(\theta)d^p - r_d(\theta)\dot{\epsilon}^p \sqrt{\frac{3}{2}}\|\alpha\|\alpha, \tag{5.96}$$

where

$$h(\theta) = h_1 - h_2\theta, \tag{5.97}$$

$$r_d(\theta) = r_{d1}e^{-r_{d2}/\theta}. \tag{5.98}$$

When kinematic hardening is desired, the hardening and recovery must be specified by the parameters h_1, h_2, r_{d1}, and r_{d2}.

State variable initialization: The initial values for the internal state variables are specified through the input values for κ_o, ϕ_o, ζ_o, and α_{ij_o}. The default values for the initial values of κ_o, ζ_o and X_o are 1.0e-6 to avoid singularities in the evolution equations.

Output variables available for this model are listed in Tables 5-16 and 5-17.

Table 5-16. State Variables for BCJ_MEM Model

Name	Variable Description
BACK_STRESS_XX	back stress tensor - xx component
BACK_STRESS_YY	back stress tensor - yy component
BACK_STRESS_ZZ	back stress tensor - zz component
BACK_STRESS_XY	back stress tensor - xy component
BACK_STRESS_YZ	back stress tensor - yz component
BACK_STRESS_ZX	back stress tensor - zx component
KAPPA	hardening scalar
DAMAGE	void volume fraction
DAMAGE_RATE	rate of change of void volume fraction
EQPS	equivalent plastic strain
THETA	temperature for adiabatic heating
AHEAT	plastic dissipation rate
EQPSDOT	plastic strain rate
SIGY_RT	room temperature yield stress at quasistatic conditions
IBOND	contact variable to switch from sliding to glued contact based on recrystallized volume fraction

Further questions about the model, or requests for changes/enhancements, can be sent to aabrown@sandia.gov.

Table 5-17. Additional State Variables for BCJ_MEM Model When Modeling Recrystallization

Name	Variable Description
GRAIN_SIZE	grain size
REX_CYCLE_A	recrystallized volume fraction for cycle A
REX_CYCLE_B	recrystallized volume fraction for cycle B
REX_CYCLE_C	recrystallized volume fraction for cycle C
REX_CYCLE_D	recrystallized volume fraction for cycle D
BACK_STRESS_NO_REX_XX	xx component of back stress in un-recrystallized volume fraction
BACK_STRESS_NO_REX_YY	yy component of back stress in un-recrystallized volume fraction
BACK_STRESS_NO_REX_ZZ	zz component of back stress in un-recrystallized volume fraction
BACK_STRESS_NO_REX_XY	xy component of back stress in un-recrystallized volume fraction
BACK_STRESS_NO_REX_YZ	yz component of back stress in un-recrystallized volume fraction
BACK_STRESS_NO_REX_ZX	zx component of back stress in un-recrystallized volume fraction
SKAPP_NO_REX	isotropic hardening in the un-recrystallized volume fraction
SKAPP_CYCLE_A	isotropic hardening in the recrystallized volume fraction for cycle A
SKAPP_CYCLE_B	isotropic hardening in the recrystallized volume fraction for cycle B
SKAPP_CYCLE_C	isotropic hardening in the recrystallized volume fraction for cycle C
SKAPP_CYCLE_D	isotropic hardening in the recrystallized volume fraction for cycle D
ZETA_NO_REX	misorientation in the un-recrystallized volume fraction
ZETA_CYCLE_A	misorientation in the un-recrystallized volume fraction for cycle A
ZETA_CYCLE_B	misorientation in the un-recrystallized volume fraction for cycle B
ZETA_CYCLE_C	misorientation in the un-recrystallized volume fraction for cycle C
ZETA_CYCLE_D	misorientation in the un-recrystallized volume fraction for cycle D



Warning: Strongly rate-dependent models may fare poorly in implicit quasistatic solution. In implicit analyses the rate term used to evaluate the current load step is the rate seen by the model in the previous load step. This may cause the solution to oscillate between high- and low-rate equilibrium states from step to step.

5.2.19. Power Law Creep Model

```
BEGIN PARAMETERS FOR MODEL POWER_LAW_CREEP
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Viscoplastic parameters
  #
  CREEP CONSTANT    = <real> A
  CREEP EXPONENT    = <real> m
  THERMAL CONSTANT  = <real> Q/R
  MAX SUBINCREMENTS = <integer> max_subincrements(100)
END [PARAMETERS FOR MODEL POWER_LAW_CREEP]
```

The power law creep model describes the secondary (or steady-state) creep and is useful in capturing the time-dependent behavior of metals, brazes, or solder at high homologous temperatures. It may also be used as a simple model for the time-dependent behavior of geologic materials such as salt. A general discussion of such creep behaviors and the associated modeling may be found in the texts of [27, 28] while the specific implementation used here is discussed in [5].

In the power law creep model, the effective creep strain rate is taken to be explicitly a function of stress and temperature. A power law relation is used for the stress dependence while an Arrhenius like expression is used to capture thermal effects. As such, the effective creep strain rate is written as,

$$\dot{\bar{\varepsilon}}^{c} = A\bar{\sigma}_{vM}^{m} \exp\left(\frac{-Q}{R\theta}\right), \tag{5.99}$$

where $\dot{\bar{\varepsilon}}^{c}$ is the effective creep strain rate, $\bar{\sigma}_{vM}$ is the von Mises stress, A is the creep constant, m is the creep exponent, Q is the activation energy, R is the universal gas constant (1.987 cal/mole K), and θ is the absolute temperature. As a slip based mechanism, it is assumed that the creep strains are deviatoric leading to a 3D evolution law of the form,

$$D_{ij}^{c} = \dot{\bar{\varepsilon}}^{c} \frac{3}{2} \frac{s_{ij}}{\bar{\sigma}_{vM}}, \tag{5.100}$$

with s_{ij} being the deviatoric stress. The corresponding incremental constitutive equation for this model is then given as,

$$\overset{\circ}{\sigma}_{ij} = \mathbb{C}_{ijkl}\left(D_{kl} - D^{c}_{kl}\right). \tag{5.101}$$

In the above command blocks:

- See Section 5.1.5 for more information on elastic constants input.
- The creep constant, A, in Equation (5.99) is defined with the `CREEP CONSTANT` command line.
- The creep exponent, m, in Equation (5.99) is defined with the `CREEP EXPONENT` command line.
- The thermal constant, Q/R in Equation (5.99) is defined with the `THERMAL CONSTANT` command line.
- time step sub incrementation within the material model may be used to accurately calculate the true creep stress. The maximum sub-increments in a load step is defined with the `MAX SUBINCREMENTS` command line. The default is 100. A larger number of steps can potentially improve accuracy if a large amount of creep happens in a single step. A smaller number of steps can sometimes improve analysis speed.

Output variables available for this model are listed in Table 5-18.

Table 5-18. State Variables for POWER LAW CREEP Model (Section 5.2.19)

Name	Description
ECREEP	equivalent creep strain
SEQDOT	equivalent stress rate

5.2.20. Viscoplastic Model

```
BEGIN PARAMETERS FOR MODEL VISCOPLASTIC
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ

  FLOW RATE     = <real> g0
  SINH EXPONENT = <real> p0
  ALPHA         = <real> α0
  ISO HARDENING = <real> A1^0
  ISO RECOVERY  = <real> A2^0
  ISO EXPONENT  = <real> A3
  KIN HARDENING = <real> A4^0
  KIN RECOVERY  = <real> A5^0
  KIN EXPONENT  = <real> A6
  FLOW STRESS   = <real> D0

  SHEAR FUNCTION    = <string> hG(θ)
  BULK FUNCTION     = <string> hK(θ)
  RATE FUNCTION     = <string> hg(θ)
  EXPONENT FUNCTION = <string> hp(θ)
  ALPHA FUNCTION    = <string> hα(θ)
  IHARD FUNCTION    = <string> h1(θ)
  IREC FUNCTION     = <string> h2(θ)
  KHARD FUNCTION    = <string> h4(θ)
  KREC FUNCTION     = <string> h5(θ)
  MAX SUBINCREMENTS = <int> itmax (2000)
END [PARAMETERS FOR MODEL VISCOPLASTIC]
```

The viscoplastic model is a rate dependent plasticity model that is useful for modeling solders and brazes and was developed by Neilsen et al. [29]. This model is formulated in terms of the stress rate for the material. Like many inelastic models, the rate of deformation, D_{ij}, is additively decomposed into an elastic, D^{e}_{ij}, and an inelastic, D^{in}_{ij} part such that,

$$D_{ij} = D^{\mathrm{e}}_{ij} + D^{\mathrm{in}}_{ij}. \tag{5.102}$$

The elastic rate of deformation is the only part that contributes to the stress rate and it does so through the elastic moduli, $\mathbb{C}_{ijkl}$, in a linear fashion leading to the relation,

$$\overset{\circ}{\sigma}_{ij} = \mathbb{C}_{ijkl} D^{\mathrm{e}}_{kl}, \tag{5.103}$$

where $\mathbb{C}_{ijkl}$ are the components of the fourth-order, isotropic elasticity tensor. The stress rate is arbitrary, as long as it is objective. Two objective stress rates are commonly used: the Jaumann rate and the Green-McInnis rate. For problems with fixed principal axes of deformation, these two rates give the same answers. For problems where the principal axes of deformation rotate during the deformation, the two rates can give different answers. Generally speaking there is no reason to pick one objective rate over another.

The inelastic strain rate is a function of the stress state, σ_{ij}, the temperature, θ, and a number of internal state variables including both scalar isotropic, D, and tensorial kinematic, B_{ij}, hardening variables. With these dependencies defined, a general form for the evolution of the inelastic deformation may be given by,

$$D^{\mathrm{in}}_{ij} = \frac{3}{2}\gamma\left(\sigma_{ij}, \theta; D, B_{ij}\right) n_{ij}, \tag{5.104}$$

where n_{ij} is the direction of inelastic deformation and is defined as,

$$n_{ij} = \frac{1}{\tau}\left(s_{ij} - \frac{2}{3}B_{ij}\right), \tag{5.105}$$

and

$$\tau = \sqrt{\frac{3}{2}\left(s_{ij} - \frac{2}{3}B_{ij}\right)\left(s_{ij} - \frac{2}{3}B_{ij}\right)}, \tag{5.106}$$

with s_{ij} being the deviatoric stress tensor. The inelastic strain rate, γ, is defined via a hyperbolic sin law,

$$\gamma = f(\theta)\left[\sinh\left(\frac{\tau}{\alpha(\theta)D}\right)\right]^{p(\theta)}, \tag{5.107}$$

where $f(\theta) = \exp(g(\theta))$. The expressions $g(\theta)$, $\alpha(\theta)$, and $p(\theta)$ are model parameters that are functions of temperature.

The evolution laws for the state variables D and B_{ij} are,

$$\dot{D} = \frac{A_1}{(D - D_0)^{A_3}}\gamma - A_2\left(D - D_0\right)^2, \tag{5.108}$$

and

$$\dot{B}_{ij} = \frac{A_4}{b^{A_6}} D^{\text{in}}_{ij} - A_5 b B_{ij}, \tag{5.109}$$

where

$$b = \sqrt{\frac{2}{3} B_{ij} B_{ij}}. \tag{5.110}$$

The parameters D_0, A_1, A_2, A_3, A_4, A_5 and A_6 are model parameters. The parameters A_1, A_2, A_4 and A_5 are also functions of temperature. The model can be simplified with the appropriate choice of these parameters.

The following material parameters are functions of temperature and have the following form

$$\begin{gathered} G(\theta) = G_0 h_G(\theta) \quad ; \quad K(\theta) = K_0 h_K(\theta) \\ g(\theta) = g_0 h_g(\theta) \quad ; \quad p(\theta) = p_0 h_p(\theta) \quad ; \quad \alpha(\theta) = \alpha_0 h_\alpha(\theta) \\ A_1(\theta) = A_1^0 h_1(\theta) \quad ; \quad A_2(\theta) = A_2^0 h_2(\theta) \\ A_4(\theta) = A_4^0 h_4(\theta) \quad ; \quad A_5(\theta) = A_5^0 h_5(\theta) \end{gathered} \tag{5.111}$$

where the functions $h_*(\theta)$ are normalized functions of temperature and the values $(*)_0$ or $(*)^0$ are the reference values that are input in the command block.

In the above command blocks:

- Since the model requires functions to describe the temperature dependence of the bulk and shear modulus, it is recommended that one inputs the bulk and shear modulus at some reference temperature. However, any two of the elastic constants can be used for input. See Section 5.1.5 for more information on elastic constants input.
- The reference value in the equation for the flow rate, $\ln f_0$, is defined with the `FLOW RATE` command line.
- The reference value for the exponent on the sinh function, p_0, is defined with the `SINH EXPONENT` command line.
- The reference value $alpha_0$ is defined with the `ALPHA` command line.
- The reference value for the isotropic hardening parameter, A_1^0, is defined with the `ISO HARDENING` command line.
- The reference value for the isotropic recovery parameter, A_2^0, is defined with the `ISO RECOVERY` command line.

- The value for the isotropic hardening exponent parameter, A_3, is defined with the `ISO EXPONENT` command line.
- The reference value for the kinematic hardening parameter, A_4^0, is defined with the `KIN HARDENING` command line.
- The reference value for the kinematic recovery parameter, A_5^0, is defined with the `KIN RECOVERY` command line.
- The value for the kinematic hardening exponent parameter, A_6, is defined with the `KIN EXPONENT` command line.
- The value for the flow stress, D_0, is defined with the `FLOW STRESS` command line.
- The user-defined and normalized function that gives the shear modulus as a function of temperature, $h_G(\theta)$, is defined with the `SHEAR FUNCTION` command line.
- The user-defined and normalized function that gives the bulk modulus as a function of temperature, $h_K(\theta)$, is defined with the `BULK FUNCTION` command line.
- The user-defined and normalized function that gives the flow rate as a function of temperature, $h_g(\theta)$, is defined with the `RATE FUNCTION` command line.
- The user-defined and normalized function that gives the sinh exponent as a function of temperature, $h_p(\theta)$, is defined with the `EXPONENT FUNCTION` command line.
- The user-defined and normalized function that gives α as a function of temperature, $h_\alpha(\theta)$, is defined with the `ALPHA FUNCTION` command line.
- The user-defined and normalized function that gives A_1 as a function of temperature, $h_1(\theta)$, is defined with the `IHARD FUNCTION` command line.
- The user-defined and normalized function that gives A_2 as a function of temperature, $h_2(\theta)$, is defined with the `IREC FUNCTION` command line.
- The user-defined and normalized function that gives A_4 as a function of temperature, $h_4(\theta)$, is defined with the `KHARD FUNCTION` command line.
- The user-defined and normalized function that gives A_5 as a function of temperature, $h_5(\theta)$, is defined with the `KREC FUNCTION` command line.
- The Viscoplastic model may need to take sub-increments to solve for the plastic flow over the current time step. The maximum number of steps that may be taken on a step prior to issuing an error can be set by the `MAX SUBINCREMENTS` command line. This value defaults to 2000.

Output variables available for this model are listed in Table 5-19.

More information on the model can be found in the report by Neilsen, et. al. [29].

Table 5-19. State Variables for VISCOPLASTIC Model (Section 5.2.20)

Name	Description
EQPS	equivalent plastic strain
SVB	kinematic hardening variable, **B**
SVB_XX	kinematic hardening variable - xx component, B_{xx}
SVB_YY	kinematic hardening variable - yy component, B_{yy}
SVB_ZZ	kinematic hardening variable - zz component, B_{zz}
SVB_XY	kinematic hardening variable - xy component, B_{xy}
SVB_YZ	kinematic hardening variable - yz component, B_{yz}
SVB_ZX	kinematic hardening variable - zx component, B_{zx}
SVD	isotropic hardening variable, D
EQDOT	inelastic strain rate, γ
COUNT	number of sub-increments
SHEAR	shear modulus, $G(\theta)$
BULK	bulk modulus, $K(\theta)$
RATE	$g(\theta)$ (see(5.107))
EXP	$p(\theta)$ (see(5.107))
ALPHA	$\alpha(\theta)$ (see(5.107))
A1	isotropic hardening parameter, $A_1(\theta)$
A2	isotropic recovery parameter, $A_2(\theta)$
A4	kinematic hardening parameter, $A_4(\theta)$
A5	kinematic recovery parameter, $A_5(\theta)$

5.2.21. Munson-Dawson Viscoplastic Model

```
BEGIN PARAMETERS FOR MODEL MD_VISCOPLASTIC
  # Elastic constants
  YOUNGS MODULUS  = <real> E
  POISSONS RATIO  = <real> ν
  SHEAR MODULUS   = <real> G
  BULK MODULUS    = <real> K
  LAMBDA          = <real> λ
  TWO MU          = <real> 2μ
  # Steady-state creep parameters
  A0              = <real> A0 (0.0)
  A1              = <real> A1 (0.0)
  A2              = <real> A2 (0.0)
  Q0oR            = <real> Q0/R (0.0)
  Q1oR            = <real> Q1/R (0.0)
  Q2oR            = <real> Q2/R (0.0)
  n0              = <real> n0 (0.0)
  n1              = <real> n1 (0.0)
  n2              = <real> n2 (0.0)
  sigma_g         = <real> σg
  B0              = <real> B0 (0.0)
  B1              = <real> B1 (0.0)
  B2              = <real> B2 (0.0)
  q               = <real> q (0.0)
  # Transient creep parameters
  K0              = <real> K0 (0.0)
  K1              = <real> K1 (0.0)
  c0              = <real> c0 (0.0)
  c1              = <real> c1 (0.0)
  m0              = <real> m0 (0.0)
  m1              = <real> m1 (0.0)
  alpha_h         = <real> αh (0.0)
  alpha_r         = <real> αr (0.0)
  beta_h          = <real> βh (0.0)
  beta_r          = <real> βr (0.0)
  # Other parameters
  alpha           = <real> α (0.0)
  a               = <real> a (1000.0)
  # Numerical implementation parameters  (see LAMÉ manual)
  _chi            = <real> χ (2.0)
  _sigma_min      = <real> σmin (μ×10^-10)
  _sqrt_omega_max = <real> √ωmax (10^-11)
  _xi             = <real> ξ (10^-4)
  _gamma          = <real> γ (0.1)
  _k_max          = <real> kmax (100)
  _j_max          = <real> jmax (10)
```

```
END [PARAMETERS FOR MODEL MD_VISCOPLASTIC]
```

The Munson-Dawson (MD) model was originally defined in [30, 31, 32], but several changes were made in [33]. This section presents the current version of the model. Note that compressive stresses and strains are treated as positive in this section, as is common in the geomechanics literature.

The MD model is an isotropic, hypoelastic, unified viscoplastic, material model. The total strain rate $\dot{\varepsilon}_{ij}$ is decomposed into an elastic strain rate $\dot{\varepsilon}^{\text{el}}_{ij}$, a thermal strain rate $\dot{\varepsilon}^{\text{th}}_{ij}$, and a viscoplastic strain rate $\dot{\varepsilon}^{\text{vp}}_{ij}$:

$$\dot{\varepsilon}_{ij} = \dot{\varepsilon}^{\text{el}}_{ij} + \dot{\varepsilon}^{\text{th}}_{ij} + \dot{\varepsilon}^{\text{vp}}_{ij}. \tag{5.112}$$

The elastic portion of the MD model utilizes the following simple linear relationship between $\dot{\varepsilon}^{\text{el}}_{kl}$ and the stress rate $\dot{\sigma}_{ij}$,

$$\dot{\sigma}_{ij} = \mathbb{C}_{ijkl}\dot{\varepsilon}^{\text{el}}_{kl} = \mathbb{C}_{ijkl}\left(\dot{\varepsilon}_{kl} - \dot{\varepsilon}^{\text{th}}_{kl} - \dot{\varepsilon}^{\text{vp}}_{kl}\right) \tag{5.113}$$

$$\mathbb{C}_{ijkl} = (B - 2/3\,\mu)\,\delta_{ij}\delta_{kl} + \mu\left(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}\right), \tag{5.114}$$

where $\mathbb{C}_{ijkl}$ is the elastic stiffness, which is composed of the bulk modulus B, the shear modulus μ, and the Kronecker Delta δ_{ij}. The thermal strain portion of the model is simply

$$\dot{\varepsilon}^{\text{th}}_{ij} = -\alpha\,\dot{\theta}\,\delta_{ij} \tag{5.115}$$

where α is the coefficient of thermal expansion, and θ is the temperature. Sierra/SM also offers thermal strain functions for adding thermal strain effects to any given model. If $\alpha \neq 0$, then MD model users should not specify a thermal strain function, otherwise thermal strains will be applied twice.

Plastic deformation is assumed to be isochoric and only occurs in the presence of shear stress. The MD model utilizes the Hosford stress as its equivalent shear stress measure $\bar{\sigma}$. The Hosford stress is

$$\bar{\sigma} = \left\{\frac{1}{2}\left[|\sigma_1 - \sigma_2|^a + |\sigma_2 - \sigma_3|^a + |\sigma_1 - \sigma_3|^a\right]\right\}^{1/a}, \tag{5.116}$$

where σ_i are the principal stresses and a is a material parameter. This definition for $\bar{\sigma}$ was proposed in [9] because it encompasses the Tresca stress ($a = 1$), the von Mises stress ($a = 2$), and a range of behaviors in-between ($1 < a < 2$). One can also reproduce the Tresca stress with $a = \infty$, the von Mises stress with $a = 4$, and behaviors in-between with $4 < a < \infty$. This second range avoids potential singularities in the first and second derivatives of (5.116), so the exponent is restricted to $a \geq 4$.

The viscoplastic strain evolves according to an associated flow rule

$$\dot{\varepsilon}^{\text{vp}}_{ij} = \dot{\bar{\varepsilon}}^{\text{vp}}\frac{\partial\bar{\sigma}}{\partial\sigma_{ij}}, \tag{5.117}$$

where $\dot{\bar{\varepsilon}}^{\text{vp}}$ is the equivalent viscoplastic strain rate. It can be decomposed into two components

$$\dot{\bar{\varepsilon}}^{\text{vp}} = \dot{\bar{\varepsilon}}^{\text{tr}} + \dot{\bar{\varepsilon}}^{\text{ss}}, \tag{5.118}$$

where $\dot{\bar{\varepsilon}}^{\text{tr}}$ is the transient equivalent viscoplastic strain rate and $\dot{\bar{\varepsilon}}^{\text{ss}}$ is the steady state equivalent viscoplastic strain rate.

The MD model decomposes the steady state behavior into four "mechanisms":

$$\begin{aligned}
\dot{\bar{\varepsilon}}^{\text{ss}} &= \sum_{i=0}^{3} \dot{\bar{\varepsilon}}_i^{\text{ss}} \\
\dot{\bar{\varepsilon}}_i^{\text{ss}} &= A_i \exp\left(-\frac{Q_i}{R\theta}\right)\left(\frac{\bar{\sigma}}{\mu}\right)^{n_i} \quad \text{for } i = 0,\ 1,\ \text{and } 2 \\
\dot{\bar{\varepsilon}}_3^{\text{ss}} &= H(\bar{\sigma} - \bar{\sigma}_{\text{g}}) \sum_{i=0}^{2} B_i \exp\left(-\frac{Q_i}{R\theta}\right) \sinh\left(q\frac{(\bar{\sigma} - \bar{\sigma}_{\text{g}})}{\mu}\right),
\end{aligned} \tag{5.119}$$

The variables A_i, B_i, Q_i, n_i, $\bar{\sigma}_{\text{g}}$, and q are all model parameters. All four mechanisms have an Arrhenius temperature dependence, where Q_i is an activation energy and $R = 8.314$ J/(K mol) is the universal gas constant. Mechanism 3 is only activated when $\bar{\sigma}$ exceeds $\bar{\sigma}_{\text{g}}$, as reflected in the heaviside function $H(\bar{\sigma} - \bar{\sigma}_{\text{g}})$. Typically, the parameters B_i are chosen to produce a smooth transition to mechanism 3 at $\bar{\sigma}_{\text{g}}$.

The simple functional forms of (5.119) suffice for the steady-state behavior, but the transient behavior is somewhat more complex. During work hardening under constant stress, $\bar{\varepsilon}^{\text{tr}}$ approaches the transient strain limit $\bar{\varepsilon}^{\text{tr}^*}$ from below, and the total viscoplastic strain rate slows down over time. During recovery under constant stress, $\bar{\varepsilon}^{\text{tr}}$ approaches $\bar{\varepsilon}^{\text{tr}^*}$ from above, and the total viscoplastic strain rate speeds up over time. The rate that $\bar{\varepsilon}^{\text{tr}}$ approaches $\bar{\varepsilon}^{\text{tr}^*}$ is governed by

$$\dot{\bar{\varepsilon}}^{\text{tr}} = (F - 1)\, \dot{\bar{\varepsilon}}^{\text{ss}}, \tag{5.120}$$

where

$$F = \exp\left[\text{sign}\,(\bar{\varepsilon}^{\text{tr}^*} - \bar{\varepsilon}^{\text{tr}})\,\kappa \left(1 - \frac{\bar{\varepsilon}^{\text{tr}}}{\bar{\varepsilon}^{\text{tr}^*}}\right)^2\right]. \tag{5.121}$$

and κ is a quantity that depends on whether the material is work hardening or recovering. These two behaviors are captured in the following equations

$$\kappa = \begin{cases} \alpha_{\text{h}} + \beta_{\text{h}} \log_{10}\left(\dfrac{\bar{\sigma}}{\mu}\right) & \bar{\varepsilon}^{\text{tr}} \le \bar{\varepsilon}^{\text{tr}^*} \\ \alpha_{\text{r}} + \beta_{\text{r}} \log_{10}\left(\dfrac{\bar{\sigma}}{\mu}\right) & \bar{\varepsilon}^{\text{tr}} > \bar{\varepsilon}^{\text{tr}^*}. \end{cases} \tag{5.122}$$

where α_j and β_j are model parameters. Note that the parameter κ must be non-negative, otherwise (5.120) produces a negative/positive $\dot{\bar{\varepsilon}}^{\text{tr}}$ when $\bar{\varepsilon}^{\text{tr}}$ is below/above $\bar{\varepsilon}^{\text{tr}^*}$. (Such behavior occurs during reverse creep, but the MD model is only designed to model forward creep.) To enforce this, (5.122) is calculated first, and then

$$\kappa \leftarrow \max(\kappa, 0) \tag{5.123}$$

is applied.

The MD model uses two mechanisms to endow $\bar{\varepsilon}^{\mathrm{tr}^*}$ with stress and temperature dependence:

$$\bar{\varepsilon}^{\mathrm{tr}^*} = \sum_{i=0}^{1} \bar{\varepsilon}_i^{\mathrm{tr}^*}$$
$$\bar{\varepsilon}_i^{\mathrm{tr}^*} = K_i \exp(c_i \theta) \left(\frac{\bar{\sigma}}{\mu} \right)^{m_i}, \tag{5.124}$$

where K_i, c_i, and m_i are parameters to be calibrated against experimental results.

In the command blocks that define the MD viscoplastic model:

- See Section 5.1.5 for more information on elastic constants input.

Output variables available for this model are listed in Table 5-20.

Table 5-20. State Variables for MD_VISCOPLASTIC Model (Section 5.2.21)

Name	Description
EQ_TR_STRAIN	equivalent transient viscoplastic strain, $\bar{\varepsilon}^{\mathrm{tr}}$
EQ_VP_STRAIN	equivalent viscoplastic strain, $\bar{\varepsilon}^{\mathrm{vp}}$
EQ_STRESS	equivalent stress, $\bar{\sigma}$
VP_RATE_SCALE_FACTOR	viscoplastic rate scale factor, s

5.2.22. Soil and Crushable Foam Model

```
BEGIN PARAMETERS FOR MODEL SOIL_FOAM
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Yield surface parameters
  #
  A0 = <real> a0
  A1 = <real> a1
  A2 = <real> a2
  PRESSURE CUTOFF   = <real> pc
  PRESSURE FUNCTION = <string> fp(εV)
END [PARAMETERS FOR MODEL SOIL_FOAM]
```

The soil and crushable foam model is a plasticity model that can be used for modeling soil, crushable foam, or other highly compressible materials. Given the right input, the model is a Drucker-Prager model.

For the soil and crushable foam model, the yield surface is a surface of revolution about the hydrostat in principal stress space. A planar end cap is assumed for the yield surface so that the yield surface is closed. The yield stress σ_{yd} is specified as a polynomial in pressure p. The yield stress is given as:

$$\sigma_{yd} = a_0 + a_1 p + a_2 p^2 , \tag{5.125}$$

where the pressure p is positive in compression. The determination of the yield stress from Equation (5.125) places severe restrictions on the admissible values of a_0, a_1, and a_2. There are three valid cases for the yield surface. In the first case, there is an elastic–perfectly plastic deviatoric response, and the yield surface is a cylinder oriented along the hydrostat in principal stress space. In this case, a_0 is positive, and a_1 and a_2 are zero. In the second case, the yield surface is conical. A conical yield surface is obtained by setting a_2 to zero and using appropriate values for a_0 and a_1. In the third case, the yield surface has a parabolic shape. For the parabolic yield surface, all three coefficients in Equation (5.125) are nonzero. The coefficients are checked to determine that a valid negative tensile-failure pressure can be derived based on the specified coefficients.

For the case of the cylindrical yield surface (e.g., $a_0 > 0$ and $a_1 = a_2 = 0$), there is no tensile-failure pressure. For the other two cases, the computed tensile-failure pressure may be too low. To

handle the situations where there is no tensile-failure pressure or the tensile-failure pressure is too low, a pressure cutoff can be defined. If a pressure cutoff is defined, the tensile-failure pressure is the larger of the computed tensile-failure pressure and the defined cutoff pressure.

The plasticity theories for the volumetric and deviatoric parts of the material response are completely uncoupled. The volumetric response is computed first. The mean pressure p is assumed to be positive in compression, and a yield function ϕ_p is written for the volumetric response as:

$$\phi_p = p - f_p(\varepsilon_V)\,, \tag{5.126}$$

where $f_p(\varepsilon_V)$ defines the volumetric stress-strain curve for the pressure. The yield function ϕ_p determines the motion of the end cap along the hydrostat.

In the above command blocks:

- See Section 5.1.5 for more information on elastic constants input.
- The constant coefficient in the equation for the yield surface (5.125) is defined with the `A0` command line.
- The coefficient for the linear term in the equation for the yield surface (5.125) is defined with the `A1` command line.
- The coefficient for the quadratic term in the equation for the yield surface (5.125) is defined with the `A2` command line.
- The user-defined maximum tensile-failure pressure is defined with the `PRESSURE CUTOFF` command line.
- The pressure as a function of volumetric strain is defined with the function named on the `PRESSURE FUNCTION` command line.

For information about the soil and crushable foam model, see the PRONTO3D document listed as Reference [34]. The soil and crushable foam model is the same as the soil and crushable foam model in PRONTO3D. The PRONTO3D model is based on a material model developed by Krieg [35]. The Krieg version of the soil and crushable foam model was later modified by Swenson and Taylor [36]. The soil and crushable foam model developed by Swenson and Taylor is the model in PRONTO3D and is also the shared model for Presto and Adagio.

Output variables available for this model are listed in Table 5-21.

Table 5-21. State Variables for SOIL FOAM Model (Section 5.2.22)

Name	Description
EVOL_MAX	maximum volumetric strain seen by the material point
EVOL_FRAC	volumetric strain for tensile fracture
EVOL	current volumetric strain
EQPS	equivalent plastic strain

5.2.23. Foam Plasticity Model

```
BEGIN PARAMETERS FOR MODEL FOAM_PLASTICITY
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  #
  #
  PHI              = <real> φ0
  SHEAR STRENGTH   = <real> A0
  SHEAR HARDENING  = <real> A1
  SHEAR EXPONENT   = <real> A2
  HYDRO STRENGTH   = <real> B0
  HYDRO HARDENING  = <real> B1
  HYDRO EXPONENT   = <real> B2
  BETA             = <real> β
  P0               = <real> p0
END [PARAMETERS FOR MODEL FOAM_PLASTICITY]
```

The foam plasticity model was developed to describe the response of porous materials (like closed-cell polyurethane foams) exhibiting irreversible, elastic-plastic like responses through large deformations. Such foams can exhibit significant plastic deviatoric and volumetric strains leading to permanent shape and volume changes, respectively. The former behavior is quite typical of metals and corresponding theories are well established. The latter response, however, is not typical of metals and a theory combining these two behaviors is needed. Given these responses of interest, the foam plasticity model is well suited to use with metal foams and many closed-cell polymeric foams (e.g. polyurethane, polystyrene bead, etc.) subjected to large deformations. As permanent strains are of interest, this model is not appropriate for use with flexible foams that return to their undeformed shape after loads are removed.

Specifically, the model developed by Neilsen et al. [37] seeks to capture the response associated with three distinct deformation regimes. First, when foams are initially compressed, they typically exhibit an elastic response. After sufficient load is applied, a plateau of nearly constant stress over a large deformation region is noted as pores start to compress and cell walls undergo substantial deformation. Eventually, the various collapsed cells and walls begin to interact and a densification response with substantial hardening is observed. Details of these deformation processes may be found in the text of Gibson and Ashby [38].

Like other plasticity-based models, the incremental constitutive law for the foam plasticity model is written as,

$$\overset{\circ}{\sigma}_{ij} = \mathbb{C}_{ijkl} D^{\mathrm{e}}_{kl}, \tag{5.127}$$

where an additive decomposition of the strain rates such that $D_{ij} = D^{\mathrm{e}}_{ij} + D^{\mathrm{p}}_{ij}$ is assumed. To describe the inelastic response of the foams of interest, Neilsen and coworkers [37] proposed a yield function of the form

$$f = \frac{\bar{\sigma}^2}{a^2} + \frac{p^2}{b^2} - 1, \tag{5.128}$$

where $\bar{\sigma}$ is the von Mises effective stress ($\bar{\sigma} = \sqrt{(3/2)\, s_{ij} s_{ij}}$ with s_{ij} being the deviatoric stress) and p being the hydrostatic pressure that is positive in compression ($p = -(1/3)\sigma_{kk}$). In such a form, the initial yield surface forms an ellipsoid about the hydrostat. The two denominators, a and b, are state variables capturing hardening effects and have the functional form of,

$$a = A_0 + A_1 \phi^{A_2}, \tag{5.129}$$

$$b = \begin{cases} B_0 + B_1 \phi^{B_2} & p \geq 0 \\ B_0 & p < 0 \end{cases}, \tag{5.130}$$

with A_0, A_1, A_2, B_0, B_1, and B_2 being model parameters and ϕ being the maximum volume fraction of solid material obtained through the loading history and is defined as,

$$\phi = \phi_0 \frac{V_0}{V}, \tag{5.131}$$

in which ϕ_0 is the initial volume fraction of solid material and V_0 and V are the initial volume and current volume, respectively, of the foam. Put in terms of the deformation,

$$\phi = \phi_0 \frac{1}{1 + \varepsilon_{\mathrm{V}}}, \tag{5.132}$$

where ε_{V} is the engineering volume strain.

To describe the inelastic plastic deformation, a non-associated flow rule is used. Specifically,

$$D^{\mathrm{p}}_{ij} = \dot{\gamma} g_{ij}, \tag{5.133}$$

where $\dot{\gamma}$ is the consistency multiplier found by enforcing the corresponding condition and

$$g_{ij} = (1 - \beta)\, g^{\mathrm{a}}_{ij} + \beta g^{\mathrm{r}}_{ij}, \tag{5.134}$$

with the superscripts “a” and “r” being used to denote associated and radial flow directions, respectively. The model parameter β is introduced in (5.134) to enable associated ($\beta = 0$), radial

($\beta = 1$), or a linear combination of the two flow rules ($0 < \beta < 1$) to be used. The two direction vectors may written as,

$$g^{\mathrm{a}}_{ij} = \frac{\frac{\partial f}{\partial \sigma_{ij}}}{|\frac{\partial f}{\partial \sigma_{kl}}|} = \frac{\frac{3}{a^2} s_{ij} - \frac{2}{3b^2} p\delta_{ij}}{|\frac{3}{a^2} s_{ij} - \frac{2}{3b^2} p\delta_{ij}|}, \tag{5.135}$$

$$g^{\mathrm{r}}_{ij} = \frac{\sigma_{ij}}{|\sigma_{kl}|} = \frac{\sigma_{ij}}{\sqrt{\sigma_{kl}\sigma_{kl}}}. \tag{5.136}$$

In alternative models of foam plasticity, the yield function is offset in the $\bar{\sigma} - p$ plane along the pressure axis. To generalize the model, an offset yield surface centered at pressure p_0 may be used in the form

$$\hat{f} = \frac{\bar{\sigma}^2}{a^2} + \frac{(p - p_0)^2}{b^2} - 1. \tag{5.137}$$

Note that with this form of the yield function, the component of the flow direction g^a_{ij} is now non-associative. This component is associative to the yield surface centered at the origin, but non-associative to the yield surface centered at p_0.

In the above command blocks:

- See Section 5.1.5 for more information on elastic constants input.
- The initial volume fraction of solid material in the foam, φ, is defined with the `PHI` command line. For example, solid polyurethane weighs 75 pounds per cubic foot (pcf); uncompressed 10 pcf polyurethane foam would have a φ of 0.133 = 10/75.
- The shear (deviatoric) strength of uncompressed foam is defined with the `SHEAR STRENGTH` command line.
- The shear hardening modulus for the foam is defined with the `SHEAR HARDENING` command line.
- The shear hardening exponent is defined with the `SHEAR EXPONENT` command line. The deviatoric strength is given by `(SHEAR STRENGTH) + (SHEAR HARDENING) * PHI**(SHEAR EXPONENT).`
- The hydrostatic (volumetric) strength of the uncompressed foam is defined with the `HYDRO STRENGTH` command line.
- The hydrodynamic hardening modulus is defined with the `HYDRO HARDENING` command line.
- The hydrodynamic hardening exponent is defined with the `HYDRO EXPONENT` command line. The hydrostatic strength is given by `(HYDRO STRENGTH) + (HYDRO HARDENING) * PHI**(HYDRO EXPONENT).`
- The prescription for non associated flow, β, is defined with the `BETA` command line. When $\beta = 0.0$, the flow direction is given by the normal to the yield surface (associated flow).

When $\beta = 1.0$, the flow direction is given by the stress tensor. It is recommended that associated flow $\beta = 0.0$ be used for most analyses.

Output variables available for this model are listed in Table 5-22.

Table 5-22. State Variables for FOAM PLASTICITY Model (Section 5.2.23)

Name	Description
ITER	iterations
EVOL	volumetric strain
PHI	phi, ϕ
EQPS	equivalent plastic strain, $\bar{\varepsilon}^p$
PA	A
PB	B

5.2.24. Low Density Foam Model

```
BEGIN PARAMETERS FOR MODEL LOW_DENSITY_FOAM
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  A    = <real> A
  B    = <real> B
  C    = <real> C
  NAIR = <real> N_air
  P0   = <real> p_0
  PHI  = <real> φ
END [PARAMETERS FOR MODEL LOW_DENSITY_FOAM]
```

The low density foam material model is a phenomenological model for rigid, low density polyurethane foams. Development of this model followed extensive characterization efforts at Sandia National Laboratory with special emphasis placed on hydrostatic and triaxial compression tests [39]. A key observation of this investigation was the impact of trapped air inside the foam on the load bearing capabilities of the material.

In constructing a model describing the response of the low-density foams, Neilsen et al. [39] decomposed the response into that of the polymeric skeleton and the air such that,

$$\sigma_{ij} = \sigma_{ij}^{\mathrm{sk}} + \sigma^{\mathrm{air}}\delta_{ij}, \tag{5.138}$$

where the super script "sk" is used to refer to variables relating to the skeleton and "air" to the air. The contribution of the air component is only present, however, in constrained cases when the internal gases are trapped and not allowed to escape. If the foam material in not encased or encapsulated in someway, the air may escape and $\sigma^{\mathrm{air}} = 0$. A model parameter, N_{air}, is included to distinguish between these cases. If N_{air} is set to 0, the air pressure term is set to zero. For any other value, it is included.

Using the ideal gas law, it can be found that for an isothermal case,

$$\sigma^{\mathrm{air}} = \frac{p_0 \varepsilon_{\mathrm{V}}}{\varepsilon_{\mathrm{V}} + 1 - \phi}, \tag{5.139}$$

where p_0, ε_{V}, and ϕ are the initial air pressure, volumetric strain, and the volume fraction of the solid (skeleton) material. Knowing the total stress of the material volume and air contribution, the skeleton stress may be found via (5.138). Furthermore, it should be noted that the foam (total) and skeleton strains are the same.

Based on their experimental observations, Neilsen et al. [39] noted a decoupling between the skeleton principal stresses. Therefore, the Poisson's ratio of the skeleton is zero and that the yielding behavior in each principal direction is independent. A yield function of the form,

$$f_i = \sigma_i^{\mathrm{sk}} - \bar{\sigma}, \tag{5.140}$$

where f_i and σ_i^{sk} are the ith yield function and skeleton principal stress, respectively, and

$$\bar{\sigma} = A\left\langle I_2' \right\rangle + B(1.0 + C\varepsilon_V) \tag{5.141}$$

with A, B, and C are material parameters, and $\langle\cdot\rangle$ denoting the Heaviside step function where

$$\langle x \rangle = \begin{cases} 0 & \text{if } x \le 0 \\ 1 & \text{if } x > 0 \end{cases}, \tag{5.142}$$

was proposed. Additionally, I_2' is the second invariant of the deviatoric strain. If a skeleton principal stress indicates yielding, it is set to the effective yield stress value, $\bar{\sigma}$.

See Section 5.1.5 for more information on elastic constants input.

State variables for this model are listed in Table 5-23.

For more information on the low density foam material model, see [39].

Table 5-23. State Variables for LOW DENSITY FOAM Model (Section 5.2.24)

Name	Description
PAIR	Air pressure

5.2.25. Hyperfoam Model

```
BEGIN PARAMETERS FOR MODEL HYPERFOAM
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Strain energy density
  #
  N = <integer> N
  SHEAR    = <real_list> μ_i
  ALPHA    = <real_list> α_i
  POISSON  = <real_list> ν_i
END [PARAMETERS FOR HYPERFOAM]
```

The hyperfoam model is a hyperelastic model that can be used for modeling elastomeric foams. It is based on a strain energy with a form given by Störakers [40] which is similar to a form presented by Ogden [41]. The strain energy depends on the principal stretch ratios of the material and is given by

$$W(\lambda_k) = \sum_{i=1}^{N} \frac{2\mu_i}{\alpha_i^2} \left[\lambda_1^{\alpha_i} + \lambda_2^{\alpha_i} + \lambda_3^{\alpha_i} - 3 + \frac{1}{\beta_i} \left(J^{-\alpha_i \beta_i} - 1 \right) \right] \tag{5.143}$$

where μ_i and α_i are input parameters and J is the determinant of the deformation gradient. The value of β_i is calculated from the parameters ν_i via

$$\beta_i = \frac{\nu_i}{1 - 2\nu_i}. \tag{5.144}$$

The ν_i can be thought of as Poisson's ratios, however in the context of the summation in (5.143) it is best to consider them as fitting parameters.

The strain energy (5.143) is a sum of N contributions. The principal Kirchoff stresses for the hyperfoam model, τ_k, can be calculated as

$$\tau_k = \lambda_k \frac{\partial W}{\partial \lambda_k} \tag{5.145}$$

which can be used to calculate the components of the Kirchoff stress, τ_{ij}, through

$$\tau_{ij} = \sum_{k=1}^{3} \tau_k \hat{e}_i^k \hat{e}_j^k. \tag{5.146}$$

where $\hat{e}_i^k$ are the components of the kth eigenvector of the left stretch tensor in the global Cartesian coordinate system. The components of the Cauchy stress are then

$$\sigma_{ij} = \frac{1}{J}\tau_{ij}. \tag{5.147}$$

Finally, it should be noted that the Hyperfoam model is also capable of reproducing the Blatz-Ko model [42, 43]. If only one term is chosen, $N = 1$, and $\mu_1 = \mu$, $\alpha_1 = -2$, and $\nu_1 = 0.25$ we get the Blatz-Ko strain energy density

$$W = \frac{\mu}{2}\left(\frac{I_2}{I_3} + 2\sqrt{I_3} - 5\right) \quad , \tag{5.148}$$

where I_2 and I_3 are the second and third invariants of the right Cauchy-Green tensor.

- See Section 5.1.5 for more information on elastic constants input.
- The number of terms in the expansion of the strain energy is defined with the `N` command line.
- The shear terms in the expansion of the strain energy, μ_i, are defined with the `SHEAR` command line.
- The alpha terms in the expansion of the strain energy, α_i, are defined with the `ALPHA` command line.
- The Poisson ratio terms in the expansion of the strain energy, ν_i, are defined with the `POISSON` command line.

There are no output variables available for this model.

5.2.26. Elastic Three-Dimensional Orthotropic Model

```
BEGIN PARAMETERS FOR MODEL ELASTIC_3D_ORTHOTROPIC
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Material coordinates system definition
  #
  COORDINATE SYSTEM             = <string> coordinate_system_name
  DIRECTION FOR ROTATION        = <real>   1|2|3
  ALPHA                         = <real>   α1 (degrees)
  SECOND DIRECTION FOR ROTATION = <real>   1|2|3
  SECOND ALPHA                  = <real>   α2 (degrees)
  #
  # Required parameters
  #
  YOUNGS MODULUS AA = <real> E_AA
  YOUNGS MODULUS BB = <real> E_BB
  YOUNGS MODULUS CC = <real> E_CC
  POISSONS RATIO AB = <real> ν_AB
  POISSONS RATIO BC = <real> ν_BC
  POISSONS RATIO CA = <real> ν_CA
  SHEAR MODULUS AB  = <real> G_AB
  SHEAR MODULUS BC  = <real> G_BC
  SHEAR MODULUS CA  = <real> G_CA
  #
  # Thermal strain functions
  #
  THERMAL STRAIN AA FUNCTION = <string> ε^th_aa(θ)
  THERMAL STRAIN BB FUNCTION = <string> ε^th_bb(θ)
  THERMAL STRAIN CC FUNCTION = <string> ε^th_cc(θ)
  #
END [PARAMETERS FOR MODEL ELASTIC_3D_ORTHOTROPIC]
```

The ELASTIC 3D ORTHOTROPIC model is an extension of the previously discussed ELASTIC routine and describes the linear elastic response of a material which exhibits orthotropic symmetry, where the orientation of the principal material directions can be arbitrary with respect to the global Cartesian axes as specified by the user.

First, a rectangular, cylindrical, or spherical reference coordinate system is defined. The material coordinate system can then be defined through two successive rotations about axes in the

reference coordinate system. Refer to Section 5.1.7 for details on the definition of material coordinate systems. These principal axes are denoted as A, B, and C in the following. Thermal strains are also defined with respect to these principal material axes.

The elastic stiffness for an orthotropic material can be described in terms of the elastic compliance which relates the strain to the stress, $\varepsilon_{ij} = \mathbb{S}_{ijkl}\sigma_{kl}$. For a material with an orthogonal ABC coordinate system, and written in that reference frame, the elastic compliance tensor is given by

$$\left[\tilde{\mathbb{S}}\right] = \begin{bmatrix} \frac{1}{E_{AA}} & -\frac{\nu_{BA}}{E_{BB}} & -\frac{\nu_{CA}}{E_{CC}} & 0 & 0 & 0 \\ -\frac{\nu_{AB}}{E_{AA}} & \frac{1}{E_{BB}} & -\frac{\nu_{CB}}{E_{CC}} & 0 & 0 & 0 \\ -\frac{\nu_{AC}}{E_{AA}} & -\frac{\nu_{BC}}{E_{BB}} & \frac{1}{E_{CC}} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2G_{AB}} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2G_{BC}} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2G_{CA}} \end{bmatrix}, \tag{5.149}$$

where the " $\tilde{}$ " is used to denote a variable in the *ABC* material system.

From the definition (5.149), it can be seen that requiring symmetry leads to relations of the form,

$$\nu_{BA} = \nu_{AB}\frac{E_{BB}}{E_{AA}} \quad ; \quad \nu_{CB} = \nu_{BC}\frac{E_{CC}}{E_{BB}} \quad ; \quad \nu_{AC} = \nu_{CA}\frac{E_{AA}}{E_{CC}}. \tag{5.150}$$

Therefore, only 9 independent constants are needed to fully define the model behavior.

The orthotropic model is also formulated in a hypoelastic fashion, leading to a constitutive equation (in the ABC material frame) of,

$$\overset{\circ}{\tilde{\sigma}}_{ij} = \tilde{\mathbb{C}}_{ijkl}\left(\tilde{D}_{kl} - \tilde{D}^{th}_{kl}\right), \tag{5.151}$$

where $\tilde{D}^{th}_{ij}$ is the thermal strain rate.

The elastic stiffness tensor, $\tilde{\mathbb{C}}_{ijkl}$, is the inverse of the compliance, $\tilde{\mathbb{C}}_{ijkl} = \tilde{\mathbb{S}}^{-1}_{ijkl}$, and as such may be determined to be,

$$\left[\tilde{\mathbb{C}}\right] = \begin{bmatrix} \mathbb{C}_{AAAA} & \mathbb{C}_{AABB} & \mathbb{C}_{CCAA} & 0 & 0 & 0 \\ \mathbb{C}_{AABB} & \mathbb{C}_{BBBB} & \mathbb{C}_{BBCC} & 0 & 0 & 0 \\ \mathbb{C}_{CCAA} & \mathbb{C}_{BBCC} & \mathbb{C}_{CCCC} & 0 & 0 & 0 \\ 0 & 0 & 0 & 2G_{AB} & 0 & 0 \\ 0 & 0 & 0 & 0 & 2G_{BC} & 0 \\ 0 & 0 & 0 & 0 & 0 & 2G_{CA} \end{bmatrix}. \tag{5.152}$$

where

$$\begin{aligned} \mathbb{C}_{AAAA} &= \frac{1-\nu_{BC}\nu_{CB}}{\Delta}E_{AA} \quad ; \quad \mathbb{C}_{BBBB} = \frac{1-\nu_{CA}\nu_{AC}}{\Delta}E_{BB} \quad ; \quad \mathbb{C}_{CCCC} = \frac{1-\nu_{AB}\nu_{BA}}{\Delta}E_{CC} \\ \mathbb{C}_{AABB} &= \frac{\nu_{BA}+\nu_{CA}\nu_{BC}}{\Delta}E_{AA} \quad ; \quad \mathbb{C}_{BBCC} = \frac{\nu_{CB}+\nu_{AB}\nu_{CA}}{\Delta}E_{BB} \quad ; \quad \mathbb{C}_{CCAA} = \frac{\nu_{AC}+\nu_{BC}\nu_{AB}}{\Delta}E_{CC} \end{aligned} \tag{5.153}$$

and $\Delta = 1-\nu_{AB}\nu_{BA}-\nu_{BC}\nu_{CB}-\nu_{CA}\nu_{RT}-2\nu_{AB}\nu_{BC}\nu_{CA}$.

See [44] for more information about the elastic three-dimensional orthotropic model.

In the above command blocks all of the following are required inputs.

- Even though they are not used within the material model itself, elastic constants are still required input for hourglass control, certain preconditioners, and other various capabilities. After examining various test problems, it has been determined that using the mean of the orthotropic properties as the isotropic elastic constants yields the best results. See Section 5.1.5 for more information on elastic constants input.
- See Section 5.1.7 for more information on material coordinates system definition commands.
- The Young's moduli corresponding to the principal material axes A, B, and C are given by the `YOUNGS MODULUS AA`, `YOUNGS MODULUS BB`, and `YOUNGS MODULUS CC` command lines.
- The Poisson's ratio defining the BB normal strain when the material is subjected only to AA normal stress is given by the `POISSONS RATIO AB` command line.
- The Poisson's ratio defining the CC normal strain when the material is subjected only to BB normal stress is given by the `POISSONS RATIO BC` command line.
- The Poisson's ratio defining the AA normal strain when the material is subjected only to CC normal stress is given by the `POISSONS RATIO CA` command line.

- The remaining Poisson's ratios needed for the orthotropic elastic relations (i.e. the BA, CB, and AC Poisson's ratios) are calculated internally. They are calculated as usual from the given Poisson's ratios, given Young's moduli, and energy considerations, which provide expressions for these parameters from the resulting symmetry of the compliance tensor.
- The shear moduli for shear in the AB, BC, and CA planes are given by the `SHEAR MODULUS AB`, `SHEAR MODULUS BC`, and `SHEAR MODULUS CA` command lines, respectively.
- The thermal strain functions for normal thermal strains along the principal material directions are given by the `THERMAL STRAIN AA FUNCTION`, `THERMAL STRAIN BB FUNCTION`, and `THERMAL STRAIN CC FUNCTION` command lines.



Warning: The `ELASTIC_3D_ORTHOTROPIC` model cannot currently be used in conjunction with the control stiffness implicit solver block.

There are no output variables available for the Elastic Three-Dimensional Orthotropic material model.

5.2.27. Wire Mesh Model

```
BEGIN PARAMETERS FOR MODEL WIRE_MESH
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Yield surface parameters
  #
  YIELD FUNCTION = <string> σ̄(ε_V)
  TENSION        = <real> τ
END [PARAMETERS FOR MODEL WIRE_MESH]
```

The wire mesh model was developed at Sandia National Laboratories for use with layered sequences of metallic wire meshes and cloth fabric. Model development was based on an extensive series of experiments performed on these materials (see [45]) and used an existing model for rigid polyurethane foams as a starting point [39].

To be able to analyze the response of this material, the Cauchy stress tensor is first decomposed into its principal components, σ^i. Each principal stress is evaluated independently and two behaviors are considered depending on whether or not the material is in tension or compression. Under a tensile load, the material is taken to be perfectly plastic above a yield stress, τ. For compressive loads, it is assumed that the materials hardens functionally with the volumetric engineering strain, ε_V. In this formulation, an arbitrary form of this hardening function, $\bar{\sigma}(\varepsilon_V)$ is assumed although in the original work [45],

$$\bar{\sigma}(\varepsilon_V) = ae^{-b\varepsilon_V}, \tag{5.154}$$

with a and b as material constants, was used.

With these assumptions, the yield function of the ith principal stress, f^i, may be written as,

$$f^i = \begin{cases} \sigma^i - \tau, & \sigma^i \geq 0 \\ -\sigma^i - \bar{\sigma}(\varepsilon_V) & \sigma^i < 0 \end{cases}. \tag{5.155}$$

where τ is the isotropic tensile strength of the material.

Similar to the rigid polyurethane foam model [45], the flow rule is defined as:

$$d^p_{ij} = \dot{\gamma}^1 P^1_{ijkl}\sigma_{kl} + \dot{\gamma}^2 P^2_{ijkl}\sigma_{kl} + \dot{\gamma}^3 P^3_{ijkl}\sigma_{kl} \tag{5.156}$$

with $\dot{\gamma}^i$ being the magnitude of the ith plastic strain increment and P^r_{ijkl} is the fourth-order principal projection operator defined as,

$$P^r_{ijkl} = n^r_i n^r_j n^r_k n^r_l \tag{5.157}$$

in which n^r_i is the corresponding direction vector of principal stress, σ^r. With this definition,

$$\sigma^r = \sigma_{ij} P^r_{ijkl} \sigma_{kl}. \tag{5.158}$$

- See Section 5.1.5 for more information on elastic constants input.
- The yield function in compression is defined with the `YIELD FUNCTION` command line.
- The tensile strength is give by the `TENSION` command line.

Output variables available for this model are listed in Table 5-24.

More information on the model can be found in the report by Neilsen, et. al. [45].

Table 5-24. State Variables for WIRE MESH Model (Section 5.2.27)

Name	Description
`EVOL`	engineering volumetric strain
`YIELD`	current yield strength in compression

5.2.28. Orthotropic Crush Model

```
BEGIN PARAMETERS FOR MODEL ORTHOTROPIC_CRUSH
  #
  # Elastic constants - Post lock-up
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  # Orthotropic Elastic properties - Pre-Crush
  #
  EX             = <real> E11
  EY             = <real> E22
  EZ             = <real> E33
  GXY            = <real> G12
  GYZ            = <real> G23
  GZX            = <real> G31
  #
  # Crush properties
  #
  CRUSH XX       = <string> σ̂11(Jc)
  CRUSH YY       = <string> σ̂22(Jc)
  CRUSH ZZ       = <string> σ̂33(Jc)
  CRUSH XY       = <string> σ̂12(Jc)
  CRUSH YZ       = <string> σ̂23(Jc)
  CRUSH ZX       = <string> σ̂31(Jc)
  VMIN           = <real> Vmin
  #
  # Post lock-up yield properties
  #
  YIELD STRESS   = <real> σy
  #
END [PARAMETERS FOR MODEL ORTHOTROPIC_CRUSH]
```

The orthotropic crush model in LAMÉ is designed to model the energy absorbing capability of crushable orthotropic materials, e.g. aluminum honeycomb, and is empirically based. The formulation follows that used for metallic honeycomb materials in LS-DYNA [46]. Three response regimes are assumed for this material: (*i*) orthotropic elastic, (*ii*) crush, and (*iii*) complete compaction (fully crushed). During the elastic regime, the model exhibits the response of an elastic, orthotropic material with *all Poisson's ratio equal to zero*. After full compaction, the response is taken to be that of an isotropic, perfectly plastic material and the response between these two stages is tailored to smoothly transition between the two extremes. Crushing, incorporating both nonlinear elastic and plastic-like behaviors, is taken to begin as soon as volumetric contraction is noted ($J = \det\left(F_{ij}\right) < 1$). As such, the purely elastic response is

primarily seen during cyclic loadings in which the material is unloaded. An internal state variable, J_c, is introduced to track the crushed state of the material and is defined as the minimum J over the entire deformation history such that,

$$J_c = \min_{t>0} \left[J(t) \right]. \tag{5.159}$$

The crushing process manifests through two distinct behaviors: (*i*) the elastic properties scale linearly with the crush state from the initial orthotropic state to the of the final isotropic completely compacted material; and (*ii*) a plastic-like response is observed associated with corresponding crush curves (analogous to hardening curves).

Before complete compaction, the incremental constitutive relation may be written in terms of the rate of deformation tensor, D_{ij}, as,

$$\begin{Bmatrix} \overset{\circ}{\sigma}_{11} \\ \overset{\circ}{\sigma}_{22} \\ \overset{\circ}{\sigma}_{33} \\ \overset{\circ}{\sigma}_{12} \\ \overset{\circ}{\sigma}_{23} \\ \overset{\circ}{\sigma}_{31} \end{Bmatrix} = \begin{bmatrix} \hat{E}_{11} & 0 & 0 & 0 & 0 & 0 \\ 0 & \hat{E}_{22} & 0 & 0 & 0 & 0 \\ 0 & 0 & \hat{E}_{33} & 0 & 0 & 0 \\ 0 & 0 & 0 & 2\hat{G}_{12} & 0 & 0 \\ 0 & 0 & 0 & 0 & 2\hat{G}_{23} & 0 \\ 0 & 0 & 0 & 0 & 0 & 2\hat{G}_{31} \end{bmatrix} \begin{Bmatrix} D_{11} \\ D_{22} \\ D_{33} \\ D_{12} \\ D_{23} \\ D_{31} \end{Bmatrix} \tag{5.160}$$

where $\hat{E}_{11}$, $\hat{E}_{22}$, and $\hat{E}_{33}$ are the normal stiffness and $\hat{G}_{12}$, $\hat{G}_{23}$, and $\hat{G}_{31}$ are the shear stiffness. A clear decoupling between the different directional components is evident in (5.160). All six stiffness components are assumed to be functions of the current compaction level which may be defined as $1 - J_c$ and the evolution of these terms is responsible for crushing behavior (*i*) alluded to previously.

The functional forms of the stiffness are given by,

$$\begin{aligned} \hat{E}_\beta &= E_\beta + \alpha \left(E - E_\beta \right) \qquad \beta = 11,\ 22,\ 33 \\ \hat{G}_\gamma &= G_\gamma + \alpha \left(G - G_\gamma \right) \qquad \gamma = 12,\ 23,\ 31, \end{aligned} \tag{5.161}$$

where E and G are the Young's and shear moduli, respectively, of the fully compacted material while E_β and G_γ are the input orthotropic elastic stiffness components of the virgin, uncompacted material. It is assumed that these stiffness vary linearly between the pre- and post-compacted material such that,

$$\alpha = \frac{(1 - J_c)}{V_{min}}, \tag{5.162}$$

with V_{min} being the minimum relative volume (or maximum compaction).

With respect to the second behavior observed during crushing, a plastic-like response governed by crush curves is observed. Given the decoupling between the different stresses and deformations, a crush curve needs to be defined for each of the six normal and shear stresses. An example of such a curve is presented in Figure 5-10, and three distinct regions are evident. Initially, at low compaction levels, a plateau is observed. This plateau is essentially an initial crush strength and prior to this stress level all nonlinear deformations associated with material compaction manifest through changes in the respective moduli. When the stress reaches the specified levels, however, the curves play a role analogous to the hardening curve and the material stress follows the curve. Physically, the plateau is associated with crushing the internal honeycomb or foam structure of the material. As the material approaches full compaction and microstructural contact effects become important, a sharp rise in the stress is noted (see $\approx 0.6 \leq 1 - J_c \leq 0.7 = V_{min}$ in Figure 5-10). After complete compaction another plateau corresponding to perfect plasticity is evident.

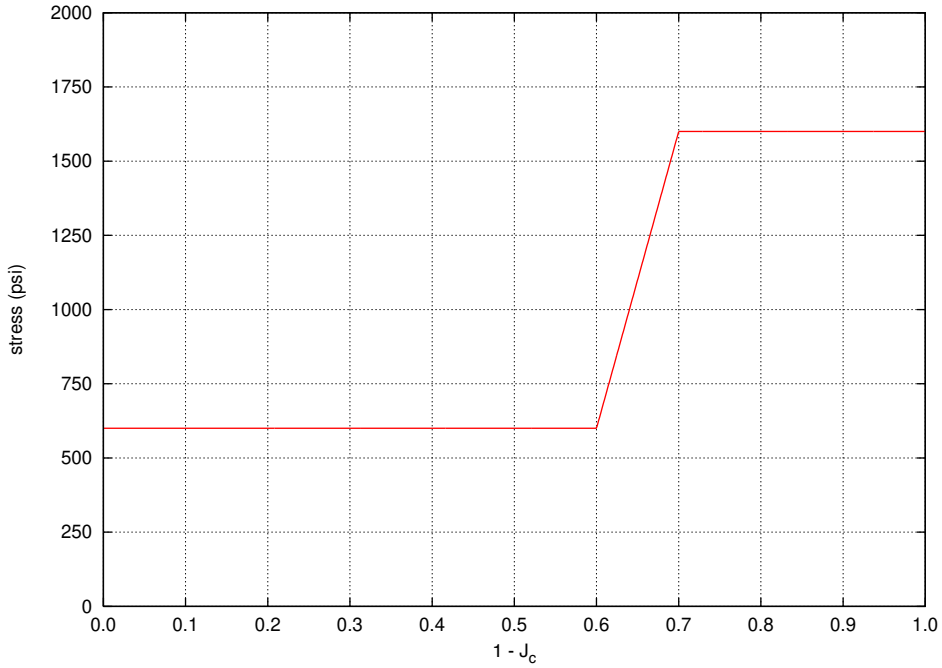


Figure 5-10. An example of an input crush curve for an aluminum honeycomb.

Above some value of compaction ($1 - J_c = V_{min}$), the material will be fully compacted and behave as an elastic, perfectly plastic material. The fully compacted response is given by the Young's modulus, E, Poisson's ratio, ν, and the yield stress, σ_y. Details of this response may be found in previous sections on the various elastic-plastic models (e.g. Section 5.2.5).

In the above command blocks:

- The `EX`, `EY`, `EZ`, `GXY`, `GYZ`, and `GZX` command lines define, respectively, the initial, pre-crush directional moduli E_{xx}, E_{yy}, E_{zz}, G_{xy}, G_{yz}, and G_{zx} from (5.160).
- `CRUSH` `XX`, `YY`, `ZZ`, `XY`, `YZ`, and `ZX` inputs require the name of a function defined via a `FUNCTION` command line in the SIERRA scope. These functions describe the directional

crush characteristics of the material and give the current stress value (in a direction) as a function of the current compaction $(1 - J_c)$.

- The command `VMIN` defines the minimum relative volume of the material that is achieved when the material is completely crushed. This parameter may also be considered as the maximum compaction.
- The elastic constant commands refer to the *post lock-up, fully compacted* isotropic response of the material. See Section 5.1.5 for more information on elastic constants input.
- `YIELD STRESS` refers to the plateau stress of the material after lock-up when the response is perfectly plastic.

Output variables available for this model are listed in Table 5-25. For information about the orthotropic crush model, consult [46].

Table 5-25. State Variables for ORTHOTROPIC CRUSH Model (Section 5.2.28)

Name	Description
`CRUSH`	current (unrecoverable) compaction/relative volume

5.2.29. Orthotropic Rate Model

```
BEGIN PARAMETERS FOR MODEL ORTHOTROPIC_RATE
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  YIELD STRESS = <real> σ_y
  #
  MODULUS TTTT = <real> E^0_TTTT
  MODULUS TTLL = <real> E^0_TTLL
  MODULUS TTWW = <real> E^0_TTWW
  MODULUS LLLL = <real  E^0_LLLL
  MODULUS LLWW = <real> E^0_LLWW
  MODULUS WWWW = <real> E^0_WWWW
  MODULUS TLTL = <real> G^0_TLTL
  MODULUS LWLW = <real> G^0_LWLW
  MODULUS WTWT = <real> G^0_WTWT
  #
  TX = <real> t̂_x
  TY = <real> t̂_y
  TZ = <real> t̂_z
  LX = <real> l̂_x
  LY = <real> l̂_y
  LZ = <real> l̂_z
  #
  MODULUS FUNCTION = <string> f_E(α)
  RATE FUNCTION    = <string> h(ε̇)
  #
  T FUNCTION  = <string> f_TT(α)
  L FUNCTION  = <string> f_LL(α)
  W FUNCTION  = <string> f_WW(α)
  TL FUNCTION = <string> f_TL(α)
  LW FUNCTION = <string> f_LW(α)
  WT FUNCTION = <string> f_WT(α)
END [PARAMETERS FOR MODEL ORTHOTROPIC_RATE]
```

The orthotropic rate model extends the functionality of the orthotropic crush constitutive model described in Section 5.2.28. The orthotropic rate model has been developed to describe the behavior of an aluminum honeycomb subjected to large deformation. The orthotropic rate model, like the original orthotropic crush model, has six independent yield functions that evolve with

volume strain. Unlike the orthotropic crush model, the orthotropic rate model has yield functions that also depend on strain rate. The orthotropic rate model also uses an orthotropic elasticity tensor with nine elastic moduli in place of the orthotropic elasticity tensor with six elastic moduli used in the orthotropic crush model.

A honeycomb orientation capability is included with the orthotropic rate model, allowing users to prescribe initial honeycomb orientations that are not aligned with the original global coordinate system. The three material directions are defined as follows: The T-direction is usually associated with the generator axis for the honeycomb, the L-direction is in the ribbon plane (defined by flat sheets used in reinforced honeycomb) orthogonal to the generator axis, and the W-direction is orthogonal to the ribbon plane.

In the above command blocks:

- For the `ORTHOTROPIC_RATE` model, only `YOUNGS MODULUS` needs to be defined; this is the Young's modulus in the fully compacted state. If two other elastic constants are provided, they will be used to define the fully compacted Young's modulus. See Section 5.1.5 for more information on elastic constants input.
- The `YIELD STRESS` command line defines the yield stress of the fully compacted honeycomb.
- The nine elastic moduli for the orthotropic, uncompacted honeycomb are defined with respect to each material direction using the `MODULUS TTTT`, `MODULUS TTLL`, `MODULUS TTWW`, `MODULUS LLLL`, `MODULUS LLWW`, `MODULUS WWWW`, `MODULUS TLTL`, `MODULUS LWLW`, and `MODULUS WTWT` command lines.
- The components of vectors defining the T- and L-directions of the honeycomb are specified using the `TX`, `TY`, and `TZ`, and `LX`, `LY`, and `LZ` command lines, respectively. The vector component values `tx`, `ty`, and `tz` define the orientation of the honeycomb's T-direction (generator axis), while `lx`, `ly`, and `lz` define the orientation of the L-direction. The vectors T and L are defined in the global coordinate system and *must* be orthogonal.
- The function describing the variation in moduli with compaction is given by the `MODULUS FUNCTION` command line. The moduli vary continuously from their initial orthotropic values to isotropic values when full compaction is obtained.
- The function describing the change in strength with strain rate is given by the `RATE FUNCTION` command line. Note that all strengths are scaled with the multiplier obtained from this function.
- The function describing the T-normal strength of the honeycomb as a function of compressive volumetric strain is given by the `T FUNCTION` command line.
- The function describing the L-normal strength of the honeycomb as a function of compressive volumetric strain is given by the `L FUNCTION` command line.
- The function describing the W-normal strength of the honeycomb as a function of compressive volumetric strain is given by the `W FUNCTION` command line.
- The function describing the TL-normal strength of the honeycomb as a function of compressive volumetric strain is given by the `TL FUNCTION` command line.

- The function describing the LW-normal strength of the honeycomb as a function of compressive volumetric strain is given by the `LW FUNCTION` command line.
- The function describing the WT-normal strength of the honeycomb as a function of compressive volumetric strain is given by the `WT FUNCTION` command line.

Note that several of the command lines in this command block reference functions. These functions must be defined in the SIERRA scope.

Output variables for this model are listed in Table 5-26.

Table 5-26. State variables for ORTHOTROPIC RATE Model (Section 5.2.29)

Index	Name	Variable Description
1	`CRUSH`	minimum volume ratio, crush is unrecoverable ($\hat{\varepsilon}_V$)



Warning: Strongly rate-dependent models may fare poorly in implicit quasistatic solution. In implicit analyses the rate term used to evaluate the current load step is the rate seen by the model in the previous load step. This may cause the solution to oscillate between high- and low-rate equilibrium states from step to step.

5.2.30. Plane Stress Rate Plasticity

```
BEGIN PARAMETERS FOR MODEL PLANE_STRESS_RATE_PLASTICITY
   #
   # Elastic constants
   #
   YOUNGS MODULUS = <real> E
   POISSONS RATIO = <real> ν
   SHEAR MODULUS  = <real> G
   BULK MODULUS   = <real> K
   LAMBDA         = <real> λ
   TWO MU         = <real> 2μ
   #
   # Optional parameters related to inelastic correction criteria
   #
   TOLERANCE          = <real> tolerance (1.0e-10)
   MAX_INEL_CORR_ITER = <int> maximum_correction_iterations (100)
   MAX_LS_CORR_ITER   = <int> maximum_line_search_cutbacks  (20)
   #
   USER RATE DEPENDENCE = 0|1(0)
   YIELD STRESS         = <real> σ_y
   #
   FORMULATION          = <int> formulation (1)
   #
   # Input Options for USER RATE DEPENDENCE = 0
   #
   # linear hardening
   HARDENING MODULUS = <real> hardening_modulus
   # power law hardening
   HARDENING CONSTANT = <real> hardening_constant
   HARDENING EXPONENT = <real> hardening_exponent (0.5)
   # multilinear hardening
   HARDENING FUNCTION = <string> K(ε̄^p)
   #
   # Rate dependence
   #
   # Johnson-Cook rate dependence
   RATE CONSTANT = <real> C
   # multilinear rate dependence
   RATE FUNCTION = <string> rate_function_name
   #
   # Input Options for USER RATE DEPENDENCE = 1
   #
   # rate-dependent yield
   YIELD STRAIN RATES = <real_list> yield_strain_rates
   YIELD CURVES       = <string_list> yield_function_names
   # rate-dependent damage
```

```
    FRACTURE STRAIN RATES = <real_list> fracture_strain_rates
    FRACTURE CURVES       = <string_list> fracture_function_names
    DECAY COEFFICIENT     = <real> C_1 (1.0)
    #
    # Thermal softening commands (Johnson-Cook)
    INITIAL TEMPERATURE   = <real> θ(t = 0)
    MELT TEMPERATURE      = <real> θ_melt
    REFERENCE TEMPERATURE = <real> θ_ref
    THERMAL EXPONENT      = <real> M
    #
END [PARAMETERS FOR MODEL PLANE_STRESS_RATE_PLASTICITY]
```

The plane stress rate plasticity model is the plane stress formulation of a J_2 plasticity model given by Simo and Taylor [47] (and described again in Simo and Hughes [6]) extended to include rate-dependent hardening and a failure model for use with shell elements.

Like other plasticity models, the components of the objective stress rate, $\overset{\circ}{\sigma}_{ij}$, are written as,

$$\overset{\circ}{\sigma}_{ij} = \mathbb{C}_{ijkl} D^{\mathrm{e}}_{kl} \tag{5.163}$$

where $\mathbb{C}_{ijkl}$ are the components of the fourth-order, isotropic elasticity tensor and D^{e}_{ij} are the components of the elastic part of the total rate of deformation tensor. An additive split of the total rate of deformation tensor into elastic and plastic contributions is assumed such that,

$$D_{ij} = D^{\mathrm{e}}_{ij} + D^{\mathrm{p}}_{ij}. \tag{5.164}$$

The plane stress formulation recasts the three-dimensional problem into a constrained subspace with plane stress conditions acting as the constraints. To do this, the plane stress rate plasticity model follows the approach of Simo and Taylor [47] to enforce $\sigma_{13} = \sigma_{23} = \sigma_{33} = 0$ and related conditions.

For the plasticity portion of the model, the formulation of Simo and Taylor [47] is used[1] in which a traditional three-dimensional J_2 plasticity model is recast in reduced subspace. To do this, it is recalled that in three-dimensions the von Mises effective stress, ϕ, is written,

$$\phi^2 = \frac{3}{2} s_{ij} s_{ij}, \tag{5.165}$$

with $s_{ij} = \sigma_{ij} - (1/3)\sigma_{kk}\delta_{ij}$ the deviatoric stress. To write an equivalent expression in the reduced subspace, the vector, $\underline{\sigma}$, and matrix, $\underline{\underline{P}}$, are introduced as[2],

[1] In the work of Simo and Taylor [47] (and later Simo and Hughes [6]) hardening is assumed to be rate and temperature independent. Here, such terms are included but do not materially change the formulation. Similarly, the earlier works also introduce kinematic hardening which is not used in the current model.

[2] Note, here the $\underline{x}$ and $\underline{\underline{X}}$ notations are introduced for vector and matrix objects, respectively, to clearly distinguish that these variables are *not* tensors. This results from operating in the constrained stress subspace and means that these terms do not have properties of a tensor and act on each other as traditional matrices and vectors.

$$\underline{\sigma} = \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{12} \end{bmatrix}, \qquad ; \qquad \underline{\underline{\bar{P}}} = \frac{1}{3}\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}, \tag{5.166}$$

such that,

$$\underline{s} = \begin{bmatrix} s_{11} \\ s_{22} \\ s_{12} \end{bmatrix} = \underline{\underline{\bar{P}}}\,\underline{\sigma}. \tag{5.167}$$

In the reduced plane-stress subspace, an alternative effective stress, $\bar{\phi}$, is given as,

$$\bar{\phi}^2 = \underline{\sigma}^T \underline{\underline{P}}\,\underline{\sigma} = \frac{2}{3}\phi^2, \tag{5.168}$$

where

$$\underline{\underline{P}} = \frac{1}{3}\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & 0 \\ 0 & 0 & 6 \end{bmatrix}, \tag{5.169}$$

in which $\underline{\underline{P}}$ and $\underline{\underline{\bar{P}}}$ differ by a two in the shear term to reflect Voigt corrections.

A yield function, f, is introduced as,

$$f = \bar{\phi}^2 - R^2, \tag{5.170}$$

with,

$$R = \sqrt{\frac{2}{3}}\bar{\sigma}\left(\bar{\varepsilon}^p, \dot{\bar{\varepsilon}}^p, \theta\right), \tag{5.171}$$

where $\bar{\varepsilon}^p$ and $\dot{\bar{\varepsilon}}^p$ are the equivalent plastic strain (isotropic hardening variable) and its rate, respectively. Various hardening options may be used with this model. In general, the current flow stress is written as,

$$\bar{\sigma}\left(\bar{\varepsilon}^p, \dot{\bar{\varepsilon}}^p, \theta\right) = \left(\sigma_y + K\left(\bar{\varepsilon}^p\right)\right)\hat{\sigma}\left(\dot{\bar{\varepsilon}}^p\right)\left(1 - \left(\frac{\theta - \theta_{\text{ref}}}{\theta_{\text{melt}} - \theta_{\text{ref}}}\right)^M\right), \tag{5.172}$$

in which σ_y is the original yield stress, K is the isotropic hardening function that may take linear, power-law, or multilinear form, $\hat{\sigma}$ the rate multiplier whose specification will be defined later, and the right-most term is the Johnson-Cook temperature dependence term that may be optionally used to give temperature dependence of the flow-stress.

To complete the theoretical formulation, the flow rules are specified as,

$$d\underline{\varepsilon}^p = \lambda \underline{\underline{P}}\, \underline{\sigma}, \tag{5.173}$$

$$d\bar{\varepsilon}^p = \dot{\bar{\varepsilon}}^p \Delta t = \lambda \sqrt{\frac{2}{3}} \phi, \tag{5.174}$$

where λ is the consistency multiplier enforcing $f = 0$ during plastic deformation and $d\underline{\varepsilon}^p$ is the plastic strain increment in the constrained subspace. It is emphasized here that the yield surface described in (5.170) is *not* homogeneous of degree one like in other three-dimensional formulations presented in this manual. As such, the consistency multiplier and equivalent plastic strain increment are not equivalent. As an example of this, by consideration of the preceding relations, it is apparent that λ has units of one over stress.

The specification of the rate dependence, $\hat{\sigma}$, is important as it enables the consideration of two different model responses. These behaviors are controlled via the `USER RATE DEPENDENCE` command. If this input parameter is zero, then either an analytical or user-defined rate-dependence may be given. Importantly, in this case failure is *not* modeled. For the analytical case, the Johnson-Cook [18, 17] rate-multiplier is used such that,

$$\hat{\sigma}\left(\dot{\bar{\varepsilon}}^p\right) = \begin{cases} 1 + C \ln\left(\frac{\dot{\bar{\varepsilon}}^p}{\dot{\bar{\varepsilon}}_0}\right) & \dot{\bar{\varepsilon}}^p > \dot{\bar{\varepsilon}}_0 \\ 1 & \dot{\bar{\varepsilon}}^p \leq \dot{\bar{\varepsilon}}_0 \end{cases}, \tag{5.175}$$

with C being the rate dependence multiplier and $\dot{\bar{\varepsilon}}_0$ is a reference rate. Note, while other models allow user specification of the reference rate, the plane stress rate plasticity model uses the value set in the original work of Johnson-Cook [18] such that $\dot{\bar{\varepsilon}}_0 = 1\text{s}^{-1}$. Alternatively, a user function may be specified for the rate multiplier, $\hat{\sigma}$.

If `USER RATE DEPENDENCE` is set to one, both rate dependence and failure may be modeled. With respect to the rate dependence, (5.172) is rewritten,

$$\bar{\sigma}\left(\bar{\varepsilon}^p, \dot{\bar{\varepsilon}}^p, \theta\right) = \tilde{\sigma}\left(\bar{\varepsilon}^p, \dot{\bar{\varepsilon}}^p\right) \left(1 - \left(\frac{\theta - \theta_{\text{ref}}}{\theta_{\text{melt}} - \theta_{\text{ref}}}\right)^M\right), \tag{5.176}$$

in which both isotropic hardening and rate dependence are described via definition of $\tilde{\sigma}$. In this case, $\tilde{\sigma}$ can*not* be specified through analytical expressions and must instead be given as a series of isotropic hardening curves; each at a different strain rate. For rates not explicitly given, interpolation is performed between relevant curves. Note, no extrapolation is performed with respect to the rates. If a rate is determined outside any specified curves, the hardening is calculated with respect to the bounding curve.

For failure, a failure parameter, α, is calculated as

$$\alpha = \int_0^t \frac{1}{\varepsilon_f\left(\eta, \dot{\bar{\varepsilon}}^p\right)} \dot{\bar{\varepsilon}}^p dt = \sum_{t_0}^{t} \frac{d\bar{\varepsilon}^p}{\varepsilon_f\left(\eta, \dot{\bar{\varepsilon}}^p\right)}, \tag{5.177}$$

in which the summation is used to imply the discrete calculation of the damage variable over a series of loadsteps and ε_f is the rate and triaxiality, η, dependent failure strain. The failure strain, ε_f, is specified in a fashion similar to $\tilde{\sigma}$. Specifically, a series of triaxiality dependent functions are defined each at a given strain rate. Interpolation is used at rates between those specified. Extrapolation outside the defined bounds is not done and the extremum curves are instead used.

The onset of damage is assumed to occur when $\alpha = 1$ and the current failure strain is taken to be the critical one such that $\varepsilon_f^{cr} = \varepsilon_f(t = t_{cr})$ with t_{cr} being the time at which $\alpha = 1$. Subsequent damage calculation is performed via,

$$\alpha(t > t_{cr}) = \int_0^{t_{cr}} \frac{1}{\varepsilon_f\left(\eta, \dot{\bar{\varepsilon}}^p\right)} \dot{\bar{\varepsilon}}^p dt + \int_{t_{cr}}^{t} \frac{1}{\varepsilon_f^{cr}} \dot{\bar{\varepsilon}}^p dt. \tag{5.178}$$

After the critical failure parameter has been reached, an exponential decay relation is used to decrease the strength of the material. In this fashion, a decay relation of the form,

$$\bar{\sigma} = \bar{\sigma} e^{C_1(1-\alpha)}, \tag{5.179}$$

is used in which C_1 is the decay coefficient.

For more information about the plane stress rate plasticity model, consult [47, 6].

In the command blocks that define the Plane Stress Rate Plasticity model:

- See Section 5.1.5 for more information on elastic constants input.
- `TOLERANCE` specifies the numerical value used for assessing convergence of the plastic correction routine.
- Optionally, the user can specify the maximum number of inelastic correction iterations for the plasticity inelastic correction routines. By default this value is 100.
- Optionally, the user can specify the maximum number of line search cutbacks in the plasticity correction routine. By default this value is 20 and only impacts the formulation not equal to 0 (plastic strain rate) case.
- The reference nominal yield stress, $\bar{\sigma}$, is defined with the `YIELD STRESS` command line.
- The formulation parameter defines whether the total strain rate (formulation = 0) or equivalent plastic strain rate (anything else) is used for calculating rate dependence.
- `INITIAL TEMPERATURE` defines the initial temperature at $t = 0$.
- `MELT TEMPERATURE` defines T_{melt} in (5.172).
- `REFERENCE TEMPERATURE` defines T_{ref} in (5.172).
- `THERMAL EXPONENT` defines M in (5.172).
- The `USER RATE DEPENDENCE` is used to control the way hardening may be specified and whether or not failure is calculated.
  - For `USER RATE DEPENDENCE = 0`, plastic hardening may be specified as linear,

power-law, OR multilinear. Failure cannot be used with `USER RATE DEPENDENCE = 0`:

  * For linear hardening, a non-zero `HARDENING MODULUS` should be specified. Do not give if using power-law or multilinear hardening.
  * For power-law hardening, the `HARDENING CONSTANT` should be specified. Optionally, the `HARDENING EXPONENT` parameter should be specified if the default value (0.5) is not to be used. Do not give for linear or multilinear hardening.
  * For multilinear hardening, a function name should be given for `HARDENING FUNCTION` command. Do not specify for linear or power-law hardening.
  * For Johnson-Cook rate-dependence, a rate constant must be specified via the `RATE CONSTANT` command. Do not use if using functionally specified rate-dependence.
  * For functionally defined rate-dependence, a function name should be given via the `RATE FUNCTION` command. Do not specify if using Johnson-Cook type rate dependence.

- For `USER RATE DEPENDENCE = 1`, plastic hardening is specified through a series of user functions. Failure can be modeled with `USER RATE DEPENDENCE = 1`:
  * Rate-dependent plastic hardening is specified jointly via the `YIELD STRAIN RATES` and `YIELD CURVES` commands. `YIELD STRAIN RATES` is a list of strain rates corresponding one-to-one to functions specified in the `YIELD CURVES` list of strings giving user-defined function names. Each `YIELD CURVES` function should give the plastic isotropic plastic hardening curve at the corresponding rate given in `YIELD STRAIN RATES`.
  * Failure strains used in calculating damage are specified via the `FRACTURE STRAIN RATES` and `FRACTURE CURVES` commands. Similarly to the plastic hardening, a list of strain rates should be given with the `FRACTURE STRAIN RATES` input. Each rate should correspond one-to-one with a user function listed via `FRACTURE CURVES`. These functions are specified as a function of triaxiality and should give the failure strain at the specified rate.
  * The decay coefficient, C_1, that controls the exponential decay of the yield stress related to failure should be specified via the `DECAY COEFFICIENT` command.

Output variables available for this model are listed in Table 5-27.



Warning: Strongly rate-dependent models may fare poorly in implicit quasistatic solution. In implicit analyses the rate term used to evaluate the current load step is the rate seen by the model in the previous load step. This may cause the solution to oscillate between high- and low-rate equilibrium states from step to step.

Table 5-27. State Variables for PLANE STRESS RATE PLASTICITY Model (Section 5.2.30)

Name	Description
EQPS	equivalent plastic strain, $\bar{\varepsilon}^p$
EQDOT	equivalent plastic strain rate, $\dot{\bar{\varepsilon}}^p$
SEFF	effective stress, ϕ

5.2.31. Incompressible Solid Model

```
BEGIN PARAMETERS FOR MODEL INCOMPRESSIBLE_SOLID
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  #
  #
  K SCALING = <real>k_scaling
  2G SCALING = <real>2g_scaling
  TARGET E = <real>target_e
  MAX POISSONS RATIO = <real>max_poissons_ratio
  REFERENCE STRAIN = <real>reference_strain
  SCALING FUNCTION = <string>scaling_function_name
END [PARAMETERS FOR MODEL INCOMPRESSIBLE_SOLID]
```

The incompressible solid model is a variation of the elastic model and can be used in both Presto and Adagio. In Adagio, the incompressible solid model is used with the control-stiffness option in the multilevel solver. The control-stiffness option is implemented via the `CONTROL STIFFNESS` command block and is discussed in Chapter 4. The model is used to model nearly incompressible materials where Poisson's ratio, ν, ≈ 0.5. In the course of solving a series of model problems in Adagio, the material response from this model incorporates scaling the bulk and/or shear behaviors to yield a material response that is more amenable to solution using Adagio's conjugate gradient solver. The final material behavior that is calculated corresponds to the actual moduli that are specified. When this model is used in Presto, the material scalings are ignored, and the model behaves like a linear elastic model.

In the above command blocks, the following definitions are applicable. Usage requirements are identified both in this list of definitions and in the discussion that follows the list.

- See Section 5.1.5 for more information on elastic constants input.
- The following material-scaling command lines are used only in Adagio and are optional as described below:
  - The nominal bulk scaling is defined with the `K SCALING` command line.
  - The nominal shear scaling is defined with the `2G SCALING` command line.
  - The target Young's modulus is defined with the `TARGET E` command line.
  - The maximum Poisson's ratio is defined with the `MAX POISSONS RATIO` command line.
  - The reference strain is defined with the `REFERENCE STRAIN` command line.
  - The `SCALING FUNCTION` command line references the name of a function defined in

a FUNCTION command line in the SIERRA scope that describes the time dependent scaling to be applied.

As noted previously, only two of the elastic constants are required to define the unscaled material response. This requirement applies to use of the incompressible solid model in Presto and in Adagio. Further, all the material-scaling command lines are only used in Adagio.

Several options exist for defining the bulk and/or shear scalings that can be used with the multilevel solver in Adagio.

- Option 1: You can provide the scalings directly by including both of the K SCALING and 2G SCALING command lines or either of them. When both command lines are input, the user-specified values for their parameters will be used. If only the K SCALING command line is input, the bulk scaling is as specified in the k_scaling parameter, and the value of the shear scaling parameter, 2g_scaling, is set to 1.0. On the other hand, if only the 2G SCALING command line is input, then the shear scaling is as specified in the 2g_scaling parameter, but the value of the bulk-scaling parameter, k_scaling, is not set to 1.0. Instead, the bulk scaling is determined by computing a scaled bulk modulus from the scaled shear modulus and a (scaled) Poisson's ratio of 0.3. Then, the bulk scaling is determined as the ratio of the scaled bulk modulus to the actual bulk modulus.
- Option 2: You can specify either or both of the TARGET E and MAX POISSONS RATIO command lines to define the scalings. If only the TARGET E command line is included, the bulk and shear scalings are computed by first finding scaled moduli using the value of the target_e parameter along with a (scaled) Poisson's ratio of 0.3. The bulk and shear scalings are then determined as the ratio of the appropriate scaled to unscaled modulus. If only the MAX POISSONS RATIO command line is included, the shear scaling is set to 1.0, and the bulk scaling is computed by first calculating a scaled bulk modulus from the actual shear modulus and the value of the max_poissons_ratio parameter. The bulk scaling is then calculated as the ratio of the scaled bulk modulus to the actual bulk modulus. If both the TARGET E and MAX POISSONS RATIO command lines are included, the bulk scaling (and shear scaling) is determined from the ratio of the bulk scaled modulus (and shear scaled modulus) computed using the values of the target_e and max_poissons_ratio parameters to the unscaled bulk (and shear) modulus.
- Option 3: You can choose not to include any of the K SCALING, 2G SCALING, TARGET E, and MAX POISSONS RATIO command lines. In such case, the shear scaling is set to 1.0, and the bulk scaling is computed as the ratio of the scaled bulk modulus coming from the real shear modulus and a (scaled) Poisson's ratio of 0.3 to the actual bulk modulus.

The function referenced by the value of the parameter scaling_function_name in the SCALING FUNCTION command line can be used to modify the bulk and shear scalings in solution time. The actual scalings used are computed by taking the scalings specified by the parameter values in the K SCALING, 2G SCALING, TARGET E, and MAX POISSONS RATIO command lines and multiplying them by the function value at the specified solution time. If the SCALING FUNCTION command line is not included, the bulk and shear scalings are fixed in time.

The REFERENCE STRAIN command line supplies a value for the reference strain that is used to create a normalized material constraint violation based on strains. Specifying a reference strain

implies the use of strains for measuring the material constraint violation (or part of the control-stiffness error in Adagio). Otherwise, the material constraint violation is determined by using the change in the scaled stress response over the current model problem.

5.2.32. Universal Polymer Model

The UPM model is commonly used in one of two ways. The most general use case is portrayed in full in the following syntax in which the user specifies both Prony series explicitly. That is, the user specifies all Prony relaxation times (τ) and weights for both the thermal/volumetric (f_v) and shear (f_s) relaxation functions. Note that in the UPM model, only a single set of Prony relaxation times can be specified and acts as the basis for both relaxation spectra. In other words, a single set of relaxation times is specified, and both functions use their own (distinct) weights.

Default parameters are not set. Any system of units can be used with the model. There are no internal units assumptions.

```
BEGIN PARAMETERS FOR MODEL UNIVERSAL_POLYMER
  #
  # Elastic constants: These Should be Set to the Glassy Moduli
  #                    for robustness considerations
  #
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  #
  ## Reference Temperature and Material CLOCK Parameters
  #
  REFERENCE TEMPERATURE   = <real> θREF # Temperature
  STRESS FREE TEMPERATURE = <real> θSF  # Temperature
  #
  WLF C1   = <real> Ĉ1
  WLF C2   = <real> Ĉ2 # Temperature
  CLOCK C3 = <real> C3 # Temperature
  CLOCK C4 = <real> C4 # Temperature
  #
  ## Glassy and Rubbery Moduli
  #  and CTE Definitions at the Reference Temperature
  #
  BULK GLASSY 0    = <real> KG # Units of Pressure
  BULK RUBBERY 0   = <real> K∞ # Units of Pressure
  SHEAR GLASSY 0   = <real> GG # Units of Pressure
  SHEAR RUBBERY 0  = <real> G∞ # Units of Pressure
  VOLCTE glassy 0  = <real> δG # Units of Inverse Temperature
  VOLCTE rubbery 0 = <real> δ∞ # Units of Inverse Temperature
  #
  FILLER VOL FRACTION = <real>
  #
  ## Relaxation Time Spectra Definitions
  #
  WWBETA 1 = <real> β1
  WWTAU 1  = <real> τ1 # Units of time
  WWBETA 2 = <real> β2
  WWTAU 2  = <real> τ2 # Units of time
```

```
 #
 SPECTRUM START TIME = <real>  # Units of time
 SPECTRUM END TIME   = <real>  # Units of time
 LOG TIME INCREMENT  = <real>  # Units of time
 #
 ## Direct Prony Spectra Inputs
 #
 RELAX TIME 1 = <real>  # Unit of time
 RELAX TIME 2 = <real>
 .
 RELAX TIME 30 = <real>
 #
 ## Thermal/Volumetric Relaxation Spectrum Prony Weights
 #
 F1 1 = <real>
 F1 2 = <real>
 .
 F1 30 = <real>
 #
 ## Shear Relaxation Spectrum Prony Weights
 #
 F2 1 = <real>
 F2 2 = <real>
 .
 F2 30 = <real>
END [PARAMETERS FOR MODEL UNIVERSAL_POLYMER]
```

Not all Prony spectra/weight parameter pairs (1-30) need to be specified. Only those specified will be used, and the ones not specified will be set to zero. Prony weights for each relaxation function should sum to 1.0, or the model will rescale the weights so that they do sum to one. This rescaling will change the underlying relaxation response.

When the model is used with both relaxation functions being specified directly, then the parameters: SPECTRUM START TIME, SPECTRUM END TIME, LOG TIME INCREMENT, WW TAU (1,2), and WW BETA (1,2) must be specified as 0 to avoid errors during the model property check. Note (1) is associated with the thermal/volumetric function, and (2) is associated with the shear relaxation function.

Another common usage of the UPM model is to specify the Williams-Watts (KWW) stretched exponential τ, β parameters for either or both relaxation functions (1 and/or 2) corresponding to the function $f = \exp(-(t/\tau)^{\beta})$. That is, a set of Prony weights, w_i corresponding to a specific set of Prony times, τ_i, will be found during the model property check routine. If the other relaxation function is directly specified as above, then the Prony times from the directly specified relaxation spectrum are used. In this case, the Prony weights for the relaxation function being fit to the KWW function are found through a Least-Squared Error minimization routine built into the UPM model over a discretely sampled set of times between the minimum and maximum Prony times.

When neither Prony spectrum is directly specified (both will be fit to KWW functions), then the

Prony times (for both relaxation functions) are determined from an evenly logarithmically spaced set of Prony times beginning with the SPECTRUM START TIME and ending with the SPECTRUM END TIME and spaced with the (base 10) LOG TIME INCREMENT. For each relaxation function that is fit with the UPM model to a KWW function, the WW TAU (1,2) and WW BETA (1,2) parameters must be specified. However, if the user specifies both a KWW form and the same Prony series directly, the model will error out during the property check.

There are many useful optional parameters for the UPM model that generally allow for: temperature dependence of moduli, coefficients of thermal expansion, deformation dependence of moduli, and/or alternative material clock parameter specifications. These parameters may optionally be added to the material input block, but are defaulted to 0.0:

```
### OPTIONAL parameters for the universal_polymer model
 CLOCK C1          = <real> C_1 # CLOCK Coef. 1 instead of "WLF C1"
 CLOCK C2          = <real> C_2 # CLOCK Coef. 1 instead of "WLF C2"
 BULK GLASSY 1     = <real> dK_G/dT # Pressure per Temperature
 BULK RUBBERY 1    = <real> dK_∞/dT # Pressure per Temperature
 SHEAR GLASSY 1    = <real> dG_G/dT # Pressure per Temperature
 SHEAR RUBBERY 1   = <real> dG_∞/dT # Pressure per Temperature
 VOLCTE GLASSY 1   = <real> dδ_G/dT # Inverse Temperature Squared
 VOLCTE RUBBERY 1  = <real> dδ_∞/dT # Inverse Temp. Squared
```

Finally, we note that the UPM model may be reduced to a finite deformation, linear thermoviscoelastic model by choosing $C_3 = 0$ and $C_4 = 0$. Under these conditions the material clock is only temperature (history) dependent but involves no deformation dependence. Moreover, if one wants to fix the laboratory and material time scales to be the same, then one should set WLF $C_1 = 0$.

The Universal Polymer Model (UPM) is a phenomenological, non-linear viscoelastic (NLVE) model that is, in the literature, named the Simplified Potential Energy Clock (SPEC) [48]. The UPM model is considerably simpler than the parent model, the Potential Energy Clock (PEC) model, labeled the NLVE polymer model in SIERRA, which itself is not phenomenological but requires extensive data and experience to calibrate [49].

The UPM model is suitable for modeling the finite deformation, thermal-mechanical behavior of glassy materials, both organic and inorganic. Successful usage of the model is widespread. Some examples include the modeling of amorphous, thermosetting polymers across and through the glass transition such as epoxies [50]. It is also suitable for modeling thermoplastics from within the melt state and down into the glass transition from polystyrene to polycarbonate. Finally, it has been used to represent inorganic glasses for glass-to-metal seals. The UPM model was developed for production analyses of encapsulated components. It predicts a full range of behavior including yielding, stress relaxation, volume relaxation, and physical aging.

The key physical principal behind the UPM model is that there exists a material time scale (material clock) separate from the laboratory time scale. If the material time scale is fast, such as in the rubbery state of a polymer, then the UPM model responds instantly to changes in temperature and strain such that the user would observe rate-independent behavior. However, if the material clock is slow relative to the laboratory time scale, viscoelastic memory builds with

any process, which causes acute history and thermodynamic path dependent behavior.

The model response is derived from a Helmholtz Free Energy density and takes as an input the unrotated rate of deformation, d_{ij}, the temperature at the start and end of the time step (θ_n and θ_{n+1}, and the time step, Δt. From these inputs, the hereditary integrals within the model are updated, and the unrotated Cauchy stress tensor is returned.

For the UPM model, the strain measure is approximated from the integrated unrotated rate of deformation tensor, which we label ϵ_{ij},

$$\epsilon_{ij} = \int_0^\infty \left(R_{mi} D_{mn} R_{nj}\right) ds, \quad D_{ij} = \frac{1}{2}\left(L_{ij} + L_{ji}\right), \quad F_{ij} = R_{im} U_{mj}. \tag{5.180}$$

Here, F_{ij}, R_{ij}, U_{ij}, L_{ij}, and D_{ij} are the deformation gradient, rotation, material stretch, velocity gradient, and rate of deformation tensors standard in Lagrangian continuum mechanics.

The UPM model allows the user to initiate an analysis from a stress-free temperature, θ_{sf}, that is different from the reference temperature, θ_{ref}, at which all material properties are defined. Here we briefly summarize the constitutive equations. The model is derived from a Helmholtz Free Energy, but we begin directly with the (unrotated) Cauchy Stress and refer the reader to reference [48] for more detail:

$$\begin{aligned}
\sigma_{ij} \;=\; & \left(K_G(\theta) - K_\infty(\theta)\right) \int_0^t f_v\left(t' - s'\right) \frac{dI_1}{ds} ds \delta_{ij} \\
& - \left(K_G(\theta)\delta_G(\theta) - K_\infty(\theta)\delta_\infty(\theta)\right) \int_0^t f_v\left(t' - s'\right) \frac{d\theta}{ds} ds \delta_{ij} \\
& + 2\left(G_G(\theta) - G_\infty(\theta)\right) \int_0^t f_s\left(t' - s'\right) \frac{d\left(\text{dev}\epsilon_{ij}\right)}{ds} ds \\
& + \left(K_\infty(\theta) I_1 - K_\infty(\theta)\delta_\infty(\theta)(\theta - \theta_{\text{sf}})\right)\delta_{ij} + 2G_\infty(\theta)\,\text{dev}\epsilon_{ij}.
\end{aligned} \tag{5.181}$$

The first three lines of terms in Equation 5.181 represent the time-dependent and dissipative (non-equilibrium) response of the model to volumetric, thermal, and shear deformation histories. Accordingly, K, δ, and G represent a bulk modulus, volumetric thermal expansion coefficient, and shear modulus while subscripts $_G$ or $_\infty$ denote a glassy or rubbery, respectively, properties. The last collection of terms in 5.181 furnish the time-independent rubbery (equilibrium) response. The variables in equation 5.181 are:

$$I_1 = \delta_{ij}\epsilon_{ij} = \text{tr}\epsilon_{ij}, \qquad \text{dev}\epsilon_{ij} = \epsilon_{ij} - \frac{I_1}{3}\delta_{ij}, \tag{5.182}$$

$$G_G(\theta) = G_G^{\text{ref}} + \frac{dG_G}{d\theta}(\theta - \theta_{\text{ref}}), \quad G_\infty(\theta) = G_\infty^{\text{ref}} + \frac{dG_\infty}{d\theta}(\theta - \theta_{\text{ref}}), \tag{5.183}$$

$$K_G(\theta) = K_G^{\text{ref}} + \frac{dK_G}{d\theta}(\theta - \theta_{\text{ref}}), \quad K_\infty(\theta) = K_\infty^{\text{ref}} + \frac{dK_\infty}{d\theta}(\theta - \theta_{\text{ref}}), \tag{5.184}$$

$$\delta_G(\theta) = \delta_G^{\text{ref}} + \frac{d\delta_G}{d\theta}(\theta - \theta_{\text{ref}}), \quad \delta_\infty(\theta) = \delta_\infty^{\text{ref}} + \frac{d\delta_\infty}{d\theta}(\theta - \theta_{\text{ref}}). \tag{5.185}$$

The first three terms in Equation 5.181 represent the material's viscoelastic response to changes in volume strain, temperature, and shear deformation. Two relaxation functions are used to characterize the thermal/volumetric (f_v) and shear (f_v) relaxation responses. The model assumes the thermal and volumetric relaxation responses are identical. Otherwise, f_v and f_s are typically quite different and are expressed as a Prony series [3]:

$$f_v(x) = \sum_{k=1}^{N} w^{(k)} \exp\left(-\frac{x}{\tau^{(k)}}\right), \quad f_s(x) = \sum_{l=1}^{M} w^{(l)} \exp\left(-\frac{x}{\tau^{(l)}}\right). \tag{5.186}$$

These relaxation functions describe the material's response to a suddenly applied volumetric/thermal or shear perturbation at the reference temperature where, under certain conditions, the material and laboratory time scales are equivalent. In Equation 5.181, the viscous terms (non-rubbery) involve hereditary integrals over the difference in material time from $s = 0$ to $s = t$, which is the current laboratory time. An increment in material time, dt', and the laboratory time, dt, are related through the (highly) history dependent shift factor, a, such that the difference in material time, $t' - s'$, is related to the corresponding difference in laboratory time, $t - s$ through:

$$adt' = dt, \quad t' - s' = \int_{u=s}^{u=t} \frac{du}{a(u)}. \tag{5.187}$$

If the material time scale is very slow compared to the laboratory time, then $a >> 1$, which is often the case inside and below the glass transition for typically glassy materials.

The shift factor is instantaneously defined through:

$$\log_{10} a = \frac{-C_1 N}{C_2 + N}, \tag{5.188}$$

$$\begin{aligned} N(t) = {} & \theta - \theta_{\text{ref}} - \int_0^t f_v\left(t' - s'\right) \frac{d\theta}{ds} ds \\ & + C_3 \left(I_1 - \int_0^t f_v\left(t' - s'\right) \frac{dI_1}{ds} ds \right) \\ & + C_4 \int_{u=0}^{u=t} \int_{s=0}^{s=t} \left(f_s\left(t' - s', t' - u'\right) \frac{d\left(\text{dev}\epsilon_{ij}\right)}{ds} \frac{d\left(\text{dev}\epsilon_{ij}\right)}{du} ds du \right). \end{aligned} \tag{5.189}$$

The key physics in the model comes form Equation 5.188. Temperature rise (generally) causes N to increase, and hence the material shift factor shrinks (the material time scale speeds up). Shrinking the volume generally causes the shift factor to increase as if the temperature had been decreased. Mechanistically, this feature is the manifestation of the trade-off between between mobility and free volume available to polymer chains. Finally, shear deformation can greatly speed up the material clock through the last term. This phenomenon is a direct manifestation of "deformation induced mobility", a key mechanism for glassy materials.

[3] Note: to distinguish between indices used with conventional summation convention and those related to Prony series terms, all Prony series summations shall be explicitly written with the relevant index given parenthetically in a superscript.

Since the shift factor involves hereditary integrals, even at a constant temperature and state of deformation, the material clock will change over time. Under stress-free conditions, the material will creep and densify if the model is out of equilibrium (when any viscous term is non-zero). These phenomena are the model's manifestations of physical aging, time-dependent material change without a change in composition or microstructure. C_1, C_2, C_3, and C_4 are all material constants. We note that the double relaxation function appearing in the last term takes on a slightly different form from f_s:

$$f_s(x,y) = \sum_{k=1}^{N} w^{(k)} \exp\left(-\frac{x}{\tau^{(k)}}\right) \exp\left(-\frac{y}{\tau^{(k)}}\right) \tag{5.190}$$

It is desirable to relate a special case of the model to the Williams-Landel-Ferry (WLF) form because of how time-temperature superposition fitting is typically performed. Specifically, one can show that the clock parameters, C_1 and C_2, relate to the WLF parameters, $\hat{C}_1$ and $\hat{C}_2$, through the following relationships: $\hat{C}_1 = C_1$ and $\hat{C}_2 = C_2 / \left(1 + C_3 \delta_\infty^{\text{ref}}\right)$.

For more information about the universal polymer model, consult [48].

Output variables available for this model are listed in Table 5-28. The user should always output the shift factor *aend* or $\log_{10} a$ as this variable is critical for interpreting the material behavior.

Table 5-28. State Variables for Universal Polymer Model

Name	Description
`aend`	The shift factor relating increments of material to laboratory time, $a\,dt^* = dt_{\text{lab}}$
`loga`	$\log_{10}$ of the shift factor, $\log_{10} a$
`epsxx`	xx component of the integrated unrotated rate of deformation, ϵ_{xx}
`epsyy`	yy component of the integrated unrotated rate of deformation, ϵ_{yy}
`epszz`	zz component of the integrated unrotated rate of deformation, ϵ_{zz}
`epsxy`	xy component of the integrated unrotated rate of deformation, ϵ_{xy}
`epsyz`	yz component of the integrated unrotated rate of deformation, ϵ_{yz}
`epszx`	zx component of the integrated unrotated rate of deformation, ϵ_{zx}
`effi2`	second (non-Cayley Hamilton) invariant of ϵ providing shear deformation, I_2
`if1p1-30`	volumetric hereditary integrals 1-30
`ikat1-30`	thermal hereditary integrals 1-30
`igxx1-30`	xx component shear hereditary integrals 1-30
`igyy1-30`	yy component shear hereditary integrals 1-30
`igzz1-30`	zz component shear hereditary integrals 1-30
`igxy1-30`	xy component shear hereditary integrals 1-30
`igyz1-30`	yz component shear hereditary integrals 1-30
`igzx1-30`	zx component shear hereditary integrals 1-30

5.2.33. Mooney-Rivlin Model

```
BEGIN PARAMETERS FOR MODEL MOONEY_RIVLIN
  #
  # Elastic constants, One of these elastic constants must be input.
  # The shear modulus is computed from C10 and C01.
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  #
  #
  C10 = <real>c10
  C01 = <real>c01
  C10 FUNCTION = <string>c10_function_name
  C01 FUNCTION = <string>c01_function_name
  BULK FUNCTION = <string>bulk_function_name
  THERMAL EXPANSION FUNCTION = <string>eth_function_name
  TARGET E = <real>target_e
  TARGET E FUNCTION = <string>etar_function_name
  MAX POISSONS RATIO = <real>max_poissons_ratio(0.5)
  REFERENCE STRAIN = <real>reference_strain
END [PARAMETERS FOR MODEL MOONEY_RIVLIN]
```

Mooney-Rivlin is a hyperelastic material model that is used to describe rubber. The Mooney-Rivlin model incorporates temperature-dependent material moduli and can be used in both Presto and Adagio. When the model is used in Adagio, it can be used with or without the control-stiffness option in Adagio's multilevel solver. The control-stiffness option is implemented via the `CONTROL STIFFNESS` command block and is discussed in Chapter 4. The model is used to simulate nearly incompressible materials where Poisson's ratio, ν, ≈ 0.5. In the course of solving a series of model problems in Adagio, the material response from this model incorporates scaling the bulk and/or shear behaviors to yield a material response that is more amenable to solution using Adagio's conjugate gradient solver. The final material behavior that is calculated corresponds to the actual moduli that are specified. When this model is used in Presto, the material scalings are ignored.

In the above command blocks the following command lines are required:

- The material constants C10 and C01 determine the shear behavior as defined by

$$shear\_modulus = 2\,(C_{10} + C_{01})\,. \tag{5.191}$$

  Thus the `C10` and `C01` command lines must be included in this model and the shear modulus elastic constant is not an available input.
- Since `C10` and `C01` define the shear behavior, one additional elastic constant, other than the shear modulus is required to define the unscaled bulk behavior. See Section 5.1.5 for more information on elastic constants input.

Other command line options include:

- The C10 FUNCTION command line references the name of a function defined in a FUNCTION command line in the SIERRA scope that describes the temperature dependence of the C10 material parameter. This command line is optional. If it is not present, there is no temperature dependence in the C10 parameter. See the usage discussion below.
- The C01 FUNCTION command line references the name of a function defined in a FUNCTION command line in the SIERRA scope that describes the temperature dependence of the C01 material parameter. This command line is optional. If it is not present, there is no temperature dependence in the C01 parameter. See the usage discussion below.
- The BULK FUNCTION command line references the name of a function defined in a FUNCTION command line in the SIERRA scope that describes the temperature dependence of the bulk modulus. This command line is optional. If it is not present, there is no temperature dependence in the bulk modulus. See the usage discussion below.
- The THERMAL EXPANSION FUNCTION command line references the name of a function defined in a FUNCTION command line in the SIERRA scope that describes the linear thermal expansion as function of temperature. This command line is optional. If it is not present, there is no thermal expansion. See the usage discussion below.
- The following material-scaling command lines are used only in Adagio:
  - The target Young's modulus is defined with the TARGET E command line. This command line is optional. See the usage discussion below.
  - The TARGET E FUNCTION command line references the name of a function defined in a FUNCTION command line in the SIERRA scope that describes the time variation of the target Young's modulus. This command line is optional. If it is not present, there is no time dependence in the Target E parameter. See the usage discussion below.
  - The maximum Poisson's ratio is defined with the MAX POISSONS RATIO command line. This command line is optional and will default to 0.5 if not specified. See the usage discussion below.
  - The reference strain is defined with the REFERENCE STRAIN command line. This command line is optional. See the usage discussion below.

The command lines for functions that specify the temperature dependence of C10, C01, and bulk modulus are optional, e.g., the C10 FUNCTION, C01 FUNCTION and BULK FUNCTION command lines. If these command lines are not included, their corresponding material parameters are taken to be independent of temperature. Mooney-Rivlin, like other material models, allows for the specification of thermal strain behavior within the material model itself, via the THERMAL EXPANSION FUNCTION command line. This command line, like the other "function-type" command lines in this model requires that a function associated with the name be defined in the SIERRA scope.

The bulk and shear scalings that can be used with the multilevel solver in Adagio are specified via a combination of the TARGET E, TARGET E FUNCTION, and MAX POISSONS RATIO command lines. If the TARGET E command line is not included (and the MAX POISSONS RATIO command line is included), the shear scaling is set to 1.0, and the bulk scaling is determined from the ratio of the scaled bulk modulus to its unscaled value, where the scaled bulk modulus is computed using the value of the max_poissons_ratio parameter along with the unscaled initial shear modulus

that is determined from the value of the parameters specified in the C10 and C01 command lines. On the other hand, if both the TARGET E command line and the MAX POISSONS RATIO command line are included, bulk and shear scaling values are computed using scaled moduli that are calculated from the target_e and max_poissons_ratio parameter values.

Including the TARGET E FUNCTION command line allows time-dependent bulk and shear scaling to be used. If this command line is not specified, the bulk and shear scalings remain constant in solution time. If the command line is specified, the target Young's modulus that is used for computing the scaled moduli is multiplied by the function value.

The REFERENCE STRAIN command line supplies a value for the reference strain used to create a normalized material constraint violation that is based on strains. Specifying a reference strain implies the use of strains for measuring the material constraint violation (or part of the control-stiffness error in Adagio). Otherwise, the material constraint violation is determined using the change in the scaled stress response over the current model problem.

Brief documentation on the theoretical basis for the Mooney-Rivlin model is given in [1].

State variables for this model are listed in Table 5-29.

Table 5-29. State Variables for MOONEY RIVLIN Model (Section 5.2.33)

Name	Name	Variable Description
1	C10	material constant
2	C01	material constant
3	K	temperature scaled bulk modulus
4	SFJTH	total volumetric strain
5	JTH	thermal volumetric strain
6	SFJTH_FLAG	thermal expansion flag (1 if thermally expanding)
7	WDEV	distortion strain energy
8	SHEAR MODULUS	temperature scaled shear modulus calculated from C10 and C01

5.2.34. Stiff Elastic

```
BEGIN PARAMETERS FOR MODEL STIFF_ELASTIC
  #
  # Elastic constants
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  SHEAR MODULUS  = <real> G
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  TWO MU         = <real> 2μ
  #
  #
  #
  SCALE FACTOR = <real>scale_factor
  REFERENCE STRAIN = <real>reference_strain
END [PARAMETERS FOR MODEL STIFF_ELASTIC]
```

The stiff elastic model is a variation of the isotropic elastic model. The stiff elastic model can be used in both implicit and explicit analysis. However, model is primarily intended to be used with implicit solution in conjunction with the control-stiffness multilevel solver. The control-stiffness option is implemented via the `CONTROL STIFFNESS` command block and is discussed in Chapter 4. The stiff elastic model is used slowly ramp of the stiffness of a stiff material during control stiffness solution to generate a material response more amenable to solution using the implicit conjugate gradient solver. The ultimate material stiffness that is used in the final control stiffness iteration corresponds to the actual moduli that are specified. Essentially when used with control stiffness this model provides an series of predictions and corrections of increasingly stiff materials. It is easier to solve this series of prediction analyses than to directly solve the original poorly conditioned full stiffness analysis.

When this model is used in explicit analysis, or without the control stiffness solver, the material scalings are ignored.

In the above command blocks:

- See Section 5.1.5 for more information on elastic constants input. These constants define the true material stiffness.
- The scaled bulk and shear moduli are computed using a Young's modulus scaled by the value given by the `SCALE FACTOR` line command. These are the scaled down moduli used at the start of the control stiffness iteration.
- The `REFERENCE STRAIN` command line supplies a value for the reference strain used to create a normalized material constraint violation that is based on strains. Specifying a reference strain implies the use of strains for measuring the material constraint violation (or part of the control-stiffness error in Adagio). Otherwise, the material constraint violation is determined using the change in the scaled stress response over the current model problem. Specifying a smaller reference strain means the material constraint is enforced more accurately.

This material has no state variables.

5.2.35. Swanson Model

```
BEGIN PARAMETERS FOR MODEL SWANSON
  #
  # Elastic constants, the shear modulus is computed
  # from A1, P1, B1, Q1, C1, and R1.
  # One of these elastic constants must be input.
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  #
  #
  A1 = <real>a1
  P1 = <real>p1
  B1 = <real>b1
  Q1 = <real>q1
  C1 = <real>c1
  R1 = <real>r1
  CUT OFF STRAIN = <real>ecut
  THERMAL EXPANSION FUNCTION = <string>eth_function_name
  TARGET E = <real>target_e
  TARGET E FUNCTION = <string>etar_function_name
  MAX POISSONS RATIO = <real>max_poissons_ratio
  REFERENCE STRAIN = <real>reference_strain
END [PARAMETERS FOR MODEL SWANSON]
```

The Swanson material model is a hyperelastic constitutive model that is used to simulate rubber. The Swanson model can be used in both Presto and Adagio. When the model is used in Adagio, it can be used with or without the control-stiffness option in Adagio's multilevel solver for nearly incompressible materials where Poisson's ratio, ν, ≈ 0.5. The control-stiffness option is implemented via the `CONTROL STIFFNESS` command block and is discussed in Chapter 4. In the course of solving a series of model problems in Adagio, the material response from this model incorporates scaling the bulk and/or shear behaviors to yield a material response that is more amenable to solution using Adagio's conjugate gradient solver. The final material behavior that is calculated corresponds to the actual moduli that are specified. When this model is used in Presto, the material scalings are ignored.

In the above command blocks the following command lines are required:

- The `A1` command line defines A_1 in Equation 5.192.
- The `P1` command line defines P_1 in Equation 5.192.
- The `B1` command line defines B_1 in Equation 5.192.
- The `Q1` command line defines Q_1 in Equation 5.192.
- The `C1` command line defines C_1 in Equation 5.192.
- The `R1` command line defines R_1 in Equation 5.192.

- The small-strain value used for computing the initial shear modulus is defined with the `CUT OFF STRAIN` command line. This defines e_c in Equation 5.193.
- The above properties define the shear behavior. One additional elastic constant other than the shear modulus is required to define the unscaled bulk behavior. See Section 5.1.5 for more information on elastic constants input.

Other command lines include:

- The `THERMAL EXPANSION FUNCTION` command line references the name of a function defined in a `FUNCTION` command line in the SIERRA scope that describes the linear thermal expansion as a function of temperature. This command line is optional. If it is not present, there is no thermal expansion. See the usage discussion below.
- The following material-scaling command lines are used only in Adagio:
  - The target Young's modulus is defined with the `TARGET E` command line. This command line is optional. See the usage discussion below.
  - The `TARGET E FUNCTION` command line references the name of a function defined in a `FUNCTION` command line in the SIERRA scope that describes the time variation of the target Young's modulus. This command line is optional. If it is not present, there is no time dependence in the Target E parameter. See the usage discussion below.
  - The maximum Poisson's ratio is defined with the `MAX POISSONS RATIO` command line. This command line is optional and will default to 0.5 if not specified. See the usage discussion below.
  - The reference strain is defined with the `REFERENCE STRAIN` command line. This command line is optional. See the usage discussion below.

The strain energy density for the Swanson model is given by

$$U = \frac{3}{2}\frac{A_1}{P_1+1}\left(\frac{\bar{I}_1}{3}-1\right)^{P_1+1} + \frac{3}{2}\frac{B_1}{Q_1+1}\left(\frac{\bar{I}_2}{3}-1\right)^{Q_1+1} + \frac{3}{2}\frac{C_1}{R_1+1}\left(\frac{\bar{I}_1}{3}-1\right)^{R_1+1} + K\left(J_m \ln\left(J_m\right) - J_m + 1\right), \tag{5.192}$$

where $\bar{I}_1$ and $\bar{I}_2$ are the first and second invariants of the left Cauchy-Green strain tensor $\bar{\mathbf{B}}$ which excludes volume change and thermal strains, J_m is the mechanical only part of the volume change which excludes thermal strains, and K is the bulk modulus.

As noted previously, only one of the standard elastic constants are required to define the unscaled bulk behavior. Together, the values for parameters in the `A1`, `P1`, `B1`, `Q1`, `C1`, and `R1` command lines define the unscaled shear behavior, so these command lines must be present. The initial unscaled shear modulus is determined from those parameter values along with the value of the parameter in the `CUT OFF STRAIN` command line via

$$shear\_modulus = A_1\left(\frac{{e_c}^2}{3}\right)^{P_1} + B_1\left(\frac{{e_c}^2}{3}\right)^{Q_1} + C_1\left(\frac{{e_c}^2}{3}\right)^{R_1} \tag{5.193}$$

The Swanson model, like a few of the material models, allows for the specification of thermal strain behavior within the material model itself, via the `THERMAL EXPANSION FUNCTION` command line. This command line, like the other "function-type" command lines in this model, requires that a function associated with the name be defined in the SIERRA scope.

The bulk and shear scalings that can be used with the multilevel solver in Adagio are specified via a combination of the TARGET E, TARGET E FUNCTION, and MAX POISSONS RATIO command lines. If the TARGET E command line is not included (and the MAX POISSONS RATIO command line is included), the shear scaling is set to 1.0, and the bulk scaling is determined from the ratio of the scaled bulk modulus to its unscaled value, where the scaled bulk modulus is computed using the value of the max_poissons_ratio parameter along with the unscaled shear modulus. On the other hand, if both the TARGET E command line and the MAX POISSONS RATIO are included, bulk and shear scaling values are computed using scaled moduli that are calculated from the target_e and max_poissons_ratio parameter values.

Including the TARGET E FUNCTION command line allows time-dependent bulk and shear scaling to be used. If this command line is not specified, the bulk and shear scalings remain constant in solution time. If the command line is specified, the target Young's modulus that is used for computing the scaled moduli is multiplied by the function value.

The REFERENCE STRAIN command line supplies a value for the reference strain used to create a normalized material constraint violation that is based on strains. Specifying a reference strain implies the use of strains for measuring the material constraint violation (or part of the control-stiffness error in Adagio). Otherwise, the material constraint violation is determined using the change in the scaled stress response over the current model problem.

Output variables available for this model are listed in Table 5-30. Brief documentation on the theoretical basis for the Swanson model is given in [1].

Table 5-30. State Variables for SWANSON Model (Section 5.2.35)

Index	Name	Variable Description
1	SFJTH	total volumetric strain
2	JTH	thermal volumetric strain
3	VMECHXX	mechanical left stretch xx component
4	VMECHYY	mechanical left stretch yy component
5	VMECHZZ	mechanical left stretch zz component
6	VMECHXY	mechanical left stretch xy component
7	VMECHYZ	mechanical left stretch yz component
8	VMECHZX	mechanical left stretch zx component
9	SFJTH_FLAG	thermal expansion flag (1 if thermally expanding)
10	WDEV	distortion strain energy

5.2.36. Viscoelastic Swanson Model

```
BEGIN PARAMETERS FOR MODEL VISCOELASTIC_SWANSON
  #
  # Elastic constants, note shear modulus is computed
  # from A1, P1, B1, Q1, C1, and R1.
  # One of these elastic constants must be input.
  #
  YOUNGS MODULUS = <real> E
  POISSONS RATIO = <real> ν
  BULK MODULUS   = <real> K
  LAMBDA         = <real> λ
  #
  #
  BULK FUNCTION <string>bulk_func
  A1 = <real>a1
  A1 FUNCTION <string>a1_func
  P1 = <real>p1
  P1 FUNCTION <string>p1_func
  B1 = <real>b1
  B1 FUNCTION <string>b1_func
  Q1 = <real>q1
  Q1 FUNCTION <string>q1_func
  C1 = <real>c1
  C1 FUNCTION <string>c1_func
  R1 = <real>r1
  R1 FUNCTION <string>r1_func
  CUT OFF STRAIN = <real>ecut
  THERMAL EXPANSION FUNCTION = <string>eth_function_name
  PRONY SHEAR INFINITY = <real>ginf
  PRONY SHEAR 1 = <real>g1
  PRONY SHEAR 2 = <real>g2
  PRONY SHEAR 3 = <real>g3
  PRONY SHEAR 4 = <real>g4
  PRONY SHEAR 5 = <real>g5
  PRONY SHEAR 6 = <real>g6
  PRONY SHEAR 7 = <real>g7
  PRONY SHEAR 8 = <real>g8
  PRONY SHEAR 9 = <real>g9
  PRONY SHEAR 10 = <real>g10
  SHEAR RELAX TIME 1 = <real>tau1
  SHEAR RELAX TIME 2 = <real>tau2
  SHEAR RELAX TIME 3 = <real>tau3
  SHEAR RELAX TIME 4 = <real>tau4
  SHEAR RELAX TIME 5 = <real>tau5
  SHEAR RELAX TIME 6 = <real>tau6
  SHEAR RELAX TIME 7 = <real>tau7
```

```
    SHEAR RELAX TIME 8 = <real>tau8
    SHEAR RELAX TIME 9 = <real>tau9
    SHEAR RELAX TIME 10 = <real>tau10
    WLF COEF C1 = <real>wlf_c1
    WLF COEF C2 = <real>wlf_c2
    WLF TREF = <real>wlf_tref
    NUMERICAL SHIFT FUNCTION = <string>ns_function_name
    TARGET E = <real>target_e
    TARGET E FUNCTION = <string>etar_function_name
    MAX POISSONS RATIO = <real>max_poissons_ratio(0.5)
    REFERENCE STRAIN = <real>reference_strain
    SHEAR MODULUS UPDATE PENALTY = <real>Sherwood
  END [PARAMETERS FOR MODEL VISCOELASTIC_SWANSON]
```

The viscoelastic Swanson model is a finite strain viscoelastic model that has an initial elastic response that matches the Swanson material model. The bulk response is elastic, while the shear response is viscoelastic. This model is commonly employed in simulating the response of rubber materials. The viscoelastic Swanson model can be used in both Presto and Adagio. When the model is used in Adagio, it can be used with or without the control-stiffness option in Adagio's multilevel solver for nearly incompressible materials where Poisson's ratio, ν, ≈ 0.5. The control-stiffness option is implemented via the `CONTROL STIFFNESS` command block and is discussed in Chapter 4. By solving a series of model problems in Adagio, the material response from this model incorporates scaling the bulk and/or shear behaviors to yield a material response that is more amenable to solution using Adagio's conjugate gradient solver. The final material behavior that is calculated corresponds to the actual moduli that are specified. When this model is used in Presto, the material scalings are ignored.

In the above command blocks the following command lines are required:

- Material constants command lines `A1`, `P1`, `B1`, `Q1`,`C1`, `R1`.
- The small-strain value used for computing the glassy shear modulus is defined with the `CUT OFF STRAIN` command line. This defines e_c in Equation 5.194.
- The above parameters define the unscaled shear modules as defined by Equation 5.194. One other elastic constant needs to be included to define the unscaled bulk behavior. The other elastic constant cannot be the shear modulus. See Section 5.1.5 for more information on elastic constants input.
- `PRONY SHEAR INFINITY` command line.
- `WLF COEF C1` command line.
- `WLF COEF C2` command line.
- `WLF TREF` command line.

Other optional command lines are:

$$shear\_modulus = A_1 \left(\frac{{e_c}^2}{3} \right)^{P_1} + 2B_1 \left(\frac{{e_c}^2}{3} \right)^{Q_1} \tag{5.194}$$

- The temperature variation of material constants `A1`, `P1`, `B1`, `Q1`, `C1`, `R1` may be defined with their respective functions `A1 FUNCTION`, `P1 FUNCTION`, `B1 FUNCTION`, `Q1 FUNCTION`,

C1 FUNCTION, and R1 FUNCTION.

- The temperature variation of the BULK MODULUS constant may be defined with the BULK FUNCTION command line.
- The THERMAL EXPANSION FUNCTION command line references the name of a function defined in a FUNCTION command line in the SIERRA scope that describes the linear thermal expansion as a function of temperature. This command line is optional. If it is not present, there is no thermal expansion. See the usage discussion below.
- The normalized relaxation spectra coefficients are specified with the PRONY SHEAR I command lines, where the value of I varies sequentially from 1 to 10.
- The normalized relaxation spectra time constants are specified with the SHEAR RELAX TIME I command lines, where the value of I varies sequentially from 1 to 10.
- NUMERICAL SHIFT FUNCTION command line.
- The following material-scaling command lines are used only in Adagio:
  - The target Young's modulus is defined with the TARGET E command line. This command line is required. See the usage discussion below.
  - The TARGET E FUNCTION command line references the name of a function defined in a FUNCTION command line in the SIERRA scope that describes the time variation of the target Young's modulus. This command line is optional. If it is not present, there is no time dependence in the Target E parameter. See the usage discussion below.
  - The maximum Poisson's ratio is defined with the MAX POISSONS RATIO command line. This command line is optional and will default to 0.5 if not specified. See the usage discussion below.
  - The reference strain is defined with the REFERENCE STRAIN command line. This command line is required. See the usage discussion below.

As noted previously, only two of the elastic constants are required to define the unscaled bulk behavior. Together, the values for parameters in the A1, P1, B1, Q1, C1, and R1 command lines define the unscaled glassy shear behavior, so these command lines must be present. The unscaled glassy shear modulus is determined from those parameter values along with the value of the parameter in the CUT OFF STRAIN command line, so this command line is also required.

The viscoelastic Swanson model, like a few of the material models, allows for the specification of thermal strain behavior within the material model itself, via the THERMAL EXPANSION FUNCTION command line. This command line, like the other "function-type" command lines in this model requires that a function associated with the name be defined in the SIERRA scope.

The bulk and shear scalings that can be used with the multilevel solver in Adagio are specified via a combination of the TARGET E, TARGET E FUNCTION, and MAX POISSONS RATIO command lines. If the TARGET E command line is not included (and the MAX POISSONS RATIO command line is included), the shear scaling is set to 1.0, and the bulk scaling is determined from the ratio of the scaled bulk modulus to its unscaled value, where the scaled bulk modulus is computed using the value of the max_poissons_ratio parameter along with the unscaled shear modulus. On the other hand, if both the TARGET E command line and the MAX POISSONS RATIO command line are included, bulk and shear scaling values are computed using scaled moduli that

are calculated from the `target_e` and `max_poissons_ratio` parameter values.

Including the `TARGET E FUNCTION` command line allows time-dependent bulk and shear scaling to be used. If this command line is not specified, the bulk and shear scalings remain constant in solution time. If the command line is specified, the target Young's modulus that is used for computing the scaled moduli is multiplied by the function value.

The `REFERENCE STRAIN` command line supplies a value for the reference strain used to create a normalized material constraint violation that is based on strains. Specifying a reference strain implies the use of strains for measuring the material constraint violation (or part of the control-stiffness error in Adagio). Otherwise, the material constraint violation is determined using the change in the scaled stress response over the current model problem.

The `SHEAR MODULUS UPDATE PENALTY` Is associated with how the viscoelastic Swanson model interacts with control stiffness. This command is not recommended for use at this time.

Output variables available for this model are listed in Table 5-31. Brief documentation on the theoretical basis for the viscoelastic Swanson model is given in [1, 51, 52, 53].

Table 5-31. State Variables for VISCOELASTIC SWANSON Model (Section 5.2.36)

Index	Name	Variable Description
1	SFJTH	total volumetric strain
2	JTH	thermal volumetric strain
3	VMECHXX	mechanical left stretch xx
4	VMECHYY	mechanical left stretch yy
5	VMECHZZ	mechanical left stretch zz
6	VMECHXY	mechanical left stretch xy
7	VMECHYZ	mechanical left stretch yz
8	VMECHZX	mechanical left stretch zx
9	VSXXDEV1 – VSXXDEV10	viscoelastic deviatoric PK2 stress xx
10	VSYYDEV1 – VSYYDEV10	viscoelastic deviatoric PK2 stress yy
11	VSZZDEV1 – VSZZDEV10	viscoelastic deviatoric PK2 stress zz
12	VSXYDEV1 – VSXYDEV10	viscoelastic deviatoric PK2 stress xy
13	VSYZDEV1 – VSYZDEV10	viscoelastic deviatoric PK2 stress yz
14	VSZXDEV1 – VSZXDEV10	viscoelastic deviatoric PK2 stress zx
15	SOXXDEV	instantaneous deviatoric PK2 stress xx
16	SOYYDEV	instantaneous deviatoric PK2 stress yy
17	SOZZDEV	instantaneous deviatoric PK2 stress zz
18	SOXYDEV	instantaneous deviatoric PK2 stress xy
19	SOYZDEV	instantaneous deviatoric PK2 stress yz
20	SOZXDEV	instantaneous deviatoric PK2 stress zx
21	WLF_AAVG	average WLF shift factor
22	ANAVG	average shift factor
23	WDEV	distortion strain energy

5.2.37. Additional Material Models in Development

In addition to the material models documented here, many additional material models are documented in the Sierra/SM 5.0 Capabilities in Development manual, including

- BARLAT PLASTICITY
- ELASTIC ORTHOTROPIC
- ELASTIC ORTHOTROPIC FAIL
- ELASTIC ORTHOTROPIC DAMAGE
- ELASTIC ORTHOTROPIC SHELL
- FOAM DAMAGE
- HONEYCOMB
- KARAGOZIAN AND CASE CONCRETE
- KAYENTA
- LINEAR ELASTIC
- HOSFORD PLASTICITY
- KARAFILLIS BOYCE PLASTICITY
- NLVE 3D ORTHOTROPIC
- PIEZO
- SHAPE MEMORY ALLOY
- THERMO EP POWER
- THERMO EP POWER WELD
- UNIVERSAL POLYMER

The models, while available for use in Sierra/SM, should be used with caution as they may not be fully verified or supported, and available documentation may not be at the production level. Models in development are periodically moved to production after they have been documented and tested.

5.3. Cohesive Zone Material Models

Several material models are available for use with cohesive zone elements or with Dash contact. These models are described in this section.

Traction separation models used for cohesive surface elements are input within `BEGIN MATERIAL` blocks in the same manner as continuum models. Although density is not a property used by cohesive zone elements, because of their specification within this block, all of these models currently require a density to be provided as part of their input. Output variables for Cohesive Zones can be found in Table 9-17. NOTE: while some of the variables do have x, y, and z components they are not related to the global coordinate system, but instead are related to normal and tangential directions.

5.3.1. Traction Decay

```
BEGIN PARAMETERS FOR MODEL TRACTION_DECAY
  NORMAL DECAY LENGTH = <real>
  TANGENTIAL DECAY LENGTH = <real>
END [PARAMETERS FOR MODEL TRACTION_DECAY]
```

The Traction Decay cohesive model is a simple model that gets initialized with a traction upon activation or insertion of the cohesive element. The traction then decays to zero over specified values of normal and tangential separation. This model is only valid to use in conjunction with dynamic cohesive zone activation through MPC deactivation or dynamic insertion of cohesive surface elements.

In the above command block:

- The density of the material is defined with the `DENSITY` command line. Although this is not used in the model, it is currently a required input parameter.
- The separation length over which the normal traction decays to zero is set by the `NORMAL DECAY LENGTH` command line.
- The separation length over which the tangential traction decays to zero is set by the `TANGENTIAL DECAY LENGTH` command line.

The state variables for this model are listed in Table 5.3.1.

Table 5-32. State Variables for TRACTION DECAY Surface Model (Section 5.3.1)

Index	Name	Variable Description
0	MAX_SEPARATION_S	maximum separation in the first tangential direction
1	MAX_SEPARATION_T	maximum separation in the second tangential direction
2	MAX_SEPARATION_N	maximum separation in the normal direction

5.3.2. Tvergaard Hutchinson

```
BEGIN PARAMETERS FOR MODEL TVERGAARD_HUTCHINSON
  INIT TRACTION METHOD = IGNORE|ADD|EXTRINSIC (IGNORE)
  LAMBDA_1 = <real>
  LAMBDA_2 = <real>
  NORMAL LENGTH SCALE = <real>
  TANGENTIAL LENGTH SCALE = <real>
  PEAK TRACTION = <real>
  PENETRATION STIFFNESS MULTIPLIER = <real>
  USE ELASTIC UNLOADING = NO|YES (YES)
END [PARAMETERS FOR MODEL TVERGAARD_HUTCHINSON]
```

The Tvergaard Hutchinson cohesive model combines the normal and tangential separation into a single normalized separation and calculates a traction per unit length based on this value. This model then calculates normal and tangential traction based on the ratio of the normal and tangential length scales.

In the above command blocks:

- For dynamically activated or dynamically inserted cohesive surface elements, an initial traction can be used to initialize the traction-separation model based on element properties specified in the `ELEMENT DEATH` or `XFEM` block. This is set via the `INIT TRACTION METHOD`. The default behavior is to ignore any initial tractions and let the traction-separation law dictate the behavior. Note that this does not maintain equilibrium upon cohesive element insertion. The `ADD` method offsets the traction-separation law by the initiation traction, which is linearly scaled to zero under further loading. The `EXTRINSIC` method initializes the state of the traction-separation law along the hardening branch. In the case that the initialization traction is greater than the specified peak traction, the traction-separation law is evaluated from the beginning of the traction plateau with the peak traction value reset to the initialization traction. Both the `ADD` and `EXTRINSIC` methods maintain equilibrium upon cohesive element insertion.
- `LAMBDA_1` indicates the normalized separation at which the traction response flattens with an additional increase in separation.
- `LAMBDA_2` indicates the normalized separation at which the traction begins to degrade with an additional increase in separation.
- The separation at which failure occurs in the normal direction is prescribed using the `NORMAL LENGTH SCALE` command.
- The maximum traction is specified through the `PEAK TRACTION` command.
- The separation at which failure occurs in the tangential direction is prescribed using the `TANGENTIAL LENGTH SCALE` command.
- The `PENETRATION STIFFNESS MULTIPLIER` command artificially increases the normal traction to help prevent interpenetration of cohesive surfaces when crack closure occurs. However, cohesive elements are not equipped to handle compression, and this penalty stiffness is an ad hoc approach to contact.
- Set the `USE ELASTIC UNLOADING` command to `YES` to force this model to unload elastically.

The state variables for this model are listed in Table 5-33.

Table 5-33. State Variables for TVERGAARD HUTCHINSON Surface Model (Section 5.3.2)

Index	Name	Variable Description
1	PEAK_TRACTION	maximum traction the model can experience
2	LAMBDA_MAX	maximum lambda the model has experienced
3	TRACTION_AT_LAMBDA_MAX	traction at LAMBDA_MAX
4	G_AT_LAMBDA_MAX	the area under the traction separation curve up to LAMBDA_MAX

5.3.3. Thouless Parmigiani

```
BEGIN PARAMETERS FOR MODEL THOULESS_PARMIGIANI
  INIT TRACTION METHOD = IGNORE|ADD|EXTRINSIC (IGNORE)
  LAMBDA_1_N = <real>
  LAMBDA_1_T = <real>
  LAMBDA_2_N = <real>
  LAMBDA_2_T = <real>
  NORMAL LENGTH SCALE = <real>
  PEAK NORMAL TRACTION = <real>
  TANGENTIAL LENGTH SCALE = <real>
  PEAK TANGENTIAL TRACTION = <real>
  PENETRATION STIFFNESS MULTIPLIER = <real>
  USE ELASTIC UNLOADING = NO|YES (YES)
END [PARAMETERS FOR MODEL THOULESS_PARMIGIANI]
```

The Thouless Parmigiani model supports mixed-mode fracture more accurately than the Tvergaard Hutchinson model by separating the normal and tangential components, allowing one to fail independently of the other. Failure of this model is dependent on the energy release of both normal and tangential components. The shape of the traction-separation curve for this model is hardening, followed by a plateau, followed by softening.

In the above command block:

- For dynamically activated or dynamically inserted cohesive surface elements, an initial traction can be used to initialize the traction-separation model based on element properties specified in the `ELEMENT DEATH` or `XFEM` block. This is set via the `INIT TRACTION METHOD`. The default behavior is to ignore any initial tractions and let the traction-separation law dictate the behavior. Note that this does not maintain equilibrium upon cohesive element insertion. The `ADD` method offsets the traction-separation law by the initiation traction, which is linearly scaled to zero under further loading. The `EXTRINSIC` method initializes the state of the traction-separation law along the hardening branch. In the case that the initialization traction is greater than the specified peak traction, the traction-separation law is evaluated from the beginning of the traction plateau with the peak traction value reset to the initialization traction. Both the `ADD` and `EXTRINSIC` methods maintain equilibrium upon cohesive element insertion.
- `LAMBDA_1_N` indicates the normalized normal separation at which the traction response flattens with an additional increase in separation.
- `LAMBDA_1_T` indicates the normalized tangential separation at which the traction response flattens with an additional increase in separation.
- `LAMBDA_2_N` indicates the normalized normal separation at which the traction begins to decrease with an additional increase in separation.
- `LAMBDA_2_T` indicates the normalized tangential separation at which the traction begins to decrease with an additional increase in separation.
- The separation at which failure occurs in the normal direction is prescribed using the `NORMAL LENGTH SCALE` command.
- The maximum normal traction is specified through the `PEAK NORMAL TRACTION`

command.

- The separation at which failure occurs in the tangential direction is prescribed using the `TANGENTIAL LENGTH SCALE` command.
- The maximum tangential traction is specified through the `PEAK TANGENTIAL TRACTION` command.
- The `PENETRATION STIFFNESS MULTIPLIER` command artificially increases the normal traction to help prevent interpenetration of cohesive surfaces when crack closure occurs. However, cohesive elements are not equipped to handle compression, and this penalty stiffness is an ad hoc approach to contact.
- Set the `USE ELASTIC UNLOADING` command to `YES` to force this model to unload elastically.

The state variables for this model are listed in Table 5-34.

Table 5-34. State Variables for THOULESS PARMIGIANI Surface Model (Section 5.3.3)

Index	Name	Variable Description
1	FRACTION_OF_FAILURE	current fraction of failure
2	PEAK_TRACTION_N	maximum normal traction the model can experience
3	PEAK_TRACTION_T	maximum tangential traction the model can experience
4	LAMBDA_MAX_N	maximum normal lambda the model has experienced
5	TRACTION_AT_LAMBDA_MAX_N	normal traction at LAMBDA_MAX_N
6	LAMBDA_MAX_T	maximum tangential lambda the model has experienced
7	TRACTION_AT_LAMBDA_MAX_T	tangential traction at LAMBDA_MAX_T
8	G_AT_LAMBDA_MAX_N	the area under the normal traction separation curve up to LAMBDA_MAX_N
9	G_AT_LAMBDA_MAX_T	the area under the tangential traction separation curve up to LAMBDA_MAX_T

5.3.4. Functional Interface Traction

```
BEGIN PARAMETERS FOR MODEL FUNCTIONAL_INTERFACE_TRACTION
  NORMAL LENGTH SCALE = <real> (1.0)
  TANGENTIAL LENGTH SCALE = <real> (1.0)
  NORMAL TRACTION FUNCTION = <string>
  TANGENTIAL TRACTION FUNCTION = <string>
END [PARAMETERS FOR MODEL FUNCTIONAL_INTERFACE_TRACTION]
```

The Functional Interface Traction model allows a general normal and tangential traction response law. The normal and tangential components of the traction are defined as functions of the normal and tangential components, respectively, of the separation displacement (appropriately normalized). The traction is evaluated given a separation displacement and subsequently applied. (Note: This model does not presently support some of the features of the other cohesive zone material models, such as elastic unloading and failure.)

The command block for the Functional Interface Traction model starts with the line:

```
BEGIN PARAMETERS FOR MODEL FUNCTIONAL_INTERFACE_TRACTION
```

and terminates with the line:

```
END [PARAMETERS FOR MODEL FUNCTIONAL_INTERFACE_TRACTION]
```

In the above command block:

- The density of the material is defined with the `DENSITY` command line. Although this is not used in the model, it is currently a required input parameter.
- The separation scale in the normal direction is specified using the `NORMAL LENGTH SCALE` command. This is not strictly necessary as this scaling may instead be effected directly within the normal traction response function. It is only provided for similarity with the other cohesive zone material models.
- The separation scale in the tangential direction is specified using the `TANGENTIAL LENGTH SCALE` command. This is not strictly necessary as this scaling may instead be effected directly within the tangential traction response function. It is only provided for similarity with the other cohesive zone material models.
- The normal traction response as a function of the (normalized) normal component of the separation displacement is specified through the `NORMAL TRACTION FUNCTION` command.
- The tangential traction response as a function of the (normalized) tangential component of the separation displacement is specified through the `TANGENTIAL TRACTION FUNCTION` command.

5.3.5. Alves-Roehl

```
begin parameters for model quadshell_model_alves_roehl
  NORMAL LENGTH SCALE = <real>
  TANGENTIAL LENGTH SCALE 1 = <real>
  TANGENTIAL LENGTH SCALE 2 = <real>
  ROTATIONAL LENGTH SCALE = <real>

  # normal work of separation
  NORMAL WOS = <real>

  # tangential work of separation
  TANGENTIAL WOS 1 = <real>
  TANGENTIAL WOS 2 = <real>

  # rotational work of separation
  ROTATIONAL WOS = <real>

  USE ELASTIC UNLOADING = NO|YES (YES)
end
```

In the above command block:

- The separation at which failure occurs in the normal direction is prescribed using the `NORMAL LENGTH SCALE` command.
- The separation scale in the tangential direction 1 is specified using the `TANGENTIAL LENGTH SCALE 1` command.
- The separation scale in the tangential direction 2 is specified using the `TANGENTIAL LENGTH SCALE 1` command.
- The characteristic length for rotational bending is prescribed using the `ROTATIONAL LENGTH SCALE` command.
- The normal work of separation (WOS) in the normal direction is set using the `NORMAL WOS` command.
- The tangential work of separation in the tangential direction 1 is set using the `TANGENTIAL WOS 1` command.
- The tangential work of separation in the tangential direction 2 is set using the `TANGENTIAL WOS 2` command.
- The rotational work of separation (WOS) is set using the `ROTATIONAL WOS` command.
- Set the `USE ELASTIC UNLOADING` command to `YES` to force this model to unload elastically.

5.4. VUMAT

```
USER SUBROUTINE FILE = <string>sub_file_name

BEGIN MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BEGIN PARAMETERS FOR MODEL ABAQUS_VUMAT
    #
    # Elastic constants
    #
    YOUNGS MODULUS = <real> E
    POISSONS RATIO = <real> ν
    SHEAR MODULUS  = <real> G
    BULK MODULUS   = <real> K
    LAMBDA         = <real> λ
    #
    # Required parameters
    #
    VUMAT SUBROUTINE = <string>sub_name
    PROPERTIES       = <real list>props
    DEPVAR           = <int>num_state
    #
    # Optional parameters
    #
    COORDINATE SYSTEM = <string>coordinate_system_name
  END
END
```

The ABAQUS_VUMAT model in Sierra/SM provides an interface for most user-defined material subroutines which conform to the Abaqus VUMAT standard. The VUMAT interface currently supports both shell and solid elements.

The following commands are available.

- Although the material response is defined solely by the user subroutine, at least two elastic constants must still be provided. The elastic constants are used in computation of hourglass stiffness, contact stiffness, solver preconditioners, and other capabilities that require a measure of approximate material stiffness. See Section 5.1.5 for more information on elastic constants input.
- The name of the VUMAT subroutine is specified by the `VUMAT SUBROUTINE` command line. The `USER SUBROUTINE FILE` should contain the source code for the material subroutine. User subroutines are built into Sierra/SM using the '`sierra --make`' option. Consult Chapter 10 for more information on building and running user subroutines.



Warning: The subroutine name `sub_name` must *not* be "`VUMAT`" or "`vumat`." Instead, choose a unique name.

- The user may supply a list of material properties for the ABAQUS_VUMAT model associated with material `mat_name` using the `PROPERTIES` command line. The order of the property list will be the same as the order of the `PROPS` array in the material subroutine.
- The `DEPVAR` command line specifies the number of state variables which should be allocated for the VUMAT material.
- Users may also specify a material coordinate system for the VUMAT. In order to use a material coordinate system, the user must specify a coordinate system using the `COORDINATE SYSTEM` command line in the VUMAT command block. Consult Section 5.1.7 for additional guidance on defining coordinate systems.
  If a custom coordinate system is specified, `STRESSOLD`, `STRAININC`, `DEFGRADOLD`, and `DEFGRADNEW` will be rotated into the material frame before entering the VUMAT subroutine. Output stress `STRESSNEW` should be computed in the material coordinate frame, since it will automatically be rotated into the global Cartesian coordinate frame upon leaving the material subroutine.
  In addition to specifying a coordinate system, the user may input up to three ordered rotation axis–Euler angle pairs for additional rotation. Axis–angle pairs may be prescribed on an element-by-element basis using an `INITIAL CONDITION` command block to set VUMAT state variables `ANGLE_1`, `AXIS_1`, `ANGLE_2`, `AXIS_2`, and `ANGLE_3`, `AXIS_3`. Angles are specified in degrees, while axes are denoted $(1,2,3)$, representing (X,Y,Z), respectively. These variables may be, e.g., read from a file or expressed as a user function. Use of `INITIAL CONDITION` command blocks is described in the chapter on Boundary Conditions and Initial Conditions, see Section 7.3.
  Built-in state variables pertaining to material coordinate system rotations are listed in Table 5-37.

The VUMAT state variables are available for output as ABQ_STATE_j, where j is a one-based index $1 \leq j \leq$ `DEPVAR`. Consult Chapter 9 for additional details on output of variables.

Input and output subroutine arguments are listed in Tables 5-35 and 5-36. Arguments which are not currently fully supported in the VUMAT interface are indicated with a **!**.

5.4.1. Other Guidelines for VUMAT Use

The shell VUMAT is currently only supported for plane stress conditions. The out-of-plane stress, `STRESSNEW`(i,3), is explicitly set to zero upon return. The VUMAT can alter the plane strain increment `STRAININC`(i,3) accordingly to satisfy plane stress.

When implementing a VUMAT, it is important to note that all internal variables are initialized to zero, so constitutive routines must be designed accordingly. For example, in the presence of a damage-type variable, it is recommended to adopt the convention that an undamaged state has damage $d = 0.0$, while a damaged state has damage $d > 0.0$. Because there is currently no way to detect the beginning of a simulation from within the VUMAT interface, there is no reliable way to use the standard VUMAT initialization.

As noted in Tables 5-35 and 5-36, internal work and plastic dissipation may not be specified or accessed within the current VUMAT interface. External work and kinetic energy are computed

Table 5-35. VUMAT Input Arguments

Argument Name	Description (Value)
JBLOCK	Number of elements in workset (always one, since the VUMAT currently processes one element at a time)
NDIR	Number of stress components (three for both solids and shells)
NSHR	Number of shear components (three for solids or one for shells)
NSTATEV	Number of state variables (input num_state)
NFIELD!	Number of user-defined field variables (zero)
NPROPS	Length of property array (length of input props)
LANNEAL!	Annealing process flag (zero)
STEPTIME	Current analysis time
TOTALTIME	Current analysis time (identical to STEPTIME)
DT	Current time step size
CMNAME!	Material name (empty string)
COORDMP!	Material point coordinates (zero)
CHARLENGTH	Element characteristic size (cube root of solid volume or square root of shell area)
PROPS	User-defined property array (input props)
DENSITY!	Current material point density (zero)
STRAININC	Strain increment over the current time step
RELSPININC!	Incremental relative rotation at each material point (zero)
TEMPOLD	Temperature at start of time step
STRETCHOLD!	Stretch at start of time step (identity tensor)
DEFGRADOLD	Deformation gradient (calculated for solids; unsupported for shells and set to identity tensor)
FIELDOLD!	User-defined field variable values at start of time step (zero)
STRESSOLD	Material stress at start of time step
STATEOLD	Material state variables at start of time step
ENERINTERNOLD!	Internal energy per unit mass at start of time step (zero)
ENERINELASOLD!	Dissipated inelastic energy per unit mass at start of time step (zero)
TEMPNEW	Temperature at end of time step
STRETCHNEW!	Stretch tensor at end of time step (identity tensor)
DEFGRADNEW	Deformation gradient (calculated for solids; unsupported for shells and set to identity tensor)
FIELDNEW!	User-defined field variable values at end of time step (zero)

Table 5-36. VUMAT Output Arguments

Argument Name	Description
STRESSNEW	Updated material stress at end of step
STATENEW	Updated material state variables at end of step
ENERINTERNNEW!	Updated internal energy per unit mass at start of time step (ignored)
ENERINELASNEW!	Updated dissipated inelastic energy per unit mass at end of time step (ignored)

outside of the material subroutine by LAMÉ and can be accessed via a USER OUTPUT block in the Sierra/SM input file (consult Chapter 9 for details). To store dissipated plastic energy, an additional internal state variable may be defined and updated within the VUMAT.

Table 5-37. Built-In State Variables for ABAQUS_VUMAT Model (Section 5.4)

Index	Name	Variable Description
num_state + 1	Q_XX	X component of the material 11 unit vector
num_state + 2	Q_YY	Y component of the material 22 unit vector
num_state + 3	Q_ZZ	Z component of the material 33 unit vector
num_state + 4	Q_XY	Y component of the material 11 unit vector
num_state + 5	Q_YZ	Z component of the material 22 unit vector
num_state + 6	Q_ZX	X component of the material 33 unit vector
num_state + 7	Q_YX	X component of the material 22 unit vector
num_state + 8	Q_ZY	Y component of the material 33 unit vector
num_state + 9	Q_XZ	Z component of the material 11 unit vector
num_state + 10	ANGLE_1	Rotation angle about axis 1 (degrees)
num_state + 11	ANGLE_2	Rotation angle about axis 2 (degrees)
num_state + 12	ANGLE_3	Rotation angle about axis 3 (degrees)
num_state + 13	AXIS_1	First axis of rotation
num_state + 14	AXIS_2	Second axis of rotation
num_state + 15	AXIS_3	Third axis of rotation

References

[1] W.M. Scherzinger and D.C. Hammerand. Constitutive models in LAME. Technical Report SAND2007-5873, Sandia National Laboratories, Albuquerque, NM, September 2007. pdf.

[2] A. E. H. Love. *A Treatise on the Mathematical Theory of Elasticity*. Dover, 4th edition, 1944.

[3] I. S. Sokolnikoff. *Mathematical Theory of Elasticity*. Krieger Publishing Company, 1956.

[4] S. Timoshenko and J.N. Goodier. *Theory of Elasticity*. McGraw Hill Book Company, 3rd edition, 1970.

[5] C.M. Stone. SANTOS – a two-dimensional finite element program for the quasistatic, large deformation, inelastic response of solids. Technical Report SAND90-0543, Sandia National Laboratories, Albuquerque, NM, 1996. pdf.

[6] J.C. Simo and T.J.R. Hughes. *Computational Inelasticity*. Springer-Verlag, New York, NY, 1998.

[7] C. M. Stone, G. W. Wellman, and R. D. Krieg. A vectorized elastic/plastic power law hardening material model including Lüders strains. Technical Report SAND90-0153, Sandia National Laboratories, Albuquerque, NM, March 1990. pdf.

[8] G. W. Wellman. A simple approach to modeling ductile failure. Technical Report SAND2012-1343, Sandia National Laboratories, Albuquerque, NM, June 2012. pdf.

[9] W.F. Hosford. A generalized isotropic yield criterion. *Journal of Applied Mechanics*, 39: 607–609, 1972.

[10] A. Graf and W. Hosford. Calculations of forming limit diagrams. *Metallurgical Transactions A*, 21A:87–94, 1990.

[11] W. M. Scherzinger. A return mapping algorithm for isotropic and anisotropic plasticity models using a line search method. *Computer Methods in Applied Mechanics and Engineering*, 317:526–553, 2017.

[12] B.T. Lester and W.M. Scherzinger. Failure formulations in modular plasticity models. Memo to distribution SAND2019-9678R, Sandia National Laboratories, Albuquerque, NM and Livermore, CA, 2019.

[13] B.T. Lester and W.M. Scherzinger. Modular growth and nucleation formulations in modular plasticity models. Memo to distribution SAND2020-7404R, Sandia National Laboratories, Albuquerque, NM and Livermore, CA, 2020.

[14] B.T. Lester and W.M. Scherzinger. Adiabatic heating in modular plasticity models. Memo to distribution SAND2019-15003R, Sandia National Laboratories, Albuquerque, NM and Livermore, CA, 2019.

[15] R. Hill. *The Mathematical Theory of Plasticity*. Oxford University Press, 1950.

[16] F. Barlat, H. Aretz, J. W. Yoon, J. C. Brem, and R. E. Dick. Linear transformation-based anisotropic yield functions. *International Journal of Plasticity*, 21:1009–1039, 2005.

[17] G.R. Johnson and W.H. Cook. Fracture characteristics of three metals subjected to various strains, strain rates, temperatures and pressures. *Engineering Fracture Mechanics*, 21(1): 31–48, 1985.

[18] G.R. Johnson and W.H. Cook. A constitutive model and data for metals subjected to large strains, high strain rates and high temperatures. In *Proc. 7th. Int. Symp. on Ballistics*, pages 541–547, The Hague, The Netherlands, 1983.

[19] D.J. Bammann, M.L. Chiesa, and G.C. Johnson. Modelling large deformation and failure in manufacturing processes. In T. Tatsumi, E. Watanabe, and T. Kambe, editors, *Proceedings of the 19th International Congress of Theoretical and Applied Mechanics*, pages 359–376, Amsterdam, 1997. Elsevier Science Publishers.

[20] D.J. Bammann, M.L. Chiesa, M.F. Horstemeyer, and L.E. Weingarten. Failure in ductile materials using finite element methods. In N. Jones and T. Wierzbicki, editors, *Structural Crashworthiness and Failure*, pages 1–53, London, 1993. Elsevier Applied Science.

[21] D.J. Bammann. Modeling temperature and strain dependent large deformations in metals. *Applied Mechanics Reviews*, 43(5):312–319, 1990. doi.

[22] A.A. Brown and D.J. Bammann. Validation of a model for static and dynamic recrystallization in metals. *International Journal of Plasticity*, 32–33:17–35, 2012.

[23] U.F. Kocks and H. Mecking. A mechanism for static and dynamic recovery. In P. Haasen, V. Gerold, and G. Kostorz, editors, *Strength of Metals and Alloys*, pages 345–350, Oxford, 1979. Pergamon Press.

[24] E. Nes. Recovery revisited. *Acta Metall. Mater.*, 43:2189–2207, 1990.

[25] S. Kok, A.J. Beaudoin, and D.A. Tortorelli. On the development of stage IV hardening using a model based on the mechanical threshold. *Acta Mater.*, 50:1653–1667, 2002.

[26] A. Cocks and M. Ashby. Intergranular fracture during power-law creep under multiaxial stresses. *Metal Sci.*, 14(8–9):395–402, 1980.

[27] J. Lemaitre and J.-L. Chaboche. *Mechanics of Solid Materials*. Cambridge University Press, 1990.

[28] F.K.G. Odqvist. *Mathematical theory of creep and creep rupture*. Clarendon Press, Oxford, 1966.

[29] M.K. Neilsen, S. N. Burchett, C. M. Stone, and J. J. Stephens. A viscoplastic theory for braze alloys. Technical Report SAND96-0984, Sandia National Laboratories, Albuquerque, NM, 1996. pdf.

[30] Darrell E Munson and PR Dawson. Constitutive model for the low temperature creep of salt (with application to WIPP). Technical Report SAND79-1853, Sandia National Laboratories, Albuquerque, NM, 1979.

[31] Darrell E Munson and PR Dawson. Transient creep model for salt during stress loading and unloading. Technical Report SAND82-0962, Sandia National Laboratories, Albuquerque, NM, 1982.

[32] Darrell E Munson, Arlo F Fossum, and Paul E Senseny. Advances in resolution of discrepancies between predicted and measured in situ WIPP room closures. Technical Report SAND88-2948, Sandia National Laboratories, Albuquerque, NM, 1989.

[33] Benjamin Reedlunn. Enhancements to the munson-dawson model for rock salt. Technical Report SAND2018-12601, Sandia National Laboratories, Albuquerque, NM, 2018.

[34] L.M. Taylor and D.P. Flanagan. Pronto3D: A three-dimensional transient solid dynamics program. Technical Report SAND87-1912, Sandia National Laboratories, Albuquerque, NM, March 1989. pdf.

[35] R.D. Krieg. A simple constitutive description for cellular concrete. Technical Report SAND SC-DR-72-0883, Sandia National Laboratories, Albuquerque, NM, 1978. pdf.

[36] D.V. Swenson and L. M. Taylor. A finite element model for the analysis of tailored pulse stimulation of boreholes. *International Journal for Numerical and Analytical Methods in Geomechanics*, 7:469–484, 1983. doi.

[37] M. K. Neilsen, W. Y. Lu, B. Olsson, and T. Hinnerichs. A viscoplastic constitutive model for polyurethane foams. In *Proceedings ASME 2006 International Mechanical Engineering Congress and Exposition*, Chicago, IL, 2006. ASME.

[38] L.J. Gibson and M.F. Ashby. *Cellular Solids: Structure and Properties*. Cambridge Solid State Science Series. Cambridge University Press, Cambridge, UK, second edition, 1997.

[39] M.K. Neilsen, H.S. Morgan, and R.D. Krieg. A phenomenological constitutive model for low density polyurethane foams. Technical Report SAND86-2927, Sandia National Laboratories, Albuquerque, NM, April 1987. pdf.

[40] B. Störakers. On material representation and constitutive branching in finite compressible elasticity. *Journal of the Mechanics and Physics of Solids*, 34:125–145, 1986.

[41] R. W. Ogden. *Non-Linear Elastic Deformaitons*. Dover, New York, 1984.

[42] P. J. Blatz and W. L. Ko. Application of finite elasticity theory to the deformation of rubbery materials. *Transactions of the Society of Rheology*, 6:223–251, 1962.

[43] G. Bolzon and R. Vitaliani. The blatz-ko model and homogenization. *Archive of Applied Mechanics*, 63:228–241, 1993.

[44] A.E. Green and W. Zerna. *Theoretical Elasticity, 2nd Edition*. Clarendon Press, Oxford, 1968.

[45] M.K. Neilsen, J.D. Pierce, and R.D. Krieg. A constitutive model for layered wire mesh and aramid cloth fabric. Technical Report SAND91-2850, Sandia National Laboratories, Albuquerque, NM, 1993. pdf.

[46] R.G. Whirley, B.E. Engelmann, and J.O. Halquist. DYNA3D user's manual, 1991.

[47] J. C. Simo and R. L. Taylor. A return mapping algorithm for plane stress elastoplasticity. *International Journal for Numerical Methods in Engineering*, 22:649–670, 1986.

[48] D. B. Adolf, R. S. Chambers, and M. A. Neidigk. A simplified potential energy clock model for glassy polymers. *Polymer*, 50:4257–4269, 2009.

[49] J. M. Caruthers, D. B. Adolf, R. S. Chambers, and P. Shrikhande. A thermodynamically consistent, nonlinear viscoelastic approach for modeling glassy polymers. *Polymer*, 45: 4577–4597, 2004.

[50] D. B. Adolf, R. S. Chambers, and J. M. Caruthers. Extensive validation of a thermodynamically consistent, non-linear viscoelastic model for glassy polymers. *Polymer*, 45:4599–4621, 2004.

[51] Hibbitt, Karlsson and Sorensen. ABAQUS version 6.6, theory manual, 2006.

[52] D.C. Hammerand. ABAQUS style finite strain viscoelasticity in Adagio. Technical Report Memo, Sandia National Laboratories, Albuquerque, NM, March 2003.

[53] D.C. Hammerand. Finite strain viscoelasticity in Adagio and ABAQUS. Technical Report Memo, Sandia National Laboratories, Albuquerque, NM, July 2003.

6. ELEMENTS

This chapter explains how material, geometric, and other properties are associated with element blocks in a mesh file. A mesh file generally contains only topological information about elements. For example, there may be a group of four-noded elements in the mesh file which define a facet in three-dimensional space. These elements could be interpreted as either shells or membranes in the finite element model, based on command lines in the input file. The specifics of a material type associated with these four node facets are also set in the input file.

Most elements can be used in both explicit and implicit calculations. If an element is available for one but not the other, this information will be noted in the relevant element section. In particular there are special element implementations available for explicit calculations in smoothed particle hydrodynamics (SPH). This chapter also includes descriptions of the commands for mass property calculations, element death, and mesh rebalancing (which is available for explicit calculations only). "Element-like" capabilities are also discussed.

Highlights of chapter contents follow. Section 6.1 discusses the `FINITE ELEMENT MODEL` command block, which provides the description of a mesh associated with the elements. Section 6.2 presents the section command blocks that are used to define the different element sections. Next in Section 6.3 are descriptions of command blocks that exhibit element-like functionality. Section 6.3.1 explains the mechanism (explicit only). Section 6.4 describes the `MASS PROPERTIES` command block, which lets the user compute the total mass of the model or the mass of sub-parts of the model once the element blocks are completely defined in terms of geometry and material. Section 6.5 details the `ELEMENT DEATH` command block, which lets the user delete (kill) elements based on various criteria during an analysis. Section 6.6 describes particle embedding (explicit only), which specifies element blocks in which each element is embedded with particles during initialization. A command block for derived quantities that are to be used with transfers or error estimation is discussed in Section 6.7. Section 6.8 presents various options available in explicit calculations for rebalancing of the initial mesh decomposition for parallel runs, which may improve the parallel performance of some problems.

Most of the command blocks and command lines described next appear within the SIERRA scope. There are some exceptions, and these exceptions are noted.

6.1. Finite Element Model

```
BEGIN FINITE ELEMENT MODEL <string>mesh_descriptor
  DATABASE NAME = <string>mesh_file_name
  DATABASE TYPE = <string>database_type(exodusII)
  ALIAS <string>mesh_identifier AS <string>user_name
  OMIT BLOCK <string>block_list
  COMPONENT SEPARATOR CHARACTER = <string>separator
  TIME SCALE FACTOR = <real>timeScaleFactor
  DECOMPOSITION METHOD = RCB|RIB|LINEAR|HSFC|BLOCK|CYCLIC|
    RANDOM|KWAY|GEOM_KWAY|KWAY_GEOM|METIS_SFC|EXTERNAL

  BEGIN BLOCK DEFAULTS
    #
    #  Set default values for each 'PARAMETERS FOR
    #  BLOCK' command
    #
  END

  BEGIN PARAMETERS FOR BLOCK [<string list>block_names]
    #
    # Command lines that define attributes for
    # a particular element block appear in this
    # command block.
    #
  END [PARAMETERS FOR BLOCK <string list>block_names]
END [FINITE ELEMENT MODEL <string>mesh_descriptor]
```

The input file must point to a mesh file that is to be used for an analysis. The name of the mesh file appears within a `FINITE ELEMENT MODEL` command block, which appears in the SIERRA scope. In this command block, the particular mesh file that describes the model is identified. There will also be one or more `PARAMETERS FOR BLOCK` command blocks in this block. (All the `PARAMETERS FOR BLOCK` command blocks are embedded in the `FINITE ELEMENT MODEL` command block.) A material type and model, a section, and various other parameters for the element block are set within the `PARAMETERS FOR BLOCK` command block. The concept of "section" is explained in Section 6.1.7.

The following element types are supported in Sierra/SM analyses:

Eight-node, uniform-gradient hexahedron Both a midpoint-increment formulation [1] and a strongly objective formulation are implemented [2]. These elements can be used with any of the material models described in Chapter 5.

Six-node, uniform-gradient wedge Both a midpoint-increment formulation [1] and a strongly objective formulation are implemented [2]. These elements can be used with any of the material models described in Chapter 5.

Five-node, uniform-gradient pyramid Both a midpoint-increment formulation [1] and a strongly objective formulation [2] are implemented. These elements can be used with any

of the material models described in Chapter 5.

Eight-node, selective-deviatoric hexahedron The strongly objective formulation is recommended. Midpoint increment strain incrementation is available for legacy compatibility. This element can be used with any of the material models described in Chapter 5. The selective-deviatoric integration rule is specified in the `BEGIN SOLID SECTION` command block using the `FORMULATION` command line as described in Section 6.2.1.

Four-node tetrahedron There is now the regular element formulation for the four-node tetrahedron and a node-based formulation for the four-node tetrahedron. For the regular element formulation, only a strongly objective formulation is implemented. The concept of a node-based four-node tetrahedron is described in [3]. The regular four-node tetrahedron can be used with any of the material models described in Chapter 5. The node-based tetrahedron can be used with any of the material models described in Chapter 5. When using node-based tetrahedron it is important that nodal quantities are used where other element types use element quantities. For example, element variable stress is evaluated with a regular four-node tetrahedron, but in node-based tetrahedron one should use nodal variable element_stress_##, where ## is a one-based integer > 0. Nodal variables for node-based tetrahedra will exist on all nodes but are zero where they are not used. Element variables for node-based elements serve as intermediate variables and should not be used in post-processing in the same way as other regular element variables.

Ten-node, uniform-gradient tetrahedron This element has been deprecated due to accuracy and robustness issues.

Ten-node, composite tetrahedron The composite tet is now the default ten-node tet element in Sierra/SMṪhis element can be used with any of the material models described in Chapter 5. This element can be used by creating an empty solid section, by not specifying a section in the block definition, or by using a Total Lagrange section. See section 6.2.2 for more details.

Ten-node, fully-integrated tetrahedron Only a strongly objective formulation is implemented. This element can be used with any of the material models described in Chapter 5. The full integration rule is specified in the `BEGIN SOLID SECTION` command block using the `FORMULATION` command line as described in Section 6.2.1.

Four-node, quadrilateral, uniform-gradient membrane Both a midpoint-increment formulation and a strongly objective formulation are implemented. This element is derived from the Key-Hoff shell formulation [4]. The strongly objective formulation has not been extensively tested, and it is recommended that the midpoint-increment formulation, which is the default, be used for this element type. These elements can be used with any of the following material models described in Chapter 5:

- Elastic
- Elastic-plastic
- Elastic-plastic power-law hardening
- Multilinear elastic-plastic hardening (no failure)

Four-node, quadrilateral shell This shell uses the Key-Hoff formulation [4]. Both a midpoint-increment formulation and a strongly objective formulation are implemented. The strongly objective formulation has not been extensively tested, and it is recommended that the midpoint-increment formulation, which is the default, be used for this element type. These elements can be used with any of the following material models described in Chapter 5:

- Elastic
- Elastic-plastic
- Elastic-plastic power-law hardening
- Multilinear elastic-plastic hardening without failure
- Multilinear elastic-plastic hardening with failure

Four-node, quadrilateral, selective-deviatoric membrane Only a midpoint-increment formulation is implemented. These elements can be used with any of the following material models described in Chapter 5:

- Elastic
- Elastic-plastic
- Elastic-plastic power-law hardening
- Multilinear elastic-plastic hardening (no failure)

Linear elastic shell The linear elastic shell element is linear in both a material and geometric sense. The linear elastic shell element can be used with any material, however it will use only the density and elastic material constants of that material. Use of the linear elastic shell element is specified with the `FORMULATION = NQUAD` command in the section command block.

Three-node, triangular shell *Explicit* This shell uses the same formulation as the three-node triangular shell in Pronto [1]. A midpoint-increment formulation is implemented. This element can be used with any of the following material models described in Chapter 5:

- Elastic
- Elastic-plastic
- Elastic-plastic power-law hardening
- Multilinear elastic-plastic hardening without failure
- Multilinear elastic-plastic hardening with failure

Two-node beam The beam element is a uniform result model. Strains and stresses are computed only at the midpoint of the element. These midpoint values determine the forces and moments for the beam. The beam element is based on an incremental kinematic formulation that is accurate for large strains and rotations (i.e., this element exactly agrees with a logarithmic strain formulation under large strain axial loading). Thinning of the cross section is taken into account through a constant volume assumption. There are several different sections currently implemented for the beam including rod, tube, bar, box, and 'I.' This element can be used with any of the following material models described in Chapter 5:

- Elastic
- Elastic-plastic

Two-node truss The two-node truss element carries only a uniform axial stress. Currently, there is a linear-elastic material model for the truss element.

Two-node spring The two-node spring element computes a uniaxial resistance force based on a nonlinear force-engineering strain function. This element can handle preloads, mass per unit length, resetting of the initial length after preload and any arbitrary loading function.

Two-node damper Explicit The two-node damping element computes a damping force based on the relative velocity of the two nodes along the axis of the element. This element uses only a damping parameter for a material property.

Point mass The point mass element allows the user to put a specified mass and/or rotational inertia at a single node. This element requires input for density, but does not make use of any other material properties.

Particle Explicit The particle element is a zero-dimensional, sphere-topology element which may be used with any of the material models described in Chapter 5.

The command block to describe a mesh file begins with

```
BEGIN FINITE ELEMENT MODEL <string>mesh_descriptor
```

and is terminated with

```
END [FINITE ELEMENT MODEL <string>mesh_descriptor]
```

where `mesh_descriptor` is a user-selected name for the mesh. This section discusses the command lines within the scope of the `FINITE ELEMENT MODEL` command block but outside the scope of the `PARAMETERS FOR BLOCK` command block, as well as the `PARAMETERS FOR BLOCK` command block and the associated command lines for this particular block.

6.1.1. Identification of Mesh File

Nested within the `FINITE ELEMENT MODEL` command block are two command lines (`DATABASE NAME` and `DATABASE TYPE`) that give the mesh name and define the type for the mesh file, respectively. The command line

```
DATABASE NAME = <string>mesh_file_name
```

gives the name of the mesh file with the string `mesh_file_name`. If the current mesh file is in the default directory and is named `job.g`, then this command line would appear as:

```
DATABASE NAME = job.g
```

If the mesh file is in some other directory, the command line would have to show the path to that directory. For parallel runs, the string `mesh_file_name` is the base name for the spread of parallel mesh files. For example, for a four-processor run, the actual mesh files associated with a base name of `job.g` would be `job.g.4.0`, `job.g.4.1`, `job.g.4.2`, and `job.g.4.3`. The database name on the command line would be `job.g`.

Two meta-characters can appear in the name of the mesh file. If the `%P` character is found in the name, it will be replaced with the number processors being used for the run. For example, a job running on 1024 processors has the name `mesh-%P/job.g`, then the name would be expanded to `mesh-1024/job.g` and the associated mesh files would be `mesh-1024/job.g.1024.0000` to `mesh-1024/job.g.1024.1023`. The other recognized meta-character is `%B` which is replaced with the base name of the input file containing the input commands. For example, if the commands are in the file `my_analysis_run.i` and the mesh database name is specified as `%B.g`, then the mesh would be read from the file `my_analysis_run.g`.

If the mesh file does not use the Exodus II format, the mesh file format must be specified using the command line

```
DATABASE TYPE = <string>database_type(exodusII)
```

Currently, only the Exodus II database format is supported by Presto and Adagio for mesh input. Other options may be added in the future.

6.1.2. Alias

It is possible to associate a user-defined name with some mesh entity. The mesh entity names for Exodus II entities are typically the concatenation of the entity type (for example, “block”, “nodelist”, or “surface”), an underscore (“_”), and the entity id. This generated name can be aliased to a more descriptive name by using the `ALIAS` command line:

```
ALIAS <string>mesh_identifier AS <string>user_name
```

This alias can then be used in other locations in the input file in place of the Exodus II name. Examples of this association are as follows:

```
Alias block_1   as Case
Alias block_10  as Fin
Alias block_12  as Nose
Alias surface_1 as Nose_Case_Interface
Alias surface_2 as OuterBoundary
```

The above examples use the Exodus II naming convention described in Section 1.6. Note that the alias may not be recognized by all code capabilities, such as contact block skinning.

6.1.3. Omit Block

If the finite element mesh contains element blocks that should be omitted from the finite element analysis, the `OMIT BLOCK` line command is used:

```
OMIT BLOCK <string>block_list
```

The element blocks listed in the command are removed from the model. Any node sets or surfaces only existing on nodes or elements in the omitted element blocks are also omitted. When

using this command, it is necessary to deactivate (comment out) or remove any references to the omitted blocks in the rest of the input file. Note that if this command is used in a parallel analysis, it is possible for the resulting model to become unbalanced if, for example, the omitted element blocks make up a large portion of the elements on one or more processors. In this case with explicit calculations, the mesh can be rebalanced using the `REBALANCE` command described in Section 6.8.1.

If the block is given an alias, then that alias name must be provided to the block omission command; otherwise, the standard entity name, "block_" followed by the entity id, must be used. See Section 1.6 for further discussion of the Exodus II block naming convention.

Examples of omitting element blocks are:

```
Omit Block block_1 block_2
Omit Block block_10
Omit Block Fin Nose
```

6.1.4. Time Scale Factor

The `TIME SCALE FACTOR` command is used to put a uniform scale factor on the time stamps in the input mesh database. This command will have an effect on capabilities such as `READ VARIABLE` (Section 7.3.5) that reads boundary condition data from the input mesh.

6.1.5. Component Separator Character

A variable defined on the mesh database can be used as an initial condition, or a prescribed temperature with the `READ VARIABLE` command. If the variable is a vector or a tensor, then the base name of the variable will be separated from the suffixes with a separator character. The default separator character is an underscore, but it can be changed with the `COMPONENT SEPARATOR CHARACTER` command.

```
COMPONENT SEPARATOR CHARACTER = <string>character|NONE
```

For example, the variable `displacement` can have the suffixes `x`, `y`, etc. By default, the base name is separated from the suffixes with an underscore character, yielding `displacement_x`, `displacement_y`, etc., in the mesh file. The underscore can be replaced as the default separator by using the above command line. If the data used the period as the separator, then the command would be

```
COMPONENT SEPARATOR CHARACTER = .
```

In the displacement example the components would then appear in the mesh file as `displacement.x`, `displacement.y`, etc. The separator can be eliminated with an empty string or `NONE`.

6.1.6. DECOMPOSITION METHOD

The DECOMPOSITION METHOD command is an optional command used to specify how the mesh will be decomposed across multiple processors. This command is used in lieu of SEACAS tools such as "decomp" and the sierra –pre option. For more information about the different decomposition types please run “decomp - -help” in your terminal.

6.1.7. Element Block Parameters

```
BEGIN PARAMETERS FOR BLOCK [<string list>block_names]
  MATERIAL = <string>material_name
  MODEL = <string>model_name
  DENSITY SCALE FACTOR = <real>density_scale_factor(1.0)
  INCLUDE ALL BLOCKS
  REMOVE BLOCK = <string list>block_names
  SECTION = <string>section_id
  LINEAR BULK VISCOSITY =
    <real>linear_bulk_viscosity_value(0.06)
  QUADRATIC BULK VISCOSITY =
    <real>quad_bulk_viscosity_value(1.20)
  HOURGLASS STIFFNESS =
    <real>hourglass_stiff_value(0.05)
  HOURGLASS VISCOSITY =
    <real>hourglass_visc_value(solid = 0.0,
      shell/membrane = 0.0)
  MEMBRANE HOURGLASS STIFFNESS =
    <real>memb_hourglass_stiff_value(0.05)
  MEMBRANE HOURGLASS VISCOSITY =
    <real>memb_hourglass_visc_value(0.0)
  BENDING HOURGLASS STIFFNESS =
    <real>bend_hourglass_stiff_value(0.05)
  BENDING HOURGLASS VISCOSITY =
    <real>bend_hourglass_visc_value(0.0)
  TRANSVERSE SHEAR HOURGLASS STIFFNESS =
    <real>tshr_hourglass_stiff_value(0.05)
  TRANSVERSE SHEAR HOURGLASS VISCOSITY =
    <real>tshr_hourglass_visc_value(0.0)
  EFFECTIVE MODULI MODEL =
    <string>PRONTO|ELASTIC|PROBED(PRONTO/ELASTIC)
  MINIMUM EFFECTIVE DILATATIONAL MODULI RATIO =
    <real>min_eff_mod_ratio(1.0e-3/1.0e-6/0.0)
  MINIMUM EFFECTIVE SHEAR MODULI RATIO =
    <real>min_eff_mod_ratio(0.05/1.0e-6/0.0)
  ELEMENT NUMERICAL FORMULATION = <string>OLD|NEW(OLD)
  ACTIVE FOR PROCEDURE <string>proc_name DURING PERIODS
    <string list>period_names
```

```
  INACTIVE FOR PROCEDURE <string>proc_name DURING PERIODS
    <string list>period_names

END [PARAMETERS FOR BLOCK <string list>block_names]
```

Alternate syntax:

```
BEGIN BLOCK [<string list>block_names]
...
END [BLOCK <string list>block_names]
```

The finite element model consists of one or more element blocks. Associated with an element block or group of element blocks will be a `PARAMETERS FOR BLOCK` command block, which is also referred to in this document as an *element-block command block*. The basic information about the element blocks (number of elements, topology, connectivity, etc.) is contained in a mesh file. Specific attributes for an element block must be specified in the input file. If for example, a block of eight-node hexahedra is to use the selective-deviatoric versus mean-quadrature formulation, then the selective-deviatoric formulation must be specified in the input file. The element library is listed at the beginning of Section 6.1.

The element-block command block begins with the input line

```
BEGIN PARAMETERS FOR BLOCK [<string list>block_names]
```

and is terminated with the input line

```
END [PARAMETERS FOR BLOCK <string list>block_names]
```

Here, `block_names` is a list of element blocks assigned to the element-block command block. Such a list must be included on the `BEGIN PARAMETERS FOR BLOCK` input line if the `INCLUDE ALL BLOCKS` line command is not used. If the format for the mesh file is Exodus II, the typical form of a `block_name` is `block_integerID`, where `integerID` is the integer identifier for the block. If the element block is 280, the value of `block_name` would be `block_280`. It is also possible to generate an alias identifier for the element block and use this for the `block_name`. If `block_280` is aliased to `AL6061`, then `block_name` becomes `AL6061`.

All element blocks listed on the `PARAMETERS FOR BLOCK` command line—or, alternatively, all element blocks included using the line command `INCLUDE ALL BLOCKS` in conjunction with `REMOVE BLOCK`—must share the same mechanics properties. The command line `MATERIAL` associates a particular material `material_name` with all elements in the block. The `SECTION` command specifies a particular element formulation for the block. The specified section differentiates between elements with the identical topology but different formulations; for example, a block of four-node quadrilateral topology may specify a membrane or shell formulation. Depending on the element formulation, the `SECTION` command line may also allow the assignment of a variety of parameters to an element block.

All command lines that can be used in a `PARAMETERS FOR BLOCK` command block are described in Sections 6.1.7.1 through 6.1.7.9.

6.1.7.1. Material Property

```
MATERIAL = <string>material_name
MODEL = <string>model_name
DENSITY SCALE FACTOR = <real>density_scale_factor(1.0)
```

The material property specification for an element block is done by using the above three command lines. The property specification references both a `MATERIAL` command block and a material-model command block, which has the general form `PARAMETERS FOR MODEL model_name`. These command blocks are described in Chapter 5. The `MATERIAL` command block contains all the parameters needed to define a material, and is associated with an element block (`PARAMETERS FOR BLOCK` command block) by use of the `MATERIAL` command line. Some of the material parameters inside the property specification are grouped on the basis of material models. A material-model command block is associated with an element block by use of the `MODEL` command line.

An optional `DENSITY SCALE FACTOR` may be specified. This command scales the density of the material (described in Chapter 5) before applying it to the block density. Users may find this command useful if they wish to modify the density for a given block without duplicating the entire material model.

Consider the following example. Suppose there is a `MATERIAL` command block with a `material_name` of steel. Embedded within this command block for steel is a material-model command block for an elastic model of steel and an elastic-plastic model of steel. Suppose that for the current element block it is desired to use the material steel with the elastic model. Then the element-block command block would contain the input lines

```
MATERIAL = steel
MODEL    = elastic
```

If, on the other hand, the elastic-plastic model is desired with a density of 95% of that specified in the steel material model, the element-block command block would contain

```
MATERIAL              = steel
MODEL                 = elastic_plastic
DENSITY SCALE FACTOR = 0.95
```



Warning: Density scaling currently does not work in conjunction with equation-of-state (EOS) material models or material models that set the density inside the model (e.g., `KAYENTA`).

6.1.7.2. Include All Blocks

The `INCLUDE ALL BLOCKS` line command is used to associate all element blocks with the same element parameters (which minimizes input).

```
INCLUDE ALL BLOCKS
```

6.1.7.3. Remove Block

The REMOVE BLOCK command line allows the user to delete blocks from the set specified in the PARAMETERS FOR BLOCK command block or INCLUDE ALL BLOCKS command line(s) through the string list block_names.

```
REMOVE BLOCK <string>block_list
```

6.1.7.4. Section

```
SECTION = <string>section_id
```

The section specification for an element-block command block is done by using the above command line. The section_id is a string associated with a section command block. The SECTION command line must specify the unique identifier or name of a SECTION command block defined by the user elsewhere in the input. The chosen section must also be compatible with the topology of the block as specified in the input mesh file. It is within the SECTION command block that element formulation-specific properties are given. Currently available section command blocks are

- SOLID SECTION,
- COHESIVE SECTION,
- SHELL SECTION,
- MEMBRANE SECTION,
- BEAM SECTION,
- TRUSS SECTION,
- SPRING SECTION,
- DAMPER SECTION,
- POINT MASS SECTION,
- PARTICLE SECTION, and
- SUPERELEMENT SECTION.

Solid sections are only compatible with three-dimensional solid topologies; cohesive, shell, and membrane sections are only compatible with two-dimensional topologies; beam, truss, spring, and damper sections are only compatible with one-dimensional mesh topologies; while point mass, and particle sections are only compatible with sphere topology in the input mesh. Many additional details on the various SECTION command blocks are given in Section 6.2.

If no SECTION command line is present in a PARAMETERS FOR BLOCK command block, the default eight-node hexahedral formulation based on mean quadrature and midpoint-increment strain incrementation is assumed.

Suppose a membrane formulation is desired for a given element block. First, a MEMBRANE SECTION command block with a unique name such as membrane_rubber must be defined elsewhere in the input file. Inside the desired element-block command block, one would then specify

```
SECTION = membrane_rubber
```

to properly assign the membrane element formulation to the block. Specific section properties, such as the thickness of the membrane, would be described in the `MEMBRANE SECTION` command block and then associated with the current element-block command block.

Only one `SECTION` command line is allowed per element-block command block. Several element-block command blocks may reference the same section command block. For example, in Figure 6-1, the section named `membrane_rubber` appears in two different `PARAMETERS FOR MODEL` command blocks, while the associated `MEMBRANE SECTION` command block is defined only once.

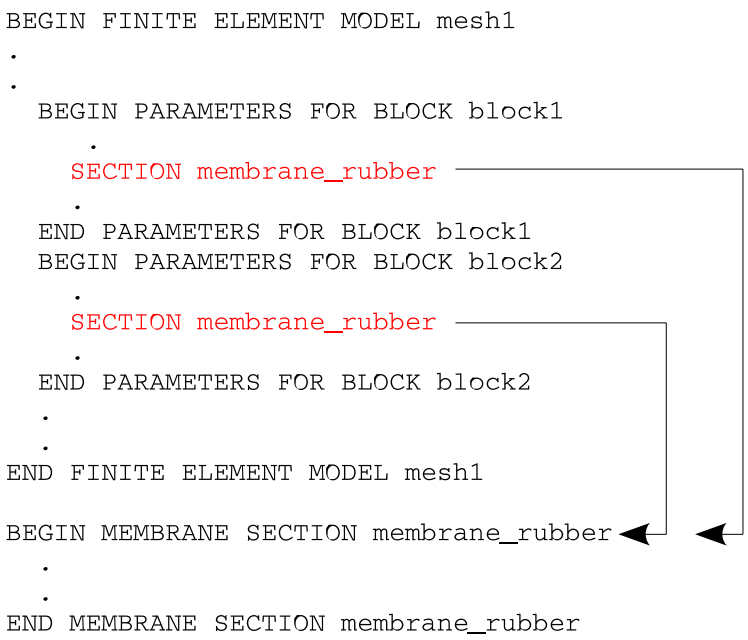


Figure 6-1. Association between `SECTION` command lines and a section command block.

6.1.7.5. Linear and Quadratic Bulk Viscosity

```
LINEAR BULK VISCOSITY =
  <real>linear_bulk_viscosity_value(0.06)
QUADRATIC BULK VISCOSITY =
  <real>quad_bulk_viscosity_value(1.20)
```

The linear and quadratic bulk viscosity are set with these two command lines. Consult the documentation for the elements [5] for a description of the bulk viscosity parameters.

6.1.7.6. Hourglass Control

```
HOURGLASS STIFFNESS = <real>hourglass_stiff_value(0.05)
```

```
HOURGLASS VISCOSITY = <real>hourglass_visc_value(0.0)
MEMBRANE HOURGLASS STIFFNESS =
  <real>memb_hourglass_stiff_value(0.05)
MEMBRANE HOURGLASS VISCOSITY =
  <real>memb_glass_visc_value(0.0)
BENDING HOURGLASS STIFFNESS =
  <real>bend_hourglass_stiff_value(0.05)
BENDING HOURGLASS VISCOSITY =
  <real>bend_glass_visc_value(0.0)
TRANSVERSE SHEAR HOURGLASS STIFFNESS =
  <real>tshr_hourglass_stiff_value(0.05)
TRANSVERSE SHEAR HOURGLASS VISCOSITY =
  <real>tshr_glass_visc_value(0.0)
HOURGLASS TRANSITION STRAIN =
  <real>hourglass_transition_strain_value(1.0)
HOURGLASS EXPONENT = <real>hourglass_exponent_value(1.0)
```

These command lines set the hourglass control parameters for elements that use hourglass control. Currently, the included elements are the eight-node, uniform-gradient hexahedral elements; the eight-node and ten-node tetrahedral elements; and the four-node membrane and shell elements. Consult the element documentation [5] for a description of the hourglass parameters.

Hourglass stiffness and viscosity parameters for under integrated elements are set using the `HOURGLASS STIFFNESS` and `HOURGLASS VISCOSITY` commands, respectively. If either of these commands are used for shell elements, they set the hourglass stiffness or viscosity for all three modes (membrane, bending, and transverse shear).

Hourglass parameters for the membrane, bending, and transverse shear modes can be set individually for shell elements. The membrane hourglass stiffness and viscosity can be set with the `MEMBRANE HOURGLASS STIFFNESS` and `MEMBRANE HOURGLASS VISCOSITY` commands. These membrane hourglass commands can also be used with membrane elements. The bending hourglass stiffness and viscosity are set with the `BENDING HOURGLASS STIFFNESS` and `BENDING HOURGLASS VISCOSITY` commands, and transverse shear hourglass stiffness and viscosity are set with the `TRANSVERSE SHEAR HOURGLASS STIFFNESS` and `TRANSVERSE SHEAR HOURGLASS VISCOSITY` commands. These commands will override either the default values and any value set in the generic `HOURGLASS STIFFNESS` and/or `HOURGLASS VISCOSITY` commands for the particular mode that is specified.

The hourglass stiffness parameter defaults to 0.05 for all elements using hourglass control. The hourglass viscosity parameter defaults to 0.0 for all elements using hourglass control.

Hourglass resistances are scaled by the material effective modulus. The hourglass resistance can be strongly affected by the method that computes the effective moduli. The command line in Section 6.1.7.7 selects the method for computing the effective moduli.

The `HOURGLASS TRANSITION STRAIN` and `HOURGLASS EXPONENT` commands only apply to the `HYPERELASTIC` hourglass formulation. The hyperelastic hourglass is a nonlinear hourglass control formulation. The nonlinearity becomes prominent once the hourglass strain reaches the `HOURGLASS TRANSITION STRAIN`. The strength of the nonlinearity is controlled by the

HOURGLASS EXPONENT, where the default exponent of 1.0 gives a linear hourglass force-displacement relation. For problems where the hourglass modes are highly excited, an exponent of 2.0 with a transition strain of $\tilde{0}.025$ is recommended to mitigate the possibility of element inversion. With these settings, the hourglass stiffness increases so that the hourglass models can hold more energy in extreme deformation situations, increasing simulation robustness but perhaps reducing accuracy. See the Sierra/SM 5.0 Theory Manual for more details.

6.1.7.7. Effective Moduli Model

```
EFFECTIVE MODULI MODEL =
  <string>PRONTO|ELASTIC|PROBED(PRONTO/ELASTIC)
```

The element-based critical time step and hourglass force computations require a measure of the material moduli, which is given in Sierra/SM by the effective moduli. The CFL condition dictates that the critical time step is directly related to the current material moduli, and the hourglass forces must be appropriately scaled based on the material moduli. For elastic, isotropic material models, the moduli are constant throughout the analysis. However, for nonlinear materials, the moduli are typically computed numerically from the stresses. For models with softening regimes or that approach perfect plasticity, the moduli may be difficult to define, and the way in which they are computed may adversely affect the analysis. Through the EFFECTIVE MODULI MODEL command line, Presto provides several methods for the computation of these effective moduli:

PRONTO This method is the default when running an explicit calculation and is identical to the method of computing effective moduli present in the Pronto3D code. This method will compute an approximate tangent modulus based on current material response. The PRONTO method includes some bounds on admissible tangent stiffness values to promote stability. The PRONTO method can produce poor results in analyses with materials that undergo substantial hardening; the PROBED modulus may be a better option for such materials.

PROBED This method calls the material stiffness calculations twice each step to compute a true tangent probe of the material stiffness. It is generally more accurate and robust than the PRONTO method, although more expensive, as it requires two rather than one material call per time step. The PROBED option may be worth using for models were the material stiffness changes greatly (such as compressible foams). Note the probe modulus is a probe of the instantaneous tangent modulus and may have difficulty taking in to the account of rate-dependent effects such as creep or strain rate hardening.

MATERIAL A handful of material models are able to compute an analytic tangent stiffness. Currently this includes only the HYPER FOAM and the MOONEY RIVLIN material models. When the analytic result is available it provides the most robust and most efficient estimate of the tangent stiffness. The PRONTO and PROBED methods will both use the analytic tangent modulus if possible and revert to a numerical tangent calculation for materials that do not compute an analytic tangent modulus.

ELASTIC This method uses the initial elastic moduli for the entire analysis. For materials that harden substantially (such as compressible foams) the elastic method could produce moduli late in the analysis that are too low causing stability problems. For materials the have

Table 6-1. Default moduli ratios for the various effective moduli models.

Model	Dilatational ratio	Shear ratio
PRONTO	0.001	0.05
PROBED	1.0E−6	1.0E−6
ELASTIC	1.0	1.0

constant stiffness or if the elastic constants are set in a way to be conservative for the entire analysis routine the elastic modulus method may be the most robust. The elastic modulus is the default method when running an implicit calculation.

The EFFECTIVE MODULI MODEL command line should be used with caution because it can strongly affect the analysis results.

The commands

- MINIMUM EFFECTIVE DILATATIONAL MODULI RATIO and
- MINIMUM EFFECTIVE SHEAR MODULI RATIO

define the minimum effective modulus that can be computed for a material with respect to the initial elastic modulus. This value prevents the computed tangent modulus from dropping to close to zero for yielding or softening materials. A larger value for this constant can sometimes promote stability by allowing the effective modulus to track with material stiffness, but not bottom out at such a low value that low stiffness instability can occur. The computed effective dilatational modulus value primarily affects timestep and bulk viscosity calculations (which can affect the speed of elastic wave propagation). The computed effective shear modulus primarily affects hourglass control in the elements. For historical reasons the default value for these parameters depends on the effective moduli model used; the default ratios are listed in Table 6-1.



6.1.7.8. Element Numerical Formulation

```
ELEMENT NUMERICAL FORMULATION = <string>OLD|NEW(OLD)
```

For calculation of the critical time step, it is necessary to determine a characteristic length for each element. In one dimension, the correct characteristic element length is the distance between the two nodes of the element. In higher dimensions, this length is usually taken to be the minimum distance between any of the nodes in the element. However, some finite element codes, primarily those based on Pronto3D [1], use as a characteristic length an eigenvalue estimate based on work by Flanagan and Belytschko [6]. That characteristic length provides a stable time step, but often is far more conservative than the minimum distance between nodes. For a cubic element with side length equal to 1, and thus also surface area of each face and volume equal to 1, the minimum distance between nodes is 1. However, the eigenvalue estimate is $\frac{1}{\sqrt{3}}$, which is only 58% of the minimum distance. As the length of the element is increased in one direction while keeping surfaces in the lateral direction squares of area 1, the eigenvalue estimate asymptotes to $\frac{1}{\sqrt{2}}$ for very long elements. If the length is decreased, the eigenvalue estimate asymptotes to the minimum distance between nodes for very thin elements. In this case, the eigenvalue estimate is

always more conservative than the minimum distance between nodes. However, consider an element whose cross section in one direction is not a square but a trapezoid with one side length much greater than the other. Assume the large side length is 1 and the other side length is arbitrarily small, ε. In this case, the minimum distance between nodes becomes ε, creating a small and inefficient time step. However, the eigenvalue estimate is related to the length across the middle of the trapezoid, which for the conditions stated is $\frac{1}{2}$. Since both distances provide stable time steps, and one or the other can be much larger in various circumstances, the most efficient calculation is obtained by using the maximum of the two lengths, either the eigenvalue estimate or the minimum distance between nodes, to determine the time step.

By using the maximum of the lengths, the computed critical time step will be very nearly unstable, and the `TIME STEP SCALE FACTOR` command line should be used to provide a margin of safety; the scale factor should not be greater than 0.9, and sometimes it may need to be reduced further. Although this time step estimate is closer to the critical value and provides better accuracy and efficiency, a smaller scale factor may need to be specified for stability. For this reason, the choice of approach to use is left to the user and is determined by the command line

```
ELEMENT NUMERICAL FORMULATION = <string>OLD|NEW(OLD)
```

If the input parameter is `OLD`, only the eigenvalue estimate is used; `NEW` means that the maximum of the two lengths is used. The default is `OLD` so that users will have to specifically choose the new approach and be aware of the scale factor for the time step.

The `ELEMENT NUMERICAL FORMULATION` command line is applicable to both the energy-dependent and purely mechanical material models. If this command line is applied to blocks using energy-dependent materials, only the determination of the characteristic length is affected. If this command line is applied to an element block with a purely mechanical model and the `OLD` option is used, the Pronto3D-based artificial viscosity, time step, and eigenvalue estimate will be used in the element calculations. If, however, the `NEW` option is used, the artificial viscosity and time step will be computed from equations associated with the energy-dependent models. Further details about the critical time step calculations and the use of this command line are available in [7].

6.1.7.9. Activation/Deactivation of Element Blocks by Time

```
ACTIVE FOR PROCEDURE <string>proc_name DURING PERIODS
  <string list>period_names
INACTIVE FOR PROCEDURE <string>proc_name DURING PERIODS
  <string list>period_names
```

This command line permits the activation and deactivation of element blocks by time period. The time periods are defined in the `TIME STEPPING BLOCK` command block (Section 3.1.1) within a specific procedure named in a `PRESTO PROCEDURE` or an `ADAGIO PROCEDURE` command block (Section 2.2.1).

The `ACTIVE FOR PROCEDURE` or `INACTIVE FOR PROCEDURE` command lines can optionally be used to deactivate element blocks for a portion of the analysis. If the

ACTIVE FOR PROCEDURE command is used, the element block is active for all periods listed for the named procedure, and is deactivated for all time periods that are absent from the list. If the INACTIVE FOR PROCEDURE command is used, the element block is deactivated for all periods listed for the named procedure. The element block is active for all time periods that are absent from the list. If neither command line is used, by default the block is active during all time periods. This command line controls the activation and deactivation of all elements in a block. Alternatively, individual elements can be deactivated with the ELEMENT DEATH command block (see Section 6.5).



Warning: Inactive element's corresponding nodes will not move while they are inactive except when the nodes are shared with active elements. It is therefore possible that inactive elements can be inverted while inactive and trigger a fatal error when activated.



Known Issue: *Implicit* Deactivation of element blocks does not currently work in conjunction with the full tangent preconditioner (see Section 4.3) in Adagio. To use this capability, one of the nodal preconditioners must be used.



Known Issue: Deactivating a block only turns off the element/material computations and does not remove the blocks degrees of freedom from the solution matrix. Thus if the nodes have some external force applied convergence may be impossible in implicit or the nodes may accelerate to infinite velocity in explicit. It is recommended that on time periods that a block is deactivated a kinematic boundary condition that fixes all degrees of freedom of the deactivated block also be used.

6.1.8. Element Block Default Parameters

```
BEGIN BLOCK DEFAULTS
  MATERIAL = <string>material_name
  MODEL    = <string>model_name
  SECTION  = <string>section_id
  LINEAR BULK VISCOSITY =
    <real>linear_bulk_viscosity_value(0.06)
  QUADRATIC BULK VISCOSITY =
    <real>quad_bulk_viscosity_value(1.20)
  HOURGLASS STIFFNESS =
    <real>hourglass_stiff_value(0.05)
  HOURGLASS VISCOSITY =
  <real>hourglass_visc_value(0.0)
  EFFECTIVE MODULI MODEL =
    <string>PRONTO|ELASTIC|PROBED(PRONTO/ELASTIC)
  MINIMUM EFFECTIVE SHEAR MODULI RATIO =
    <real>min_eff_mod_ratio
```

```
    MINIMUM EFFECTIVE DILATATIONAL MODULI RATIO =
      <real>min_eff_mod_ratio
  END [PARAMETERS FOR BLOCK <string list>block_names]
```

The block defaults block may be used modify the default values for commands used inside the `BEGIN PARAMETERS FOR BLOCK` blocks within the same `BEGIN FINITE ELEMENT MODEL` command block. See section 6.1.7 for the description of each command.

Block defaults can be used to simplify input when a model has a large number of blocks. For example to try an alternative effective moduli method or hourglass stiffness at all blocks at once. If a given parameter is set in both block defaults and a specific block the specific block value overrides the block default value. The following examples demonstrate use of the block defaults command.

Example 1:

```
begin block defaults
  hourglass stiffness = 0.10
  hourglass viscosity = 0.01
end
begin parameters for block block_1
  material = steel
  model    = elastic
end
begin parameters for block block_2
  material = rubber
  model    = mooney_rivlin
end
```

In this example the hourglass stiffness and viscosity defaults have been changed for all blocks. This would be equivalent to the following input:

```
begin parameters for block block_1
  material = steel
  model    = elastic
  hourglass stiffness = 0.10
  hourglass viscosity = 0.01
end
begin parameters for block block_2
  material = rubber
  model    = mooney_rivlin
  hourglass stiffness = 0.10
  hourglass viscosity = 0.01
end
```

Example 2:

```
begin block defaults
  material = aluminum
  model    = elastic_plastic
  section  = solid_1
end
begin parameters for block block_1
end
begin parameters for block block_2
end
begin parameters for block block_3
end
begin parameters for block block_4
  material = tungsten
  model    = elastic
end
```

A default material is used for blocks one two and three, but the defaults are partially overridden for block 4. This would be equivalent to the following input:

```
begin parameters for block block_1 block_2 block_3
  material = aluminum
  model    = elastic_plastic
  section  = solid_1
end
begin parameters for block block_4
  material = tungsten
  model    = elastic
  section  = solid_1
end
```

6.2. Element Sections

Element sections are defined by section command blocks. There are currently nine different types of section command blocks. The section command blocks appear in the SIERRA scope, at the same level as the FINITE ELEMENT MODEL command block. In general, a section command block has the following form:

```
BEGIN section_type SECTION <string>section_name
  command lines dependent on section type
END [section_type SECTION <string>section_name]
```

Currently, section_type can be SOLID, COHESIVE, SHELL, MEMBRANE, BEAM, TRUSS, SPRING, DAMPER, POINT MASS, PARTICLE, or SUPERELEMENT. These various section types are identified as individual section command blocks and are described below. The corresponding section_name parameter in each of these command blocks, e.g., truss_section_name in the TRUSS SECTION command block, is selected by the user. The method used to associate these names with individual SECTION command lines in PARAMETERS FOR BLOCK command blocks is discussed in Section 6.1.7.4.

Following is a list of most of the different element formulations that are offered in Sierra/SolidMechanics. The sections are colored red, green, and blue based on where the documentation can be found. Red sections do not have any documentation posted; prospective users should contact a developer before use. Green sections can be found in the Sierra/SM 5.0 Capabilities in Development manual. Blue links to the relevant section in this manual.

Element	# nodes	Section	Formulation
ANY	ANY	RIGID BODY	
		SUPERELEMENT	
Sphere	1	DEM	
		PARTICLE	SPH POINT MASS
		PERIDYNAMICS	
BAR	2	BEAM	
		BOLT	
		CYLINDRICAL JOINT	
		DAMPER	
		LINEAR BEAM	
		SPRING	
		SPRING SUPPORT	
		TRUSS	
		USERMOUNT	

Element	# nodes	Section	Formulation	Strain Incrementation
Shell	3	SHELL	ORIG_TRI_SHELL	MIDPOINT_INCREMENT STRONGLY_OBJECTIVE
			C0_TRI_SHELL	MIDPOINT_INCREMENT STRONGLY_OBJECTIVE
	4	COHESIVE		
		MEMBRANE	MEAN_QUADRATURE MEAN_QUADRATURE	MIDPOINT_INCREMENT STRONGLY_OBJECTIVE
			SELECTIVE_DEVIATORIC	MIDPOINT_INCREMENT
		SHELL	FIBER_SHELL	MIDPOINT_INCREMENT STRONGLY_OBJECTIVE
			NQUAD	
			SEVEN_PARAMETER	
			BT_SHELL	MIDPOINT_INCREMENT
			KH_SHELL	MIDPOINT_INCREMENT STRONGLY_OBJECTIVE
			BT_SC_SHELL	MIDPOINT_INCREMENT STRONGLY_OBJECTIVE
			BL_SHELL	MIDPOINT_INCREMENT STRONGLY_OBJECTIVE
			HEX_SHELL	MIDPOINT_INCREMENT STRONGLY_OBJECTIVE
TETRA		SOLID	MEAN_QUADRATURE	STRONGLY_OBJECTIVE NODE_BASED
			FULLY_INTEGRATED	STRONGLY_OBJECTIVE NODE_BASED
			VOID	
PYRAMID	5	SOLID	MEAN_QUADRATURE	MIDPOINT_INCREMENT STRONGLY_OBJECTIVE TOTAL_LAGRANGE
WEDGE	6	COHESIVE		
		LOCAL		
		SOLID	MEAN_QUADRATURE	MIDPOINT_INCREMENT STRONGLY_OBJECTIVE TOTAL_LAGRANGE
HEX	8	COHESIVE		
		LOCAL		
		SOLID	FULLY_INTEGRATED	
			MEAN_QUADRATURE	MIDPOINT_INCREMENT STRONGLY_OBJECTIVE TOTAL_LAGRANGE
			SELECTIVE_DEVIATORIC	STRONGLY_OBJECTIVE
			THICKSHELL	
			VOID	
TETRA	8	**NOT SUPPORTED**		
TETRA	10	SOLID	(NONE_SPECIFIED = COMPOSITE_TET)	STRONGLY_OBJECTIVE
			FULLY_INTEGRATED	
HEX	20	SOLID	FULLY_INTEGRATED	STRONGLY_OBJECTIVE
HEX	27	SOLID	FULLY_INTEGRATED	STRONGLY_OBJECTIVE

TOTAL LAGRANGE SECTION

Element	# nodes	Formulation	Strain Incrementation	cubature
TETRA	4	FULLY_INTEGRATED	LOGARITHMIC_MAPPING STRONGLY_OBJECTIVE	1
			LOGARITHMIC_MAPPING STRONGLY_OBJECTIVE	2
			LOGARITHMIC_MAPPING STRONGLY_OBJECTIVE	3
HEX	8	FULLY_INTEGRATED	LOGARITHMIC_MAPPING STRONGLY_OBJECTIVE	ANY
TETRA	10	COMPOSITE_TET	LOGARITHMIC_MAPPING STRONGLY_OBJECTIVE	2
			LOGARITHMIC_MAPPING STRONGLY_OBJECTIVE	3
		FULLY_INTEGRATED	LOGARITHMIC_MAPPING STRONGLY_OBJECTIVE	2
			LOGARITHMIC_MAPPING STRONGLY_OBJECTIVE	3
			LOGARITHMIC_MAPPING STRONGLY_OBJECTIVE	4
PYRAMID	13	FULLY_INTEGRATED	LOGARITHMIC_MAPPING STRONGLY_OBJECTIVE	ANY
WEDGE	15	FULLY_INTEGRATED	LOGARITHMIC_MAPPING STRONGLY_OBJECTIVE	ANY
HEX	20	FULLY_INTEGRATED	LOGARITHMIC_MAPPING STRONGLY_OBJECTIVE	ANY
HEX	27	FULLY_INTEGRATED	LOGARITHMIC_MAPPING STRONGLY_OBJECTIVE	3
			LOGARITHMIC_MAPPING STRONGLY_OBJECTIVE	5

6.2.1. Solid Section

```
BEGIN SOLID SECTION <string>solid_section_name
  RVE COORDINATE SYSTEM = <string>Coordinate_system_name
  FORMULATION = <string>MEAN_QUADRATURE|SELECTIVE_DEVIATORIC|
    FULLY_INTEGRATED|VOID(MEAN_QUADRATURE)
  DEVIATORIC PARAMETER = <real>deviatoric_param
  STRAIN INCREMENTATION = <string>MIDPOINT_INCREMENT|
    STRONGLY_OBJECTIVE|NODE_BASED|TOTAL_LAGRANGE(depends)
  STRESS MEASURE = CAUCHY|HENCKY(CAUCHY)
  NODE BASED ALPHA FACTOR = <real>bulk_stress_weight(0.01)
  NODE BASED BETA FACTOR = <real>shear stress_weight(0.35)
  NODE BASED STABILIZATION METHOD = <string>EFFECTIVE_MODULI|
    MATERIAL(MATERIAL)
  HOURGLASS FORMULATION = <string>TOTAL|INCREMENTAL|
    HYPERELASTIC(INCREMENTAL)
  HOURGLASS INCREMENT = <string>ENDSTEP|MIDSTEP (ENDSTEP)
  HOURGLASS ROTATION = <string> APPROXIMATE|SCALED (APPROXIMATE)
  RIGID BODY = <string>rigid_body_name
  RIGID BODIES FROM ATTRIBUTES = <integer>first_id
    TO <integer>last_id
  MASS LUMPING METHOD = DEFAULT|ROWSUM|EVEN (DEFAULT)
END [SOLID SECTION <string>solid_section_name]
```

The `SOLID SECTION` command block specifies the properties for solid elements (hexahedra and tetrahedra). This command block is to be referenced by an element block made up of solid elements. The two types of solid-element topologies currently supported are hexahedra and tetrahedra. The parameter `solid_section_name` is user-defined and is referenced by a `SECTION` command line in a `PARAMETERS FOR BLOCK` command block.

The `RVE COORDINATE SYSTEM` command line specifies the name of a coordinate system definition block command used to define a local coordinate system on each element of a block that uses this `SOLID SECTION`. This coordinate system only has an effect on RVE analysis (see the Sierra/SM 5.0 Capabilities in Development manual for details).

The `FORMULATION` command line specifies whether the element will use a single-point integration rule (mean quadrature), a selective-deviatoric rule, a selectively integrated rule, or a fully-integrated rule, or act as a void element. The selective-deviatoric and fully-integrated integration rules are higher-order integration schemes, which are discussed below.

If the user wishes to use the selective-deviatoric rule, the `DEVIATORIC PARAMETER` command line must also appear in the `SOLID SECTION` command block.

Choosing the `SELECTIVE_DEVIATORIC` formulation indicates that the element is fully integrated with respect to the deviatoric stress response and under integrated with respect to the hydrostatic pressure response. The under-integration of the hydrostatic pressure response is achieved through averaging the pressure over the integration points. The `SELECTIVE_DEVIATORIC` formulation is often used to alleviate volumetric locking associated with isochoric motion. By default, this formulation involves no hourglass control. However, the user may include an additional input

command within the element section, DEVIATORIC PARAMETER, which may set deviatoric_param between 0.0 and 1.0, and which blends the MEAN_QUADRATURE and SELECTIVE_DEVIATORIC integration schemes. The MEAN_QUADRATURE and SELECTIVE_DEVIATORIC schemes are recovered at 0.0 and 1.0, respectively. By default, this parameter is set to 1.0, which is the recommended value.

The selective-deviatoric elements, when used with a value greater than 0.0, provide hourglass control without artificial hourglass parameters.

Alternatively, a fully-integrated rule can be defined by setting FORMULATION = FULLY_INTEGRATED. This will apply an eight-point integration scheme for eight-noded hexahedral elements and a four-point integration scheme for ten-noded tetrahedral elements. This definition is currently unavailable for four-noded and eight-noded tetrahedral elements.

The VOID formulation is valid for 8-node hexahedral and 4-node tetrahedral element blocks. Void elements only compute volume. They do not contribute internal forces to the model. The material model and density associated with void elements are ignored. The volume and the first and second derivatives of the volume for each element are stored in the element variables volume, volume_first_derivative, and volume_second_derivative. The volume derivatives are computed using least squares fits of the volume history, which is stored for the previous five time steps.

In addition to the per-element volume and derivatives, the total volume and derivatives of that total volume for all elements in each void element block are written to the results file as global variables. The names for these variables are voidvol_blockID, voidvol_first_deriv_blockID, and voidvol_second_deriv_blockID. In these global variable names, blockID is the ID of the block. For example, the void volume for block 8 would be stored in voidvol_8.

The STRAIN INCREMENTATION command defines the strain incrementation formulation to use for solid element types that support multiple formulations. The element summary at the beginning of Section 6.1 specifies the strain incrementation formulations available for each element. Many elements have two choices for the strain incrementation formulation: MIDPOINT_INCREMENT or STRONGLY_OBJECTIVE. Generally, the midpoint-increment formulation is slightly faster and the strongly-objective formulation is slightly more accurate. Consult [2, 5] for descriptions of these strain formulations.

The four-noded tetrahedron has a NODE_BASED option for the STRAIN INCREMENTATION, which indicates usage of the node-based tetrahedron (see Section 6.2.1.1).

The eight-noded mean-quadrature hexahedron has a TOTAL_LAGRANGE option for the STRAIN INCREMENTATION, which is based on the total deformation and is provided for usage with the HYPERELASTIC hourglass formulation.

The STRESS MEASURE option is specifically for the eight-noded mean-quadrature hexahedron with the TOTAL_LAGRANGE strain incrementation and the HYPERELASTIC hourglass formulation. The HENCKY option is slightly more accurate for the hyperelastic hourglass formulation, but CAUCHY (the default) is necessary for optimal compatibility with the material models.

The HOURGLASS FORMULATION command is used to switch between total, hyperelastic and incremental forms of hourglass control. The total hourglass option can only be used with eight-noded mean-quadrature hexahedral elements using strongly objective strain incrementation (STRAIN INCREMENTATION = STRONGLY_OBJECTIVE). In contrast, the hyperelastic hourglass option can only be used with eight-node mean-quadrature hexahedral elements using total-Lagrange strain incrementation (STRAIN INCREMENTATION = TOTAL_LAGRANGE). One of the following three arguments can be used with this command: TOTAL, HYPERELASTIC, or INCREMENTAL.

The total formulation performs stiffness hourglass force updates based on the rotation tensor from the polar decomposition of the total deformation gradient. The hyperelastic formulation computes hourglass strains relative to the reference/model configuration, and has a well defined hyperelastic hourglass strain energy; as a result this formulation results in a symmetric contribution to the stiffness matrix, is fully reversible, and allows for a nonlinear hourglass resistance. The incremental formulation is the default and performs stiffness hourglass force updates based on the hourglass velocities and the incremental rotation tensor. The viscous hourglass forces and the hourglass parameters are unchanged by this command. Consult the element documentation [5] for a description of the hourglass forces and the incremental hourglass formulation.

The HOURGLASS INCREMENT and HOURGLASS ROTATION commands control the speed and accuracy of the hourglass control computation. These commands are only applicable to the mean-quadrature hexahedron with midpoint strain incrementation (STRAIN INCREMENTATION = MIDPOINT_INCREMENT). The HOURGLASS INCREMENT line command specifies whether the hourglass resistance increment is to be computed at the middle or end of the time step. The end-step calculation has a slightly reduced computational cost while the mid-step computation is more accurate. The default is ENDSTEP. The HOURGLASS ROTATION command controls whether the hourglass resistance will be scaled after rotation to preserve the magnitude. Scaling requires additional computation time but will be more accurate, particularly when large rotations are present in the analysis. The default is APPROXIMATE, meaning no scaling is done.

Rigid elements in a section are indicated by including the RIGID BODY command line. The RIGID BODY command line specifies an identifier that maps to a rigid body command block. See Section 6.3.1 for a full discussion of how to create rigid bodies and Section 6.3.1.1 for information on the use of the RIGID BODIES FROM ATTRIBUTES command.



Known Issue: For problems with large rotations the hourglass energies are known to spike exponentially with increases in total rotation. This may be due to a coupling with hypoelastic material models.

6.2.1.1. Node-Based Tetrahedra

The node-based formulation can only be used with four node tetrahedral elements. If the element-block (i.e., PARAMETERS FOR BLOCK) command block has a SECTION command line that references a SOLID SECTION command block that uses

```
STRAIN INCREMENTATION = NODE_BASED
```

then the tetrahedral element block will use the node-based tetrahedron formulation.

The node-based formulation calculates a solution that is a mixture of an element-based formulation (information from the center of an element) and a node-based formulation (information at a node that is based on the average of all elements attached to the node). The node-based tetrahedron allows a model with four-node tetrahedral elements to avoid most of the locking problems of regular tetrahedral elements. Regular tetrahedral elements are much too stiff and can produce inaccurate results in many situations.

The mixture of node-based versus element-based information incorporated into the solution may be adjusted with the `NODE BASED ALPHA FACTOR` and `NODE BASED BETA FACTOR` command lines. These two lines apply only if the `NODE_BASED` option is specified on the `STRAIN INCREMENTATION` command line. The value for `bulk_stress_weight` on the `NODE BASED ALPHA FACTOR` command line sets the element bulk stress weighting factor, while the value for `shear_stress_weight` on the `NODE BASED BETA FACTOR` command line sets the element shear stress weighting factor. The weighting factors are described in detail in [3]. A strictly node-based formulation results when both factors are set to zero, while a strictly element-based formulation results when both factors are one.

The mixing of node-based and element-based solutions for the node-based formulation is a stabilization technique to eliminate zero energy modes of a pure node-based formulation. With this in mind, two options are provided for the stress update of the element-based solution through the `NODE BASED STABILIZATION METHOD` command (applicable only with `STRAIN INCREMENTATION = NODE_BASED`). The default option, `MATERIAL`, runs the same material model in the element as is run at the nodes, with independent material states. The other option, `EFFECTIVE_MODULI`, uses a linearized update for the element stress state based on the element average value of the nodal effective moduli. The nodal effective moduli are computed from the nodal stress and strain increments at each time step. This second option avoids potential issues due to having independent node and element material states.

Node based elements store material state, stress, and strain data at the nodes in addition to at the element center. Most element based quantities such as stress have similarly named nodal equivalents.

If a node is attached to only elements of one material type that node will have one set of state, stress and strain data that is the average of the attached element quantities.

If elements of multiple materials are attached to the same node then that node will store multiple independent sets of state variables. To accommodate this the node based state data is given a unique name by appending a unique index number to the end of each state variable name.

The log file contains a table that defines the mapping from the index number the nodal element block. For example:

```
========NODAL BASED TETRAHEDRA ELEMENT BLOCK AND MATERIAL INFORMATION ==========

   Block     Nodal State Index
  -------    -----------------
  block_1      1
```

```
  block_7     2
  block_200   3

================================================================================
```

Block 1, 7, and 200 are using nodal based tetrahedra element formulation. The nodal based stress in block 7 would be found in the nodal state variable `stress_2`. If a node was on a block boundary and attached to elements of block block 1 and block 7 then that node would have both `stress_1`, the stress associated with the block 1 material, and `stress_2`, the stress associated with the block 7 material. Note that for nodal based tetrahedra these nodal stresses and material states are the most relevant quantities to visualize. Additionally note that many derived output quantities such as the von Mises stress norm are not hooked to the nodal based output quantities.

The `MASS LUMPING METHOD` is used to set the algorithm for determination of the lumped nodal masses. The `ROWSUM` option means first compute the consistent mass matrix then set the lumped mass of each node as the sum of the consistent mass matrix row. The `ROWSUM` method gives more accurate gravity loads and a more consistently stable explicit time step. Row sum is the default method for most elements. The `EVEN` option means take the total element mass and divide it evenly between the elements nodes. The `EVEN` method exists mostly to allow backwards compatibility with previous Sierra/SM versions where the even mass lumping method was always used. The `DEFAULT` method means the code automatically picks the best mass lumping option based on element type. For most elements this is the row sum method. A few elements perform better with even mass lumping and use this method as the default.

Note that there also exist other solid element formulations which can be specified in the separate `TOTAL LAGRANGE` section block. These options are currently documented in the in development manual.

6.2.1.2. 10-Node Tetrahedron Default Formulation

The `MEAN_QUADRATURE` formulation has been deprecated for 10-node tetrahedron elements for robustness and accuracy reasons. The default formulation for the 10-node tetrahedron is the `COMPOSITE_TET`. This element can be used by creating an empty solid section, by not specifying a section in the block definition, or by using a Total Lagrange section. The default values for the `COMPOSITE_TET` are `VOLUME AVERAGE J = ON` and `STRAIN_INCREMENTATION = STRONGLY_OBJECTIVE`. See section 6.2.2 for more details.

6.2.2. Total Lagrange Section: Composite Tet

```
BEGIN TOTAL LAGRANGE SECTION <string>section_name
  FORMULATION = COMPOSITE_TET
  STRAIN INCREMENTATION = <string>STRONGLY_OBJECTIVE|
    LOGARITHMIC_MAPPING(STRONGLY_OBJECTIVE)
  VOLUME AVERAGE J = <string>ON|OFF(ON)
  VEM STABILIZATION PARAMETER = <real>vem_shear_penalty(0.1)
```

```
  VEM BULK STABILIZATION PARAMETER = <real>vem_bulk_penalty(1.0e-6)
  VEM EXPONENT = <real>vem_exponent(5.0)
END [TOTAL LAGRANGE SECTION <string>section_name]
```

The composite tetrahedral element formulation can only be used with 10-noded tetrahedral elements.

As stated in Reference [8], the element is classified as composite because conceptually it consists of 12 sub-elements, each of which is a linear, 4-noded tetrahedron. Thus, the basis functions used to represent nodal fields are piecewise linear within the element. The deformation gradient and stress fields are assumed to live in a linear space, and the resulting gradient operator incorporates the projection of the piecewise discontinuous gradients into the assumed linear basis. The current implementation does not restrict the mid-edge nodes to be co-linear with the vertex nodes, allowing for better geometric resolution for curved bodies.

For hypoelastic constitutive models, the default `STRAIN INCREMENTATION` option uses a `STRONGLY_OBJECTIVE` algorithm to compute the rate of deformation. The `LOGARITHMIC_MAPPING` option is thought to be more accurate at an additional computational expense.

A volume-averaging scheme, applied to the determinant of the deformation gradient and the pressure, is used to avoid volumetric locking. Volumetric locking can occur when the material response involves isochoric motion or nearly incompressible behavior, which are common for ductile materials and rubbery materials respectively. By default the volume averaging scheme is `ON`, which averages among the four integration points of the element.

The `VOLUME AVERAGE J = ON` option avoids pressure modes but sacrifices the coercivity of the system, which may lead to displacement instabilities. A stabilization inspired by the virtual element method [9] (VEM) is provided to suppress the displacement modes with the options `VEM STABILIZATION PARAMETER = <real>vem_shear_penalty`, `VEM BULK STABILIZATION PARAMETER = <real>vem_bulk_penalty`, and `VEM EXPONENT = <real>vem_exponent`. `VEM STABILIZATION PARAMETER` and `VEM BULK STABILIZATION PARAMETER` act like hourglass stiffnesses scaled by the shear and bulk moduli, respectively, and the `VEM EXPONENT` controls how quickly the stabilization stiffens under large dilitation within the element. These VEM command options are only activated for `VOLUME AVERAGE J = ON`. The defaults are generally recommended but the most ideal options can be somewhat problem dependent, so each can be set at the user's discretion.

This is the only option in the `TOTAL LAGRANGE SECTION` that is production ready. For other information about the `TOTAL LAGRANGE SECTION` please refer to the Sierra/SM 5.0 Capabilities in Development manual.

6.2.3. Cohesive Section

```
BEGIN COHESIVE SECTION <string>cohesive_section_name
  NUMBER OF INTEGRATION POINTS = <integer>num_int_points(1)
  LAYER THICKNESS = <real>layer_thickness(0.0)
END [COHESIVE SECTION <string>cohesive_section_name]
```





Warning: In order for the cohesive elements to contribute to the global stable time step, a node based time step must be used. See Section 3.2.3 for details.

The `COHESIVE SECTION` command block specifies the properties for cohesive zone elements (quadrilateral and triangular). The name of this block (given by `cohesive_section_name`) is referenced by the element block for cohesive elements. If the option for adaptive insertion of cohesive zone elements is used, the name of this block is referenced by the `COHESIVE SECTION` command defined in Section 6.5.5.

`NUMBER OF INTEGRATION POINTS = <integer>num_int_points(1)` specifies the number of integration points for the cohesive element; the default value is one. However, it should be noted that with a single integration point, spurious hourglass-like modes may be introduced to the deformation of the cohesive element. Currently, the quadrilateral cohesive element supports either one or four integration points, while the triangular cohesive element supports either one or three integration points.

The `LAYER THICKNESS` specifies the thickness to use for determining the mass of the cohesive element layer. A cohesive element is formulated as a surface element which implies that there is no volume to the element. Specifying a layer thickness allows the cohesive element to remain a surface element but have mass. This option is especially useful for cohesive elements that are meshed independently and MPC'd to the remaining mesh. Mass is required for motion to occur in dynamics. The element wise mass `element_mass` will be the `cse_initial_area` multiplied by the `LAYER_THICKNESS` and the `DENSITY` specified in the cohesive material model.



Warning: `LAYER_THICKNESS` is not supported with shell and wedge cohesive elements.

6.2.4. *Localization Section*

```
BEGIN LOCALIZATION SECTION <string>localization_section_name
  MEAN DILATATION FORMULATION
  NUMBER OF INTEGRATION POINTS = <integer>num_int_points
  THICKNESS = <real>thickness
  MEMBRANE FORCES = ON | OFF (ON)
END [LOCALIZATION SECTION <string>localization_section_name]
```





Warning: Currently the localization elements do not contribute to the global stable time step. As such, the user must specify a time step via the `INITIAL TIME STEP` and `TIME STEP INCREASE FACTOR` command lines. See Section 3.1 for details.

6.2.4.1. Kinematics

Localization elements [10] are planar elements that lie between bulk (volumetric) elements and can employ the same underlying bulk material model. Topologically, localization elements are identical to cohesive surface elements. The reason this 2D surface element can access 3D material models is due to the multiplicative decomposition of the deformation gradient $\boldsymbol{F}$ such that $\boldsymbol{F} = \boldsymbol{F}^{\parallel}\boldsymbol{F}^{\perp}$. Each portion of the decomposition can be expressed as

$$\boldsymbol{F}^{\perp} = \boldsymbol{I} + \frac{[\![\boldsymbol{\Phi}]\!]}{h} \otimes \boldsymbol{N}, \qquad \boldsymbol{F}^{\parallel} = \boldsymbol{g}_i \otimes \hat{\boldsymbol{G}}^i \tag{6.1}$$

where $\boldsymbol{F}^{\parallel}$ encapsulates in-plane stretching and $\boldsymbol{F}^{\perp}$ reflects the displacement jump $[\![\boldsymbol{\Phi}]\!]$ in the intermediate configuration which can be pushed to the current configuration through $[\![\boldsymbol{\varphi}]\!] = \boldsymbol{F}^{\parallel}[\![\boldsymbol{\Phi}]\!]$. Figure 6-2 illustrates the kinematic construction. The jump is normalized by h

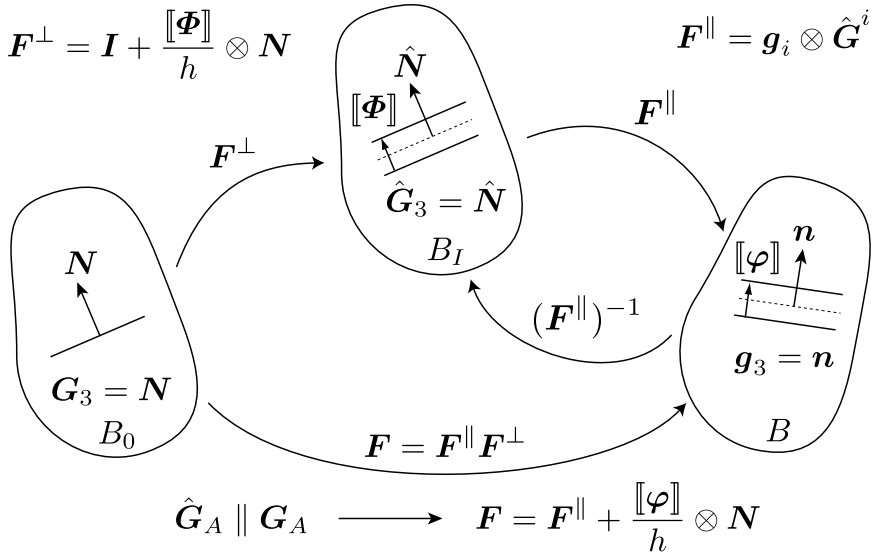


Figure 6-2. The reference B_0, intermediate B_I, and current configuration B of the body. By redefining the jump in the intermediate configuration, the formulation simplifies to an additive decomposition having contributions from membrane stretching and the jump.

which one can envision as an element thickness or a characteristic length scale governing separation. Quantities are considered to be constant through the thickness h. The curvilinear basis vectors in the reference and current configuration are $\boldsymbol{G}_A$ and $\boldsymbol{g}_i$, respectively. Given that $\boldsymbol{N}$ is constructed to be normal to the in-plane basis vectors $\boldsymbol{G}_A$ in the reference configuration, one can prove that the in-plane basis vectors in the intermediate configuration are equivalent to the in-plane basis vectors in the reference configuration ($\hat{\boldsymbol{G}}_A = \boldsymbol{F}^{\perp}\boldsymbol{G}_A = \boldsymbol{G}_A$). One can then express the multiplicative decomposition as an additive decomposition thus simplifying the initial formulation given in [10].

$$\boldsymbol{F} = \boldsymbol{F}^{\parallel} + \frac{[\![\boldsymbol{\varphi}]\!]}{h} \otimes \boldsymbol{N} \tag{6.2}$$

Because the length scale h is independent of the discretization, the methodology is regularized and ideal for employing a local, softening material model to simulate the failure process. Details concerning the force calculation are available in [10].

6.2.4.2. Command summary

The `LOCALIZATION SECTION` command block specifies the properties for localization elements (quadrilateral and triangular). The name of this block (given by `localization_section_name`) is referenced by the element block for localization elements.

`MEAN DILATATION FORMULATION` This command line will yield a constant pressure formulation. The average pressure is obtained at all integration points through a modification of the kinematic quantities. For hypo-elastic materials, tr[$\boldsymbol{D}$] is volume-averaged. For uncoupled hyperelastic models, det[$\boldsymbol{F}$] is volume-averaged. After volume averaging tr[$\boldsymbol{D}$] and det[$\boldsymbol{F}$], local dilatational contributions are removed, and additively (hypoelastic) or multiplicatively (hyperelastic) include the volume-averaged response. This results in an average pressure (by construction) and has been shown to be effective in avoiding element locking during isochoric deformations. Note that one can use a single integration point to achieve a constant pressure, but hourglass control is not provided to suppress spurious modes. If isochoric deformations are of concern, it is recommended to use a fully-integrated element with the `MEAN DILATATION FORMULATION`. The `MEAN DILATATION FORMULATION` will only be applied if the command is given in the `LOCALIZATION SECTION`.

`NUMBER OF INTEGRATION POINTS = <integer>num_int_points` The default number of integration points for a localization element depends on the topology, but is always sufficient for full integration. For an eight-noded hexahedron (planar four-noded quadrilateral) the default is four integration points while for a six-noded wedge (planar three-noded triangle) the default is one integration point. However, it should be noted that with a single integration point, spurious hourglass modes can be introduced into the deformation of the quadrilateral localization element. Currently, the quadrilateral localization element supports one and four integration points while the triangular localization element supports one integration point.

`THICKNESS = <real>thickness` The `THICKNESS` command line sets the localization element thickness h. In many respects, h should be considered a material parameter as it governs the evolution of surface separation. The introduction of h generates the true length scale in the problem: the process zone size. Care must be taken to adequately resolve the process zone size or the methodology will not be regularized.

`MEMBRANE FORCES = ON | OFF (ON)` The `MEMBRANE FORCES` command line should be used when the element size l is on the order of h. In this specific case, the membrane forces will be sufficiently large to corrupt the crack-tip fields. If $h \sim l$, we recommend that the membrane forces be `OFF`. Turning off membrane forces alters the force calculation to only employ the referential traction $\boldsymbol{T} = \boldsymbol{PN}$. In this sense, the formulation becomes "cohesive." But because $\boldsymbol{F}^{\parallel}$ is still employed in the kinematics to capture critical quantities such as the triaxiality, the stiffness matrix is no longer symmetric; however, the residual is correct. Numerous studies in implicit quasi-statics and implicit dynamics have confirmed that CG is still robust. To avoid non-monotonic reductions in the residual, one might employ solvers that can handle non-symmetric systems (ML) and solution methods other than CG (Newton with a line search) to obtain improved rates of convergence.



Warning: Domain decomposition may generate domains in which localization elements are isolated. Neither the upper or lower surface is connected to a bulk element. Because the localization element does contain a zero-energy membrane mode that is suppressed through bulk element assembly, the user will receive warnings that indicate additional rigid body modes on particular processors. To remedy the issue, the user can employ tangent diagonal scaling in the preconditioner. Studies have show this solution to be effective with minimal impact on the rate of convergence.

6.2.4.3. Usage guidelines

In contrast to cohesive surface elements, localization elements can employ bulk constitutive models for surface separation. For a typical application, the analyst might

1. Identify a constitutive model that captures the failure process. This might include a local damage model or any model that employs strain softening to facilitate strain localization.
2. Conduct mesh-dependent simulations with bulk elements of size s to understand potential paths for crack initiation and growth in specimen geometries targeted for parameterizing constitutive model parameters.
3. Seed potential paths in specimen geometries and incorporate h into the fitting process. The fitting process may not be unique in that the same far-field response might be obtained from multiple combinations of both h and the material parameters that govern the failure process. An initial guess to being the iterative process might include $h = 1/4s$. Ensure that for all cases of h that the process zone that evolves is resolved.
4. Consider the possibility of only enabling strain softening in the constitutive model that drives surface separation. Although this will assure that localization will occur in the surface elements, regulating softening to the seeded plane can introduce non-smoothness in the failure process.
5. Understand the impact of h. If h is too large, the fields governing the failure process will be less smooth and the additional compliance can create additional nonlinearities that can complicate implicit solution. Please consider refitting model parameters with lower values of h. If h is too small, the localization element thickness will drive the condition number of the implicit system. item Explore component or system level geometries with seeded paths for crack initiation and propagation. Additional simulations with bulk elements may be conducted to highlight potential failure paths. Refine the mesh to ensure that the far-field predictions are indeed mesh independent and that the process zone that evolves from the given micromechanics regularized by the length scale h is resolved.
6. Inspect of the process zone with a focus on the local metrics that drive the failure process. For example, if stress triaxiality drives void growth, one should examine fields of triaxiality in both the bulk and localization elements. In contrast to other methods like the J-integral that focus on a driving force evaluated at some radius from the crack tip, localization elements seek to capture the resistance at the crack tip. Consequently, crack-tip fields must be resolved.

7. Reflect on the fields employed for model parameterization and the fields evolving in component and system level models. If possible, align field evolution in component/system geometries with field evolution in specimen geometries. Disparities may drive the need for additional calibration experiments.

Although these usage guidelines have not focused on incorporating stochastic processes, one may sample distributions in material parameters. The inclusion of a method for regularization enables such findings in that the mesh-dependence associated with fracture/failure is not convoluted with a stochastic representation of the micro-mechanical process.

6.2.5. Shell Section

```
BEGIN SHELL SECTION <string>shell_section_name
  THICKNESS = <real>shell_thickness
  THICKNESS MESH VARIABLE =
    <string>THICKNESS|<string>var_name
  THICKNESS TIME STEP = <real>time_value
  THICKNESS SCALE FACTOR = <real>thick_scale_factor(1.0)
  INTEGRATION RULE = TRAPEZOID|GAUSS|LOBATTO|SIMPSONS|
    USER|ANALYTIC|CODE_SELECTED(CODE_SELECTED)
  NUMBER OF INTEGRATION POINTS = <integer>num_int_points(5)
  FORMULATION = KH_SHELL|BT_SHELL|BL_SHELL|NQUAD|BT_SC_SHELL
    THICKSHELL|SEVEN_PARAMETER|FIBER_SHELL|C0_TRI_SHELL|
    ORIG_TRI_SHELL (BT_SHELL``QUAD'' or C0_TRI_SHELL``Tri'')
  RIGID BODY PROJECTOR =
    NONE|DRILLING_ONLY|ROTATIONS_AND_DRILLING (ROTATIONS_AND_DRILLING)
  BEGIN USER INTEGRATION RULE
    <real>location_1 <real>weight_1
    <real>location_2 <real>weight_2
    .
    .
    <real>location_n <real>weight_n
  END [USER INTEGRATION RULE]
  LOFTING FACTOR = <real>lofting_factor(0.5)
  LOFTING FACTOR MESH VARIABLE = <string>var_name
  OFFSET MESH VARIABLE = <string>var_name
  ORIENTATION = <string>orientation_name
  DRILLING STIFFNESS FACTOR = <real>stiffness_factor(1.0e-5)
  RIGID BODY = <string>rigid_body_name
  RIGID BODIES FROM ATTRIBUTES = <integer>first_id
    TO <integer>last_id
Explicit
  STABLE TIME STEP ESTIMATE = COURANT|GERSCHGORIN(GERSCHGORIN)

  CONTACT THICKNESS = <real>contactThickness
  CONTACT LOFTING FACTOR = <real>contactLofting

  FIBER ORIENTATION = <real>angle(0)
```

```
  R VECTOR READ VARIABLE = <string>rVecName
  T VECTOR READ VARIABLE = <string>tVecName
END [SHELL SECTION <string>shell_section_name]
```

The SHELL SECTION command block specifies the properties for a shell element. If this command block is referenced in an element block of three-dimensional, four-node elements, the elements in the block will be treated as shell elements. The parameter, shell_section_name, is user-defined and is referenced by a SECTION command line in a PARAMETERS FOR BLOCK command block.

Either a THICKNESS command line or a THICKNESS MESH VARIABLE command line must appear in the SHELL SECTION command block.

If a shell element block references a SHELL SECTION command block with the command line:

```
THICKNESS = <real>shell_thickness
```

then the thickness of all shell elements in the block will be initialized to the value shell_thickness.

Sierra/SM can also initialize the thickness using an attribute defined on elements in the mesh file. Meshing programs such as PATRAN and Cubit typically set the element thickness as an attribute on the elements. If the elements have one and only one attribute defined on the mesh, the THICKNESS MESH VARIABLE command line should be specified as

```
THICKNESS MESH VARIABLE = THICKNESS
```

which causes the thickness of the element to be initialized to the value of the attribute for that element. If there are zero attributes or more than one attribute, the thickness variable will not be automatically defined, and the command will fail.

The thickness may also be initialized by any other field present on the input mesh. To specify a field other than the single-element attribute, use the following form of the THICKNESS MESH VARIABLE command line:

```
THICKNESS MESH VARIABLE = <string>var_name
```

Here, the string var_name is the name of the initializing field.

The input mesh may have values defined at more than one point in time. To choose the point in time in the mesh file that the variable should be read, use the command line

```
THICKNESS TIME STEP = <real>time_value
```

The default time point in the mesh file at which the variable is read is 0.0.

Once the thickness of a shell element is initialized by using either the THICKNESS command line or the THICKNESS MESH VARIABLE command line, this initial thickness value can then be scaled using the scale-factor command line:

```
THICKNESS SCALE FACTOR = <real>thick_scale_factor
```

If the initial thickness of the shell is 0.15 inch, and the value for `thick_scale_factor` is 0.5, then the scaled thickness of the membrane will be 0.075.

The thickness mesh variable specification may be coupled with the `THICKNESS SCALE FACTOR` command line. In this case, the thickness mesh variable is scaled by the specified factor.

The shell formulation can be selected via the `FORMULATION` command line. Several options are available:

- The `BT_SHELL` option is used to select the Belytschko-Tsay (BT) shell formulation. This is the default. The BT shell formulation has constant transverse shear, and is the fastest shell element available. Note, this element can experience stability issues at large in-plane strains.
- The `BL_SHELL` option is used to select the Belytschko-Leviathan (BL) shell formulation. The BL shell element is similar to the Belytschko-Tsay element but has additional shear terms and hourglass controls that the BT does not have. It also includes a projection of the angular velocities and the internal forces. The BL can better solve problems with high gradients of transverse shear (such as twisted beams), and no hourglass control or drilling stiffness is needed. Note, this element can experience stability issues at large in-plane strains.
- The `KH_SHELL` option specifies the Key-Hoff shell formulation, which includes a nonlinear transverse shear term. The Key-Hoff shell can better solve problems with high gradients of transverse shear (such as twisted beams), but distorted (non-rectangular) elements can be excessively stiff. Note, this element can experience stability issues at large in-plane strains.
- The `BT_SC_SHELL` option selects the Belytschko-Tsay formulation and adds an additional strongly co-rotational formulation for in plane stretching and shear. This option can remedy the problems seen with large in-plane strains in the standard `BL`, `KH`, and `BT` shell formulations. However, this option adds significant cost. Additionally, this option has not been thoroughly verified; it should be considered "in-development" and used with caution.
- The `NQUAD` option selects a formulation for a completely elastic plate.
- The `THICKSHELL` option selects an experimental shell element that treats hexahedral elements with a shell-based formulation, the option is in development and generally should not be used in production runs.
- The `SEVEN_PARAMETER` option selects an experimental shell formulation, the option is in development and generally should not be used in production runs.
- The `FIBER_SHELL` option is an experimental fiber reinforced laminate shell. The option is in development and generally should not be used in production runs. The option only functions with a handful of specialized materials.
- The `C0_TRI_SHELL` option specifies the triangular shell formulation developed by Kennedy, Belytschko, and Lin (1986). The C^0 tri shell is a robust, versatile element that uses an incremental strain computation, making it applicable to large strains. Note that this element has no resistance to rotations about the axis perpendicular to its plane.
- The `TRI_SHELL` option is a specialized triangular shell formulation. The option is in development and generally should not be used in production runs.

For the Belytschko-Leviathan shell formulation (`BL_SHELL`), the user may specify which rigid body modes to project out of the local velocity and force vectors via the

RIGID BODY PROJECTOR command. The options are NONE, DRILLING_ONLY, and ROTATIONS_AND_DRILLING (the default). The NONE option indicates no rigid body modes are projected out and is only recommended for explicit analyses with relatively flat geometry. The DRILLING_ONLY option removes modes associated with the drilling degrees of freedom which are unconstrained in the 5-DoF family of shell elements. The ROTATIONS_AND_DRILLING option is a full projection that removes rigid body rotations and the drilling modes and is appropriate if warped shells are present. The last option also preserves angular momentum but leads to a stiffer response for the BL shell [11, 12].



Warning: The RIGID BODY PROJECTOR command is currently only available for the Belytschko-Leviathan shell formulation. The use of this command with other shell formulations has no effect.

For shell elements, the user may select from a number of integration rules, including a user-defined integration option. The integration rule is selected with the command line

```
INTEGRATION RULE = <string>TRAPEZOID|GAUSS|LOBATTO|SIMPSONS|
  ANALYTIC|CODE_SELECTED|USER(CODE_SELECTED)
```

Consult the element documentation [5] for a description of different integration schemes for shell elements.

For most material models the code will by default select the five point TRAPEZOID through the thickness integration rule. The code selected integration scheme for the elastic material model is ANALYTIC, which is an analytic through-thickness integration scheme. This scheme is significantly faster and slightly more accurate than any of the point-wise rules but only valid for fully elastic shell formulations.

Currently the analytic integration scheme is only available for the elastic material model when using the BT_SHELL and BL_SHELL shell formulations for quad shell elements.

Shells using the analytic integration rule do not compute integration point material or stress quantities such as stress, strain, or the various quantities derived from stress and strain. If integration point quantities are desired for elastic material shells, the shells can be forced to use a point-wise integration scheme by manually setting the integration rule to TRAPEZOID or another point-wise rule.

For all integration schemes, the through-thickness integration points are indexed starting at the point closest to the bottom surface with respect to the positive normal direction (See Figure 6-3). For example, when requesting unrotated_stress_xx for a shell element using 5 integration points through the thickness, unrotated_stress_xx_1 corresponds to the integration point closest to the bottom surface and unrotated_stress_xx_5 corresponds to the integration point closest to the top surface.

The number of integration points for the piecewise integration point rules can be set to any number greater than one by using the following command line:

```
NUMBER OF INTEGRATION POINTS = <integer>num_int_points(5)
```

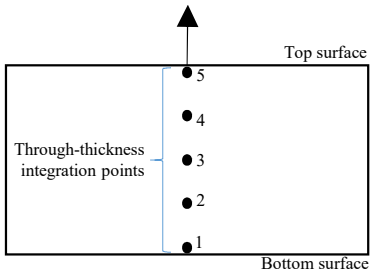


Figure 6-3. Indexing convention for through-thickness integration points

The SIMPSONS, GAUSS, and LOBATTO integration schemes in the INTEGRATION RULE command line all default to five integration points. The number of integration points for these three schemes can be reset by using the NUMBER OF INTEGRATION POINTS command line. There are limitations on the number of integration points for some of these integration rules. The SIMPSONS rule can be set to any number greater than one, the GAUSS scheme can be set to one through seven integration points, and the LOBATTO integration scheme can be set to two through seven integration points.

In addition to these standard integration schemes, the USER INTEGRATION RULE command block may be used to define a custom integration scheme.

```
BEGIN USER INTEGRATION RULE
  <real>location_1 <real>weight_1
  <real>location_2 <real>weight_2
  .
  .
  <real>location_n <real>weight_n
END [USER INTEGRATION RULE]
```

A standard integration scheme and a user scheme may *not* be specified simultaneously. If the USER option is specified in the INTEGRATION RULE command line, a set of integration locations with associated weight factors must be specified. This is done with tabular input command lines inside the USER INTEGRATION RULE command block. The number of command lines inside this command block should match the number of integration points specified in the NUMBER OF INTEGRATION POINTS command line. For example, to use a user-defined scheme with three integration points, the NUMBER OF INTEGRATION POINTS command line should specify three (3) integration points and the number of command lines inside the USER INTEGRATION RULE command block should be three (to give three locations and three weight factors).

For the user-defined rule, the integration point locations should fall between −1.0 and 1.0, and the weights should sum to 2.0.

The command line

```
LOFTING FACTOR = <real>lofting_factor(0.5)
```

allows the user to shift the location of the mid-surface of a shell element relative to the geometric location of the shell element. By default, the geometric location of a shell element in a mesh represents the mid-surface of the shell. If a shell has a thickness of 0.2 inch, the top surface of the shell is 0.1 inch above the geometric surface defined by the shell element, and the bottom surface of the shell is 0.1 inch below the geometric surface defined by the shell element. (The top surface of the shell is the surface with a positive element normal; the bottom surface of the shell is the surface with a negative element normal.)

Figure 6-4 shows an edge-on view of shell elements with a thickness of *t* and the location of the geometric plane in relation to the shell surfaces for three different values of the lofting factor—0.0, 0.5, and 1.0. For a lofting factor of 0.0, the geometric surface defined by the shell corresponds to the top surface of the shell element. A lofting factor of 1.0 puts the geometric surface at the bottom surface of the shell element. The geometric surface is midway between the top and bottom surfaces for a lofting factor of 0.5, which is the default. Lofting factors greater than 1.0 or less than 0.0 are typically reserved for layering of shells—otherwise, this is not recommended.

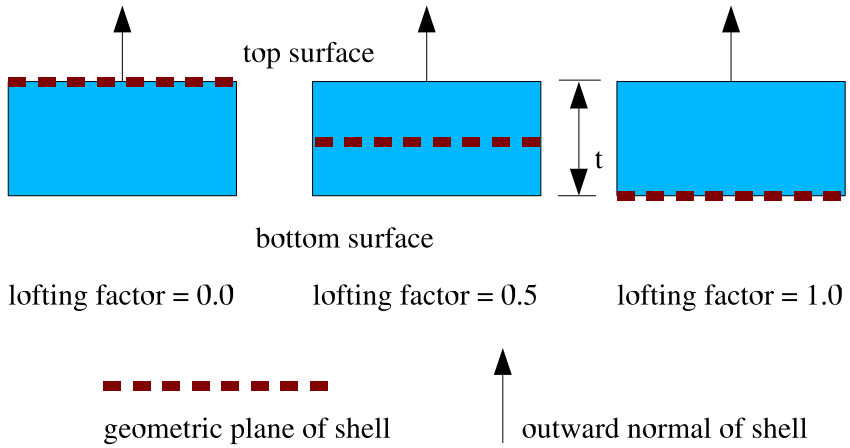


Figure 6-4. Location of geometric plane of shell for various lofting factors.

Consider the example of a lofting factor set to 1.0 for a shell with thickness of 0.2 inch. In this case, the top surface of the shell will be located at a distance of 0.2 inch from the geometric surface (measuring in the direction of the positive shell normal), and the bottom surface will be located at the geometric surface.

Both the shell mechanics and contact use shell lofting. See Section 8.4 for a discussion of lofting surfaces for shells and contact. It is recommended that shell lofting values other than 0.5 not be used if the shell is thick. If the shell is thicker than its in-plane width, the shell lofting algorithms may become unstable.

Sierra/SM can also initialize the lofting factor using an attribute defined on elements in the mesh file. The lofting factor can be set using any field present on the input mesh. To specify a field other than the single-element attribute, use the following form of the `LOFTING FACTOR MESH VARIABLE` command line:

```
LOFTING FACTOR MESH VARIABLE = <string>var_name
```

Here, the string `var_name` is the name of the initializing field.

A third, and slightly different way lofting can be implemented is through the command

```
OFFSET MESH VARIABLE = <string>var_name
```

which allows an offset value to be read from an attribute (or variable) on the mesh file. This attribute (or variable) must be named but may have any name, and this name is specified in place of `var_name` in the command. An offset is a dimensional (i.e., not scaled by the shell thickness) value that gives the shell mid-plane shift in the positive shell normal direction. Internal to Sierra/SM the offset value is converted to an equivalent lofting factor by dividing by the initial thickness and adding 0.5. Thus, an offset of zero gives a lofting factor of 0.5, an offset of one half of the thickness gives a lofting factor of one, and an offset of negative one half of the thickness gives a lofting factor of zero. There is no check that given offset values produce lofting values between zero and one, which allows any offset to be specified but requires that care be exercised to avoid unstable lofted shells.



Warning: An offset and a lofting factor may not both be specified in the same shell section. Both determine the shell lofting and may conflict.

The `ORIENTATION` command line lets the user select a coordinate system for output of in-plane stresses and strains. The `ORIENTATION` option makes use of an embedded coordinate system r, s, t associated with each shell element. The r, s, t coordinate system for a shell element is shown in Figure 6-5. The r-axis extends from the center of the shell to the midpoint of the side of the shell defined by nodes 1 and 2. The t-axis is located at the center of the shell and is normal to the surface of the shell at the center point. The s-axis is the cross-product of the t-axis and the r-axis. The r, s, t-axes form a local coordinate system at the center of the shell; this local coordinate system moves with the shell element as the element deforms.

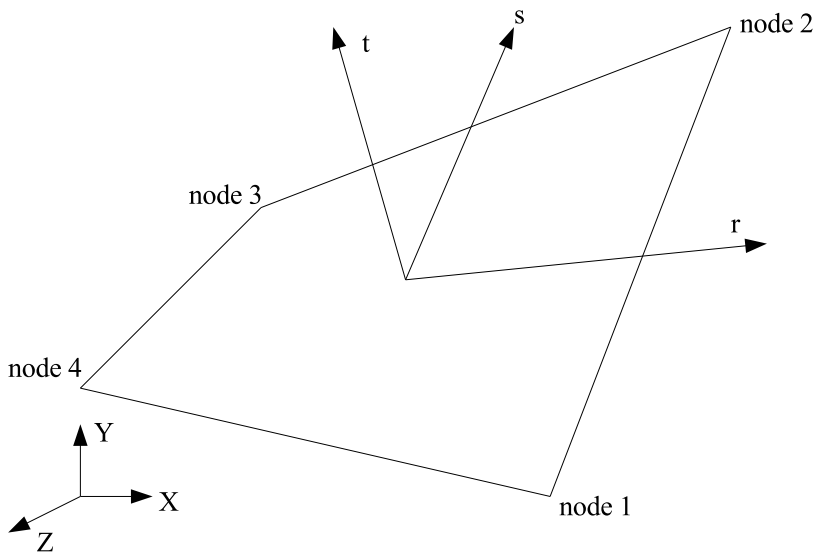


Figure 6-5. Local *rst* coordinate system for a shell element.

The ORIENTATION command line in the SHELL SECTION command block references an ORIENTATION command block that appears in the SIERRA scope. As described in Chapter 2 of this document, the ORIENTATION command block can be used to define a local co-rotational coordinate system $X''Y''Z''$ at the center of a shell element. In the original shell configuration (time 0), one of the axes—X'', Y'', or Z''—is projected onto the plane of the shell element. The angle between this projected axis of the $X''Y''Z''$ coordinate system and the r-axis is used to establish the transformation for in-plane stresses and strains. We will illustrate this with an example.

Suppose that in an ORIENTATION command block a rotation of 30 degrees about the 1-axis (X'-axis) has been specified. The command line for this rotation in the ORIENTATION command block would be

```
ROTATION ABOUT 1 = 30
```

For this case, the Y''-axis is projected onto the plane of the shell (Figure 6-6). The angle between this projection and the r-axis establishes a transformation for the in-plane stresses of the shell (the stresses in the center of the shell lying in the plane of the shell). What will be output as the in-plane stress σ_{xx}^{ip} will be in the Y''-direction; what will be output as the in-plane stress σ_{yy}^{ip} will be in the Z''-direction. The in-plane stress σ_{xy}^{ip} is a shear stress in the $Y''Z''$-plane. The $X''Y''Z''$ coordinate system maintains the same relative position in regard to the rst coordinate system. This means that the $X''Y''Z''$ coordinate system is a local coordinate system that moves with the shell element as the element deforms.

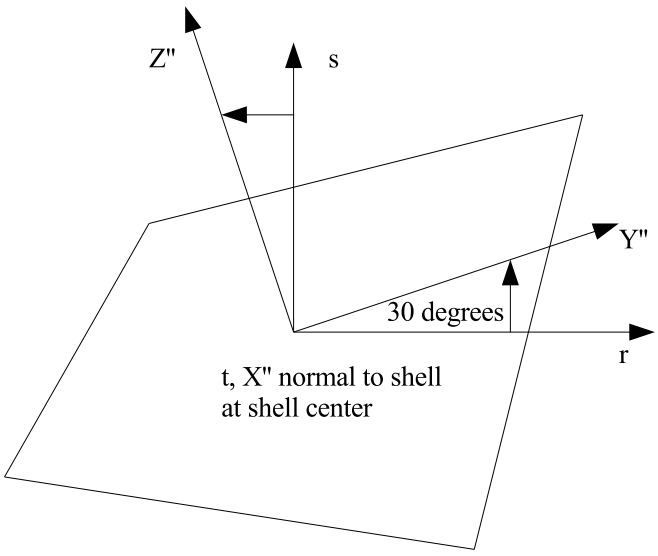


Figure 6-6. Rotation of 30 degrees about the 1-axis (X'-axis).

The following permutations for output of the in-plane stresses occur depending on the axis (1, 2, or 3) specified in the ROTATION ABOUT command line:

- Rotation about the 1-axis (X'-axis): The in-plane stress σ_{xx}^{ip} will be in the Y''-direction; the in-plane stress σ_{yy}^{ip} will be in the Z''-direction. The in-plane stress σ_{xy}^{ip} is a shear stress in

the $Y''Z''$-plane.

- Rotation about the 2-axis (Y'-axis): The in-plane stress σ_{xx}^{ip} will be in the Z''-direction; the in-plane stress σ_{yy}^{ip} will be in the X''-direction. The in-plane stress σ_{xy}^{ip} is a shear stress in the $Z''X''$-plane.
- Rotation about the 3-axis (Z'-axis): The in-plane stress σ_{xx}^{ip} will be in the X''-direction; the in-plane stress σ_{yy}^{ip} will be in the Y''-direction. The in-plane stress σ_{xy}^{ip} is a shear stress in the $X''Y''$-plane.

If no orientation is specified, the in-plane stresses and strains are output in the consistent axes from a default orientation, which is a rectangular system with the X' and Y' axes taken as the global X and Y axes, respectively, and `ROTATION ABOUT 1 = 0.0`.

The command line

```
DRILLING STIFFNESS FACTOR = <real>stiffness_factor
```

adds stiffness in the drilling degrees of freedom to quadrilateral shells. Drilling degrees of freedom are rotational degrees of freedom in the direction orthogonal to the plane of the shell at each node. The formulation used for the quadrilateral shells has no rotational stiffness in this direction. This can lead to spurious zero-energy modes of deformation similar in nature to hourglass deformation. This makes obtaining a solution difficult in quasistatic problems and can result in singularities when using the full tangent preconditioner.

The `stiffness_factor` should be chosen as a quantity small enough to add enough stiffness to allow the solve to be successful without unduly affecting the solution. The default value for `stiffness_factor` is 1.0e-5, which places drilling stiffness about 10000X lower than the bending stiffness. If singularities are encountered in the solution or hourglass-like deformation is observed in the drilling degrees of freedom, setting a larger value may be appropriate.

Elements in a section can be made rigid by including the `RIGID BODY` command line. The `RIGID BODY` command line specifies an indenter that maps to a rigid body command block. See Section 6.3.1 for a full discussion of how to create rigid bodies and Section 6.3.1.1 for information on the use of the `RIGID BODIES FROM ATTRIBUTES` command.

The command line `STABLE TIME STEP ESTIMATE` specifies the method used to calculate the size of the critical time step. The default method is `GERSCHGORIN` circle theorem, which is used to find a bound to the maximum membrane, shear and bending frequencies of the problem. The `COURANT` method uses the characteristic length of the element and the membrane wave speed to approximate the stable time step. This estimate tends to be less conservative than `GERSCHGORIN`, but might not be appropriate for problems with dominant shear or bending responses. This command line option is only available for quadrilateral shells. Triangular shells will only use the Courant estimate.



The command lines `CONTACT SHELL THICKNESS` and `CONTACT SHELL LOFTING` can be used to set the thickness and lofting factor for use in contact. This can be a different thickness or lofting factor than used for defining the element stiffness.

6.2.5.1. Orthotropic Properties

Orthotropic shell materials define their properties within a characteristic coordinate system. This element-local R, S, T coordinate system may be defined using the `FIBER ORIENTATION`, `R VECTOR READ VARIABLE`, and `T VECTOR READ VARIABLE` commands. Command lines `R VECTOR READ VARIABLE` and `T VECTOR READ VARIABLE` specify the element fields which contain vectors R_{input} and T_{input}, respectively, used to define this coordinate system, while `FIBER ORIENTATION` specifies an in-plane rotation of this element-wise coordinate system.

Construction of the local coordinate system, demonstrated graphically in Figure 6-7, is performed in the undeformed configuration; the coordinate system moves with its associated shell element throughout the analysis. The orientation of T_{input} determines whether the positive or negative shell normal is selected as the out-of-plane T-direction. Intermediate direction R' is computed by projecting R_{input} into the plane of the shell, such that it is orthogonal to T. The third orthogonal direction is computed as $S' = T \times R'$. Rotating intermediate directions R' and S' about the T-axis by the specified `FIBER ORIENTATION` angle Θ yields the final local R, S, T coordinate system. The R, S, and T directions can be visualized by requesting element fields 'r_vector,' 's_vector,' and 't_vector,' respectively, in results output. Additional guidance on the output and visualization of element quantities is available in Section 9.3.

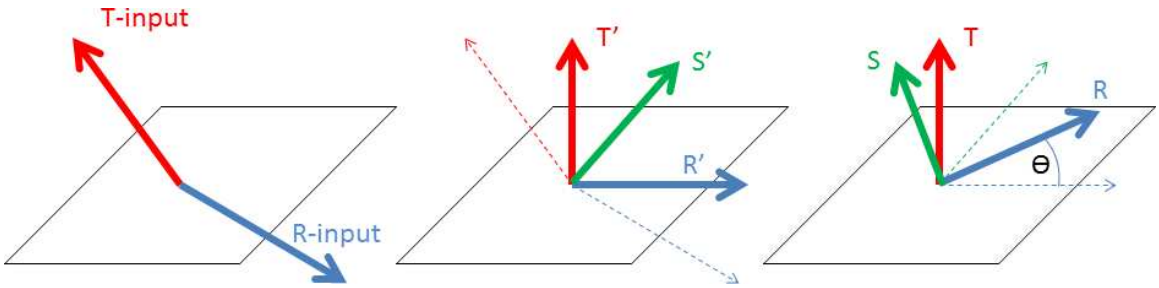


Figure 6-7. Definition of shell R, S, T coordinate system

6.2.6. *Layered Shell Section*

```
BEGIN LAYERED SHELL SECTION <string>sect
  LAYER <int>layerId = <string>subSectName
                       <string>material <string>model
  LOFTING FACTOR = <real>loft(0.5)
  CONTACT THICKNESS = <real>contactThickness
  CONTACT LOFTING FACTOR = <real>contactLofting
END
```

The `LAYERED SHELL SECTION` command block is used to define a single shell element that is a composite of layers with different properties. The layers of the shell can have different materials and different shell formulations.

Internally to the code a layered shell is represented by several individual elements that share the same nodes and same topology. Layered shells will create new element blocks and new elements

in the output file to represent each distinct shell layer. Boundary conditions such as contact or element death can be performed on either the original shell element or on the individual layers specifying the appropriate block name.

The `LAYER` defines a layer for the current shell. The layers of the shell will be stacked from the bottom to the top based on the order of the `LAYER` commands on the input deck. A shell may have up to 99 different layers defined. The `layerId` input is a positive integer provided by the user. This number will be used to label the shell layers to aid in visualization. The new element block sub-layers will be named based on the provided `layerId` using the convention

```
<block_name>_SIERRA_CREATED_LAYER<layerID>
```

The `subSectName` is the name of the shell section associated with the layer. The shell section will be used to define the thickness of the layer and any shell formulation options to use in the layer. The `material` and `model` names define the material and material model that the layer is composed of. Any material that is valid for shell elements can be used as a shell layer material. A default `material` and `model` may be provided in the `parameters for block` command block. Any layer in the section that does not define a material and model will use this default material and model.

Note that the first order approximations used in the shell element formulation does not accurately compute transverse shear strains for layered shells. This is handled and corrected internally by applying a shear correction factor [13].

The lofting factor for the entire shell stack can be specified with the `LOFTING FACTOR` option. Similar to standard shells this lofting factor can vary from 0.0 to 1.0 and has a default value of 0.5 (shell element passes through mid-plane of the layered shell stack.) The lofting of the individual layers will be computed automatically based on the layer thicknesses and layered shell lofting factor.

The command lines `CONTACT SHELL THICKNESS` and `CONTACT SHELL LOFTING` can be used to set the thickness and lofting factor for use in contact. These commands define the contact behavior of the layered shell stack as a whole.

Example of layered shell syntax:

```
begin shell section SHELL_1
  thickness = 0.1
end shell section SHELL_1

begin shell section SHELL_2
  thickness = 0.2
end shell section SHELL_2

begin shell section SHELL_3
  thickness = 0.1
end shell section SHELL_3
```

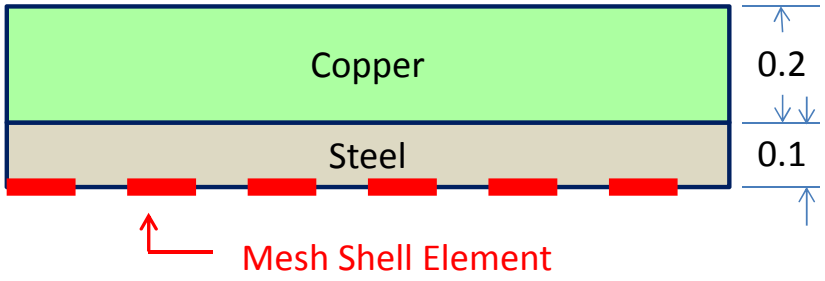


Figure 6-8. Layered Shell Example

```
begin layered shell section LAYERED_SHELL
  layer 1 = SHELL_1 steel  elastic
  layer 2 = SHELL_2 copper elastic_plastic
  lofting factor = 1.0
end

begin parameters for block block_1
  section = LAYERED_SHELL
end
```

In the above example a two layer shell is defined. The composite shell has thickness of 0.3. The bottom layer of the shell is made of elastic steel with a thickness of 0.1. On top of the steel is a 0.2 thick layer of elastic plastic copper. The entire layer stack-up has a lofting factor of 1.0 with respect to the shell element. The configuration is shown in Figure 6-8. The layer specific resultants on the different shell layers can be seen by looking in the results output file element blocks named block_1_SIERRA_CREATED_LAYER_1 and block_1_SIERRA_CREATED_LAYER_2.

6.2.7. Membrane Section

```
BEGIN MEMBRANE SECTION <string>membrane_section_name
  THICKNESS = <real>mem_thickness
  THICKNESS MESH VARIABLE =
    <string>THICKNESS|<string>var_name
  THICKNESS TIME STEP = <real>time_value
  THICKNESS SCALE FACTOR = <real>thick_scale_factor(1.0)
  FORMULATION = <string>MEAN_QUADRATURE|
    SELECTIVE_DEVIATORIC(MEAN_QUADRATURE)
  DEVIATORIC PARAMETER = <real>deviatoric_param
  LOFTING FACTOR = <real>lofting_factor(0.5)
  LOFTING FACTOR MESH VARIABLE = <string>var_name
  RIGID BODY = <string>rigid_body_name
  RIGID BODIES FROM ATTRIBUTES = <integer>first_id
    TO <integer>last_id
  CONTACT THICKNESS = <real>contactThickness
  CONTACT LOFTING FACTOR = <real>contactLofting
```

```
END [MEMBRANE SECTION <string>membrane_section_name]
```

The MEMBRANE SECTION command block specifies the properties for a membrane element. If a section defined by this command block is referenced in the parameters for a block of four-noded elements, the elements in that block will be treated as membranes. The parameter membrane_section_name is user-defined and is referenced by a SECTION command line in a PARAMETERS FOR BLOCK command block.

Either a THICKNESS command line or a THICKNESS MESH VARIABLE command line must appear in the MEMBRANE SECTION command block.

If a membrane element block references a MEMBRANE SECTION command block with the command line

```
THICKNESS = <real>mem_thickness
```

then all the membrane elements in the block will have their thickness initialized to the value mem_thickness.

Sierra/SM can also initialize the thickness using an attribute defined on elements in the mesh file. Meshing programs such as PATRAN and CUBIT typically set the element thickness as an attribute on the elements. If the elements have one and only one attribute defined on the mesh, the THICKNESS MESH VARIABLE command line should be specified as

```
THICKNESS MESH VARIABLE = THICKNESS
```

which causes the thickness of the element to be initialized to the value of the attribute for that element. If there are zero attributes or more than one attribute, the thickness variable will not be automatically defined, and the command will fail.

The thickness may also be initialized by any other field present on the input mesh. To specify a field other than the single-element attribute, use this form of the THICKNESS MESH VARIABLE command line:

```
THICKNESS MESH VARIABLE = <string>var_name
```

where the string var_name is the name of the initializing field.

The input mesh may have values defined at more than one point in time. To choose the point in time in the mesh file that the variable should be read, use the command line:

```
THICKNESS TIME STEP = <real>time_value
```

The default time point in the mesh file at which the variable is read is 0.0.

Once the thickness of a membrane element is initialized by using either the THICKNESS command line or the THICKNESS MESH VARIABLE command line, this initial thickness value can then be scaled by using the scale factor command line:

```
THICKNESS SCALE FACTOR = <real>thick_scale_factor
```

If the initial thickness of the membrane is 0.15 units, and the value for thick_scale_factor is 0.5, then the scaled thickness of the membrane will be 0.075 units.

The FORMULATION command line specifies whether the element will use a single-point integration rule (mean quadrature) or a selective-deviatoric integration rule:

```
FORMULATION = <string>MEAN_QUADRATURE|SELECTIVE_DEVIATORIC
  (MEAN_QUADRATURE)
```

If the user wishes to use the selective-deviatoric rule, the DEVIATORIC PARAMETER command line must also appear in the MEMBRANE SECTION command block:

```
DEVIATORIC PARAMETER = <real>deviatoric_param
```

The selective-deviatoric elements, when used with a parameter greater than 0.0, provide hourglass control without artificial hourglass parameters. The selective-deviatoric parameter, deviatoric_param, which is valid from 0.0 to 1.0, indicates how much of the deviatoric response should be taken from a uniform-gradient integration and how much should be taken from a full integration of the element. A value of 0.0 will give a pure uniform-gradient response with no hourglass control. Thus, this value is of little practical use. A value of 1.0 will give a fully integrated deviatoric response. Although any value between 0.0 and 1.0 is perfectly valid, lower values are generally preferred.

The command line

```
LOFTING FACTOR = <real>lofting_factor(0.5)
```

allows the user to shift the location of the mid-surface of a membrane element relative to the geometric location of the membrane element. By default, the geometric location of a membrane element in a mesh represents the mid-surface of the membrane. If a membrane has a thickness of 0.2 inch, the top surface of the membrane is 0.1 inch above the geometric surface defined by the membrane element, and the bottom surface of the membrane is 0.1 inch below the geometric surface defined by the membrane element. (The top surface of the membrane is the surface with a positive element normal; the bottom surface of the membrane is the surface with a negative element normal.)

Figure 6-4 illustrates shell lofting, which is also applicable to membranes. For membranes, Figure 6-4 represents an edge-on view of membrane elements with a thickness of t and the location of the geometric plane in relation to the membrane surfaces for three different values of the lofting factor—0.0, 0.5, and 1.0. For a lofting factor of 0.0, the geometric surface defined by the membrane corresponds to the top surface of the membrane element. A lofting factor of 1.0 puts the geometric surface at the bottom surface of the membrane element. The geometric surface is midway between the top and bottom surfaces for a lofting factor of 0.5, which is the default.

Consider the example of a lofting factor set to 1.0 for a membrane with thickness of 0.2 inch. In this case, the top surface of the membrane will be located at a distance of 0.2 inch from the geometric surface (measuring in the direction of the positive shell normal), and the bottom surface will be located at the geometric surface.

Both the membrane mechanics and contact use membrane lofting. See Section 8.4 for a discussion of lofting surfaces for membranes and contact.

Sierra/SM can also initialize the lofting factor using an attribute defined on elements in the mesh file. The lofting factor can be set using any field present on the input mesh. To specify a field

other than the single-element attribute, use the following form of the `LOFTING FACTOR MESH VARIABLE` command line:

```
LOFTING FACTOR MESH VARIABLE = <string>var_name
```

Here, the string `var_name` is the name of the initializing field.

Elements in a section can be made rigid by including the `RIGID BODY` command line. The `RIGID BODY` command line specifies an indenter that maps to a rigid body command block. Consult Section 6.3.1 for a full description of how to create rigid bodies and Section 6.3.1.1 for information on the use of the `RIGID BODIES FROM ATTRIBUTES` command.

The command lines `CONTACT SHELL THICKNESS` and `CONTACT SHELL LOFTING` can be used to set the thickness and lofting factor for use in contact. The thickness used for contact can be different that the thickness used for the element stiffness calculations.

6.2.8. Beam Section

```
      BEGIN BEAM SECTION <string>beam_section_name
        SECTION = ROD|TUBE|BAR|BOX|I|L|T|T1|C1|C2|Z|HAT|I2
#
#  Section dimensions, definition depends on particular section type.
#
        D1 = <real>d1
        D1 VARIABLE = <STRING>d1_var
        D2 = <real>d2
        D2 VARIABLE = <STRING>d2_var
        D3 = <real>d3
        D3 VARIABLE = <STRING>d3_var
        D4 = <real>d4
        D4 VARIABLE = <STRING>d4_var
        D5 = <real>d5
        D5 VARIABLE = <STRING>d5_var
        D6 = <real>d6
        D6 VARIABLE = <STRING>d6_var

        T AXIS = <real>tx <real>ty <real>tz
        T AXIS VARIABLE = <string>t_axis_var
        REFERENCE AXIS = <string>CENTER|RIGHT|
          TOP|LEFT|BOTTOM(CENTER)
        AXIS OFFSET = <real>s_offset <real>t_offset
        AXIS OFFSET GLOBAL = <real>x_offset <real>y_offset <real>z_offset
        AXIS OFFSET VARIABLE = <string>axis_offset_var
        RIGID BODY = <string>rigid_body_name
        RIGID BODIES FROM ATTRIBUTES = <integer>first_id
          TO <integer>last_id
        VISUALIZE INTEGRATION POINTS = OFF|ON(OFF)
        ROTATIONAL MASS SCALING = OFF|ON(ON)
```

```
END [BEAM SECTION <string>beam_section_name]
```

The BEAM SECTION command block specifies the properties for a beam element. If this command block is referenced in an element block of three-dimensional, two-node elements, the elements in the block will be treated as beam elements. The name, beam_section_name, can be used by the SECTION command line in a PARAMETERS FOR BLOCK command block.

One of several different types of beam cross section shapes may be specified with the SECTION command. The geometry of each section is defined by up to six geometric parameters, "d1" through "d6." Some sections require all six parameters some as few as two. If a command of the form D1 = <real>d1 is used the d1 dimension will be constant for all elements in an element block. If a command of the form D1 VARIABLE = <STRING>d1_var the d1 dimension will be read from the mesh file from the element variable named d1_var. When using the DN VARIABLE commands the dimension can be different on every element allowing modeling of tapered beams or other similar structures.

All dimensions used by the beam cross section must be defined in the input deck. If a given beam section geometry requires a certain "d" dimension then that dimension must be specified with exactly one DN = command or DN VARIABLE = command.

The following beam cross sectional geometries are supported:

- ROD: Solid circular or elliptical section. See Figure 6-9.
- TUBE: Thin walled circular or elliptical tube section. See Figure 6-10.
- BAR: Sold rectangular section. See Figure 6-11.
- BOX: Thin walled rectangular section. See Figure 6-12. D3 is constant for all four walls.
- I: Standard symmetric I beam section. See Figure 6-13. Width of both flanges given by D4 width of the web given by D3.
- L: Thin walled L section. See Figure 6-14.
- C1: Thin walled C section with open side oriented towards the *s* axis. See Figure 6-15.
- C2: Thin walled C section with open side oriented towards the *t* axis. See Figure 6-16.
- T: Thin walled T section with web oriented along the *t* axis. See Figure 6-17.
- T1: Thin walled T section with web oriented along the *s* axis. See Figure 6-18.
- Z: Thin walled Z section. See Figure 6-19.
- HAT: Thin walled hat section. See Figure 6-20. All plates of the same thickness D4.
- I2: Asymmetric I beam section. See Figure 6-21. The widths and thickness of the two flanges may be specified independently for this section.

Many sections (all but the ROD and BAR) have numerical integration schemes that assume thin walls. This means there is only a single integration point through the thickness of the beam flanges, plates, webs, walls, etc. If a thin walled beam section is defined with plate thickness that are relatively thick compared to the overall beam width and height the numerical properties of the beam may be inaccurate. Specifically the thin walled formulation will tend to underestimate the bending torsional stiffness and strength of thick walled beams.

Most of the sections require the T AXIS or T AXIS VARIABLE command line. If the beam has a circular or tube cross section, and the width of the beam exactly equals the height, then the T_AXIS need not be specified. If the T_AXIS is not specified for one of these circularly

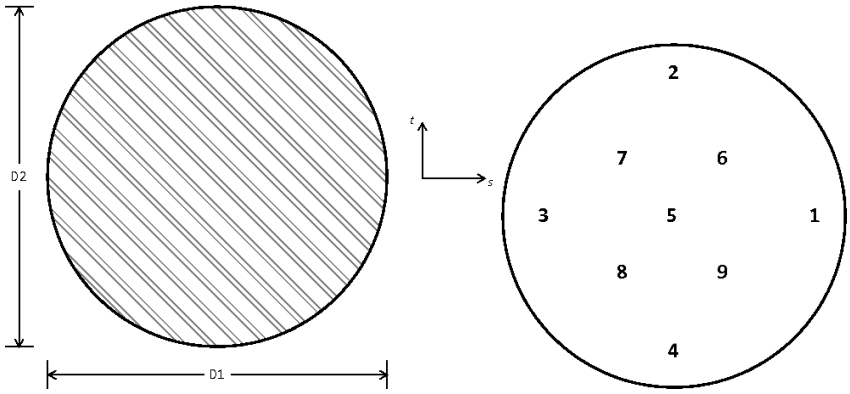


Figure 6-9. Beam ROD Cross Section

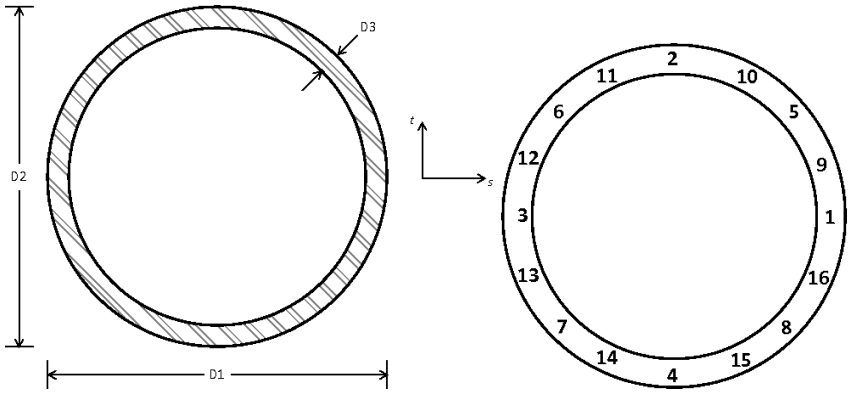


Figure 6-10. Beam TUBE Cross Section

symmetric cross sections the code will arbitrarily pick a t-axis at each beam that is perpendicular to the beam.

Beam section parameters can be specified as constant for all beams in the section with commands such as `D3` or `T AXIS`. Alternatively a set of beam parameters that vary from element to element can be specified with variants of these commands with `VARIABLE` at the end, such as `D3 VARIABLE` or `T AXIS VARIABLE`. When the `VARIABLE` variants of commands are used, the command specifies the name of an attribute field on the input mesh that contains the parameter. For `D1`, `D2`, `D3`, `D4`, `D5`, and `D6` the field should contain one entry per element. For `AXIS OFFSET VARIABLE` the field should contain either two or three entries per element. If the field for the axis offset contains two entries, it specifies the offset in the local *s*, *t* coordinate system. If it contains three entries, it specifies the offset in the global *x*, *y*, *z* coordinate system. For `T AXIS VARIABLE` the field should contain three entries per element, the vector components of the T axis at each element.

For beams it is necessary to specify a local Cartesian *r*, *s*, and *t* coordinate system for each beam. The *r*-axis lies along the length of the beam. The *t*-axis is specified by the user as a vector in the global coordinate system. If the *t*-axis provided by the user is not orthogonal to *r* then the code will alter *t* so that it is orthogonal to *r* as described below. The *s*-axis is computed from the cross product of the *t*-axis and the *r*-axis. The *t*-axis is then recomputed as the cross product of the *r*-axis and the *s*-axis to ensure that the *t*-axis is orthogonal to the *r*-axis. These local direction

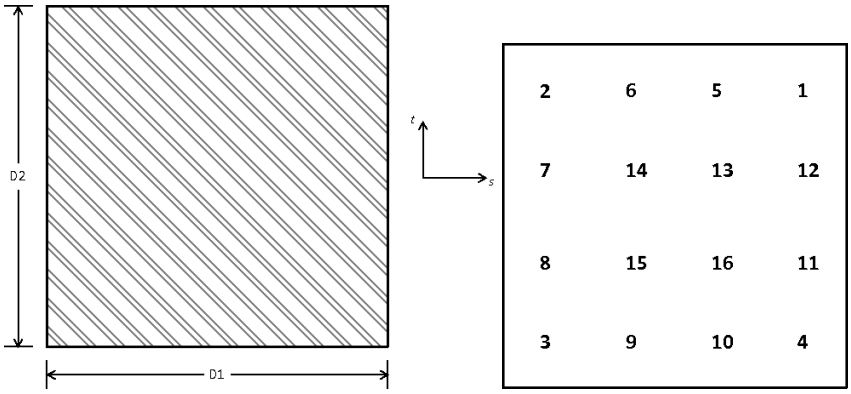


Figure 6-11. Beam BAR Cross Section

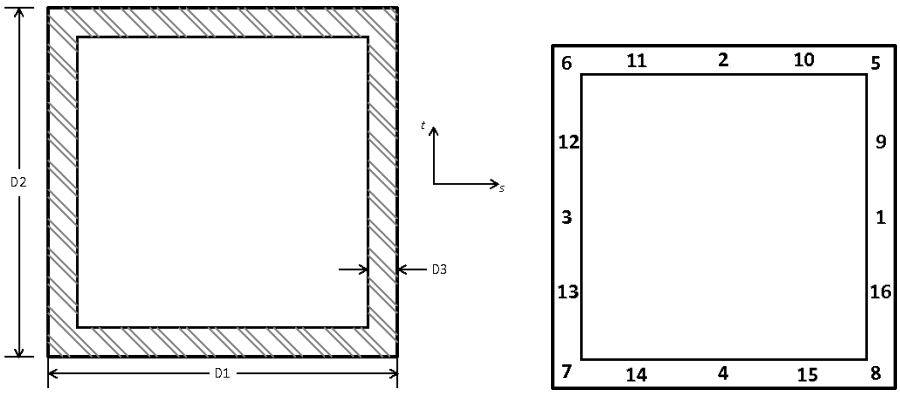


Figure 6-12. Beam BOX Cross Section

vectors are all normalized, so the user-input vectors do not have to be unit vectors.

If the initial position of the *t*-axis is to be parallel to the global Z-axis, then the command line

```
T AXIS = 0 0 1
```

should be used. If the initial position of the *t*-axis is to be parallel to a vector (0.5, 0.8660, 0) in the global coordinate system, then the command line

```
T AXIS = 0.5 0.8660 0.0
```

should be used. The *t*-axis and *s*-axis must always be orthogonal to the beam *r* axis so that they will change position as the beam moves through space. This change in position is tracked internally by the computations for the beam element. Note that the t-axis given to the beam section also affects how the beam is oriented in lofted contact.

6.2.8.1. Reference Axis Offsets

By default, the *r*-axis coincides with the geometric center line of the beam. The geometric center line of the beam is defined by the location of the two nodes defining the beam connectivity. It is possible to offset the local *r*-axis, *s*-axis, and *t*-axis from the geometric center line of the beam. To do this, one can use of one of commands the `REFERENCE AXIS`, `AXIS OFFSET`,

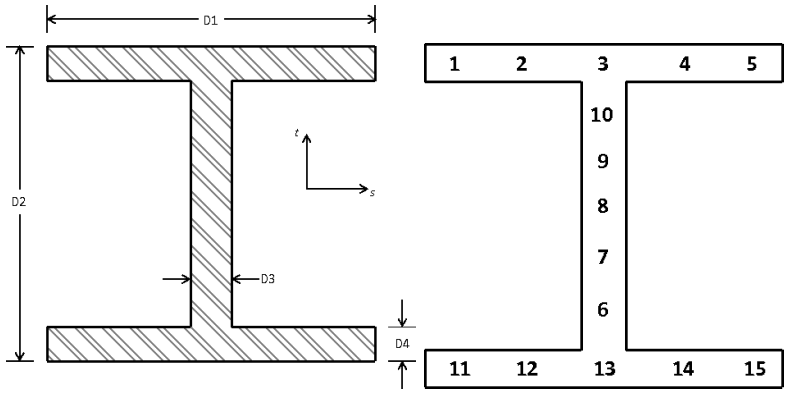


Figure 6-13. Beam I Cross Section

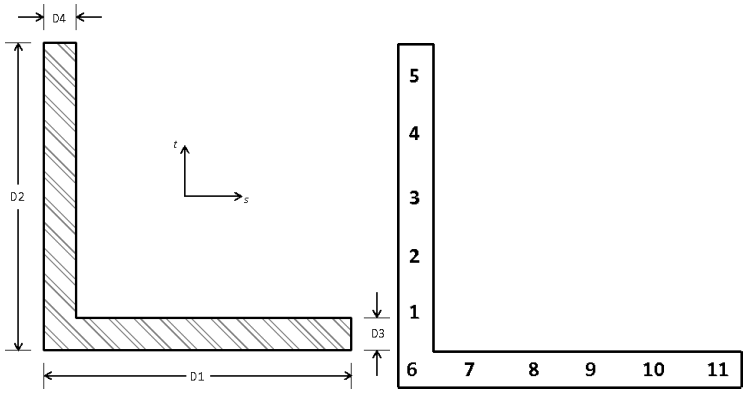


Figure 6-14. Beam L Cross Section

`AXIS OFFSET GLOBAL`, or `AXIS OFFSET VARIABLE`. Only one of these commands may be used at a time.

The `REFERENCE AXIS` command line has the options `CENTER`, `TOP`, `RIGHT`, `BOTTOM`, and `LEFT`. The `CENTER` option is the default, which means that the *r*-axis coincides with the centroid of the beam section. If the `TOP` option is used, the *r*-axis is moved to the top of the middle of the beam. `BOTTOM` corresponds to the bottom middle of the beam. `LEFT` corresponds to the left center of the beam. `Right` corresponds to the right center of the beam. The location of the *r*-axis when using these options is shown for two beam sections in Figure 6-22. For all options, the *s*-axis and the *t*-axis remain parallel to their original positions before the translation of the *r*-axis.

Note for doubly symmetric cross sections the point `CENTER` always lies at the exact center of the cross section between the `LEFT` and `RIGHT` points and the `TOP` and `BOTTOM` points. For non doubly symmetric beam cross sections the beam centroid will depend on the particular beam dimensions and will move in relation to the `LEFT`. `RIGHT`, `TOP`, and `BOTTOM` points.

The `AXIS OFFSET` command line allows the user to offset the local coordinate system from the geometric center line by an arbitrary distance. The `s_offset` value shifts the *r*-axis along the beam local *s*-axis. The `t_offset` value shifts the *r*-axis along the beam local *t*-axis. The *s*, *t*, and *r* axes are only shifted, not rotated by this offset.

Alternatively, the axis offset can be specified in global coordinates using the

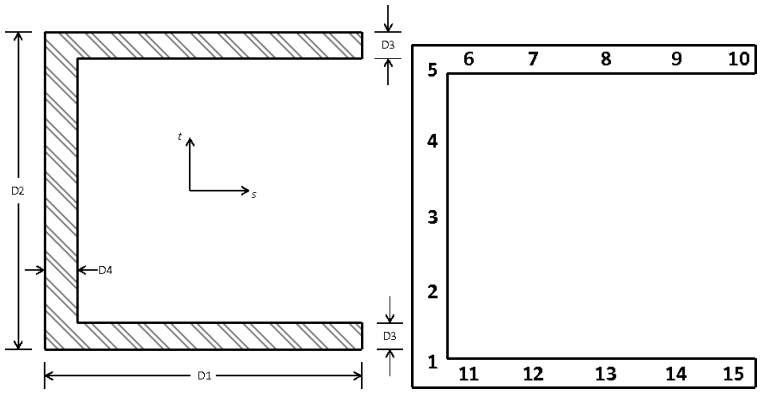


Figure 6-15. Beam C1 Cross Section

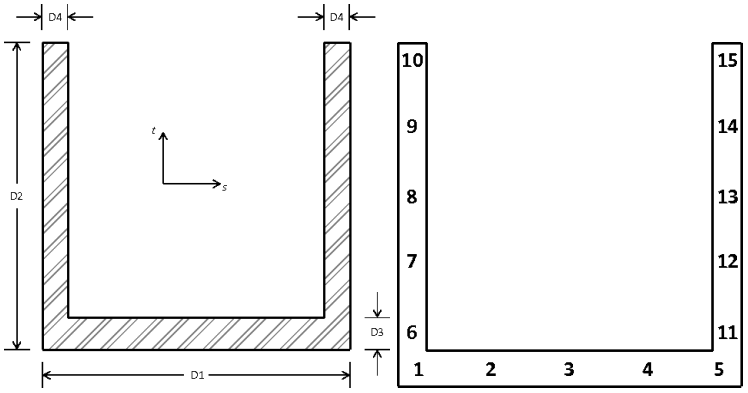


Figure 6-16. Beam C2 Cross Section

`AXIS OFFSET GLOBAL` command. This command takes three parameters, which are the *x*, *y*, and *z* components of the offset in the global coordinate system. Only the portion of the offset that lies in the element local *s t* plane will be used for the offset.

The `AXIS OFFSET VARIABLE` variable command behaves similar to the `AXIS OFFSET` command except that the offsets are specified element by element as defined by the input `axis_offset_var` field. This field should have two entries per element to define the element local *s* and *t* offsets.

6.2.8.2. Section Integration

Strains and stresses are computed at the midpoint of the two noded beam element beam. The integration of the stresses over the cross section at the midpoint is used to compute the internal forces in the beam. Each beam section has its own integration scheme.

The integration scheme for each of the sections is shown in Figure 6-9 through Figure 6-21. The numbers in these figures show the relative location of the integration points in the *s t* coordinate system.

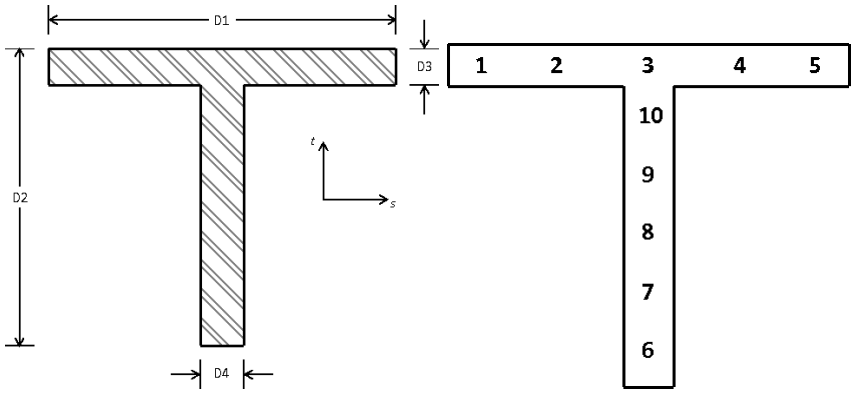


Figure 6-17. Beam T Cross Section

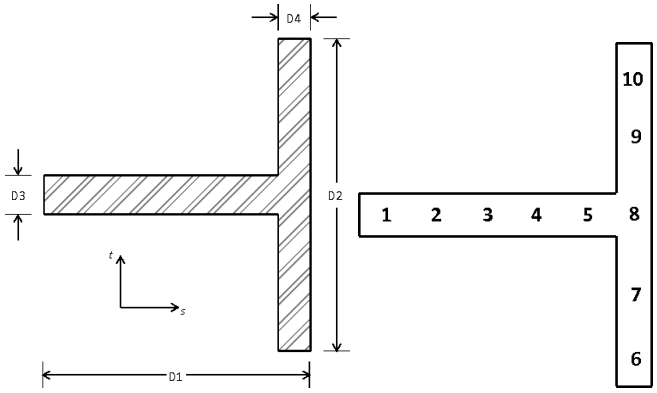


Figure 6-18. Beam T1 Cross Section



Warning: Beam integration point shear stresses only account for torsional shear strains and do not contain effects due to transverse shear strains. Transverse shear forces are calculated from a linear elastic approximation using the given elastic constants and cross section. See the following discussion for more details.

At each integration point the beam element calculates strains and stresses that are related by the chosen material model. The formulation of the beam is like the typical Timoshenko theory and allows for transverse shear strains. However, in the Sierra/SM beam element the transverse shear is treated separately from the other strains. The axial stresses and strains are fully accounted for at the integration points, but the shear strains at the integration points are due only to the torsional deformation. In addition, the shear stress at the integration points is the circumferential torsional shear stress, with respect to the *r* axis, and the radial torsional shear stress is assumed to be zero (accurate for circular cross sections, an approximation for others). The transverse shear response is considered to be linear elastic, and the transverse shear contribution to the internal forces and moments is added apart from the contributions from the integration over the cross section. The linear elastic response is based on the user specified elastic constants.

Users should exercise caution when using the beam element with plasticity in problems that have large deformations or thick beams. Since the transverse shear response is only linear elastic, it will not properly account for other material models. Also, users should be careful to account for

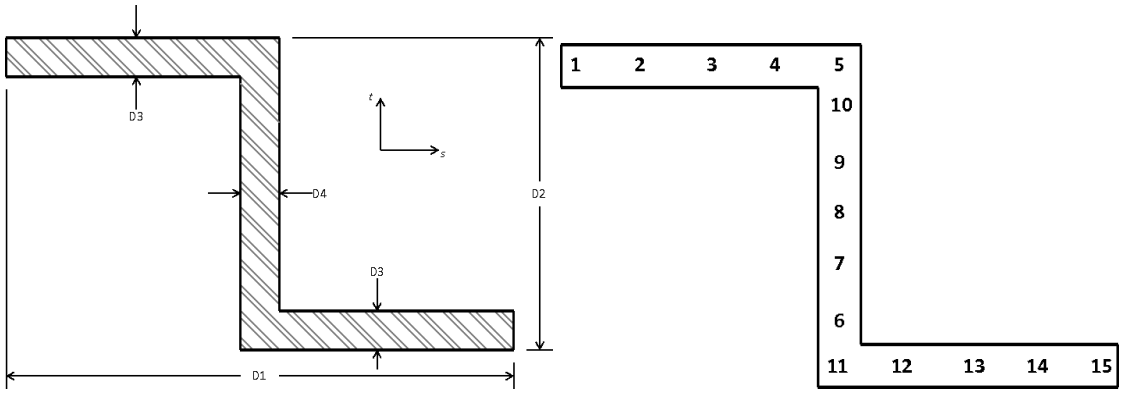


Figure 6-19. Beam Z Cross Section

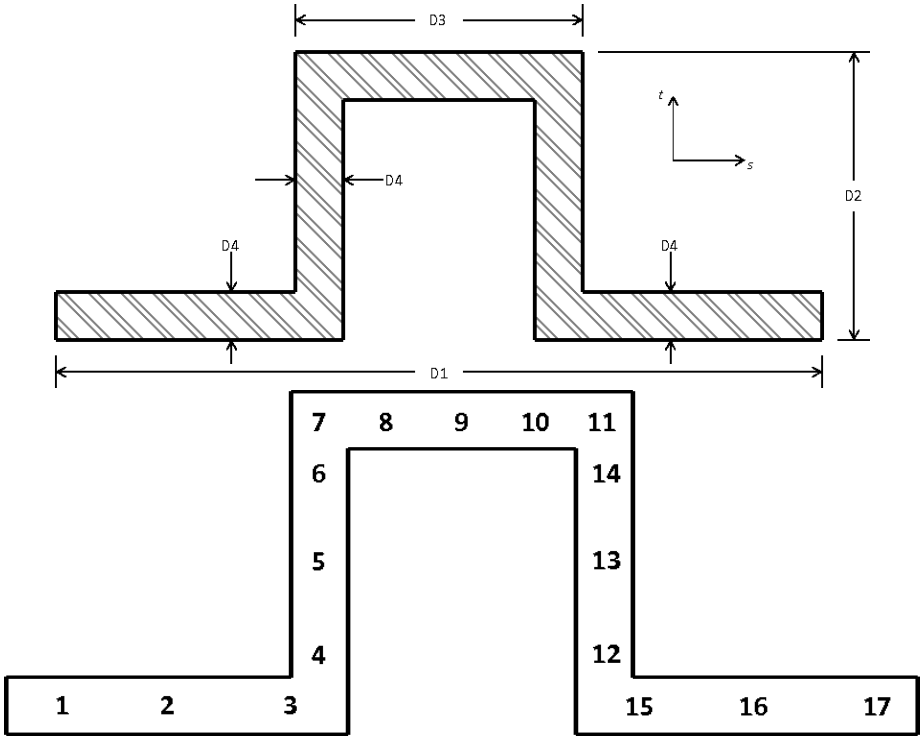


Figure 6-20. Beam HAT Cross Section

the integration point shear stresses only containing circumferential torsional shear stress.

Integration point data can be output for beam elements in much the same manner that it is requested for other elements by using commands in the `RESULTS OUTPUT` command block as described in Chapter 9. At each integration point, there are two components of strain and two components of stress. These components of stress and strain are in the axial and in-plane circumferential directions. The integration-point strain variables are named `beam_strain_axial` and `beam_strain_shear`, and the integration-point stress variables are `beam_stress_axial` and `beam_stress_shear`. In addition, state data is available at each integration point.

An important aspect of integration point output for beam elements is that the data is output in

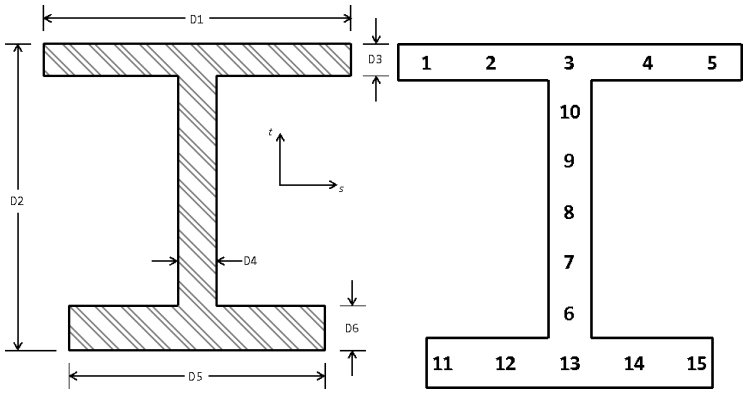


Figure 6-21. Beam I2 Cross Section

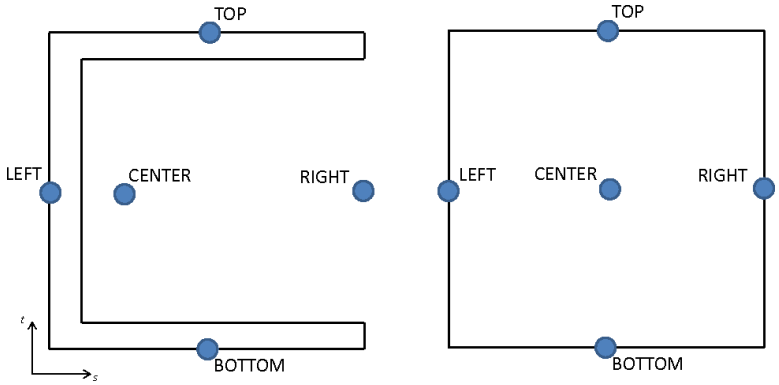


Figure 6-22. Locations of REFERENCE AXIS command options

arrays sized by the number of integration points for the chosen beam cross-section (for example 9 for a rod and 15 for an I).

The beam axial and shear strains and stresses are output in sequence with the integration point numbering. For example, the first value of `beam_strain_axial`, designated in the output as `_1` for a rod or `_01` for the other sections, is the axial strain at the first integration point. The numbering continues to the number of integration points for that beam section—up to `_9` for rod sections, `_15` for I sections, and `_16` for other sections.

An alternative method of accessing the stress is through the symmetric tensor `unrotated_stress`. Because a beam element uses local axes (i.e., a co-rotational system), `unrotated_stress` is given with respect to the *r*, *s*, and *t* coordinate system, and its `_xx` and `_xy` components are equivalent to the axial and torsional circumferential shear stresses, respectively, discussed above. Stress in general is stored as a symmetric tensor that contains six components, but for beams four of these components are never used. The axial stress at the first integration point is designated by `unrotated_stress_xx_01` (or `unrotated_stress_xx_1` for a rod section, with this pattern true for all integration point quantities discussed below) and the circumferential torsional shear stress at the first integration point is `unrotated_stress_xy_01`. The other four components, `unrotated_stress_yy_01`, `unrotated_stress_zz_01`, `unrotated_stress_yz_01` and `unrotated_stress_xz_01`, are unused and are set to zero. The stresses at other integration points are named `unrotated_stress_xx_NN` and

unrotated_stress_xy_NN, where NN is a number from 01 to 16.

Optionally the lofted geometry and a visual representation of beam integration point locations may be included in the output mesh file with use of the VISUALIZE INTEGRATION POINTS = ON command. When the extra visualization is turned on each beam will be represented both by the original line element and also and by a set of brick elements. The brick elements will represent the shape of the beam cross section, there will be one brick element per beam integration point. The visualize integration points option can be helpful in ensuring that the beam is being setup as desired both in size, orientation, and offsets from the neutral axis.

When using the VISUALIZE INTEGRATION POINTS command some beam integration point quantities can be visualized directly on the lofted hexahedral geometry. Any field that is defined at beam integration points will be copied out to the hexahedral representation elements. For example this includes fields such as:

- intg_point: Integration point number
- intg_weight: Integration point weight, should sum to 1.0 over all the integration points.
- intg_shear_flow_dir: Integration point shear flow direction
- beam_stress_axial: Axial component of beam stress
- beam_stress_shear: Shear component of beam stress
- beam_strain_axial: Axial component of beam strain
- beam_strain_shear: Shear component of beam strain
- lame_state_<model_name>: Material state variables.

It is possible to access mean values for the internal forces at the midpoint of the beam. The axial force at the midpoint of the beam is obtained by referencing the variable beam_axial_force. The transverse forces at the midpoint of the beam in the *s*-direction and the *t*-direction are obtained by referencing beam_transverse_force_s and beam_transverse_force_t, respectively. The torsion at the midpoint of the beam (the moment about the *r*-axis), is obtained by referencing beam_moment_r. The moments about the *s*-axis and the *t*-axis are obtained by referencing beam_moment_s and beam_moment_t, respectively.

Elements in a section can be made rigid by including the RIGID BODY command line. The RIGID BODY command line specifies an identifier that maps to a rigid body command block. See Section 6.3.1 for a full discussion of how to create rigid bodies and Section 6.3.1.1 for information on the use of the RIGID BODIES FROM ATTRIBUTES command.



Under certain conditions, the rotational mass of beams will be automatically scaled. This scaling is performed only when the computed time step from the axial modes is larger than the computed time step using the bending modes. This allows for larger critical time steps to be taken during the simulation. However, the user does have the option to turn it OFF via the ROTATIONAL MASS SCALING line command.

6.2.8.3. Beam Element Accuracy

It is worth noting that due to a finite accuracy integration rule the integration error in beam behavior is dependent on the loading case. For pure axial loading, the error is nearly zero

throughout both the elastic and plastic regimes for all cross sections. This is due to axial resistance being a function of sectional area, which is explicitly computed from section properties.

However, in the bending cases there is varying levels in error dependent on section, bending direction, and loading value. In general, the error percentage is low (roughly 1 to 2%) for very elastic loadings in the *s*- and *t*- bending directions when strain is about 10^{-6}. Thin-walled sections with more integration points, such as the hat, are more accurate in bending in both the elastic and plastic regimes when bent in the *s*- and *t*-directions . Error can be larger (up to 10%) for elastic bending in directions of than the *s* and *t* axis. Additionally errors will tend to be larger (up to 10%) in the elastic plastic or fully plastic transition regimes.

Torsional loading for some cross sections can potentially produce large error. The integration assumes no out of plane warping during torsion. Thus the torsional resistance of sections that do have out of plane warping (open sections) may be greatly over estimated. For example, the circular rod will see no sectional warping thus having less deviation from the analytical solution, particularly in the elastic regime. However, the hat cross section would see high amounts of sectional warping which is not fully accounted for in the solution produced by the code. In cases such as this, results will be vastly different than their analytical solutions. Additionally, it is important to point out that there is no bend-twist twist coupling for open sections, as this produces spurious unstable behavior.

6.2.9. Truss Section

```
BEGIN TRUSS SECTION <string>truss_section_name
  AREA = <real>cross_sectional_area
  DIAMETER = <real>diameter
  AREA MESH VARIABLE = <string>area_var_name
  INITIAL LOAD = <real>initial_load
  PERIOD = <real>period
  RIGID BODY = <string>rigid_body_name
  RIGID BODIES FROM ATTRIBUTES = <integer>first_id
    TO <integer>last_id
END [TRUSS SECTION <string>truss_section_name]
```

The `TRUSS SECTION` command block specifies the properties for a truss element. If this command block is referenced in an element block of three-dimensional, two-node elements, the elements in the block will be treated as truss elements. The parameter, `truss_section_name`, is user-defined and is referenced by a `SECTION` command line in a `PARAMETERS FOR BLOCK` command block.

The cross-sectional area for truss elements can specified by the `AREA` command line. Alternatively the area can be set by defining the diameter for the cylindrical truss element. If specified the value `cross_sectional_area` or `diameter` will be used for all truss elements of the block element block. Alternatively the `AREA MESH VARIABLE` may be used to provide the name of a variable on the input genesis mesh that lists the element by element attributes. If the truss area is provided by an element attribute, the `area_var_name` should be set to “attribute.” Note one and only one of `AREA`, `DIAMETER`, or `AREA MESH VARIABLE` must be given for the truss section.

The truss can be given some initial load over some given time period. The magnitude of the load is specified by the `INITIAL LOAD` command line. If the load is compressive, the sign on the value `initial_load` should be negative; if the load is tensile, the sign on the value `initial_value` should be positive. The period is specified by the `PERIOD` command line.

The initial load is applied over some period by specifying the axial strain rate in the truss, $\dot{\varepsilon}$, over some period p. At some given time t, the strain rate is

$$\dot{\varepsilon} = \frac{ap}{2}\left[1 - \cos\left(\pi t/p\right)\right], \tag{6.3}$$

where

$$a = \frac{2F_i}{EAp}. \tag{6.4}$$

In Equation (6.4), F_i is the initial load, E is the modulus of elasticity for the truss, and A is the area of the truss. Over the period p, the total strain increment generates the desired initial load in the truss.

During the initial load period, the time increments should be reasonably small so that the integration of $\dot{\varepsilon}$ over the period is accurate. The period should be set long enough so that if the model was held in a steady state after time p, there would only be a small amount of oscillation in the load in the truss.

Certain boundary conditions should not be activated until after the pre-stressing is done. During the pre-stressing, time-independent boundary conditions such as fixed displacement will most likely be turned on. Time-dependent boundary conditions such as prescribed acceleration or prescribed force will most likely be activated after the pre-stressing is complete.

Elements in a section can be made rigid by including the `RIGID BODY` command line. The `RIGID BODY` command line specifies an indenter that maps to a rigid body command block. See Section 6.3.1 for a full discussion of how to create rigid bodies and Section 6.3.1.1 for information on the use of the `RIGID BODIES FROM ATTRIBUTES` command.



Warning: Truss elements currently do not work correctly as a rigid body due to lack of cross-section geometry to define the moment of inertia about the truss axis. They should be used only if the user explicitly includes the command line `INERTIA =` in the `RIGID BODY` command block to specify the mass moment of inertia for the entire rigid body. (See Section 6.3.1.) The recommendation is to use beam elements for rigid bodies rather than truss elements if possible.

Axial stress, which is the only stress component for a truss element, is output as a symmetric tensor with six components to be compatible with the notion of volumetric stress. Because of this, for every truss element, six values of stress are stored and output, but only the fist value is ever used. The axial stress is therefore output as `stress_xx` and the other five stress components, `stress_yy`, `stress_zz`, `stress_xy`, `stress_yz`, `stress_zx` are all unused and set to zero.

6.2.10. Spring Section

```
BEGIN SPRING SECTION <string>spring_section_name
  FORCE STRAIN FUNCTION = <string>force_strain_function
  DEFAULT STIFFNESS <real>default_stiffness
  PRELOAD = <real>preload_value
  PRELOAD DURATION = <real>preload_duration
  RESET INITIAL LENGTH AFTER PRELOAD = <string>NO|YES
  MASS PER UNIT LENGTH = <real>mass_per_unit_length
END [SPRING SECTION <string>spring_section_name]
```

The `SPRING SECTION` command block specifies the properties for a spring element. If this command block is referenced in an element block of three-dimensional, two node elements, the elements in the block will be treated as spring elements. The parameter, `spring_section_name`, is user-defined and is referenced by a `SECTION` command line in a `PARAMETERS FOR BLOCK` command block.

The spring behavior is governed by the force-engineering strain function which is specified by the `FORCE STRAIN FUNCTION` command line. The force generated by the spring element is based on the evaluation of the user specified `force_strain_function`, which has units of force. The strain is defined by $(L_i - L_0)/L_0$ with L_i the current spring length and L_0 the original spring length. This allows the force-strain function to be length independent.

Positive values of the force strain function correspond to tension and negative values to compression. The force-strain function should be defined over the full range of strains that the spring may see, both positive and negative. The following are examples of possible spring stiffness functions.

```
BEGIN FUNCTION FORCE_STRAIN_2
  # "x" is the signed input strain value
  # This is a linear spring with stiffness of 1000
  TYPE = ANALYTIC
  EVALUATE EXPRESSION = "1000*x"
END

BEGIN FUNCTION FORCE_STRAIN_3
  # Nonlinear "chain" element.  Has stiffness in tension only
  TYPE = PIECEWISE LINEAR
  BEGIN VALUES
     0     0
     5  5000
  END
END

BEGIN FUNCTION FORCE_STRAIN_1
  # Nonlinear complex spring with gaps and variable stiffness
```

```
  TYPE = PIECEWISE LINEAR
  BEGIN VALUES
    -0.5 -1000
    -0.1     0
     0       0
     0.1  1000
     0.2  5000
     0.5 10000
  END
END
```

The DEFAULT STIFFNESS command block specifies the spring stiffness used during the first time step and during preload. In all other situations the spring stiffness is based on the slope of the force-strain function evaluated at the previous time step and the current time step. The unit of default_stiffness is force.

To specify a preload on the spring both the PRELOAD and PRELOAD DURATION command line must be specified. The PRELOAD command line specifies the magnitude of the preload force, while the PRELOAD DURATION command line specifies how long the preload application should take, in seconds. The preload force will be initially set to 0 and ramp up to the PRELOAD value when your simulation reaches 95% of the PRELOAD DURATION. After reaching 95%, the preload will remain constant.

An optional preload input, RESET INITIAL LENGTH AFTER PRELOAD, is used when the user would like the initial length of the spring to reset to the displaced length after the preload has occurred. If this command line is not specified, or is set to NO, the initial length of the spring is the undeformed length as calculated from the input mesh.

Springs can have a mass as defined by the MASS PER UNIT LENGTH command line. When defining the spring section this value must be specified. A zero mass spring is allowed by setting MASS PER UNIT LENGTH explicitly to zero. Note, in explicit computations if the spring has no mass the spring has no defined element time step. For a mass-less spring the effect of the spring on the global time step will be factored into the nodal 3.2.3.1 or Lanczos 3.2.1.5 time step estimates.



6.2.11. Damper Section

```
BEGIN DAMPER SECTION <string>damper_section_name
  AREA = <real>damper_cross_sectional_area
END [DAMPER SECTION <string>damper_section_name]
```

The DAMPER SECTION command block specifies the properties for a damper element. If this command block is referenced in an element block of three-dimensional, two-node elements, the elements in the block will be treated as damper elements. The parameter, damper_section_name, is user-defined and is referenced by a SECTION command line in a PARAMETERS FOR BLOCK command block.

The cross-sectional area for damper elements is specified by the `DAMPER AREA` command line. The value `damper_cross_sectional_area` is the cross-sectional area of the dampers in the element block.

The damper area is used only to generate mass associated with the damper element. The mass is the density for the damper element multiplied by the original volume of the element (original length multiplied by the damper area).

The force generated by the damper element depends on the relative velocity along the current direction vector for the damper element. If **n** is a unit normal pointing in the direction from node 1 to node 2, if $\mathbf{v}_1$ and $\mathbf{v}_2$ are the velocity vectors at nodes 1 and 2, respectively, then the force generated by the damper element is

$$F_d = \eta \mathbf{n} \cdot (\mathbf{v}_2 - \mathbf{v}_1), \tag{6.5}$$

where η is the damping parameter. Currently, the damping parameter must be specified by using an elastic material model for the damper element. The value for Young's modulus in the elastic material model is used for the damping parameter η.



6.2.12. Point Mass Section

```
BEGIN POINT MASS SECTION <string>pointmass_section_name
  VOLUME = <real>volume
  MASS = <real>mass
  IXX = <real>Ixx
  IYY = <real>Iyy
  IZZ = <real>Izz
  IXY = <real>Ixy
  IXZ = <real>Ixz
  IYZ = <real>Iyz
  RIGID BODY = <string>rigid_body_name
  RIGID BODIES FROM ATTRIBUTES = <integer>first_id
    TO <integer>last_id
  MASS VARIABLE = <string>mass_variable_name
  INERTIA VARIABLE = <string>inertia_variable_name
  OFFSET VARIABLE = <string>offset_variable_name
  ATTRIBUTES VARIABLE NAME = <string>attrib_variable_name
END [POINT MASS SECTION <string>pointmass_section_name]
```

A point mass element is a mass at a node, which can be a convenient modeling tool in certain instances. The user can create an element block with one or more point masses. A point mass is a sphere element attached to a single node. A point mass will have its mass added to the mass at the connected node. (Other mass at the node will be derived from mass due to elements attached to the node.) The mass at a node due to a point mass is treated like any other mass at a node derived from an element. The mass due to point mass will be included in body force calculations and kinetic energy calculations, for example.

Point masses are a convenient modeling tool to be used in conjunction with rigid bodies. An element block including one or more point masses can be included in a collection of element

blocks used to define a rigid body. The element block of point masses can be used to adjust the total mass and inertia properties for the rigid body. (The point mass element does not have to be used only in conjunction with rigid bodies. One can place a point mass at a node associated with solid or structural elements.)

An element block in which the connectivity for each element contains only one node can be used as a collection of point masses. This command block would have the following form:

```
BEGIN PARAMETERS FOR BLOCK <string>block_id
  MATERIAL = <string>material_name
  MODEL    = <string> material_model_name
  SECTION = <string>point_mass_section_name
END PARAMETERS FOR BLOCK <string>block_id
```

The element block associated with the point mass must reference a material command block just like any other element block. If the point mass section specifies a volume, then the product of the density specified in the material block and the volume specified in the section block is used to calculate the element mass.

The `VOLUME` command specifies the volume of a point mass element. The mass of the element is then found by multiplying that volume by the material model density.

The `MASS` command explicitly specifies the mass.

The `IXX`, `IYY`, `IZZ`, `IXY`, `IXZ`, `IYZ` commands are used to explicitly define the symmetric inertia tensor entries for the point mass. Note that currently, the trace of the inertial tensor is added to the nodal rotational mass and the rest of the inertia tensor entries are ignored.

The `OFFSET` is used to define the offset of the mass with respect to the connected node. Currently the offset affects only the rotational properties of the point mass. The rotational mass of the point mass is increased by mass times offset distance squared.

The `MASS VARIABLE` specifies that the mass of the point mass elements is to be read from the input mesh file from the variable with the specified name (use the name ”attribute” to read the mass from the mesh block attributes.)

The `INERTIA VARIABLE` specifies that the inertia tensor of the point mass elements is to be read from the input mesh file from the variable with the specified name (use the name ”attribute” to read the inertia tensor from the mesh block attributes.) When this command is used, the mesh file variable should contain six entries per element, ordered as follows: Ixx, Iyy, Izz, Ixy, Ixz, Iyz.

The `OFFSET VARIABLE` specifies that the offset vector of the point mass elements is to be read from the input mesh file from the variable with the specified name (use the name ”attribute” to read the offset vector from the mesh block attributes.) When this command is used, the mesh file variable should contain three entries per element, ordered as follows: Offset_x, Offset_y, Offset_z.

The `ATTRIBUTES VARIABLE NAME` specifies that all properties of the point mass element are to be read from the input mesh file of the variable with the specified name (use the name ”attribute” to read from the mesh block attributes.) When this command is used, the mesh file variable

should contain ten entries per element, ordered as follows: Mass, Ixx, Iyy, Izz, Ixy, Ixz, Iyz, Offset_x, Offset_y, Offset_z.

Multiple ways are provided to specify the raw attributes of mass, inertia and offset for the point mass element. The user must be careful to only use one of these ways to define each property. Using multiple ways to define the mass will result in errors or warnings because of ambiguities.

If the user has access to the SEACAS codes from Sandia National Laboratories, The codes in this library may be used to generate element blocks with point mass elements. See Reference [14] for an overview of the SEACAS codes. By using various SEACAS codes, an element block with one or more point masses may be generated. For each point mass in the element block, an eight-noded hexahedral element centered at the point mass location should be created. The hexahedron may have arbitrary dimensions, but it is best to work with a unit cube. For example, to generate a point mass at coordinates (13.5, 27.0, 3.1415), a $1 \times 1 \times 1$ (unit) hexahedron centered at (13.5, 27.0, 3.1415) should be created. The SEACAS program SPHGEN3D may be used to convert the hexahedron to a zero-dimensional element located at the center of the hexahedron in three-dimensional space. In the provided example, SPHGEN3D would create a zero-dimensional, single-noded element at coordinates (13.5, 27.0, 3.1415).

Elements in a section can be made rigid by including the `RIGID BODY` command line. The `RIGID BODY` command line specifies an indenter that maps to a rigid body command block. See Section 6.3.1 for a full discussion of how to create rigid bodies and Section 6.3.1.1 for information on the use of the `RIGID BODIES FROM ATTRIBUTES` command.



6.2.13. Particle Section

```
BEGIN PARTICLE SECTION <string>sph_section_name
  RADIUS MESH VARIABLE = <string>var_name|<string>attribute|
    SPHERE INITIAL RADIUS = <real>rad
  RADIUS TIME STEP = <string>time
  PROBLEM DIMENSION = <integer>1|2|3(3)
  CONSTANT SPHERE RADIUS
  FINAL RADIUS MULTIPLICATION FACTOR = <real>factor(1.0)
  FORMULATION = <string>MASS_PARTICLE|SPH(SPH)
  MONAGHAN EPSILON = <real>monaghan_epsilon(0.0)
  MONAGHAN N = <real>monaghan_n(0.0)
  SPH ALPHAQ PARAMETER = <real>alpha(1.0)
  SPH BETAQ PARAMETER = <real>beta(2.0)
  DENSITY FORMULATION = <string>MATERIAL|KERNEL(MATERIAL)
  MAXIMUM RADIUS CHANGE = <real>minVal <real>maxVal
END [PARTICLE SECTION <string>sph_section_name]
```

SPH (smoothed particle hydrodynamics) is useful for modeling fluids or for modeling materials that undergo extremely large distortions. One must be careful when using SPH for modeling. SPH tends to exhibit both accuracy and stability problems, particularly in tension. An SPH particle interacts with other nearest-neighbor SPH particles based on radius properties of all the elements involved; SPH particles react with other elements, such as tetrahedra, hexahedra, and shells, through contact. Many more details on the theoretical background of SPH are available in [15].

Contact interactions between the particles in a particle element block and other element types may be defined using the `CONTACT NODE SET` option described in Section 8.4.2. All particles in the particle element block must be included in a node set if the `CONTACT NODE SET` option is used.

The particle section should only be associated with elements with the `SPHERE` topology. Sphere elements are point elements that are connected to only a single node.

All the particles contained in a particle element block must be given some initial radius. There are two options for setting the initial radius for each particle. First, each particle can be given the same radius. To set the radius for each particle in an element block to the same value, use the `SPHERE INITIAL RADIUS` command line. The parameter `rad` on this command line sets the radius for all the particles in the element block.

Alternatively, the radius for each particle can be read from a mesh file. The radii can be read from a variable on the mesh file from attributes associated with the element block. To read some variable from the mesh file for the radii, the command

```
RADIUS MESH VARIABLE = sph_radius
```

may be used, where `sph_radius` is the variable name on the mesh file. To use the variable associated with a specific time on the mesh file, the `RADIUS TIME STEP` command line should be used to select the specific time. To read the attributes associated with the particles, the command line

```
RADIUS MESH VARIABLE = attribute
```

should be inserted (as shown) into the `PARTICLE SECTION` command block. Pronto3D [1] only offers the attribute option. The attribute option should be used to compare Sierra/SM and Pronto3D results.

If the attribute option is used, each particle should have two attributes defined per element. The first attribute is the particle radius, and the second attribute is the particle volume. The most common way to generate a particle mesh is to first generate a solid element mesh and then use the utility 'sphgen3d' to convert the solid elements into sphere elements. Sphgen3d will convert each solid element into a sphere and place it at the source element centroid. In the output mesh file generated by sphgen3d The radius attribute of the element blocks will be proportional to the source element size and the volume attribute will be the volume of the source element.

The `FINAL RADIUS MULTIPLICATION FACTOR` command may be used to increase or decrease the initial particle radii. The factor specified will scale the particle radii input with either a constant or a mesh file attribute.

After the radii are initialized, the user may specify whether the radii are to remain constant or are to change throughout the analysis. The `CONSTANT SPHERE RADIUS` command line is an optional command line that prevents the sphere radius from changing over the course of the calculation. By default, the sphere radii will expand or contract based on the changing density in the elements to satisfy the relation that element mass (a constant) equals element volume times element density. If the `CONSTANT SPHERE RADIUS` command line appears, then the radii for all particles will remain constant.

The DENSITY FORMULATION command controls the method used to update the density, and thus SPH radii, during the analysis. The default MATERIAL option uses the material density and nodal mass to compute a volume for each particle at a given time. The radius is then computed as the cube root of that volume. The alternative KERNEL option computes the current density for each particle using the mass of that particle and the SPH kernel density function. The KERNEL option may be necessary when large expansions of particles are expected (such as when modeling large density changes in gases). The MATERIAL option generally changes particle densities, and thus radii, less than the KERNEL option, so is appropriate for analyses that do not have large density fluctuations.

The MAXIMUM RADIUS CHANGE command may be used to control how much a given SPH particle is allowed to expand or shrink over the course of the analysis. For example, setting

```
MAXIMUM RADIUS CHANGE = 0.2 10.0
```

would indicate that the radius of any particle cannot become less than 20% or more than 1000% of the particle's original radius. By default, the particle can shrink or expand as much as its density change requires. This option may be useful, for example, in explosive modeling if the particle radii are expanding excessively causing analysis performance and stability problems.

An analysis using SPH may be inherently one-, two-, or three-dimensional. Dimensionality of the problem may be specified using the PROBLEM DIMENSION command line. The possible value for this command line is 1 (one-dimensional), 2 (two-dimensional), or 3 (three-dimensional). The default value is 3 for three-dimensional. The internal SPH kernel functions are modified depending on the value set on the PROBLEM DIMENSION command line. If, for example, a problem is inherently two-dimensional in nature, more accurate results may be obtained by specifying the dimension of the problem as 2.

The FORMULATION command defines the SPH formulation to use, and has the following options:

- SPH, general smoothed particle hydrodynamics.
- MASS_PARTICLE, Particles only have mass, and no stiffness. Particles do not interact with each other in any way.

The MONAGHAN EPSILON and MONAGHAN N commands set the Monaghan viscosity coefficients. Monaghan viscosity may allow the SPH algorithm to behave better, particularly in tension. Standard values to use are 0.2 for epsilon and 4 for n. See (Reference [16]) for more information on the usage of these parameters.

The SPH ALPHAQ PARAMETER and SPH BETAQ PARAMETER commands control the SPH viscosity terms. Larger values lead to more viscosity. Alpha is associated with a viscosity parameter that is proportional to the dilatational modulus of the material. Betaq is associated with a viscosity parameter that is proportional to the change in material density over a time step.

Utility commands. In addition to the SPH-related command lines described above (which appear in the PARTICLE SECTION command block), the BEGIN SPH OPTIONS is another SPH-related command block:

```
BEGIN SPH OPTIONS
  SPH SYMMETRY PLANE <string>+X|+Y|+Z|-X|-Y|-Z
    <real>position_on_axis(0.0)
  SPH DECOUPLE STRAINS: <string>material1 <string>material2
END [SPH OPTIONS]
```

If used, this command block should be placed in the Presto region scope (in contrast to most SPH-related commands which are in the `PARTICLE SECTION` command block). The symmetry conditions are applied to all SPH element blocks.

The `SPH SYMMETRY PLANE` command line may be used to reduce model sizes by specifying symmetry planes and modeling only a portion of the model. Due to the non local nature of SPH element integration, symmetry planes cannot be defined with boundary conditions alone; these planes must be explicitly defined. A plane is defined by an outward normal vector aligned with one of the axes (`+X`, `+Y`, `+Z`, `-X`, `-Y`, `-Z`) and some point on the axis, which represents a point in the plane. Suppose for example, the outward normal to the plane of symmetry is in the negative *Y*-direction (`-Y`) and the plane of symmetry passes through the *y*-axis at $y = +2.56$. Then the definition for the symmetry plane would be:

```
SPH SYMMETRY PLANE -Y +2.56
```

The `SPH DECOUPLE STRAINS` command line prevents two dissimilar materials from directly interacting. Generally, the material properties at a particle are the average of the material properties from nearby particles. If particles with very dissimilar material properties are interacting, this interaction can create problems. The `SPH DECOUPLE STRAINS` command line ensures that particles with very dissimilar material properties do not directly interact by material-property averaging, but instead interact with a contact-like interaction. The two material types that are not to interact are specified by the parameters `material1` and `material2`. These parameters will appear as material names on a `MATERIAL` command block.

Conversion to particles at initialization. To facilitate the creation of particle-based meshes, an additional SPH-related command block is the `BEGIN CONVERSION TO PARTICLES AT INITIALIZATION` block:

```
BEGIN CONVERSION TO PARTICLES AT INITIALIZATION
  BLOCK = <string list>block_names
  SECTION = <string>particle_section
END [CONVERSION TO PARTICLES AT INITIALIZATION]
```

This block, which must be placed in the Presto region scope, specifies that an input mesh consisting of *solid elements* be converted into particles at initialization. Particle elements are created at the centroid of each element with appropriate mass and volume, and the initial solid elements of the input mesh are deleted. Any initial and boundary conditions are inherited from the corresponding initial solid element; furthermore, the particle will be associated with the same node set or side set which *any node* of the parent element belongs to.

Only elements within the blocks specified by the BLOCK command will be converted to particles. Each new element will be assigned an arbitrary, but new and unique global numerical ID.

The SECTION command must specify a user-defined section of SPH type; it is important to note that the SECTION command originally associated with the initial solid element-block command block(s) will therefore be disregarded. It is worth noting that a particle section should *not* be specified in the initial solid element-block command block(s), since particle sections are *not* compatible with solid element input mesh topologies; instead, a compatible solid element section should be specified in the initial element-block definition(s).

Display. For purposes of visualizing the element stresses, it may be necessary to copy these element variables into nodal variables. This can easily be done by defining a USER VARIABLE command block (Section 10.2.4) in conjunction with a USER OUTPUT command block (Section 9.3.2). Once the nodal variable is defined, it can be output in a RESULTS OUTPUT command block (Section 9.3.1). An example is provided below. The SPH element blocks for the problem are element blocks 20, 21, and 22. All other element blocks are non-SPH elements.

- In the region scope:

```
# Define the variable
BEGIN USER VARIABLE nodal_stress
  TYPE = NODE SYM_TENSOR LENGTH = 1
END
#
# Copy element stress to nodal stress
BEGIN USER OUTPUT
  BLOCK = block_20 block_21 block_22
  COPY ELEMENT VARIABLE stress TO NODAL VARIABLE nodal_stress
END
# Output nodal stress
BEGIN RESULTS OUTPUT output_presto
  DATABASE NAME = sph.e
  DATABASE TYPE = exodusII
  AT TIME 0.0 INCREMENT = 1.0e-04
  NODAL nodal_stress
END RESULTS OUTPUT output_presto
```



Known Issue: For problems with rotations, the SPH algorithm generates rotational spring like forces on the boundary because of known deficiencies in the algorithm in distinguishing between rigid body rotation and deformation. The SPH algorithm does not conserve rotational or angular momentum.



Known Issue: If a BEGIN PRESSURE (Section 7.6.1) block is used with the CONVERSION TO PARTICLES AT INITIALIZATION command, the pressure is not applied. Pressure can be applied to particles by using the ELEMENT DEATH command block (Section 6.5).



6.2.14. Superelement Section

```
BEGIN SUPERELEMENT SECTION <string>section_name
   BEGIN MAP
     <integer>node_index_1 <integer>component_index_1
     <integer>node_index_2 <integer>component_index_2
     ...
     <integer>node_index_n <integer>component_index_n
   END
   BEGIN STIFFNESS MATRIX
     <real>k_1_1 <real>k_1_2 ...  <real>k_1_n
     <real>k_2_1 <real>k_2_2 ...  <real>k_2_n
     ...         ...          ...  ...
     <real>k_n_1 <real>k_n_2 ...  <real>k_n_n
   END
   BEGIN DAMPING MATRIX
     <real>c_1_1 <real>c_1_2 ...  <real>c_1_n
     <real>c_2_1 <real>c_2_2 ...  <real>c_2_n
     ...         ...          ...  ...
     <real>c_n_1 <real>c_n_2 ...  <real>c_n_n
   END
   BEGIN MASS MATRIX
     <real>m_1_1 <real>m_1_2 ...  <real>m_1_n
     <real>m_2_1 <real>m_2_2 ...  <real>m_2_n
     ...          ...        ...  ...
     <real>m_n_1 <real>m_n_2 ...  <real>m_n_n
   END
   FILE = <string>netcdf_file_name
   CONNECTIVITY MAPPING =
                 BY GLOBAL NODE PROXIMITY <real>dist
   CONNECTIVITY MAPPING = BY GLOBAL NODE ID
   DIAGONALIZE MASS MATRIX = OFF|ON (OFF)
   COROTATION FORMULATION = OFF|ON (OFF)
END [SUPERELEMENT SECTION <string>section_name]
```

A superelement allows definition of an arbitrary sized connectivity element with full stiffness, damping, and mass matrices. The superelement stiffness is linear and remains constant in time.



Known Issue: Superelements are not compatible with several modeling capabilities:

- They cannot be used with
  - contact,
  - element death,
  - node-based, power method, or Lanczos critical time step estimation methods, and
  - some preconditioners (such as FETI) for implicit solutions.
- Superelements are not fully tested with restart.

6.2.14.1. Defining a Superelement

A superelement may be defined either (i) in a separate NetCDF file or (ii) directly in the input deck.

A superelement may be defined in an input file using the `FILE` option. This command expects a NetCDF file generated by the Salinas Craig-Bampton reduction capability. The Salinas output file contains superelement mass, damping, and stiffness matrices, as well as locations and nodal IDs of the superelement interface nodes. When defining a superelement from an input file, the element block used to define the superelement should be excluded from the Sierra/SM analysis input mesh; instead, the superelement input file will fully define this block.

Superelement interface nodes can be merged onto Sierra/SM model nodes to make a contiguous link. The default method to merge superelement interface nodes onto the rest of the mesh is to merge nodes with the same nodal ID using the command

```
CONNECTIVITY MAPPING = BY GLOBAL NODE ID
```

Alternatively, spatially coincident nodes may be merged with the command

```
CONNECTIVITY MAPPING = BY GLOBAL NODE PROXIMITY
```

When using proximity mapping, an absolute search distance may be given to merge nearby nodes. Any superelement interface node not merged on to a mesh node will be created on the fly and given an ID number and location as defined by the NetCDF file.

A superelement may also be defined directly in the input deck. When using this method, each component of the superelement mass, damping, and stiffness matrices must be specified. The superelement itself must also be represented by an element in the input mesh file. The block of elements used to define a superelement must contain exactly one finite element of arbitrary topology of the user's choosing, e.g., a valid geometric topology (hexahedral, rod, tetrahedral, etc.). The nodes of a superelement can be shared with other elements or attached only to the superelement. The superelement can have additional internal degrees of freedom that are not present in the mesh file. However, if output values are desired on a node, that node must be present in the input mesh file.

- The `MAP` command block defines the mapping from nodal degrees of freedom to the local degrees of freedom in the stiffness and mass matrices for the superelement. If the superelement is defined using the `FILE` command, this mapping information is contained in that file and need not be specified in the input deck.
  If defined, the map is of size $M \times 2$, where M is the number of degrees of freedom in the superelement. The first column contains each index i_{nod} of the node in the superelement, $0 \leq i_{nod} \leq N$, where N is the number of nodes in the superelement, while the second column contains the component corresponding to each superelement degree of freedom j_{comp}, $0 \leq j_{comp} \leq 6$.
  Any $i_{nod} = 0$ indicates a degree of freedom internal to the superelement. A superelement may have any number of internal degrees of freedom which need not correspond to any nodes actually present in the mesh. Any $i_{nod} > 0$ represents the corresponding node index in the element. For example, a superelement consisting of an eight-node hexahedral element

would have $N = 8$ and $1 \leq i_{nod} \leq 8$ corresponding to each node in the hexahedron. Component indices $j_{comp} = 0$ represent internal degrees of freedom, while $1 \leq j_{comp} \leq 6$ indicate physical components on nodes in the mesh; for a standard node, component indices 1, 2, and 3 correspond to x, y, and z translational degrees of freedom. Components 4, 5, and 6 correspond to the x, y, and z rotational degrees of freedom for structural elements.
The following is an example superelement definition for a three-degree-of-freedom truss element lying along the x-axis. The superelement is defined in the mesh file using a two-node rod element. Degrees of freedom 1 and 3 are mapped to x degrees of freedom of the end nodes of the rod element. Degree of freedom 2 is internal to the superelement.

```
BEGIN SUPERELEMENT SECTION truss_x3
  BEGIN MAP
    1 1
    0 0
    2 1
  END
  BEGIN STIFFNESS MATRIX
    100 -100    0
   -100  200 -100
      0 -100  100
  END
  BEGIN DAMPING MATRIX
    1 -1  0
   -1  2 -1
    0 -1  1
  END
  BEGIN MASS MATRIX
    0.25 0.00 0.00
    0.00 0.50 0.00
    0.00 0.00 0.025
  END
END [SUPERELEMENT SECTION <string>section_name]
```

- The `STIFFNESS MATRIX` command block defines the $N \times N$ stiffness matrix for the superelement. If the superelement is defined using the `FILE` command, this stiffness information is contained in that file and need not be specified in the input deck. The number of rows and columns in the stiffness matrix must be the same as the number of rows in the `MAP` command block. The stiffness matrix should be symmetric. If the input matrix is not symmetric, it will be made symmetric by $K_{sym} = \frac{1}{2}(K_{input} + K_{input}^T)$. To guarantee stability for explicit dynamics and solution convergence for implicit statics/dynamics, the stiffness matrix should be positive definite.
- The `DAMPING MATRIX` command block defines the $N \times N$ damping matrix for the superelement. If the superelement is defined using the `FILE` command, this damping information is contained in that file and need not be specified in the input deck. The number of rows and columns in the damping matrix must be the same as the number of rows in the `MAP` command block. The damping matrix should generally be symmetric. This command block is optional. If a damping matrix is not defined, no damping will be used by default.
- The `MASS MATRIX` command block defines the $N \times N$ mass matrix for the superelement. If

the superelement is defined using the FILE command, this mass information is contained in that file and need not be specified in the input deck. The mass matrix must have the same dimensions as the stiffness matrix. The mass matrix need not be symmetric; however, to guarantee stability for explicit dynamics and solution convergence for implicit statics/dynamics, the mass matrix should be positive definite.

- The FILE command provides an alternative to defining stiffness, damping, mass, and connectivity in the input file. Instead, this information can be imported from an external file. External codes (such as Salinas) are able to compress larger models into superelements and output the matrices in the NetCDF format. The netcdf_file_name defines the path to the file that contains the mass, damping, and stiffness matrix definitions as well as interface node locations and connectivity. All superelement matrices should be defined either in the input deck or in the NetCDF file. The damping matrix is optional; if it is not defined in the NetCDF file, no damping will be used by default. In the NetCDF file the stiffness matrix must be named Kr, the mass matrix Mr, and the damping matrix Cr.
- Superelements may have a full, consistent mass matrix. This consistent mass matrix is incompatible with some code capabilities such as contact, MPCs, or kinematic boundary conditions. The mass matrix may be optionally diagonalized via the DIAGONALIZE MASS MATRIX, enabling the superelement to be more compatible with some of these capabilities.
- The COROTATION FORMULATION command should be used when superelement connection nodes experience large rigid body rotations. The corotation formulation incorporates a rotation matrix to minimize the root mean square deviations between the model coordinates and the current coordinates of the superelement connection nodes into superelement computations.

6.2.14.2. Superelement Output Variables

Superelement nodes have the same variables as traditional nodes (mass, displacement, velocity, etc.). Only the element time step is defined as an output variable on the superelement itself.

6.2.14.3. Specifying Initial Conditions and Loads on Superelements

Superelements currently support constant initial velocities and gravity loads. Initial velocities and gravity loads on superelements are specified using special input syntax in the INITIAL VELOCITY and GRAVITY command blocks (consult Section 7.5 and Section 7.8 for general details).

Superelements only support constant initial velocities given by a magnitude and direction. To specify an initial unit velocity in the *y*-direction on all blocks in a given model, including superelement super_elem:

```
BEGIN INITIAL VELOCITY
  INCLUDE ALL BLOCKS
  SUPER ELEMENT = super_elem
  DIRECTION = Y
```

```
  MAGNITUDE = 1.
END
```

Superelements only support constant gravity loadings given by a gravitational constant and component direction. To specify a gravity loading in the *y*-direction on all blocks in a given model, including superelement `super_elem`:

```
BEGIN GRAVITY
  INCLUDE ALL BLOCKS
  SUPER ELEMENT = super_elem
  COMPONENT = Y
  GRAVITATIONAL CONSTANT = 386.4
  SCALE FACTOR = 1.
END
```



6.2.15. User Mount Section

```
BEGIN USER MOUNT SECTION <string>user_mount_section_name
  MOUNT TYPE = FAIL_TRUSS|SPRING_DASH_POT|SNUBBER|
     TORSIONAL_SPRING|6_DOF_LINEAR_SPRING|USER_SUBROUTINE
  PARAMETERS = <real list>data
  PARAMETERS VARIABLE = <string>attribute_name
  T AXIS = <real> <real> <real>
  T AXIS VARIABLE <string>attribute_name
  S AXIS = <real> <real> <real>
  S AXIS VARIABLE <string>attribute_name
  ZERO LENGTH R AXIS = <real> <real> <real>

  MOUNT FORCE SUBROUTINE = <string>mountForceSub
  MOUNT INIT  SUBROUTINE = <string>mountInitSub
  MOUNT SIZE  SUBROUTINE = <string>mountSizeSub
END [USER MOUNT SECTION <string>user_mount_section_name]
```

The `USER MOUNT SECTION` command block provides a capability to link two nodes with an arbitrary nonlinear connection. If this command block is referenced in an element block of three-dimensional, two node elements, the elements in the block will be treated as user mount elements. The name, `user_mount_section_name`, can be used by the `SECTION` command line in a `PARAMETERS FOR BLOCK` command block.

The `MOUNT TYPE` is command specifies a small number of `USER MOUNT`s that have been implemented directly in the code rather than through user subroutines. Note that the user mounts were originally developed for use in Naval applications and are also called Navy Mounts (Nmounts). When defining a mount via user subroutine (the typical use case), specify `MOUNT TYPE = USER_SUBROUTINE`.

User mount parameters can be specified for all mounts in an element block with the `PARAMETERS` command. `PARAMETERS` specifies the user defined parameters for this mount, constant for all elements that use this section. The number of parameters should match the number required by

the specified MOUNT TYPE; if the wrong number of parameters are provided an error is issued. Orientation dependent parameters may need re-ordering given a different initial orientation (see below).

Alternatively, the PARAMETERS command, along with the optional AXIS commands discussed below, has a variant with VARIABLE that allows the setting of mount parameters that vary from element to element. When the VARIABLE variants of commands are used, the command specifies the name of an attribute field on the input mesh that contains the parameter(s). These attributes should be sized appropriate to the MOUNT TYPE when PARAMETERS VARIABLE is used. Similarly, an attribute specified by T AXIS VARIABLE must contain three entries per element, the vector components of the *t*-axis at each element.

Optionally, the T AXIS and S AXIS commands are used to specify an initial orientation of the user mount in a local Cartesian *r*, *s*, and *t* coordinate system (T AXIS VARIABLE and S AXIS VARIABLE should be used when the initial orientation varies by element). The *r*-axis lies along the length of the user mount. Either the *s*-axis or the *t*-axis can be specified as a vector in the global coordinate system. If the axis provided is not orthogonal to *r* then the code will alter it so that it is orthogonal to *r* as described below. The *t*-axis is computed from the cross product of the *s*-axis and the *r*-axis. The *s*-axis is then recomputed as the cross product of the *r*-axis and the *t*-axis to ensure that the *s*-axis is orthogonal to the *r*-axis. These local direction vectors are all normalized, so the input vectors do not have to be unit vectors. If the initial position of the *t*-axis is to be parallel to the global Z-axis, the command line

```
T AXIS = 0 0 1
```

should be used.

The *s*-axis and *t*-axis are orthogonal to the user mount *r*-axis, so they will change position as the user mount moves through space. This change in position is tracked internally by the computations for the element. If neither a *s*-axis nor a *t*-axis is specified, a default one will be used, following the above rule-set.

The ZERO LENGTH R AXIS command can be used to explicitly specify the *r*-axis. By default the user mount r-axis aligns with the bar element. The ZERO LENGTH R AXIS will override this default so that the R axis is always in the given constant direction. This command is particularly useful for zero length mounts where the bar aligned r-axis is ill-defined. If a mount has approximately zero length, and the ZERO LENGTH R AXIS command is not given the global X-axis will be used for the r-axis.

6.2.15.1. User Mount API

The MOUNT FORCE|INIT|SIZE SUBROUTINE commands can be used to define dynamically linked mount response functions. See Chapter 10 for more information on building and executing user subroutines. The API for these specific subroutines is given below.

6.2.15.2. Coordinate Frame

Typically mount interaction routines are written entirely within the mount element local coordinate frame. The frame is shown in Figure 6-23.

- X_{elem} is the normalized vector form node 1 to node 2. This may change as the structure deforms.
- V is the user provided orientation vector. This vector must not be completely parallel to X_{elem} in the initial configuration.
- Z_{elem} is the normalized cross product of X_{elem} and V
- Y_{elem} is the normalized cross product of Z_{elem} and Y_{elem}

The direction cosines are available as part of the API, should the mount require them. Note that the frame may rotate as the structure deforms. Zero length coordinate systems can pose particular problems. At this time, such mounts require an associated coordinate frame, and do not rotate with the structure. Note that there are several definitions of the element coordinate frame. In this API a convention with the y element direction in the same plane as the orientation vector, v, and the beam axis is adopted.

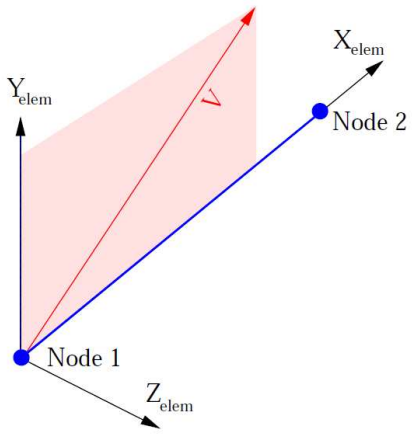


Figure 6-23. User Mount Coordinate System

6.2.15.3. Mount Output Variables

The user mount defines a number of specific element output variables.

- `user_mount_r_axis`, `user_mount_s_axis`, `user_mount_t_axis`: vector quantities that can be used to visualize the user mount coordinate system ensuring it is setup and rotated as expected.
- `element_state_variables`: state variables used for the mount computations. The state variable array size and meaning will vary based on the user subroutine calls and/or mount type

6.2.15.4. Mount Sizing Routine

The host code must allocate memory in which to read the parameters associated with each mount and the state variables used by the mount. For a given mount type these sizes must be constant, but the size can vary on different types of mounts. The mount sizing routine is used to define these sizes and has the following definition.

```
SUBROUTINE SIZE_MOUNT:

  INPUT DATA:
    NONE

  OUTPUT DATA:
    INTEGER nParam: The number of parameters used to define each
                    mount. For example gap length in a gap element

    INTEGER nState: The number of state variables used by the
                    mount.

    INTEGER nActive: An indicator of which degrees of freedom are
                     active. This integer is a logical sum of
                     contributions from each dof, where X=1, Y=10,
                     Z=100, RotX=1000, RotY=10000, RotZ=100000. Thus,
                     a mount that couples only translations has
                     nActive=111, while a mount with only rotations is
                     111000. A mount with all six is 111111.

  RESTRICTIONS:
    For a given mount type the sizing routine must always return the
    same number of parameters and number of state variables. Note,
    This routine may be called by the host code at any time, for example
    at initialization or immediately before writing a restart file.
```

6.2.15.5. Mount Initialization Routine

The mount initialize routine provides an opportunity for the mount to set its initial state data. The host code will call the mount initialize routine once at the start of the run. The mount initialize routine will be called on each mount element individually.

```
SUBROUTINE INITIALIZE_MOUNT:

  INPUT DATA:
    REAL startTime: The analysis time at the start of the first analysis
                    load step or time step.
```

REAL X_Elem(3): Mount inputs and outputs will be defined in a local coordinate system computed by the host code. X_Elem is a vector aligned with the mount element. Y_elem and Z_elem are orthogonal to X_Elem. The origin of the element coordinate system is located at the first node of the mount element, with X_Elem as the unit vector of the element coordinate system in global XYZ coordinates. Based on mount configuration at start of analysis.

REAL Y_Elem(3): Unit vector of the element coordinate system in global XYZ coordinates. Based on mount configuration at start of analysis.

REAL Z_Elem(3): Unit vector of the element coordinate system in global XYZ coordinates. Based on mount configuration at start of analysis.

REAL coord1(6): The element coordinates of node 1 on the mount at the beginning of the analysis. Order is (translational_x, translational_y, translational_z, rotational_x, rotational_y, rotational_z)

REAL coord2(6): The element coordinates of node 2 on the mount at the beginning of the analysis.

REAL param(nParam): The parameters associated with the current mount element. These parameters may be set in the input deck or read from the mesh file attributes. The number of parameters read by the host code and passed to this routine is the same 'nParam' value that is defined by the 'SIZE_MOUNT' routine.

OUTPUT DATA:

REAL sNew(nState): The initial mount state variables at the start of the analysis. The number of state variables passed out is the same 'nState' value that is defined by the 'SIZE_MOUNT' routine.

REAL Z_ElemOut(3): The X_Elem of the local coordinate system will always be mount aligned. Element frame is always orthogonal. The host code will compute a valid element coordinate system and pass it to the initialization routine. The mount initialize routine may modify the initial layout of the element coordinate system by passing back a new

Z_Elem in Z_ElemOut. Z_ElemOut is defined in global XYZ coordinates. Note, the Z_Elem used by the host code must be orthogonal to X_Elem. If Z_ElemOut is not orthogonal to X_Elem the host code will update it such that: Y_Elem = Cross(Z_ElemOut, X_Elem), and Z_Elem = Cross(X_Elem, Y_Elem)

INTEGER errorCode: If the mount routine completed without hitting any errors the errorCode should be set to zero. A positive errorCode indicates the mount routine hit a fatal error. If a mount returns a fatal error the host code will report the mount ID and errorCode in the log file and then abort execution gracefully as soon as possible. A negative errorCode indicates the mount hit a non fatal warning. If a mount returns a warning code the host code will report the mount ID and errorCode in the log file and continue execution.

RESTRICTIONS:

a) The mount initialize routine may not modify in any way any of the input arguments. Any such changes will at best be ignored and at worst may corrupt the host code's database causing unpredictable results.

b) The mount initialize routine must never call abort or stop, if the mount encounters an error it must return an error code.

c) The mount must explicitly set all output variables. All sNew values should always be set to something. The mount should always set the errorCode to something (zero if routine executed without any errors). If the mount just uses the default local coordinate system, initialize routines should copy ''Orient'' into ''OrientOut.''

d) The mount initialize routine must not make any parallel calls and should not be aware that it is running in parallel or running on a specific processor (the host code may move it from one processor to another during the run).

e) Reading or writing to/from any files within the mount code is highly discouraged. If done the file should be opened and closed each call rather than left open.

f) The mount must not store any recoverable information in any location other than the state variables that are passed out of he

```
    INITIALIZE MOUNT routine. For example the mount routine should
    not set and then retrieve a static variable or attempt to do its
    own memory allocation and data storage.
```

6.2.15.6. Mount Execution Routine

The host code will call the mount execute routine any time the mount force must be evaluated. The mount execute routine takes as input the state of the mount at the beginning of a load step or time step. The execute routine returns the force in the mount during the load step. The execute routine should also update any internal mount state variables to the end of load step values. The mount execute routine will be called on each mount element individually. Note the mount execute routine may be called several times during a load step, for example for use in an iterative solver.

```
SUBROUTINE COMPUTE_MOUNT_FORCE:

  INPUT DATA:
    REAL startTime: The analysis time at the start of the current
                    analysis load step or time step.

    REAL deltaT: The length of the current analysis time step or load
                 step. Note, the code may call the mount execute routine
                 to probe the mount stiffness, if that is the
                 case the deltaT passed in for the probe may be 0.0.

    REAL X_Elem(3): Mount inputs and outputs will be defined in a local
                    element coordinate system. X is a vector aligned
                    with the mount element. Y and Z are orthogonal to
                    X. The origin of the element coordinate system is
                    located at the first node of the mount element. The
                    host code will update the element coordinate system
                    if the mount element undergoes rotation.  Vectors
                    defined in the mount configuration at end of load
                    step.

    REAL Y_Elem(3): Unit Y vector of the element coordinate system in
                    global XYZ coordinates. Based on mount configuration
                    at end of load step.


    REAL Z_Elem(3): Unit Z vector of the local coordinate system in
                    global XYZ coordinates. Based on mount configuration
                    at end of load step.

    REAL coord1(6): The element coordinates of node 1 on the mount at
                    the end of the load step. Order is (translational_x,
                    translational_y, translational_z, rotational_x,
```

rotational_y, rotational_z)

REAL coord2(6): The element coordinates of node 2 on the mount at the end of the load step.

REAL disp1(6): The element displacement of node 1 at the end of the load step.

REAL disp2(6): The element displacement of node 2 at the end of the load step.

REAL vel1(6): The element velocity of node 1 at the middle of the load step.

REAL vel2(6): The element velocity of node 2 at the middle of the load step.

REAL param(nParam): The parameters associated with the current mount element. These parameters may be set in the input deck or read from the mesh file attributes. The number of parameters read by the host code and passed to this routine is the same 'nParam' value that is defined by the 'SIZE_MOUNT' routine.

REAL sOld(nState): The mount state variables at the start of the load step. The number of state variables passed in is the same 'nState' value that is defined by the 'SIZE_MOUNT' routine.

OUTPUT DATA:

REAL force1(6): The element coordinate forces and moments at node 1 at the end of the load step.

REAL force2(6): The element coordinate forces and moments at node 2 during the load step.

REAL sNew(nState): The updated mount state variables at the end of the load step. The number of state variables passed out is the same 'nState' value that is defined by the 'SIZE_MOUNT' routine.

INTEGER errorCode: If the mount routine completed without hitting any errors the errorCode should be set to zero. A positive errorCode indicates the mount routine hit a fatal error. If a mount returns a fatal

```
                    error the host code will report the mount ID and
                    errorCode to the user then abort execution
                    gracefully as soon as possible. A negative
                    errorCode indicates the mount hit a non fatal
                    warning.  If a mount returns a warning code the
                    host code will report the mount ID and errorCode
                    to the user and continue execution.

RESTRICTIONS:
  a) The mount execute routine may not modify in any way any of the
     input arguments. Any such changes will at best be ignored and at
     worst may corrupt the host code's database causing unpredictable
     results.

  b) The mount execute routine must never call abort or stop, if the
     mount encounters an error it must return an error code.

  c) The mount must explicitly set all output variables. The return
     force should be set to zero if there is no force. sOld values
     should be copied into sNew if the mount state variables were not
     changed. The mount should always set the errorCode to something
     (zero if routine executed without any errors).

  d) The mount execute routine must not make any parallel calls and
     should not be aware that it is running in parallel or running on
     a specific processor (the host code may move it from one
     processor to another during the run).

  e) Reading or writing to/from any files within the mount code is
     highly discouraged.

  f) The mount must not store any recoverable information in any
     location other than the state variables that are passed into and
     out of the COMPUTE MOUNT FORCE routine. For example the mount
     routine should not set and then retrieve a static variable or
     attempt to do its own memory allocation and data storage.
```

6.2.15.7. Example

The following is an example of use of the API for defining a simple mount. This example is written in Fortran 77. The mount designed here is a truss element with a failure criterion. The truss element behaves as a linear spring. If the strain in the truss becomes too large the truss breaks. A broken truss can never again support tensile or compressive loads even if later the strain is reduced below the break strain value.

```
//================================================================
```

```
      subroutine fail_truss_size(int nParam, int nState, int nActive)
      int nParam
      int nState
c
c Param(1) is the stiffness of the truss, expressed in force per strain
c Param(2) is the critical value of tensile strain at which the truss
c breaks.
c
      nParam = 2
c
c State(1) is the initial length of the truss State(2) indicates if the
c truss is broken or not, 1 is intact, 2 is broken
c
      nState = 2
c Truss does not couple to rotation,
      nActive = 111
      end
//=================================================================
subroutine fail_truss_initialize(startTime,
     &                           X_Elem, Y_Elem, Z_Elem,
     &                           coord1, coord2, param,
     &                           sNew, Z_ElemOut, errorCode)
c
c Input arguments
c
      double startTime
      double X_Elem(3), Y_Elem(3), Z_Elem(3)
      double coord1(6), coord2(6)
      double param(2)
c
c Output arguments
c
      double sNew(2)
      double Z_ElemOut(3)
      int errorCode
c
c Compute and store initial truss length, this is just the length in R
c Set the truss as initially unbroken
c
      sNew(1) = fabs(coord1(1)-coord2(1))
      sNew(2) = 1
c
c If truss is initially zero length, problem ill defined, abort
c
      if(sNew(1) == 0.0) then
        errorCode = 1
        return
```

```
      endif
c
c Use default element coordinate system
c
      Z_ElemOut(1) = Z_Elem(1)
      Z_ElemOut(2) = Z_Elem(2)
      Z_ElemOut(3) = Z_Elem(3)
      errorCode = 0
      end
//=====================================================================
subroutine fail_truss_compute(startTime, deltaT,
     &                         X_Elem, Y_Elem, Z_Elem,
     &                         coord1, coord2,
     &                         disp1, disp2,
     &                         vel1, vel2,
     &                         param,
     &                         sOld,
     &                         force1, force2,
     &                         sNew,
     &                         errorCode)
c
c Input arguments
c
      double startTime, deltaT
      double X_Elem(3), Y_Elem(3), Z_Elem(3)
      double coord1(6), coord2(6)
      double disp1(6), disp2(6)
      double vel1(6), vel2(6)
      double param(2)
      double sOld(2)
c
c Output arguments
c
      double force1(6), force2(6)
      double sNew(2)
      int errorCode
c
c Local variables
c
      double newLen
      double curStrain
      double curForce
c
c Check if truss is 'broken' if so it no force
c
      if(sOld(2) == 2) then
        force1(1) = 0.0
```

```
        force1(2) = 0.0
        force1(3) = 0.0
        force1(4) = 0.0
        force1(5) = 0.0
        force1(6) = 0.0
        force2(1) = 0.0
        force2(2) = 0.0
        force2(3) = 0.0
        force2(4) = 0.0
        force2(5) = 0.0
        force2(6) = 0.0
        sNew(1) = sOld(1)
        sNew(2) = sOld(2)
        errorCode = 0
        return
      endif
c
c Compute the new truss length, this is just the length in R
c
      newLen = fabs(coord1(1)- coord2(1))
c
c If length compressed to zero, element is invalid, make host code abort
c
      if(newLen == 0.0) then
        errorCode = 1
        return
      endif
c
c Compute the current strain as new length over original length.
c Compute tensile (positive) or compressive (negative) force based on
c stiffness time strain. force is applied along R with now off axis
c component. If strain exceeds critical value break the truss
c
      curStrain = (newLen/sOld(1))
      if(curStrain > param(2)) then
        sNew(2) = 2
        force1(1) = 0.0
        force1(2) = 0.0
        force1(3) = 0.0
        force1(4) = 0.0
        force1(5) = 0.0
        force1(6) = 0.0
        force2(1) = 0.0
        force2(2) = 0.0
        force2(3) = 0.0
        force2(4) = 0.0
        force2(5) = 0.0
```

```
      force2(6) = 0.0
      sNew(1) = sOld(1)
    else
      if(curStrain > 1.0) then
        curForce = (curStrain-1.0) * param(1)
      else
        curForce = -(1.0/curStrain) * param(1)
      endif
      force1(1) = curForce
      force1(2) = 0.0
      force1(3) = 0.0
      force1(4) = 0.0
      force1(5) = 0.0
      force1(6) = 0.0
      force2(1) = -curForce
      force2(2) = 0.0
      force2(3) = 0.0
      force2(4) = 0.0
      force2(5) = 0.0
      force2(6) = 0.0
      sNew(1) = sOld(1)
      sNew(2) = sOld(2)
    endif
    errorCode = 0
    end
```

6.2.16. Cylindrical Joint

```
BEGIN CYLINDRICAL JOINT SECTION <string>cylj_section_name
  AXIAL DIRECTION = <vector_3D> axial_direction
   axial direction = 0.0 0.0 -1.0
  AXIAL ROTATION STIFFNESS = <string>theta_stiffness
  AXIAL TRANSLATION STIFFNESS = <string>axial_stiffness
  INNER SURFACE = <string>inner_surface
  OUTER SURFACE = <string>outer_surface
  RADIAL ROTATION STIFFNESS = <string>yaw_stiffness
  RADIAL TRANSLATION STIFFNESS = <string>radial_stiffness
END [CYLINDRICAL JOINT SECTION <string> truss_section_name]
```

The `CYLINDRICAL JOINT SECTION` command block provides the ability to add nonlinear stiffness against the relative average motion of two surfaces (`inner_surface` and `outer_surface`) using a cylindrical system assumption. (In contrast, to kinematically prescribe a cylindrical boundary condition a user should use the `CYLINDRICAL AXIS` line command within a prescribed kinematic boundary condition command block, e.g., for prescribed displacement see 7.4.2.2.) The cylindrical joint capability is intended for modeling joints with cylindrical constraints (e.g., a disc mounted on a post) and it works best on joints that are

axisymmetric or nearly so. In addition, unspecified stiffness are assumed to be zero for these joints so that free motion may be specified (e.g., a disc free to rotate on a post which has zero `AXIAL ROTATION STIFFNESS`, while all other stiffness are specified).

This command block should be referenced in an element block of a single three-dimensional two node element (i.e., a bar or beam element in typical meshing tools). The two node element is used to define the reference axis against which the relative motion of the two surfaces is computed. The element variables for this section type are listed in 6-5 and are stored on the single two node element.

The element variables `cylj_rst_origin`, `cylj_r_axis`, `cylj_s_axis`, and `cylj_t_axis` can be used to visualize the local co-rotational Cartesian reference axes, which is constructed in much the same way as for beam or bar elements. The value of `cylj_rst_origin` is at the center of the two node element and the `cylj_r_axis` is in the direction from first node to the second node as defined by the two node element connectivity (note that the `cylj_r_axis` is *not* the radial direction). The directions `cylj_s_axis` and `cylj_t_axis` are internally computed and are orthogonal to each other and `cylj_r_axis`.

The parameter, `cylj_section_name` is user-defined and is referenced by a `SECTION` command line in a `PARAMETERS FOR BLOCK` command block.

The parameter, `axial_direction` can optionally be input to define the direction along the axis of the cylindrical joint. Sierra/SM will take this direction and determine if the normal of the beam element needs to be swapped to the opposite direction. If the command is not present, the beam normal will be used.

The function defined with `theta_stiffness` on the command line `AXIAL ROTATION STIFFNESS` is used to determine moment applied as a function of the two surfaces' relative motion illustrated in 6-24.

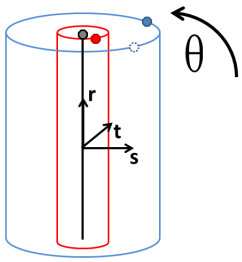


Figure 6-24. Cylindrical Joint Axial Rotation

The function defined with `axial_stiffness` on the command line `AXIAL TRANSLATION STIFFNESS` is used to determine force applied as a function of the two surfaces' relative motion illustrated in 6-25.

The function defined with `yaw_stiffness` on the command line `RADIAL ROTATIONAL STIFFNESS` is used to determine moment applied as a function of the

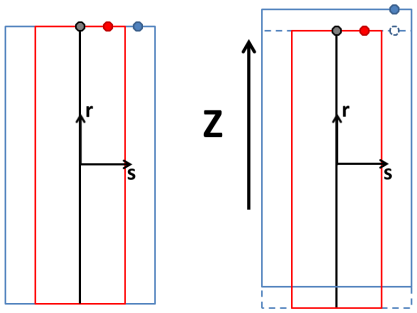


Figure 6-25. Cylindrical Joint Axial Translation

two surfaces' relative motion illustrated in 6-26. This function should generally start at (0,0) and increase monotonically positive X positive Y quadrant.

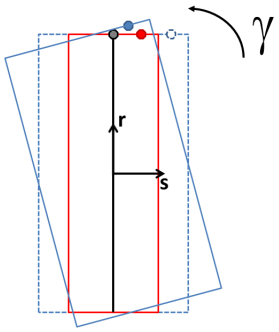


Figure 6-26. Cylindrical Joint Radial Rotation

The function defined with `radial_stiffness` on the command line `RADIAL TRANSLATION STIFFNESS` is used to determine force applied as a function of the relative motion illustrated in 6-27. This function should generally start at (0,0) and increase monotonically positive X positive Y quadrant.

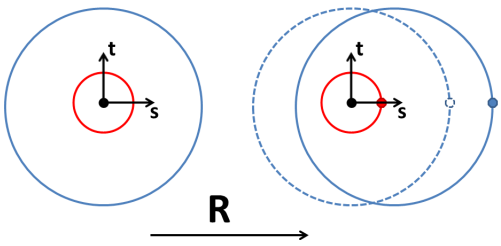


Figure 6-27. Cylindrical Joint Radial Translation

The INNER SURFACE and OUTER SURFACE commands specify the surfaces where relative motion is measured and the resulting force and moments are distributed. In a given time step, the averaged step motion in the reference Cartesian coordinate system is computed on the inner_surface and outer_surface. The accumulated translational and rotational motion is stored in the three-component vector element variables cylj_rst_coord1 and cylj_rst_theta1, respectively, for the inner_surface. Similarly, the accumulated motion for outer_surface is stored in cylj_rst_coord2 and cylj_rst_theta2.

The relative motion used for evaluating the stiffness functions is calculated by the vector math of (cylj_rst_coord2 - cylj_rst_coord1) and (cylj_rst_theta2 - cylj_rst_theta1). The relative motion of the coord values is used with the translational stiffness and the resulting force is stored in the element variable cylj_rst_force. Similarly, the theta values use the rotational stiffness and result in a moment stored in cylj_rst_moment. The element internal force and moment values are then applied equally and of opposite sign to the inner_surface and outer_surface and are distributed across each surface's nodes for the cylindrical joint element's internal force.

For output the "x" vector components of the coord, theta, force, and moment element variables correspond to the axial direction cylj_r_axis . The "y" and "z" components correspond to the cylj_s_axis and cylj_t_axis radial directions respectively, which are treated with the same translational and rotational stiffness due to the axisymmetric assumption.

Table 6-5. Variables For Cylindrical Joint

Variable Name	Type	Comments
cylj_rst_origin	Vector_3D	(Reference axis origin)
cylj_r_axis	Vector_3D	(Reference axis r direction)
cylj_s_axis	Vector_3D	(Reference axis s direction)
cylj_t_axis	Vector_3D	(Reference axis t direction)
cylj_rst_coord1	Vector_3D	(Inner surface translational coordinates)
cylj_rst_theta1	Vector_3D	(Inner surface rotational coordinates)
cylj_rst_coord2	Vector_3D	(Outer surface translational coordinates)
cylj_rst_theta2	Vector_3D	(Outer surface rotational coordinates)
cylj_rst_force	Vector_3D	(Translational force vector)
cylj_rst_moment	Vector_3D	(Rotational moment vector)



Warning: The cylindrical joint element has no mass and only increases the stiffness of the nodes in inner_surface and outer_surface. This additional stiffness is currently not reflected in the computation of the stable time step. A time step scale factor may be required to obtain a stable time step if nodes in the cylindrical joint are part of the element that is controlling the time step. No checks are made of the validity of the stiffness functions provided but the functions should only have values in the upper right and lower left quadrants of a force versus displacement or moment versus rotation plot.

6.3. Element-like Functionality

This section describes functionality in Sierra/SM that behaves similarly to finite elements, although are not implemented as such. The functionality described in this section are specified through command blocks that appear in the SIERRA scope and region scope, respectively.

6.3.1. Rigid Body

```
BEGIN RIGID BODY <string>rb_name
  MASS = <real>mass
  POINT MASS = <real>mass [AT <real>X <real>Y <real>Z]
  REFERENCE LOCATION = <real>X <real>Y <real>Z
  INERTIA = <real>Ixx <real>Iyy <real>Izz <real>Ixy
    <real>Iyz <real>Izx
  POINT INERTIA = <real>Ixx <real>Iyy <real>Izz <real>Ixy
    <real>Iyz <real>Izx
  MAGNITUDE = <real>magnitude_of_velocity
  DIRECTION = <string>direction_definition
  ANGULAR VELOCITY = <real>omega
  CYLINDRICAL AXIS = <string>axis_definition
  INCLUDE NODES IN <string>surface_name
    [IF <string>field_name <|<=|=|>=|> <real>value]
END [RIGID BODY <string>rb_name]
```

A rigid body can consist of any combination of elements—solid elements, structural elements, and point masses—except SPH elements. All nodes associated with a rigid body maintain their relative position to each other as determined at time zero when there is no deformation of the body. This means that the elements associated with the rigid body can translate and rotate through space, but they cannot deform. Element blocks defining the rigid body do not have to be contiguous. They can also adjoin deformable element blocks. Multiple rigid bodies are allowed in a model.



Warning: Truss elements currently do not work correctly when used in a rigid body unless the user includes the `INERTIA` command in the rigid body command block to specify the total mass moment of inertia of the rigid body. The code currently does not compute the inertia about the longitudinal axis of a truss because only the area is input for a truss section. The recommendation is to use a beam section rather than a truss section for rigid bodies.



Warning: The non-deformable nature of the rigid body can put it in conflict with other constraints. Refer to Appendix D for information on how conflicting constraints are handled.



Warning: Kinematic boundary conditions prescribed on a rigid body must be applied to the rigid body reference location, e.g., its center of mass. Kinematic boundary conditions for a rigid body that are not prescribed on the rigid body reference location will not be enforced. Note that, unlike non-rigid blocks, a rigid body block may rotate about the rigid body reference location if only translational boundary conditions are prescribed, so rotational boundary conditions may also be prescribed to constrain rotational movement. To apply kinematic boundary conditions to a rigid body, use the `BLOCK =` or the `RIGID BODY =` line command with the rigid body's block id or name, respectively, in the appropriate boundary condition command block. Refer to Chapter 7.

Sierra/SM creates a new node for each rigid body in the analysis. The new nodes are true nodes in that they are associated with solution fields such as displacement, velocity, and rotational velocity. These nodes will appear in a results file along with other nodes. The global node number given to the new nodes is the total number of nodes in the mesh plus one, repeated for each new rigid body node.

If an element block is declared to be a part of a rigid body, the internal force calculations are not called for the elements in that block and element time steps for explicit dynamics are not computed. Hence, for a model where all the element blocks are included in rigid bodies, there is no estimate for a global explicit dynamics time step. In this case an initial time must be specified in the `TIME CONTROL` command block (Section 3.1.1) with the command line

Explicit

```
INITIAL TIME STEP = <real>time_step
```

For the completely rigid explicit dynamics case, one also should avoid using a value larger than 1.0 for the time step scale factor (see Section 3.1.1). A time step value of 1.0×10^{-6} should work well for most such problems, though any time step may be specified. However since a completely rigid problem does not impose a stability limit on the explicit dynamics time step, specifying too large of a time step could cause loss of accuracy in the solution of the problem (due to numerical integration error, large rotations over a time step, etc.) even though the problem runs to completion. The default value of the time step scale factor is 0.9.

Specification of a rigid body requires the above command block, which appears in the SIERRA scope, plus the `RIGID BODY` command line that appears in the various `SECTION` command blocks described in this chapter. Suppose, for example, `rigidbody_1` consists of element blocks 100, 110, and 280. The `PARAMETERS FOR BLOCK` command blocks for element blocks 100, 110, and 280 must all contain a `SECTION` command line. In each case, the Section must contain a line such as:

```
RIGID BODY = rigidbody_1
```

Once element block(s) have been declared a rigid body and named appropriately (through the Section chosen), that rigid body name must appear as the name in a `RIGID BODY` command block. In the example, a `RIGID BODY` command block with the value for `rb_name` set to `rigidbody_1` must exist, i.e., in the Sierra scope,

```
BEGIN RIGID BODY rigidbody_1
END [RIGID BODY rigidbody_1]
```

The RIGID BODY command block has several optional command lines, comprising four groups of commands. One group consists of the MASS, POINT MASS, REFERENCE LOCATION, POINT INERTIA, and INERTIA command lines, a second group consists of the paired MAGNITUDE and DIRECTION command lines, a third group consists of the paired ANGULAR VELOCITY and CYLINDRICAL AXIS command lines, and a final group consists of the INCLUDE NODES IN command line. The command block can include none or any combination of these groups. If none of these commands is included, the command block supplies the value for rb_name.

Input to the MASS command line consists of a single real number that defines the total mass of the rigid body. If this line command is not present, the mass of the rigid body will be computed using the elements in the rigid body and their densities. This is the translational mass at the reference location. This command does not change the inertia terms.

The POINT MASS command line requires a real number specifying the amount of mass added to the rigid body. Optionally, the location of that mass may be specified with three real numbers. If the location is not given, the additional mass will be placed at the reference location (and will not affect the moments and products of inertia).

The REFERENCE LOCATION command line requires three real numbers defining the reference location for the rigid body. If this line command is not present, the center of mass will be used. The center of mass is calculated from the mesh and the element densities.

Input to the INERTIA command line consists of six real numbers. If present, this command line will set the inertia for the rigid body. If it is not present, moments and products of inertia are computed for the rigid body based on the reference location of the rigid body and the element masses.

Input to the POINT INERTIA command line consists of six real numbers that define moments (Ixx, Iyy, Izz) and products (Ixy, Iyx, Izx) of inertia to be added to the inertia tensor of the rigid body. This modified inertia tensor (rather than the inertia tensor based solely on element mass) is then used to calculate the motion of the rigid body.

If a rigid body contains an element that has rotational resistance (e.g., shell, beam, etc.) the nodal moment of inertia of that element is added to the diagonal components of the rigid body's inertia tensor.

An initial, translational-only velocity for the rigid body at the reference location should be specified with the MAGNITUDE and DIRECTION command lines. The MAGNITUDE command line gives the magnitude of the initial velocity applied at the reference location, and the DIRECTION command line gives a defined direction.

An initial rotational and translational velocity for a rigid body can be specified with the ANGULAR VELOCITY and CYLINDRICAL AXIS command lines. The ANGULAR VELOCITY command line gives the initial translational velocity of and rotational velocity about the reference location due to an angular velocity about some defined axis given on the CYLINDRICAL AXIS command line. If the defined cylindrical axis passes through the reference location, this command will impart only an initial rotational velocity to the rigid body. If the axis does not pass through the reference location, this command will impart both an initial translational velocity and the

initial rotational velocity to the rigid body.

The INCLUDE NODES IN command line allows a rigid body to include nodes of a surface or block of the mesh. Optionally, the nodes in the surface or block will be included in the rigid body only if the value of a field on the nodes meets a given criterion. For example, consider the rigid body block below.

```
begin rigid body rigid1
  include nodes in block_1
  include nodes in surface_3 if height = -1.0
end
```

Rigid body rigid1 will include all nodes in block_1 and those nodes on surface_3 whose value of the height field are equal to -1.0. The optional field (height in this case) must be a field known in the analysis. This field may be read in from restart, be a native field initialized upon start up, or be an initialized user-defined field. Since the entire set of nodes in block_1 will be in the rigid body, internal forces and critical time steps (for explicit) will not be computed for elements in block_1.

In summary, if a rigid body is used in an analysis, one or more of the following steps must be carried out:

- Include a RIGID BODY command block in the SIERRA scope. If desired, set reference location, mass, point mass, etc.
- Create a rigid body using one or more element blocks (except PARTICLE or element blocks). A RIGID BODY command line must appear in the SECTION command block used in the PARAMETERS FOR BLOCK command block for any element block associated with a rigid body.
- Include point mass element blocks with the rigid body if appropriate. To include point mass element blocks in a rigid body, a RIGID BODY command line must appear in the SECTION command block used in the PARAMETERS FOR BLOCK command block for those point mass element blocks.
- Associate an initial velocity or initial rotation about an axis with the rigid body, if appropriate. If any of the blocks associated with a rigid body have been given an initial velocity or initial rotation, the rigid body must have the same specification for the initial velocity or initial rotation.

The above steps involve many different command blocks. To demonstrate how to fully implement a rigid body, a specific example that exercises the various options available to a user is provided.

Suppose it is desired to create a rigid body named part_a consisting of three element blocks. Two of the element blocks, element block 100 and element block 535, are eight-node hexahedra; one of the element blocks, element block 600, consists of only point masses. The RIGID BODY command block, SECTION command block, and the element blocks to be associated with the rigid body are as follows:

```
begin solid section hex_section
```

```
  rigid body = part_a
end
begin point mass section pm_section
  rigid body = part_a
  volume = 0.1
end

begin parameters for block block_100
  material = steel
  model = elastic
  section = hex_section
end
begin parameters for block block_535
  material = aluminum
  model = elastic
  section = hex_section
end
begin parameters for block block_600
  material = mass_for_pointmass
  model = elastic
  section = pm_section
end
```

Suppose it is desired to have the rigid body initially rotating at 600 radians/sec about an axis parallel to the *x*-axis and passing through a point at coordinates (0, 10, 20). This axis would be defined using the following set of command lines:

```
define direction parallel_to_x with vector 2.0 0.0 0.0
define point off_axis with coordinates 0.0 10.0 20.0
define axis body_axis with point off_axis direction parallel_to_x
```

The initial angular velocity specification in the `RIGID BODY` command block is as follows:

```
begin rigid body part_a
  cylindrical axis = body_axis
  angular velocity = 600
end rigid body part_a
```

6.3.1.1. Multiple Rigid Bodies from a Single Block

Typically, all the elements in a block are assigned to a single rigid body. However, it is sometimes necessary to define a large number of rigid bodies. It would be unwieldy to create a separate block for each rigid body. To avoid this, it is possible to create an element attribute in the input mesh file that contains an integer ID used to denote the rigid body in which each element should belong. The `RIGID BODIES FROM ATTRIBUTES` command is used to associate the attribute with the

rigid body IDs. This command must be used in conjunction with the RIGID BODY command. These commands are used in the context of the section block. They are shown here in the SOLID SECTION block, but they can be used with any section type that supports rigid bodies.

```
BEGIN SOLID SECTION <string>sec_name
  RIGID BODIES FROM ATTRIBUTES = <integer>first_id to
    <integer>last_id
  RIGID BODY = <string>rigid_body_name
END
```

If the RIGID BODIES FROM ATTRIBUTES command is used, a series of rigid bodies will be created. These rigid bodies are named using the specified rigid_body_name, followed by an underscore (_), and then by an integer ID, which ranges between the specified value of first_id and last_id. Elements having an attribute value equal to a given ID are added to the corresponding rigid body.

When referencing the rigid body name for to apply boundary conditions, it is important to specify the entire name of the rigid body, including the underscore and ID. For example, a name such as part_a_1 would be used in a boundary condition. Similarly, specific rigid bodies created with RIGID BODIES FROM ATTRIBUTES can be have rigid body properties REFERENCE LOCATION, MASS , POINT MASS , INERTIA , and POINT INERTIA set using a name such as part_a_1.

6.3.2. Output

Sierra/SM automatically outputs quantities such as displacement for the reference location of the rigid body. The name assigned to a rigid body will be used to construct variable names that give the quantities. This lets the user identify the output associated with a rigid body based on the name assigned to the rigid body.

Immediately before the results file is written, the accelerations for nodes associated with a rigid body are updated to reflect the accelerations due to the rigid-body constraints. This ensures that the accelerations sent to the results output are correct for a given time.

Sierra/SM automatically generates and outputs global data associated with the rigid body (e.g., displacement and quaternion). See Section 9.9 for details on global variable output. Individual field components may be output as discussed in Section 9.1.4 and Table 9-4. In general the field component fieldi of the rigid body named part_a will be written to the output file(s) with the name FIELDI_PART_A.

In addition to being represented in the mesh file by the grouping of element objects that represent the rigid body each rigid body will also be represented by a sphere element. These rigid body elements will be placed in a new element block in the results output file named "RigidBodyOutputPart." Net quantities of the rigid body such as rotation rate, net force, and velocity can be plotted on these sphere elements.

6.4. Mass Property Calculations

```
BEGIN MASS PROPERTIES
  #
  # block set commands
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE BLOCK = <string list>block_names
  #
  # structure command
  STRUCTURE NAME = <string>structure_name
END [MASS PROPERTIES]
```

Sierra/SM automatically gives mass property information for the total model, which consists of all the element blocks. (The mass for the total model, for example, is the total mass of all the element blocks.) Sierra/SM also automatically outputs mass property information for each element block.

In addition to the generated mass property information, Sierra/SM allows the user to optionally define a structure composed of multiple element blocks and calculate the mass properties for this structure using the `MASS PROPERTIES` command block in the region scope.

For the total model, each element block, and any user-defined structure, Sierra/SM reports the mass and the center of mass in the global coordinate system. It also reports the moments and products of inertia, as computed in the global coordinate system, about the center of mass.

The `MASS PROPERTIES` command block contains two groups of commands—block set and structure. Each of these groups is independent of the other. Following are descriptions of the two command groups.

6.4.1. Block Set Commands

The `block set commands` portion of the `MASS PROPERTIES` command block defines a set of blocks for which mass properties are being requested, and can include some combination of the following command lines:

```
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of blocks. See Section 7.1.1 for more information about the use of these command lines for creating a set of blocks used by the command block. There must be at least one `BLOCK` or `INCLUDE ALL BLOCKS` command line in the command block.

The `REMOVE BLOCK` command line allows the user to delete blocks from the set specified in the `BLOCK` and/or `INCLUDE ALL BLOCKS` command line(s) through the string list `block_names`.

The REMOVE BLOCK command line is typically used with the INCLUDE ALL BLOCKS command line. To include all but a few of the element blocks, a combination of the REMOVE BLOCK command line and INCLUDE ALL BLOCKS should minimize input information.

Suppose that only one element block, block_300, is specified on the BLOCK command line. Then only the mass properties for that block will be calculated. If several element blocks are specified on the BLOCK command line, then that collection of blocks will be treated as one entity, and the mass properties for that single entity will be calculated. For example, if two element blocks, block_150 and block_210, are specified on the BLOCK command line, the total mass for the two element blocks will be reported as the total mass property.

6.4.2. Structure Command

The output for the mass properties will be identified by the command line:

```
STRUCTURE NAME = <string>structure_name
```

where the string structure_name is a user-defined name for the structure.

```
BEGIN ELEMENT DEATH <string>death_name
  #
  # block set commands
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  INCLUDE BLOCKS WITH MATERIAL =
           <string list>material_names
  REMOVE BLOCK = <string list>block_names
  #
  # criterion commands
Explicit  CRITERION IS AVG|MAX|MIN NODAL VALUE OF
    <string>var_name <|<=|=|>=|> <real>tolerance
Explicit  CRITERION IS ELEMENT VALUE OF
    <string>var_name <|<=|=|>=|> <real>tolerance [KILL WHEN
    <integer>num_intg INTEGRATION POINTS REMAIN]
Explicit  CRITERION IS GLOBAL VALUE OF
    <string>var_name <|<=|=|>=|> <real>tolerance
Explicit  CRITERION IS ALWAYS TRUE
  MATERIAL CRITERION = ON|OFF(OFF) [KILL WHEN
    <integer>num_intg INTEGRATION POINTS REMAIN]
  #
  # subroutine commands
Explicit  ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
Explicit  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
Explicit  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
Explicit  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
Explicit  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # evaluation commands
Explicit  CHECK STEP INTERVAL = <integer>num_steps
Explicit  CHECK TIME INTERVAL = <real>delta_t
Explicit  DEATH START TIME = <real>time
  #
  # miscellaneous option commands
  SUMMARY OUTPUT STEP INTERVAL = <integer>output_step_interval
  SUMMARY OUTPUT TIME INTERVAL = <real>output_time_interval
Explicit  DEATH ON INVERSION = OFF|ON(OFF)
Explicit  DEATH ON ILL DEFINED CONTACT = OFF|ON(OFF)
Explicit  DEATH STEPS = <integer>death_steps(1)
Explicit  AGGRESSIVE CONTACT CLEANUP = <string>OFF|ON(OFF)
Explicit  DEATH METHOD = <string>DEACTIVATE ELEMENT|DEACTIVATE NODAL MPCS|
      DISCONNECT ELEMENT|(DEACTIVATE ELEMENT)
```

```
  LOGFILE OUTPUT = FULL|SOME|MINIMAL(SOME)
  #
  # death on proximity commands
Explicit
  BEGIN DEATH PROXIMITY
    BLOCK = <string list>block_names
    NODE SET|NODESET = <string list>nodelist_names
    SURFACE = <string list>surface_names
    REMOVE BLOCK = <string list>block_names
    REMOVE NODE SET = <string list>nodelist_names
    REMOVE SURFACE = <string list>surface_names
    TOLERANCE = <real>search_tolerance
  END [DEATH PROXIMITY]
  #
  # particle conversion commands
Explicit
  BEGIN PARTICLE CONVERSION
    PARTICLE SECTION = <string>section_name
    NEW MATERIAL = <string>material_name
      MODEL NAME = <string>model_name
    TRANSFER = <string>NONE|ALL(ALL)
    MAX NUM PARTICLES = <integer>num(1)
    MIN PARTICLE CONVERSION VOLUME = <real>vol
  END [PARTICLE CONVERSION]
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
  REMOVE OVERLAP AFTER DEATH = ON|OFF (ON)
  #
  # cohesive zone setup commands
  COHESIVE SECTION = <string>sect_name
  COHESIVE MATERIAL = <string>mat_name
  COHESIVE MODEL = <string>model_name
  COHESIVE ZONE INITIALIZATION METHOD = <string>NONE|
    ELEMENT STRESS AVG(NONE)
  ALWAYS EVALUATE ALL CRITERION = OFF|ON(OFF)
END [ELEMENT DEATH <string>death_name]
```

The `ELEMENT DEATH` command block is used to remove elements from an analysis. For example, the command block can be used to remove elements that have fractured, that are no longer important to the analysis results, or that are nearing inversion (explicit only). This command block is located within the region scope. The name of the command block, `death_name`, is user-defined and can be referenced in other commands to update boundary or contact conditions based on the death of elements creating new exposed surfaces.

Any element in an element block or element blocks selected in the `ELEMENT DEATH` command block is removed (killed) when one of the criteria specified in the `ELEMENT DEATH` command block is satisfied by that element. When an element dies, it is removed permanently. Any number of `ELEMENT DEATH` command blocks may exist within a region.

When an element is killed, the contribution of that element's mass to the attached nodal mass is

removed from the attached nodes. If all the elements attached to a node are killed, the mass for the node and all associated nodal quantities will be set to zero.

In implicit calculations, elements may be killed based off all available criterion and used in conjunction with control failure in a multilevel solver block. However, material criterion is the only criterion that is truly integrated with the control failure to control the failure progression. Control failure is described in Section 4.7.

In explicit calculations, elements may be killed based on any internal variable, derived variable, material state variable, or user defined variable. Note that user-defined variables for triggering element death should be used with care. There are timing and parallel issues regarding the use of a user-defined variable with element death. User subroutines may work for element death in some situations where one might want to reference a user variable.



The quantity used in an element death criterion statement should have only a single value at each integration point. For example,

```
criterion is element value of stress(xx) > 10.0
```

is unambiguous and valid. However,

```
criterion is element value of stress > 10.0
```

is ambiguous, and will result in an error. Stress is a 3x3 tensor and there is no rational way to compare that tensor to the scalar value 10. If a variable is defined on multiple integration points, the criterion will be triggered if the value is exceeded at any of the integration points. If the command

```
criterion is element value of stress(xx,:) > 10.0
```

were used with shells with five integration points, the shells would be killed if the value of `stress(xx)` at any of the five integration points exceeded 10.

The `ELEMENT DEATH` command block contains five groups of commands—block set, criteria, evaluation, miscellaneous, and cohesive zone setup. The command block must contain commands from the block set and criteria groups. Command lines from the evaluation and miscellaneous groups are optional, as are the cohesive zone commands.

Following are descriptions of the different command groups, an example of using the `ELEMENT DEATH` command block, and some concluding remarks related to element death visualization.

6.5.1. Block Set Commands

```
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
INCLUDE BLOCKS WITH MATERIAL = <string list>material_names
REMOVE BLOCK = <string list>block_names
```

The `block set commands` portion of the `ELEMENT DEATH` command block defines a set of blocks for selecting the elements to be referenced. These command lines, taken collectively,

constitute a set of Boolean operators for constructing a set of blocks, as described in Section 7.1.1.

Element death must apply to a group of elements. There are three commands for selecting the elements to be referenced: `BLOCK`, `INCLUDE ALL BLOCKS`, and `INCLUDE BLOCKS WITH MATERIAL`. In the `BLOCK` command line, the user may list a series of blocks through the string list `block_names`. This command line may also be repeated multiple times. The `INCLUDE ALL BLOCKS` command line adds all the element blocks present in the region to the current element death definition. The `INCLUDE BLOCKS WITH MATERIAL` command line adds all the element blocks that use the specified materials to the current element death definition. There must be at least one `BLOCK`, `INCLUDE ALL BLOCKS`, or `INCLUDE BLOCKS WITH MATERIAL` command line in the `ELEMENT DEATH` command block.

The `REMOVE BLOCK` command line allows the user to delete blocks from the set specified in the `BLOCK` and/or `INCLUDE ALL BLOCKS` command line(s) through the string list `block_names`. The `REMOVE BLOCK` command line is typically used with the `INCLUDE ALL BLOCKS` command line. To include all but a few of the element blocks, a combination of the `REMOVE BLOCK` command line and `INCLUDE ALL BLOCKS` command line should minimize input information.

6.5.2. Criterion Commands

Any combination of death criteria (`CRITERION IS NODAL`, `CRITERION IS ELEMENT`, `CRITERION IS GLOBAL`, `CRITERION IS ALWAYS TRUE`, `ELEMENT BLOCK SUBROUTINE`, `MATERIAL CRITERION` as appropriate in explicit or implicit) can be specified within a single `ELEMENT DEATH` command block. However, only one user subroutine criterion (available for explicit only) may appear in a set of criteria command lines. If multiple death criteria are specified for a given element, it will be killed when the first of those criteria is met. In other words, element death with multiple criteria is an `OR` condition rather than an `AND` condition.

An explicit dynamics simulation with a material that has both a tension cutoff and a compression cutoff is an example for which element death with multiple criteria would be used—one criterion to set failure in tension and another criterion to set failure in compression. If either the tension cutoff value, as set by the tension criterion, or the compression cutoff value, as set by the compression criterion, was exceeded for an element, the element would be killed.

As a second explicit dynamics example, consider a problem using a nodal criterion and a user subroutine criterion. For this second example, the user is precluded from using a second subroutine criterion because one already exists. A combination of nodal, element, global, or material criteria could be added to the existing nodal and subroutine criteria, but another subroutine criterion could not be added.



6.5.2.1. Nodal Variable Death Criterion

```
CRITERION IS AVG|MAX|MIN NODAL VALUE OF
  <string>var_name <|<=|=|>=|> <real>tolerance
```

Any nodal variable may be used by an element death criterion. The input parameters are

described as follows:

- Nodal variables are present on the nodes of an element, and these nodal values must be reduced to a single element value for use by the criterion. The available types of reduction are `AVG`, which takes the average of the nodal values; `MAX`, which takes the maximum of the nodal values; and `MIN`, which takes the minimum of the nodal values.
- The string `var_name` gives the name of the variable. See Section 9.10 for a listing of the variables. Parenthesis syntax may be used in the variable name for selection specific variable components or integration points, see Section 9.1.
- The specified variable, with an optional component specification if the variable has components, may be compared to a given tolerance with one of five operators. The operator is specified with the appropriate symbol for less than (<) less than or equal to (<=), equal to (=), greater than or equal to (>=), or greater than (>) The given tolerance is specified with the real value `tolerance`.



6.5.2.2. Element Variable Death Criterion

```
CRITERION IS ELEMENT VALUE OF
  <string>var_name[(<integer>component_num)]
  <|<=|=|>=|> <real>tolerance |
  <string>derived_quantity[(<integer>int_num)]
  <|<=|=|>=|> <real>tolerance [KILL WHEN <integer>num_intg
  INTEGRATION POINTS REMAIN]
```

Any element variable may be used by an element death criterion. An element variable is present on the element itself, so no reduction is required, which is why the first line in the format of the above command line differs from that of the nodal criterion command line in Section 6.5.2.1.

For a criterion using an element variable, the variable name, component or integration point number, and tolerance can be specified in the same manner as defined for the nodal criterion command line.

Element state variables can also be used as criteria for element death. The syntax used to specify the variable name differs if a specific integration point is desired.

- For LAMÉ solid material models, the variable name is called out directly by name.
- For shell material models, the variable name is called out directly by name. By default the element will be killed if the criteria is met at any integration point. Parenthesis syntax (See Section 9.1) may be used to kill based on the state of a specified integration point. For example:

  ```
  criterion = element value of crack_opening_strain(2) > 0.01
  ```

  would use the value of the state variable `crack_opening_strain` at the second shell integration point through the shell thickness as the criterion for element death.

Refer to Section 9.10.2 for tables with listings of state variables for the various material models.

Derived quantities can also be used for element death. For example the following command:

```
criterion = element value of von_mises > 1.0e+05
```

will cause elements to be killed any of the integration points reach a critical value of `von_mises`. Alternatively, the command:

```
criterion = element value of von_mises(3) > 1.0e+05
```

is used to kill an element if a specific integration point reaches the critical value of `von_mises`.

For elements with multiple integration points, if an element criterion is used with an integration point variable, the integration points are killed individually as the criterion is exceeded on each integration point. The integration points are killed by linearly tapering down all components of the stress over the specified number of death steps. See Section 6.5.4.4 for more information on death steps.

The optional `KILL WHEN num_intg INTEGRATION POINTS REMAIN` portion of the command line is used to kill the entire element when enough integration points have been killed. This occurs when the number of remaining points drops down to the specified `num_intg`. For example, for shells with five integration points, the following command:

```
criterion is element value of stress(xx,:) > 10.0 kill when 2 \#
  integration points remain
```

will check the `xx` component of `stress` at each integration point. When the first three integration points reach that criterion, their stresses are linearly tapered down to zero over the specified number of death steps. Once the contributions from each of those three integration points have completely decayed to zero, leaving two remaining integration points, the element is killed. By default the element itself is killed after the first integration point dies.



6.5.2.3. Global Death Criterion

```
CRITERION IS GLOBAL VALUE OF
<string>var_name[(<integer>component_num)]
<|<=|=|>=|> <real>tolerance
```

Any global variable may be used in an element death criterion. Once the global criterion is reached, all elements specified in the `ELEMENT DEATH` command block are killed. The variable name, component number, and tolerance can be specified in the same manner as defined for the nodal or element criterion command line.

The input parameters are described as follows:

- The string `var_name` gives the name of the global variable. See Section 9.10 for a listing of the global variables.
- Parenthesis syntax may be used in the variable name to specify specific components of the variable. See Section 9.1 for more information.
- The specified variable, with an optional component specification if the variable has components, may be compared to a given tolerance with one of five comparison operators as follows:
  - less than (<),
  - less than or equal to (<=),

- equal to (=),
- greater than or equal to (>=), or
- greater than (>).

The given tolerance is specified with the real value `tolerance`.



6.5.2.4. Always Death Criterion

```
CRITERION IS ALWAYS TRUE
```

This command forces the criterion that determines whether to kill elements specified in the current death block to always be true. In other words, it causes all of those elements to immediately be killed. This command can be used to instantaneously kill a set of elements or convert that set of elements to particles. Converting a set of elements to particles in this manner at the beginning of an analysis can be an alternative to generating a mesh with particle elements.

This command is often used together with the `DEATH START TIME` command (see Section 6.5.3) to cause a set of elements to be killed or converted to particles at a certain time. This can be used as an alternative to block deactivation using the `ACTIVE FOR PROCEDURE` command (see Section 6.1.7.9).

6.5.2.5. Material Death Criterion

```
MATERIAL CRITERION = ON|OFF(OFF) [KILL WHEN
  <integer>num_intg INTEGRATION POINTS REMAIN]
```

Some material models have a failure criterion. When this material failure criterion is satisfied within an element the material models reduce the stress in these fractured or disintegrated elements to zero. The `MATERIAL CRITERION` command line can be used to link the decay of stress to zero with deletion of the element from an analysis. Removal of the fractured elements can speed computations, enhance visualization, and prevent spurious inversion of these elements that may stop the analysis.

The material models that currently support the use of the `MATERIAL CRITERION` command line are:

- `ELASTIC_FRACTURE` (Solid only, see Section 5.2.4)
- `DUCTILE_FRACTURE` (Solid only, see Section 5.2.7)
- `ML_EP_FAIL` (Solid and Shell, see Section 5.2.9)
- `PLANE_STRESS_RATE_PLASTICITY` (Shell only, see Section 5.2.30)

These material models compute the element variables “failure_flag”, “failure_decay”, and “failure_measure.” In its normal computations the “failure_flag” will have a value of 0.0. When the material is decaying the “failure_flag” will have a value of 3.0 and when it is finished decaying the “failure_flag” will have a value of 4.0. As the material is decaying the “failure_decay” will reduce from 1.0 to 0.0. The “failure_measure” value and function is explained in the Control Failure Section 4.7.

Element death will kill an element based on a material criterion when the material model has set the "failure_flag" to 4.0. For a single integration point element, this occurs when the one integration point in the element reaches a "failure_flag" = 4.0. For elements with multiple integration points, the default behavior is for the element to be killed when more than half of the integration points has failed. This behavior is the default because for multi-integration point elements, particularly shells, if there is only a single integration point left, the element is severely under-integrated. The final integration point will generally not attract more load and will never fail. This default behavior can be changed by using the optional `KILL WHEN num_intg INTEGRATION POINTS REMAIN` command. In this command, `num_intg` specifies the number of remaining integration points when the element is to be killed.

Consider an element block named `part1_ss304` that references a material named `SS304`. This material, `SS304`, uses the `DUCTILE_FRACTURE` material model (see Section 5.2.7). Consider a second element block named `ring5_al6061` that references a material named `al6061`. This material uses the `ML_EP_FAIL` material model (see Section 5.2.9). An `ELEMENT DEATH` command block with the command lines

```
BLOCK = part1_ss304 ring5_al6061
MATERIAL CRITERION = ON
```

would signal any element in `part1_ss304` that fails according to the material model `DUCTILE_FRACTURE` (in material `SS304`) and any element in `ring5_al6061` that fails according to the material model `ML_EP_FAIL` (in material `al6061`) to be killed by element death.



6.5.2.6. Subroutine Death Criterion

```
ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

A death criterion can be specified via a user-defined subroutine (see Chapter 10), which is invoked by the `ELEMENT BLOCK SUBROUTINE` command line. The string `subroutine_name` is the name of a FORTRAN subroutine that is written by the user. The user-defined subroutine for element death must be an element subroutine signature (see Section 10.2.2). The element subroutine will return an output values array and a flag array of one flag per element (see Table 10-2 in Chapter 10). The output values array is ignored. Death is determined by the flag return value. For user-defined subroutines, a flag return value of –1 indicates that the element should die. A flag return value greater than or equal to 0 indicates that the element should remain alive.

Following the `ELEMENT BLOCK SUBROUTINE` command line are other command lines that may

be used to implement the user subroutine option. These command lines are described in Section 10.2.2 and consist of SUBROUTINE DEBUGGING OFF, SUBROUTINE DEBUGGING ON, SUBROUTINE REAL PARAMETER, SUBROUTINE INTEGER PARAMETER, and SUBROUTINE STRING PARAMETER. Examples of using these command lines are provided throughout Section 10.

6.5.3. Evaluation Commands

```
CHECK STEP INTERVAL = <integer>num_steps
CHECK TIME INTERVAL = <real>delta_t
DEATH START TIME = <real>time
```

Evaluation of element death criteria may be time consuming. Additionally, reconstruction of contact or other boundary conditions after element death can be time consuming. For these reasons, three command lines are available for determining the frequency at which element death is evaluated. The default is to evaluate element death at every time step. The number of times at which element death evaluation is done may be limited using the following commands.

- The CHECK STEP INTERVAL command line instructs element death to evaluate the element death criteria only every num_steps time steps.
- The CHECK TIME INTERVAL command line instructs element death to evaluate the element death criteria only every delta_t time units.
- The DEATH START TIME command line instructs element death not to evaluate death criteria before a user-specified time, as given by the real value time.

Both the CHECK STEP INTERVAL and CHECK TIME INTERVAL command lines may be used in a command block. Evaluations for element death will be made at both the time and step intervals if both of these command lines are included.

All three of CHECK STEP INTERVAL, CHECK TIME INTERVAL, and DEATH START TIME are optional command lines.

6.5.4. Miscellaneous Option Commands

The command lines listed below need not be present in the ELEMENT DEATH command block unless the conditions addressed by each call for their inclusion.

6.5.4.1. Summary Output Commands

At the end of a run, a summary of all element death blocks is output to the log file. The SUMMARY OUTPUT STEP INTERVAL or SUMMARY OUTPUT STEP INTERVAL commands can be used to request that the summary be output to the log file at regular intervals during the run. The SUMMARY OUTPUT TIME INTERVAL command results in the summary being printed every output_step_interval steps, while the SUMMARY OUTPUT TIME INTERVAL results in the summary being printed once every output_time_interval time units. These two commands

can be supplied individually, together or not at all. If neither are used, the summary is printed only at the end of execution.

This command applies to all element death blocks. If these commands appear in multiple element death blocks, the values specified in the last instance of each of these commands prevails.



6.5.4.2. Death on Inversion

```
DEATH ON INVERSION = ON|OFF(OFF)
```

If the `DEATH ON INVERSION` command line is set to `ON`, any element that inverts will be killed. This command currently only works for uniform gradient eight-noded hexahedral elements, four-, eight-, and ten-noded tetrahedral elements, and four-noded shells.

6.5.4.3. Death on Ill-defined Contact

```
DEATH ON ILL DEFINED CONTACT = ON|OFF(OFF)
```

If the `DEATH ON ILL DEFINED CONTACT` command line is set to `ON`, any element that would create invalid geometry for contact will be killed. This currently only works for uniform gradient eight-noded hexahedral elements and four-noded tetrahedra elements. An example of an element that creates an invalid contact geometry is an element that is partially inverted in such a way that some regions of the element have negative volume and exterior faces of the element point in the wrong directions.

A highly deformed element can have a positive total volume, permitting internal force to still be calculated, but have inverted local regions that make it invalid for contact. For this reason, killing elements due to ill-defined contact can be important to obtain solutions for contact problems in which elements are experiencing large distortions. Using death on inversion alone is not sufficient for those cases.



6.5.4.4. Death Steps

```
DEATH STEPS = <integer>death_steps(1)
```

If the `DEATH STEPS` command line is used and the value for `death_steps` is set to a value greater than 1, all components of the stress and the hourglass forces in a killed element are linearly scaled down until they reach 0 over the specified number of steps. Gradually releasing the energy due to killing an element over a number of steps can avoid instabilities that might occur if that element were suddenly killed. If this is done over an excessively large number of steps, however, the killed element may participate in the analysis long after it is relevant, and may cause problems if it undergoes excessive deformation or inverts while its stresses are being tapered off. The default number of steps, as provided in the integer value `death_steps`, is 1.

The value selected for `death_steps` will depend on the loads, material models, and model complexity associated with an analysis. A small number (3 to 5) may be sufficient to prevent instabilities for most cases. However, it may be necessary to use a value of 10 or larger.

If an element death criterion is used in an element with multiple integration points, the stresses for the integration points that have exceeded the criterion are decayed individually over the specified number of death steps. When the required number of death steps for an integration point has been exceeded, that integration point no longer contributes to the element internal force, and is considered dead. When the required number of integration points to kill the entire element (based on the user-input `KILL WHEN num_intg INTEGRATION POINTS REMAIN` – See Section 6.5.2.2) have reached that state, the process of killing the element is initiated, and the stresses in the remaining integration points and the hourglass forces are all scaled down over the specified number of death steps.

If an element reaches a death criteria and while in the process of fading out reaches another death criteria whichever death criteria has the lowest number of remaining steps will be used. For example if a material has a death criteria on stress and fades out of 10 death steps and in the process of that fade out has a bad shape and is killed in a single death step the single death step inversion criteria will overwrite the stress fade out criteria.

See Section 8.4.6 for more information on how contact interacts with element death.

6.5.4.5. Aggressive Contact Cleanup

```
AGGRESSIVE CONTACT CLEANUP = <string>OFF|ON(OFF)
```

The `AGGRESSIVE CONTACT CLEANUP` command line enables the use of element death in an attempt to guarantee a valid contact mesh. Certain complex meshes with extensive element death can be problematic for the contact algorithms on some mesh decompositions. This command allows element death to query the contact algorithm and then remove any elements that the contact algorithm flags as having the potential to cause contact to fail.

6.5.4.6. Death Method

```
DEATH METHOD = <string>DEACTIVATE ELEMENT|DEACTIVATE NODAL MPCS|
  DISCONNECT ELEMENT|(DEACTIVATE ELEMENT)
```

The `DEATH METHOD` command specifies what happens when an element meets the death criterion. The following strings can be used as arguments to this command: `DEACTIVATE ELEMENT`, `DEACTIVATE NODAL MPCS`, `DISCONNECT ELEMENT`. The behavior controlled by these options is described below.

- With the default option, `DEACTIVATE ELEMENT`, the element is deactivated, effectively removing it from the mesh.
- The `DEACTIVATE NODAL MPCS` option removes the nodes of a killed element from any multi-point constraints. This only has an effect if the element has nodes that are in multi-point constraints. The multi-point constraint deactivation option can be used to break an element away from the mesh, allowing it to move independently. It can also be used to activate cohesive zones.

- The `DISCONNECT ELEMENT` option disconnects the element from the mesh, allowing the element to move independently. The disconnected element will no longer share any nodes with neighboring elements, and will only interact with the remainder of the mesh through contact.

When using element disconnection, it is necessary to use at least some hourglass stiffness and/or viscosity to prevent large deformations of the disconnected elements in zero energy modes. In addition, it is recommended that a secondary shape-based criterion be used in a separate element death command block to fully remove the disconnected elements if they begin to deform too much. The following example of an additional element death block to be used together with element disconnection shows recommended shape criteria:

```
begin element death
  include all blocks
  #Kill an element if it has negative volume
  death on inversion = on
  #Kill an element if it has become locally inverted (i.e., concave)
  criterion is element value of nodal_jacobian_ratio <= 0.0
  #Kill an element that is becoming very large (shells only)
  criterion is element value of perimeter_ratio > 10.0
end
```

6.5.4.7. Overlap Removal After Death

The `REMOVE OVERLAP AFTER DEATH` option removes overlap after element death when using the element disconnection capability. After each time step where elements are disconnected, the overlap removal algorithm runs to help prevent spurious energy from being introduced. This option has no effect for particle conversion.

6.5.4.8. Logfile Output

```
LOGFILE OUTPUT = FULL|SOME|MINIMAL(SOME)
```

The `LOGFILE OUTPUT` command controls the verbosity of log output from element death.

- The `FULL` option is the most verbose and detailed setting, which triggers printing of each and every element which is killed in a given time step, including its global identifier, containing block name, and criterion which caused its death. For example,

```
Block: block_1   Killed Element: 901   Death Criterion: \
    Death Block: 'dx' : Avg Nodal Value Of: coordinates(x)
Block: block_1   Killed Element: 902   Death Criterion: \
    Death Block: 'dx' : Avg Nodal Value Of: coordinates(x)
Block: block_1   Killed Element: 903   Death Criterion: \
    Death Block: 'dx' : Avg Nodal Value Of: coordinates(x)
Block: block_1   Killed Element: 904   Death Criterion: \
    Death Block: 'dx' : Avg Nodal Value Of: coordinates(x)
```

```
Block: block_1   Killed Element: 905   Death Criterion: \
    Death Block: 'dx' : Avg Nodal Value Of: coordinates(x)
```

- The SOME option is the default setting, which triggers printing of only a summary of the number of elements killed by each criterion in a given timestep. For example,

```
At time: 2.3e-06 100 elements died due to Death Block: 'dx' : \
    Avg Nodal Value Of: coordinates(x)
```

- The MINIMAL option is the least verbose setting, which prints only a global total of the number of elements killed in a given time step, e.g.,

```
***  Killed 200 Elements on time step 22 for a total of 200 dead.
```



6.5.4.9. Death on Proximity

```
BEGIN DEATH PROXIMITY
  BLOCK = <string list>block_names
  NODE SET|NODESET = <string list>nodelist_names
  SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  TOLERANCE = <real>search_tolerance
END [DEATH PROXIMITY]
```

Sierra/SM provides the option to kill elements that get within a search tolerance of nodes in a specified mesh entity set. Proximity is measured by the smallest distance from the nodes and element centroids of the elements specified in the BEGIN ELEMENT DEATH command block to the nodes in the mesh entity set defined in the command block BEGIN DEATH PROXIMITY.

The commands BLOCK, NODE SET|NODESET, SURFACE, REMOVE BLOCK, REMOVE NODE SET, and REMOVE SURFACE inside the command block BEGIN DEATH PROXIMITY form the mesh entity set of nodes used to search against for the proximity measure.

The command TOLERANCE specifies the distance tolerance. Once an element's proximity measure is less than or equal to the distance tolerance the element will be killed.



6.5.4.10. Particle Conversion

```
BEGIN PARTICLE CONVERSION
  PARTICLE SECTION = <string>section_name
  NEW MATERIAL = <string>material_name
    MODEL NAME = <string>model_name
  TRANSFER = <string>NONE|ALL(ALL)
  MAX NUM PARTICLES = <integer>num(1)
  MIN PARTICLE CONVERSION VOLUME = <real>vol
END [PARTICLE CONVERSION]
```

Sierra/SM provides the option to convert elements to particles when they are killed, rather than removing them from the mesh. This option is activated by using the

BEGIN PARTICLE CONVERSION command block inside the ELEMENT DEATH command block. Particle conversion can be useful when elements become too highly deformed to be represented by solid elements. This allows for the highly deformed solid elements to be removed from the mesh, while retaining the mass associated with those elements in the form of particles.

The name of a particle section block (defined with a PARTICLE SECTION command block, see Section 6.2.13) must be provided with the PARTICLE SECTION command line to define the properties of the created particles. Particles may be created that use the SPH or mass particle formulation, as defined by the section.

The material models for the created particles may be optionally prescribed by providing the name and model for a defined material in the NEW MATERIAL command. By default, the particle uses the material model of the solid element from which it has been converted, and all element data that can be retained, including material state variables, are transferred during conversion. This behavior may be modified by specifying the transfer type through the TRANSFER command. If all the applicable element data should be transferred, the ALL option should be specified (this is the default if no transfer type is specified). If the particles should be created with reinitialized state data, the NONE option should be specified. Moreover, if a new material type is specified for conversion, the TRANSFER option will apply, but likely much of the material state data will not be transferable and will be lost as the element is converted to a particle. The best option when using a new material is to also specify a transfer type of NONE.



Warning: It is important that material state data be correctly initialized on SPH particles. It is possible to generate incompatible material states with the command TRANSFER=NONE, or by changing the material type. An incorrect state initialization during conversion may result in errors or in the worst cases, hanging of the analysis.

Particle conversion works with most element geometries including hexahedra, tetrahedra, shells, beams, and other particles. The converted particle is placed at the centroid of the removed element. For shells, beams and particles the new particle is placed inside of the lofted volume that the structural element represents. The mass of the particle equals the mass of the converted element. The velocity and/or temperature of the converted particle is set to the average velocity/temperature of the source element's nodes. The radius of the converted particle is computed to allow the particle to fit inside the void space left by removal of the solid element. If no particle can fit in that space, no particle is created. This typically occurs when an element has inverted or the center of mass is not on the interior of the element, as seen in Figure 6-28.

Particle conversion is supported primarily for single integration point elements, with limited support for the 8 integration point selective deviatoric and total Lagrange hexahedral element types. In this case, particle state is transferred by copying the values contained in the first integration point of the parent element.

A single element can optionally be converted into multiple particles. The number of particles to convert to is specified with the MAX NUM PARTICLES command and MIN PARTICLE CONVERSION VOLUME command. The usage of these commands is as follows:

If neither of these command lines is present, one particle is placed at the centroid of the

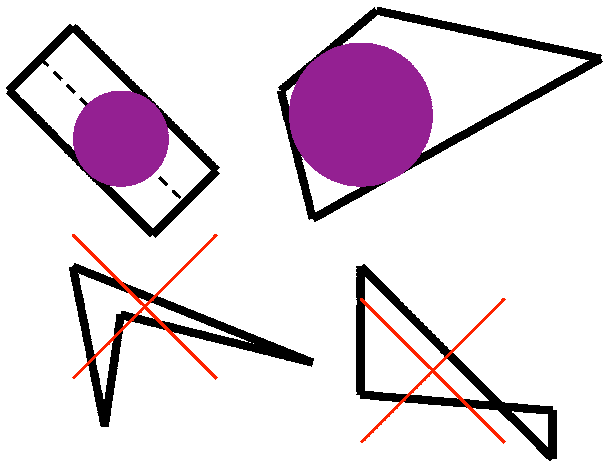


Figure 6-28. Examples of how an element is converted and when it is not converted.

element.

If only the `MAX NUM PARTICLES` command line is present, the element will be cut into the specified number of roughly equal volume sections. A particle will then be placed at the centroid of each of these sections.

If only the `MIN PARTICLE CONVERSION VOLUME` command line is present, the element will be subdivided until a number of volumes each of roughly the specified volume is reached. A new particle will then be placed at the centroid of each of these sub-volumes.

If both the `MAX NUM PARTICLES` and `MIN PARTICLE CONVERSION VOLUME` commands are specified, the element will be subdivided until either the minimum volume of the sub-parts is reached, or the maximum number of sub-parts is reached. When one of these criteria is reached, a new particle is placed into each created sub-volume.



Warning: When performing element to particle conversion, sometimes it is beneficial to deactivate particles created from element death later on in the analysis. To accomplish this use case, all death blocks involving particle conversion should occur before death blocks deactivating elements. This will allow the particle block to be created at the time the death block is parsed, and thus, it will exist when the "deactivate element" death block is parsed.

6.5.4.11. Active Periods

The following command lines can optionally be used in the `ELEMENT DEATH` command block:

```
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

These command lines determine when element death is active. See Section 2.6 for more information about these optional command lines.

6.5.5. Cohesive Zone Setup Commands

The commands listed here are all related to activation of cohesive zones by element death.

```
COHESIVE ZONE INITIALIZATION METHOD = <string>NONE|
  ELEMENT STRESS AVG(NONE)
```

The COHESIVE ZONE INITIALIZATION METHOD command controls the initialization of cohesive zones that are activated through element death. This command should only be used if there are cohesive zones between elements and all nodes of those elements are initially attached together via multi-point constraints. When element death is used to deactivate the constraints (with the DEATH METHOD = DEACTIVATE NODAL MPCS), the exposed cohesive zone can be given an initial state. The options are to either do nothing (NONE) or to initialize the tractions in the cohesive element based on the stresses in the two elements on either side of the cohesive zone (ELEMENT STRESS AVG). How the cohesive zone uses the initial traction will depend on the cohesive surface material model used.



6.5.6. Example

The following example provides instructions to kill elements in block_1 when they leave a bounding box. This type of element death can be useful in an analysis where some peripheral parts, because of fracture, separate and fly away from a central body, where this central body is the part of interest. In this case, these peripheral parts no longer impact the solution. The instructions in this ELEMENT DEATH command block will cause the parts to be killed, thus speeding up the computation.

```
begin element death out_of_bounds
  block = block_1
  # check x coordinates
  criterion is avg nodal value of coordinates(1) >=  10
  criterion is avg nodal value of coordinates(1) <= -10
  # check y coordinates
  criterion is avg nodal value of coordinates(2) >=  10
  criterion is avg nodal value of coordinates(2) <= -10
  # check z coordinates
  criterion is avg nodal value of coordinates(3) >=  10
  criterion is avg nodal value of coordinates(3) <= -10
end element death out_of_bounds
```

6.5.7. Element Death Visualization

When an element dies, information about this element will still be sent, along with information for all other elements, to the Exodus II results file. (Chapter 9 describes the output of element variables to the results file.) The death status of the elements may be output to the results file by requesting element variable output for the element variable DEATH_STATUS. Including the command line

```
ELEMENT DEATH_STATUS as death_var
```

in a RESULTS OUTPUT command block (Chapter 9) will output this element variable with the name death_var in the results file. The convention for DEATH_STATUS is as follows: An element with a value of 1.0 for DEATH_STATUS is a living element. An element with a value of less than or equal to 0.0 for DEATH_STATUS is a dead element.

In the case other than MATERIAL CRITERION, a DEATH_STATUS value between 1.0 and 0.0 indicates an element is in the process of dying. A dying element has its material stress scaled down over a number of time steps. The current scaling factor for an element is given by DEATH_STATUS. Whether an element can have a value other than 0.0 or 1.0 for DEATH_STATUS will depend on whether the DEATH STEPS option has been used in the ELEMENT DEATH command block. If the number of steps over which death occurs is greater than 1, then DEATH_STATUS can be some value between 0.0 and 1.0.

If DEATH_STATUS is written to a results file, it is possible to use DEATH_STATUS to exclude killed elements from any view of the model in visualization software. A subset of the mesh showing only living elements can be created by visualizing only those elements for which DEATH_STATUS > 0.0. The procedure for filtering results in this way varies for different post-processing tools.

When an analysis is using element death, the log file contains a table of marker values that will be applied to dead elements. The marker values allow determining which elements where killed and by which criterion. The marker variables are stored in the element variable KILLED_BY_CRITERION and are available for output on the mesh results file. Additionally, a global count of how many elements were killed by each criterion is printed at the end of the run log file.



6.6. Particle Embedding

```
BEGIN PARTICLE EMBEDDING <string>particle_embedding_name
  BLOCK = <string>block_name
  INCLUDE ALL BLOCKS
  REMOVE BLOCK = <string>block_name
  PARTICLE SECTION = <string>section_name
  NEW MATERIAL = <string>material_name
    MODEL NAME = <string>model_name
  TRANSFER = <string>NONE|ALL(ALL)
  MAX NUM PARTICLES = <integer>num(1)
  MIN PARTICLE CONVERSION VOLUME = <real>vol
END [PARTICLE EMBEDDING]
```

Particle embedding allows the user to specify element blocks in which each element is embedded with particles during initialization, before the first time step is taken. This capability can remove the need to create particles during element death and may be used for particle-solid interactions.

The embedded particles are created in the same way as during particle conversion for element death, except that the original solid element remains alive after the particles are created. Particles

that are created are constrained to move with a material point in their controlling element (i.e., the element in which they were created). This constraint is released only if the controlling element dies. In addition, these embedded particles may interact with non-embedded particles to produce internal forces on the embedded particles. These internal forces are transferred to the controlling element using the embedded particle's location in the element. Note that the mass of the particles embedded in an element is assigned as in particle conversion for element death but that the particles' mass does not affect the solution until the controlling element dies. Once a particle's controlling element dies it is freed and behaves as a particle element.

The commands for this block are discussed in Section 6.5.4.10 and work for particle embedding in the same way, save `TRANSFER`. Particle embedding has two opportunities for transferring element data: at initialization and at controlling element death. For particle embedding the `TRANSFER` command applies to both of these transfers. If the `ALL` option is used, an embedded particle will be initialized with the initial element data of its controlling element and will have its element data reset to the evolved element data of its controlling element when its controlling element dies. If the `NONE` option is used, the embedded particle's element data will be initialized as a non-embedded particle's would be, and the particle's element data will not be modified at controlling element death.

6.7. Explicitly Computing Derived Quantities

```
BEGIN DERIVED OUTPUT
  COMPUTE AND STORE VARIABLE =
    <string>derived_quantity_name
END DERIVED OUTPUT
```

The above command block is used to explicitly compute and store a derived quantity into an internal field. This is useful if the field is needed by an outside capability such as a transfer or error estimation.

For example, to use a derived quantity in a transfer, a `DERIVED OUTPUT` command block must be used. For instance, to transfer the von Mises stress norm:

```
BEGIN DERIVED OUTPUT
  COMPUTE AND STORE VARIABLE = von_mises
END DERIVED OUTPUT
```

Tables 9-12 through 9-15 in Section 9 list element variables available for different types of elements. Variables that are only computed at user request are designated with a `yes` in the `Derived` column of these tables. These are the quantities that must be listed in a `BEGIN DERIVED OUTPUT` command block if they are to be transferred during a coupled analysis.



6.8. Mesh Rebalancing

Mesh rebalancing is a feature in Presto that may improve the efficiency of an analysis. Two command blocks can be used to control mesh rebalancing: `REBALANCE` and `ZOLTAN PARAMETERS`. The `REBALANCE` command block is required; the `ZOLTAN PARAMETERS` command block is optional. Sections 6.8.1 and 6.8.2 describe these command blocks.

6.8.1. Rebalance

```
BEGIN REBALANCE
  ELEMENT GROUPING TYPE = SPLIT SPH AND STANDARD ELEMENTS|
    UNIFORM UNIFIED| SPLIT SPH AND DEM AND STANDARD ELEMENTS|
    SPLIT DEM AND STANDARD ELEMENTS |
    SPLIT PARTICLE AND STANDARD ELEMENTS
    (SPLIT SPH AND STANDARD ELEMENTS)
  REBALANCE TYPE = AUTO_PERIODIC_ONLY|INITIAL_AND_AUTO_PERIODIC|
                   INITIAL_AND_PERIODIC|INITIAL_ONLY|PERIODIC_ONLY
  REBALANCE STRATEGY = SIMPLE|MULTICRITERIA(SIMPLE)
  REBALANCE STEP INTERVAL = <integer>step_interval(infinity)
  LOAD RATIO THRESHOLD = <real>load_ratio(1.05)
  COMMUNICATION RATIO THRESHOLD = <real>communication_ratio(1.15)
  ZOLTAN PARAMETERS = <string>parameter_nam
```

```
  DELETE DEACTIVATED ELEMENTS AFTER REBALANCE = ON|OFF(OFF)
  ACTIVE PERIODS = <string list>active_periods
  INACTIVE PERIODS = <string list>inactive_periods
END [REBALANCE]
```

Initial decomposition of a mesh for parallel runs with Sierra/SM is done by a utility called `decomp`. The initial decomposition provided by `decomp` may become sub-optimal as the simulation evolves, especially for cases involving large deformations. Therefore, Presto supports a simple mesh-rebalancing capability that can be used to improve the parallel performance of some problems. When mesh rebalancing is invoked, the parallel decomposition is changed, and elements are moved among the processors to balance the computational load and minimize the processor-to-processor communication. Mesh rebalancing may be useful in the following circumstances:

- The mesh decomposition produced by `decomp` for sphere elements (i.e., SPH) can be poor. It is recommended that an initial mesh rebalance be done for all problems involving sphere elements.
- If a problem experiences very large deformations, periodic rebalancing may be helpful. In contact or SPH problems, communication is performed between physically nearby contact surfaces or SPH particles. To maintain optimum performance, it is helpful to have neighboring particles located on the same processors. Periodic mesh rebalancing can ensure that neighboring entities tend to remain on the same processor during large mesh deformations.

The `REBALANCE` command block is placed in the Presto region scope. The mesh rebalancing in Presto uses a mesh balancing library called Zoltan (Reference [17]). Zoltan performs the actual rebalancing. By default, Presto creates a Zoltan object with a default set of parameters. However, a Zoltan object with a customized set of parameters can be created and referenced from the `REBALANCE` command block.

6.8.1.1. Rebalance Command Lines

```
ELEMENT GROUPING TYPE = SPLIT SPH AND STANDARD ELEMENTS|
  UNIFORM UNIFIED| SPLIT SPH AND DEM AND STANDARD ELEMENTS|
  SPLIT DEM AND STANDARD ELEMENTS |
  SPLIT PARTICLE AND STANDARD ELEMENTS
  (SPLIT SPH AND STANDARD ELEMENTS)
REBALANCE TYPE = AUTO_PERIODIC_ONLY|INITIAL_AND_AUTO_PERIODIC|
                 INITIAL_AND_PERIODIC|INITIAL_ONLY|PERIODIC_ONLY
REBALANCE STEP INTERVAL = <integer>step_interval(infinity)
LOAD RATIO THRESHOLD = <real>load_ratio(1.05)
COMMUNICATION RATIO THRESHOLD = <real>ratio(1.15)
```

The above command lines control how and when the rebalancing is done.

The `ELEMENT GROUPING TYPE` command line specifies how the elements are grouped for rebalancing. If set to `SPLIT SPH AND STANDARD ELEMENTS`, which is the default behavior,

this command causes SPH and standard elements to be split into separate groups. If it is set to the UNIFORM UNIFIED option, all elements can be grouped together. This command also has options that relate to DEM an experimental element formulation. These DEM options are not recommended for general use.

The REBALANCE TYPE command line is used to specify the type of rebalance that is performed. If the INITIAL_ONLY option is used, the mesh will be rebalanced at time zero before any calculations occur. This option should be used if the initial mesh decomposition passed to Presto is poor. If the PERIODIC_ONLY option is used, the mesh will be rebalanced every step_interval steps, where step_interval is the parameter specified by the REBALANCE STEP INTERVAL command line. If the AUTO_PERIODIC_ONLY option is used, the mesh will be rebalanced every step_interval steps, when the communication ratio reaches a critical value, or when the load ratio reaches a critical value. All other values of REBALANCE TYPE are a combination of the above three options.

The REBALANCE STRATEGY command line specifies which algorithm to use to perform the rebalance. SIMPLE is the legacy approach which simply distributes the elements or load evenly across all of the processors used. The MULTICRITERIA option dynamically manages different physical models and their relative cost. For example, the MULTICRITERIA option accounts for contact boundaries and the relative cost of different element topologies (such as shell and solid elements). Note that not all element types have been assigned weights for multicriteria rebalance.

The communication ratio, currently defined only for SPH problems, is a measure of how much communication is required in the current mesh decomposition versus an estimate of the amount of communication with an optimal decomposition. Mesh rebalancing is expensive, so rebalancing should be done rarely. The COMMUNICATION RATIO THRESHOLD command line specifies the value of the communication ratio that triggers a rebalance. Setting this to a value between 1.25 and 1.5 is usually optimal.

The load ratio is the ratio of the current load (maximum number of elements on a processor divided by the average number of elements per processor). The LOAD RATIO THRESHOLD command specifies that a rebalance should occur when the load ratio exceeds the specified threshold. A specified value of 1.5 would trigger a rebalance when the load ratio is 50% higher than could be optimally obtained.

The DELETE DEACTIVATED ELEMENTS AFTER REBALANCE command line can be used to remove any inactive elements before performing the mesh rebalance. If a large number of elements have been deactivated due to element death, or due to use of time step sub-cycling, this may lead to a better load balance. With this option rebalance will get an optimal decomposition considering only those elements that are currently alive and active and remove any other elements permanently from the data structure.

The ACTIVE PERIODS AND INACTIVE PERIODS commands can turn rebalancing on or off by time period. See Section 2.6 for more information regarding these commands.

6.8.1.2. Zoltan Command Line

The command line

```
ZOLTAN PARAMETERS = <string>parameter_name
```

references a `ZOLTAN PARAMETERS` command block named `parameter_name`. Various parameters for Zoltan can be set in the `ZOLTAN PARAMETERS` command block. If the `ZOLTAN PARAMETERS` command line is not provided, a default set of parameters is used. The default parameter command block is shown as follows:

```
BEGIN ZOLTAN PARAMETERS
  LOAD BALANCING METHOD = recursive coordinate bisection
    # string parameter
  OVER ALLOCATE MEMORY  = 1.5 # real parameter
  REUSE CUTS = true # string parameter
  ALGORITHM DEBUG LEVEL = 0 # integer parameter
  CHECK GEOMETRY = true # string parameter
  ZOLTAN DEBUG LEVEL = 0 # integer parameter
END ZOLTAN PARAMETERS
```

See Section 6.8.2 for a discussion of the `ZOLTAN PARAMETERS` command block. Section 6.8.2 lists the command lines that can be used to set Zoltan parameters.

6.8.2. Zoltan Parameters

```
BEGIN ZOLTAN PARAMETERS
  LOAD BALANCING METHOD  = <string>recursive coordinate bisection|
    recursive inertial bisection|Hilbert space filling curve
  DETERMINISTIC DECOMPOSITION = <string>false|true
  IMBALANCE TOLERANCE = <real>imb_tol
  OVER ALLOCATE MEMORY = <real>over_all_mem
  REUSE CUTS = <string>false|true
  ALGORITHM DEBUG LEVEL = <integer>alg_level
    #  0<=(alg_level)<=3
  CHECK GEOMETRY = <string>false|true
  KEEP CUTS = <string>false|true
  LOCK RCB DIRECTIONS = <string>false|true
  SET RCB DIRECTIONS = <string>do not order cuts|xyz|xzy|
    yzx|yxz|zxy|zyx
  RECTILINEAR RCB BLOCKS = <string>false|true
  RENUMBER PARTITIONS = <string>false|true
  ZOLTAN DEBUG LEVEL = <integer>zoltan_level
    #  0<=(zoltan_level)<=10
  DEBUG PROCESSOR NUMBER = <integer>proc  #  1<=proc
  TIMER = <string> wall|cpu
  DEBUG MEMORY = <integer>dbg_mem  #  0<=(dbg_mem)<=3
```

```
END [ZOLTAN PARAMETERS]
```

The ZOLTAN PARAMETERS command block is used to set parameters for Zoltan (see Reference [17]), a program that can be used for mesh rebalancing in Presto. The ZOLTAN PARAMETERS command block is used in association with the REBALANCE command block. A REBALANCE command block may reference a ZOLTAN PARAMETERS command block via the name, parameter_name, for the parameter command block. Reference Section 6.8.1 regarding the use of the ZOLTAN PARAMETERS command block for mesh rebalancing in Presto.

Setting the parameters for Zoltan involves some understanding of how Zoltan works. Consult Reference [17] for a discussion of the parameters that can be set by the various command lines in the ZOLTAN PARAMETERS command block. Note that some of the command lines in this command block have comments that provide additional information about the parameters. The "#" symbol precedes a comment.

In the above command block, = and IS are the allowed delimiters, which is different from the usual Presto convention of =, IS, and ARE. Note that the ZOLTAN PARAMETERS command block should be specified in the SIERRA scope when it is referenced from the ZOLTAN PARAMETERS command line in the REBALANCE command block. When the default set of parameters is used for a Zoltan object, the ZOLTAN PARAMETERS command block need not be included in the input file.

References

[1] L.M. Taylor and D.P. Flanagan. Pronto3D: A three-dimensional transient solid dynamics program. Technical Report SAND87-1912, Sandia National Laboratories, Albuquerque, NM, March 1989. pdf.

[2] M.M. Rashid. Incremental kinematics for finite element applications. *International Journal for Numerical Methods in Engineering*, 36:3937–3956, 1993. doi.

[3] C.R. Dohrman, M.W. Heinstein, J. Jung, S.W. Key, and W.R. Witkowski. Node-based uniform strain elements for three-node triangular and four-node tetrahedral meshes. *International Journal for Numerical Methods in Engineering*, 47:1549–1568, 2000. doi.

[4] S.W. Key and C.C. Hoff. An improved constant membrane and bending stress shell element for explicit transient dynamics. *Computer Methods in Applied Mechanics and Engineering*, 124(1–2):33–47, 1995. doi.

[5] T.A. Laursen, S.W. Attaway, and R.I. Zadoks. SEACAS theory manuals: Part III. finite element analysis in nonlinear solid mechanics. Technical Report SAND98-1760/3, Sandia National Laboratories, Albuquerque, NM, 1999. pdf.

[6] D.P. Flanagan and T. Belytschko. A uniform strain hexahedron and quadrilateral with orthogonal hourglass control. *International Journal for Numerical Methods in Engineering*, 17:679–706, 1981. doi.

[7] J.W. Swegle. SIERRA: PRESTO theory documentation: Energy dependent materials version 1.0. Technical Report SAND2009-3801P, Sandia National Laboratories, Albuquerque, NM, October 2001.

[8] J.T. Ostien, J.W. Foulk, A. Mota, and M.G. Veilleux. A 10-node composite tetrahedral finite element for solid mechanics. *International Journal for Numerical Methods in Engineering*, 2016.

[9] R. L. Taylor and E. Artioli. *VEM for Inelastic Solids*, pages 381–394. Springer International Publishing, Cham, 2018. ISBN 978-3-319-60885-3. doi: 10.1007/978-3-319-60885-3_18. URL https://doi.org/10.1007/978-3-319-60885-3_18.

[10] Q. Yang, A. Mota, and M. Ortiz. A class of variational strain-localization finite elements. *International Journal for Numerical Methods in Engineering*, 62:1013–1037, 2005. doi.

[11] T. Belytschko and I. Leviathan. Physical stabilization of the 4-node shell element with one point quadrature. *Computer Methods in Applied Mechanics and Engineering*, 113:321–350, 1994.

[12] T. Belytschko and I. Leviathan. Projection schemes for one-point quadrature shell elements. *Computer Methods in Applied Mechanics and Engineering*, 115:277–286, 1994.

[13] P. Madabhusi-Raman and J.F. Davalos. Static shear correction factor for laminated rectangular beams. *Composites Part B: Engineering*, 27(3):285–293, 1996. doi.

[14] G.D. Sjaardema. Overview of the Sandia National Laboratories engineering analysis code access system. Technical Report SAND92-2292, Sandia National Laboratories, Albuquerque, NM, January 1993. pdf.

[15] J.W. Swegle, S.W. Attaway, M.W. Heinstein, F.J. Mello, and D.L. Hicks. An analysis of smoothed particle hydrodynamics. Technical Report SAND93-2513, Sandia National Laboratories, Albuquerque, NM, March 1994. pdf.

[16] J. Monaghan. SPH without a tensile instability. *Journal of Computational Physics*, 159(2): 290–311, 2000. doi.

[17] K. Devine, E. Boman, R. Heapby, B. Hendrickson, and C. Vaughan. Zoltan data management service for parallel dynamic applications. *Computing in Science and Engineering*, 4:90–97, 2002. doi. See also the ZOLTAN web site. link.

7. BOUNDARY CONDITIONS AND INITIAL CONDITIONS

Sierra/SM offers a variety of options for defining boundary and initial conditions. Typically, boundary and initial conditions are defined on some subset of mesh entities (node, element face, element) defining a model. Sierra/SM offers a flexible means to define subsets of mesh entities. Section 7.1.1 describes commands that will let the user define some subset of a mesh entity using a collection of commands that constitute a set of Boolean operators.

The remaining parts of this chapter discuss the following functionality:

- Section 7.3 presents methods for setting the initial values of variables in Sierra/SM. Sierra/SM has the flexibility to set a complex initial state for some variable such as nodal velocity or element stress.
- Kinematic boundary conditions typical of a solid mechanics code (fixed displacement, prescribed acceleration, etc.) are options in Sierra/SM and described in Section 7.4. Most of these boundary conditions allow the specification of a time history using a function, a user subroutine, or by reading values from a mesh file.
- Section 7.5 documents many initial velocity options available in Sierra/SM.
- Force boundary conditions typical of a solid mechanics code (prescribed force, traction, etc.) are options in Sierra/SM and described in Section 7.6. Most of these force boundary conditions allow the specification of a time history using a function or a user subroutine.
- Section 7.8 discusses the gravity load option. A gravity load is a body force boundary condition.
- Section 7.9 provides a description of centripetal force boundary conditions can be applied about a prescribed axis.
- Section 7.11 details many options available for describing a temperature field in Sierra/SM.
- Section 7.12 details the options available for describing a pore pressure field.
- Section 7.13 documents how fluid pressure boundary conditions can be applied as a hydrostatic pressure to each node pertaining to a set of prescribed surfaces.
- Section 7.14 documents how the inertia relief boundary condition can be applied to counterbalance the other force boundary conditions on the specified blocks, surfaces, and/or nodesets.
- Section 7.15 describes many specialized boundary conditions.

7.1. General Boundary Condition Concepts

There are general principles that apply to all the available types of boundary conditions. To apply a boundary condition, a set of mesh entities and the magnitude and/or direction in which it is to be applied must be specified. Sierra/SM provides several methods for both specifying the set of mesh entities and for prescribing how the boundary condition is to be applied. The general concepts on how this is done are applicable to all the boundary condition types, and are described in the following sections.

7.1.1. Mesh-Entity Assignment Commands

A number of standard command lines exist to define a set of mesh entities (node, element face, element) associated with some type of boundary, initial, or load condition. All these command lines exist within the command blocks for the various prescribed conditions, which in turn exist within the region scope. These command lines, taken collectively, constitute a set of Boolean operators for constructing sets of mesh entities.

The first set of command lines we will consider is as follows:

```
NODE SET|NODESET = <string list>nodelist_names
NODE     = <integer list>node_ids

SURFACE|SIDESET|SIDE SET = <string list>surface_names

BLOCK = <string list>block_names
ELEMENT = <integer list>element_ids

RIGID BODY = <string list>rigid_body_names

INCLUDE ALL BLOCKS
```

In the above command lines, the string list `nodelist_names` is used to represent one or more node sets as discussed in Section 1.6. A node set is referenced as `nodelist_id`, where id is some integer. For example, suppose there are three node lists in a model: 10, 23, and 105. To combine all these node lists so that they form one set of nodes for, say, a boundary condition or initial condition, the user would invoke the command line

```
NODE SET = nodelist_10 nodelist_23 nodelist_105
```

This convention applies as well to any surface-related command line that uses the string list `surface_names`, any block-related command line that uses the string list `block_names`, and any rigid body command line that uses the string list `rigid_body_names`.

The `NODE SET|NODESET` command line associates a set of nodes with an initial, boundary, or load condition. A condition may be applied to multiple node sets by putting multiple node set

names on the command line or by repeating the command line multiple times. The NODE SET|NODESET line may not be used to apply kinematic boundary conditions to a rigid body; a BLOCK or RIGID BODY line must be used.

The NODE line adds individual nodes to the boundary by condition by their nodal ID in the input mesh. Multiple nodes may be specified on one line or this command may be specified multiple times. The NODE line may not be used to apply kinematic boundary conditions to a rigid body; a BLOCK or RIGID BODY line must be used.

The ELEMENT line adds individual elements to the boundary by condition by their element ID in the input mesh. Multiple elements may be specified on one line or this command may be specified multiple times. The ELEMENT line may not be used to apply kinematic boundary conditions to a rigid body; a BLOCK or RIGID BODY line must be used.

The SURFACE|SIDESET|SIDE SET command line associates a set of element faces or their attached nodes with a boundary, initial, or load condition. A condition may be applied to multiple surfaces by putting multiple surface names on the command line or by repeating the command line multiple times. For example, suppose we wish to use the fixed displacement kinematic boundary condition. Although this is a nodal boundary condition (the condition is applied to individual nodes), a SURFACE command line can be used to establish the set of nodes. If the command line

```
SURFACE = surface_101
```

appears in a fixed displacement boundary condition, then all the nodes associated with surface 101 will be associated with the boundary condition. The SURFACE line may not be used to apply kinematic boundary conditions to a rigid body; a BLOCK or RIGID BODY line must be used.

The BLOCK command line associates a set of elements and its nodes and faces with a boundary condition. A boundary condition may be applied to multiple blocks by putting multiple block names on the command line or by repeating the command line multiple times.

For example, suppose we wish to use the fixed displacement kinematic boundary condition as in the previous example. If the command line

```
BLOCK = block_50
```

appears in a fixed displacement kinematic boundary condition, then all the nodes associated with block 50 will be associated with the boundary condition.

The RIGID BODY command line associates the boundary condition with a rigid body. The rigid body must have been defined in the input file within a BEGIN RIGID BODY command block. See Section 6.3.1 for an explanation of how to create rigid bodies.

Either a BLOCK or a RIGID BODY command line must be used for kinematic boundary conditions on rigid bodies. If the block_name specified in the BLOCK line has been defined as a rigid body, or if the RIGID BODY is used, the specified kinematic condition will be applied to the rigid body reference node.

The INCLUDE ALL BLOCKS command line associates all blocks and hence all elements and nodes in the model with a boundary, initial, or load condition. The block command lines associated with boundary conditions, initial conditions, and gravity will NOT generate surfaces.

Any combination of the above command lines can be used to create a union of mesh entities. Suppose, for example, that the command lines

```
NODE SET = nodelist_2
SURFACE = surface_3
NODE = 7 8 15
ELEMENT = 31
```

appear in a FIXED DISPLACEMENT command block for a kinematic boundary condition. The set of nodes associated with the boundary condition will be the union of the set of nodes associated with surface 3, the set of nodes associated with node set 2, the nodes with ids 7, 8 and 15, and the nodes of element 31.

When a union of mesh entities is created by using two or more of the above command lines, a node or element face may appear in more than one node set, surface or block. However, the prescribed condition is applied to each node or face only once. For example, node 67 may be a part of nodelist 2 and surface 3 but the boundary condition will only be applied to node 67 once.

The set of mesh entities associated with a boundary, initial, or load condition can be edited (mesh entities can be deleted from the set) by using the following command lines:

```
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

The REMOVE NODE SET command line deletes the nodes in the specified node set from the set of nodes used by the condition.

The REMOVE SURFACE command line deletes a set of element faces and their associated nodes from the set of element faces used by the prescribed condition.

The REMOVE BLOCK command line deletes a set of elements and their associated nodes from the set of elements used by the prescribed condition.

7.1.2. Methods for Specifying Boundary Conditions

There are three main methods which can be used to prescribe most types of boundary conditions available in Sierra/SM.

- The boundary condition can be prescribed using commands in the input file. These commands are categorized as “specification commands” in this document. Depending on the type of the boundary condition, it is necessary to prescribe its direction and/or magnitude. Boundary conditions can be specified this way when a set of mesh entities is to

experience a similar condition with a time variation that can be expressed by a function. One of the following commands specifies the direction of the boundary condition: `COMPONENT`, `DIRECTION`, `CYLINDRICAL AXIS`, or `RADIAL AXIS`. The magnitude is defined using one of `MAGNITUDE`, `FUNCTION` or `ANGULAR VELOCITY`. These commands are used in various combinations depending on the type of the boundary condition. The details of how to use them are provided in the descriptions of the various boundary condition types.

- If the nature of the boundary condition is such its variation in time and space cannot be described easily by the combination of a function and a direction, it may be necessary to use a user-defined subroutine. User subroutines provide a general capability to define how kinematic or force boundary conditions are applied. The use of user-defined subroutines does increase the complexity of defining the model, however. The user must write and debug the subroutine and compile and link it in with Sierra/SM. Because of the added complexity, user subroutines should only be used if the needed capability is not provided by the other methods of prescribing boundary conditions.
- For some types of boundary conditions, the values of the field to be prescribed can be read in from an existing output database. This is often used as a method to transfer results from one analysis code to another. One of the common uses for this capability is to compute temperatures using a thermal code, and then transfer the temperature fields to Sierra/SM to study combined mechanical and thermal effects. This capability can be used either to read in initial values or to read in a series of values that vary over time.

In the following sections describing specific types of boundary conditions, the commands are grouped according to these three categories.

7.2. Initial Mesh Modification

```
BEGIN INITIAL MESH MODIFICATION
 BLOCK = <string list>block_name
 INCLUDE ALL BLOCKS
 REMOVE BLOCK = <string list>block_name
 ROTATE ANGLE <real>angle
    ABOUT ORIGIN <real>Ox <real>Oy <real>Oz
    DIRECTION <real>Dx <real>Dy <real>Dz
  MOVE X <real>Mx  Y <real>My  Y <real>Mz

  ROTATE BLOCK <string list>block_names
    ANGLE <real>angle
    ABOUT ORIGIN <real>Ox <real>Oy <real>Oz
    DIRECTION <real>Dx <real>Dy <real>Dz
  MOVE BLOCK <string list>block_names
    X <real>Mx  Y <real>My  Y <real>Mz
  APPLY AFTER RESTART = YES|NO (YES)
END
```



Warning: The mesh is modified according to the commands specified in this block at each initialization, including after restart. Note that performing this mesh initial modification after restart is unnecessary in most workflows as it is contained in the restart file. Most use cases should use the APPLY AFTER RESTART = NO command.

A handful of transformations can be applied to the initial mesh from within the input deck. These transformations happen before any other operations being performed such as contact overlap removal or evaluating initial conditions. Initial mesh modification will only affect the initial mesh read into the file and ignore anything created during the run.

The `ROTATE` command will rotate the blocks set by INCLUDE ALL BLOCKS or BLOCK. The rotation of these blocks is defined by a rotation axis and angle of rotation about that axis. The origin and direction of the rotation axis are defined by the `ABOUT ORIGIN` and `DIRECTION` commands, respectively. The `ANGLE` command sets the rotation angle, in degrees, by following the right-hand rule.

The `MOVE` command will translate the blocks set by INCLUDE ALL BLOCKS or BLOCK with the specified `X`, `Y`, and `Z` translation vector.

The `ROTATE BLOCK` command will rotate the named set of blocks, `block_names`. The rotation of these blocks is defined by a rotation axis and angle of rotation about that axis. The origin and direction of the rotation axis are defined by the `ABOUT ORIGIN` and `DIRECTION` commands, respectively. The `ANGLE` command sets the rotation angle, in degrees, by following the right-hand rule.

The `MOVE BLOCK` command will translate the named set of blocks with the specified `X`, `Y`, and `Z` translation vector.

The syntax and behavior of the mesh modification commands mimic those in Cubit. The defined transformations will be evaluated in order. Thus moving a block then rotating it about an axis will give a different result from rotating the block first then moving it.

The `APPLY AFTER RESTART` command is used to specify whether Initial Mesh Modification should be applied after restart or not. The current default is `YES` to preserve legacy behavior.

7.3. Initial Variable Assignment

```
BEGIN INITIAL CONDITION
  #
  # mesh-entity set commands
  NODE SET|NODESET = <string list>nodelist_names
  SURFACE = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # variable identification commands
  INITIALIZE VARIABLE NAME = <string>var_name
  VARIABLE TYPE = NODE|EDGE|FACE|ELEMENT|GLOBAL
  #
  # specification command
  MAGNITUDE = <real list>initial_values
  #
  # Weibull probability distribution commands
  WEIBULL SHAPE = <real>shapeVal
  WEIBULL SCALE = <real>scaleVal
  WEIBULL MEDIAN = <real>medianVal
  WEIBULL SEED = <int>seedVal(123456)
  WEIBULL SCALING FIELD TYPE = NODE|EDGE|FACE|ELEMENT|
    GLOBAL(ELEMENT)
  WEIBULL SCALING FIELD NAME = <string>fieldVal(VOLUME)
  WEIBULL SCALING REFERENCE VALUE = <real>value
  WEIBULL SCALING EXPONENT SCALE = <real>scaleVal(1.0)
  #
  # external database commands
  READ VARIABLE = <string>var_name
  COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
    MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
  TIME = <real>time|FIRST|LAST
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name |
    SURFACE SUBROUTINE = <string>subroutine_name |
    ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF|ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
```

```
    = <string>param_value
  #
  # distance function commands
  DISTANCE TYPE = XTYPE|YTYPE|ZTYPE|RTYPE|
    RXTYPE|RYTYPE|RZTYPE
  FUNCTION = <string>function_name
  #
  # additional command
  SCALE FACTOR = <real>scale_factor(1.0)
  #
  # Specialized initialization method
  CALCULATE AS CLOSEST PROJECTION TO SURFACE =
    <string list>closeSurfs

END [INITIAL CONDITION]
```

Sierra/SM supports a general initialization procedure for setting the value of any variable. This procedure can be used to set material state variables, shell thickness, initial stress, etc. The initialization is performed both before and after the element and material model initialization. This allows the elements and material models to compute other initial variables based on variables specified by the user and also ensures that the variables specified by the user are not overwritten by the elements and material models. However, there is minimal checking in Sierra/SM to ensure that the changes made yield a consistent system. There is also no guarantee that the changes will not be overwritten or misinterpreted by some other internal routine depending on what variable is being changed. Thus, caution is advised when using this capability.

The `INITIAL CONDITION` command block, which appears in the region scope, is used to select a method and set values for initializing a variable. The command block specifies the initial value of a global variable or a variable associated with a set of mesh entities (nodes, edges, faces, or elements). The user has three options for setting initial values: with a constant magnitude, with an input mesh variable, or by a user subroutine. Only one of these three options can be specified in the command block.

The command block contains five groups of commands—mesh-entity set, variable identification, magnitude, input mesh variable, and user subroutine. In addition to the command lines in the five groups, there is one additional command line: `SCALE FACTOR`. Following are descriptions of the different command groups and the `SCALE FACTOR` command line.

7.3.1. Mesh-Entity Set Commands

The `mesh-entity set commands` portion of the `INITIAL CONDITION` command block specifies the nodes, element faces, or elements associated with the variable to be initialized. This portion of the command block can include some combination of the following command lines:

```
NODE SET|NODESET = <string list>nodelist_names
SURFACE = <string list>surface_names
```

```
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes, element faces, or elements. See Section 7.1.1 for more information about the use of these command lines for mesh entities. There must be at least one `NODE SET|NODESET`, `SURFACE`, `BLOCK`, or `INCLUDE ALL BLOCKS` command line in the command block.

7.3.2. Variable Identification Commands

Any variable used in the `INITIAL CONDITION` command block must exist in Sierra/SM. The variable can be any currently defined variable in Sierra/SM or any user-defined variable created with the `USER VARIABLE` command block (see Section 10.2.4).

There are two command lines that identify the variable:

```
INITIALIZE VARIABLE NAME = <string>var_name
VARIABLE TYPE = [NODE|EDGE|FACE|ELEMENT|GLOBAL]
```

The `INITIALIZE VARIABLE NAME` command line gives the name of the variable for which initial values are being assigned. The variable name must be a name known by Sierra/SM. The variable name may use component syntax to initialize only specific integration points or components of the variable, see Section 9.1 for more information on component syntax.

The `VARIABLE TYPE` command line provides additional information about the variable being initialized. The options `NODE`, `EDGE`, `FACE`, `ELEMENT`, and `GLOBAL` on the command line indicate whether the variable is, respectively, a nodal, edge, face, element, or global quantity. One of these options must appear in the `VARIABLE TYPE` command line. The `GLOBAL` variable type cannot be read from an external mesh database (see Section 7.3.5).

Both of these command lines are required regardless of the option selected to set values for the variable.

7.3.3. Specification Command

If the constant magnitude command is used, one or more initial values are specified directly in the command block. This is done using the following command line:

```
MAGNITUDE = <real list>initial_values
```

The `initial_values` specified on the `MAGNITUDE` command line will set the values for the variable given by `var_name` in the `INITIALIZE VARIABLE NAME` command line. The number

of values is dependent on the type of the variable specified in the INITIALIZE VARIABLE NAME command line. For example, if the user wanted to initialize the velocity at a set of nodes, three quantities would have to be specified since the velocity at a node is a vector quantity. If the user wanted to initialize the stress tensor for a set of uniform-gradient, eight-node hexahedral elements, six quantities would have to be specified since the stress tensor for this element type is described with six values.

7.3.4. Weibull Probability Distribution Commands

The field to be initialized can optionally be initialized or scaled with random numbers generated to conform to a specified Weibull probability distribution function. This is accomplished by including some of the following commands:

```
WEIBULL SHAPE = <real>shapeVal
WEIBULL SCALE = <real>scaleVal
WEIBULL MEDIAN = <real>medianVal
WEIBULL SEED = <int>seedVal (123456)
```

The Weibull commands may be used in conjunction with the MAGNITUDE command or the READ VARIABLE command to set the initial value, m_i, that will be scaled by the distribution. Otherwise m_i starts off equal to 1.0.

Two and only two of the parameters WEIBULL SHAPE, WEIBULL SCALE, and WEIBULL MEDIAN may be specified. The relationship among WEIBULL SHAPE, k, (also known as the Weibull modulus) WEIBULL SCALE, λ, and WEIBULL MEDIAN, μ, is shown in the following equation:

$$\mu = \lambda(ln(2)^{\frac{1}{k}}) \tag{7.1}$$

The WEIBULL SEED parameter sets the starting point for the random number generation. Be careful when initializing two independent fields with Weibull distributions: if the default seed (or the same user specified seed) is used for both field, the random numbers generated will be the same for both fields and the fields will therefore be completely correlated element by element.

The values at each point can be further scaled by another factor, α_i. If no scaling commands are used then $\alpha_i = 1.0$. Scaling is accomplished by using the Weibull scaling commands:

```
WEIBULL SCALING FIELD TYPE = NODE|EDGE|FACE|ELEMENT|
  GLOBAL(ELEMENT)
WEIBULL SCALING FIELD NAME = <string>fieldVal (VOLUME)
WEIBULL SCALING REFERENCE VALUE = <real>value
WEIBULL SCALING EXPONENT SCALE = <real>scaleVal (1.0)
```

The scaling commands are implemented to allow the use of any field variable for scaling; however, the field type must match the type of field being set. The WEIBULL SCALING FIELD NAME gives the name of a variable and the

`WEIBULL SCALING FIELD TYPE` gives the variable field type which can be one of nodal, edge, face, element, or global type. The `WEIBULL SCALING FIELD TYPE` and `WEIBULL SCALING FIELD NAME` default to `ELEMENT` and `VOLUME` respectively. The `WEIBULL SCALING REFERENCE VALUE` gives the reference value v_{ref}. A `WEIBULL SCALING EXPONENT SCALE`, β, may also be defined and will default to 1.0 if not defined. The scaling factor α_i is formed with the following formula:

$$\alpha_i = \left(\frac{v_{ref}}{v_i} \right)^{\frac{\beta}{k}} \tag{7.2}$$

where v_i is the value of a field variable specified with `WEIBULL SCALING FIELD TYPE` and `WEIBULL SCALING FIELD NAME` at the current point i, and the rest of the variables are those defined previously.

The scaled initial value is determined by the equation:

$$InitialValue = m_i \alpha_i \mu \left(\frac{ln(R)}{ln(\frac{1}{2})} \right)^{\frac{1}{k}} \tag{7.3}$$

where R is a random number in an uniform distribution between 0 and 1.

7.3.5. External Mesh Database Commands

```
READ VARIABLE = <string>var_name
COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
  MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
TIME = <real>time|FIRST|LAST
```



Warning: At this time the `COPY VARIABLE` command can only be used with nodal variables. Neither the `COPY VARIABLE` command nor the `READ VARIABLE` command is compatible with global variables.

A number of boundary conditions work by applying some field defined on either the input mesh database or an external mesh database. For example a prescribed temperature may be applied by reading the spatial time history of temperature from a mesh file. The mesh files are defined via the `FINITE ELEMENT MODEL` command block described in Section 6.1.

The `READ VARIABLE = <string>var_name` command reads data from the input mesh for the region. The `read_var_name` string specifies the name of the variable as it appears on the mesh database. Generally the size and types of variables that are read need to be compatible with the types of target fields they are defining. For example, if `read_var_name` refers to a vector variable stored in the region's finite element mesh database and `init_var_name` refers to a vector variable being initialized, then the following command only works if both vectors have the same number of components:

```
INITIALIZE VARIABLE NAME = <string>init_var_name
READ VARIABLE = <string>read_var_name
```

Alternatively, individual components of a variable can be assigned using `<string>var_name(<int>component)`. For example, the following command line works for reading the first component of a vector and assigning it to the first component of an initialized vector:

```
INITIALIZE VARIABLE NAME = <string>init_var_name(1)
READ VARIABLE = <string>read_var_name(1)
```

The variable name used on the mesh file can be arbitrary. The name can be identical to or different from the variable name specified on the `INITIALIZE VARIABLE NAME` command line. For more information on component variable syntax see Section 9.1.

The field to be read may be defined at a number of different time slices on the mesh file. The default behavior is interpolate these values to the current simulation time to extract the value of the field to transfer. The `TIME` command line can optionally be used to select a specific time from which to read the variable from. If necessary, values are interpolated to the specified time. In addition to specifying a time, the keyword `FIRST` or `LAST` may be used to read values corresponding to the first or last time step in the input mesh.

The `READ VARIABLE` command may be used in conjunction with `SCALE FACTOR` or `FUNCTION` commands, in such cases these source terms are multiplied together.

The copy variable function more flexible version of the read variable command. However, the copy variable command only works for nodal variables. The `COPY VARIABLE` command may be used for the following can be used for the following purposes:

- if the source field is defined on the input mesh file, but the analysis will be doing some extensive topology modifications, the copy variable command can be used to map the field that exists on the original geometry to the new geometry. This is necessary when performing such operations as dynamic mesh re-balancing, particle creation, or XFEM geometry modification.
- if the field is defined on some mesh database other than the input mesh file this command may be used to automatically map data between the two meshes databases. For example the temperature could have been calculated via an Aria run on one mesh, and then applied to the same geometry, but different mesh topology in Sierra/SM. For this mapping to function properly either the two meshes must occupy the same volume (`MAP_BY_PROXIMITY`) or the meshes must have identical topology and node and element ID numbers (`MAP_BY_ID`).

This `COPY VARIABLE` command specifies that the variable named `copy_var_name` will be used as the source variable. This `copy_var_name` is the name of a nodal field on an input mesh database. The `FROM MODEL <string>model_name` portion of the command is optional. If it is specified, the results are read from the mesh database named `model_name`, see Section 6.1. If the `model_name` is not provided, the region's finite element mesh database will be used to read the source variable.

Two types of mapping are available. The default MAP_BY_PROXIMITY option can be used to map data from a model with that occupies the same volume, but has a different topology than the analysis model. When using MAP_BY_PROXIMITY a node in the target mesh will define data by interpolating from nearby nodes in the source mesh. This mapping is done in the original coordinates and is not affected by displacement. The option MAP_BY_ID can be used to directly send data between nodes in the source and target mesh with the exact same ID numbers. The MAP_BY_ID option can provide a faster and potentially more accurate mapping if the two models being mapped between are known to have completely identical topology.

Read Variable Examples

The following command would initialize the unrotated_stress in the model based on reading the "stress" values that existed in the input mesh file at time 1.5.

```
BEGIN INITIAL CONDITION
  INCLUDE ALL BLOCKS
  INITIALIZE VARIABLE NAME = unrotated_stress(:)
  VARIABLE TYPE = ELEMENT
  READ VARIABLE = stress
  TIME = 1.5
END
```

The following command would define the temperature in a block as a function of time based on reading in the state variable at index five of a state array defined in the input mesh.

```
BEGIN PRESCRIBED TEMPERATURE
  block = block_32
  READ VARIABLE = aria_state_output(5)
END
```

Copy Variable Examples

The following command would define the temperature in a block by copying the variable temp of a block from model tempRead based on mapping the element ID to the analysis mesh.

```
BEGIN PRESCRIBED TEMPERATURE
  block = block_32
  copy variable = temp from model tempRead  MAP_BY_ID
END
```

The following command would map the displacement defined in some other mesh to the boundary of the analysis mesh. Note the dispOut variable must be a three component vector field on the mesh on which it is defined.

```
BEGIN PRESCRIBED DISPLACEMENT
  node set = nodelist_1 nodelist_50
  COPY VARIABLE = dispOut FROM MODEL dispSourceModel \#
    MAP_BY_PROXIMITY
  COMPONENT = <string>X|Y|Z
END
```

7.3.6. User Subroutine Commands

If the user subroutine option is used, the initial values will be calculated by a subroutine that is written by the user explicitly for this purpose. The subroutine will be called by Sierra/SM at the appropriate time to perform the calculations.

Following are the command lines related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name |
  SURFACE SUBROUTINE = <string>subroutine_name |
  ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

Only one of the first three command lines listed above can be specified in the command block. The particular command line selected depends on the mesh-entity type of the variable being initialized. For example, variables associated with nodes would be initialized if using the `NODE SET SUBROUTINE` command line, variables associated with faces if using the `SURFACE SUBROUTINE` command line, and variables associated with elements if using the `ELEMENT BLOCK SUBROUTINE` command line. The string `subroutine_name` is the name of a FORTRAN subroutine that is written by the user.

Following the selected subroutine command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 10.2.2 and consist of `SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`, `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and `SUBROUTINE STRING PARAMETER`. Examples of using these command lines are provided throughout Chapter 10.

The application of user subroutines for variable initialization is essentially the same as the application of user subroutines in general. See Section 7.4.10 and Chapter 10 for more details on implementing the user subroutine option.

When the user subroutine option is used for variable initialization, the user subroutine is called only once. Also, when a user subroutine is being used, the returned value is the new (initial) variable value at each mesh entity, and the flags array is ignored.

7.3.7. Distance Function Commands

```
DISTANCE TYPE = XTYPE|YTYPE|ZTYPE|RTYPE|
                RXTYPE|RYTYPE|RZTYPE
FUNCTION = <string>function_name
```

Usage of the DISTANCE TYPE command line in the INITIAL CONDITION command block indicates that initial values will be assigned based on the positions of the mesh entities in relation to the origin or the axes of the global coordinate system. A FUNCTION command line must also be given in the same INITIAL CONDITION block as the DISTANCE TYPE command line. The given function must be an xy-function, where the independent x-value is a distance quantity and the dependent y-value is returned as the initial condition value. The xy-function is evaluated at each mesh entity. The XTYPE, YTYPE, and ZTYPE options quantify distance as the signed x-, y-, or z-coordinate, respectively, of each mesh entity. The RTYPE option quantifies distance as the unsigned radial distance from the origin to the mesh entity. The RXTYPE, RYTYPE, and RZTYPE options quantify distance as the unsigned radial distance from the global x-axis, y-axis, and z-axis, respectively, to the mesh entity.



Warning: The distance function option is only implemented for VARIABLE TYPE = ELEMENT and the distance quantity for each element is evaluated at the averaged nodal coordinates (approximately the centroid) of the element.

7.3.8. Additional Command

This command line provides an additional option for the INITIAL CONDITION command block:

```
SCALE FACTOR = <real>scale_factor(1.0)
```

Any initial value can be scaled by use of the SCALE FACTOR command line. An initial value generated by any one of the three initial-value-setting options in this command block (i.e., constant magnitude, input mesh, or user subroutine) will be scaled by the real value scale_factor.

7.3.9. Specialized Initialization Methods

A handful of highly specialized initial condition routine options are documented below.

The CALCULATE AS CLOSEST PROJECTION TO SURFACE initializes a three component vector at an element centroid to the direction pointing to the closest point on the given provided sidesets closeSurfs. Such a direction is often used for defining body fitted coordinate systems. For example, a sideset could be placed on the top of a bolt head. Then this point direction direction would define the bolt aligned length direction for use in artificial strain or orthotropic material definition.

7.3.10. Examples

Initialize the stress tensors in an element block

```
# Initialize the material stress tensor on block 1 such that
#   s_xx = s_yy = s_zz = 2.3
#   s_xy = s_xz = s_yz = 0.0
BEGIN INITIAL CONDITION
  BLOCK = block_1
  INITIALIZE VARIABLE NAME = unrotated_stress
  VARIABLE TYPE = element
  MAGNITUDE = 2.3, 2.3, 2.3, 0.0, 0.0, 0.0
END
#
# Initialize the ZZ component of the material stress tensor on
# block_2 to 1.5
#
BEGIN INITIAL CONDITION
  BLOCK = block_2
  INITIALIZE VARIABLE NAME = unrotated_stress(zz)
  VARIABLE TYPE = element
  MAGNITUDE = 1.5
END
#
# Initialize the bolt aligned coordinate system for later
# use in defining artificial preload strain
#
BEGIN USER VARIABLE pdir
  TYPE = element vector
END
BEGIN INITIAL CONDITION
  BLOCK = block_3
  INITIALIZE VARIABLE NAME = pdir
  VARIABLE TYPE = element
  CALCULATE AS CLOSEST PROJECTION TO SURFACE = sideset_3
END
BEGIN ARTIFICIAL STRAIN
  BLOCK = block_3
  DIRECTION FIELD pdir FUNCTION = shrinkFunc
END
```

7.4. Kinematic Boundary Conditions

The various kinematic boundary conditions available in Sierra/SM are described in this section. The kinematic boundary conditions are nested inside the region scope.

Kinematic constraints can potentially be in conflict with other constraints. Refer to Appendix D for information on how conflicting constraints are handled.

7.4.1. Fixed Displacement

```
BEGIN FIXED DISPLACEMENT
  #
  # node set commands
  NODE SET|NODESET = <string list>nodelist_names
  SURFACE|SIDESET|SIDE SET = <string list>surface_names
  BLOCK = <string list>block_names
  RIGID BODY = <string list>rigid_body_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # direction commands
  COMPONENT = <string>X|Y|Z | COMPONENTS = <string>X/Y/Z |
    DIRECTION = <string>dirName
  #
  # additional commands
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [FIXED DISPLACEMENT]
```

The `FIXED DISPLACEMENT` command block fixes displacement components (X, Y, Z, or some combination thereof, or some other direction) for a set of nodes to be zero. This command block contains two groups of commands—node set and component. Each of these command groups is basically independent of the other. In addition to the command lines in the two command groups, there are two additional command lines: `ACTIVE PERIODS` and `INACTIVE PERIODS`. These are used to activate or deactivate this kinematic boundary condition for certain time periods.

For rigid bodies, any kinematic boundary condition is applied directly to the rigid body reference node. The kinematics for the nodes attached to the rigid body are applied via the rigid body constraint conditions. Since rigid bodies are defined using element blocks, only the `BLOCK` or `RIGID BODY` commands may be used to specify fixed displacement conditions on rigid bodies.



Warning: Note the fixed displacement command sets the displacement of a node to ZERO. To instead keep the current nodal displacement, a zero velocity boundary condition should be used.

Following are descriptions of the different command groups.

7.4.1.1. Node Set Commands

The `node set commands` portion of the `FIXED DISPLACEMENT` command block specifies the nodes associated with the boundary condition. This portion of the command block can include some combination of the following command lines:

```
NODE SET|NODESET = <string list>nodelist_names
SURFACE|SIDESET|SIDE SET = <string list>surface_names
BLOCK = <string list>block_names
RIGID BODY = <string list>rigid_body_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 7.1.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `NODE SET|NODESET`, `SURFACE`, `BLOCK`, `RIGID BODY`, or `INCLUDE ALL BLOCKS` command line in the command block.

7.4.1.2. Specification Commands

There are two component specification commands available in the `FIXED DISPLACEMENT` command block:

```
COMPONENT = X|Y|Z | COMPONENTS = X/Y/Z |
   DIRECTION = <string>dirName
```

A single displacement component may be set with the `COMPONENT` command line. A single direction of constraint can be set with the `DIRECTION` command. Multiple components can be constrained simultaneously with the `COMPONENTS`. One and only one `COMPONENT`, `DIRECTION`, or `COMPONENTS` command must be specified in the block. The `COMPONENTS` command can constraint multiple components simultaneously, for example

```
COMPONENTS = X Y
```

7.4.1.3. Additional Commands

The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines can optionally appear in the `FIXED DISPLACEMENT` command block:

```
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

These command lines determine when the boundary condition is active. See Section 2.6 for more information about these optional command lines.



Warning: If `FIXED DISPLACEMENT` becomes active the displacement is returned to zero. If the intention is to maintain the current displacement when activated `PRESCRIBED VELOCITY` with a value of zero should be used.

7.4.2. Prescribed Displacement

```
BEGIN PRESCRIBED DISPLACEMENT
  #
  # node set commands
  NODE SET|NODESET = <string list>nodelist_names
  SURFACE|SIDESET|SIDE SET = <string list>surface_names
  BLOCK = <string list>block_names
  RIGID BODY = <string list>rigid_body_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # specification commands
  DIRECTION = <string>defined_direction |
    COMPONENT = <string>X|Y|Z |  COMPONENTS = <string>X/Y/Z |
    CYLINDRICAL AXIS = <string>defined_axis |
    RADIAL AXIS = <string>defined_axis
  FUNCTION = <string>function_name
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # external database commands
  READ VARIABLE = <string>var_name
  COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
  TIME = <real>time|FIRST|LAST
  #
```

```
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [PRESCRIBED DISPLACEMENT]
```

The PRESCRIBED DISPLACEMENT command block prescribes a displacement field for a given set of nodes. The displacement field associates a vector giving the magnitude and direction of the displacement with each node in the set of nodes. The displacement field may vary over time and space. If the displacement field has only a time-varying magnitude and uses one of four methods for setting direction, the specification commands in the above command block can be used to specify the displacement field. If the displacement field is more complex, a user subroutine specifies the displacement field. The displacement field can also be read from an external mesh database. In a given boundary condition command block, commands from only one of the command groups (specification commands, user subroutine commands, or external database commands) may be used.

For rigid bodies, any kinematic boundary condition is applied directly to the rigid body reference node. The kinematics for the mesh nodes attached to the rigid body are applied via the rigid body constraint conditions. Since rigid bodies are defined using element blocks, only the BLOCK or RIGID BODY command lines may be used to specify prescribed displacement conditions on rigid bodies.

The PRESCRIBED DISPLACEMENT command block contains four groups of commands—node set, function, user subroutine, and external database. Each of these command groups is basically independent of the others. In addition to the command lines in the four command groups, there are three additional command lines: SCALE FACTOR, ACTIVE PERIODS, and INACTIVE PERIODS. The SCALE FACTOR command line can be used in conjunction with the specification commands the user subroutine commands, or the external database command. The ACTIVE PERIODS and INACTIVE PERIODS command lines are used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups and the SCALE FACTOR and ACTIVE PERIODS command lines.

7.4.2.1. Node Set Commands

The node set commands portion of the PRESCRIBED DISPLACEMENT command block defines a set of nodes associated with the prescribed displacement field and can include some combination of the following command lines:

```
NODE SET|NODESET = <string list>nodelist_names
SURFACE|SIDESET|SIDE SET = <string list>surface_names
BLOCK = <string list>block_names
RIGID BODY = <string list>rigid_body_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
```

```
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 7.1.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `NODE SET|NODESET`, `SURFACE`, `BLOCK`, `RIGID BODY`, or `INCLUDE ALL BLOCKS` command line in the command block.

7.4.2.2. Specification Commands

If the specification commands are used, the displacement vector at any given time is the same for all nodes in the node set associated with the particular `PRESCRIBED DISPLACEMENT` command block.

Following are the command lines used to specify the prescribed displacement with a direction and a function:

```
DIRECTION = <string>defined_direction |
  COMPONENT = <string>X|Y|Z |
  CYLINDRICAL AXIS = <string>defined_axis |
  RADIAL AXIS = <string>defined_axis
FUNCTION = <string>function_name
```

The displacement can be specified along an arbitrary user-defined direction, along a component direction (X, Y, or Z), along the azimuthal direction in a cylindrical coordinate system (defined in reference to an axis), or along a radial direction (defined in reference to an axis). Only one of these options is allowed per command block. The displacement is prescribed only in the specified direction. A prescribed displacement boundary condition does not influence the motion in directions orthogonal to the prescribed direction.

- The `DIRECTION` command line is used to prescribe displacement in an arbitrary user-defined direction. The name in the string `defined_direction` is a reference to a direction, which is defined using the `DEFINE DIRECTION` command block within the SIERRA scope.
- The `COMPONENT` command line specifies that the prescribed displacement vector lies along one of the component directions. The `COMPONENT` command line is a shortcut to an internally defined direction vector; for example, component `x` corresponds to using direction vector (1, 0, 0).
- The `CYLINDRICAL AXIS` command line specifies that the prescribed displacement is to be applied in the azimuthal direction of a cylindrical coordinate system. The string `defined_axis` refers to the name of the axis of the cylindrical coordinate system, and which is defined via a `DEFINE AXIS` command block in the SIERRA scope. The displacement is prescribed as a rotation in radians about the axis. Nodes with this type of boundary condition are free to move in the radial and height directions in the cylindrical coordinate system. Restraints can be placed on the node set in those directions if desired by

applying separate kinematic boundary conditions that contain RADIAL AXIS or DIRECTION commands that refer to the same axis. Note that this type of boundary condition is not a rotational boundary condition; it only affects translational degrees of freedom.



Known Issue: If a prescribed displacement with the CYLINDRICAL AXIS option is applied to nodes that fall on the axis, it will have no effect. Separate boundary conditions should be applied to those nodes to fix them in the plane normal to the axis.

- The RADIAL AXIS command line requires an axis definition that appears in the SIERRA scope. The string defined_axis uses an axis_name that is defined in the SIERRA scope (via a DEFINE AXIS command line). For this option, a radial line is drawn from a node to the radial axis. The prescribed displacement vector lies along this radial line from the node to the radial axis.

The magnitude of the displacement is specified by the FUNCTION command line. This references a function_name (defined in the SIERRA scope using a FUNCTION command block) that specifies the magnitude of the displacement vector as a function of time. The magnitude can be scaled by use of the SCALE FACTOR command line described in Section 7.4.2.5.

7.4.2.3. User Subroutine Commands

If the user subroutine option is used, the displacement vector may vary spatially at any given time for each of the nodes in the node set associated with the particular PRESCRIBED DISPLACEMENT command block. The user subroutine option allows for a more complex description of the displacement field than do the specification commands, but also requires writing a user subroutine to implement this capability. The user subroutine will be used to define a displacement direction and a magnitude for every node to which the boundary condition will be applied. The subroutine will be called by Sierra/SM at the appropriate time to generate the displacement field.

Following are the command lines related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the NODE SET SUBROUTINE command line. The string subroutine_name is the name of a FORTRAN subroutine that is written by the user.

Following the NODE SET SUBROUTINE command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 10.2.2

and consist of SUBROUTINE DEBUGGING OFF, SUBROUTINE DEBUGGING ON, SUBROUTINE REAL PARAMETER, SUBROUTINE INTEGER PARAMETER, and SUBROUTINE STRING PARAMETER. Examples of using these command lines are provided throughout Chapter 10.

The magnitude set in the user subroutine can be scaled by use of the SCALE FACTOR command line, as described in Section 7.4.2.5.

See Section 7.4.10 and Chapter 10 for more details on implementing the user subroutine option.

7.4.2.4. External Mesh Database Commands

If the external database option is used, the displacement vector (or specified components of the vector) is read from an external mesh database. The displacements are read from a finite element model defined via the FINITE ELEMENT MODEL command block described in Section 6.1. The finite element model can either be the model used by the region for its mesh definition as specified with the USE FINITE ELEMENT MODEL command (see Section 2.3), or it can be a different (but compatible) model. The following command lines control the use of an external mesh database to prescribe the displacement:

```
READ VARIABLE = <string>var_name
COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
  MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
TIME = <real>time |FIRST|LAST
```

Specifying multiple components using external mesh database commands is not currently implemented. To read all components from an external mesh database, please specify three separate prescribed displacement blocks and specify the three different components

See Section 7.3.5 for more details.

7.4.2.5. Additional Commands

These command lines in the PRESCRIBED DISPLACEMENT command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

The SCALE FACTOR command line is used to apply an additional scaling factor, which is constant in both time and space, to all vector magnitude values of the field defined by the specification commands or the user subroutine. For example, if the magnitude of the displacement in a time history function is given as 1.5 from time 1.0 to time 2.0 and the scale factor is 0.5, then the magnitude of the displacement from time 1.0 to 2.0 is 0.75. The default value for the scale factor is 1.0.

The ACTIVE PERIODS and INACTIVE PERIODS command lines determine when the boundary condition is active. See Section 2.6 for more information about these command lines.

7.4.3. Prescribed Velocity

```
BEGIN PRESCRIBED VELOCITY
  #
  # node set commands
  NODE SET|NODESET = <string list>nodelist_names
  SURFACE|SIDESET|SIDE SET = <string list>surface_names
  BLOCK = <string list>block_names
  RIGID BODY = <string list>rigid_body_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # specification commands
  DIRECTION = <string>defined_direction |
    COMPONENT = <string>X|Y|Z | COMPONENTS = <string>X/Y/Z |
    CYLINDRICAL AXIS = <string>defined_axis |
    RADIAL AXIS = <string>defined_axis
  FUNCTION = <string>function_name
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # external database commands
  READ VARIABLE = <string>var_name
  COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
    MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
  TIME = <real>time|FIRST|LAST
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [PRESCRIBED VELOCITY]
```

The PRESCRIBED VELOCITY command block prescribes a velocity field for a given set of nodes.

The velocity field associates a vector giving the magnitude and direction of the velocity with each node in the node set. The velocity field may vary over time and space. If the velocity field has only a time-varying magnitude and uses one of four methods for setting direction, the specification commands in the above command block can be used to specify the velocity field. If the velocity field is more complex, a user subroutine specifies the velocity field. The velocity field can also be read from an external mesh database. In a given boundary condition command block, commands from only one of the command groups (specification commands, user subroutine commands, or external database commands) may be used.

For rigid bodies, any kinematic boundary condition is applied directly to the rigid body reference node. The kinematics for the mesh nodes attached to the rigid body are applied via the rigid body constraint conditions. Since rigid bodies are defined using element blocks, only the `BLOCK` and `RIGID BODY` command lines may be used to specify prescribed velocity conditions on rigid bodies.

The `PRESCRIBED VELOCITY` command block contains four groups of commands—node set, function, user subroutine, and external database. Each of these command groups is basically independent of the others. In addition to the command lines in the four command groups, there are three additional command lines: `SCALE FACTOR`, `ACTIVE PERIODS`, and `INACTIVE PERIODS`. The `SCALE FACTOR` command line can be used in conjunction with either the specification commands or the user subroutine commands. The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines are used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups.

7.4.3.1. Node Set Commands

The `node set commands` portion of the `PRESCRIBED VELOCITY` command block defines a set of nodes associated with the prescribed velocity field and can include some combination of the following command lines:

```
NODE SET|NODESET = <string list>nodelist_names
SURFACE|SIDESET|SIDE SET = <string list>surface_names
BLOCK = <string list>block_names
RIGID BODY = <string list>rigid_body_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 7.1.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `NODE SET|NODESET`, `SURFACE`, `BLOCK`, `RIGID BODY`, or `INCLUDE ALL BLOCKS` command line in the command block.

7.4.3.2. Specification Commands

If the specification commands are used, the velocity vector at any given time is the same for all nodes in the node set associated with the particular PRESCRIBED VELOCITY command block.

Following are the command lines used to specify the prescribed velocity with a direction and a function:

```
DIRECTION = <string>defined_direction |
  COMPONENT = <string>X|Y|Z |
  CYLINDRICAL AXIS = <string>defined_axis |
  RADIAL AXIS = <string>defined_axis
FUNCTION = <string>function_name
```

The velocity can be specified along an arbitrary user-defined direction, along a component direction (X, Y, or Z), along the azimuthal direction in a cylindrical coordinate system (defined in reference to an axis), or along a radial direction (defined in reference to an axis). Only one of these options is allowed per command block. The velocity is prescribed only in the specified direction. A prescribed velocity boundary condition does not influence the motion in directions orthogonal to the prescribed direction.

- The DIRECTION command line is used to prescribe velocity in an arbitrary user-defined direction. The name in the string defined_direction is a reference to a direction, which is defined using the DEFINE DIRECTION command block within the SIERRA scope.
- The COMPONENT command line specifies that the prescribed velocity vector lies along one of the component directions. The COMPONENT command line is a shortcut to an internally defined direction vector; for example, component x corresponds to using direction vector (1, 0, 0).
- The CYLINDRICAL AXIS command line specifies that the prescribed velocity is to be applied in the azimuthal direction of a cylindrical coordinate system. The string defined_axis refers to the name of the axis of the cylindrical coordinate system, and which is defined via a DEFINE AXIS command block in the SIERRA scope. The velocity is prescribed as a rotation in radians about the axis. Nodes with this type of boundary condition are free to move in the radial and height directions in the cylindrical coordinate system. Restraints can be placed on the node set in those directions if desired by applying separate kinematic boundary conditions that contain RADIAL AXIS or DIRECTION commands that refer to the same axis. Note that this type of boundary condition is not a rotational boundary condition; it only affects translational degrees of freedom.

  Known Issue: If a prescribed velocity with the CYLINDRICAL AXIS option is applied to nodes that fall on the axis, it will have no effect. Separate boundary conditions should be applied to those nodes to fix them in the plane normal to the axis.
- The RADIAL AXIS command line requires an axis definition that appears in the SIERRA scope. The string defined_axis uses an axis_name that is defined in the SIERRA scope (via a DEFINE AXIS command line). For this option, a radial line is drawn from a node to the radial axis. The velocity vector lies along this radial line from the node to the radial axis.

The magnitude of the velocity is specified by the FUNCTION command line. This references a function_name (defined in the SIERRA scope using a FUNCTION command block) that specifies the magnitude of the velocity vector as a function of time. The magnitude can be scaled by use of the SCALE FACTOR command line described in Section 7.4.3.5.

7.4.3.3. User Subroutine Commands

If the user subroutine option is used, the velocity vector may vary spatially at any given time for each of the nodes in the node set associated with the particular PRESCRIBED VELOCITY command block. The user subroutine option allows for a more complex description of the velocity field than do the specification commands, but also requires writing a user subroutine to implement this capability. The user subroutine will be used to define a velocity direction and a magnitude for every node to which the boundary condition will be applied. The subroutine will be called by Sierra/SM at the appropriate time to generate the velocity field.

Following are the command lines related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the NODE SET SUBROUTINE command line. The string subroutine_name is the name of a FORTRAN subroutine that is written by the user.

Following the NODE SET SUBROUTINE command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 10.2.2 and consist of SUBROUTINE DEBUGGING OFF, SUBROUTINE DEBUGGING ON, SUBROUTINE REAL PARAMETER, SUBROUTINE INTEGER PARAMETER, and SUBROUTINE STRING PARAMETER. Examples of using these command lines are provided throughout Chapter 10.

The magnitude set in the user subroutine can be scaled by use of the SCALE FACTOR command line, as described in Section 7.4.3.5.

7.4.3.4. External Mesh Database Commands

If the external database option is used, the velocity vector (or specified components of the vector) is read from an external mesh database. The velocities are read from a finite element model defined via the FINITE ELEMENT MODEL command block described in Section 6.1. The finite element model can either be the model used by the region for its mesh definition as specified with the USE FINITE ELEMENT MODEL command (see Section 2.3), or it can be a different (but

compatible) model. The following command lines control the use of an external mesh database to prescribe the velocity:

```
READ VARIABLE = <string>var_name
COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
  MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
TIME = <real>time|FIRST|LAST
```

See Section 7.3.5 for more details.

7.4.3.5. Additional Commands

These command lines in the `PRESCRIBED VELOCITY` command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

The `SCALE FACTOR` command line is used to apply an additional scaling factor, which is constant in both time and space, to all vector magnitude values of the field defined by the specification commands or the user subroutine. For example, if the magnitude of the velocity in a time history function is given as 1.5 from time 1.0 to time 2.0 and the scale factor is 0.5, then the magnitude of the velocity from time 1.0 to 2.0 is 0.75. The default value for the scale factor is 1.0.

The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines determine when the boundary condition is active. See Section 2.6 for more information about these command lines.

7.4.4. Prescribed Acceleration

```
BEGIN PRESCRIBED ACCELERATION
  #
  # node set commands
  NODE SET|NODESET = <string list>nodelist_names
  SURFACE|SIDESET|SIDE SET = <string list>surface_names
  BLOCK = <string list>block_names
  RIGID BODY = <string list>rigid_body_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # specification commands
  DIRECTION = <string>defined_direction |
    COMPONENT = <string>X|Y|Z | COMPONENTS = <string>X/Y/Z |
```

```
  FUNCTION = <string>function_name
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # external database commands
  READ VARIABLE = <string>var_name
  COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
    MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
  TIME = <real>time|FIRST|LAST
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [PRESCRIBED ACCELERATION]
```

The `PRESCRIBED ACCELERATION` command block prescribes an acceleration field for a given set of nodes. The acceleration field associates a vector giving the magnitude and direction of the acceleration with each node in the node set. The acceleration field may vary over time and space. If the acceleration field has only a time-varying component, the specification commands in the above command block can be used to specify the acceleration field. If the acceleration field has both time-varying and spatially varying components, a user subroutine specifies the acceleration field. The acceleration field can also be read from an external mesh database. In a given boundary condition command block, commands from only one of the command groups (specification commands, user subroutine commands, or external database commands) may be used.

For rigid bodies, any kinematic boundary condition is applied directly to the rigid body reference node. The kinematics for the mesh nodes attached to the rigid body are applied via the rigid body constraint conditions. Since rigid bodies are defined using element blocks, only the `BLOCK` and `RIGID BODY` commands may be used to specify prescribed acceleration conditions on rigid bodies.

The `PRESCRIBED ACCELERATION` command block contains four groups of commands: node set, function, user subroutine, and external database. Each of these command groups is independent of the others. In addition to the command lines in the four command groups, there are three additional command lines: `SCALE FACTOR`, `ACTIVE PERIODS`, and `INACTIVE PERIODS`. The `SCALE FACTOR` command line can be used in conjunction with either the specification commands or the user subroutine commands. The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines are used to activate or deactivate this kinematic boundary

condition for certain time periods. Following are descriptions of the different command groups.

7.4.4.1. Node Set Commands

The `node set commands` portion of the `PRESCRIBED ACCELERATION` command block defines a set of nodes associated with the prescribed acceleration field and can include some combination of the following command lines:

```
NODE SET|NODESET = <string list>nodelist_names
SURFACE|SIDESET|SIDE SET = <string list>surface_names
BLOCK = <string list>block_names
RIGID BODY = <string list>rigid_body_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 7.1.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `NODE SET|NODESET`, `SURFACE`, `BLOCK`, `RIGID BODY`, or `INCLUDE ALL BLOCKS` command line in the command block.

7.4.4.2. Specification Commands

If the specification commands are used, the acceleration vector at any given time is the same for all nodes in the node set associated with the particular `PRESCRIBED ACCELERATION` command block. The direction of the acceleration vector is constant for all time; the magnitude of the acceleration vector may vary with time, however.

Following are the command lines used to specify the prescribed acceleration with a direction and a function:

```
DIRECTION = <string>defined_direction |
  COMPONENT = <string>X|Y|Z
FUNCTION = <string>function_name
```

The acceleration can be specified along an arbitrary user-defined direction or along a component direction (X, Y, or Z). Only one of these options is allowed per command block. The acceleration is prescribed only in the specified direction. A prescribed acceleration boundary condition does not influence the motion in directions orthogonal to the prescribed direction.

- The `DIRECTION` command line is used to prescribe acceleration in an arbitrary user-defined direction. The name in the string `defined_direction` is a reference to a direction, which is defined using the `DEFINE DIRECTION` command block within the SIERRA scope.

- The COMPONENT command line specifies that the prescribed acceleration vector lies along one of the component directions. The COMPONENT command line is a shortcut to an internally defined direction vector; for example, component x corresponds to using direction vector (1, 0, 0).

The magnitude of the acceleration is specified by the FUNCTION command line. This references a function_name (defined in the SIERRA scope using a FUNCTION command block) that specifies the magnitude of the acceleration vector as a function of time. The magnitude can be scaled by use of the SCALE FACTOR command line described in Section 7.4.4.5.

7.4.4.3. User Subroutine Commands

If the user subroutine option is used, the acceleration vector may vary spatially at any given time for each of the nodes in the node set associated with the particular PRESCRIBED ACCELERATION command block. The user subroutine option allows for a more complex description of the acceleration field than do the specification commands, but also requires writing a user subroutine to implement this capability. The user subroutine will be used to define an acceleration direction and a magnitude for every node to which the boundary condition will be applied. The subroutine will be called by Sierra/SM at the appropriate time to generate the acceleration field.

Following are the command lines related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the NODE SET SUBROUTINE command line. The string subroutine_name is the name of a FORTRAN subroutine that is written by the user.

Following the NODE SET SUBROUTINE command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 10.2.2 and consist of SUBROUTINE DEBUGGING OFF, SUBROUTINE DEBUGGING ON, SUBROUTINE REAL PARAMETER, SUBROUTINE INTEGER PARAMETER, and SUBROUTINE STRING PARAMETER. Examples of using these command lines are provided throughout Chapter 10.

The magnitude set in the user subroutine can be scaled by use of the SCALE FACTOR command line, as described in Section 7.4.4.5.

See Section 7.4.10 and Chapter 10 for more details on implementing the user subroutine option.

7.4.4.4. External Mesh Database Commands

If the external database option is used, the acceleration vector (or specified components of the vector) is read from an external mesh database. The accelerations are read from a finite element model defined via the `FINITE ELEMENT MODEL` command block described in Section 6.1. The finite element model can either be the model used by the region for its mesh definition as specified with the `USE FINITE ELEMENT MODEL` command (see Section 2.3), or it can be a different (but compatible) model. The following command lines control the use of an external mesh database to prescribe the acceleration:

```
READ VARIABLE = <string>var_name
COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
  MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
TIME = <real>time|FIRST|LAST
```

See Section 7.3.5 for more details.

7.4.4.5. Additional Commands

These command lines in the `PRESCRIBED ACCELERATION` command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

The `SCALE FACTOR` command line is used to apply an additional scaling factor, which is constant in both time and space, to all vector magnitude values of the field defined by the specification commands or the user subroutine. For example, if the magnitude of the acceleration in a time history function is given as 1.5 from time 1.0 to time 2.0 and the scale factor is 0.5, then the magnitude of the acceleration from time 1.0 to 2.0 is 0.75. The default value for the scale factor is 1.0.

The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines determine when the boundary condition is active. See Section 2.6 for more information about these command lines.

7.4.5. Periodic Boundary Condition

```
BEGIN PERIODIC
  SIDE B  = <string>side_b_node_set
  SIDE A = <string>side_a_node_or_face_set
  COMPONENT = <string>X/Y/Z
  PRESCRIBED QUANTITY = DISPLACEMENT|FORCE|VELOCITY
  SEARCH TOLERANCE = <real>tol
```

```
  FUNCTION = <string>func_name
  SCALE FACTOR = <real>func_scale
  POINT ON AXIS = <string>point_name
  REFERENCE AXIS = <string>direction_name
  THETA = <real>theta
  ACTIVE PERIODS = <string list>periods_names
  INACTIVE PERIODS = <string list>periods_names
END [PERIODIC]
```

The PERIODIC boundary condition command is used to define a set of periodic boundary constraints between two surfaces.

The side_b_node_set is the name of the side B node set for the constraints. The side_a_node_or_face_set is the name of either the side A node set or side A surface for the constraint. If a side A node set is used the side A node set and side B node set must have the same number of nodes. Additionally the side A node set and side B node set must be related by a simple translation. When two node sets are specified, a periodic constraint is defined between each pair of coincident (after applying the translation) nodes in the two node sets.

When a node set and a surface are specified the constraints will be defined between the side B nodes and the faces of the side A surface. When using surfaces to specify periodic boundary conditions, the number of nodes must be the same on the side A and side B sides. The side B node set and side A face set must still be related to one another by a simple translation.

COMPONENT specifies which directional component should enforce the prescribed quantity. This quantity can be DISPLACEMENT, VELOCITY, or FORCE as defined by the PRESCRIBED QUANTITY command. Note, all degrees of freedom on the node are considered periodic. Only the COMPONENT will have the PRESCRIBED QUANTITY enforced, the other degrees of freedom will be effectively linked with zero offset unless overridden by another periodic boundary condition.



Warning: At this time the COPY VARIABLE command can only be used with nodal variables. Neither the COPY VARIABLE command nor the READ VARIABLE command is compatible with global variables.



Warning: Multi-component periodic BCs are not currently supported with FETI/-Full Tangent Preconditioner. Please break-up each component of your periodic BCs into separate periodic BC command blocks.

Note that neither of the COMPONENT or PRESCRIBED QUANTITY command are needed when dealing with rotational symmetry.

The FUNCTION and SCALE FACTOR commands are used to define a relative difference in the prescribed quantity on the side B and side A sets. This function can be used to define an isotropic expansion or contraction of a periodic boundary. For the common default use case the function is set to zero zero meaning the prescribed quantities on the two sets are the same.

The SEARCH TOLERANCE command specifies a search tolerance used for matching up nodes and faces between the side A and side B sets. The search tolerance is necessary to account for slightly varying mesh topologies between the side A and side B sets. The search tolerance should be something small, generally some small fraction of the characteristic face length of facets on the surfaces.
The commands POINT ON AXIS, REFERENCE AXIS, and THETA are used to define axisymmetric periodic wedge conditions. point_name gives the name of a point defined by a domain level DEFINE POINT command. direction_name gives the name of a point defined by a domain level DEFINE DIRECTION command. Taken together these two commands define the axis of rotation for the wedge body. THETA defines the angle, in degrees, covered by the wedge. By way of the thumb rule, THETA should always be positive when sweeping from side A to side B, in a positive rotation.





Warning: The axisymmetric periodic boundary condition capability is still in development and has only been evaluated for implicit, serial analysis.

Examples:

```
SIDE B               = nodeset_11
SIDE A               = nodeset_12
COMPONENT            = xz
PRESCRIBED QUANTITY = displacement
SEARCH TOLERANCE     = 0.001
FUNCTION             = function_1
SCALE FACTOR         = 0.0
```

The above commands will create a periodic boundary condition between node sets 11 and 12. The translational offset between nodeset_11 and nodeset_12 will be automatically determined. The coincident (after the translation is applied) nodes in these node sets will be matched up. For each node pair a constraint will be created to ensure the x- and z-displacements of the two nodes in the pair are equal.



Warning: Periodic boundary conditions should generally not be combined with other kinematic boundary conditions (such as fixed displacement.) The periodic boundary conditions define a full set of kinematic constraints and additional constraints may conflict with the periodic constraints in unexpected ways.

7.4.6. Fixed Rotation

```
BEGIN FIXED ROTATION
  #
  # node set commands
  NODE SET|NODESET = <string list>nodelist_names
  SURFACE|SIDESET|SIDE SET = <string list>surface_names
```

```
  BLOCK = <string list>block_names
  RIGID BODY = <string list>rigid_body_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # specification commands
  COMPONENT = <string>X/Y/Z | COMPONENTS = <string>X/Y/Z |
    DIRECTION = <string>dir_name
  #
  # additional commands
  ACTIVE PERIODS = <string list>periods_names
  INACTIVE PERIODS = <string list>periods_names
END [FIXED ROTATION]
```

The `FIXED ROTATION` command is only applied to nodes that have rotational degrees of freedom such as shell element nodes and rigid body reference nodes. In the case of rigid bodies, the boundary condition must be specified on the block that defines the rigid body using the `BLOCK` or `RIGID BODY` line command.

The `FIXED ROTATION` command block fixes rotation about direction components (X, Y, Z, or some combination thereof) for a set of nodes to zero. This command block contains two groups of commands—node set and component. Each of these command groups is basically independent of the other. In addition to the command lines in the two command groups, there are additional command lines: `ACTIVE PERIODS` and `INACTIVE PERIODS`. These command lines are used to activate or deactivate this kinematic boundary condition for certain time periods.



Warning: Note the fixed rotation command sets the rotation of a node to ZERO. To instead keep the current nodal displacement, a zero rotational velocity boundary condition should be used.

Following are descriptions of the different command groups.

7.4.6.1. Node Set Commands

The `node set commands` portion of the command block specifies the nodes associated with the boundary condition. This portion of the command block can include some combination of the following command lines:

```
NODE SET|NODESET = <string list>nodelist_names
SURFACE|SIDESET|SIDE SET = <string list>surface_names
BLOCK = <string list>block_names
RIGID BODY = <string list>rigid_body_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
```

```
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 7.1.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `NODE SET|NODESET`, `SURFACE`, `BLOCK`, `RIGID BODY`, or `INCLUDE ALL BLOCKS` command line in the command block.

7.4.6.2. Specification Commands

There are two component specification commands available in the `FIXED ROTATION` command block:

```
COMPONENT = X|Y|Z | COMPONENTS = X/Y/Z |
   DIRECTION = <string>dirName
```

A single rotation component may be set with the `COMPONENT` command line. A single direction of constraint can be set with the `DIRECTION` command. Multiple components can be constrained simultaneously with the `COMPONENTS`. One and only one `COMPONENT`, `DIRECTION`, or `COMPONENTS` command must be specified in the block. The `COMPONENTS` command can constraint multiple components simultaneously, for example

```
COMPONENTS = X Y
```

7.4.6.3. Additional Commands

The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines can optionally appear in the `FIXED ROTATION` command block:

```
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

This command line determines when the boundary condition is active. See Section 2.6 for more information about this optional command line.

7.4.7. Prescribed Rotation

```
BEGIN PRESCRIBED ROTATION
  #
  # node set commands
  NODE SET|NODESET = <string list>nodelist_names
  SURFACE|SIDESET|SIDE SET = <string list>surface_names
```

```
  BLOCK = <string list>block_names
  RIGID BODY = <string list>rigid_body_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # specification commands
  DIRECTION = <string>defined_direction
  COMPONENT = X|Y|Z
  FUNCTION = <string>function_name
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # external database commands
  READ VARIABLE = <string>var_name
  COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
    MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
  TIME = <real>time|FIRST|LAST
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [PRESCRIBED ROTATION]
```



Warning: The `BEGIN PRESCRIBED ROTATION` command is only applied to nodes that have rotational degrees of freedom such as shell element nodes and rigid body reference nodes. Displacements consistent with rotation about a fixed axis for nodes that do not have rotational degrees of freedom, are imposed using the `CYLINDRICAL AXIS` command line in the `PRESCRIBED DISPLACEMENT` command block described in Section 7.4.2.



Warning: The behavior of the BEGIN PRESCRIBED ROTATION command differs for implicit and explicit analyses. For implicit analyses, the BEGIN PRESCRIBED ROTATION command fully prescribes all rotational degrees of freedom. Consequently, a prescribed rotation in a given direction restricts rotation in the remaining two orthogonal directions. If multiple prescribed rotation boundary conditions are applied to a node, only the last one specified in the input is enforced.

In explicit analyses, the BEGIN PRESCRIBED ROTATION command block prescribes rotational degree of freedom in only the specified direction. The node is free to rotate in the directions orthogonal to that direction. Multiple prescribed rotation boundary conditions can be applied to constrain rotations in those other directions if desired.

For nodal rotational degrees of freedom, the rotations applied using the BEGIN PRESCRIBED ROTATION command are the three components of the rotational degree of freedom itself. For rigid bodies, the rotations must be the nonlinear components that result in the desired quaternion on the rigid body reference node. For rigid bodies, the boundary condition is applied to the reference node and therefore must be specified on the block that defines the rigid body using the BLOCK or RIGID BODY line command.

The PRESCRIBED ROTATION command block prescribes the rotation about an axis for a given set of nodes. The rotation field associates a vector giving the magnitude and direction of the rotation with each node in the node set. The rotation field may vary over time and space. If the rotation field has only a time-varying component, the specification commands in the above command block can be used to specify the rotation field. If the rotation field has both time-varying and spatially varying components, a user subroutine is used to specify the rotation field. The rotation field can also be read from an external mesh database. In a given boundary condition command block, commands from only one of the command groups (specification commands, user subroutine commands, or external database commands) may be used.

The PRESCRIBED ROTATION command block contains four groups of commands—node set, function, user subroutine, and external database. Each of these command groups is basically independent of the others. In addition to the command lines in the four command groups, there are three additional command lines: SCALE FACTOR, ACTIVE PERIODS, and INACTIVE PERIODS. The SCALE FACTOR command line can be used in conjunction with either the specification commands or the user subroutine commands. The ACTIVE PERIODS and INACTIVE PERIODS command lines are used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups.

7.4.7.1. Node Set Commands

The node set commands portion of the PRESCRIBED ROTATION command block defines a set of nodes associated with the prescribed rotation field and can include some combination of the following command lines:

```
NODE SET|NODESET = <string list>nodelist_names
SURFACE|SIDESET|SIDE SET = <string list>surface_names
BLOCK = <string list>block_names
RIGID BODY = <string list>rigid_body_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 7.1.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `NODE SET|NODESET`, `SURFACE`, `BLOCK`, `RIGID BODY`, or `INCLUDE ALL BLOCKS` command line in the command block.

7.4.7.2. Specification Commands

If the specification commands are used, the rotation vector at any given time is the same for all nodes in the node set associated with the particular `PRESCRIBED ROTATION` command block. The direction of the rotation vector is constant for all time; the magnitude of the rotation vector may vary with time, however.

Following are the command lines used to specify the prescribed rotation with a direction and a function:

```
DIRECTION = <string>defined_direction
COMPONENT = X|Y|Z
FUNCTION = <string>function_name
```

The `DIRECTION` or `COMPONENT` command line are used to prescribe the direction of rotation. The name in the string `defined_direction` is a reference to a direction, which is defined using the `DEFINE DIRECTION` command block within the SIERRA scope. The rotation is prescribed only in the specified direction. A prescribed rotation boundary condition does not influence the rotation in directions orthogonal to the prescribed direction.

The magnitude of the rotation is specified by the `FUNCTION` command line. This references a `function_name` (defined in the SIERRA scope in a `FUNCTION` command block) that specifies the magnitude of the rotation vector as a function of time. The magnitude can be scaled by use of the `SCALE FACTOR` command line described in Section 7.4.7.5.

The magnitude of the rotation, as specified by the product of the function and the scale factor, has units of radians per second.

7.4.7.3. User Subroutine Commands

If the user subroutine option is used, the rotation vector may vary spatially at any given time for each of the nodes in the node set associated with the particular `PRESCRIBED ROTATION`

command block. The user subroutine option allows for a more complex description of the rotation field than do the specification commands, but also requires writing a user subroutine to implement this capability. The user subroutine will be used to define a rotation direction and a magnitude for every node to which the boundary condition will be applied. The subroutine will be called by Sierra/SM at the appropriate time to generate the rotation field.

Following are the command lines related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the `NODE SET SUBROUTINE` command line. The string `subroutine_name` is the name of a FORTRAN subroutine that is written by the user.

Following the `NODE SET SUBROUTINE` command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 10.2.2 and consist of `SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`, `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and `SUBROUTINE STRING PARAMETER`. Examples of using these command lines are provided throughout Chapter 10.

The magnitude set in the user subroutine can be scaled by use of the `SCALE FACTOR` command line, as described in Section 7.4.7.5.

See Section 7.4.10 and Chapter 10 for more details on implementing the user subroutine option.

7.4.7.4. External Mesh Database Commands

If the external database option is used, the rotation vector (or specified components of the vector) is read from an external mesh database. The rotations are read from a finite element model defined via the `FINITE ELEMENT MODEL` command block described in Section 6.1. The finite element model can either be the model used by the region for its mesh definition as specified with the `USE FINITE ELEMENT MODEL` command (see Section 2.3), or it can be a different (but compatible) model. The following command lines control the use of an external mesh database to prescribe the rotation:

```
READ VARIABLE = <string>var_name
COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
  MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
TIME = <real>time|FIRST|LAST
```

See Section 7.3.5 for more details.

7.4.7.5. Additional Commands

These command lines in the PRESCRIBED ROTATION command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

The SCALE FACTOR command line is used to apply an additional scaling factor, which is constant in both time and space, to all vector magnitude values of the field defined by the specification commands or the user subroutine. For example, if the magnitude of the rotation in a time history function is given as 1.5 from time 1.0 to time 2.0 and the scale factor is 0.5, then the magnitude of the rotation from time 1.0 to 2.0 is 0.75. The default value for the scale factor is 1.0.

The ACTIVE PERIODS and INACTIVE PERIODS command lines determine when the boundary condition is active. See Section 2.6 for more information about these command lines.

7.4.8. Prescribed Rotational Velocity

```
BEGIN PRESCRIBED ROTATIONAL VELOCITY
  #
  # node set commands
  NODE SET|NODESET = <string list>nodelist_names
  SURFACE|SIDESET|SIDE SET = <string list>surface_names
  BLOCK = <string list>block_names
  RIGID BODY = <string list>rigid_body_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # specification commands
  DIRECTION = <string>defined_direction |
   COMPONENT = <string>X|Y|Z
  FUNCTION = <string>function_name
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
```

```
  #
  # external database commands
  READ VARIABLE = <string>variable_name
  COPY VARIABLE = <string>variable_name [FROM MODEL
    <string>model_name]
  TIME = <real>time|FIRST|LAST
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [PRESCRIBED ROTATIONAL VELOCITY]
```



Warning: The PRESCRIBED ROTATIONAL VELOCITY command is only applied to nodes that have rotational degrees of freedom such as shell element nodes and rigid body reference nodes. Velocities consistent with rotation about a fixed axis for nodes that do not have rotational degrees of freedom, are imposed with the CYLINDRICAL AXIS command line in the PRESCRIBED VELOCITY command block described in Section 7.4.3.



Warning: The behavior of the BEGIN PRESCRIBED ROTATIONAL VELOCITY command differs for implicit and explicit analyses. For implicit analyses, the BEGIN PRESCRIBED ROTATIONAL VELOCITY command fully prescribes all rotational degrees of freedom. Consequently, a prescribed rotational velocity in a given direction restricts rotation in the remaining two orthogonal directions. If multiple prescribed rotational velocity boundary conditions are applied to a node, only the last one specified in the input is enforced.

In explicit analyses, the BEGIN PRESCRIBED ROTATIONAL VELOCITY command block prescribes rotational degree of freedom in only the specified direction. The node is free to rotate in the directions orthogonal to that direction. Multiple prescribed rotational velocity boundary conditions can be applied to constrain rotations in those other directions if desired.

The PRESCRIBED ROTATIONAL VELOCITY command block prescribes the rotational velocity about an axis for a given set of nodes. The rotational velocity field associates a vector giving the magnitude and direction of the rotational velocity with each node in the node set. The rotational velocity field may vary over time and space. If the rotational velocity field has only a time-varying component, the specification commands in the above command block can be used to specify the rotational velocity field. If the rotational velocity field has both time-varying and spatially varying components, a user subroutine is used to specify the rotational velocity field. The rotational velocity field can also be read from an external mesh database. In a given boundary condition command block, commands from only one of the command groups (specification commands, user subroutine commands, or external database commands) may be used.

For rigid bodies, the user must specify the rotational velocity components such that they define the desired quaternion on the rigid body reference node. For rigid bodies, the boundary condition is applied to the reference node and therefore must be specified on the block that defines the rigid body using the BLOCK or RIGID BODY line command.

The PRESCRIBED ROTATIONAL VELOCITY command block contains four groups of commands—node set, function, user subroutine, and external database. Each of these command groups is basically independent of the others. In addition to the command lines in the four command groups, there are three additional command lines: SCALE FACTOR, ACTIVE PERIODS, and INACTIVE PERIODS. The SCALE FACTOR command line can be used in conjunction with either the specification commands or the user subroutine commands. The ACTIVE PERIODS and INACTIVE PERIODS command lines are used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups.

7.4.8.1. Node Set Commands

The node set commands portion of the PRESCRIBED ROTATIONAL VELOCITY command block defines a set of nodes associated with the prescribed rotational velocity field and can include some combination of the following command lines:

```
NODE SET|NODESET = <string list>nodelist_names
SURFACE|SIDESET|SIDE SET = <string list>surface_names
BLOCK = <string list>block_names
RIGID BODY = <string list>rigid_body_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 7.1.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one NODE SET|NODESET, SURFACE, BLOCK, RIGID BODY, or INCLUDE ALL BLOCKS command line in the command block.

7.4.8.2. Specification Commands

If the specification commands are used, the rotational velocity vector at any given time is the same for all nodes in the node set associated with the particular PRESCRIBED ROTATIONAL VELOCITY command block. The direction of the rotational velocity vector is constant for all time; the magnitude of the rotational velocity vector may vary with time, however.

Following are the command lines used to specify the prescribed rotational velocity with a direction and a function:

```
DIRECTION = <string>defined_direction |
  COMPONENT = <string>X|Y|Z
FUNCTION = <string>function_name
```

The rotational velocity can be specified either along an arbitrary user-defined direction or along a component direction (X, Y, or Z), but not both. The rotational velocity is prescribed only in the specified direction. A prescribed rotational velocity boundary condition does not influence the rotational velocity in directions orthogonal to the prescribed direction.

- The `DIRECTION` command line is used to prescribe rotational velocity in an arbitrary user-defined direction. The name in the string `defined_direction` is a reference to a direction, which is defined using the `DEFINE DIRECTION` command block within the SIERRA scope.
- The `COMPONENT` command line specifies that the prescribed rotational velocity vector lies along one of the component directions. The `COMPONENT` command line is a shortcut to an internally defined direction vector; for example, component `x` corresponds to using direction vector (1, 0, 0).

The magnitude of the rotational velocity is specified by the `FUNCTION` command line. This references a `function_name` (defined in the SIERRA scope in a `FUNCTION` command block) that specifies the magnitude of the rotational velocity vector as a function of time. The magnitude can be scaled by use of the `SCALE FACTOR` command line described in Section 7.4.8.5.

The magnitude of the rotational velocity, as specified by the product of the function and the scale factor, has units of radians per second.

7.4.8.3. User Subroutine Commands

If the user subroutine option is used, the rotational velocity vector may vary spatially at any given time for each of the nodes in the node set associated with the particular `PRESCRIBED ROTATIONAL VELOCITY` command block. The user subroutine option allows for a more complex description of the rotational velocity field than do the specification commands, but also requires writing a user subroutine to implement this capability. The user subroutine will be used to define a rotational velocity direction and a magnitude for every node to which the boundary condition will be applied. The subroutine will be called by Sierra/SM at the appropriate time to generate the rotational velocity field.

Following are the command lines related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the NODE SET SUBROUTINE command line. The string subroutine_name is the name of a FORTRAN subroutine that is written by the user.

Following the NODE SET SUBROUTINE command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 10.2.2 and consist of SUBROUTINE DEBUGGING OFF, SUBROUTINE DEBUGGING ON, SUBROUTINE REAL PARAMETER, SUBROUTINE INTEGER PARAMETER, and SUBROUTINE STRING PARAMETER. Examples of using these command lines are provided throughout Chapter 10.

The magnitude set in the user subroutine can be scaled by use of the SCALE FACTOR command line, as described in Section 7.4.8.5.

See Section 7.4.10 and Chapter 10 for more details on implementing the user subroutine option.

7.4.8.4. External Mesh Database Commands

If the external database option is used, the rotational velocity vector (or specified components of the vector) is read from an external mesh database. The rotational velocities are read from a finite element model defined via the FINITE ELEMENT MODEL command block described in Section 6.1. The finite element model can either be the model used by the region for its mesh definition as specified with the USE FINITE ELEMENT MODEL command (see Section 2.3), or it can be a different (but compatible) model. The following command lines control the use of an external mesh database to prescribe the rotational velocity:

```
READ VARIABLE = <string>var_name
COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
  MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
TIME = <real>time|FIRST|LAST
```

See Section 7.3.5 for more details.

7.4.8.5. Additional Commands

These command lines in the PRESCRIBED ROTATIONAL VELOCITY command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

The SCALE FACTOR command line is used to apply an additional scaling factor, which is constant in both time and space, to all vector magnitude values of the field defined by the specification commands or the user subroutine. For example, if the magnitude of the rotational velocity in a time history function is given as 1.5 from time 1.0 to time 2.0 and the scale factor is

0.5, then the magnitude of the rotational velocity from time 1.0 to 2.0 is 0.75. The default value for the scale factor is 1.0.

The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines determine when the boundary condition is active. See Section 2.6 for more information about these command lines.

7.4.9. Reference Axis Rotation

```
BEGIN REFERENCE AXIS ROTATION
  #
  # block commands
  BLOCK = <string list>block_names
  RIGID BODY = <string list>rigid_body_names
  #
  # specification commands
  REFERENCE AXIS X FUNCTION = <string>function_name
  REFERENCE AXIS Y FUNCTION = <string>function_name
  REFERENCE AXIS Z FUNCTION = <string>function_name
  #
  # rotation commands
  ROTATION = <string>function_name
  ROTATIONAL VELOCITY = <string>function_name
  #
  # torque command
  TORQUE = <string>function_name
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [REFERENCE AXIS ROTATION]
```

The `REFERENCE AXIS ROTATION` command block is intended to control the rotation of a rigid body. The three `REFERENCE AXIS` line commands are required and together define a time-varying reference axis or vector. The rigid body will rotate to follow the vector. Depending on other line commands in the block, the rigid body will have either zero or one free rotational degree of freedom. If one degree of freedom is present, it is the rotation about the time-varying reference vector.

The rotation of the rigid body described by the axis functions over any time step is determined using the reference axis position at the beginning and the end of the time step. If a reference axis is desired to represent movement of a principal axis of the rigid body, its position at the start of rotation must coincide with the position of that principal axis. A constant function can only be used to represent a spin around a principal axis if the principal axis and the reference axis are the same. To represent spin about a principal axis that is moving or is stationary but in a different position than the principal axis, the time history for the axis functions must define the reference axis in the same position as the principal axis at the beginning of the rotation.

At most, one of the ROTATION, ROTATIONAL VELOCITY, or TORQUE line commands may be present. If the ROTATION or ROTATIONAL VELOCITY command line is present, the rigid body will be fully prescribed for rotation. The ACTIVE PERIODS and INACTIVE PERIODS command lines are used to activate or deactivate this kinematic boundary condition for certain time periods.

Use of the REFERENCE AXIS ROTATION command block will cause extra global variables to be written to the results file. These variables describe the reaction force, rotational reaction moment, rotational displacement, and rotational velocity associated with the boundary condition and in the local coordinate system described by the boundary condition. The names of the variables are:

```
REACTR_<NAME>
REACTS_<NAME>
REACTT_<NAME>

RREACTR_<NAME>
RREACTS_<NAME>
RREACTT_<NAME>

ROTDS_<NAME>

ROTVR_<NAME>
ROTVS_<NAME>
ROTVT_<NAME>
```

The variables beginning with REACT give the reaction forces in the boundary condition's local coordinate system. Those beginning with RREACT give the reaction moments.

Only the S component is given for the rotational displacement. This is because that the other coordinate directions are poorly defined, and therefore, rotational displacements in those directions provide no value.

The variables beginning with ROTV give the rotational velocity in the local coordinate system.

For each global variable, <NAME> is the name of the rigid body controlled by this boundary condition.

7.4.9.1. Block Command

The block command portion of the REFERENCE AXIS ROTATION command block defines a block associated with the prescribed rotation field and must include the following command line:

```
BLOCK = <string list>block_names
```

If the element block in the BLOCK command has been defined as a rigid body, the rotation is applied to the rigid body reference node.

Alternatively, the command line:

```
RIGID BODY = <string list>rigid_body_names
```

may be used to specify the rotation on the reference node of the specified rigid body.

See Section 7.1.1 for more information about the use of this command line.

7.4.9.2. Specification Commands

The three specification commands are required and together define a time-varying vector that gives the orientation of a reference axis for rotation. The magnitude of the vector is ignored.

Following are the command lines used to specify the reference axis rotation with a direction:

```
REFERENCE AXIS X FUNCTION = <string>function_name
REFERENCE AXIS Y FUNCTION = <string>function_name
REFERENCE AXIS Z FUNCTION = <string>function_name
```

At a given time, the function referred to by the `REFERENCE AXIS X FUNCTION` line command gives the x component of a vector defining the reference axis. The pattern holds for the y and z components as well. Since the magnitude of the resulting axis or vector is not used, the three functions do not need to describe a unit vector in time.

7.4.9.3. Rotation Commands

It is possible to prescribe the complete rotation of a rigid body through the use of the `ROTATION` or `ROTATIONAL VELOCITY` line commands. Without these line commands, the rigid body is free to rotate about the reference axis.

Following are the command lines to prescribe the rotation about the reference axis:

```
ROTATION = <string>function_name
ROTATIONAL VELOCITY = <string>function_name
```

The function referred to by the `ROTATION` line command gives the magnitude of rotation in radians about the reference axis as a function of time. This function should not be changed across a restarted analysis.

The function referred to by the `ROTATIONAL VELOCITY` line command gives the rotational velocity in radians per second about the reference axis as a function of time. This function should not be changed across a restarted analysis.

7.4.9.4. Torque Command

A user may specify a moment about the reference axis through the use of the `TORQUE` line command.

```
TORQUE = <string>function_name
```

The function referred to by the TORQUE line command gives the torque or moment about the reference axis as a function of time.

7.4.9.5. Additional Commands

The SCALE FACTOR, ACTIVE PERIODS, and INACTIVE PERIODS command lines can optionally appear in the REFERENCE AXIS ROTATION command block:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

The SCALE FACTOR command line applies the specified value as a scale factor to the rotation, rotational velocity, or torque functions.

The final two command lines can activate or deactivate the reference axis rotation for certain time periods. See Section 2.6 for more information about these command lines.

7.4.10. Subroutine Usage for Kinematic Boundary Conditions

The prescribed kinematic boundary conditions may be defined by a user subroutine. All these conditions use a node set subroutine. See Chapter 10 for an in-depth discussion of user subroutines. The kinematic boundary conditions will be applied to nodes. The subroutine will have to return six output values per node and one output flag per node. The usage of the output values depends on the returned flag value for a node, as follows:

- If the flag value is negative, no constraint will be applied to the node.
- If the flag value is equal to zero, the constraint will be absolute. All components of the boundary condition will be specified. For example, suppose the user has specified a subroutine to be called from a prescribed displacement subroutine. The prescribed displacements are to be passed through an array output_values. For a given node inode, the output_values array would have the following values:

```
output_values(1,inode) = displacement in x at inode
output_values(2,inode) = displacement in y at inode
output_values(3,inode) = displacement in z at inode
output_values(4,inode) = not used
output_values(5,inode) = not used
output_values(6,inode) = not used
```

- If the flag value is equal to one, the constraint will be a specified amount in a given direction. For example, suppose the user has specified a subroutine to be called from a prescribed displacement subroutine. The prescribed displacements are to be passed through an array output_values. For a given node inode, the output_values array would have the following values:

```
output_values(1,inode) = magnitude of displacement
output(values(2,inode) = not used
output_values(3,inode) = not used
output_values(4,inode) = x component of direction vector
output_values(5,inode) = y component of direction vector
output_values(6,inode) = z component of direction vector
```

The direction in which the constraint will act is given by `output_values` 4 through 6 for `inode`. The magnitude of the displacement in the specified direction is given by `output_values` 1 at `inode`. To compute the constraint, Sierra/SM first normalizes the direction vector. Next, Sierra/SM multiplies the normalized direction vector by the magnitude of the displacement and applies the resultant constraint vector.

Displacements or velocities orthogonal to the prescribed direction will not be constrained. (This is true regardless of whether one uses a user subroutine for the prescribed kinematic boundary conditions.) Take the case of a prescribed displacement condition. The displacement orthogonal to a prescribed direction of motion depends on the internal and external forces orthogonal to the prescribed direction. Displacement orthogonal to the prescribed direction may or may not be zero.

7.5. Initial Velocity Conditions

```
BEGIN INITIAL VELOCITY
  #
  # node set commands
  NODE SET|NODESET = <string list>nodelist_names
  SURFACE|SIDESET|SIDE SET = <string list>surface_names
  BLOCK = <string list>block_names
  RIGID BODY = <string list>rigid_body_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # direction specification commands
  COMPONENT = <string>X|Y|Z |
    DIRECTION = <string>defined_direction
  #
  #  magnitude specification commands
  MAGNITUDE = <real>magnitude_of_velocity
  FUNCTION = <string>function_name
  SCALE FACTOR = <real>function_scale
  #
  # angular velocity specification commands
  CYLINDRICAL AXIS = <string>defined_axis
  ANGULAR VELOCITY = <real>angular_velocity
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
END [INITIAL VELOCITY]
```

The `INITIAL VELOCITY` command block specifies an initial velocity field for a set of nodes. There are two simple options for specifying the initial velocity field: by direction and by angular velocity. The user subroutine option available is also available to specify an initial velocity. Only one of the available options—direction specification, angular velocity specification, or user subroutine—may be used.

The `INITIAL VELOCITY` command block contains four groups of commands—node set, direction specification, angular velocity specification, and user subroutine. Command lines associated with the node set commands must appear. As mentioned, command lines associated

with one of the options must also appear. Following are descriptions of the different command groups.



Warning: The `INITIAL VELOCITY` command block may *not* be used to set an initial rotational velocity for rigid bodies. To do this with rigid bodies, see 6.3.1. This block only allows the specification of translational degrees of freedom for rigid bodies.



Warning: Initial velocity follows the same constraint enforcement hierarchy as other kinematic boundary conditions. Thus, if two non-orthogonal initial velocities are specified the result will not be an addition of the two velocities, rather the last boundary condition defined in the input deck will override the non-orthogonal components of previous boundary conditions. See Section D for further details and examples.

If it is desired to set an initial spin and translational velocity on the same part a subroutine exists in the user subroutine library to accomplish this task. See Section 10.5.3.

7.5.1. Node Set Commands

The `node set commands` portion of the `INITIAL VELOCITY` command block defines a set of nodes associated with the initial velocity field and can include some combination of the following command lines:

```
NODE SET|NODESET = <string list>nodelist_names
SURFACE|SIDESET|SIDE SET = <string list>surface_names
BLOCK = <string list>block_names
RIGID BODY = <string list>rigid_body_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

For rigid bodies, the boundary condition is applied to the reference node and must be specified on the block that defines the rigid body using the `BLOCK` or `RIGID BODY` line command.

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 7.1.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `NODE SET|NODESET`, `SURFACE`, `BLOCK`, `RIGID BODY`, or `INCLUDE ALL BLOCKS` command line in the command block.

7.5.2. Direction Specification Commands

If the direction specification commands are used, the initial velocity is applied along a defined direction with a specific magnitude. Following are the command lines for the direction option:

```
COMPONENT = <string>X|Y|Z |
  DIRECTION = <string>defined_direction
MAGNITUDE = <real>magnitude_of_velocity
FUNCTION = <string>function_name
SCALE FACTOR = <real>function_scale
```

The initial velocity can be specified either along an arbitrary user-defined direction or along a component direction (X, Y, or Z), but not both. The velocity is prescribed only in the specified direction. A prescribed velocity boundary condition does not influence the movement in directions orthogonal to the prescribed direction.

- The `DIRECTION` command line is used to prescribe initial velocity in an arbitrary user-defined direction. The name in the string `defined_direction` is a reference to a direction, which is defined using the `DEFINE DIRECTION` command block within the SIERRA scope.
- The `COMPONENT` command line specifies that the initial velocity vector lies along one of the component directions. The `COMPONENT` command line is a shortcut to an internally defined direction vector; for example, component `x` corresponds to using direction vector (1, 0, 0).

The magnitude of the initial velocity can be given in a few ways. One option is the `MAGNITUDE` which defines a constant initial velocity for the boundary condition. A second option is use of a `FUNCTION` command. The `FUNCTION` is evaluated at the initial time to set the initial velocity. Generally the `FUNCTION` command would be used with spatially dependent analytic functions. The `SCALE FACTOR` command line defines an additional multiplier on the magnitude or magnitude function.

Either the `COMPONENT` command line or the `DIRECTION` command line must be specified when using the `MAGNITUDE` or `FUNCTION` command lines.

7.5.3. Angular Velocity Specification Commands

If the angular velocity specification commands are used, the initial velocity is applied as an initial angular velocity about some axis. Following are the command lines for angular velocity specification:

```
CYLINDRICAL AXIS = <string>defined_axis
ANGULAR VELOCITY = <real>angular_velocity
```

The axis about which the body is initially rotating is given by the `CYLINDRICAL AXIS` command line. The string `defined_axis` uses an `axis_name` that is defined in the SIERRA scope (via a `DEFINE AXIS` command line).

The magnitude of the angular velocity about this axis is specified by the ANGULAR VELOCITY command line with the real value angular_velocity. This value is specified in units of radians per unit of time. Typically, the value for the angular velocity will be radians per second.

Both the CYLINDRICAL AXIS and ANGULAR VELOCITY command lines are required when using the angular velocity specification commands.

7.5.4. User Subroutine Commands

If the user subroutine option is used, the initial velocity vector may vary spatially at any given time for each of the nodes in the node set associated with the particular INITIAL CONDITION command block. The user subroutine option allows for a more complex description of the initial velocity field than do the direction and angular-velocity options, but the user subroutine option also requires writing a user subroutine to implement this capability. The user subroutine will be used to define a velocity direction and a magnitude for every node to which the initial velocity field will be applied. The subroutine will be called by Sierra/SM at the appropriate time to generate the initial velocity field.

Following are the command lines related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the NODE SET SUBROUTINE command line. The string subroutine_name is the name of a FORTRAN subroutine that is written by the user.

Following the NODE SET SUBROUTINE command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 10.2.2 and consist of SUBROUTINE DEBUGGING OFF, SUBROUTINE DEBUGGING ON, SUBROUTINE REAL PARAMETER, SUBROUTINE INTEGER PARAMETER, and SUBROUTINE STRING PARAMETER. Examples of using these command lines are provided throughout Chapter 10.

See Section 7.4.10 and Chapter 10 for more details on implementing the user subroutine option.

7.6. Force Boundary Conditions

A variety of force boundary conditions are available in Sierra/SM. This section describes these boundary conditions.

7.6.1. Pressure

```
BEGIN PRESSURE
  #
  # surface set commands
  SURFACE|SIDESET|SIDE SET = <string list>surface_names
  REMOVE SURFACE = <string list>surface_names
  BLOCK = <string list> block_names
  INCLUDE ALL BLOCKS
  #
  # specification commands
  FUNCTION = <string>function_name
  VELOCITY DAMPING COEFFICIENT = <real>damping_coefficient
  # user subroutine commands
  SURFACE SUBROUTINE = <string>subroutine_name |
    NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # external pressure sources
  READ VARIABLE = <string>var_name
  COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
    MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
  TIME = <real>time|FIRST|LAST
  OBJECT TYPE = <string>NODE|FACE(NODE)

  FIELD VARIABLE = <string>field_variable
  SCALAR SOURCE VARIABLE = <string>var_name INDEX
    <integer>var_index REGION <string>region_name
  #
  # pressure masking
  MASKING METHOD = OFF|ON|CONTACT(OFF)
  ADDITIONAL MASKING FIELD = <string>var_name
  #
  # output external forces from pressure
  EXTERNAL FORCE CONTRIBUTION OUTPUT NAME =
```

```
    <string>variable_name
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names

  INTEGRATION METHOD =
    CODE SELECTED|MULTILINEAR|QUADRATIC(CODE_SELECTED)

END [PRESSURE]
```

The PRESSURE command block applies a pressure to each face of the associated surfaces. The pressure field can be constant over the faces and vary in time, dependent on the average face velocity based on a damping coefficient, or determined by a user subroutine. If the pressure field is constant over the faces and has only a time-varying component, the function command in the above command block can be used to specify the pressure field. For a velocity-dependent pressure, a damping coefficient is specified. If the pressure field has both time-varying and spatially varying components, user subroutine commands are used to specify the pressure field. The pressure field may also be obtained from a mesh file or from another SIERRA code through a transfer operator. Only one of these options—function, velocity damping coefficient, user subroutine, mesh file, transfer from another code—may be used to specify the pressure field in a particular PRESSURE command block.

The PRESSURE command block can be used for surfaces that have faces derived from solid elements (eight-node hexahedra, four-node tetrahedra, eight-node tetrahedra, etc.), membranes, and shells. The PRESSURE command block can also be used for particle like elements if the elements are created through the use of element death particle conversion, Section 6.5.4.10. The surfaces must be defined on the original solid elements.

Pressure is integrated in the current deformed model configuration. Thus if due to displacement a face grows, shrinks, or rotates the net force produced by the pressure boundary condition will change.

A pressure boundary condition generates nodal forces that are summed into the external force vector that is used to calculate the motion of a body. The external force vector contains the contribution from all forces acting on the body. There is an option in the PRESSURE command block to save information about the contribution to the external force vector due only to pressure loads. This option does not change the magnitude or time history of the pressure load (regardless of how they are defined), but merely stores information in a user-accessible variable.



Warning: Pressure is integrated over each face and applied at the nodes with forces. The contribution to each node on a face is based on topology, which in some cases is simplified from that of the underlying element. For quadrilateral, triangular, and a few higher order faces (e.g., eight and nine node quadrilaterals) the topology used is exact, but in some cases it is an approximation formed from subdividing the face into multiple triangular faces.

The PRESSURE command block contains five groups of commands—surface set, function, user subroutine, external pressure, and output external forces. Each of these command groups is basically independent of the others. In addition to the command lines in the five command groups, there are two additional command lines: SCALE FACTOR and ACTIVE PERIODS. The SCALE FACTOR command line will apply an additional scaling factor to the pressure value computed. The ACTIVE PERIODS and INACTIVE PERIODS command lines are used to activate or deactivate this force boundary condition for certain time periods. Following are descriptions of the different command groups.

7.6.1.1. Surface Set Commands

The surface set commands portion of the PRESSURE command block defines a set of surfaces associated with the pressure field and can include some combination of the following command lines:

```
SURFACE = <string list>surface_names
BLOCK =  <string list>block_names
INCLUDE ALL BLOCKS
REMOVE SURFACE = <string list>surface_names
```

In the SURFACE command line, a series of surfaces may be given through the string list surface_names. When using the BLOCK command the pressure will be applied to the exterior boundary of the block. When using the INCLUDE ALL BLOCKS command the pressure will be applied to the exterior boundary of all blocks in the model. Note, the NODE SET|NODESET command is not valid for specifying an entity for pressure boundary conditions.

There must be at least one SURFACE, BLOCK, or INCLUDE ALL BLOCKS command line in the command block. The REMOVE SURFACE command allows creation of an application surface formed from the union of surfaces specified in the SURFACE, BLOCK, or INCLUDE ALL BLOCKS minus those surfaces specified in the REMOVE SURFACE command.

See Section 7.1.1 for more information about the use of these command lines for creating a set of surfaces used by the boundary condition.

7.6.1.2. Specification Commands

If the function command is used, the pressure vector at any given time is the same for all surfaces associated with the particular PRESSURE command block. The direction of the pressure vector is constant for all time; the magnitude of the pressure vector may vary with time.

Following is the command line used to specify the pressure with a function:

```
FUNCTION = <string>function_name
```

The pressure is applied in the opposite direction to the outward normals of the faces that define the surfaces. The magnitude of the pressure is specified by the FUNCTION command line. This

references a function_name (defined in the SIERRA scope in a FUNCTION command block) that specifies the magnitude of the pressure vector as a function of time. The magnitude can be scaled by use of the SCALE FACTOR command line described in Section 7.6.1.7.

A face force based on a velocity damping coefficient can be applied using the command line:

```
VELOCITY DAMPING COEFFICIENT = <real>damping_coefficient
```

In this case, nodal forces will be applied to the nodes of any faces included in the surfaces associated with the PRESSURE command block. A force vector acting normal to the face is computed by multiplying the average nodal velocity normal to the face by the damping coefficient. This face force is then evenly distributed among the nodes of the face.

7.6.1.3. User Subroutine Commands

If the user subroutine option is used, the pressure may vary spatially at any given time for each of the surfaces associated with the particular PRESSURE command block. The user subroutine option allows for a more complex description of the pressure field than does the function command, but also requires writing a user subroutine to implement this capability. The user subroutine will be used to define a pressure for every face to which the boundary condition will be applied. The subroutine will be called by Sierra/SM at the appropriate time to generate the pressure field.

Following are the command lines related to the user subroutine option:

```
SURFACE SUBROUTINE = <string>subroutine_name |
  NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the SURFACE SUBROUTINE command line or the NODE SET SUBROUTINE command line. The string subroutine_name in both command lines is the name of a FORTRAN subroutine that is written by the user. The particular command line selected depends on the mesh-entity type for which the pressure field is being calculated. Associating pressure values with faces would require the use of a SURFACE SUBROUTINE command line. Associating pressure values with nodes would require the use of a NODE SET SUBROUTINE command line.

Following the selected subroutine command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 10.2.2 and consist of SUBROUTINE DEBUGGING OFF, SUBROUTINE DEBUGGING ON, SUBROUTINE REAL PARAMETER, SUBROUTINE INTEGER PARAMETER, and

SUBROUTINE STRING PARAMETER. Examples of using these command lines are provided throughout Chapter 10.

The magnitude set in the user subroutine can be scaled by use of the SCALE FACTOR command line, as described in Section 7.6.1.7.

Usage requirements. Following are the usage requirements for the two types of subroutines:

- The surface subroutine operates on a group of faces. The subroutine will return one output value per face. Suppose a provided user subroutine returns the pressure information through an array output_value. The value output_value(1,iface) corresponds to the average pressure on face iface. The values of the flags array are not used.
- The node set subroutine provided will return one value per node. Suppose a provided user subroutine returns the pressure information through an array output_value. The return value output_value(1,inode) is the pressure at the node inode. The total pressure on the each face is found by integrating the pressures at the nodes. The values of the flags array are not used.

See Chapter 10 for more details on implementing the user subroutine option.

7.6.1.4. External Pressure Sources

Pressure may be obtained from two different external sources. The first option for obtaining pressure uses a mesh database. The pressures are read from a finite element model defined via the FINITE ELEMENT MODEL command block described in Section 6.1. The finite element model can either be the model used by the region for its mesh definition as specified with the USE FINITE ELEMENT MODEL command (see Section 2.3), or it can be a different (but compatible) model. The following command lines control the use of an external mesh database to prescribe the pressure:

```
READ VARIABLE = <string>var_name
COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
  MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
TIME = <real>time|FIRST|LAST
OBJECT TYPE = <string>NODE|FACE(FACE)
```

See Section 7.3.5 for more details.

The OBJECT TYPE command line specifies whether the pressure field on the mesh file is specified for nodes (the mesh object type is NODE) or for faces (the mesh object type is FACE). If the OBJECT TYPE command line is not present, it is assumed that the named variable is defined on faces.

If the pressure field is defined on the nodes, the field used should have one value per node which is the node centered pressure. If the pressure field is defined on faces, there are two options for field format. The first option is to have one value defined per face, which defines a constant

pressure over each face. The second option is to define *numNode* values per face where *numNode* is the number of nodes on each face (four per quad, three per triangle, etc.) When this option is used, the pressure field is defined at each corner of the face; this linearly varying pressure is then integrated over the face to compute the net forces. These nodal pressure weightings are sometimes called pressure distribution factors in other codes.

The second option for obtaining pressure from an external source relies on the transfer of information from another SIERRA code. The command for obtaining pressure information by transfer from another code is:

```
FIELD VARIABLE = <string>var_name
```

Here, `var_name` is the name of the variable where pressure information is to be stored. The pressure information will be transferred into this variable from another SIERRA code via a transfer operator.

The third option for obtaining pressure from an external source relies on the exchanging of information in a two-way coupled simulation with another SIERRA code. The command for obtaining pressure in this manner is:

```
SCALAR SOURCE VARIABLE = <string>var_name INDEX
      <integer>var_index REGION <string>region_name
```

In this command, `var_name` is the name of the variable where pressure information is stored in the other code. `INDEX var_index` is the index into the variable for multi-component fields. If the pressure source is a single global, the command is `INDEX 0`. `REGION region_name` is the name of the Sierra region Sierra/SM is coupling to.

An example of using this command would be the following: In the Sierra code Aria, a bulk fluid node can be used to calculate the temperature and pressure of air in an enclosed volume. Once this is calculated, the bulk fluid node pressure can then be applied to the interior surface of the container enclosing the air in the Adagio region. Similarly, this command can also be used in thermal-mechanical foam decomposition simulations.

7.6.1.5. Masking

Masking will remove the external force contribution of pressure on some of the nodes included in the pressure boundary condition. This capability can be used to avoid duplicity in boundary conditions, for example by masking the nodes with non-zero contact forces. The masking command options are:

```
MASKING METHOD = OFF|ON|CONTACT(OFF)
ADDITIONAL MASKING FIELD = <string>var_name
```

The `MASKING METHOD` defines whether masking is active for the pressure boundary condition and how it should be applied. The default behavior (`OFF`) is to not mask. The `ON` masking method will

apply masking by evaluating a user-specified nodal field. The ADDITIONAL MASKING FIELD defines this masking field, which can be any scalar or vector nodal field stored on the nodes included in the pressure boundary condition. The external force contribution of the pressure will be zero for every node where the magnitude of the masking field is non-zero. The CONTACT masking method will automatically mask pressure on nodes with non-zero contact forces. The CONTACT masking method will mask pressure on any nodes with non-zero contact and any nodes with a non-zero user-specified field if the ADDITIONAL MASKING FIELD is specified.

7.6.1.6. Output Command

This command line lets the user create a variable that stores information about the contribution to the external force vector at a node arising solely from a pressure:

```
EXTERNAL FORCE CONTRIBUTION OUTPUT NAME =
  <string>variable_name
```

If the above command line appears in a PRESSURE command block, then there will be a variable created with whatever name the user specifies for variable_name. The variable defines a three-dimensional vector at each node associated with this particular command block. The three-dimensional vector at each node represents the external force due solely to the pressure on the elements attached to that node. For example, if one of the nodes associated with this particular command block has four elements attached to it and each element has a pressure load, then the external force contribution at the node would be summed from the pressure load for all four elements.

Once this variable for the external force contribution from a pressure load is specified, it may be used like any other nodal variable. The user can, for example, specify the variable as a nodal variable to be output in a RESULTS OUTPUT command block. Or the user can reference the variable in a user subroutine.

7.6.1.7. Additional Commands

These command lines in the PRESSURE command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

The SCALE FACTOR command line is used to apply an additional scaling factor, which is constant in both time and space, to all vector magnitude values of the field defined by the function command or the user subroutine. For example, if the magnitude of the pressure in a time history function is given as 1.5 from time 1.0 to time 2.0 and the scale factor is 0.5, then the magnitude of the pressure from time 1.0 to 2.0 is 0.75. The default value for the scale factor is 1.0.

Generally if multiple pressure sources are defined they are multiplied together. For example if all the FUNCTION, SCALE FACTOR, and READ VARIABLE commands were defined the total pressure at each face would be the value of the function at the current time, times the value of the mesh variable at the specific faces, times the scaling factor. However, Some combinations of sources conflict and are not allowed to be used in the same boundary condition such as SUBROUTINE and READ VARIABLE.

The ACTIVE PERIODS and INACTIVE PERIODS command lines determine when the boundary condition is active. See Section 2.6 for more information about these command lines.

7.6.1.8. Surface Integration Schemes

```
INTEGRATION METHOD =
  CODE SELECTED|MULTILINEAR|QUADRATIC(CODE_SELECTED)
```

By default Sierra/SMwill pick the pressure integration method that is most consistent with the underlying element type. For example using a quadratic pressure integration scheme on the face of a twenty seven node hexahedron and a linear pressure integration scheme on the face of eight node hexahedron. The higher order tetrahedra elements present some special cases. Fully integrated ten node tetrahedra are more consistent with a quadratic surface integration scheme and by default will use a quadratic pressure integration. The composite or mean quadrature ten node tetrahedra are more consistent with a multi-linear surface integration scheme and by default will use the linear integration method. An optional command INTEGRATION METHOD can be used to override this default behavior and explicitly set the integration scheme for pressure on six node triangle faces.

7.6.1.9. Example

```
BEGIN FUNCTION One
  TYPE = CONSTANT
  BEGIN VALUES
    1.0
  END VALUES
END FUNCTION One

BEGIN PRESSURE
 SURFACE = sideset_1
 FUNCTION = One
 SCALE FACTOR =1
END PRESSURE
```

This PRESSURE applies a force vector with a magnitude of 1.0 times the area associated with each node in sideset_1 as shown in Figure 7-1. The force vectors are normal to the surface at the point which they are applied.

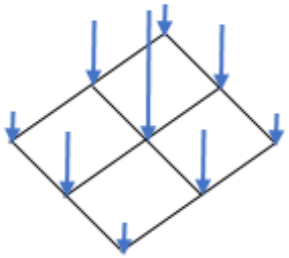


Figure 7-1. Result of a unit pressure load

7.6.2. Traction

```
BEGIN TRACTION
  #
  # surface set commands
  SURFACE|SIDESET|SIDE SET = <string list>surface_names
  REMOVE SURFACE = <string list>surface_names
  BLOCK = <string list> blocks
  INCLUDE ALL BLOCKS
  #
  # specification commands
  DIRECTION = <string>direction_name
  FUNCTION = <string>function_name
  #
  # external traction sources
  READ VARIABLE = <string>var_name
  COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
    MAP_BY_PROXIMITY| MAP_BY_ID(MAP_BY_PROXIMITY)
  TIME = <real>time|FIRST|LAST
  OBJECT TYPE = <string>FACE|NODE(FACE)
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SURFACE SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
  #
  # surface integration schemes
  TRACTION INTEGRATION METHOD =
    NODAL_LUMPING|GAUSS_QUADRATURE
END [TRACTION]
```

The `TRACTION` command block applies a traction to each face in the associated surfaces. The traction has units of force per unit area. (A traction, unlike a pressure, may not necessarily be in the direction of the normal to the face to which it is applied.) The given traction is integrated over the surface area of a face.

The traction field can be determined by a SIERRA function or a user subroutine. If the traction

field is constant over the faces and has only a time-varying component, the specification commands in the above command block can be used to specify the traction field. If the traction field has both time-varying and spatially varying components, a user subroutine is used to specify the traction field.

The traction force is integrated in the current deformed model configuration. The value of the traction force is the value of the traction field defined by this boundary condition times the face area. Thus if due to displacement a face grows or shrinks the net force produced by the traction boundary condition will change. As the traction field is defined explicitly with a direction a rotation of a face will not change the traction force on that face.

The traction field can only be controlled by one method. Accordingly, a `TRACTION` command block can only contain one of the options: function or user subroutine.

Currently, the `TRACTION` command block can be used for surfaces that have faces derived from solid elements (eight-node hexahedra, four-node tetrahedra, eight-node tetrahedra, etc.), membranes, and shells.

The `TRACTION` command block contains three groups of commands surface set, specification, and user subroutine. Each of these command groups is basically independent of the others. In addition to the command lines in the four command groups, there are three additional command lines: `SCALE FACTOR`, `ACTIVE PERIODS`, and `INACTIVE PERIODS`. The `SCALE FACTOR` command line can be used in conjunction with the specification commands or the user subroutine option. The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines are used to activate or deactivate this force boundary condition for certain time periods. Following are descriptions of the different command groups.

7.6.2.1. Surface Set Commands

The `surface set commands` portion of the `TRACTION` command block defines a set of surfaces associated with the traction field. The surface specification commands work the same as for the pressure boundary condition as described in section 7.6.1.1.

7.6.2.2. Traction Specification Commands

If the direction command is used, the traction vector at any given time is the same for all surfaces associated with the particular `TRACTION` command block. The direction of the traction vector is constant for all time; the magnitude of the traction vector may vary with time, however.

Following are the command lines used to specify the traction with a direction and a function:

```
DIRECTION = <string>defined_direction
FUNCTION = <string>function_name
```

The traction is specified in an arbitrary user-defined direction, and is defined using the `DIRECTION` command line. The name in the string `defined_direction` is a reference to a direction, which is defined using the `DEFINE DIRECTION` command block within the SIERRA

scope. Note that the direction is always a unit vector, even if it is specified with a non-unit vector in the DEFINE DIRECTION command block.

The magnitude of the traction is specified by the FUNCTION command line. This references a function_name (defined in the SIERRA scope in a FUNCTION command block) that specifies the magnitude of the traction vector as a function of time. The magnitude can be scaled by use of the SCALE FACTOR command line described in Section 7.6.2.5.

7.6.2.3. User Subroutine Commands

If the user subroutine option is used, the traction vector may vary spatially at any given time for each of the surfaces associated with the particular TRACTION command block. The user subroutine option allows for a more complex description of the traction field than does the function option, but also requires writing a user subroutine to implement this capability.

A node set user subroutine will be used to define a traction at each node attached to a face which the boundary condition will be applied. This nodal traction is then integrated over the faces. A surface subroutine can be used to define a constant traction on each face. The subroutines will be called by Sierra/SM at the appropriate time to generate the traction field.

Following is the command line related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name
SURFACE SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the NODE SET or SURFACE subroutine command lines. The string subroutine_name is the name of a FORTRAN subroutine that is written by the user. Associating traction values with nodes requires the use of a NODE SET SUBROUTINE command line. Associating traction values with faces requires the use of a NODE SET SUBROUTINE command line.

Following the NODE SET SUBROUTINE command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 10.2.2 and consist of SUBROUTINE DEBUGGING OFF, SUBROUTINE DEBUGGING ON, SUBROUTINE REAL PARAMETER, SUBROUTINE INTEGER PARAMETER, and SUBROUTINE STRING PARAMETER. Examples of using these command lines are provided throughout Chapter 10.

The magnitude set in the user subroutine can be scaled by use of the SCALE FACTOR command line, as described in Section 7.6.2.5.

7.6.2.3.1. Usage requirements for the node set subroutine. The provided subroutine will return three values per node or face. Suppose the user subroutine provided passes the output values through an array `output_values`. For a given node inode, the `output_values` array would have the following values:

```
output_values(1,inode) = x component of traction vector
output_values(2,inode) = y component of traction vector
output_values(2,inode) = z component of traction vector
```

The values of the flags array are not used in the traction subroutine.

See Chapter 10 for more details on implementing the user subroutine option.

7.6.2.4. External Mesh Database Commands

The traction distribution may be defined by a field on the mesh database. When using a mesh database defined distribution the traction magnitude and/or direction is read from a finite element model defined via the `FINITE ELEMENT MODEL` command block described in Section 6.1. The finite element model is the model used by the region for its mesh definition as specified with the `USE FINITE ELEMENT MODEL` command (see Section 2.3).

```
READ VARIABLE = <string>var_name
COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
  MAP_BY_PROXIMITY| MAP_BY_ID(MAP_BY_PROXIMITY)
TIME = <real>time|FIRST|LAST
OBJECT TYPE = <string>FACE|NODE(FACE)
```

See Section 7.3.5 for more details.



Warning: At this time the `COPY VARIABLE` command can only be used with nodal variables. Neither the `COPY VARIABLE` command nor the `READ VARIABLE` command is compatible with global variables.

The `OBJECT TYPE` command line specifies whether the traction field on the mesh file is specified for nodes (the mesh object type is `NODE`) or for faces (the mesh object type is `FACE`). If the `OBJECT TYPE` command line is not present, it is assumed that the named variable is defined on faces.

The `READ VARIABLE` command gives the mesh file variable which defines the traction. There are two ways to define this traction field: the first, is to define a field with a single scalar value for each node or face and couple that with a `DIRECTION` command to define the traction direction; the second, is to define a vector field with three components. In the latter case, the field can define a unique traction magnitude and direction at each node or face. Furthermore, this option should not be used with the `DIRECTION` command.

The COPY VARIABLE command is a more flexible version of the READ VARIABLE command. This command allows a user to read a mesh variable from another mesh database and map it to another mesh by proximity or ID then use it to apply the traction.

If present, the TIME command defines the time slice in the input mesh file from which to read the traction field. If the TIME field is not present, the traction will be defined by interpolation between time slices closest to the current analysis time. Specifying the keyword FIRST or LAST will read from the first or last time slice in the input mesh file, respectively.

7.6.2.5. Additional Commands

These command lines in the TRACTION command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

The SCALE FACTOR command line is used to apply an additional scaling factor, which is constant in both time and space, to all vector magnitude values of the field defined by the specification commands or the user subroutine. For example, if the magnitude of the traction in a time history function is given as 1.5 from time 1.0 to time 2.0 and the scale factor is 0.5, then the magnitude of the traction from time 1.0 to 2.0 is 0.75. The default value for the scale factor is 1.0.

The ACTIVE PERIODS and INACTIVE PERIODS command lines determine when the boundary condition is active. See Section 2.6 for more information about these command lines.

7.6.2.6. Surface Integration Schemes

```
TRACTION INTEGRATION METHOD =
  NODAL_LUMPING|GAUSS_QUADRATURE
```

By default, Sierra/SM will automatically choose the traction integration method. This input command can over-ride the default behavior for most face topologies. The exceptions are eight- and nine-noded quadrilateral faces (e.g., the faces of higher-order hexahedron elements), which always integrate tractions with full-order Gauss quadrature. For all other face topologies, the default traction calculation method is NODAL_LUMPING: the traction evaluated at the nodal coordinates is multiplied by an area attributed to that node to get the nodal force contribution. This default can be replaced by full-order Gauss quadrature for any face topology by specifying GAUSS_QUADRATURE as the traction integration method.

7.6.2.7. Example

```
DEFINE DIRECTION Fdir WITH VECTOR -1 0 -1
```

```
BEGIN FUNCTION One
  TYPE = CONSTANT
  BEGIN VALUES
    1.0
  END VALUES
END FUNCTION One

BEGIN TRACTION
 SURFACE = sideset_1
 DIRECTION = Fdir
 FUNCTION = One
 SCALE FACTOR =1
END TRACTION
```

This `TRACTION` applies a force vector with a magnitude of 1.0 time by the area associated with each node in sideset_1 as shown in Figure 7-2. The force vectors are not required to be normal to the surface.

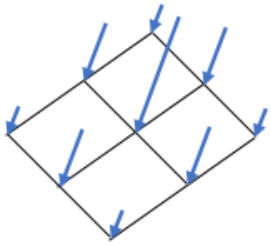


Figure 7-2. Result of a unit traction load

7.6.3. Prescribed Force

```
BEGIN PRESCRIBED FORCE
  #
  # node set commands
  NODE SET|NODESET = <string list>nodelist_names
  SURFACE|SIDESET|SIDE SET = <string list>surface_names
  BLOCK = <string list>block_names
  RIGID BODY = <string list>rigid_body_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # specification commands
  DIRECTION = <string>defined_direction |
    COMPONENT = <string>X|Y|Z
  FUNCTION = <string>function_name
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
  # external force sources
  READ VARIABLE = <string>var_name
  COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
    MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
  TIME = <real>time|FIRST|LAST
  OBJECT TYPE = <string>FACE|NODE(FACE)
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [PRESCRIBED FORCE]
```

The `PRESCRIBED FORCE` command block prescribes a force field for a given set of nodes. The force field associates a vector giving the magnitude and direction of the force with each node in the node set. The force field may vary over time and space. If the force field has only a time-varying component, the specification commands in the above command block can be used to specify the force field. If the force field has both time-varying and spatially varying components,

a user subroutine is used to specify the force field. In a given boundary condition command block, commands from only one of the command groups (specification commands or user subroutine commands) may be used.

The PRESCRIBED FORCE command block contains three groups of commands—node set, function, and user subroutine. Each of these command groups is basically independent of the others. In addition to the command lines in the three command groups, there are three additional command lines: SCALE FACTOR, ACTIVE PERIODS, and INACTIVE PERIODS. The SCALE FACTOR command line can be used in conjunction with either the specification commands or the user subroutine commands. The ACTIVE PERIODS and INACTIVE PERIODS command lines are used to activate or deactivate this force boundary condition for certain time periods. Following are descriptions of the different command groups.

7.6.3.1. Node Set Commands

The node set commands portion of the PRESCRIBED FORCE command block defines a set of nodes associated with the prescribed force field and can include some combination of the following command lines:

```
NODE SET|NODESET = <string list>nodelist_names
SURFACE|SIDESET|SIDE SET = <string list>surface_names
BLOCK = <string list>block_names
RIGID BODY = <string list>rigid_body_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 7.1.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one NODE SET|NODESET, SURFACE, BLOCK, or INCLUDE ALL BLOCKS command line in the command block.

7.6.3.2. Specification Commands

If the specification commands are used, the force vector at any given time is the same for all nodes in the node set associated with the particular PRESCRIBED FORCE command block. The direction of the force vector is constant for all time; the magnitude of the force vector may vary with time, however.

Following are the command lines used to specify the prescribe force with a direction and a function:

```
DIRECTION = <string>defined_direction |
```

```
  COMPONENT = <string>X|Y|Z
FUNCTION = <string>function_name
```

The force can be specified either along an arbitrary user-defined direction or along a component direction (X, Y, or Z), but not both.

- The `DIRECTION` command line is used to prescribe force in an arbitrary user-defined direction. The name in the string `defined_direction` is a reference to a direction, which is defined using the `DEFINE DIRECTION` command block within the SIERRA scope.
- The `COMPONENT` command line specifies that the force vector lies along one of the component directions. The `COMPONENT` command line is a shortcut to an internally defined direction vector; for example, component `x` corresponds to using direction vector (1, 0, 0).

The magnitude of the force is specified by the `FUNCTION` command line. This references a `function_name` (defined in the SIERRA scope in a `FUNCTION` command block) that specifies the magnitude of the force vector as a function of time. The magnitude can be scaled by use of the `SCALE FACTOR` command line described in Section 7.6.3.5.

The force is applied only in the prescribed direction, and is not applied in any direction orthogonal to that direction.

7.6.3.3. User Subroutine Commands

If the user subroutine option is used, the force vector may vary spatially at any given time for each of the nodes in the node set associated with the particular `PRESCRIBED FORCE` command block. The user subroutine option allows for a more complex description of the force field than does the function option, but also requires writing a user subroutine to implement this capability. The user subroutine will be used to define a force direction and a magnitude for every node to which the boundary condition will be applied. The subroutine will be called by Sierra/SM at the appropriate time to generate the force field.

Following are the command lines related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the `NODE SET SUBROUTINE` command line. The string `subroutine_name` is the name of a FORTRAN subroutine that is written by the user.

Following the `NODE SET SUBROUTINE` command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 10.2.2

and consist of SUBROUTINE DEBUGGING OFF, SUBROUTINE DEBUGGING ON, SUBROUTINE REAL PARAMETER, SUBROUTINE INTEGER PARAMETER, and SUBROUTINE STRING PARAMETER. Examples of using these command lines are provided throughout Chapter 10.

The magnitude set in the user subroutine can be scaled by use of the SCALE FACTOR command line, as described in Section 7.6.3.5.

7.6.3.3.1. Usage requirements for the node set subroutine. The provided subroutine will return three output values per node. Suppose a user subroutine is provided that passes the output values through an array output_values. For a given node inode, the output_values array would have the following values:

```
output_values(1,inode) = x component of force at inode
output_values(2,inode) = y component of force at inode
output_values(3,inode) = z component of force at inode
```

The three components of the force vector are given in output_values 1 through 3. The values of the flags array are ignored.

See Chapter 10 for more details on implementing the user subroutine option.

7.6.3.4. External Mesh Database Commands

The prescribed force may be defined by a field on the mesh database. When using a mesh database defined force the magnitude and/or direction is read from a finite element model defined via the FINITE ELEMENT MODEL command block described in Section 6.1. The finite element model is the model used by the region for its mesh definition as specified with the USE FINITE ELEMENT MODEL command (see Section 2.3).

```
READ VARIABLE = <string>var_name
COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
TIME = <real>time|FIRST|LAST
OBJECT TYPE = <string>FACE|NODE(FACE)
```

See Section 7.3.5 for more details.

The OBJECT TYPE command line specifies whether the traction field on the mesh file is specified for nodes (the mesh object type is NODE) or for faces (the mesh object type is FACE). If the OBJECT TYPE command line is not present, it is assumed that the named variable is defined on faces.

The READ VARIABLE command gives the mesh file variable which defines the force. There are two ways to define this force field: the first, is to define a field with a single scalar value for each node or face and couple that with a DIRECTION command to define the direction of the force; the second, is to define a vector field with three components. In the latter case, the field can define a

unique traction magnitude and direction at each node or face. Furthermore, this option should not be used with the DIRECTION command.

The COPY VARIABLE command is a more flexible version of the READ VARIABLE command. This command allows a user to read a mesh variable from another mesh database and map it to another mesh by proximity or ID then use it to apply the force.

If present, the TIME command defines the time slice in the input mesh file from which to read the force field. If the TIME field is not present, the force will be defined by interpolation between time slices closest to the current analysis time. Specifying the keyword FIRST or LAST will read from the first or last time slice in the input mesh file, respectively.

7.6.3.5. Additional Commands

These command lines in the PRESCRIBED FORCE command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

The SCALE FACTOR command line is used to apply an additional scaling factor, which is constant in both time and space, to all vector magnitude values of the field defined by the specification commands or the user subroutine. For example, if the magnitude of the force in a time history function is given as 1.5 from time 1.0 to time 2.0 and the scale factor is 0.5, then the magnitude of the force from time 1.0 to 2.0 is 0.75. The default value for the scale factor is 1.0.

The ACTIVE PERIODS and INACTIVE PERIODS command lines determine when the boundary condition is active. See Section 2.6 for more information about these command lines.

7.6.3.6. Example

```
DEFINE DIRECTION Fdir WITH VECTOR -1 0 -1

BEGIN FUNCTION One
  TYPE = CONSTANT
  BEGIN VALUES
    1.0
  END VALUES
END FUNCTION One


BEGIN PRESCRIBED FORCE
 SURFACE = sideset_1
 DIRECTION = Fdir
 FUNCTION = One
```

```
 SCALE FACTOR =1
END PRESCRIBED FORCE
```

The resulting `PRESCRIBED FORCE` applies a force vector of magnitude 1.0 to each node in sideset_1 as shown in Figure 7-3.

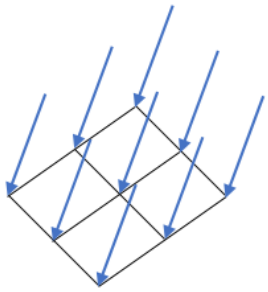


Figure 7-3. Result of a unit prescribed force

7.6.4. Prescribed Moment

```
BEGIN PRESCRIBED MOMENT
  #
  # node set commands
  NODE SET|NODESET = <string list>nodelist_names
  SURFACE|SIDESET|SIDE SET = <string list>surface_names
  BLOCK = <string list>block_names
  RIGID BODY = <string list>rigid_body_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # specification commands
  DIRECTION = <string>defined_direction |
    COMPONENT = <string>X|Y|Z
  FUNCTION = <string>function_name
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
    = <string>param_value
  #
```

```
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [PRESCRIBED MOMENT]
```

The PRESCRIBED MOMENT command block prescribes a moment field for a given set of nodes. Moments can only be defined for nodes with rotational degrees of freedom. This includes nodes attached to beam and shell elements, rigid body nodes, and potentially superelement nodes.

The moment field associates a vector giving the magnitude and direction of the moment with each node in the set. If the moment field has only a time-varying component, the specification commands in the above command block can be used to specify the moment field. If the moment field has both time-varying and spatially varying components, a user subroutine option is used to specify the moment field. In a given boundary condition command block, commands from only one of the command groups (specification commands or user subroutine commands) may be used.

The PRESCRIBED MOMENT command block contains three groups of commands—node set, function, and user subroutine. Each of these command groups is basically independent of the others. In addition to the command lines in the four command groups, there are three additional command lines: SCALE FACTOR, ACTIVE PERIODS, and INACTIVE PERIODS. The SCALE FACTOR command line can be used in conjunction with either the specification commands or the user subroutine commands. The ACTIVE PERIODS and INACTIVE PERIODS command lines are used to activate or deactivate this force boundary condition for certain time periods. Following are descriptions of the different command groups.

7.6.4.1. Node Set Commands

The node set commands portion of the PRESCRIBED MOMENT command block defines a set of nodes associated with the prescribed moment field and can include some combination of the following command lines:

```
NODE SET|NODESET = <string list>nodelist_names
SURFACE|SIDESET|SIDE SET = <string list>surface_names
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 7.1.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one NODE SET|NODESET, SURFACE, BLOCK, or INCLUDE ALL BLOCKS command line in the command block.

7.6.4.2. Specification Commands

If the specification commands are used, the moment vector at any given time is the same for all nodes in the node set associated with the particular `PRESCRIBED MOMENT` command block. The direction of the moment vector is constant for all time; the magnitude of the moment vector may vary with time, however.

Following are the command lines used to specify the prescribed moment with a function and a direction:

```
DIRECTION = <string>defined_direction |
  COMPONENT = <string>X|Y|Z
FUNCTION = <string>function_name
```

The moment can be specified either along an arbitrary user-defined direction or along a component direction (X, Y, or Z), but not both.

- The `DIRECTION` command line is used to prescribe the moment in an arbitrary user-defined direction. The name in the string `defined_direction` is a reference to a direction, which is defined using the `DEFINE DIRECTION` command block within the SIERRA scope.
- The `COMPONENT` command line specifies that the moment vector lies along one of the component directions. The `COMPONENT` command line is a shortcut to an internally defined direction vector; for example, component `x` corresponds to using direction vector (1, 0, 0).

The magnitude of the moment is specified by the `FUNCTION` command line. This references a `function_name` (defined in the SIERRA scope in a `FUNCTION` command block) that specifies the magnitude of the moment vector as a function of time. The magnitude can be scaled by use of the `SCALE FACTOR` command line described in Section 7.6.4.4.

The moment is applied only in the prescribed direction, and is not applied in any direction orthogonal to that direction.

7.6.4.3. User Subroutine Commands

If the user subroutine option is used, the moment vector may vary spatially at any given time for each of the nodes in the node set associated with the particular `PRESCRIBED MOMENT` command block. The user subroutine option allows for a more complex description of the moment field than do specification commands, but also requires writing a user subroutine to implement this capability. The user subroutine will be used to define a moment direction and a magnitude for every node to which the boundary condition will be applied. The subroutine will be called by Sierra/SM at the appropriate time to generate the moment field.

Following are the command lines related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
```

```
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the NODE SET SUBROUTINE command line. The string subroutine_name is the name of a FORTRAN subroutine that is written by the user.

Following the NODE SET SUBROUTINE command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 10.2.2 and consist of SUBROUTINE DEBUGGING OFF, SUBROUTINE DEBUGGING ON, SUBROUTINE REAL PARAMETER, SUBROUTINE INTEGER PARAMETER, and SUBROUTINE STRING PARAMETER. Examples of using these command lines are provided throughout Chapter 10.

The magnitude set in the user subroutine can be scaled by use of the SCALE FACTOR command line, as described in Section 7.6.4.4.

7.6.4.3.1. Usage requirements for the node set subroutine. The provided subroutine will return three output values per node. Suppose a provided user subroutine passes the output values through an array output_values. For a given node inode, the output_values array would have the following values:

```
output_values(1,inode) = moment about x-direction at inode
output_values(2,inode) = moment about y-direction at inode
output_values(3,inode) = moment about z-direction at inode
```

The three components of the moment vector are given in output_values 1 through 3. The values of the flags array are ignored.

See Chapter 10 for more details on implementing the user subroutine option.

7.6.4.4. Additional Commands

These command lines in the PRESCRIBED MOMENT command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

The SCALE FACTOR command line is used to apply an additional scaling factor, which is constant in both time and space, to all vector magnitude values of the field defined by the specification commands or the user subroutine. For example, if the magnitude of the moment in a time history

function is given as 1.5 from time 1.0 to time 2.0 and the scale factor is 0.5, then the magnitude of the moment from time 1.0 to 2.0 is 0.75. The default value for the scale factor is 1.0.

The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines determine when the boundary condition is active. See Section 2.6 for more information about these command lines.

7.7. Distributed Force and Moment

```
BEGIN DISTRIBUTED FORCE|MOMENT
  #
  # Node set commands
  NODE SET|NODESET = <string list>nodelist_names
  SURFACE|SIDESET|SIDE SET = <string list>surface_names
  BLOCK = <string list>block_names
  ELEMENT = <int list>elem_numbers
  NODE = <int list>node_numbers
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # Load definition commands
  SCALE FACTOR = <real>scale_factor(1.0)
  FUNCTION = <string>function_name
  DIRECTION = <string>defined_direction |
    !COMPONENT = <string>X|Y|Z
  COORDINATE SYSTEM = <string>sys_name(global)
  #
  # additional commands
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [DISTRIBUTED FORCE|MOMENT]
```

The distributed force and moment boundary conditions are used to apply a specific total net force or moment that is automatically distributed to a finite set of nodes. The distributed force and moment boundary conditions differ from the prescribed force 7.6.3 and moment 7.6.4 boundary conditions. The previously described prescribed force boundary conditions apply user defined point loads to each and every node used by the boundary condition. The distributed force BC instead spreads a single user defined target load among every node in the set. The distributed moments are applied via translational force couples and thus can be used on any element type. This differs from the prescribed point load moments 7.6.4 which can only be used on elements with rotational degrees of freedom such as shells.

The distributed load boundary conditions are targeted to the following types of use cases.

- Apply a moment to a solid meshed body, for example torque on a shaft or bending moment on the end of a solid meshed bar.
- Apply an exact known force to a model boundary. For example to model a load-control tension test.
- Apply a load to known load to a model distributed over a finite volume. An analysis with this distributed load will converge with mesh refinement. An analysis with a concentrated load applied to a single point will generate a singularity at that point and not converge with mesh refinement.

7.7.0.1. Node Set Commands

The `Node set commands` portion of the `DISTRIBUTED FORCE` and `DISTRIBUTED MOMENT` command blocks define a set of nodes associated with the boundary condition. These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 7.1.1 for more information about the use of these command lines. The boundary condition must contain at least one `NODE SET|NODESET`, `NODE`, `SURFACE`, `BLOCK`, `ELEMENT`, or `INCLUDE ALL BLOCKS` command line. Note, the `RIGID BODY` command is not available with this boundary condition.

7.7.0.2. Load Definition Commands

The load commands define the magnitude and direction of the force or moment applied by the distributed load boundary condition. The load magnitude is defined by the time history function `function_name` multiplied by the value of `scale_factor`. Each distributed load BC takes a single load magnitude, the `FUNCTION` command by default takes a function of time and may also take a analytic function that involves other global variables. One and only one of the `COMPONENT` command line or `DIRECTION` command line must be given to define the load direction. For the distributed force BC the load will be in the given direction. For the distributed moment BC the load will be around the given direction using the right hand rule. If no coordinate system is given the direction or component refer to the global XYZ coordinate system. If an optional coordinate system is specified the directions and components are given with respect to that coordinate system. Note if the coordinate system is dynamic (linked to coordinates of node sets) the direction of distributed load application will update continuously as the model evolves.

7.7.0.3. Miscellaneous Commands

The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines provides an additional option for the distributed load BC. These command lines can activate or deactivate the load for certain time periods. See Section 2.6 for more information about these command lines.

7.7.0.4. Restrictions and Assumptions

The distributed load boundary conditions are applied by defining a set of translational forces on the nodes in the application region. Generally forces will be mass proportional. Example force distributions for Z force and Z moment are shown in Figure 7-4 and 7-5. Force distributions are constructed to apply net load in only a single component. Thus the force distribution shown in Figure 7-4 produces no net moment on the surface and no net Force in the X and Y directions. The moment distribution shown in Figure 7-5 produces no net force on the surface nor any net moment around the X or Y axes.

As moments are constructed via translational force couples no net moment can be applied to a single node. Additionally if all the applied nodes fall nearly in a line no net moment can be applied around that line. If these degenerate cases occur the inapplicable component of the net

moment will be zero. For node sets that are close to collinear potentially unphysical large forces may be required to apply due to the small axial moment arm. When using the distributed moment BC generally the application region should be a fairly large set of nodes on a block or surface to prevent these excessive force concentrations.

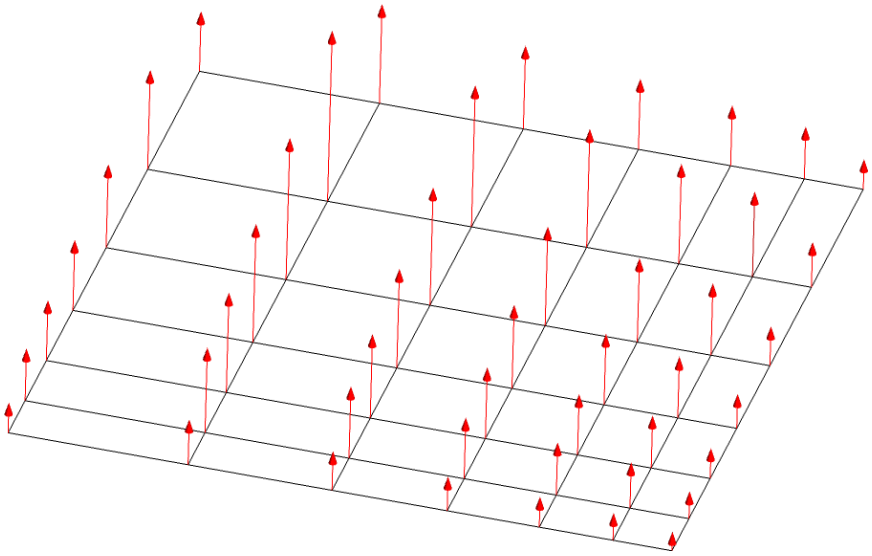


Figure 7-4. Distributed Z Force Load Distribution

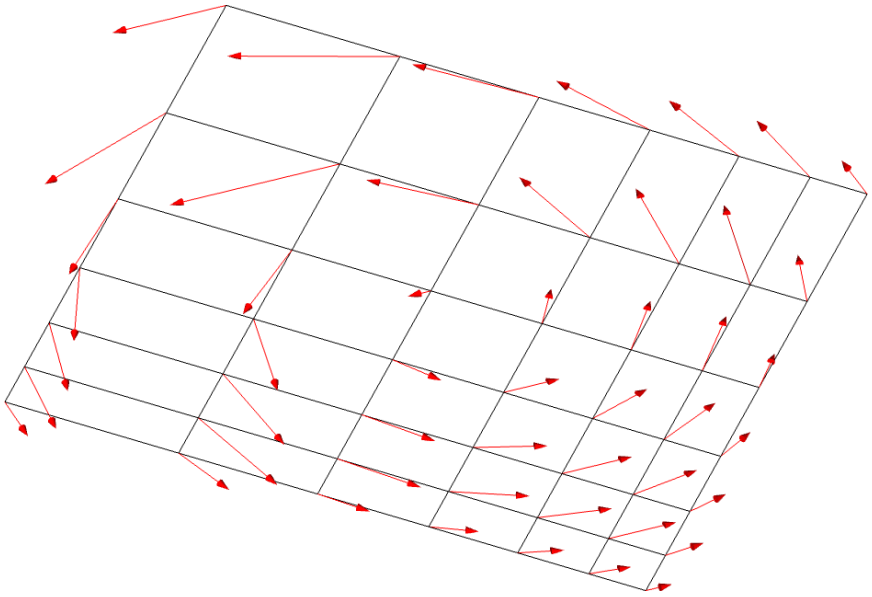


Figure 7-5. Distributed Z Moment Load Distribution

7.7.0.5. Example: Applying Torque to Solid Bar

The following example for applying torque around an arbitrarily oriented shaft shown in Figure 7-6. The coordinate system `rod_1` is linked to nodes on the body and will update as the rod moves so that the torque is always applied around the rod axis.

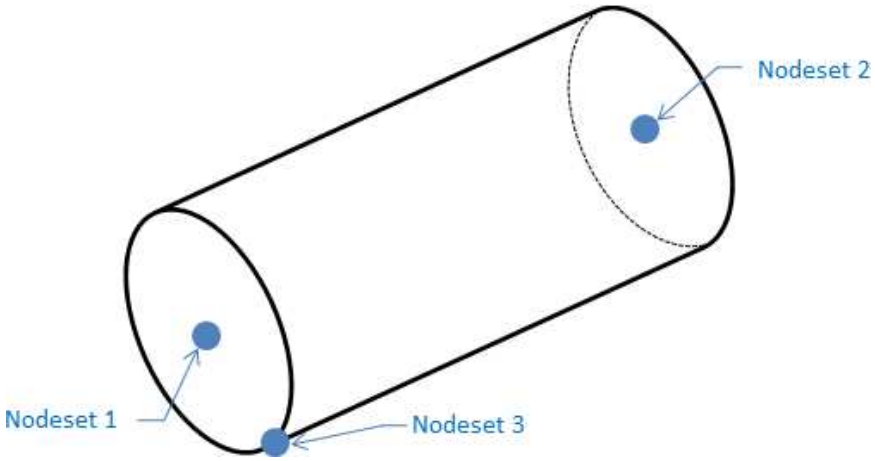


Figure 7-6. Distributed Load Example

```
#  Define a body fitted coordinate system on the rod
begin coordinate system rod_1
  type   = rectangular
  # Base of Cylinder
  origin node = nodelist_1
  # Vector from origin to vector nodes defines Z axis
  vector node = nodelist_2
  # A point on the XY plane
  point  node = nodelist_3
end

#  Ramp up loading function
begin function ramp
  type is piecewise linear
   begin values
     0.00 0.0
     0.01 1.0
   end values
end

#  Apply a distributed load to the rod block, around the
#  axis of the cylindrical rod.
begin distributed moment
  block = block_1
  coordinate system = rod_1
  function = ramp
  # Z coordinate direction of coordinate system lies along
  # the rod axis
  component = z
end
```

7.8. Gravity

```
BEGIN GRAVITY
  #
  # node set commands
  NODE SET|NODESET = <string list>nodelist_names
  SURFACE|SIDESET|SIDE SET = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  DIRECTION = <string>defined_direction
  COMPONENT = <string>X|Y|Z

  FUNCTION = <string>function_name (SIERRA_CONSTANT_FUNCTION_ONE)
  GRAVITATIONAL CONSTANT = <real>g_constant(1.0)
  SCALE FACTOR = <real>scale_factor(1.0)
  EXTERNAL FORCE CONTRIBUTION OUTPUT NAME =
    <string>variable_name
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [GRAVITY]
```

A gravity load is generally called a body force boundary condition. A gravity load generates a force at a node that is proportional to the mass of the node. This section describes how to apply a gravity load to a body.

The `GRAVITY` command block specifies a gravity load that is applied to all nodes selected within a command block. The gravity load boundary condition uses the function and scale (gravitational constant and scale factor) information to generate a body force at a node based on the mass of the node. Multiple `GRAVITY` command blocks can be defined on different sets of nodes. If two different `GRAVITY` command blocks reference the same node, the node will have gravity loads applied by both of the command blocks. Care must be taken to make sure multiple gravity loads are not applied to one block if only one gravity load condition is desired.

The `node set commands` portion of the `GRAVITY` command block defines a set of nodes associated with the gravity load and can include some combination of the following command lines:

```
NODE SET|NODESET = <string list>nodelist_names
SURFACE|SIDESET|SIDE SET = <string list>surface_names
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 7.1.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `NODE SET|NODESET`, `SURFACE`, `BLOCK`, or `INCLUDE ALL BLOCKS` command line in the command block.

The gravity load is specified along an arbitrary user-defined direction, and is defined using the `DIRECTION` command line. The name in the string `defined_direction` is a reference to a direction, which is defined using the `DEFINE DIRECTION` command block within the SIERRA scope. Alternatively the gravity can be applied along the `X`, `Y`, or `z COMPONENT` directions.

The strength of the gravitational field can be varied with time by using the `FUNCTION` command line. This command line references a `function_name` defined in the SIERRA scope in a `FUNCTION` command block. The gravity force is constant in time if the `FUNCTION` command line is not specified.

A gravitational constant is specified by the `GRAVITATIONAL CONSTANT` command line in the real value `g_constant`. For example, the gravitational constant in units of inches and seconds would be 386.4 inches per second squared. This quantity must be set based on the actual units for the model.

The Gravity that will be applied will be a multiplication of `SCALE FACTOR`, `GRAVITATIONAL CONSTANT`, and `FUNCTION` (G = SF*GC*F). For example, if `GRAVITATIONAL CONSTANT` = 2, `SCALE FACTOR` = 2, and `FUNCTION` has a constant value of 2 then you will get a constant gravity of 8.

The dependent variables in the function can be scaled by the real value `scale_factor` in the `SCALE FACTOR` command line. At any given time, the strength of the gravitational field is a product of the gravitational constant, the value of the function at that time, and the scale factor.

If the `EXTERNAL FORCE CONTRIBUTION OUTPUT NAME` command line appears there will be a variable created with whatever name the user specifies for `variable_name`. The variable defines a three-dimensional vector at each node associated with this particular command block. The three-dimensional vector at each node represents the external force due solely to just this particular external force boundary contribution.

Once this variable for the external force contribution is specified, it may be used like any other nodal variable. The user can, for example, specify the variable as a nodal variable to be output in a `RESULTS OUTPUT` command block. Or the user can reference the variable in a user subroutine.

The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines provides an additional option for the gravity load condition. These command lines can activate or deactivate the gravity load for certain time periods. See Section 2.6 for more information about these command lines.

7.9. Centripetal Force

```
BEGIN CENTRIPETAL FORCE
  NODE SET|NODESET = <string list>nodelist_names
  SURFACE|SIDESET|SIDE SET = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  CYLINDRICAL AXIS = <string>cylindrical_axis
  ROTATIONAL VELOCITY FUNCTION = <string>rotational_vel_name
  ROTATIONAL VELOCITY SCALE FACTOR = <real>rvsf(1.0)
  FORCE SCALE FACTOR = <real>fsf(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [CENTRIPETAL FORCE]
```

The centripetal force boundary condition adds an external force to the body that is equivalent to the centripetal force on the body rotating at the defined rotational velocity. The centripetal force boundary condition can be used, for example, in a static analysis to calculate the deformation of a body spinning with constant rotational velocity around some axis.

The centripetal force boundary condition will be applied to all nodes selected within the command block. The equation for the centripetal force on a node is given by Equation 7.4, where F is the magnitude of the centripetal force, m is the mass of the node, R is the current distance from the node to the rotational axis, and ω is the rotational velocity of the node about the rotational axis.

$$F = mR\omega^2 \tag{7.4}$$

The `CYLINDRICAL AXIS` command defines the name of the axis of rotation. The name given is the name of a previously defined axis created with the `DEFINE AXIS` command. The rotational velocity as a function of time is defined by multiplying the function named by the `ROTATIONAL VELOCITY FUNCTION` by the `ROTATIONAL VELOCITY SCALE FACTOR`. Additionally a scale factor on the final return force may be specified with he `FORCE SCALE FACTOR` command.

7.10. Initial Temperature

```
BEGIN INITIAL TEMPERATURE
  #
  # block set commands
  INCLUDE ALL BLOCKS
  BLOCK = <string list>block_names
  REMOVE BLOCK = <string list>block_names
  ELEMENT = <int_list>elem_ids

  NODE SET|NODESET = <string list>nodeset_names
  REMOVE NODE SET = <string list>nodeset_names
  NODE = <int_list>node_ids
  #
  # specification command
  MAGNITUDE = <real>initial_temperature_value
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
  #
  # external database commands
  READ VARIABLE = <string>var_name
  COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
    MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
  TIME = <real>time|FIRST|LAST
END [PRESCRIBED TEMPERATURE]
```

The `INITIAL TEMPERATURE` command block initializes a temperature field on the prescribed mesh entities. The temperature field is set on a nodal field `temperature` and interpolated to the integration points of an element field `temperature`.

The temperature field may evolve later through interaction with material models that support temperature evolution features such as adiabatic heating. In this case, only the element temperature field is changed and will become out-of-sync with the nodal temperature field.

The simplest example for initializing temperature is as follows:

```
begin initial temperature
    include all blocks
    magnitude = 294.0
end
```

The initial temperature block may appear multiple times. In the case where temperature has already been prescribed on one node (e.g. two merged blocks with different temperatures), the last initial temperature specification wins.

A `PRESCRIBED TEMPERATURE` command block may exist within the same scope as an `INITIAL TEMPERATURE` command block. It is possible that the prescribed temperature at the initial analysis time conflicts with what is specified in the `INITIAL TEMPERATURE` block. In this case, the `PRESCRIBED TEMPERATURE` block wins and will subsequently provide a temperature at every time step.

7.10.1. Block Set Commands

The `block set commands` portion of the `INITIAL TEMPERATURE` command block defines a set of nodes associated with the prescribed temperature field and can include some combination of the following command lines:

```
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE BLOCK = <string list>block_names
ELEMENT = <int_list>elem_ids
NODE SET|NODESET = <string list>nodeset_names
REMOVE NODE SET = <string list>nodeset_names
NODE = <int_list>node_ids
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes derived from some combination of element blocks. See Section 7.1.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `BLOCK`, `INCLUDE ALL BLOCKS`, `ELEMENT`, `NODE SET|NODESET`, or `NODE` command line in the command block.

7.10.2. Specification Commands

The `MAGNITUDE` command may be used to specify a constant temperature for the selected mesh entities.

The initial temperature field may also vary over space by using either the user subroutine commands, or the external mesh database commands. For example, the exodus results database of a SIERRA/TF analysis may be used to initialize a SIERRA/SM analysis. The usage details for this follow directly from the description in Section 7.11.

7.11. Prescribed Temperature

```
BEGIN PRESCRIBED TEMPERATURE
  #
  # block set commands
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE BLOCK = <string list>block_names
  ELEMENT = <int_list>elem_ids

  NODE SET|NODESET = <string list>nodeset_names
  REMOVE NODE SET = <string list>nodeset_names
  NODE = <int_list>node_ids
  #
  # specification command
  FUNCTION = <string>function_name
  #
  # user subroutine commands
  NODE SET SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
  # external database commands
  READ VARIABLE = <string>var_name
  COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
    MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
  TIME = <real>time|FIRST|LAST
  TEMPERATURE TYPE = SOLID_ELEMENT|SHELL_ELEMENT (SOLID_ELEMENT)
  #
  # coupled analysis commands
  RECEIVE FROM TRANSFER [FIELD TYPE = NODE|ELEMENT(NODE)]
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [PRESCRIBED TEMPERATURE]
```

The `PRESCRIBED TEMPERATURE` command block prescribes a temperature field for a given set of nodes. The prescribed temperature is for each node in the node set. The temperature field may vary over time and space. If the temperature field has only a time-varying component, the function command in the above command block can be used to specify the temperature field. If the temperature field has both time-varying and spatially varying components, a user subroutine option can be used to specify the temperature field. Finally, the temperature may also be read as a

variable from the mesh file. Only one of these options—function, user subroutine, or read variable—may be specified in a command block.

Temperature is applied to nodes, but it is frequently used at the element level, such as in the case for thermal strains. If the temperatures are used at the element level, the nodal values are averaged (depending on element) connectivity to produce an element temperature. The temperatures must be defined for all the nodes defining the connectivity for any given element. For this reason, we use block commands to derive a set of nodes at which to define temperatures. If the temperatures are used on an element basis, then the temperature at all the necessary nodes will be defined.

The `PRESCRIBED TEMPERATURE` command block contains four groups of commands—block set, function, user subroutine, and read variable. Each of these command groups is basically independent of the others. In addition to the command lines in the four command groups, there are three additional command lines: `SCALE FACTOR`, `ACTIVE PERIODS`, and `INACTIVE PERIODS`. The `SCALE FACTOR` command line can be used in conjunction with the function command, the user subroutine option, or the read variable option. The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines are used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups.

7.11.1. Block Set Commands

The `block set commands` portion of the `PRESCRIBED TEMPERATURE` command block defines a set of nodes associated with the prescribed temperature field and can include some combination of the following command lines:

```
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE BLOCK = <string list>block_names
ELEMENT = <int_list>elem_ids
NODE SET|NODESET = <string list>nodeset_names
REMOVE NODE SET = <string list>nodeset_names
NODE = <int_list>node_ids
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes derived from some combination of element blocks. See Section 7.1.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `BLOCK`, `INCLUDE ALL BLOCKS`, `ELEMENT`, `NODE SET|NODESET`, or `NODE` command line in the command block.

7.11.2. Specification Command

If the function command is used, the temperature at any given time is the same for all nodes in the node set associated with the particular `PRESCRIBED TEMPERATURE` command block. The command line

```
FUNCTION = <string>function_name
```

references a `function_name` (defined in the SIERRA scope using a `FUNCTION` command block) that specifies the temperature as a function of time. The temperature can be scaled by use of the `SCALE FACTOR` command line described in Section 7.11.6.

7.11.3. User Subroutine Commands

If the user subroutine option is used, the temperature field may vary spatially at any given time for each of the nodes in the node set associated with the particular `PRESCRIBED TEMPERATURE` command block. The user subroutine option allows for a more complex description of the temperature field than does the function command, but also requires writing a user subroutine to implement this capability. The user subroutine will be used to define a temperature for every node to which the boundary condition will be applied. The subroutine will be called by Sierra/SM at the appropriate time to generate the temperature field.

Following are the command lines related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the `NODE SET SUBROUTINE` command line. The string `subroutine_name` is the name of a FORTRAN subroutine that is written by the user.

Several other commands control the behavior of user subroutines:
`SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`,
`SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and
`SUBROUTINE STRING PARAMETER`. These are described in Section 10.2.2. Examples of using these command lines are provided throughout Chapter 10.

The temperature set in the user subroutine can be scaled by use of the `SCALE FACTOR` command line, as described in Section 7.11.6.

See Chapter 10 for more details on implementing the user subroutine option.

7.11.4. External Mesh Database Commands

```
READ VARIABLE = <string>var_name
COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
  MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
```

```
TIME = <real>time|FIRST|LAST
TEMPERATURE TYPE = SOLID_ELEMENT|SHELL_ELEMENT (SOLID_ELEMENT)
```

See Section 7.3.5 for more details.

If temperature is to be prescribed for shell elements, a linear or quadratic thermal gradient can optionally be specified through the thickness of the shells using the `TEMPERATURE TYPE` command line. The `SOLID_ELEMENT` option to this command is the default behavior, and results in a single temperature being read for each node into the `temperature` variable. With the `SHELL_ELEMENT` option, temperatures are read into the `shell_temperature` variable. Shell elements may potentially define a temperature gradient though the thickness, in which case there will be multiple temperatures at a node to describe the temperature gradient through the shell. The `TEMPERATURE_TYPE` command is only valid in conjunction with the `READ VARIABLE` command.

Though-thickness shell temperatures follow the Aria convention. If there is a single shell temperature defined at a node, the temperature is constant through the thickness.

If there are two shell temperatures defined at a node, the first is the temperature on the bottom of the shell and the second the temperature at the top. The temperature varies linearly between the top and bottom.

If there are three shell temperatures defined at node, the first is the temperature at the bottom of the shell, the second the temperature at the middle of the shell, and the third the temperature at the top of the shell. The temperature varies quadratically through the thickness.

`SOLID_ELEMENT` and `SHELL_ELEMENT` temperatures may be defined simultaneously in the same analysis through two different temperature command blocks. If both are defined, the shell element temperature results override any solid element temperature results on the shell elements.

7.11.5. Coupled Analysis Commands

The `RECEIVE FROM TRANSFER` command provides the ability to set the temperature in a coupled analysis by transferring results from another SIERRA code, such as Aria.

```
RECEIVE FROM TRANSFER [FIELD TYPE = NODE|ELEMENT(NODE)]
```

If this command is used in its default form, it is expected that the temperature is transferred to a nodal field named `temperature` in the Sierra/SM region. Sierra/SM performs an interpolation from the nodal field to an element field named `temperature`. The temperature can also be transferred directly to the element `temperature` field by using the optional `FIELD TYPE = ELEMENT` argument to this command.

If the `RECEIVE FROM TRANSFER` command is used, but the appropriate commands to perform the transfer between the two regions are missing, the temperature will be zero during the entire simulation.

It is also possible to use this line command to cause the initial temperatures obtained from a restart file to remain constant in time.

7.11.6. Additional Commands

These command lines in the PRESCRIBED TEMPERATURE command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

The SCALE FACTOR command line is used to apply an additional scaling factor, which is constant in both time and space, to all temperature values of the field defined by the function command, the user subroutine, or the read variable option. For example, if the temperature in a time history function is given as 100.5 from time 1.0 to time 2.0 and the scale factor is 0.5, then the temperature from time 1.0 to 2.0 is 50.25. The default value for the scale factor is 1.0.

The ACTIVE PERIODS and INACTIVE PERIODS command lines determine when the boundary condition is active. See Section 2.6 for more information about these command lines.

7.12. Pore Pressure

```
BEGIN PORE PRESSURE
  #
  # block set commands
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE BLOCK = <string list>block_names
  #
  # specification command
  FUNCTION = <string>function_name
  #
  # user subroutine commands
  ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
    = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
    = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
  # external database commands
  READ VARIABLE = <string>var_name
  COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
    MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
  TIME = <real>time|FIRST|LAST
  #
  # coupled analysis commands
  RECEIVE FROM TRANSFER [FIELD TYPE = NODE|ELEMENT(NODE)]
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [PORE PRESSURE]
```

The `PORE PRESSURE` command block prescribes a pore pressure field for a given set of elements. The pore pressure is prescribed for each element in the block. The pore pressure field may vary over time and space. If the pore pressure field has only a time-varying component, the function command in the above command block can be used to specify the pore pressure field. If the pore pressure field has both time-varying and spatially varying components, a user subroutine option can be used to specify the pore pressure field. Finally, the pore pressure can be read as a variable from the mesh file. Only one of these options—function, user subroutine, or read variable—may be specified in a command block.

The `PORE PRESSURE` command block contains four groups of commands—block set, function, user subroutine, and read variable. Each of these command groups is basically independent of the others. In addition to the command lines in the four command groups, there are three additional

command lines: SCALE FACTOR, ACTIVE PERIODS, and INACTIVE PERIODS. The SCALE FACTOR command line can be used in conjunction with the function command, the user subroutine option, or the read variable option. The ACTIVE PERIODS and INACTIVE PERIODS command lines are used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups.

Biot's coefficient can be defined when prescribing pore pressure. See Section 5.1.2 for more information on Biot's coefficient.

7.12.1. Block Set Commands

The block set commands portion of the PORE PRESSURE command block defines a set of elements associated with the pore pressure field and can include a combination of the following command lines:

```
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of elements derived from some combination of element blocks. See Section 7.1.1 for more information about the use of these command lines for creating a set of elements used by the boundary condition. There must be at least one BLOCK or INCLUDE ALL BLOCKS command line in the command block.

7.12.2. Specification Command

If the FUNCTION command is used, the pore pressure at any given time is the same for all elements in the element set associated with the particular PORE PRESSURE command block. The command line

```
FUNCTION = <string>function_name
```

references a function_name (defined in the SIERRA scope using a FUNCTION command block) that specifies the pore pressure as a function of time. The pore pressure can be scaled using the SCALE FACTOR command line described in Section 7.12.6.

7.12.3. User Subroutine Commands

If the user subroutine option is used, the pore pressure field may vary spatially at any given time for each of the elements in the element set associated with the particular PORE PRESSURE command block. The user subroutine option allows for a more complex description of the pore pressure field than does the FUNCTION command, but the user subroutine option also requires that

a user subroutine be written to implement this capability. The user subroutine will be used to define a pore pressure for every element to which the boundary condition will be applied. The subroutine will be called by Sierra/SM at the appropriate time to generate the pore pressure field.

Following are the command lines related to the user subroutine option:

```
ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the `ELEMENT BLOCK SUBROUTINE` command line. The string `subroutine_name` is the name of a FORTRAN subroutine that is written by the user.

Several other commands control the behavior of user subroutines:
`SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`,
`SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and
`SUBROUTINE STRING PARAMETER`. These are described in Section 10.2.2. Examples of using these command lines are provided throughout Chapter 10.

The pore pressure set in the user subroutine can be scaled by use of the `SCALE FACTOR` command line, as described in Section 7.12.6.

See Chapter 10 for more details on implementing the user subroutine option.

7.12.4. External Mesh Database Commands

The pore pressure field can be read from an external mesh database. The finite element model from which pore pressures are read is defined via the `FINITE ELEMENT MODEL` command block described in Section 6.1. The finite element model can either be the model used by the region for its mesh definition as specified with the `USE FINITE ELEMENT MODEL` command (see Section 2.3), or it can be a different (but compatible) model. The following command lines control the use of an external mesh database to prescribe the pore pressure:

```
READ VARIABLE = <string>var_name
COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
  MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
TIME = <real>time|FIRST|LAST
```

See Section 7.3.5 for more details.

7.12.5. Coupled Analysis Commands

The RECEIVE FROM TRANSFER command provides the ability to set the pore pressure in a coupled analysis by transferring results from another SIERRA code, such as Aria.

```
RECEIVE FROM TRANSFER [FIELD TYPE = NODE|ELEMENT(NODE)]
```

If this command is used in its default form, it is expected that the pore pressure is transferred to a nodal field named pore_pressure in the Sierra/SM region. Sierra/SM performs an interpolation from the nodal field to an element field named pore_pressure. The pore pressure can also be transferred directly to the element pore_pressure field by using the optional FIELD TYPE = ELEMENT argument to this command.

If the RECEIVE FROM TRANSFER command is used, but the appropriate commands to perform the transfer between the two regions are missing, the pore pressure will be zero during the entire simulation.

7.12.6. Additional Commands

These command lines in the PORE PRESSURE command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

The SCALE FACTOR command line is used to apply an additional scaling factor, which is constant in both time and space, to all pore pressure values of the field defined by the function command, the user subroutine, or the read variable option. For example, if the pore pressure in a time history function is given as 100.5 from time 1.0 to time 2.0 and the scale factor is 0.5, then the pore pressure from time 1.0 to 2.0 is 50.25. The default value for the scale factor is 1.0.

The ACTIVE PERIODS and INACTIVE PERIODS command lines determine when the boundary condition is active. See Section 2.6 for more information about these command lines.

7.13. Fluid Pressure

```
BEGIN FLUID PRESSURE
  #
  # surface set commands
  SURFACE|SIDESET|SIDE SET = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  #
  # specification commands
  DENSITY = <real>fluid_density
  DENSITY FUNCTION = <string>density_function_name
  GRAVITATIONAL CONSTANT = <real>gravitational_acceleration
  FLUID SURFACE NORMAL = <string>direction_name
  DEPTH = <real>fluid_depth
  DEPTH FUNCTION = <string>depth_function_name
  REFERENCE POINT = <string>reference_point_name
  #
  # additional commands
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
  EXTERNAL FORCE CONTRIBUTION NAME = <string>varName
END [FLUID PRESSURE]
```

The `FLUID PRESSURE` command block applies a hydrostatic pressure to each node of each face in the associated surfaces. The pressure at any node is determined from

$$P = \rho g h \tag{7.5}$$

where P is the pressure, ρ is the fluid density at the current time, g is the gravitational constant, and h is the current depth of the fluid above the node. The depth of the fluid is computed as the distance from the node up to the current fluid surface in the direction of the fluid surface normal. The global location of the fluid surface is found by adding the current depth to the projection of a datum point along the direction defined in the `FLUID SURFACE NORMAL` command. The datum point is either the point specified in the `REFERENCE POINT` line command, or, in the absence of this command, the minimum point on the applied pressure surface in the direction defined in the `FLUID SURFACE NORMAL` command. Once the current location of the fluid surface is computed, the depth at each node on the pressure surface is computed as the distance from the node to the fluid surface in the direction of the fluid surface normal. See Figure 7-7.

Currently, the `FLUID PRESSURE` command block can be used for surfaces that have faces derived from solid elements (eight-node hexahedra, four-node tetrahedra, eight-node tetrahedra, etc.), membranes, and shells.

The `FLUID PRESSURE` command block contains three groups of commands—surface set, specification, and additional optional commands.

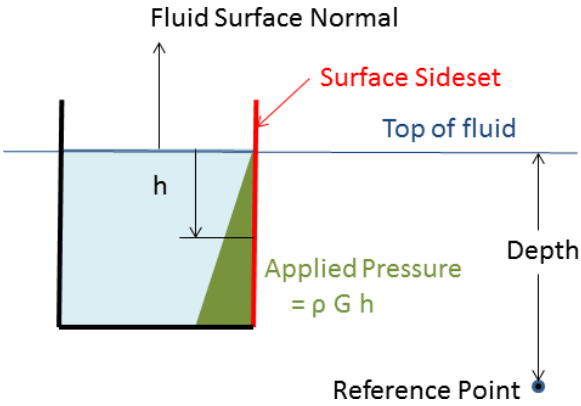


Figure 7-7. Fluid Pressure Setup

7.13.1. Surface Set Commands

The `surface set commands` portion of the `FLUID PRESSURE` command block defines a set of surfaces associated with the pressure field. The surface specification commands work the same as for the basic pressure boundary condition as described in section 7.6.1.1.

The force computed from the hydrostatic pressure will be in the opposite direction of the face normal. When using shells or membranes, the analyst must ensure that all face normals composing the pressure application surface are in the correct direction.

7.13.2. Specification Commands

The density and the gravitational acceleration must be input by the user in units consistent with other material properties and the lengths in the mesh. To facilitate convergence of the initial load step, the gradual application of a hydrostatic load may be specified through a time history function for the density. A combination of the `DENSITY` and the `DENSITY FUNCTION` sets the value for the fluid density at each time. Either the `DENSITY` or the `DENSITY FUNCTION` must be input and both can be used together. If the `DENSITY` command is input without the `DENSITY FUNCTION` command, the density will be constant in time at that value. If the `DENSITY FUNCTION` command is input without a `DENSITY` command, the function is used as a time history of the density. If both the `DENSITY` and the `DENSITY FUNCTION` commands are input, the density value is used as a scale factor on the time history function. Finally, the `GRAVITATIONAL CONSTANT` sets the value for the acceleration due to gravity, g.

```
DENSITY = <real>fluid_density
DENSITY FUNCTION = <string>fluid_density_function
GRAVITATIONAL CONSTANT = <real>G
```

The following command lines are used to define the location of the fluid surface at any time during the analysis:

```
FLUID SURFACE NORMAL = <string>normal_direction
```

```
DEPTH = <real>initial_fluid_depth
DEPTH FUNCTION = <string>depth_function_name
REFERENCE POINT = <string>point_name
```

The FLUID SURFACE NORMAL command sets the outward normal of the fluid surface. The fluid depth is then assumed to be in the direction opposite this global direction. The DEPTH command is used with the DEPTH FUNCTION command to determine the fluid depth at any time. At least one of these commands must be input. If the DEPTH command is input without the DEPTH FUNCTION command, the fluid depth will be constant in time with that value. If the DEPTH FUNCTION command is input without a DEPTH command, the function is used as a time history of the depth. If both the DEPTH and the DEPTH FUNCTION commands are input, the specified depth is used as a scale factor on the time history function.

The depth and/or depth function are used to determine the current depth, which is added to the appropriate position of a datum point to compute the current location of the fluid surface in the FLUID SURFACE NORMAL direction. Unless the optional REFERENCE POINT command described below is used, the datum point is chosen to be a point on a surface defined in the SURFACE command that minimizes the projection along the FLUID SURFACE NORMAL direction.

The REFERENCE POINT command line specifies the fluid surface relative to an external datum. When applying an external fluid pressure in a quasistatic analysis, a corresponding stiffness due to the external fluid is added to the diagonal terms of the stiffness matrix for the full tangent preconditioner to enhance convergence of the solver.

7.13.3. Additional Commands

These command lines in the FLUID PRESSURE command block provide additional options for the boundary condition:

```
EXTERNAL FORCE CONTRIBUTION OUTPUT NAME =
  <string>variable_name
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

If the EXTERNAL FORCE CONTRIBUTION OUTPUT NAME command line appears there will be a variable created with whatever name the user specifies for variable_name. The variable defines a three-dimensional vector at each node associated with this particular command block. The three-dimensional vector at each node represents the external force due solely to just this particular external force boundary contribution.

Once this variable for the external force contribution is specified, it may be used like any other nodal variable. The user can, for example, specify the variable as a nodal variable to be output in a RESULTS OUTPUT command block. Or the user can reference the variable in a user subroutine.

By default, the FLUID PRESSURE boundary condition will be active throughout an analysis. However, use of the ACTIVE PERIODS and INACTIVE PERIODS commands can be used to limit

the action of the boundary condition to specific time periods. The `ACTIVE PERIODS` command line specifies when the boundary condition is active implying that it is inactive during any periods not included on the command line. Alternatively, the `INACTIVE PERIODS` determines when the boundary condition will not be active. See Section 2.6 for more information about these commands.

7.14. Inertia Relief

```
BEGIN INERTIA RELIEF
  #
  # node set commands
  NODE SET|NODESET = <string list>nodelist_names
  SURFACE|SIDESET|SIDE SET = <string list>surface_names
  BLOCK = <string list>block_names
  ELEMENT = <integer list>element_ids
  NODE = <integer list>node_ids
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # component commands
    COMPONENT = <string>X|Y|Z
  # additional commands
  EXTERNAL FORCE CONTRIBUTION OUTPUT NAME = <string>output_field
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [INERTIA RELIEF]
```

The `INERTIA RELIEF` command block computes a force or a moment in the given component directions that counteracts the external forces/moments in the problem. This capability is useful when a body is flying through space given an external force contribution, but the user would like the body to remain in a constant frame of reference.

The `INERTIA RELIEF` command block contains two groups of commands—node set and component. Each of these command groups is basically independent of the others. In addition to the command lines in the two command groups, there are three additional command lines: `EXTERNAL FORCE CONTRIBUTION OUTPUT NAME`, `ACTIVE PERIODS`, and `INACTIVE PERIODS`. Following are descriptions of the different command groups.

7.14.1. Node Set Commands

The `node set commands` portion of the `INERTIA RELIEF` command block defines a set of nodes associated with the inertia relief field and can include some combination of the following command lines:

```
NODE SET|NODESET = <string list>nodelist_names
SURFACE|SIDESET|SIDE SET = <string list>surface_names
BLOCK = <string list>block_names
ELEMENT = <integer list>element_ids
NODE = <integer list>node_ids
INCLUDE ALL BLOCKS
```

```
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list>surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes. See Section 7.1.1 for more information about the use of these command lines for creating a set of nodes used by the boundary condition. There must be at least one `NODE SET|NODESET`, `SURFACE`, `BLOCK`, `ELEMENT`, `NODE`, or `INCLUDE ALL BLOCKS` command line in the command block.

7.14.2. Component Command

The `COMPONENT` command specifies the direction of the counterbalance force and is the same for all nodes in the node set associated with the particular `INERTIA RELIEF` command block. As an example, a component `x` corresponds to using direction vector (1, 0, 0). By default, all components are used in the inertia relief calculation.

7.14.3. Additional Commands

These command lines in the `INERTIA RELIEF` command block provide additional options for the boundary condition:

```
EXTERNAL FORCE CONTRIBUTION OUTPUT NAME =
  <string>name_of_output_field
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

The `EXTERNAL FORCE CONTRIBUTION OUTPUT NAME` command line can be used to specify the name of the output variable to be computed that is the counterbalance force calculated inside the inertia relief code. Note, in order to output the variable, the variable must still be requested as a nodal variable in the necessary output sections.

The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines determine when the boundary condition is active. See Section 2.6 for more information about these command lines.

7.14.4. General Notes

- If using a single `INERTIA RELIEF` command block, all blocks in the model must be connected in some way. To apply inertia relief to separated blocks in the model, multiple `INERTIA RELIEF` command blocks must be defined for each separate block.
- Inertia relief does not counterbalance kinematic boundary conditions.
- It is not guaranteed that the displacement of the model is zero. The only guarantee is for the momentum to be zero or near zero.

7.14.5. Example

```
Begin Inertia Relief
  include all blocks
End Inertia Relief
```

The resulting `INERTIA RELIEF` applies forces and moments that balance out the external forces and moments of all blocks in the model. The net forces and moments applied by the inertia relief boundary condition are available for output in the global variables `force_inertia_relief` and `moment_inertia_relief`. The centroid about which moments are applied is placed in the global variable `centroid_inertia_relief`.

7.15. Specialized Boundary Conditions

Specialized boundary conditions that are provided to enforce kinematic conditions or apply loads are described in this section.



7.15.1. Cavity Expansion

```
BEGIN CAVITY EXPANSION
  SURFACE|SIDESET|SIDE SET = <string list>surface_names
  BLOCK = <string list>block_list
  INCLUDE ALL BLOCKS
  REMOVE SURFACE = <string list>surface_names
  EXPANSION RADIUS = <string>SPHERICAL|CYLINDRICAL
    (SPHERICAL)
  FREE SURFACE = <real>top_surface_zcoord
    <real>bottom_surface_zcoord
  NODE SETS TO DEFINE BODY AXIS =
    <string>nodelist_1 <string>nodelist_id2
  TIP RADIUS = <real>tip_radius
  BEGIN LAYER <string>layer_name
    LAYER SURFACE = <real>top_layer_zcoord
      <real>bottom_layer_zcoord
    PRESSURE COEFFICIENTS = <real>c0 <real>c1 <real>c2
    SURFACE EFFECT = <string>NONE|SIMPLE_ON_OFF(NONE)
    FREE SURFACE EFFECT COEFFICIENTS = <real>coeff1
      <real>coeff2
  END [LAYER <string>layer_name]
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [CAVITY EXPANSION]
```

The `CAVITY EXPANSION` command block is used to apply a cavity expansion boundary condition to a surface on a body. This boundary condition is typically used for earth penetration studies where some type of projectile (penetrator) strikes a target. For a more detailed explanation of the numerical implementation of the cavity expansion boundary condition and the parameters for this boundary condition, consult Reference [1]. The cavity expansion boundary condition is a complex boundary condition with several options, and the detailed explanation of the implementation of the boundary condition in the above reference is required reading to fully understand the input parameters for this boundary condition.

There are two types of cavity expansion—cylindrical expansion and spherical expansion. Either the spherical or cylindrical option may be selected using the `EXPANSION RADIUS` command line; the default is `SPHERICAL`. Reference [1] describes these two types of cavity expansion.

The `surface set commands` portion of the `CAVITY EXPANSION` command block defines a set of surfaces associated with the cavity expansion pressure field. The surface specification commands work the same as for the pressure boundary condition as described in

section 7.6.1.1.

The cavity expansion boundary condition generates a pressure at a node based on the velocity and surface geometry at the node. Since cavity expansion is essentially a pressure boundary condition, cavity expansion must be specified for a surface.

The target has a top free surface with a normal in the global positive z-direction; the target has a bottom free surface with a normal in the global negative z-direction. The point on the global z-axis intersected by the top free surface is given by the parameter `top_surface_zcoord` on the `FREE SURFACE` command line. The point on the global z-axis intersected by the bottom free surface is given by the parameter `bottom_surface_zcoord` on the `FREE SURFACE` command line.

It is necessary to define two points that lie on the axis (usually the axis of revolution) of the penetrator. These two nodes are specified with the `NODE SETS TO DEFINE BODY AXIS` command line. The first node should be a node toward the tip of the penetrator (`nodelist_1`), and the second node should be a node toward the back of the penetrator (`nodelist_2`). Only one node is allowed in each node set.

It is necessary to compute either a spherical or cylindrical radius for nodes on the surface where the cavity expansion boundary condition is applied. This is done automatically for most nodes. The calculations for these radii break down if the node is close to or at the tip of the axis of revolution of the penetrator. For nodes where the radii calculations break down, a user-defined radius can be specified with the `TIP RADIUS` command line. For more information, consult Reference [1].

Embedded within the target can be any number of layers. Each layer is defined with a `LAYER` command block. The command block begins with

```
BEGIN LAYER <string>layer_name
```

and is terminated with:

```
END [LAYER <string>layer_name]
```

Here the string `layer_name` is a user-selected name for the layer. This name must be unique to all other layer names defined in the `CAVITY EXPANSION` command blocks. The layer properties are defined by several different command lines—`LAYER SURFACE`, `PRESSURE COEFFICIENTS`, `SURFACE EFFECT`, and `FREE SURFACE EFFECT COEFFICIENTS`. These command lines are described next.

- ```
  LAYER SURFACE = <real>top_layer_zcoord
     <real>bottom_layer_zcoord
  ```

  The layer has a top surface with a normal in the global positive z-direction; the layer has a bottom surface with a normal in the global negative z-direction. In the `LAYER SURFACE` command line, the point on the global z-axis intersected by the top layer surface is given by the parameter `top_layer_zcoord`, and the point on the global z-axis intersected by the bottom layer surface is given by the parameter `bottom_layer_zcoord`.
  ```

- PRESSURE COEFFICIENTS = <real>c0 <real>c1 <real>c2
  The value of the pressure at a node is derived from an equation that is quadratic based on some scalar value derived from the velocity vector at the node. The three coefficients for the quadratic equation (c0, c1, c2) in the PRESSURE COEFFICIENTS command line define the impact properties of a layer.
- SURFACE EFFECT = <string>NONE|SIMPLE_ON_OFF(NONE)
  There can be no surface effects associated with a layer, or there can be a simple on/off surface effect model associated with a layer. The type of surface effect is determined by the SURFACE EFFECT command line. The default is no surface effects. If the SIMPLE_ON_OFF model is chosen, it is necessary to specify free surface effect coefficients with the FREE SURFACE EFFECT COEFFICIENTS command line.
- FREE SURFACE EFFECT COEFFICIENTS = <real>coeff1 <real>coeff2
  All the parameters defined in a LAYER command block apply to that layer. If a simple on/off surface effect is applied to a layer, the surface effect coefficients are associated with the layer values. The surface effect parameter associated with the top of the layer is coeff1; the surface effect parameter associated with the bottom of the layer is coeff2.

The ACTIVE PERIODS and INACTIVE PERIODS command lines provide an additional option for cavity expansion. These command lines can activate or deactivate cavity expansion for certain time periods. See Section 2.6 for more information about these command lines.



Warning: At this time the CAVITY EXPANSION command block does not support any surface faces other than QUAD4.



7.15.2. Silent Boundary

```
BEGIN SILENT BOUNDARY
  SURFACE|SIDESET|SIDE SET = <string list>surface_names
  REMOVE SURFACE = <string list>surface_names
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [SILENT BOUNDARY]
```

The SILENT BOUNDARY command block is also called a non-reflecting surface boundary condition. A wave striking this surface is not reflected. This boundary condition is implemented with the techniques described in Reference [2]. The method described in this reference is excellent at transmitting the low- and medium-frequency content through the boundary. While the method does reflect some of the high-frequency content, the amount of energy reflected is usually minimal. On the whole, the silent boundary condition implemented in Presto is highly effective.

In the SURFACE command line, a series of surfaces may be listed through the string list surface_names. There must be at least one SURFACE command line in the command block. The REMOVE SURFACE command line allows the user to delete surfaces from the set specified in the SURFACE command line(s) through the string list surface_names. See Section 7.1.1 for

more information about the use of these command lines for creating a set of surfaces used by the boundary condition.

The ACTIVE PERIODS and INACTIVE PERIODS command lines provide an additional option for the boundary condition. These command lines are used to activate or deactivate the boundary condition for certain time periods. See Section 2.6 for more information about these command lines.

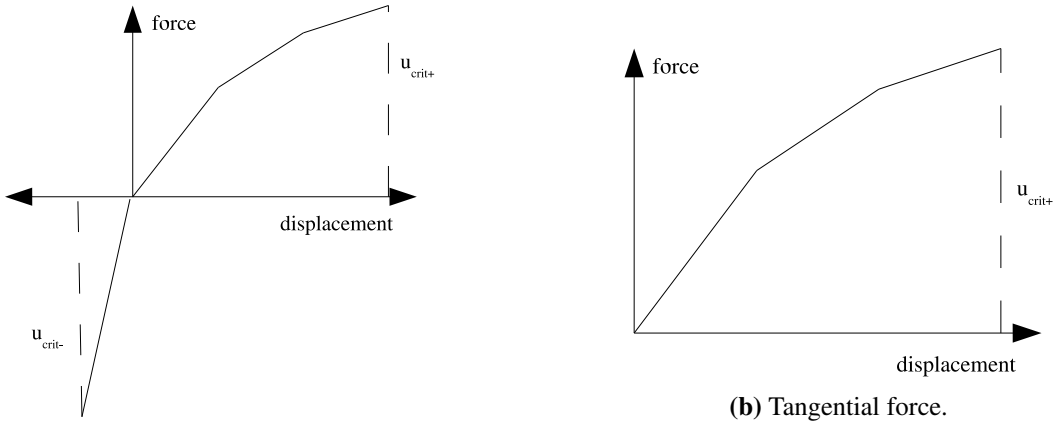
7.15.3. Spot Weld

```
BEGIN SPOT WELD
  NODE SET|NODESET = <string list>nodelist_ids
  REMOVE NODE SET = <string list>nodelist_ids
  SURFACE|SIDESET|SIDE SET = <string list>surface_ids
  REMOVE SURFACE = <string list>surface_ids
  SECOND SURFACE = <string>surface_id
  NORMAL DISPLACEMENT FUNCTION =
    <string>function_nor_disp
  NORMAL DISPLACEMENT SCALE FACTOR =
    <real>scale_nor_disp(1.0)
  TANGENTIAL DISPLACEMENT FUNCTION =
    <string>function_tang_disp
  TANGENTIAL DISPLACEMENT SCALE FACTOR =
    <real>scale_tang_disp(1.0)
  FAILURE ENVELOPE EXPONENT = <real>exponent(2.0)
  FAILURE FUNCTION = <string>fail_func_name
  FAILURE DECAY CYCLES = <integer>number_decay_cycles(10)
  SEARCH TOLERANCE = <real>search_tolerance
  IGNORE INITIAL OFFSET = NO|YES(NO)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
  CREATE VISUALIZATION ELEMENTS = OFF|ON(OFF)
END [SPOT WELD]
```

The spot weld option lets the user model an “attachment” between a node on one surface and a face on another surface. This option can model a weld, small screw, or bolt.

The weld uses functions to represent either force or traction–displacement relations. Two force or traction–displacement curves are required for the spot weld model: one curve models normal behavior, while the other models tangential response. Behavioral differences between normal and tangential components necessitate separating the response components; the node may interpenetrate the surface, resulting in a negative normal displacement between the connection points. Therefore, the normal force displacement curve *must* have the ability to capture negative displacements (and forces). Although this penalty stiffness approach will work to prevent interpenetration, it is best to model this behavior using contact. Tangential displacement, however, is always positive.

Sample normal and tangential force–displacement curves are illustrated in Figures 7-8a and 7-8b, respectively. The displacements shown are the normal or tangential distances that the node moves from the nearest point on the face as measured in the original configuration. The corresponding forces shown are the forces at the attachment as functions of the distance between the two attachment points. The force–displacement curve assumes the two attachment points are originally at the same location and the initial distance and force are zero at time zero.



(a) Normal force.

(b) Tangential force.

Figure 7-8. Spot weld force–displacement curves.

The sign of the normal displacement of the spot weld (generally tension at positive displacement or compression at negative displacement) is determined by the relative motion of the attached node and face as well as the normal direction of the face; see Figure 7-9. Relative motion of the face *along* the face normal causes compressive spot weld response, while relative motion of the face *opposite* to the face normal leads to tensile spot weld response.

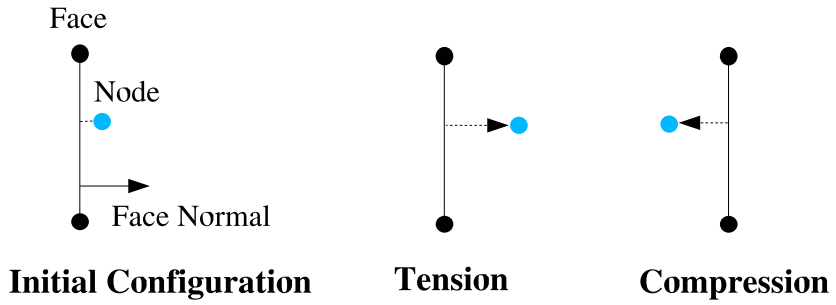


Figure 7-9. Sign convention for spot weld normal displacements.

The attachment is defined between a node on one surface and the closest point on an element face on the other surface. Since a face is used to define one of the attachment points, it is possible to compute a normal vector and a tangent vector associated with the face. This allows resolution of displacement (distance, both positive and negative) and force (both tensile and compressive) into normal and tangential components.

Note that the direction vector used to decompose normal and tangential deformations in the weld is the current normal of the face associated with the spot weld. See Figure 7-10 for a depiction of

how normal and tangential displacements are computed in the spot weld. The spot weld deformation is decomposed into into normal (SW_ND) and tangential (SW_TD) components.

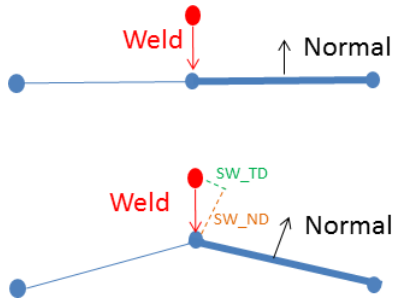


Figure 7-10. Spot weld normal and tangent.

With normal and tangential vectors associated with the attachment, the attachment can be characterized for the case of pure tension and pure shear. The pure tensile and pure shear response of the spot weld will be governed by the input force/traction–displacement functions. Adagio includes two mechanisms for determining failure for cases that fall between pure tension and pure shear. In the first case, failure is governed by the equation

$$(u_n/u_{n_{crit}})^p + (u_t/u_{t_{crit}})^p >= 1.0\,. \tag{7.6}$$

In Equation (7.6), the distance from the node to the original attachment point on the face as measured normal to the face is u_n, which is defined as the normal distance. The maximum value given for u_n in the normal displacement curve is $u_{n_{crit}}$, but is different for positive and negative displacements. In Figure 7-8a, the value used for $u_{n_{crit}}$ is u_{crit+} in the positive direction and u_{crit-} in the negative direction. The distance from the node to the original attachment point on the face as measured along a tangent to the face is u_t, which is defined as the tangential distance. The maximum value given for u_t in the tangential displacement curve is $u_{t_{crit}}$. The value p is a user-specified exponent that controls the shape of the failure surface.

Alternatively, Adagio permits a user-specified function to determine the failure surface. The function defines the ratio of $u_t/u_{t_{crit}}$ at which failure will occur as a function of $u_n/u_{n_{crit}}$. The function must range from 0.0 to 1.0, and have a value of 1.0 at 0.0 and a value of 0.0 at 1.0. These restrictions preserve proper failure for the cases of pure tension and pure shear.

To use the spot weld option in Presto, a `SPOT WELD` command block begins with the input line:

```
BEGIN SPOT WELD
```

and is terminated with the input line :

```
END [SPOT WELD]
```

A set of element faces which define one side of the spot weld (denoted the *first* surface) is specified with the `SURFACE` command line. The `SURFACE` command line can list one or more

surfaces. Any surface listed on the SURFACE command line can be deleted from the list of surface by using a REMOVE SURFACE command line

When a spot weld is defined by a node set using a NODE SET|NODESET command line and a surface, the spot weld normal and tangential displacement functions define force/displacement relations. Alternatively a spot weld can be defined by a surface and a SECOND SURFACE command line. When a spot weld is defined by two surfaces the normal and tangential displacement functions define traction/displacement relations. A spot weld command block must include either a NODE SET|NODESET command line or a SECOND SURFACE command line but not both. In either case the faces used for the spot weld constraints are always defined by the SURFACE command line.

For the NODE SET|NODESET surface spot weld, each node in NODE SET|NODESET will look for a nearby face in SURFACE. If a sufficiently close face is found a spot weld will be created between the node and face using the force/displacement relations defined by the force and normal displacement curves. For the SECOND SURFACE surface spot weld each node in SECOND SURFACE will look for a nearby face in SURFACE. If a sufficiently close face is found a spot weld will be created between the node and the face. The node face relation will use a force/displacement relation defined by multiplying the weld displacement function (assumed to be a traction/displacement relation) by the tributary surface area of the node to compute a force/displacement function unique to that particular node. Thus the total max force resisted by a NODE SET|NODESET based weld is the number of nodes in the set times the maximum value of the provided force/displacement curve. The total max force resisted by a SECOND SURFACE based weld is the tributary surface area of the second surface times the maximum value of the provided traction/displacement curve.

The normal force/traction–displacement curve is specified by a function named by the string function_nor_disp in the NORMAL DISPLACEMENT FUNCTION command line. The last points in the positive and negative directions are used as the displacements beyond which the spot weld fails. This function can be scaled by the real value scale_nor_disp in the NORMAL DISPLACEMENT SCALE FACTOR command line. The scaling factor scales the y values (force/traction) of the normal displacement function. Thus a scaling factor of 2.0 would indicate that for a given displacement the spot weld has twice the force defined in function_nor_disp. The default scale factor is 1.0.

The tangential force/traction–displacement curve is specified by a function named by the string function_tang_disp in the TANGENTIAL DISPLACEMENT FUNCTION command line. The last point in the positive direction in the tangential curve is used as the displacement beyond which the spot weld fails. This function can be scaled by the real value scale_tang_disp given in the TANGENTIAL DISPLACEMENT SCALE FACTOR command line; the default for this factor is 1.0. As with the normal displacement functions the scaling factors apply to the y value (force/tractions) used by the spot weld.

The failure surface between pure tension and pure shear is controlled by specifying either the failure envelope exponent, p in Equation (7.6), or a failure function. The failure exponent is specified by the real value exponent in the FAILURE ENVELOPE EXPONENT command line. The failure function is specified by the FAILURE FUNCTION command line. If both a failure function and a failure exponent are given, then the failure function is used.

For an explicit transient dynamics, it is better to remove the force for the spot weld over several load steps rather than over a single load step once the failure criterion is exceeded. The `FAILURE DECAY CYCLES` command line controls the number of load steps over which the final force is removed (default value is 10). To remove the final force at a spot weld over five time steps, the integer specified by `number_decay_cycles` would be set to 5. Once the force at the spot weld is reduced to zero, it remains zero for all subsequent time (despite the function definition). That is, once a weld has reached the failure displacement, it will forever after be broken and will not reconnect if the node and face later move together again.

The user must set a tolerance for the node-to-face search with the

```
SEARCH TOLERANCE = <real>search_tolerance
```

command line. The value selected for `search_tolerance` will depend upon the distance between the nodes and surfaces used to define the spot weld.

If the user sets `IGNORE INITIAL OFFSET = YES`, the initial distance between the node and the face will be taken as the zero normal displacement distance for the spot weld. When using the `IGNORE INITIAL OFFSET = YES` command the initial normal force in the will be the value of the force-displacement curve at displacement=zero. The sign convention used to determine if a spot weld is in compression or tension is the same as without using `IGNORE INITIAL OFFSET = YES` as shown Figure 7-11.

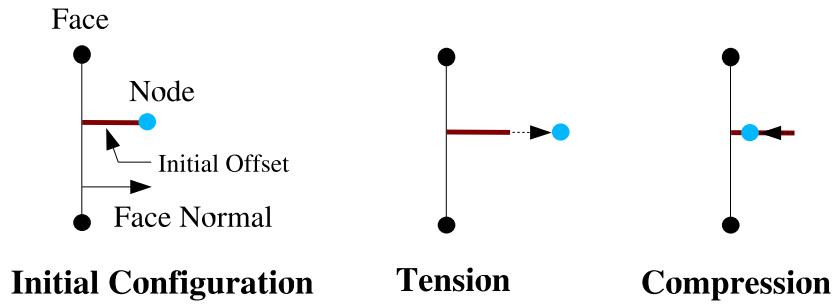


Figure 7-11. Sign convention for spot weld normal displacements with `IGNORE INITIAL OFFSET = YES`.

The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines are used to activate or deactivate the boundary condition for certain time periods. See Section 2.6 for more information.

The `CREATE VISUALIZATION ELEMENTS` command line triggers the creation of beam elements to visualize the spot weld in output. These elements are for visualization purposes only and have no effect on the spot weld behavior. The default value is `OFF`. Each beam element is drawn between the welded node and the corresponding contact point on the welded surface.

Some special output data can be obtained from spot welds. The list of available output variables is documented in Table 9-8.

7.15.3.1. Spot Weld Limitations and Usage Guidelines



Warning: In explicit dynamics, spot welds do not inherently have an *element* critical time step because they have zero mass. However, spot welds may still decrease the global system time step. When spot welds with high stiffness relative to their attached elements are used in the model, the default *element-based* critical time step estimate may lead to instability in the spot weld.

When a node-based time step estimate is used, spot welds contribute their stiffness to the nodes associated with the spot weld, thus influencing the nodal time step (see Section 3.2.3.1). Thus, the node-based time step should remain stable, even in the presence of a stiff spot weld.

The spot weld stiffness is calculated conservatively as the sum of the current derivatives of the normal and tangential force functions. If the nodes attached to an element that governs the critical time step have spot weld stiffness contributions, log file output will indicate the approximate percentage reduction in the critical time step due to the spot weld stiffness contribution.



Warning: If the offset between the spot weld node and the attached face becomes large, the weld may be become unstable. This instability is related to potentially inconsistent handling of moments induced by weld forces acting over a finite gap. It is recommended that the normal gap offset between the spot weld node and the spot weld face not exceed roughly one-half of the characteristic element length in the neighboring mesh. Note that a large gap in either initial or deformed coordinates are both problematic. Thus, welds that can undergo large normal displacements prior to breakage may encounter stability problems once the normal gap in the weld is large.

If spot welds are used to represent bolts that are larger than the mesh size, it is recommended to use rigid patches on both sides of the interface. Rigid patches prevent concentrated forces and large deformations at the spot weld attachment point, such as in Figure 7-10. Spot welds attached to rigid patches will also exhibit enhanced stability, for instance, under mesh refinement, since the characteristic face size is bound by the rigid patch size rather than the size of an individual face.



Warning: Although the spot weld is available for implicit computations, this use case is considered less mature than the explicit use case. Spot welds may adversely affect the conditioning of the linear systems solved in implicit analyses, and as a result can lead to poor convergence behavior.



Known Issue: Use of `SECOND SURFACE` is less mature and reliable than using `NODE SET|NODESET`. Instead of using a second surface, it is better to create a node set containing all the nodes of the desired surface.

Some other recommendations when using spot welds:

- When using node sets, be aware that the force will be distributed among all the nodes of the set. The distribution will also be uneven among the nodes which can cause failure of the spot weld due to concentrations of force on a particular node. If there is unexpected behavior, try starting with node sets that have only one node to better control the forces.
- If possible, use rigid surfaces for the surface and node set. This significantly improves behavior.
- The choice of `SEARCH TOLERANCE` greatly affects the performance of the spot weld. The value should be between 0 and the surface size, but probably no more than 25 percent of the length of the surface's largest characteristic length. In general, select the largest in a range of reasonable values.

7.15.3.2. Example

The following is an example of a basic spot weld defined between the nodes of `nodelist_17` and the faces of surface `surface_3`. Note the spot weld has a stiffness of 500×10^2 in compression, a stiffness of 100×10^2 in tension, and a stiffness of 50×10^2 in shear. Once either the normal or tangential displacement in the weld reaches a value of approximately 1.0×10^{-2}, the weld will fracture.

```
BEGIN FUNCTION spot_norm
  TYPE = PIECEWISE LINEAR
  BEGIN VALUES
    -1.0e-2 -500.0
     0.0       0.0
     1.0e-2  100.0
  END
END

BEGIN FUNCTION spot_tang
  TYPE = PIECEWISE LINEAR
  BEGIN VALUES
     0.0e-2   0.0
     1.0e-2  50.0
  END
END

BEGIN SPOT WELD
  NODE SET = nodelist_17
  SURFACE = surface_3
  NORMAL DISPLACEMENT FUNCTION = spot_norm
  TANGENTIAL DISPLACEMENT FUNCTION = spot_tang
  SEARCH TOLERANCE = 1.0e-3
END
```



7.15.4. Line Weld

```
BEGIN LINE WELD
  SURFACE|SIDESET|SIDE SET = <string list> surface_names
  REMOVE SURFACE = <string list> surface_names
  BLOCK = <string list> block_names
  REMOVE BLOCK = <string list>block_names
  SEARCH TOLERANCE = <real>search_tolerance
  R DISPLACEMENT FUNCTION = <string>r_disp_function_name
  R DISPLACEMENT SCALE FACTOR = <real>r_disp_scale
  S DISPLACEMENT FUNCTION = <string>s_disp_function_name
  S DISPLACEMENT SCALE FACTOR = <real>s_disp_scale
  T DISPLACEMENT FUNCTION = <string>t_disp_function_name
  T DISPLACEMENT SCALE FACTOR = <real>t_disp_scale
  R ROTATION FUNCTION = <string>r_rotation_function_name
  R ROTATION SCALE FACTOR = <real>r_rotation_scale
  FAILURE ENVELOPE EXPONENT = <real>k(2.0)
  FAILURE DECAY CYCLES = <integer>number_decay_cycles
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END LINE WELD
```



Warning: If a function provided by a `R DISPLACEMENT FUNCTION`, `S DISPLACEMENT FUNCTION`, `T DISPLACEMENT FUNCTION`, or `R ROTATION FUNCTION` command does not have a maximum x-limit (e.g., constant and analytic functions), then the line weld will not fail for displacements or rotations in that direction.

The line-weld capability is used to weld the edge of a shell to the face of another shell. The bond can transmit both translational and rotational forces. When failure of the line weld occurs, it breaks and no longer transmits any forces.

The edge of the shell that is tied to a surface is modeled with a block of one-dimensional elements (truss, beam, spring, etc.). The edge of the shell and the one-dimensional elements will share the same nodes. We will refer to the shell edge and the one-dimensional elements associated with it as the one-dimensional part of the line-weld model. The element blocks with the one-dimensional elements are specified by using the `BLOCK` command line. More than one element block can be listed on this command line. The element blocks referenced by the `BLOCK` command line must be one-dimensional elements—truss, beam, spring, etc.

The other part of the line weld is a set of faces defined by shell elements; this set of faces is the two-dimensional part of the line weld. The surface (the two-dimensional part of the model) to which the nodes (from the one-dimensional part of the model) are to be bonded is defined by any surface of element faces derived from shell elements. The line weld will bond each node in the element blocks listed in the `BLOCK` command line to the closest face (or faces) of element faces in

the surfaces listed in the SURFACE command line. More than one surface can be listed on this command line.

The command line SEARCH TOLERANCE sets a tolerance on the search for node-to-face interactions. For a given node, only those faces within the distance set by the search_tolerance parameter will be searched to determine whether the node should be welded to the face.

Each section of the line weld has its own local coordinate system (r, s, t). The r-direction lies along the one-dimensional element. The s-direction is perpendicular to the one-dimensional element and lies in the plane of the face that contains the one-dimensional element. The t-direction lies normal to the face and is orthogonal to the r- and s-directions. Force-per-unit-length-vs-displacement functions may be specified for all axes in the local coordinate system. A moment-per-unit-length-vs-rotation function may be specified for the r-direction, where rotations are in radians. All functions are required to be specified; SIERRA_CONSTANT_FUNCTION_ZERO can be used to give a zero stiffness. All functions only evaluate on the positive side and the relationship is assumed to be symmetric in compression versus tension.

The force-per-unit-length-vs-displacement function in the r-direction represents shear resistance in the direction of the weld; this function is specified by a SIERRA function name on the R DISPLACEMENT FUNCTION command line. The force-per-unit-length-vs-displacement in the s-direction represents shear resistance tangential to the weld; this function is specified by a SIERRA function name on the S DISPLACEMENT FUNCTION command line. The force-per-unit-length-vs-displacement in the t-direction function represents tearing resistance normal to the surface; this function is specified by a SIERRA function name on the T DISPLACEMENT FUNCTION command line. The moment-per-unit-length-vs-rotation about the r-axis represents a rotational tearing resistance; this is specified by a SIERRA function name on the R ROTATION FUNCTION command line, where rotations are in radians. Note that each SIERRA function used in this command block is defined via a FUNCTION command block in the SIERRA scope.

Any of the above functions can be scaled by using a corresponding scale factor. For example, the force-per-unit-length-vs-displacement function on the R DISPLACEMENT FUNCTION command line can be scaled by the parameter r_disp_scale on the R DISPLACEMENT SCALE FACTOR command line. The force-per-unit-length values of the force-per-unit-length-vs-displacement curve will be scaled; the displacement values are not scaled.

The failure for the line weld is similar to that for the spot weld. Denote the displacement or rotation associated with a line weld as δ. Suppose that δ_i is a displacement in the r-direction. The force-per-unit-length-vs-displacement curve specified on the R DISPLACEMENT FUNCTION command line has a maximum displacement value η. This is the maximum displacement the weld can endure in the r-direction before breaking. Associate this value of η with δ_i by designating it as η_i. Repeat this pairing process for all the displacements and the rotation r defining the line weld. Each displacement component in the line weld will be paired with one of the three maximum displacement values associated with the line weld. The r rotation component in the line weld will be paired with the maximum rotation value in the R ROTATION FUNCTION. Note that if a function provided by a R DISPLACEMENT FUNCTION, S DISPLACEMENT FUNCTION,

T DISPLACEMENT FUNCTION, or R ROTATION FUNCTION command does not have a maximum x-limit (e.g., constant and analytic functions), then the line weld will not fail for displacements or rotations in that direction.

The weld breaks under the following condition:

$$\sqrt[k]{\sum \left(\frac{\delta_i}{\eta_i}\right)^k} > 1\,. \tag{7.7}$$

In the above equation, the parameter k is set by the user. A typical value for k is 2. The summation takes place over all the failure functions (force-per-unit-length-vs-displacement and moment-per-unit-length-vs-rotation) for all the nodes. The value for k is specified on the FAILURE ENVELOPE EXPONENT command line.

For an explicit, transient dynamics analysis, it is better to remove the forces for the line weld over several time steps rather than over a single time step once the failure criterion is exceeded. The FAILURE DECAY CYCLES command line controls the number of time steps over which the final force is removed. To remove the final force at a line weld over five time steps, the integer specified by number_decay_cycles would be set to 5. Once the force in the line weld is reduced to zero, it remains zero for all subsequent time. Note, the line weld forces during the failure decay are independent of the line weld displacements or rotation and simply linearly decay from the force and moments reached at failure to zero over the specified number_decay_cycles.

The ACTIVE PERIODS and INACTIVE PERIODS command lines provide an additional option for this boundary condition. These command lines are used to activate or deactivate the boundary condition for certain time periods. See Section 2.6 for more information about these command lines.

Some special output data can be obtained from line welds. The list of available output variables is documented in Tables 9-9 and 9-20.



Warning: In explicit dynamics, line welds do not inherently have an *element* critical time step because they have zero mass. However, line welds may still decrease the global system time step. When line welds with high stiffness relative to their attached elements are used in the model, the default *element-based* critical time step estimate may lead to instability in the line weld.



7.15.5. Shell Solid Joint

```
BEGIN SHELL SOLID JOINT
  # block of shell elements
  BLOCK = <string list>block_names
  # Set of solid nodes
  NODE SET = <string list>nodelist_names
  SEARCH TOLERANCE = <real>tol
  TRUE GAUGE TEST = OFF|ON(OFF)  #EXPERIMENTAL DO NOT USE
  SIDE A SELECTION = NORMAL PROJECTION|
```

```
    NEAREST NODE(NORMAL PROJECTION)
  FOLLOW THICKNESS CHANGE = OFF|ON (OFF)
  ROTATION UPDATE TYPE = REISSNER|KIRCHHOFF (REISSNER)
  MEAN QUADRATURE ROTATION = OFF|ON (OFF)
END [SHELL SOLID JOINT]
```

The shell solid joint capability connects a set of solid nodes to a shell element block. Typically it is used to connect the nodes on the surface of a high-fidelity solid element block to the edge of a low-fidelity shell element block.

A shell solid joint enforces that the translations of the solid nodes remain compatible with the shell translations and rotations at the interface. The solid nodes will conform to the shell kinematic assumptions (e.g., plane sections remain plane) as if embedded in the shell. The through-thickness thinning or expansion of the solid mesh in the joint region is constrained by the thickness change of the shells. The shell solid joint constraint equations are written at the solid nodes, i.e., they constrain each solid node within search tolerance to a single shell element.

The `BLOCK` command selects one or more sets of shell elements to constrain the solid nodes to. These must be two dimensional shell—not membrane—formulation elements. The `NODE SET` command selects a set of nodes to constrain to the shells. These may be nodes connected to any type of solid element (i.e., they should not have rotational degrees of freedom).

The `SEARCH TOLERANCE` command optionally sets the tolerance for how close nodes need to be to shells to become constrained. If the `SEARCH TOLERANCE` command is not used a search tolerance is selected automatically based on the shell thickness, lofting factor, and in-plane size. The automatic search tolerances are set such that the joint will constrain any solid node found within or nearby the volumetric domain occupied by the theoretical shell material. A pictorial representation of how search tolerances are used is shown in Figure 7-12.

The `TRUE GAUGE TEST` option is experimental and should not be used.

The `SIDE A SELECTION` command is used to change how the constrained nodes are projected to the plane of the shell. By default the constraint is made via a projection of the node along the shell normal onto the plane of the shell. The `NEAREST NODE` option will instead project the constrained node to the nearest node on the shell. The `NEAREST NODE` option can potentially be useful when the shell and solid meshes have been constructed contiguously such that a line of nodes on the solid side directly one to one maps to a single node on the shell side. For such a case use of the `NEAREST NODE` option will remove any possibility of mapping errors due to shell warping or curvature. However, the default `NORMAL PROJECTION` option should always be used for all general or non-matching shell and solid meshes.

7.15.5.1. Shell Solid Joint in Implicit Computations



Warning: Although the shell solid joint is available for implicit computations, this use case is considered experimental. There are many known issues with the implicit shell solid joint, leading to poor convergence in serial analyses and potential inconsistencies in parallel analyses.

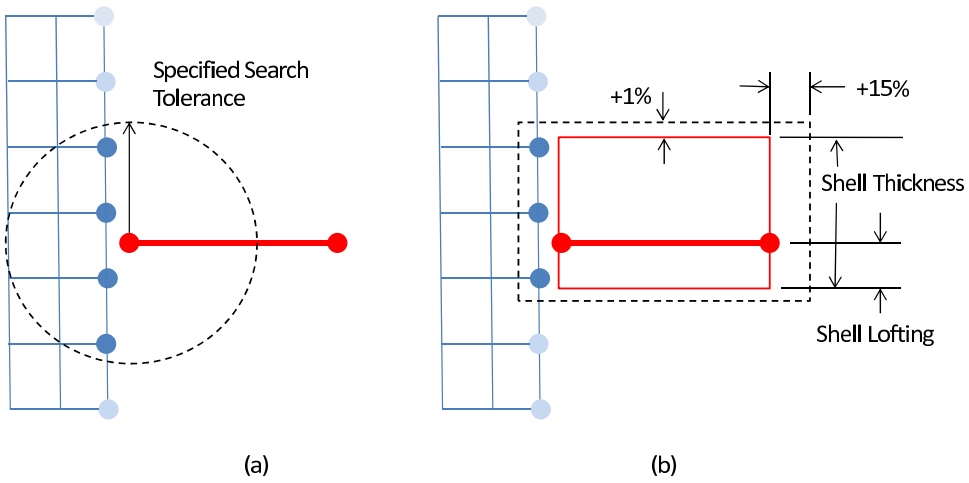


Figure 7-12. Search tolerances for shell-solid joints. (a) input; (b) automatic

A number of implicit-specific options exist for the shell solid joint.

The shell solid joint constraint is enforced through the control contact multilevel solver in implicit analyses. Thus, a `CONTROL CONTACT` solver block must be present in the solver settings.

By default the movement of the solid modes on the interface interface is uncoupled to any Poisson thinning or thickening of the attached shell. The `FOLLOW THICKNESS CHANGE = ON` option will turn on coupling of this degree of freedom ensuring that the solid nodes move up or down the shell normal consistently with the shell thickness changes. This option can potentially lead to a more accurate coupling of the joint during large membrane stretching. However, the coupling may also in some circumstances over constrain the joint causing robustness issues.

The `ROTATION UPDATE TYPE` command defines how shell solid joints computes coupling rotations. The default `REISSNER` option couples rotations to the rotational degrees of freedom of the shell nodes. The `KIRCHHOFF` option couples rotations to changes in shell element normal. Generally both options should yield the similar results. The `KIRCHHOFF` option could in some cases produces a stiffer but more stable result. This stability can be particular important if constraining a stiff material shell to a soft material solid.

The `MEAN QUADRATURE ROTATION` command allows tying rotations to the average shell rotation (`ON`) option rather than the nodal point (default `OFF`) option. This command has a similar effect to the `ROTATION UPDATE TYPE` where the `ON` option can create a potentially stiffer but more stable joint.

7.15.5.2. Restrictions and Assumptions

The following guidelines apply to both the explicit and implicit shell solid joint.

- Though the shell solid joint is a kinematic MPC, it can potentially artificially stiffen the nearby elements and thus it may be necessary to manually reduce the analysis time step a small amount to ensure integration stability.
- It is generally assumed that there are minimal or no lateral gaps or overlaps in the joint.

Gaps or overlaps between the shell block(s) and solid(s) will tend to introduce additional artificial stiffness into the joint.

- Any offset between the solid cross-section neutral bending plane and the shell midplane will result in an effective lofting distance, which may increase the stiffness of the structure unexpectedly. (This is because the enforcement of shell kinematics on the solid nodes forces the solid to rotate and bend around the shell midplane as if part of the shell.)
- In order to correctly enforce moment and rotation compatibility across the joint, the joint must contain at least two solid nodes in the shell normal (i.e., thickness) direction. It is preferable to have five or more solid nodes in the shell normal direction for a smoothly connected joint.
- For best results the solid side of the joint should be meshed as fine or finer than the shell side. A finely meshed shell connected to a coarsely meshed solid will result in constraints at only the solid nodes, leading to unconstrained shells nodes at the joint.
- The shell solid joint constraint can constrain each solid node to only one shell. In ambiguous cases the "best" matching shell will be selected based on proximity.
- Care should be taken if using the shell solid joint on the same part of a mesh as contact or other kinematic or rigid body constraints. If multiple constraint types come into conflict generally either one of the constraints sets will not be enforced or stability problems will result.



7.15.6. Viscous Damping

```
BEGIN VISCOUS DAMPING <string>damp_name
  #
  # block set commands
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE BLOCK = <string list>block_names
  #
  MASS DAMPING COEFFICIENT = <real>mass_damping
  ROTATIONAL MASS DAMPING COEFFICIENT = <real>rot_mass_damping
  STIFFNESS DAMPING COEFFICIENT = <real>stiff_damping
  #
  VELOCITY DAMPING COEFFICIENT = <real>damping_coefficient
  VELOCITY DAMPING FUNCTION = <string>function_name
  #
  RIGID BODY INVARIANCE = OFF|ON(OFF)
  #
  # additional commands
  ACTIVE PERIODS = <string list>period names
  INACTIVE PERIODS = <string list>period_names
END [VISCOUS DAMPING <string>damp_name]
```

The `VISCOUS DAMPING` command block adds simple Rayleigh viscous damping to mesh nodes. At each node, Presto computes a damping coefficient, which is then multiplied by the node velocity to create a damping force. The damping coefficient is the sum of the mass times a mass

damping coefficient and the nodal stiffness times a stiffness damping coefficient. In general, the mass damping portion damps out low-frequency modes in the mesh, while the stiffness damping portion damps out higher-frequency terms. Appropriate values for the damping coefficients depend on the frequencies of interest in the mesh. The general expression for the critical damping fraction, c_d, for a given frequency is

$$c_d = \frac{1}{2}\left(k_d\omega + \frac{m_d}{\omega}\right), \tag{7.8}$$

where k_d is the stiffness damping coefficient, m_d is the mass damping coefficient, and ω is the frequency of interest. The stiffness damping portion must be used with caution. Because this term depends on the stiffness, it can affect the critical time step. Thus certain ranges of values for the stiffness damping coefficient can change the critical time step for the mesh. As Presto does not currently modify the critical time step based on the selected values for this coefficient, some choices for this parameter can cause solution instability.

Velocity damping applies forces that, in the absence of other forces, have the effect of directly opposing nodal velocity according to the expression

$$\mathbf{v}_{new} = (1.0 - v_d)\mathbf{v}_{old}, \tag{7.9}$$

where v_d is the velocity damping coefficient and $\mathbf{v}$ represents the velocity. That is, the coefficient v_d represents the fraction by which the previous nodal velocity will be reduced by the applied damping forces in the absence of other forces.

7.15.6.1. Block Set Commands

The `block set commands` portion of the `VISCOUS DAMPING` command block defines a set of element blocks associated with the viscous damping and can include some combination of the following command lines:

```
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE BLOCK = <string list>block_names
```

These command lines provide a set of Boolean operators for constructing a set of element blocks. See Section 7.1.1 for more information about the use of these command lines for creating a set of element blocks used by viscous damping. There must be at least one `BLOCK` or `INCLUDE ALL BLOCKS` command line in the command block. All the nodes associated with the elements specified by the block set commands will have viscous damping forces applied.

7.15.6.2. Viscous Damping Coefficient

The mass damping coefficient, m_d, in Equation (7.8) is specified using the parameter `mass_damping` on the command line:

```
MASS DAMPING COEFFICIENT = <real>mass_damping
ROTATIONAL MASS DAMPING COEFFICIENT = <real>rot_mass_damping
```

Note, the `MASS DAMPING COEFFICIENT` affects only translational degrees of freedom while the `ROTATIONAL MASS DAMPING COEFFICIENT` affects only the rotational degrees of freedom on shells, beams, and rigid bodies. Mass damping most strongly damps the low-frequency modes, and in Presto, is applied as a boundary condition which does not affect the critical time step.

The stiffness damping coefficient, k_d, in Equation (7.8), is specified as the parameter `stiff_damping` on the command line:

```
STIFFNESS DAMPING COEFFICIENT = <real>stiff_damping
```

Stiffness damping most strongly damps high-frequency modes and is applied through a stress correction of the pressure term at the element level. Because this type of damping is done at the element level, it has a direct effect on the critical time step as seen in Equation (7.10), where $\epsilon = \frac{k_d \omega}{2}$ in the absence of mass damping and $\frac{2}{\omega} \approx \frac{m}{\kappa}$.

$$\Delta t \leq \frac{2}{\omega}\left(\sqrt{1+\epsilon^2} - \epsilon\right) \tag{7.10}$$

The velocity damping coefficient, v_d, in Equation (7.9), is specified by *either* the constant parameter `vel_damping` on the command line:

```
VELOCITY DAMPING COEFFICIENT = <real>vel_damping
```

or the user defined time-dependent function $v_d(t)$ `function_name` on the command line:

```
VELOCITY DAMPING FUNCTION = <string>function_name
```

Velocity damping applies a damping term that has the effect of lowering velocity (the end result is described by Equation (7.9)), and damping out all modes. Velocity damping is primarily used to drive a dynamic solution towards static equilibrium during or after a preload procedure.

Velocity damping and mass damping *may not* be used in combination.

A velocity damping coefficient and a velocity damping function *may not* be used in combination. The constant damping coefficient v_d or the damping function $v_d(t)$ must be between 0 and 1:

$$0 \leq v_d(t) \leq 1 \quad \text{for all} \;\; t \in \{t_s\} \tag{7.11}$$

where $t \in \{t_s\}$ are actual simulation time steps. A velocity damping function that evaluates outside these bounds may result in analysis termination. Note that since arbitrary user-defined functions may depend on runtime solution variables, this check cannot be performed at parse time.

```
RIGID BODY INVARIANCE = OFF|ON(OFF)
```

By default mass and velocity damping apply a force to each node that opposes the current velocity of that node. This acts as if the part is moving through a viscous fluid. This force will damp out both vibration and net rigid body motions. The rigid body invariance option applies damping forces based on the difference between the node velocity and the net average velocity of the node set. The intent is to damp out vibrations in the part but not affect the net rigid body motions. Note, this option only makes sense if the damping is applied on contiguous parts. For example applying rigid body invariant damping to two pieces moving in opposite directions would still slow both parts as the net rigid motion of the parts moving away from one another is zero. An additional effect of the `rigid body invariance = on` option is that the net force and moment applied by viscous damping will be assured to be zero.

7.15.6.3. Additional Command

The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines can optionally appear in the `VISCOUS DAMPING` command block:

```
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

This command line can activate or deactivate the viscous damping for certain time periods. See Section 2.6 for more information about this command line.

7.15.7. Artificial Strain

```
BEGIN ARTIFICIAL STRAIN
  #
  # Zone of application commands
  INCLUDE ALL BLOCKS
  ELEMENT = <int list>elem_nums
  BLOCK = <string list>block_names
  REMOVE BLOCK = <string list>block_names
  #
  # Strain definition commands
  STRAIN TYPE = LOG|ENGINEERING(ENGINEERING)
  FUNCTION = <string>function_name
    [SCALE FACTOR = <real>scale]
  DIRECTION <string>dir_name FUNCTION = <string>dir_func
    [SCALE FACTOR = <real>scale]
  DIRECTION FIELD <string>field_name FUNCTION =
    <string>field_dir_func [SCALE FACTOR = <real>scale]
  R FUNCTION = <string>r_func [SCALE FACTOR = <real>scale]
  S FUNCTION = <string>s_func [SCALE FACTOR = <real>scale]
  T FUNCTION = <string>t_func [SCALE FACTOR = <real>scale]
END [ARTIFICIAL STRAIN]
```

A The ARTIFICIAL STRAIN command block defines an artificial strain applied on a block by block or element by element basis. Artificial strains may be useful for fitting parts together for a preload analysis or modeling a variety of other physics that could cause material expansion or contraction (such as radiation induced expansion, material desiccation, etc.)

Artificial strains may also be defined in the material block, see Section 5.1.3. However, defining artificial strains by boundary condition as shown in this section is more flexible.

A variety of commands can be used to describe the set of elements this command block applies to as detailed in Section 7.1.1.

Artificial strains may be applied based in either the LOG or ENGINEERING strain space as defined from the STRAIN TYPE command. There are several ways to define the applied strain tensor. One and only one of the following command sets may be used.

- FUNCTION is a single function defining the isotropic strain to apply. The function is a time history function optionally scaled by an input constant.
- DIRECTION may be used up to three times to define up to three orthogonal directions paired with separate time history functions in which to apply strain. The directions' names used here are defined in a Sierra scope DIRECTION command. The direction command is generally only applicable to three dimensional elements and defines the strain directions in the global XYZ coordinate system. The functions may be optionally scaled by an input constant.
- DIRECTION FIELD may be used along with DIRECTION lines to define up to three orthogonal directions paired with separate time history functions in which to apply strain. The field names given to DIRECTION FIELD refer to name of an element direction variable. The element direction variable must be a defined element output field and must have exactly three components at each element. The direction field may be populated by a number of methods including USER OUTPUT to define the field on the fly or use of INITIAL CONDITION blocks to read the field from the mesh file. Note, though the field used can potentially vary in time the behavior of artificial strain is not well defined for time variant directions and it is recommended that only directions that are constant in time be used. Additionally at each and every element the set of up to three DIRECTION FIELD directions and DIRECTION directions must be orthogonal to each other. The functions may be optionally scaled by an input constant.
- the R, S, T FUNCTION commands define strain in the element local RST coordinate system. For solids this generally corresponds to the global XYZ coordinate system. For shells and membranes R and S are the in plane directions and T is the through thickness direction. For beams and trusses R is the axial direction and S and T are the lateral directions. These functions are time history functions and may be optionally scaled by an input constant.

Note, the artificial strain is not an absolute constraint. An element could be artificially shrunk and then stretched by actual material strain in a way that the element shape doesn't change, but the element does accumulate stress and strain energy. If artificial strain is applied uniformly to an unconstrained body that strain will grow or shrink the body as described by the artificial strain in a nominally stress free manner.

7.15.7.1. Restrictions and Usage Guidelines

Artificial strain cannot be simultaneously defined in a material and in the artificial strain block for the same element. Thermal strain may be defined in a material and artificial strain in an artificial strain block. However, in such cases the strain must be of matching type (`LOG` or `ENGINEERING`.)

One common error is assuming strain applied in the direction `(1, 1, 0)` has the same effect of applying an equal R and S strain. Applying a strain in the direction of `1, 1, 0` expands the body uniformly along the 45 degree direction vector. Applying a strain in both the R and S directions expands the body uniformly in the RS plane.

It is possible to apply any arbitrary strain state by computing the eigenvectors and values of the strain state and then applying direction strains in the directions of the eigenvectors with the magnitudes of the eigenvalues.

Note, strains should generally be applied by functions that are smooth in time. A sharp jump in the target artificial strain over a time step will tend to produce a strong shock in the system.

7.15.8. Tied Multi-Point Constraints

```
BEGIN TIED MPC
  #
  TIED FACES = <string list>BLOCKS/SURFACES
  TIED NODES = <string list>BLOCKS/SURFACES/NODE SETS
  #
  # Tied contact search commands
  SEARCH TOLERANCE =  AUTO / <real> tolerance (AUTO)
  VOLUMETRIC SEARCH TOLERANCE = <real>vtolerance
  SEARCH CONFIGURATION = MODEL|CURRENT (CURRENT)
  #
  # DOF subset selection
  COMPONENTS = <enum>X|Y|Z|RX|RY|RZ
  REMOVE GAPS AND OVERLAPS = OFF|ON(OFF)
END [TIED MPC]
```

Sierra/SM provides a tied contact multi-point constraint (MPC) capability that allows a code user to tie a set of nodes to a set of faces. The `BEGIN TIED MPC` and `BEGIN EQUATION MPC` (see Section 7.15.9) command blocks replace the capabilities of `BEGIN MPC` (deprecated in the 4.56 release). For tied contact MPCs, a proximity search is performed to create a set of MPCs that act as tied contact constraints. If the MPCs are created in this way, the search is performed at the time of initialization to find pairings of nodes (defined from the `TIED NODES` command line) to faces (defined from the `TIED FACES` command line). `TIED FACES` may be names of blocks and/or surfaces and `TIED NODES` may be names of blocks, surfaces and/or node sets. A separate constraint is created for each node in the `TIED NODES`. This type of MPC is equivalent to using

pure tied contact and is significantly faster than standard tied contact for explicit dynamics analyses.

```
SEARCH TOLERANCE = AUTO / <real> tolerance (AUTO)
```

The SEARCH TOLERANCE command line specifies the maximum distance between a node and a face to create an MPC. This has a similar meaning to the search tolerance used in standard tied contact. By default, the search tolerance is set to AUTO which automatically computes the search tolerance value internally. The value that is computed as the search tolerance is 15 percent of the second smallest edge length of the face.

```
VOLUMETRIC SEARCH TOLERANCE = <real>vtolerance
```

In addition to creating node/face constraints, a search can be used to create volumetric constraints, in which nodes among the TIED NODES that are not within the SEARCH TOLERANCE of the TIED FACES will also be constrained if their distance from nodes on the TIED FACES is less than or equal to the VOLUMETRIC SEARCH TOLERANCE. A set of node/face constraints is first created for the TIED NODES within the standard search tolerance of any TIED FACES. Next, a volumetric constraint is created for any remaining unconstrained nodes within the VOLUMETRIC SEARCH TOLERANCE of nodes on the TIED FACES. Each volumetric constraint gets distance-weighted MPC contributions from the nodes of the TIED FACES that are within the search tolerance.

```
SEARCH CONFIGURATION = MODEL|CURRENT (CURRENT)
```

The configuration in which nodes are searched for to create the MPCs can be changed via the SEARCH CONFIGURATION command. This can be useful if a restart or procedural transfer is performed and the MPCs need to change or remain the same as the start of the simulation.

```
COMPONENTS = <enum>X|Y|Z|RX|RY|RZ
```

By default, tied multi-point constraints are applied to all degrees of freedom of the nodes involved. For nodes that only have translational degrees of freedom, all three components (X, Y and Z) are constrained. Likewise, for nodes that have both translational and rotational degrees of freedom, all six components (X, Y, Z, RX, RY and RZ) are constrained. The COMPONENTS command line can be used to enforce the constraint on a subset of the degrees of freedom. If the COMPONENTS command line is included in a TIED MPC command block, only the components listed would be constrained.

To use tied MPCs, the TIED FACES and TIED NODES must be defined. It is important to note that the TIED FACES cannot contain any node sets.

The following example demonstrates how to use Tied MPCs between two surfaces:

```
BEGIN TIED MPC
  TIED FACES = surface_10
  TIED NODES = surface_11
END MPC
```

The following example demonstrates the usage of Tied MPCs for both tied and volumetric constraints. Assuming that `surface_10` is adjacent to the exterior surface of `block_1`, this command block would result in tied contact between the exterior nodes of `block_1` and the faces of `surface_10`, and volumetric constraints between the interior nodes of `block_1` and the exterior nodes of `block_1`.

```
BEGIN TIED MPC
  TIED FACES = surface_10
  TIED NODES = block_1
  VOLUMETRIC SEARCH TOLERANCE = 3.0
END MPC
```

Tied MPCs can potentially be in conflict with other constraints. Refer to Appendix D for information on how conflicting constraints are handled.

For a list of fields available for output, see Table 9-10.

7.15.8.1. Resolve Multiple MPCs

The behavior of tied multi-point constraints is ill-defined when a node within the `TIED FACES` is constrained to more than one set of nodes within `TIED NODES`. Sierra/SM's tied MPC capability can handle chained MPCs, where a node in the `TIED FACES` is a node within the `TIED NODES` in another constraint, but it cannot simultaneously enforce multiple tied MPCs that have the same `TIED FACES` node.

```
RESOLVE MULTIPLE MPCS = ERROR|FIRST WINS|LAST WINS|ENFORCE ALL|
  IGNORE(ERROR)
```

The `RESOLVE MULTIPLE MPCS` command line, used within the region scope, controls how to resolve cases where a node within the `TIED NODES` is constrained to more than one set of nodes within the `TIED FACES`. Although multiple MPCs cannot be simultaneously enforced, this command provides ways to work around this problem that may be acceptable in many situations. The default option is `ERROR`, which results in an error message and terminates the code if this occurs. Alternatively, this command can take the following options:

`FIRST WINS` Keep only the first MPC seen at a node (MPC order generally determined by input deck order).

`LAST WINS` Keep only the last MPC seen at a node (MPC order generally determined by input deck order).

ENFORCE ALL Attempt to simultaneously enforce all MPCs seen at a node. As long as the MPCs are orthogonal this may work. If the MPCs are not orthogonal this will likely lead to an erroneous result or stability issues.

IGNORE If a node is found with multiple MPCs, ignore all the conflicting MPCs and enforce none.

7.15.8.2. Remove Gaps and Overlaps

Depending on how the model is constructed, there may be a gap between the `TIED FACES` and the `TIED NODES` in the initial configuration. Tied MPCs are best posed when this gap is zero. The option `REMOVE GAPS AND OVERLAPS`, if set to `ON`, moves the constrained node to the constraint point during initialization, explicitly zeroing the gap. The default value for this command is `OFF`.

7.15.9. Equation Multi-Point Constraints

```
BEGIN EQUATION MPC
  #
  # Equation MPC commands
  SIDE A NODE SET = <string list>side_a_nset
  SIDE A NODES = <integer list>side_a_nodes
  SIDE A WEIGHTS = <real list>side_a_weights
  SIDE A SURFACE = <string list>side_a_surf
  SIDE A BLOCK = <string list>side_a_block
  SIDE B NODE SET = <string list>side_b_nset
  SIDE B NODES = <integer list>side_b_nodes
  SIDE B SURFACE = <string list>side_b_surf
  SIDE B BLOCK = <string list>side_b_block
  #
  # DOF subset selection
  COMPONENTS = <enum>X|Y|Z|RX|RY|RZ
END [EQUATION MPC]

# Control handling of multiple MPCs
RESOLVE MULTIPLE MPCS = ERROR|FIRST WINS|LAST WINS|
                        ENFORCE ALL|IGNORE(ERROR)
```

Sierra/SM provides an equation multi-point constraint (MPC) capability that allows a code user to specify arbitrary constraints between sets of nodes. The `BEGIN EQUATION MPC` and `BEGIN TIED MPC` (see Section 7.15.8) command blocks replace the capabilities of `BEGIN MPC` (deprecated in the 4.56 release). The commands to define an equation MPC are all listed within a `EQUATION MPC` command block. The equation type of MPC imposes a constraint between a set of side A nodes and a set of side B nodes. The motion of the side B nodes is constrained to be equal to the average motion of the side A nodes. This type of MPC is most useful if there is either

a single side A node and one or more side B nodes, or multiple side A nodes and a single side B node. If there are multiple side B nodes, they are constrained to move together as a set.

The sets of side A and side B nodes used in the MPC can be defined by using a node set on the mesh file, a list of nodes provided in the input file, or a surface on the mesh file from which a list of nodes is extracted. This can be done for the set of side A nodes using one or more of the following commands:

```
SIDE A NODE SET = <string list>side_a_nset
SIDE A NODES = <integer list>side_a_nodes
SIDE A SURFACE = <string list>side_a_surf
SIDE A BLOCK = <string list>side_a_block
SIDE A WEIGHTS = <real list>side_a weights
SIDE B NODE SET = <string list>side_b_nset
SIDE B NODES = <integer list>side_b_nodes
SIDE B SURFACE = <string list>side_b_surf
SIDE B BLOCK = <string list>side_b_block
```

The `SIDE A NODE SET` and `SIDE B NODE SET` command lines specify the names of node sets in the mesh file. The nodes in these node sets are included in the sets of side A or side B nodes for the constraint.

The `SIDE A NODES` and `SIDE B NODES` command lines specify lists of integer IDs of nodes to be included in the sets of side A or side B nodes for the constraint.

The `SIDE A SURFACE` and `SIDE B SURFACE` command lines specify the name of a surface in the mesh file. The nodes contained in this surface are included in the sets of side A or side B nodes for the constraint.

The `SIDE A BLOCK` and `SIDE B BLOCK` command lines specify the names of a block in the mesh file. The nodes contained in these blocks are included in the sets of side A or side B nodes for the constraint.

The `SIDE A WEIGHTS` command can be used to explicitly define a constraint equation. This command is subject to several restrictions. The `SIDE A WEIGHTS` command must be used with a `SIDE A NODES` command and cannot be used with a `SIDE A NODE SET`, `SIDE A SURFACE`, or `SIDE A BLOCK` command. If the `SIDE A WEIGHTS` is used the number number of weights must be the same as the number of nodes in the `SIDE A NODES` command. Negative weights are allowed. However, negative MPC weights can potentially cause stability problems in explicit analysis.

```
COMPONENTS = <enum>X|Y|Z|RX|RY|RZ
```

By default, multi-point constraints are applied to all degrees of freedom of the nodes involved. For nodes that only have translational degrees of freedom, all three components (X, Y and Z) are constrained. Likewise, for nodes that have both translational and rotational degrees of freedom, all six components (X, Y, Z, RX, RY and RZ) are constrained. The `COMPONENTS` command line can be used to enforce the constraint on a subset of the degrees of freedom. If the `COMPONENTS`

command line is included in a `EQUATION MPC` command block, only the components listed would be constrained.

The following is an example of use of the `SIDE A WEIGHTS` command to define an equation MPC.

```
BEGIN EQUATION MPC
  SIDE B NODES    = 23
  SIDE A NODES   = 26    12   16
  SIDE A WEIGHTS = 0.5 0.25 0.25
  COMPONENTS = X
END MPC
```

The above MPC adds a constraint that the X displacement at node 23 will always be equal to 0.5 times the X displacement at node 26 plus 0.25 the X displacement at node 12 plus 0.25 the X displacement at node 16.

Equation MPCs can potentially be in conflict with other constraints. Refer to Appendix D for information on how conflicting constraints are handled.

For a list of fields available for output, see Table 9-10.

7.15.9.1. Resolve Multiple MPCs

The behavior of multi-point constraints is ill-defined when a side A node is constrained to more than one set of side B nodes. Sierra/SM's MPC capability can handle chained MPCs, where a side A node is a side B node in another constraint, but it cannot simultaneously enforce multiple MPCs that have the same side B node.

Refer to Section 7.15.9.1 for documentation of the `RESOLVE MULTIPLE MPCS` command line.

7.15.10. Submodel

```
BEGIN SUBMODEL
  #
  EMBEDDED BLOCKS = <string list>embedded_block
  ENCLOSING BLOCKS = <string list>enclosing_block
END [SUBMODEL]
```

Sierra/SM provides a method to embed a submodel in a larger finite element model. The element blocks for both the submodel and the larger system model should exist in the same mesh file. The space occupied by the embedded blocks should also be occupied by the enclosing blocks.

This capability ties each node of the submodel to an element in the larger finite element model. The code makes no correction for mass due to volume overlap. However, this correction often can be done easily by hand simply by adjusting the density of the submodel block so that it is the difference between the density of the submodel block and the enclosing block.

The embedded blocks (the submodel blocks) and the enclosing blocks (the system model blocks) are specified using the following two line commands:

```
EMBEDDED BLOCKS = <string list>embedded_block
ENCLOSING BLOCKS = <string list>enclosing_block
```

For example, to embed `block_7` and `block_8` inside a system model where the embedded blocks are within `block_2`, `block_3`, and `block_5`, the following can be used:

```
BEGIN SUBMODEL
  EMBEDDED BLOCKS = block_7 block_8
  ENCLOSING BLOCKS = block_2 block_3 block_5
END
```

7.15.11. RBE3

```
BEGIN RBE3
  #
  REFERENCE BLOCK|SURFACE|NODE SET|NODES = <string list>NAMES
  CONNECTION BLOCK|SURFACE|NODE SET|NODES = <string list>NAMES
  #
  # RBE3 WEIGHTS of 6 DOFs
  WEIGHTS =  <double list(6)> X Y Z RX RY RZ
  #
  # Control of reference node mass distribution
  DISTRIBUTE REFERENCE NODE MASS = TRUE|FALSE(TRUE)
END [RBE3]
```

Sierra/SM provides the ability to define a rigid body element, or so-called RBE3, constraint between sets of nodes in a model. In the RBE3 formulation a set of 'connection' nodes and a set of 'reference' nodes are specified. Reference nodes are specified using one of the `REFERENCE BLOCK`, `REFERENCE SURFACE`, `REFERENCE NODE SET`, or `REFERENCE NODES` commands, while connection nodes can similarly be specified using one of `CONNECTION BLOCK`, `CONNECTION SURFACE`, `CONNECTION NODE SET`, or `CONNECTION NODES`. The RBE3 constraint prescribes a weighted average of the motion of the reference nodes to the connection nodes; currently, a mass-weighting scheme is used. Note that RBE3s are *not* actually rigid elements. If used correctly, RBE3s should not add stiffness to the system.

RBE3 constraints in Sierra/SM currently only provide a force-type coupling: forces and moments prescribed at the reference nodes are applied as equivalent forces on the connection nodes.

```
WEIGHTS = <double list(6)> X Y Z RX RY RZ
```

The `WEIGHTS` command line specifies the coupling weight applied to each corresponding force and moment component. For example, specifying `WEIGHTS = 1.0 1.0 1.0 1.0 0.0 0.0`

will apply equivalent forces to the connection nodes computed from the x-, y-, and z-components of the forces and the x-component of the moments at the reference nodes (i.e., excluding the y- and z-components of the nodal moments from the constraint). By default, all six component weights are set to a value of 1.0.

The following example demonstrates how to use an RBE3 to tie two blocks:

```
BEGIN RBE3
  REFERENCE BLOCK = BLOCK_1
  CONNECTION BLOCK = BLOCK_2
  WEIGHTS = 1 1 1 1 1 1
END RBE3
```

The `DISTRIBUTE REFERENCE NODE MASS` command specifies whether the total mass of the reference nodes is used to update the accelerations—and, consequently, the forces—of the connection nodes. If this option is `FALSE`, the only effective masses $\tilde{m}_{ci}$ contributing to the accelerations of the connection nodes are the standard nodal masses m_{ci}. If this option is `TRUE`, the effective masses of the connection nodes are computed as

$$\tilde{m}_{ci} = m_{ci} + \sum_j w_{ji} m_{rj}$$

where m_{rj} are the reference node masses, and w_{ji} mass-weighting factors for distribution (the same weighting used for the RBE3 force and moment distribution). Note that when distribution of the reference node mass is enabled, the effective connection node masses $\tilde{m}_{ci}$ are only factored into the accelerations of the connection nodes; the actual nodal masses m_{ci} are not altered. `TRUE` is the default setting.



Warning: RBE3s do *not* currently transmit kinematic boundary conditions.



Warning: The reference node positions of the RBE3 are currently not updated with respect to the connection nodes.



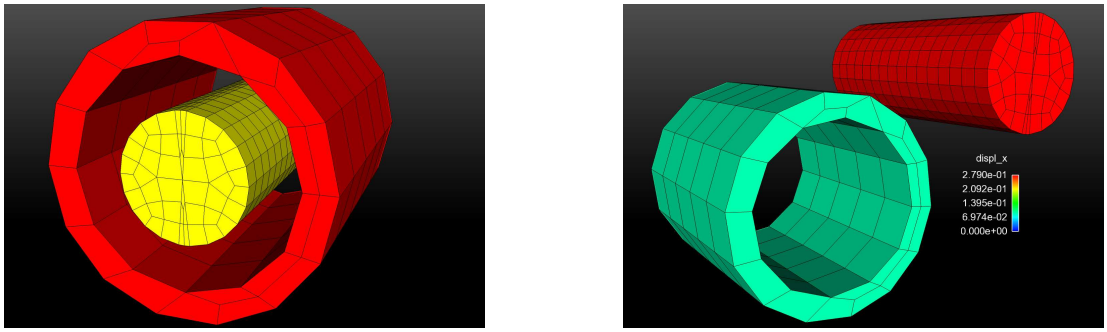
Known Issue: The RBE3 constraint is currently incompatible with mesh rebalance.

7.15.11.1. Sample RBE3 Usage

The following example illustrates how the RBE3 is capable of transferring only the x-component of any forces applied to the reference block. The problem consists of a cylinder at the center of a pipe that has a larger inner radius than the cylinder's outer radius such that there is no contact. The problem setup is shown in Figure 7-13a. In this example, the cylinder is the reference block and the pipe is the connection block in the RBE3 input command block. A prescribed force is

applied to the cylinder in the x-, y-, and z-directions, while a distribution weight is specified only in the x-direction, as shown below. Thus, the connection block transfers only the x-component of forces from the cylinder, as shown in Figure 7-13b.

```
BEGIN RBE3
  REFERENCE BLOCK = BLOCK_2
  CONNECTION BLOCK = BLOCK_1
  WEIGHTS = 1 0 0 0 0 0
END RBE3
```



(a) Initial position. **(b)** Final position.

Figure 7-13. RBE3 test positions.

7.15.12. Bolt Preloading

A capability for bolt preloading exists to help streamline the definition of a very specific, but common, operation. Preloading is achieved through the strategic application of artificial strain to a specified element block. The preload target value is approached by incorporating an preload controller/solver that monitors a user-defined global output variable. The bolt preloading function makes heavy use of a production-level preloading subroutine, but with some further enhancements to improve usability and efficiency.

Note that the preload capability is designed to be most useful in explicit analyses, although implicit static and implicit dynamic solution types may also work without issue. Further enhancements for implicit solution types are anticipated in the future.

The available bolt preloading command syntax is as follows:

```
BEGIN PRELOAD
  #
  # REQUIRED: PRELOAD DEFINITIONS
  #
  PRELOAD BOLT <string>block_name1 TO TARGET <string>global_target_var1
      = <real>target_value1 [IN DIRECTION <string>dir_name]
  PRELOAD BOLT <string>block_name2 TO TARGET <string>global_target_var2
      = <real>target_value2 [IN DIRECTION <string>dir_name]
```

```
  ...
  PRELOAD BOLT <string>block_nameN TO TARGET <string>global_target_varN
      = <real>target_valueN [IN DIRECTION <string>dir_name]
  (preload commands may be repeated multiple times)

  PRELOAD BOLT <string>block_name TO TARGET MAX VON MISES
      = <real>target_valueN [IN DIRECTION <string>dir_name]
  PRELOAD BOLT <string>block_name TO TARGET INTERNAL FORCE
      = <real>target_valueN [IN DIRECTION <string>dir_name]

  #
  # OPTIONAL: PRELOAD SOLVER & OTHER SETTINGS
  #
  ITERATION TIME = <real>iter_time
  SOLVER ADAPTIVITY = ON|OFF(OFF)
  COMPUTE INTERNAL REACTION OUTPUTS = ON|OFF(OFF)

  #
  # OPTIONAL: ACTIVE PERIOD SETTINGS
  #
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END
```

Bolt preloading in SIERRA can be achieved in a multitude of ways. The preload functionality described herein is offered as one specific method in which bolt preload may be achieved. Although the `BEGIN PRELOAD` command block will simply apply artificial strain as directed, users are recommended to adopt a bolted-joint geometry creation approach that mirrors what is depicted in Figure 7-15. Specifically, the mesh entities that the user wishes to preload (the shank, in this example) should be in a separate element block. The reason for this is two-fold: first, the preload operation is generally more physically appropriate and successful when other portions (bolt head, threads) are not included in the preload algorithm, and second, the target preload variable (e.g., normal force, maximum von Mises) must be computed on only the block of interest. Geometry definitions that significantly depart from this recommendation may not work as desired.

Users are also recommended to place bolts in individual blocks so that preload commands may be specified and evaluated on a bolt-by-bolt basis. However, some methods (such as a von Mises stress-based preload target) may work with all bolt shanks in a single block, but this approach is not advisable since the global target values would necessarily be global among all bolts.

The preload capability is constrained by the time period during which users allow preloading to take place. As preloading is still a dynamic event when applied as part of an explicit dynamics solution, users must take care to provide sufficient time for bolt preloading to take place. For many analyses, a time period in the range of 1ms to 10ms is sufficient, but the exact time period needed will vary and may require some experimentation.

Note that there are several practical usage concerns associated with the preloading subroutine in

general (e.g., monotonic relationship, proper constraints on joint, avoidance of yielding during iteration, etc.) to achieve a successful preload; these are outlined more explicitly in the chapter describing the user subroutines.

A dissipation mechanism is recommended to assist with achieving a quasi-static preload state. The `VELOCITY DAMPING` line command in a `VISCOUS DAMPING` command block may be used for this purpose. An appropriate value may be in the range of 0.0001 to 0.005, although smaller or larger values may be needed and some experimentation will be necessary. The damping may be selectively disabled by using the `ACTIVE PERIODS` or `INACTIVE PERIODS` after the preloading time period has ended if desired.

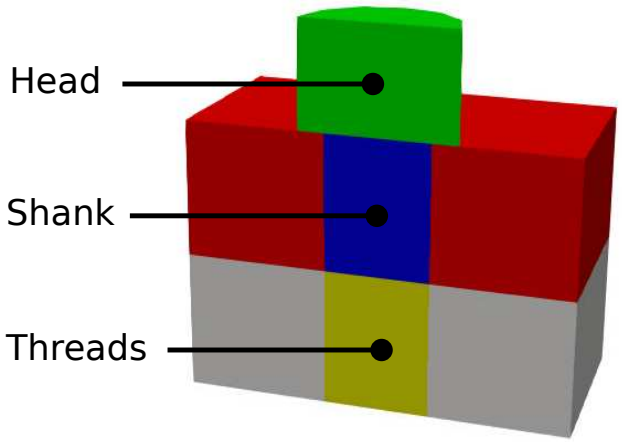


Figure 7-14. Anatomy of a typical preloaded bolted joint. Users are recommended to place the portion of the bolt shank to be preloaded in a separate element block.

Inside the `BEGIN PRELOAD` command block, users may specify the `PRELOAD BOLT` line command any number of times.

```
#
# REQUIRED: PRELOAD DEFINITIONS
#
PRELOAD BOLT <string>block_name TO TARGET <string>global_target_var
    = <real>target_value [IN DIRECTION <string>dir_name]
PRELOAD BOLT <string>block_name TO TARGET MAX VON MISES
    = <real>target_valueN [IN DIRECTION <string>dir_name]
PRELOAD BOLT <string>block_name TO TARGET INTERNAL FORCE
    = <real>target_valueN [IN DIRECTION <string>dir_name]
```

The `PRELOAD BOLT` line command will cause uni-directional artificial strain to be applied to `block_name` in the specified direction.

Establishing a preload direction The preload direction may be explicitly specified using a direction defined at the SIERRA scope. Predefined directions `GLOBAL_(X|Y|Z)` are also

available without additional definition at the SIERRA scope for convenience. Alternatively, if the preload direction is omitted a default direction will be automatically computed. For solid elements, the automatically-computed direction is determined from the geometry of `block_name` as the principal axis corresponding to the minimum eigenvalue of the moment of inertia tensor. For the common case of a relatively-long cylindrical shaft, this is the desired axial direction; however, results should be verified when using automatic directions (in results output, and also in the log file where an initialization summary table is available). For beam elements, the local x-direction is the only meaningful preload direction and is automatically used regardless of geometry.

Setting preload target (built-in) Preload commands must specify the desired method of preload, generally as a global quantity. Two common use-cases are available as "presets" and can be used without any further user input: `TO TARGET VON MISES` and `TO TARGET INTERNAL FORCE`. These preload commands will cause the preload algorithm to iterate until the maximum von Mises stress or the net internal axial force reach the specified target values, respectively.

Setting preload target (user-defined) Preloading to reach other target quantities is also supported, but requires a user-defined global variable `global_target_var` to be specified in the input deck. This must be done via a `USER OUTPUT` command block which computes some global quantity of interest (such as average von Mises stress) over the block with name `block_name`. The line command `COMPUTE AT EVERY STEP` must be specified as part of the user output block. Depending on the nature of the desired target variable, it may also be necessary to manually provide an initial guess.

Active preload period During the time period in which the `PRELOAD` command block is active, a preload controller/solver will incrementally apply an artificial strain until a target value `target_value` is reached, or until the active time period (or analysis) terminates. At the end of the preload operation, a table containing diagnostic information will be output to the log file for reference. Data included in the table assists with knowing the final preload in each bolt, the required strain to reach that preload (may be used in subsequent analyses directly, without preload iterations), and a quick go/no-go indicator for if all preload targets were reached.

Preloading multiple bolts The `PRELOAD BOLT` line command may be repeated within the `BEGIN PRELOAD` command block any number of times. Generally each bolt should be placed in a separate block to prevent issues. For example, automatic direction determination is incompatible with multiple bolts/bodies in the same block. Additionally, quantities such as internal force are computed over the block, and would be additive when multiple bolts are present in a single block. For example, preloading each of six bolts in a bolted joint interface to 1000lb would result in a net force of 6000lb if all bolts are in the same block. Thus, if target value of 6000lb is properly set, all bolts are aligned in the same direction, and the direction is set manually, then this can be done with a single command line. Preloaded values will be computed

for the entire joint and values in each bolt cannot be monitored individually with this approach. However, with each bolt in a separate block the target force can be set directly to 1000lb and it is possible to monitor preload in each bolt individually. In contrast, preloading to a target stress (such as max von Mises) is not additive in the interface, and in this case all bolts may be preloaded to the desired value directly without modification of the target value.

A separate preload controller/solver will be created for each line command. Currently, the `aupst_preload_solver` subroutine is used to drive preloads to their target values. The subroutine is agnostic to preload of other bolts, and so is more a driver/controller than a true "solver." Future improvements may enable more complex solver options for all bolts specified in the `BEGIN PRELOAD` command block.

Tabular preload summaries A tabular summary will be output at the end of the preload time period or analysis, whichever comes first. The tabular summary will list the block name being preloaded, the percentage and value of preload achieved relative to the target, and the amount of artificial strain applied at the termination of the preloading operation. These values are offered for reference or may be used to directly apply artificial strain to a specific value in future repeat analyses. For example, an explicit preloading analysis may be used to determine the required artificial strain to achieve a desired preload. The results of that analysis may be used in an implicit analysis to quasi-statically apply preload over just a few time steps since the required value is known *a priori*.

Variables for monitoring preload A set of output variables are also created for each preload line command. A variable named `sm_preload_strain_<block_name>` will be created for each `PRELOAD BOLT` line command. For example, the preload line command given as:

```
PRELOAD BOLT bolt1_shank TO TARGET max_von_mises_bolt1 = 8000
    IN DIRECTION GLOBAL_X
```

would generate a global variable named `sm_preload_strain_bolt1_shank` that could be monitored as necessary. The preload magnitude could be monitored using the user-defined output variable `max_von_mises_bolt1`. The preload subroutine employed also uses a number of state variables (defined in Chapter 10) that may be evaluated by examining the global variables named `sm_preload_state_<block_name>`.

Optional commands Although the default options are generally designed to achieve target preload without further intervention, several optional commands are available to influence the preload algorithm.

```
#
# OPTIONAL: PRELOAD SOLVER & OTHER SETTINGS
#
ITERATION TIME = <real>iter_time
SOLVER ADAPTIVITY = ON|OFF(OFF)
COMPUTE INTERNAL REACTION OUTPUTS = ON|OFF(OFF)
```

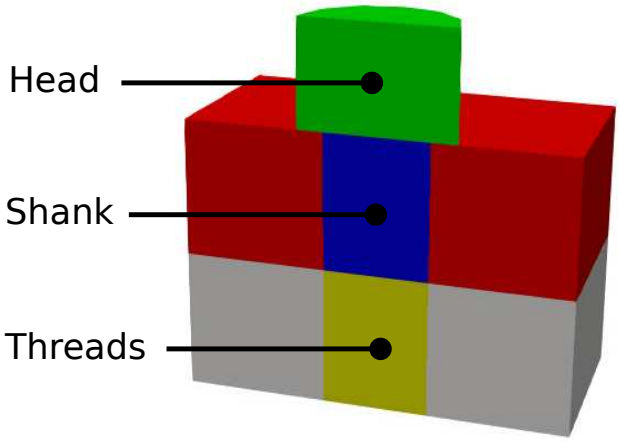


Figure 7-15. Anatomy of a typical preloaded bolted joint.

The preload algorithm incrementally applies artificial strain over a time period given by `iter_time`. By default, this value is automatically determined by the duration of the allotted preload period such that 20 strain incrementation cycles will occur. The user may override this default value by specifying a value for `ITERATION TIME`. The strain incrementation over each iteration is automatically determined based on preload geometry, material properties, and target value.

The *experimental* `SOLVER ADAPTIVITY` line command may be used to enhance the preload subroutine. By monitoring the preload solution for overshoot and convergence, the time period over which strain increments are applied, or the magnitude of a strain increment, may be adaptively changed to increase the solution efficiency. For example, if a very small overshoot is detected the time increment could be cut back such that a target preload is achieved more quickly. The solver adaptivity options are currently limited (for example, there is no growth when overshoot *is* detected) so this option is disabled by default. However, users may experiment with the option as desired. Users are advised to only enable this command for troubleshooting a very specific issue in which the automatically determined strain increment is not sufficient to reach target preload.

The `COMPUTE INTERNAL REACTION OUTPUTS` line command will optionally compute global output variables named `sm_preload_axial_force_<block_name>` whose values are the net axial force in the bolt. The net axial force is computed using the same method as the `COMPUTE ... AS INTERNAL REACTION` line command offers in a `USER OUTPUT` command block (see Section 9.3.2.2). This output is disabled by default to avoid unnecessary calculation when it is not needed. This output is provided as a convenience for monitoring preload values during active preload time periods; however, the values will not be updated after the preload period ends.

The `BEGIN PRELOAD` may also be restricted to specific time periods via the usual `ACTIVE PERIODS` and `INACTIVE PERIODS` line commands as described in Section 2.6. Although the use of these commands is optional, users are advised to provide only a single active period for preloading. The first time period in which preloading is active will determine the

duration over which strain increments are applied, which may influence the amount of noise and overshoot introduced in each preload solver iteration. The preload solver will continue being active into subsequent time periods as specified. Thus, if one wishes to cause the preload solver to iterate more than the default iteration time allows for, a second time period could be added. In some circumstances, this approach may allow for further dynamic relaxation and reduction of noise in the system.

```
#
# OPTIONAL: ACTIVE PERIOD SETTINGS
#
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

There are several recommendations and limitations that users must be aware of when using the bolt preloading command block.

1. Bolts of solid element should be constructed nominally as depicted in Figure 7-15. The objective is to isolate the bolt shank from the rest of the body so that it can be preloaded separate from the head or threaded portion of the bolt shank. Beam elements should be straight and should not specify a direction.
2. The `COMPUTE AT EVERY STEP` option must be specified in the user-defined target variable to ensure accurate and up-to-date values are available for the preload controller.
3. The use of a single time period is recommended. The preload solver option uses the starting and stopping times of the first preload time period to determine the time period over which preload is incremented when not specified explicitly by the `ITERATION TIME` command.
4. A sufficiently long time period, relative to the physical system under consideration, must be specified for the preload to occur. This duration is often on the order of 1 ms to 10 ms.
5. Users are advised to place bolts into separate blocks, with separate preload line commands specified for each bolt to be preloaded.
6. Some form of viscous damping is recommended for explicit analyses.

References

[1] K.H. Brown, J.R. Koteras, D.B. Longcope, and T.L. Warren. CavityExpansion: A library for cavity expansion algorithms, version 1.0. Technical Report SAND2003-1048, Sandia National Laboratories, Albuquerque, NM, April, 2003. pdf.

[2] J. Lysmer and R.L. Kuhlmeyer. Finite dynamic model for infinite media. In *Proceedings of the American Society of Civil Engineers*, pages 859–877, 1979.

8. CONTACT

This chapter describes the input syntax for defining interactions of contact surfaces. For more information on the computational details of contact, consult References [1, 2, 3].

Contact is the interaction of bodies when they touch. Contact algorithms ensure that surfaces do not interpenetrate and that the interface behavior is computed based on specified surface physics models.

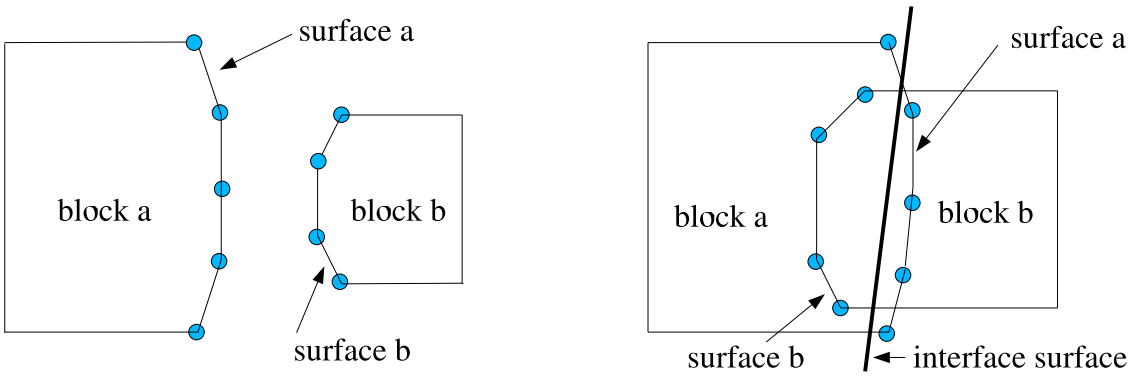
To specify contact enforcement in a model, the user must:

- Identify all surfaces to be considered for contact. Usually this is the exterior surfaces of element blocks used in the analysis.
- Specify which contact surfaces will interact with each other. This could involve tying surfaces together, allowing surfaces to slide against each other, or allowing part of a surface to slide against other parts of the same surface.

Additional algorithmic options may be included to improve the robustness and accuracy of the contact computations.

8.1. Overview of Contact Constraints

Contact between surfaces is modeled by first defining and then enforcing a set of contact constraints. A simple example of contact is shown in Figure 8-1a. Block a has exterior surface a, and block b has exterior surface b, each defined by a collection of finite element faces. For this two-dimensional drawing, the faces are straight lines between two nodes. Figure 8-1a shows the



(a) Time step n before contact. **(b)** Time step $n+1$ after penetration.

Figure 8-1. Two blocks in contact.

two blocks at time step n. Figure 8-1b shows the blocks at time step $n+1$ when the blocks have

moved and deformed under the influence of external and internal forces. Before contact is taken into account, the two blocks interpenetrate one another. This initial, potentially interpenetrated configuration is known as the initial predicted configuration. The interpenetration is removed by applying the contact algorithm.

There are three major types of contact constraints available in Sierra/SM. Node/face contact constraints are based on the concept of preventing nodes from penetrating faces on the opposing surface. Face/face constraints prevent faces on opposing surfaces from penetrating each other. Node/analytic surface constraints prevent nodes from penetrating analytically defined surfaces.

For node/face constraints, interpenetration occurs as shown in Figure 8-1b when any node on surface *a* passes through a face of surface *b* or a node on surface *b* passes through a face of surface *a*. One option to remove interpenetration would be to force all the nodes on surface *a* to lie on surface *b*, where surface *b* has the configuration shown in Figure 8-1b. In this case, surface *b* would be the side A surface and surface *a* would be the side B surface. Alternatively, all nodes on surface *b* could be forced to lie on surface *a*, in which case surface *a* is the side A surface and surface *b* is the side B. It is generally best to let the code decide the side A and side B surfaces, but the specific designations can also be input by the user.

More typically, nodes on both surfaces are moved to what can be described as an interface surface, which is shown by a thick black line in Figure 8-1b. The interface surface shown here as a straight line could be curved in reality. For each node on surface *a* that has penetrated a face on surface *b* forces based on the amount of node penetration are placed on both the node on *a* and the nodes associated with the face on *b*. Likewise, forces are computed for each node on surface *b* that has penetrated some face on surface *a* and for the nodes associated with the penetrated face on surface *a*. The combination of all these forces will remove the interpenetration of the two blocks and produce the final interface configuration.

The preceding two-dimensional example is analogous to much of the contact that is encountered when contact is used in an analysis. Surfaces are generated that consist of a collection of faces. Contact interactions consisting of node-face pairs are defined on the current geometry and then contact enforcement is used to remove the interpenetration of the surfaces and satisfy surface physics.

8.2. Overview of Contact Enforcement

A contact search algorithm provides the contact interactions on which contact constraints could be enforced. (An interaction in its simplest form is a node-face pairing as described in Section 8.1.) Contact enforcement must first determine which configuration dictates these interactions and which of these interactions need enforcement. Active set logic is used to determine which interactions provided by the search require interpenetration removal based on tolerances and other algorithmic issues. In addition to enforcing interpenetration constraints, the enforcement algorithm must satisfy constraint conditions defined by the friction model. In the simple example of a frictionless model, contact enforcement must ensure no penetration and a balance of forces.

The active set logic differs in the enforcement algorithms used in explicit and implicit time integration. For explicit time integration, the determination of the active set is usually made from the initial predicted position and remains unchanged for the time step. For example, in a frictionless node-face interaction, contact will enforce the constraint conditions on the pair if the node is penetrating the face in the initial predicted position. However, for large motion problems the active set can be updated within the time step by setting `NUM GEOMETRY UPDATE ITERATIONS` (See Section 8.9.7). For implicit time integration, the active set is iterated with the solve (See Section 4.5) because the motion over a time step is generally expected to be large.

Two basic algorithmic approaches exist in Sierra/SM for contact constraint enforcement. Kinematic enforcement prescribes zero interpenetration and iterates to achieve equal and opposite forces. Augmented Lagrange enforcement applies equal and opposite forces and iterates to achieve zero interpenetration. Augmented Lagrange is generally more robust, able to handle multiple conflicting constraints, and more easily incorporates complicated friction model definitions. As such, Augmented Lagrange is the default enforcement algorithm in both implicit and explicit simulations with Sierra/SM.

In explicit, the contact enforcement is linked to the search algorithm. The DASH search algorithm activates Augmented Lagrange enforcement, while ACME search activates kinematic enforcement (See Section 8.9). Both enforcement approaches use a fixed number of iterations which can be set with `MOMENTUM BALANCE ITERATIONS` (See Section 8.9.7).

In implicit, the contact enforcement algorithm is specified with the `ENFORCEMENT` command (See Section 8.9.3). For both Augmented Lagrange and Kinematic enforcement, the number of iterations is determined by convergence criteria that are specified in the `CONTROL CONTACT` section (See Section 4.5).

8.3. Contact Definition Block

The following set of commands are used to define contact. The contact definition block is placed in the REGION command scope. For an example of how to create a basic contact definition block, see Section 8.12. Commands which are specifically related to ACME contact are provided in Section 8.14.

```
BEGIN CONTACT DEFINITION <string>name

  # Contact surface definition commands:

  SKIN ALL BLOCKS = <string>ON|OFF(OFF)
    [EXCLUDE <string list> block_names]
    # see Section 8.4.4

  CONTACT SURFACE <string>name CONTAINS <string list>surface_names
    # see Section 8.4.1

  CONTACT NODE SET <string>surface_name
    CONTAINS <string>nodelist_names
    # see Section 8.4.2

  BEGIN CONTACT SURFACE <string>name
    # ... see Section 8.4.3
  END [CONTACT SURFACE <string>name]

  # friction model definition commands:
    # ... see Section 8.5

  #  interaction definition commands:

  BEGIN INTERACTION DEFAULTS [<string>name]
    # ... see Section 8.6.1
  END [INTERACTION DEFAULTS <string>name]

  BEGIN INTERACTION [<string>name]
    # ... see Section 8.6.2
  END [INTERACTION <string>name]

  ACTIVE PERIODS = <string list>period_names
    # see Section 8.7

  # contact algorithm option commands:

  SEARCH = <string>ACME|DASH(DASH)
    # see Section 8.9.1
```

```
  BEGIN DASH OPTIONS
    # ... see Section 8.9.2
  END

Implicit  ENFORCEMENT = <string>AL|KINEMATIC(AL)
    # see Section 8.9.3

Implicit  CONTACT RESIDUAL NORM SCALE FACTOR = <real>scale


  INITIAL OVERLAP REMOVAL = ON|OFF(OFF)
  BEGIN REMOVE INITIAL OVERLAP
    # ... see Section 8.9.4
  END [REMOVE INITIAL OVERLAP]
  OVERLAP REMOVAL ONLY = OFF|ON(OFF)


  BEGIN SURFACE OPTIONS
    # ... see Section 8.9.5
  END [SURFACE OPTIONS]

  BEGIN SEARCH OPTIONS [<string>name]
    # ... see Section 8.9.6
  END [SEARCH OPTIONS <string>name]

Explicit  BEGIN ENFORCEMENT OPTIONS [<string>name]
    # ... see Section 8.9.7
  END [ENFORCEMENT OPTIONS <string>name]

Explicit  COMPUTE CONTACT VARIABLES = ON|OFF|EVERY_STEP(ON)
    # see Section 8.10

  BEGIN DEBUG [<string>name]
    # ... see Section 8.10.1
  END [DEBUG <string>name]

END [CONTACT DEFINITION <string>name]
```

The command block begins with the input line:

```
BEGIN CONTACT DEFINITION <string>name
```

and is terminated with the input line:

```
END [CONTACT DEFINITION <string>name]
```

The string `name` is a name for this contact definition. The name should be unique among all the contact definitions in an analysis and is used in diagnostics printed in the log file or error messages related to the contact definition.

A typical analysis will have only one `CONTACT DEFINITION` command block. However, more than one contact definition may be used. Each `CONTACT DEFINITION` command block creates its own set of contact constraints that operate independently from one another. Use of fewer contact definition commands blocks will tend to provide more efficient processing and better conflict resolutions when different types of contact constraints come into conflict.

Contact behaves somewhat differently in explicit versus implicit analyses. Additionally there are multiple contact enforcement libraries available: ACME and Dash. The detailed descriptions of each of the commands list what options and commands are available based on what type of analysis is being done and contact library is being used. Unless otherwise noted the behavior of a command is the same regardless of the type of analysis or the contact algorithm being used.

8.4. Defining the Contact Surfaces

The user must provide a description of which surfaces will be considered for contact. There are many options available to either automatically detect and create contact surfaces or fine tune specific contact surface definitions.

Many separate contact surfaces can be defined by the analysis. Each contact surface can contain one or more of the following:

- The contact surface can include the exterior faces or "skin" of a block of finite elements. An exterior face is an outward facing finite element face that is not sandwiched between two contiguous finite elements. For more information on blocks skinning algorithms see Section 8.4.7.
- The contact surface can include a set of faces defined as a sideset on the input mesh. All faces included in the face set are considered for contact even if those faces are not exterior faces of the finite elements.
- The contact surface can include a set of nodes defined by a node set, surface, or block on the input mesh. These nodes need not be attached to finite element faces. The nodes could for example be the nodes of beam elements or particle elements.
- The contact surface can be a lofted set of faces around structural or lower dimensional elements. For example contact can define a lofted brick around a two dimensional shell element, a lofted prism around a one dimensional beam element, or lofted sphere around a zero dimensional particle element.
- The contact surface can be defined by an analytic function. More information on this capability can be found in the Sierra/SM 5.0 Capabilities in Development manual.

8.4.1. Contact Surface Command Line

```
CONTACT SURFACE <string>name CONTAINS <string list>surface_names
```

This command line identifies a set of sidesets and element blocks considered as a single contact surface; the string `name` is the unique name given to this contact surface. The list denoted by `surfaces_names` is a list of strings identifying surfaces and blocks that are to be associated with this contact surface. The surfaces can be side sets, element blocks, or any combination of the two as defined in the mesh file.

Any specified element blocks in the `surface_names` list that are solid elements (bricks, tetrahedra, wedges, etc.) are "skinned," meaning that a surface is created from all exterior faces of that element block. See Section 8.4.7 for more information on block skinning.

For element blocks in the `surface_names` list that are lower dimensional (shells, beams, and particles), a lofted geometry is created for use with contact enforcement.



Known Issue: The ACME contact algorithm is unable to loft beam and particle elements and will ignore any beam or particle element blocks. See Section 8.9.5 for more information on lofted geometry.

The name you create for a surface can be referenced in command blocks that specify how that surface will interact with other contact surfaces or interact when that surface contacts itself. See Section 8.6.1.2 and Section 8.6.2.1.

The surfaces can contain heterogeneous combinations of finite element faces and lofted geometry.



Warning: A given finite element face or lofted element can be placed in at most one contact surface. See Section 8.4.5 for more information on what happens if different contact surfaces try to include the same mesh objects.

8.4.2. Contact Node Set Command Line

```
CONTACT NODE SET <string>surface_name
  CONTAINS <string list>nodelist_names
```

The CONTACT NODE SET command line creates a contact surface named surface_name that contains a set of nodes. The list denoted by nodelist_names is a list of strings identifying node sets, surfaces, and blocks that are to be associated with the contact node set surface_name. All nodes present in the listed node sets, surfaces, or blocks will be added to the contact node set. The created node set contains only the mesh nodes of surfaces or blocks, not the faces, and not any lofted geometry. Thus, the contact node set can be used only in a node to face interaction.

8.4.3. Contact Surface Command Block

```
BEGIN CONTACT SURFACE <string>name
  INCLUDE ALL BLOCKS
  BLOCK = <string list>block_names
  ELEMENT = <int list>elem_numbers
  SURFACE = <string list>surface_names
  NODE SET|NODESET = <string list>node_set_names
  NODE = <int list>node_numbers
  REMOVE BLOCK = <string list>block_names
  REMOVE SURFACE = <string list>surface_names
  REMOVE NODE SET = <string list>node_set_names
  SUBSET <string>subname WITH NORMAL <real> <real> <real>
END [CONTACT SURFACE <string>name]
```

The CONTACT SURFACE command block can be used to define a contact surface consisting of a more complex collection of finite element faces or nodes than can be defined using the CONTACT SURFACE command line.

A surface named name that is a set of faces or lofted elements can be defined by a combination of the command lines INCLUDE ALL BLOCKS, BLOCK, SURFACE, ELEMENT, REMOVE BLOCK, and REMOVE SURFACE. Blocks listed in the block_names list or via INCLUDE ALL BLOCKS will

be skinned for exterior faces if the block contains solid elements (like hexahedra, tetrahedra, or wedges.) For blocks that contain shells, beams, or particles, the lofted geometry around those shells beams and particles will be included in the contact surface. The ELEMENT command can be used to pull individual lofted elements into the contact surface by element id.

If a BLOCK command line and a SURFACE command line are both specified within the same CONTACT SURFACE command block, the contact surface created will be the union of the faces in the skin of the blocks and the faces in the specified surfaces in the mesh file.

A surface named name that is a set of nodes without face geometry can be defined by a combination of the command lines NODE SET|NODESET, NODE, REMOVE NODE SET, REMOVE SURFACE, and REMOVE BLOCK. The NODE command can be used to pull individual nodes into the contact surface by node id.

The contact surface created by the CONTACT SURFACE command block must be either a set of faces and lofted geometry or a set of nodes. It cannot be a mix of the two. For example a single CONTACT SURFACE block may not contain both a NODE SET|NODESET and SURFACE command.

Contact surfaces may not be empty. At least one block, element, node, or surface must be defined in the input.

8.4.3.1. Surface Subsetting

```
SUBSET <string>subname WITH NORMAL <real> <real> <real>
```

A contact surface can optionally be subdivided into multiple sub-surfaces based on the surface normal. This command will create a new surface that contains the faces mostly aligned with the specified normal direction. Faces for which the dot product of the unitized face normal vector and the unitized specified normal is greater than 0.45 are placed in the new surface. All other faces are left in the original surface. For example, the following command:

```
BEGIN CONTACT SURFACE surf1
  BLOCK = block_1
  SUBSET xp WITH NORMAL  1  0  0
  SUBSET zm WITH NORMAL  0  0 -1
END
```

will create three contact surfaces. Surface surf1_xp will contain the faces most closely aligned with the positive x-axis. Surface surf1_zm will contain the faces mostly aligned with the negative z-axis. Surface surf1 will contain all remaining faces in the skin of block_1.

8.4.3.2. Known Issues

The BEGIN CONTACT SURFACE block may sometimes resolve contact faces slightly differently than the basic SKIN ALL BLOCKS or CONTACT SURFACE command. One potential distinction is faces interior to the model. When using global block skinning a face sandwiched between two

element blocks is not considered for contact. When faces are constructed with a more complex combination of individual block skins plus or minus side sets, those in-between block faces might be considered as contact faces. Generally the Dash contact enforcement algorithm will ultimately ignore any spurious internal faces if they exist, but these internal faces may cause fatal errors for the ACME contact enforcement algorithm. This consistency issue may be resolved in the future if as an unambiguous definition of how the skinning algorithm should behave in all addition/removal cases is developed.

8.4.4. Skin All Blocks Command

```
SKIN ALL BLOCKS = <string>ON|OFF(OFF)
  [EXCLUDE <string list>block_names]
```

The `SKIN ALL BLOCKS` command provides a convenient way to create a number of contact surfaces covering the exterior skin or lofted geometry of all or most of the element blocks in the analysis. This command may create many surfaces, where the name of each surface is the same as the element block it was generated from. The optional `EXCLUDE` keyword can be used to skin all element blocks except the listed blocks. For more information on how block skinning works see Section 8.4.7

The `SKIN ALL BLOCKS` is useful for large models in which individual specifying each contact surfaces would be unwieldy. The behavior of skin all blocks is best described by example. If a finite element analysis contains five element blocks:

- `block_1` (eight-noded hexahedral elements)
- `block_3` (tetrahedral elements)
- `block_4` (shell elements)
- `block_5` (beam elements)

The command:

```
SKIN ALL BLOCKS = ON EXCLUDE block_2
```

Would be equivalent to defining several independent surfaces with:

```
CONTACT SURFACE block_1 CONTAINS block_1
CONTACT SURFACE block_3 CONTAINS block_3
CONTACT SURFACE block_4 CONTAINS block_4
CONTACT SURFACE block_5 CONTAINS block_5
```

The generated contact surfaces would be named `block_1`, `block_3`, `block_4`, and `block_5`. The contact surfaces named `block_1` and `block_3` would contain the exterior faces of the solid element blocks `block_1` and `block_3`, respectively. The contact surfaces named `block_4` and `block_5` would contain lofted geometry around the shell and beam elements in `block_4` and `block_5`, respectively.

The `SKIN ALL BLOCKS` command can be combined with any of the other surface definition methods available in the `CONTACT DEFINITION` command block.

8.4.5. Faces and Nodes in Multiple Contact Surfaces

The contact enforcement algorithms in Sierra/SM only allow for a face, node, or lofted object to be associated with a single contact surface. If such a contact entity were allowed to belong to more than one contact surface, ambiguities would arise in determining the interaction properties for that entity.

A situation where an entity could potentially be placed in more than one contact surface is when overlapping sidesets are used in contact. This could also occur if a combination of skinned block surfaces, side sets, and node sets on the mesh file are used to define a contact surface. The side sets can potentially include the same finite element faces as the block skin.

If a given finite element face or node is potentially part of more than one contact surface, then that face or node is included in the first contact surface to which it belongs that is listed in the `CONTACT DEFINITION` block. If the same face or node also potentially belongs to another contact surface listed later in the `CONTACT DEFINITION` block, it will not be included again in this later contact surface. A warning message is generated in the log file for any potentially ambiguous cases.

Use of overlapping contact surfaces can be used to implement more advanced behavior such as surfaces tied together at local patches.

The behavior described above is best illustrated by the following examples.

Example 1: Consider the model setup shown in Figure 8-2a. In the contact definition block, the contact surfaces are defined by the following input commands:

```
contact surface surfB contains surface_2
contact surface surfA contains block_1
```

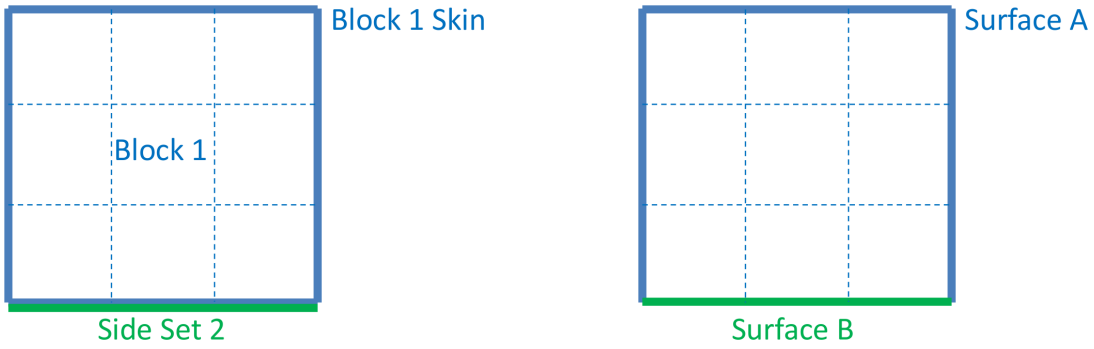
This results in a set of surfaces as shown in Figure 8-2b. The contact surface `surfB` is listed first in the input deck. The faces on side set 2 are placed into `surfB`, as `surfA` is listed second in the input deck. The remaining faces on the skin are placed in `surfA`. This means that the faces that are in both surface_2 and in block_1 are only placed in `surfB`. The sub-setting mechanism shown in this example may be used to specify a different friction model, such as tied contact, on a small piece of a larger part.

Example 2: A more complex example is shown in Figure 8-3a. Consider the following two contact surface definitions:

Contact surface definition 1:

```
contact surface surfD contains surface_4
contact surface surfB contains surface_2
contact surface surfA contains block_1
contact surface surfC contains block_2
```

Contact surface definition 2:



(a) Setup **(b)** Results

Figure 8-2. Faces in Multiple Contact Surfaces: Example 1

```
contact surface surfB contains surface_2
contact surface surfA contains block_1
contact surface surfC contains block_2
contact surface surfD contains surface_4
```

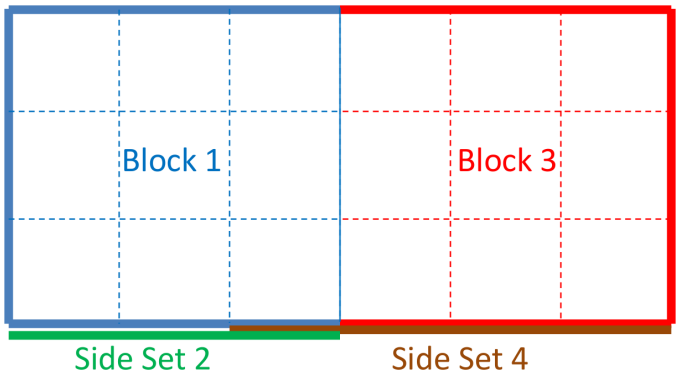
The first contact definition results in the organization of surfaces, as shown in Figure 8-3b. The second contact definition results in a different organization of the contact surfaces, as show in Figure 8-3c. Unlike most commands, the order in which the contact surface definition commands are placed in the input deck may change the analysis behavior. One final note, contact surface definition two above would also be equivalent to the following:

```
contact surface surfB contains surface_2
skin all blocks = on
contact surface surfD contains surface_4
```

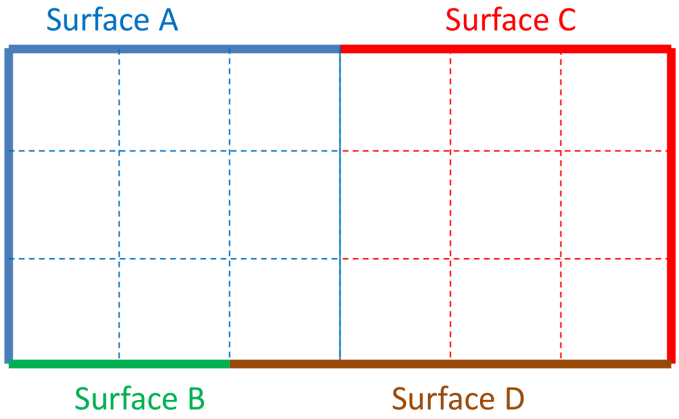
If the skin all blocks command is present, that command is equivalent to specifying contact on all blocks individually.

8.4.6. Element Death and Surface Updates

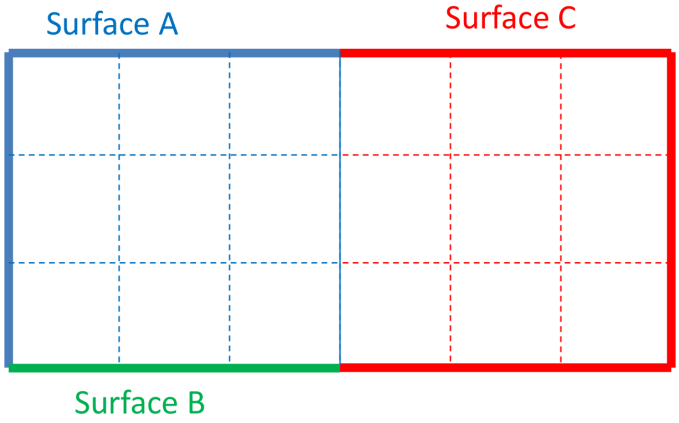
When elements are killed in an analysis, contact surfaces may need to be updated to account for the removal of faces attached to killed elements or the addition of faces exposed by element death. The updates happen automatically when elements in a skinned block are removed due to element death (See Section 6.5). The update of these surfaces requires a full re-initialization of contact. Thus, surface-physics models that involve state data may lose some information when the new contact surfaces are created. If contact surfaces are defined by side sets rather than by block skinning, element death will cause the facets attached to killed elements to be removed from the contact surface, but no new facets will be added.



(a) Setup



(b) Results 1



(c) Results 2

Figure 8-3. Faces in Multiple Contact Surfaces: Example 2

8.4.7. Block Skinning and Surface Definition Examples

A surface is defined as a collection of finite element faces, nodes, and lofted geometry. For a continuum element, any face that is not shared with another element can be considered for contact. A shell element has both a top face and a bottom face. Top, bottom, and side surfaces of shells are automatically created by the contact algorithm and may be lofted by a user-specified thickness. Contact is enforced on shells by computing the contact forces on the top and bottom

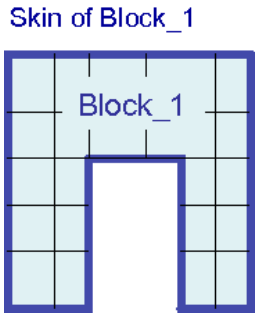


Figure 8-4. Simple block skinning example

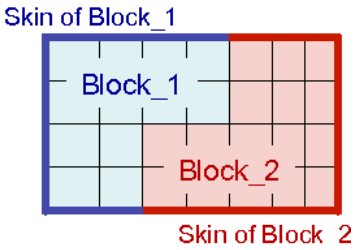


Figure 8-5. Block skinning with multiple blocks example

surfaces of the shells and then moving the resulting forces back to the original shell nodes.

The easiest method to create contact surfaces is by skinning blocks. Block skinning identifies the exterior surfaces of a block and puts them in a contact surface. The following examples demonstrate how block skinning works in various circumstances.

8.4.7.1. Simple Block Skinning

See Figure 8-4. This first example is a single block being skinned. This skinning could occur by one of the input lines

```
CONTACT SURFACE block_1 CONTAINS block_1
```

or if the model contains only `block_1`

```
SKIN ALL BLOCKS = ON
```

For either syntax all the exterior facets of `block_1` are automatically identified and are placed into a contact surface which is also named `block_1`.

8.4.7.2. Block Skinning With Multiple Element Blocks

Figure 8-5 shows an example of two blocks that are equivalenced on their interface. An equivalenced interface means that the two blocks share nodes at the interface so that there are no exposed contact faces between `block_1` and `block_2`. The syntax to create the contact surfaces shown in Figure 8-5 is:

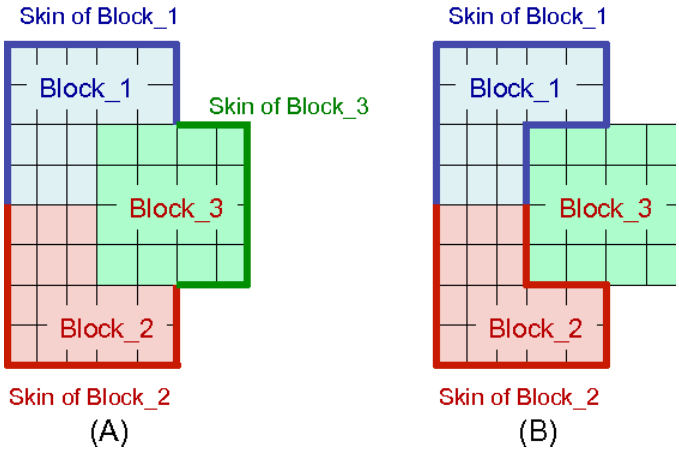


Figure 8-6. Block skinning advanced example

```
CONTACT SURFACE block_1 CONTAINS block_1
CONTACT SURFACE block_2 CONTAINS block_2
```

Alternatively, if the model contains only `block_1` and `block_2`, this could be accomplished by requesting that all blocks be skinned:

```
SKIN ALL BLOCKS = ON
```

For either syntax, all the exterior facets that are connected to elements that are part of `block_1` are placed in contact surface `block_1`. All the exterior facets that are connected to elements of `block_2` are placed in contact surface `block_2`. Any interior facet (one that is sandwiched between two elements) is not placed in any contact surface.

8.4.7.3. More Complex Block Skinning Cases

Block skinning considers only those elements actually present in the contact definition when deciding if a face is exposed. The surfaces shown by heavy lines in Figure 8-6(A) will result from the following syntax:

```
CONTACT SURFACE block_1 CONTAINS block_1
CONTACT SURFACE block_2 CONTAINS block_2
CONTACT SURFACE block_3 CONTAINS block_3
```

All the exterior facets of the model have been placed in the contact surfaces associated with elements of the named blocks. Any facet sandwiched between two contact surface blocks is not considered to be on the exterior, and is thus not placed in any contact surface. The surfaces shown in Figure 8-6(B) will result from the following syntax:

```
CONTACT SURFACE block_1 CONTAINS block_1
CONTACT SURFACE block_2 CONTAINS block_2
```

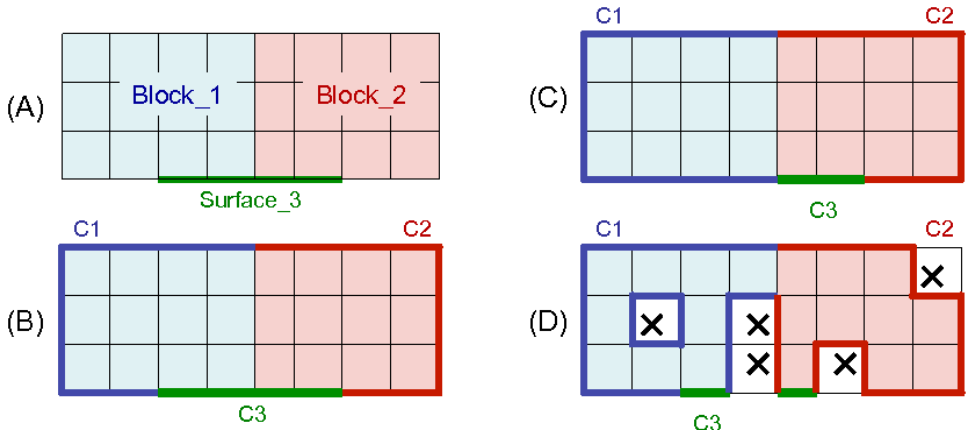


Figure 8-7. Contact surface definition examples

or

```
SKIN ALL BLOCKS = ON EXCLUDE block_3
```

In this case, `block_3` was not included in the contact definition. Thus the faces on the `block_1` `block_3` and `block_2` and `block_3` boundaries will be considered exterior for the purposes of contact. Since `block_3` is not part of the contact definition none of the remaining faces of `block_3` will be part of any contact surface.

8.4.7.4. Combining Block Skinning, Side Sets, and Element Death

Figure 8-7(A) shows a set of element blocks and surface sidesets as defined in a mesh file. Blocks `block_1` and `block_2` are equivalenced sharing nodes on their boundary. The surface `surface_3` is a sideset defined on part of the exterior of `block_1` and `block_2`. The contact surfaces shown in Figure 8-7(B) will be generated by the following syntax:

```
CONTACT SURFACE c3 CONTAINS surface_3
CONTACT SURFACE c1 CONTAINS block_1
CONTACT SURFACE c2 CONTAINS block_2
```

As described in Section 8.4.5, the first surface defined in the input file that includes a facet owns that facet. If any subsequently defined contact surfaces attempt to include a facet that already belongs to a contact surface, it will be left in the original contact surface that it belongs to, and will not be made part of the surface that is defined later in the input file. In other words, a given mesh facet can belong in at most one contact surface, and when a conflict arises, the facet will belong to the first contact surface in the input file that included it. Thus, `c3` contains the facets associated with `surface_3`. Contact surfaces `c1` and `c2` do not contain any of the facets of side set `surface_3`.

The surfaces in Figure 8-7(C) are be generated by the following syntax:

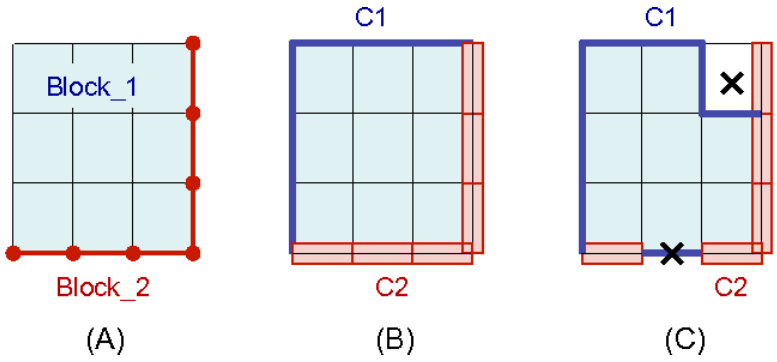


Figure 8-8. Hexahedral and shell contact surface definitions

```
CONTACT SURFACE c1 CONTAINS block_1
CONTACT SURFACE c3 CONTAINS surface_3
CONTACT SURFACE c2 CONTAINS block_2
```

Since contact surface `c1` is listed before contact surface `c3` in the input file, `c1`, the skin of `block_1`, contains the contact facets that could have been part of `c1` or `c3`. Since contact surface `c3` is listed before contact surface `c2` in the input file, contact surface `c3` includes the remaining portions of `surface_3`, while `c2`, the skin of `block_2`, contains only the remaining exterior facets of `block_2`.

When element death occurs, new contact facets may be exposed and old contact facets may be deleted. After element death, the ownership of contact facets in the updated contact surfaces will be evaluated as if the dead elements never existed. Figure 8-7(D) shows the effect of element death on the contact surfaces. For this case the contact surfaces are defined by the same syntax as the case shown in Figure 8-7(B):

```
CONTACT SURFACE c3 CONTAINS surface_3
CONTACT SURFACE c1 CONTAINS block_1
CONTACT SURFACE c2 CONTAINS block_2
```

Elements shown in the figure with an X are dead elements. Any newly exposed facets of `block_1` and `block_2` are added to the relevant contact surfaces. If all the elements attached to contact facet are deleted, the contact facet itself is also deleted. If a contact surface is defined via a side set, that surface can only lose faces. In other words, a contact surface defined by a contact side set cannot gain any faces that were not part of the original side set during the process of element death.

8.4.7.5. Equivalenced Solid and Shell Meshes

A mesh that includes both solid elements such as hexahedral and shell elements obeys some special rules when defining contact surfaces via block skinning. Figure 8-8(A) shows `block_1`, (solid hexahedral elements) and `block_2` (quadrilateral shell elements). The nodes of the shell element block are equivalenced with the nodes of the hexahedral element block, or in other

words, the two blocks share nodes on their interface. Figure 8-8(B) shows the contact surfaces that would be generated using the following commands:

```
CONTACT SURFACE c1 CONTAINS block_1
CONTACT SURFACE c2 CONTAINS block_2
```

No matter what the order of the contact surfaces in the input file is, the shell block will always be considered "outside" the solid element block for this case. On portions of `block_1` without any cladding shells, the standard block skinning rules apply. On the interface between `block_1` and `block_2` the quad faces on the `block_1` hexahedral elements are considered sandwiched between the `block_1` hexahedral elements and the `block_2` shell elements. Thus, `c1` contains none of the interface facets. Contact surface `c2` contains the lofted shell geometry associated with the shell elements of `block_2`.

In Figure 8-8(C), some elements have been removed due to element death. As in the example in Section 8.4.7.4, when elements are killed via element death the contact surfaces are reevaluated as if those elements were never there. When a cladding shell is removed the exterior face of `block_1` is exposed and joins contact surface `c1`. The death of one of the lofted shell elements removes all faces associated with that element's lofted geometry.

8.5. Friction Models

Friction models are used to describe the physics of interactions that occur between contact surfaces. The user then associates these friction models to pairs of surfaces in the interaction-definition blocks (see Section 8.6.1 and Section 8.6.2). During the search phase of contact, node-face or face-face interactions are identified, and the designated friction model is used to determine how the resulting contact forces are resolved between these surface pairs.

Not all friction models are available with all analysis types and all contact libraries. If a model is not available for certain analysis types, that fact is noted in the documentation of those models below.



In addition to the friction models, any of the cohesive zone models available for use with interface elements (see Section 5.3) can be used as interface models within explicit dynamic analyses. See Section 8.6.2.5 for more details on the usage of those models within contact.

All friction models are command blocks, although some of the models do not have any command lines inside the command block. The commands for defining the available friction models are described next. Friction models are associated with specific pairings of contact surfaces through the interaction-definition blocks in Section 8.6.1 and Section 8.6.2.

By default, interactions between contact surfaces with no friction model assigned are treated as frictionless.

8.5.1. Frictionless Model

```
BEGIN FRICTIONLESS MODEL <string>name
END [FRICTIONLESS MODEL <string>name]
```

The `FRICTIONLESS MODEL` command block defines frictionless contact between surfaces. In frictionless contact, contact forces are computed normal to the contact surfaces to prevent penetration, but no forces are computed tangential to the contact surfaces. The string `name` is a user-selected name for this friction model that is used when identifying this model in the interaction definitions. No command lines are needed inside the command block. A default named frictionless model named `frictionless` can be used without defining this command block.

8.5.2. Constant Friction Model

```
BEGIN CONSTANT FRICTION MODEL <string>name
  FRICTION COEFFICIENT = <real>coeff
END [CONSTANT FRICTION MODEL <string>name]
```

The `CONSTANT FRICTION MODEL` command block defines a constant coulomb friction coefficient between two surfaces as they slide past each other in contact. No resistance is provided to keep the surfaces together if they start to separate. The string `name` is a user-selected name for

this friction model that is used to identify this model in the interaction definitions, and `coeff` is the constant coulomb friction coefficient. There is no default value for the friction coefficient.



8.5.3. Velocity Dependent Coulomb Friction Model

```
BEGIN VELOCITY DEPENDENT COULOMB MODEL <string>name
  COULOMB COEFFICIENT = <string>coeff_func
END [VELOCITY DEPENDENT COULOMB MODEL <string>name]
```

The `VELOCITY DEPENDENT COULOMB MODEL` command block defines a Coulomb friction law between two surfaces as they slide past each other in contact. The value of the Coulomb coefficient at any given time is a function of the relative velocity at the interface. No resistance is provided to keep the surfaces together if they start to separate. The string `name` is a user-selected name for this friction model that is used to identify this model in the interaction definitions. The `COULOMB COEFFICIENT` command provides `coeff_func`, the name of a previously defined XY function, where X is the velocity magnitude and Y is the coulomb coefficient. The velocity dependent coulomb model can be used to define stick-slip friction behavior where the friction coefficient is the stick value when velocity is nominally small and the slip value when velocity is larger than some nominal value.



8.5.4. Variable Coefficient Coulomb Model

```
BEGIN VARIABLE COEFFICIENT COULOMB MODEL <string>name
  FUNCTION ARGUMENT: <name> = SIDE A NODAL <var_name>
  FUNCTION ARGUMENT: <name> = SIDE B  NODAL <var_name>
  COEFFICIENT FUNCTION      = <string>coeff_func
END VARIABLE COEFFICIENT COULOMB MODEL <string>name]
```

The `VARIABLE COEFFICIENT COULOMB MODEL` command block can be used to define a Coulomb friction law where the friction coefficient is dependent on some arbitrary nodal variable. For example, a different coefficient based on spatial coordinates or from some surface dependent property. See section 2.1.6 for more information on generalized functions. An example of using this friction model to compute a friction model that depends on an oxidation state nodal surface variable follows.

```
#
#  Friction coefficient based on oxide state.  Assume
#  oxide thickness < 1.0e-6 means "clean" surface
#  and thickness > 1.0e-6 is a rough oxidized surface.
#  Friction coefficient is then subject to following rules:
#
#  oxide against oxide: u = 0.4
#  oxide against clean: u = 0.2
#  clean against clean: u = 0.1
#
```

```
begin function oxide_fric_coeff
  type is analytic
  expression variable: ox1
  expression variable: ox2
  evaluate expression = "    #
    (ox1 < 1.0e-6) ? (       #
      (ox2 < 1.0e-6) ? (     #
        0.1                  #
      ) : (                  #
        0.2                  #
      )                      #
    ) : (                    #
      (ox2 < 1.0e-6) ? (     #
        0.2                  #
      ) : (                  #
        0.4                  #
      )                      #
    )                        #
  "
end

begin variable coefficient coulomb model oxide_fric
  coefficient function = oxide_fric_coeff
  Function Argument: ox1 = side A nodal oxide_thick
  Function Argument: ox2 = side B  nodal oxide_thick
end
```

8.5.5. Tied Model

```
BEGIN TIED MODEL <string>name
END [TIED MODEL <string>name]
```

The `TIED` friction model attaches together surfaces that are close in the initial model geometry. Tied faces will stay attached throughout the analysis. The string `name` is a user-selected name for this friction model that is used to identify this model in the interaction definitions. No command lines are needed inside the command block. A default tied model named `tied` can be used without defining this command block.

8.5.6. Inactive Friction Model

```
BEGIN INACTIVE FRICTION MODEL <string>name
END [INACTIVE FRICTION MODEL <string>name]
```

The `INACTIVE FRICTION MODEL` command block specifies that the interacting surfaces product no force and contact is effectively ignored on the interaction pair. The inactive friction

model can be used to selectively turn off interaction pairs. One common use is integrating the inactive friction model to a `TIME VARIANT` friction model to turn contact interactions off selectively by time period.



8.5.7. Dynamic Tied Model

```
BEGIN DYNAMIC TIED MODEL <string>name
END [DYNAMIC TIED MODEL <string>name]
```

The `DYNAMIC TIED MODEL` command block defines a contact interaction that constrains opposing faces to stay in the same relative location to each other throughout the analysis. No command lines are needed within this command block. A default dynamic tied model named `dynamic_tied` can be used without defining this command block. The dynamic tied model is similar to the tied model except that the dynamic tied model enforces zero relative velocity at the interface while the tied model enforces zero relative displacement. In the limit, both of these models are identical. If the analysis involves substantial topology changes, such as element death, the dynamic tied model may be more stable, but slightly less accurate.

8.5.8. Glued Model

```
BEGIN GLUED MODEL <string>name
END [GLUED MODEL <string>name]
```

Note, this model is available for explicit and implicit analyses. For implicit analyses, it can only be used with augmented Lagrange enforcement.

The `GLUED` model defines a contact interaction that allows the interacting faces to move independently until they come into contact, at which point they become permanently attached together. Once attached the faces can have no relative normal or tangential motion for the rest of the analysis. The string `name` is a user-specified name for this friction model that is used to reference this model in the interaction definitions. No command lines are needed inside the command block. A default glued model named `glued` can be used without defining this command block. The glued model is enforced via a zero-velocity constraint.



Warning: Because glued contact is set up using the predicted configuration during an implicit simulation the loadstep must be turned off by setting the loadstep predictor to:

```
begin loadstep predictor
type = scale_factor
scale factor = 0 0
end loadstep predictor
```

8.5.9. Glued Normal Model

```
BEGIN GLUED NORMAL MODEL <string>name
END [GLUED NORMAL MODEL <string>name]
```

The GLUED NORMAL model defines a contact interaction that allows the interacting faces to move independently until they come into contact, but once they come into contact they become permanently attached together, but are allowed to slide along each other. Once attached the faces can have no relative normal motion for the rest of the analysis. The string name is a user-specified name for this friction model that is used to reference this model in the interaction definitions. No command lines are needed inside the command block.



Warning: Because glued normal contact is set up using the predicted configuration during an implicit simulation the loadstep must be turned off by setting the loadstep predictor to:

```
begin loadstep predictor
type = scale_factor
scale factor = 0 0
end loadstep predictor
```

8.5.10. Cohesive Zone Model

```
BEGIN COHESIVE ZONE MODEL <string>name
  TRACTION DISPLACEMENT FUNCTION = <string>func_name
  TRACTION DISPLACEMENT SCALE FACTOR = <real>scale_factor(1.0)
  CRITICAL NORMAL GAP = <real>crit_norm_gap
  CRITICAL TANGENTIAL GAP = <real>crit_tangential_gap
END [COHESIVE ZONE MODEL <string>name]
```

The COHESIVE ZONE MODEL command block defines a friction model that prevents penetration when contact surfaces are touching, and provides an additional force when the distance between the node and face in an interaction increases. This force is determined by a user-specified function. Once the distance exceeds a user-specified value in the normal direction or the tangential direction, the force is no longer applied. This model can be used to simulate the energy required to separate two surfaces that are initially touching.

In the above command block, the string name is a user-selected name for this friction model that is used to identify this model in the interaction definitions. The traction-separation relationship is given by the TRACTION DISPLACEMENT FUNCTION command line, where the string func_name is the name of a function defined in a FUNCTION command block in the SIERRA scope. The y-values of this function can be scaled by the real value scale_factor in the TRACTION DISPLACEMENT SCALE FACTOR command line; the default for this factor is 1.0. In the CRITICAL NORMAL GAP command line, the real value crit_norm_gap specifies the normal distance between the node and face past which the cohesive zone no longer provides a

force. Similarly, in the `CRITICAL TANGENTIAL GAP` command line, the real value `crit_tangential_gap` specifies the tangential distance between the node and face past which the cohesive zone no longer provides a force.

The function referenced by the `TRACTION DISPLACEMENT FUNCTION` command line must be an XY function (see Section 2.1.5). The X-values correspond to ϵ, defined as:

$$\epsilon = \frac{current\ \ displacement}{critical\ \ gap\ \ length} \tag{8.1}$$

ϵ is a non-dimensional quantity which should start at zero corresponding to zero traction, and end at 1.0 corresponding to the critical gap length. The Y-values are the ϵ-dependent tractions between faces sharing a cohesive zone. All traction and separation quantities should be positive.

The following example is the problem shown in Figure 8-9: 2 bricks being pulled apart that are initially connected by a cohesive zone.

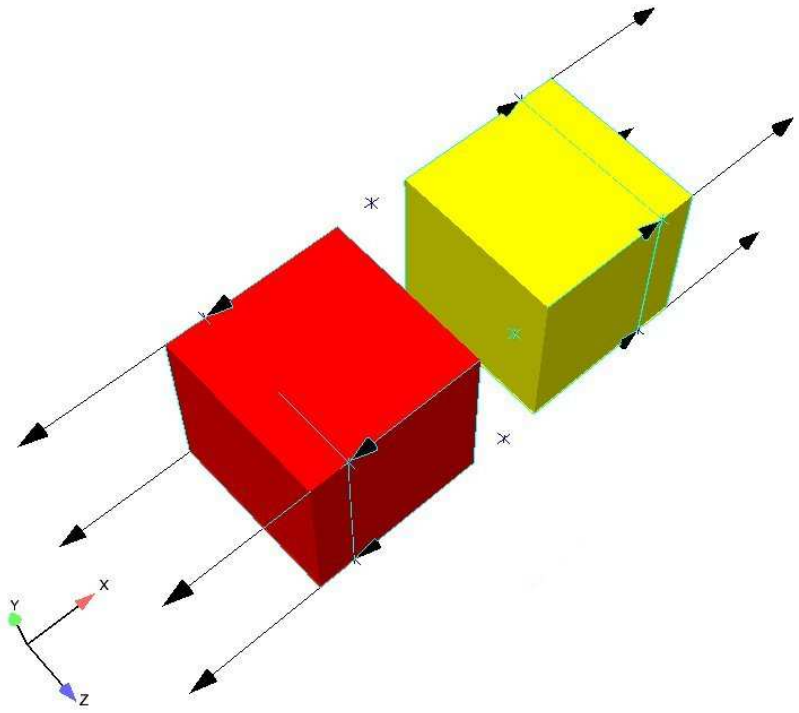


Figure 8-9. Bricks sharing a cohesive zone and being pulled apart.

The function describing the cohesive zone is given by:

```
begin function spring_restore
  type = piecewise linear
  begin values
    # separation(epsilon) traction
    0.0   0.0
    0.1   1.0
```

```
    0.3   9.0
    0.4  16.0
    0.5  25.0
    0.9  81.0
    1.0 100.0
  end values
end function spring_restore
```

The cohesive zone model is given by:

```
begin cohesive zone model cohesive_zone
  critical normal gap = 0.05
  critical tangential gap = 0.05
  traction displacement function = spring_restore
  traction displacement scale factor = 2.5E+02
end cohesive zone model cohesive_zone
```

A plot of the x-displacement for node 8 (one of the nodes on the cohesive zone) over time is given in Figure 8-10. The corresponding cohesive force in the x-direction over time for node 8 is shown in Figure 8-11 .

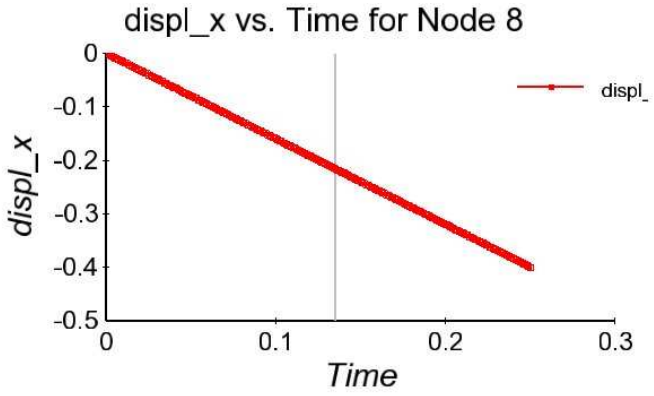


Figure 8-10. Displacement X vs. time for node 8

As expected, when the displacement exceeds the critical normal gap of .05, which happens at .016 seconds, the cohesive force goes to zero. The cohesive zone force peaks at 6250, which is equal to:

$$\frac{force\ at\ critical\ gap \times traction\ displacement\ scale\ factor}{number\ of\ nodes\ on\ face} = \frac{100 \times 250}{4}. \qquad (8.2)$$

8.5.11. Hybrid Model

```
BEGIN HYBRID MODEL <string>name
  INITIALLY CLOSE = <string>close_name
```

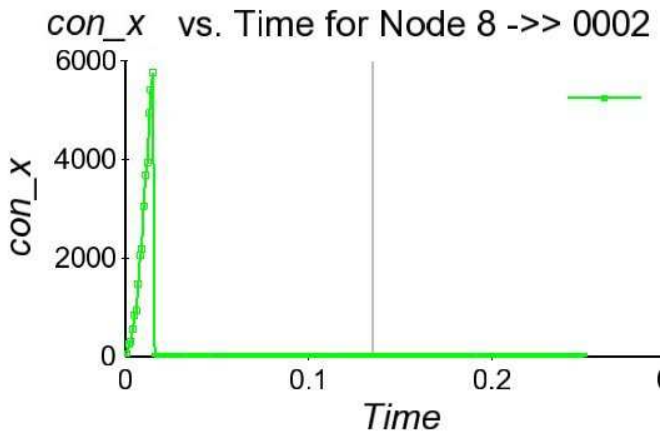


Figure 8-11. Cohesive force in x vs. time for node 8

```
  INITIALLY FAR   = <string>far_name
END [HYBRID MODEL <string>name]
```

The `HYBRID MODEL` command block defines a single friction model that is a composite of two other models: one for faces that are initially close together, and one for faces that are initially far apart. The friction law specified in the `INITIALLY CLOSE` command is used for faces that are close together in the initial mesh geometry. The friction law specified in the `INITIALLY FAR` command is used for faces that are far apart in the initial mesh geometry.

A common usage of the hybrid model is to tie together parts of two surfaces that are close together, while allowing the remaining parts of those surfaces to have a sliding contact interaction. If the tied model were used in that situation, the faces that are initially close would be tied together, but no contact would be enforced on the remaining faces on those surfaces, so they could pass through one another. Another potential usage of the hybrid model would be to define a high friction coefficient for faces that are initially close, and then use a lower friction coefficient once the faces start to slide away from the initial configuration.

A hybrid model is automatically created when an interaction-specific and a default friction model are both specified and the interaction-specific friction law is only applicable when the surfaces are initially close together. For example, the hybrid model contact interaction definition:

```
BEGIN HYBRID MODEL hmod
  INITIALLY CLOSE = tied
  INITIALLY FAR   = frictionless
END

BEGIN INTERACTION
  SURFACES = block_1 block_2
  friction model = hmod
END
```

Is equivalent to:

```
BEGIN INTERACTION DEFAULTS
  FRICTION MODEL = frictionless
END
BEGIN INTERACTION
  SURFACES = block_1 block_2
  friction model = tied
END
```

The hybrid model definition above was used for the example model shown in Figure 8-12.

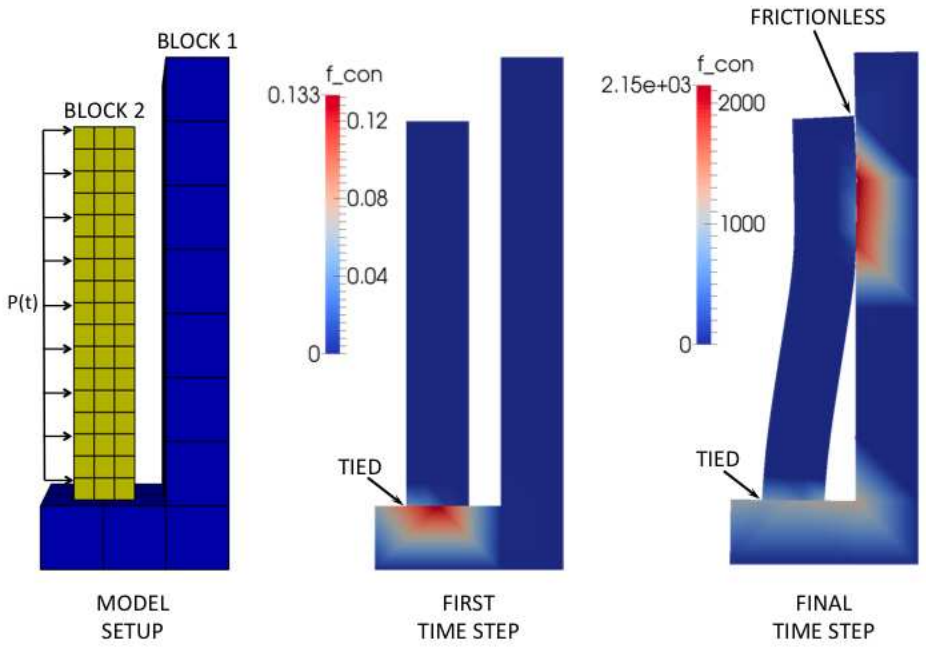


Figure 8-12. Example usage of the hybrid friction model. The contour plots show the magnitude of the contact force.



Warning: Using the hybrid friction model may cause your simulation to have poor performance. This is especially the case when you have a large number of element blocks.

8.5.12. Time Variant Model

```
BEGIN TIME VARIANT MODEL <string>name
  MODEL = <string>model DURING PERIODS <string list>time_periods
END [TIME VARIANT MODEL]
```

Note: This feature is only available for Dash contact with explicit dynamics and Dash augmented Lagrange contact for implicit analyses.

The TIME VARIANT MODEL command block does not directly define a friction model, but provides a mechanism to switch between separately defined friction models between time periods. The only command line in the block is the MODEL command, which is used to reference a separately-defined friction model model and provide a list of time periods during which that model is active. Multiple MODEL commands may be used to activate different friction models throughout the analysis. See Section 2.6 for more information on controlling functionality by time period.



Warning: Using the time variant friction model may cause your simulation to have poor performance. This is especially the case when you have a large number of element blocks.

8.5.13. Nodal Field Dependent Model

```
BEGIN NODAL FIELD DEPENDENT FRICTION MODEL <string>name
  INITIAL CONTACT TRANSITION VALUE = <real>transition_value
  CONTACT TRANSITION REDUCTION METHOD =
    <string>MIN|MAX|INTERPOLATE(MAX)
END [NODAL FIELD DEPENDENT FRICTION MODEL]
```

Note: This feature is only available for Dash contact with explicit dynamics and Dash augmented Lagrange contact for implicit analyses.

The NODAL FIELD DEPENDENT FRICTION MODEL command block allows the opportunity to vary the friction coefficient and switch between Coulomb and glued friction conditions based on the value of a specific nodal variable, named contact_transition. For every interaction pair for which this friction model is applied, the contact_transition field is evaluated on the side B nodes to determine the type of contact: a non-negative contact_transition value results in Coulomb contact and a negative value results in glued contact. For Coulomb contact, the coefficient of friction is set equal to the contact_transition value.

The contact_transition field is controlled by the user. The input command INITIAL CONTACT TRANSITION VALUE sets an initial value for all nodes in the model. The field can then be varied spatially and temporally by a USER OUTPUT command block (Section 9.3.2), where contact_transition is the variable name. This allows the user to define contact_transition as a function of other field variables. For example, the following input commands make the contact conditions dependent on the temperature of element block block_1:

```
BEGIN FUNCTION my_temperature
  TYPE IS PIECEWISE ANALYTIC
  BEGIN EXPRESSIONS
    # time  temperature
    0.0     "600.0"
    0.00006 "600.0+1400.0*(x-0.00006)/0.00394"
```

```
    0.004   "2000.0-1600.0*(x-0.004)/0.0005"
    0.0045  "400.0"
  END
END FUNCTION my_temperature

BEGIN FUNCTION temp_dependent_fric
  TYPE IS ANALYTIC
  EXPRESSION VARIABLE: q = GLOBAL temperature_max
  EVALUATE EXPRESSION IS "q > 500.0 ? (2000.0-q)/4000.0 : -1.0"
END FUNCTION temp_dependent_fric

BEGIN PRESCRIBED TEMPERATURE
  BLOCK = block_1
  FUNCTION = my_temperature
END PRESCRIBED TEMPERATURE

BEGIN USER OUTPUT
  BLOCK = block_1
  COMPUTE GLOBAL temperature_max AS MAX OF NODAL temperature
END USER OUTPUT

BEGIN USER OUTPUT
  BLOCK = block_1 block_2
  COMPUTE NODAL contact_transition AS FUNCTION temp_dependent_fric
END USER OUTPUT

BEGIN CONTACT DEFINITION slide_and_glued
  SEARCH = DASH
  SKIN ALL BLOCKS = ON

  BEGIN INTERACTION
    SURFACES = block_1 block_2
    FRICTION MODEL = temperature_dependent
  END INTERACTION

  BEGIN NODAL FIELD DEPENDENT FRICTION MODEL temperature_dependent
    INITIAL CONTACT TRANSITION VALUE = 0.35
    CONTACT TRANSITION REDUCTION METHOD = MAX
  END NODAL FIELD DEPENDENT FRICTION MODEL temperature_dependent
END CONTACT DEFINITION slide_and_glued
```

In this example, the friction conditions are Coulomb at above 500.0 degrees and glued at 500.0 degrees or less. Above 500.0 degrees, the friction coefficient varies linearly with temperature. The temperature of `block_1` starts out above 500.0 degrees and varies significantly causing a variation in the friction coefficient, until a time just prior to 0.0045 when the friction conditions become glued. The temperature of `block_1` is stored as the global variable `temperature_max` so that both blocks involved in the interaction, `block_1` and `block_2`, will have the correct

contact_transition value, which is computed by calling the temperature_dependent_friction function in a USER OUTPUT command block. By default, the Dash algorithms internally determine the side A and side B surfaces of a contact interaction, so it is necessary to set the contact_transition field to the desired values on both surfaces involved in the interaction.

The input command CONTACT TRANSITION REDUCTION METHOD defines how the nodal contact_transition field is reduced to a single scalar at the point of interaction inside a face. The MIN and MAX options take the minimum and maximum, respectively, of the face's nodal values. The INTERPOLATE option interpolates the nodal values to the point of interaction.

8.6. Definition of Interactions

Each contact surface can interact with itself and other contact surfaces subject to the conditions specified in a contact interaction. A contact interaction must specify which surfaces should be considered for interaction and parameters governing the interaction.

Defining contact in complex models can be greatly simplified by using "general" contact, in which a single command is used to indicate possible contact between one contact surface and every other contact surface in the model. This command used with a default interactions block to define parameters for the interaction behavior can define contact for an entire model in only a few lines of input.

It is recommended that default interactions blocks be used as much as possible to keep the definition of individual interactions to a minimum. However, if needed, contact interactions can also be defined individually for specific sets of contact surfaces. These can be used to override parameters set in the default command block or can be used without specifying any default parameters. The latter may be more efficient if there are relatively few surfaces in a model and those surfaces interact with a small number of other surfaces, but for complex models with many potential contact surfaces, individually defining all possible contact interactions can be a tedious, error-prone process.

Contact can also be enforced between a surface and itself. This is known as "self" contact, and is used to prevent a part of a surface from penetrating another part of that surface when it folds in on itself. As a general rule, enforcing self contact is more expensive than enforcing contact between different surfaces.

Contact surfaces can be based either on sets of nodes or faces. Node-based surfaces can only interact with face-based surfaces. Face-based surfaces can interact with node or face based surfaces.

The following sections document the commands used to define contact interactions and the properties of those interactions.

8.6.1. Interaction Defaults Block

```
BEGIN INTERACTION DEFAULTS
  GENERAL CONTACT = <string>ON|OFF(OFF)
  SELF CONTACT = <string>ON|OFF(OFF)
  SURFACES = <string list>surface_names
  FRICTION MODEL = <string>friction_model_name(FRICTIONLESS)
  INTERACTION BEHAVIOR = <string>SLIDING|NO_INTERACTION(SLIDING)
  CONSTRAINT FORMULATION = <string>NODE_FACE|FACE_FACE
  AL PENALTY = <real>normalPenalty(1.0) [<real>tangentialPenalty]
END [INTERACTION DEFAULTS]
```

Implicit

The `INTERACTION DEFAULTS` command block enables contact enforcement either on all contact surfaces or on a subset of the contact surfaces. It is also used to set default parameters for contact

interactions. These defaults can be overridden for specific interactions by defining them in INTERACTION blocks. Note that contact enforcement will not occur in a model unless some combination of the INTERACTION DEFAULTS command block and INTERACTION command blocks (see Section 8.6.2) is used to set up surface interactions.

The values specified by the command lines in the INTERACTION DEFAULTS command block are applied by default to all interaction contact surfaces unless overridden by a specific interaction definition. Note that if a friction model is specified for a specific interaction, but that friction model is relevant only for some of the faces in the surfaces involved in the interaction, the friction model specified in the interaction defaults will be used for the remaining faces. For example, if a tied model is specified for an interaction, only the faces that are initially close will be tied. The remaining faces will be enforced using the default friction model. See Section 8.5.11 for more information.

8.6.1.1. General Contact and Self-Contact

```
GENERAL CONTACT = <string>ON|OFF(OFF)
SELF CONTACT = <string>ON|OFF(OFF)
```

Setting GENERAL CONTACT to ON specifies that default values set in the command lines of this INTERACTION DEFAULTS command block apply all surfaces defined in the enclosing CONTACT DEFINITION block or to a subset of those surfaces specified in a CONTACT SURFACES line. The default value is OFF. The GENERAL CONTACT command does not impose self-contact.

To enforce self-contact for all surfaces in the enclosing CONTACT DEFINITION block or for a subset of those surfaces specified in a CONTACT SURFACES line, the following command line must be present:

```
SELF CONTACT = ON
```

The default value for SELF CONTACT is OFF.

If general contact or self-contact is enabled, contact enforcement may be disabled for individual surfaces or sets of surfaces by using the INTERACTION BEHAVIOR = NO_INTERACTION line in specific INTERACTION command blocks.

8.6.1.2. Specific Surfaces Identification

```
SURFACES = <string list>surface_names
```

This command line identifies the contact surfaces to which the surface interaction values defined in the INTERACTION DEFAULTS command block will apply. The surface_names list can include any surface specified in a CONTACT SURFACE command line, a CONTACT SURFACE

command block, or created automatically by a SKIN ALL BLOCKS command line in the enclosing BEGIN CONTACT DEFINITION scope.

In the absence of a SURFACES command line, the interaction defaults will apply to all contact surfaces defined in the enclosing BEGIN CONTACT DEFINITION scope.

8.6.1.3. Friction Model

```
FRICTION MODEL = <string>friction_model_name(FRICTIONLESS)
```

The FRICTION MODEL command line specifies how surfaces interact with each other. Several simple friction models can be used for friction_model_name including FRICTIONLESS, TIED, and GLUED. Alternatively friction_model_name can be the name of a more complex friction model defined elsewhere in the CONTACT DEFINITION command block. See Section 8.5 for a listing of the available friction models.

The following set of commands will apply the Coulomb friction model fric1 to all surface interactions.

```
BEGIN CONSTANT FRICTION MODEL fric1
 COULOMB COEFFICIENT = 0.2
END

BEGIN INTERACTION DEFAULTS
  SELF CONTACT = ON
  GENERAL CONTACT = ON
  FRICTION MODEL = fric1
END
```

If the FRICTION MODEL command is not present, the default interface model is frictionless contact.

8.6.1.4. Interaction Behavior

```
INTERACTION BEHAVIOR = <string>SLIDING|NO_INTERACTION(SLIDING)
```

The INTERACTION BEHAVIOR command line specifies whether contact interactions are active. By default, all the interactions specified in the INTERACTION DEFAULTS command block through the GENERAL CONTACT, SELF CONTACT, or CONTACT SURFACES commands are active, that is SLIDING. However, setting INTERACTION BEHAVIOR to NO_INTERACTION will deactivate contact for all these surfaces. In either case, the value for INTERACTION BEHAVIOR set in the INTERACTION DEFAULTS block can be overwritten with the use of the same command line in individual INTERACTION command blocks.

8.6.1.5. Constraint Formulation

```
CONSTRAINT FORMULATION = NODE_FACE|FACE_FACE
```

The `CONSTRAINT FORMULATION` command specifies either node/face or face/face enforcement for contact constraints. This command only applies to Dash contact because implicit kinematic contact and ACME only support node/face contact. With Dash contact, face/face contact is used by default except when one of the surfaces is explicitly defined by a node set. In that case, Dash will automatically use node/face in which the "side B" surface will be the one defined by a set of nodes and the "side A" surface will be the contact surface defined by faces. See Section 8.3 for defining contact surfaces by nodes or faces.

See Section 8.6.4 for more information on the difference between node/face and face/face constraints.



8.6.1.6. Augmented Lagrange Penalty

```
AL PENALTY = <real>normalPenalty(1.0) [<real>tangentialPenalty]
```

The `AL PENALTY` command is used to scale the penalty stiffness that is internally set based on the nodal stiffness of the parts in contact. Both normal and tangential penalty scale factors default to 1.0. If only one value is specified, it will be used for both the normal and tangential penalty scale factors. When both `normalPenalty` and `tangentialPenalty` are entered, the normal penalty scale factor is set to `normalPenalty`, and the tangential penalty scale factor is set to `tangentialPenalty`.

By default, the AL penalty is scaled adaptively as the problem progresses (Section 4.5.1.5. For most problems, the default values of `AL PENALTY` and `LAGRANGE ADAPTIVE PENALTY` are robust (all penetration is removed and convergence is achieved). However, for more complicated implicit contact problems, it is sometimes required to modify the `AL PENALTY` to achieve convergence. However, note that using a small penalty value may result in significant penetration remaining in the model (even though convergence is achieved).

8.6.2. Specific Interactions

```
BEGIN INTERACTION [<string>name]
  # surface identification
  SURFACES = <string list>surfaces [EXCLUDE <string list>surfaces]
  SIDE A   = <string list>surfaces [EXCLUDE <string list>surfaces]
  SIDE B    = <string list>surfaces [EXCLUDE <string list>surfaces]
  #
  INTERACTION BEHAVIOR = <string>SLIDING|NO_INTERACTION(SLIDING)
  CONSTRAINT FORMULATION = <string>NODE_FACE|FACE_FACE
  FRICTION MODEL = <string>friction_model_name(FRICTIONLESS)

  #
```

```
  # tolerances
  NORMAL TOLERANCE = <real>norm_tol
  TANGENTIAL TOLERANCE = <real>tang_tol
  OVERLAP NORMAL TOLERANCE = <real>over_norm_tol
  OVERLAP TANGENTIAL TOLERANCE = <real>over_tang_tol

  # interaction specific output
  BEGIN INTERACTION SPECIFIC OUTPUT
    COMPUTE INTERACTION MASK
    COMPUTE INTERACTION FORCE CONTACT
    COMPUTE INTERACTION RELATIVE VELOCITY
    COMPUTE INTERACTION CONTACT STATUS
  END
Implicit  CAPTURE TOLERANCE = <real>cap_tol
   #
   # other implicit only commands
Implicit  AL PENALTY = <real>normalPenalty(1.0) [<real>tangentialPenalty]
Implicit  PUSHBACK FACTOR = <real>pushback_factor(1.0)
Implicit  TENSION RELEASE = <real>ten_release(0.0)
Implicit  TENSION RELEASE FUNCTION = <string>ten_release_func
  #
END [INTERACTION <string>name]
```

Values for specific contact interactions can be set using the `INTERACTION` command block. If an `INTERACTION DEFAULTS` command block is present within a `CONTACT DEFINITION` command block, the values provided by an `INTERACTION` command block override the defined defaults. If an `INTERACTION DEFAULTS` command block is not present, only those interactions defined by `INTERACTION` command blocks are searched for contact, and values without system defaults must be specified.

The `INTERACTION` command block begins with:

```
BEGIN INTERACTION [<string>name]
```

and ends with:

```
END [INTERACTION <string>name]
```

where `name` is a name for the interaction. Note that this name is currently used only for informational output purposes and is not required.

The valid commands within an `INTERACTION` command block are described in Section 8.6.2.1 through Section 8.6.2.2. Section 8.14.3.

8.6.2.1. Surface Identification

```
# preferred for explicit and implicit face/face contact
SURFACES = <string list>surfaces [EXCLUDE <string list>surfaces]
#
# preferred for implicit node/face enforcement
SIDE A   = <string list>surfaces [EXCLUDE <string list>surfaces]
SIDE B    = <string list>surfaces [EXCLUDE <string list>surfaces]
```

There are two methods to identify the surfaces described by a specific interaction. One method is to identify all surfaces in a single line with the `SURFACES` command line. The surface list must contain at least two surfaces. This command will define a symmetric two-way contact interaction between each surface on the command line and every other surface listed. To specify self contact, the surface must appear in the `SURFACES` command line twice.

The second method is to use the `SIDE A` and `SIDE B` command lines. In Sierra/SM, contact surfaces must be identified using the `SIDE A` and `SIDE B` command lines for kinematic enforcement. For augmented Lagrange enforcement, either method of identifying the surfaces may be used. The nodes of the side B surfaces are searched against the faces of the side A surfaces. Each of these command lines takes as input a list of names of contact surfaces defined in the `CONTACT DEFINITION` block. This interaction will be defined between each surface in the side A list and each surface in the side B list. Additionally, a surface may not be present in both lists.

The option `all_surfaces` is a special reserved word equivalent to directly specifying all contact surfaces known by the contact block in a list. Optionally, the `EXCLUDE` keyword can used along with a list of surface names to exclude from the list.

If the `SURFACES`, `SIDE A`, or `SIDE B` command lines appear multiple times within an `INTERACTION` block, their surface lists will be concatenated. The effect is equivalent to specifying all the surface names on a single line. The `SURFACES` command line cannot be used in the same `INTERACTION` block as `SIDE A` and `SIDE B` commands.

The following examples demonstrate ways to identify contact surfaces involved in an interaction:

This command defines a symmetric interaction between `s1` and `s2`:

```
SURFACES = s1 s2
```

This defines a symmetric self contact interaction between `s1` and itself:

```
SURFACES = s1 s1
```

This defines a set of symmetric interactions between `s1` and `s2`, `s1` and `s3`, `s2` and `s3`:

```
SURFACES = s1 s2 s3
```

This defines a full set of symmetric interactions between `s1` and itself, `s2` and itself, and between `s1` and `s2`:

```
SURFACES = s1 s1 s2 s2
```

This command defines a set of interactions between all pairs of defined surfaces in the model, with the exception of surfaces `s7` and `s8`, which have no interactions:

```
SURFACES = all_surfaces exclude s7 s8
```

This command defines a set of interactions between all pairs of defined surfaces in the model, as well interactions between each surface and itself, with the exception of surfaces `s7` and `s8`, which have no interactions:

```
SURFACES = all_surfaces all_surfaces exclude s7 s8
```

These commands define a one-way interaction between the nodes of `s1` and the faces of `m1`:

```
SIDE A = m1
SIDE B = s1
```

These commands define a set of one-way interactions between the nodes of `s1` and the faces of both `m1` and `m2`, and between the nodes of `s2` and the faces of both `m1`, the nodes of `s2` and `m2`.

```
SIDE A = m1 m2
SIDE B = s1 s2
```

These commands define that the nodes of surface `s1` are searched against all other contact faces in the contact definition block.

```
SIDE A = all_surfaces exclude s1
SIDE B = s1
```

8.6.2.2. Interaction Behavior

```
INTERACTION BEHAVIOR = <string>SLIDING|NO_INTERACTION(SLIDING)
```

This command sets the search behavior for a specific contact surface pair or for self-contact of a surface. See Section 8.6.1.4 for a discussion of this command line.

Since the default value is `SLIDING`, this command line is most often used to deactivate enforcement for the specific surfaces listed in this `INTERACTION` by setting the value to `NO_INTERACTION`.

8.6.2.3. Constraint Formulation

```
CONSTRAINT FORMULATION = NODE_FACE|FACE_FACE
```

The `CONSTRAINT FORMULATION` command specifies node/face or face/face enforcement for contact constraints for the specific surfaces listed in the `INTERACTION` block. See Section 8.6.1.5 for a discussion of this command line. See Section 8.6.4 for more information on the difference between node/face and face/face constraints.

8.6.2.4. Friction Model

```
FRICTION MODEL = <string>fric_model_name(FRICTIONLESS)
```

You can set the friction model for the interaction for a specific contact surface pair or for self-contact of a surface by using the above command line in an `INTERACTION` command block. See Section 8.6.1.3 for a discussion of this command line.

8.6.2.5. Interface Material

```
INTERFACE MATERIAL = <string>int_matl_name
  MODEL = <string>int_model_name
```

The Tvergaard Hutchinson and Thouless Parmigiani cohesive models (see Section 5.3) can be used as interface models within contact. Interactions between faces are created by a contact search performed in the initial configuration (similar to tied contact), so cohesive zones are only created between faces that are initially contacting. The relative motion of the faces on opposing sides of an interface is governed by the behavior of the cohesive zone model.

Note that these interface models can only be used with Dash contact. Additionally, due to the need to store state variables, these models can only be used with node-face contact interactions. A further restriction is the interactions must be defined between sidesets, to compute the correct area contribution. The side B sideset determines the nodal area of the cohesive interaction.

The interface model always takes one momentum balance iteration since we are not trying to close a gap in an Augmented Lagrangian fashion. Instead, whatever traction/force is computed for the gap, that force is what is applied to the nodes.



Warning: Interface contact models can only be used with node-face contact interactions, and they must be defined between two sidesets.

8.6.2.6. Tolerances

```
NORMAL TOLERANCE = <real>norm_tol
TANGENTIAL TOLERANCE = <real>tang_tol
OVERLAP NORMAL TOLERANCE = <real>over_norm_tol
OVERLAP TANGENTIAL TOLERANCE = <real>over_tang_tol
CAPTURE TOLERANCE = <real>cap_tol
```

Implicit

Tolerances for the interaction of specific contact surfaces or for self-contact of surfaces are set with the above tolerance-related command lines in an `INTERACTION` command block. See Section 8.9.6.1 on search tolerances and Section 8.9.4 on overlap tolerances for a complete discussion of tolerances for contact.

The contact functionality in Sierra/SM uses a box defined around each face to locate nodes that may potentially contact the face. This box is defined by a tolerance normal to the face and another tolerance tangential to the face (see Figure 8-21). The code adds the maximum motion over a time step to these tolerances when identifying interactions. In the above command lines, the parameter `norm_tol` is the normal tolerance (defined on the `NORMAL TOLERANCE` command line) for the search box and the parameter `tang_tol` is the tangential tolerance (defined on the `TANGENTIAL TOLERANCE` command line) for the search box.

In most cases, the normal and tangential tolerances computed internally are optimal values, and manual setting of tolerances should be avoided. If specific normal and tangential tolerances are set, the tolerances used should not exceed roughly half of the minimum characteristic face size. Tolerances beyond that length may cause instability in the contact solution.

The `CAPTURE TOLERANCE`, which should be no larger than the `NORMAL TOLERANCE`, is used to determine which side B nodes near a side A surface should be pulled to the side A surface and considered for contact. (This applies to side B nodes which have not penetrated the side A surface. Side B nodes that have penetrated the side A surface will be pushed to the surface regardless of the `CAPTURE TOLERANCE`.) If later checks determine that a side B node that was pulled to the surface (because it was nearer the surface than the `CAPTURE TOLERANCE` value) is in fact in tension and should be released, that side B node will not be considered as a potential contact node again during the load step as long as the node remains within the `NORMAL TOLERANCE`.





Warning: If `CAPTURE TOLERANCE` is set with anything other than KINEMATIC enforcement it will be ignored.

8.6.2.7. Interaction Specific Output

```
BEGIN INTERACTION SPECIFIC OUTPUT
  COMPUTE INTERACTION MASK
  COMPUTE INTERACTION FORCE CONTACT
  COMPUTE INTERACTION RELATIVE VELOCITY
  COMPUTE INTERACTION CONTACT STATUS
END
```

`INTERACTION SPECIFIC OUTPUT` allows a user to output special quantities of interest for a contact interaction pair. The interaction output is generated with respect to each `Side A` vs `Side B` node-face or face-face pair.

The `COMPUTE INTERACTION MASK` command creates a nodal variable named `<interaction_name>_interaction`. This variable shows which nodes are a part of the interaction defined. A value of 0 means the node is not in the interaction, a value of 1 means the node is a part of the `SIDE A` surface and a value of 2 means the node is a part of the `SIDE B` surface.

The `COMPUTE INTERACTION FORCE CONTACT` command creates a nodal variable named `<interaction_name>_force_contact`. This variable shows the contact force at the nodes that was generated for this interaction.

The COMPUTE INTERACTION RELATIVE VELOCITY command creates a number of output variables related to the relative velocity at the contact interaction. A global variable named <interaction_name>_normal_relative_velocity is the max relative velocity in the normal direction of impact over time. A nodal variable named <interaction_name>_relative_velocity is the relative velocity of each node-face or face-face pair, interpolated to the nodes of each face using the shape functions of the interaction. A nodal variable named <interaction_name>_max_normal_relative_velocity is the max relative velocity in the normal direction that a node sees during a timestep. For example, if a face on the SIDE A side contacts multiple faces on the SIDE B side, the max value of those face interactions is saved to the nodes.

The COMPUTE INTERACTION CONTACT STATUS command creates a nodal variable named <interaction_name>_contact status. A value of 0 means the node is not in contact with another face in the interaction. A value of 1 means the node is currently in contact with another face in the interaction.

A summary of all of the output variables is explained in Table 8-1 below.

Table 8-1. Output Variables Generated For Each Interaction Specific Output

Interaction Output	Variable Type	Variable Name
INTERACTION MASK	NODAL	<interaction_name>_interaction
INTERACTION FORCE CONTACT	NODAL	<interaction_name>_force_ contact
INTERACTION RELATIVE VELOCITY	NODAL	<interaction_name>_relative_ velocity
	NODAL	<interaction_name>_max_normal_ relative_velocity
	GLOBAL	<interaction_name>_normal_ relative_velocity



8.6.2.8. Augmented Lagrange Penalty

```
AL PENALTY = <real>normalPenalty(1.0) [<real>tangentialPenalty]
```

See Section 8.6.1.5 for a discussion of this command line.



8.6.2.9. Pushback Factor

```
PUSHBACK FACTOR = <real>pushback_factor
```

Implicit contact feature.

The command line PUSHBACK FACTOR can be used to set the fraction of the gap to be removed in a contact model problem. The default value is 1.0 which removes the entire gap in one contact

model problem. Setting the pushback factor to 0.25 will result in the gap being removed in 4 contact model problems.



8.6.2.10. Tension Release

```
TENSION RELEASE = <real>ten_release(0.0)
```

The command line `TENSION RELEASE` can be used to set a traction threshold. Below the threshold, side B nodes (that are in contact with side A faces) will not be released.

When this value is set and a side B node is in tension, it will only be allowed to pull away from the side A surface if the side B node's traction is greater than the `TENSION RELEASE` tolerance. The default value for `TENSION RELEASE` is 0.0.



8.6.2.11. Tension Release Function

```
TENSION RELEASE FUNCTION = <real>ten_release_func
```

The command line `TENSION RELEASE FUNCTION` provides a way for the tension release threshold to be set using a function from the input file. If the `TENSION RELEASE` line command is also present, the threshold used by the code will be the `TENSION RELEASE` value multiplied by the value obtained from the function.

8.6.3. Interaction Behavior for Particle Element Blocks

Element blocks containing particle elements (SPH, Mass Particle, or point masses) undergo special treatment, as described below.

Because the SPH element algorithms already define contact like interactions elements, it is typically not desirable to enforce self-contact within those element blocks, so self-contact is disabled by default for these blocks. For the same reason, contact between blocks of SPH elements is disabled by default.

Mass particles define both self-contact and inter-block interactions by default. Point masses behave similar to mass particles, but the 'skin' of a point mass block is just the nodes of the block. The 'skin' of a mass particle block is lofted spheres. Node/face contact for other particle types can be used by defining contact on the particle node sets, using the 'contact nodeset' command, or specifying the constraint formulation to `NODE_FACE`.

Elements converted to particles are placed in a new element block. The name of the new element block is the name of the parent block with `_particles` appended. For example, particles created from `block_1` are placed in a new element block named `block_1_particles`. The `SKIN ALL BLOCKS` command creates a new contact surface containing the particles with the same name as the particle block (`block_1_particles` in this example).

By default, the contact surface for the particles inherits interaction definitions from the parent block, subject to the exceptions listed above for interactions between particles blocks. However,

interactions specifying a friction model as TIED or DYNAMIC_TIED will be converted to FRICTIONLESS by default to prevent spurious constraints from being created. Spurious constraints are created when using the TIED or DYNAMIC_TIED friction models because the reference configurations between converted particles and solid elements are incompatible. If possible, the GLUED friction model is advised as a surrogate for the TIED friction model for interactions involving particles.

Any of the above defaults can be overridden by defining individual interactions. For example, if block_1 and block_2 are SPH particle blocks, they will not interact though contact by default. However, the following command block can be included to cause contact to be enforced between them.

```
BEGIN INTERACTION
  SURFACES = block_1 block_2
END
```

If a single contact surface is defined by combining multiple types of objects, the special particle-specific default behavior described in this section is disabled. For example, the following command block will turn on self-contact for all entities within the surface s1, including between the particles in block_2_particles.

```
BEGIN INTERACTION DEFAULTS
  SELF CONTACT = ON
END
CONTACT SURFACE s1 CONTAINS block_1 block_2_particles surface_3
```

8.6.4. Face/Face vs. Node/Face constraints

Interactions can be formulated as either face/face constraints or node/face constraints. Face/face is the default for all surface on surface interactions. Node/face is the default of node set on surface interactions or certain types of particle on surface interactions. The constraint formulation type can be set in the INTERACTION or INTERACTION DEFAULTS command blocks. The two types of constraints have trade-offs in terms of robustness and accuracy discussed in this section.

A node/face constraint is a constraint between a single node and a single face. In a node/face constraint a node is not allowed move through a face. The node is tied to a specific spot on the face, allowed to slide along the face, or allowed to move away from the face depending on the friction model used. An example of node for surface nodes (red) sliding off of a block (blue) is shown in figure 8-14. In the figure the constraints themselves are indicated with green arrows.

Node/face constraints have some limitations. In order to correctly resolve nodes sliding around corners of curved surfaces node/face constraints generally require use of some tangential tolerance (shown in dashed line). Note that with node face/face constraints the red nodes of the constraints may not penetrate the blue faces. The blue nodes however, may penetrate the red faces. Node/face constraints can potentially be setup symmetrically so that both sides of the

interactions contain both the nodes and the faces of constraints. However, this symmetric setup leads to chaining of constraints that may be difficult to resolve particularly in implicit analyses.

A face/face constraint is a constraint between two faces. The face/face constraint constrains the overlapping area of the two faces. In a face/face constraint no part of either face is allowed to move through any part of the other face. Faces remain tied together, can slide against one another, or move apart depending on the friction model used. An example of a surface (red) sliding off of a block (blue) with face/face constraints is shown in Figure 8-14. In the figure the constraints are represented with green regions of overlap between faces.

Face/face constraints do not require any tangential tolerances, though do have some limitations. Face/face constraints are most accurate when the two constrained faces are parallel. Face/face constraints can un-align faces, though sometimes with accuracy limitations particularly for frictional quantities. Face/face constraints are generally symmetric by nature, with no distinction between side A and side B of the interaction.

When tying together meshes with node/face constraints the forces are applied to discrete points potentially resulting in artificial stress concentrations. When tying with face/face constraints the contact forces are instead applied via tractions on the constrained faces. Thus face/face constraints generally results in a smoother tied solution with less severe artificial stress concentrations.

For node/face constraints the friction law is evaluated and the forces applied directly at the side B nodes of the constraints. This has the advantage that frictional output quantities can be accumulated easily over time at the nodal points. For face/face constraints the exact location of the constraint moves in time. This can smear the history of frictional quantities making them less distinct and less accurate.

In some cases, face/face constraints are more robust than node/face constraints and can more accurately solve certain classes of problems. Face/face constraints often work better than node/face for problems with high loading rates and stiffening materials. Node/face contact enforcement tends to aggravate hourglass modes when used on these types of problems. Additionally, problems with many blocks meeting at corners may often run more robustly with face/face constraints. In node/face constraints a node will only interact with one other face at a time making corner contacts inherently ambiguous. For face/face constraints the ambiguity is removed by moving the constraints to faces where the interaction is unambiguous even for complex geometry.

Node/Face constraints are generally a bit faster to evaluate than face/face constraints, though the performance difference is usually negligible in most production sized analyses.

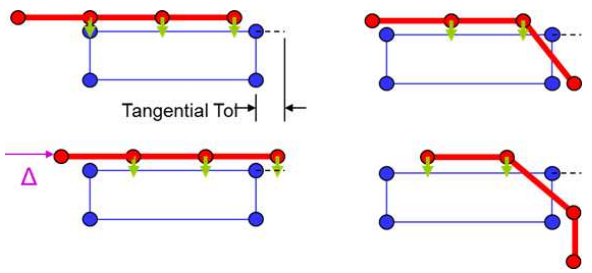


Figure 8-13. Node/Face Constraint example

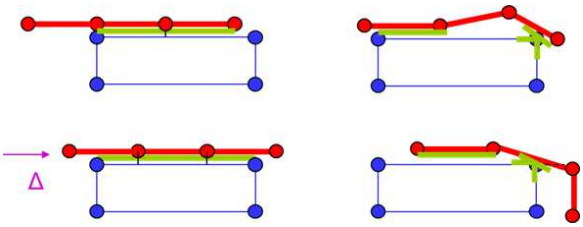


Figure 8-14. Face/Face Constraint example

8.7. Active Periods

```
ACTIVE PERIODS = <string list>period_names
```

A `CONTACT DEFINITION` is active for all periods by default. The contact definition can be activated for specific time periods with the `ACTIVE PERIODS` command line (See Section 2.6).

8.8. Contact with Analytic Objects

Contact with analytic objects is desired if a user would like to enforce contact on their meshed body but not modify the mesh file to create the object to come into contact with. This object can be created with the below syntax:

```
BEGIN ANALYTIC OBJECT <string>analytic_object_name
  TYPE = <string> PLANE
  NORMAL DIRECTION = <real>x_dir <real>y_dir <real>z_dir
  ORIGIN = <real>x <real>y <real>z
  SCALE FACTOR = <real> scale_factor
END [ANALYTIC OBJECT]
```

This syntax should be placed in the `REGION` command scope.

When using an analytic object, it is required to give your analytic object a name. This name is then used when specifying contact surfaces. For example, if you were to name your analytic object "analytic_obj1", you would specify the contact surface:

```
CONTACT SURFACE c1 CONTAINS analytic_obj1
```

The rest of the contact definition is defined using the usual commands. See section Section 8.3 for details.

The input command `TYPE` specifies the type of analytic object to create. Currently the only option is to create an analytic plane.

The input command `NORMAL DIRECTION` specifies the normal direction of the plane to be created.

The input command `ORIGIN` specifies the origin at which the plane is created.

By default, the plane that is created is 3X the size of the maximum length of the meshed body. If it is desired to make the plane a different size, the input command `SCALE FACTOR` can be used to scale the size of the plane.

8.8.1. Kinematic BCs with Analytic Objects

A kinematic boundary condition (such as fixed displacement, prescribed displacement, etc.) can be applied to the analytic object that is created. Within the boundary condition definition, the block name or surface name should be the name given to the analytic object.

If no kinematic BC is defined on the analytic object, a fixed displacement BC will be applied by default.

Other boundary conditions such as pressure, gravity, or force will not function on analytic planes.

8.9. Contact Algorithm Options

8.9.1. Contact Library

```
SEARCH = ACME|DASH(DASH)
```

Dash and ACME are two separate search and enforcement algorithms for defining explicit dynamics or implicit contact. Generally any existing ACME or Dash contact command block may be converted to a Dash or ACME command block by using this one command. However, Dash will ignore some ACME options and ACME will ignore some Dash options.

8.9.1.1. How Dash and ACME differ

Dash offers a facet-based enforcement algorithm. By default, Dash enforces face-face constraints. Both ACME and Dash offer a node-face based enforcement algorithm.

Dash is designed to work correctly for a wide variety of problems with minimal user intervention. It is generally recommended that Dash be used with a minimal set of inputs, for example `skin all blocks`, `self contact = on`, and/or `general contact = on`, and definition of the needed interaction friction models. Dash ignores tangential tolerances for face/face constraints. For more details (see discussion in Section 8.9.1.2).

For explicit contact, Dash uses an iterative augmented Lagrange stiffness enforcement scheme. Dash enforcement explicitly balances momentum by always applying equal and opposite forces. In contrast, ACME uses an iterative kinematic enforcement scheme. The kinematic enforcement scheme can more exactly remove gaps, but may not exactly satisfy force balance at a contact interface. With sufficient iterations, both the Dash and ACME enforcement schemes will converge to the same results.

For implicit contact, the Dash contact search can be used with either kinematic or augmented Lagrange enforcement. When Dash is used with the implicit kinematic enforcement algorithm, only node-face interactions are enforced, and the default behavior is changed from face-face to node-face.

Dash represents lofted structural elements differently than ACME. Dash can only represent shells via lofted geometry and has no option to treat shells as if they have no thickness for contact. Dash adds the edge faces to lofted shells, allowing edge and edge to face contact to function correctly on shell elements. Dash also handles beam on beam contact via lofted geometry. Beam objects are turned into representative hexagonal prisms that have roughly the same cross sectional area that the beam elements have. Beam contact is functional for edge-on or end-on cases.

These differences in the way structural elements are treated for contact make Dash significantly more capable than ACME in this area. ACME cannot treat any contact between shell edges and other entities, and cannot handle any contact with beam edges.

8.9.1.2. Current Dash Usage Guidelines

ACME does not perform well on changing topologies. Dash is more robust on problems with element death. ACME often must artificially kill a large number of additional elements other than those which have reached a user defined death criteria. ACME requires these additional element removals to work around certain surface topology restrictions. Dash has no topology restrictions so does not need to kill any extra elements to maintain a valid contact topology.

When performing contact involving shells, beams, and particles, Dash turns the elements into lofted volumes. ACME is able to loft shells and particles, but cannot loft beams.

Both ACME and Dash support basic friction models (tied, Coulomb friction, frictionless, etc.) However, some other friction models (spring weld, surface weld, area weld, adhesion, junction, threaded, pv_dependent, and user subroutine) are not yet available for use with Dash.

Prefer using block skinning with Dash, and avoid defining contact surfaces based on side sets. Dash uses ray tracing algorithms to determine if certain contact points are inside or outside of bodies. If the surfaces used for contact with Dash do not define fully enclosed bodies, it is difficult for the algorithm to determine wither a point is inside or outside a body.

Dash accepts the same surface definitions as does ACME. Dash also uses identical syntax for interaction definition.

8.9.2. Dash Specific Options

```
BEGIN DASH OPTIONS
  INTERACTION DEFINITION SCHEME = EXPLICIT|AUTOMATIC (AUTOMATIC)
  SEARCH LENGTH SCALING = <real>scale(0.15)
  TIED SEARCH LENGTH SCALING = <real>scale(0.25)
  SUBDIVISION LEVEL = <int>sublevel(0)
  QUAD PROJECTION TYPE =
    TRIANGULAR_FACETS|QUAD(TRIANGULAR_FACETS)
  HIDDEN SELF CONTACT POSSIBLE = FALSE|TRUE (TRUE)
  MAX CONTACT SUB STEPS = <int>maxSteps(100)
  NODE NORMAL SMOOTHING = AREA_WEIGHTED|ANGLE_WEIGHTED (ANGLE_WEIGHTED)
  SEPARATE DISCONNECTED MESH COMPONENTS = TRUE|FALSE (FALSE)
END
```

The `DASH OPTIONS` command block contains a few options unique to Dash. Generally Dash is designed to run accurately with as few user specified parameters as possible, however, some extra options are available. Note, the options in this block should generally be used only by experienced analysts for specific purposes.

The `INTERACTION DEFINITION SCHEME` command controls how contact interactions are defined. By default, most Dash interactions are one sided and the full volume of overlap is removed. By default, Dash automatically picks an optimal enforcement order for each interaction pair based on face size. However, the ability exists to explicitly use the interactions defined in the input deck by supplying the `EXPLICIT` option to this command. This may allow better control of

interaction setup in some circumstances or can be used to force Dash to use a symmetric interaction when it would by default use a one sided interaction.

The SEARCH LENGTH SCALING how Dash computes automatic contact tolerances. The automated search length used by contact at each face is the characteristic size of the face times this value. Note that the default value of this command should be used in almost all circumstances. One use of this command is to zero the automatic tolerances to force the contact algorithm to use the user provided search tolerances (normally contact uses the larger of the user provided or the default search tolerance). In some cases such as high velocity contact where objects move further than the default search length over a single step use of a larger factor can be helpful. Increasing the SEARCH LENGTH SCALING parameter may also be used to help resolve interpenetration issues in implicit contact that arise from large motion over a single time step. In such cases, values as large a 1.0 may be necessary. Note though that use of a scaling factor larger than approximately 0.50 can cause instability in the contact solution, especially for thin-walled structures, and should be used with caution in such cases.

Tied interactions used the length scaling given by TIED SEARCH LENGTH SCALING, by default tied interaction use a somewhat larger search length. If a tied interaction is initially further than the characteristic face size times the search length scaling tied contact will not be found. A larger or smaller value for this scaling number can be provided. However, values larger than 0.50 should be avoided as they can result in instability.

The SUBDIVISION LEVEL command provides a means to force a triangular face to be subdivided. Each quadrilateral face is originally split into four triangular faces. A subdivision level splits each tri-face edge into sublevel +1 segments. An increased subdivision level give a more accurate solution because the overlap volume is more accurately calculated. This increased accuracy, however, comes with a significantly increased computational cost.

The QUAD PROJECTION TYPE command provides a means to select the projection scheme of contact interactions to potentially warped quad surfaces. The default TRIANGULAR_FACETS option computes the projection point by dividing each quad into four flat triangles. The QUAD option computes the true position of the projection point on the warped quad. The TRIANGULAR_FACETS is faster to compute and is more robust in loading where deformations and potential warping is large. The QUAD option is somewhat more expensive to compute but can give a more consistent numerical result in an analysis where deformation is small.

The HIDDEN SELF CONTACT POSSIBLE command is used to control how Dash automatically sets up interactions. Self contact interactions use special more expensive contact logic. This more complex logic is needed to resolve constraint chaining issues that can happen in self contact. If two surfaces share (say contiguous blocks) and those two surfaces contact then by default the contact logic is applied as well as the same constraint chaining issues arise. The HIDDEN SELF CONTACT POSSIBLE command can be used to turn off this more expensive self contact logic. This can sometimes improve performance but may introduce instabilities in the contact solution.

In high velocity contact situations explicit contact enforcement may subdivide the time step to help accurately capture interactions. The MAX CONTACT SUB STEPS can be used to set the maximum number of sub steps taken over a step. In practice the only use of this command would

be to drop number of sub steps taken to one in a problem that is encountering some issue with the sub stepping computations.

8.9.2.1. Node Normal Smoothing

```
BEGIN DASH OPTIONS
  NODE NORMAL SMOOTHING = AREA_WEIGHTED|ANGLE_WEIGHTED(ANGLE_WEIGHTED)
END
```

The `NODE NORMAL SMOOTHING` affects how the nodal normals are computed for Dash node/face contact. The node normal is used to determine which face a given node will contact. In cases were multiple faces are potential contact candidates the node will contact the face with the face normal that most opposed the node normal.

There are two options for normal smoothing. The default `ANGLE_WEIGHTED` option computes the nodal normal as the sector angle weighted normals of attached faces. Consider the example shown in Figure 8-15 two faces come together at right angles. The sector angle (the angle at the face corner that attaches to the node) of both faces is 90 degrees. Thus the two faces contribute equally to the node normal which is at 45 degrees. The `AREA_WEIGHTED` option computes the nodal normal as the face angle weighted normals of attached faces. In Figure 8-15 the top face has more area than the side face. This pulls the node normal more in the direction of the top face.

The `ANGLE_WEIGHTED` is generally more accurate and has several beneficial property. With the angle weighted normal the node normal at edges and corners is not dependent on the specific mesh or element shapes. This is a very useful property for convergence studies. Additional for any convex geometry the angle weighted normal is guaranteed to be at least partially aligned with all attached face normals. This can make finding corner node/face interactions more robust. The primary advantage of the `AREA_WEIGHTED` option is speed. Use of the `AREA_WEIGHTED` option can improve runtime by up to 20% in some cases. Additionally the `AREA_WEIGHTED` option was the Sierra/SM default behavior before version 4.46 and is maintained for backward compatibility.

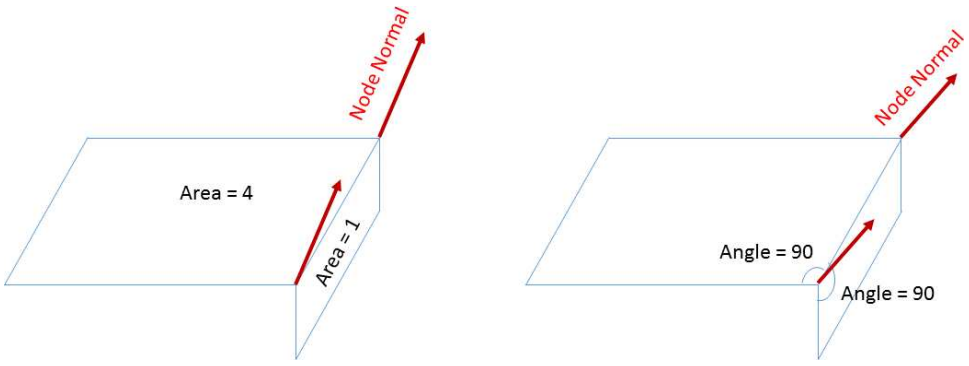


Figure 8-15. Node Normal Computation Options

8.9.2.2. Separate Disconnected Mesh Components

```
BEGIN DASH OPTIONS
  SEPARATE DISCONNECTED MESH COMPONENTS = TRUE|FALSE (FALSE)
END
```

The SEPARATE DISCONNECTED MESH COMPONENTS command treats separate mesh components as individual contact entities, even if they are in the same block, sideset, or nodeset. This option should be used when a model contains blocks with disconnected mesh components. Otherwise, a more expensive and less robust contact search is used to find interaction pairs, and the definition of the side A and side B surfaces is questionable.



Warning: Currently, this option can not be used with the Dash visualization options or when outputting certain contact variables. The simulation will run, but a warning will print to the log file.



8.9.3. Enforcement

```
ENFORCEMENT = <string>AL|KINEMATIC(AL)
CONTACT RESIDUAL NORM SCALE FACTOR = <real>scale
```

The ENFORCEMENT command line controls whether kinematic or Augmented Lagrange (AL) enforcement will be used in the current CONTACT DEFINITION block.

The AL option enforces contact using an Augmented Lagrange algorithm. This option can be used only in conjunction with the Dash search, which is enabled using the SEARCH = DASH command (See Section 8.9.1), and is the default option when using the Dash search. The FRICTION MODEL commands are necessary with this command and varying contact models can be enforced within the same CONTACT DEFINITION block.

If kinematic enforcement is enabled (by using the KINEMATIC option), the enforcement algorithm and the friction model are linked, and only one friction model can be used per CONTACT DEFINITION block. If a combination of tied, frictionless, and frictional contact is to be enforced, multiple CONTACT DEFINITION blocks must be included in the input file.

The CONTACT RESIDUAL NORM SCALE FACTOR Applies scale factor to contact residual norm of the contact definition in which this command is embedded. This allows a specific contact definition to contribute with different weight factor to the total contact residual norm. For example, one may want the contact residual from ARS to be the only contribution to the contact residual norm. This can be achieved by suppressing all other contributions using a scale factor of 0. The default value is unity.

8.9.4. Remove Initial Overlap

```
INITIAL OVERLAP REMOVAL = <string>ON|OFF(OFF)
```

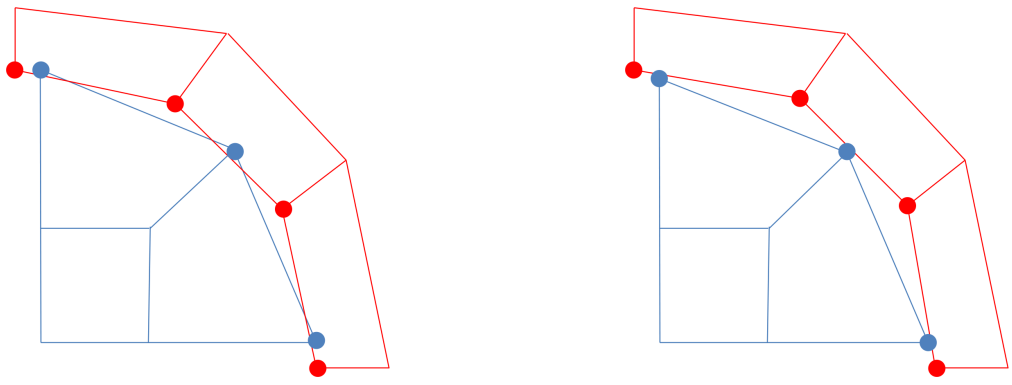
```
BEGIN REMOVE INITIAL OVERLAP
  OVERLAP NORMAL TOLERANCE = <real>over_norm_tol
  OVERLAP TANGENTIAL TOLERANCE = <real>over_tang_tol
  OVERLAP ITERATIONS = <integer>iterations(100)
  DEBUG ITERATION PLOT = <string>ON|OFF(OFF)
  REMOVE OVERLAP AT START OF TIME PERIODS = <string list>periods
END REMOVE INITIAL OVERLAP

OVERLAP REMOVAL ONLY = OFF|ON(OFF)
```

Meshes supplied for finite element analyses often have some level of initial mesh overlap, where finite element nodes reside inside the volume of nearby elements. In particular, this will happen when meshes meet at non-contiguous curved boundaries.

In explicit analyses the contact algorithm will attempt to push the overlapped constraints to zero gap. If a mesh has initial overlap contact will attempt to push out that overlap over the first explicit time step. As the explicit time step is often small, this may result in an extremely large force, velocity, and energy input into the initial mesh. Additionally, these initial forces are unphysical and may produce erroneous stress waves. Initial overlap removal is a mechanism for modifying the initial mesh to eliminate any initial overlap of contact surfaces. This is done before taking the first explicit dynamics time step to avoid the aforementioned issues. The overlap removal algorithm is by default turned off, but can be turned on by including the `INITIAL OVERLAP REMOVAL = ON` command line or use of the `BEGIN REMOVE INITIAL OVERLAP` block. The `REMOVE INITIAL OVERLAP` command block can also be used to modify the default overlap removal settings described later in this section.

For an example of overlap removal, see Figure 8-16.



(a) Initial geometry. **(b)** Geometry after overlap removal.

Figure 8-16. Effect of overlap removal on model geometry.

The process used to remove the initial overlap for three-dimensional solid elements involves changing the original coordinates of nodes on contact surfaces. Moving these coordinates yields a new mesh with the overlap removed; the overlap removal adds no initial stresses. Normal and

tangential tolerances specific to overlap removal may be specified by the user for all contact surfaces in the REMOVE INITIAL OVERLAP command block. It is also possible to specify overlap normal and tangential tolerances on each surface pairing separately by using the INTERACTION command block. Overlap removal tolerances specified in INTERACTION command blocks will overwrite the tolerances specified in the REMOVE INITIAL OVERLAP command block. See Section 8.6.2 for details. The REMOVE INITIAL OVERLAP command block only removes overlaps that are detected among the surfaces and elements defined for contact, not all surfaces in the mesh. Overlap removal tolerances will be set automatically if no specific tolerances are given.

Overlap tolerances are used to designate a maximum amount of overlap to search for and remove. If the amount of overlap in the mesh is larger than those tolerances, then the overlap will not be found and removed.

The mesh modification done by the overlap removal feature changes the meshed geometry; therefore, also changing the mass and time step of affected elements. The modified mesh is returned in the results file and should be checked to ensure that the modifications are acceptable.

8.9.4.1. Overlap Removal Diagnostics

The log file will contain a listing for maximum remaining overlap detected in the model after each overlap iteration. This maximum remaining overlap number should drive to zero over the overlap removal iterations. The code will output a warning if any initial overlap still exists after the overlap removal algorithms complete all iterations. Some initial overlap may still exist if there was excessive overlap in the initial configuration that could not be removed without inverting elements, or if there exists some pathology in the model geometry that the overlap removal algorithm cannot repair.

The log file contains two tables that relate to overlap removal. The first table lists the maximum amount any node of each block has been moved as a result of overlap removal. This quantity can be viewed on the mesh on a node by node basis by plotting the REMOVED_OVERLAP nodal variable on the results file. The second table lists the approximate maximum amount of overlap that exists in any element at the end of the overlap removal process. This variable can can be viewed on the mesh on an element by element basis by plotting the OVERLAP_VOLUME_RATIO element variable on the results file.

Note the remaining overlap value reported in the overlap removal iterations is computed in a slightly different way then the final overlap reported by the overlap volume ratio table. The overlap iteration number if based on surface intersections and nodal coordinates. The overlap volume ratio number is based on block volumes. The overlap volume ratio will detect overlaps such as whole volumes covering the same space that are not detected by the surface based algorithm. However, the overlap volume ratio also treats edge on edge overlaps and warped faces slightly differently than the surface based algorithm. As a result, small overlap volume ratios may be reported in a mesh even after the overlap removal algorithm completes successfully. If the overlap iterations drive the reported overlap to zero and the overlap volume ratio is some small number (generally one percent or lower) then the overlap removal algorithm was successful and

all overlap was removed. Also note that the overlap removal algorithm will not remove overlap between bodies not in contact. The overlap volume ratio report will however still report overlaps for such bodies. Overlap of non-contacting bodies usually will not cause issues for models but is still reported to the log file for completeness.

8.9.4.2. Overlap Removal Commands

The `OVERLAP ITERATIONS` sets the number of overlap iterations that will be performed. The default value for `OVERLAP ITERATIONS` is 100 and is usually sufficient to remove overlap. More iterations may be able to remove more overlap. However, if the overlap cannot be completely removed in the default 100 iterations, it is likely that there is some model geometric pathology that cannot be resolved and additional iterations are unlikely to help. Overlap removal iterations may also be used after particle insertion, geometric disconnection, and other operations that modify the mesh topology. A high allowed iteration count can make these operations more expensive. An alternative use of the `OVERLAP ITERATIONS` command is to visualize initial input mesh overlap to help detect potential meshing or merging errors. If `OVERLAP ITERATIONS` is set to 0 overlap diagnostics will be computed but no overlap will be removed. Thus with zero overlap iterations output variables such as element `overlap_volume_ratio` will show the overlap metrics found in the initial unmodified mesh.

The `DEBUG ITERATION PLOT` command line allows visualization of the overlap removal iterations. Turning the command line on will print each overlap removal iteration to the results output file. The nodal variable "removed_overlap" can be used to show the progress of each overlap removal iteration. If the `DEBUG ITERATION PLOT` command is used after overlap removal, the analysis will terminate. Consequently, this command should only be used for debugging overlap removal.

The `REMOVE OVERLAP AT START OF TIME PERIODS` command causes the overlap removal to be done at the start of each of the listed time periods. By default, overlap removal is done only at the start of the first time period. This command can be used to remove small overlaps between the pieces which are in contact before starting a new analysis time period.



Warning: Do not use `REMOVE OVERLAP AT START OF TIME PERIODS`, when having only one time period in your input file. In this case, this command is not needed and might cause unexpected behavior due to additional unnecessary overlap iterations.

If the `OVERLAP REMOVAL ONLY = ON` is used, the contact definition will only be used to remove initial overlap from the mesh. No contact enforcement will take place after overlap removal. This option can be used if it is desired for the overlap removal operations to use a different contact setup from the friction contact enforcement which is defined in a separate `BEGIN CONTACT DEFINITION` block.

8.9.4.3. Remove Initial Overlap Usage Guidelines

The following is a set of guidelines and restrictions which apply to overlap removal:

- Overlap removal works only on the nodes which are part of the contact surfaces. This means overlap removal generally cannot successfully remove large amounts of overlap. Attempts to remove overlap of greater than half an element length may lead to inverted or poorly shaped elements, making further analysis impossible. In such cases, the large overlap needs to be removed manually by remeshing.
- It is recommended to use the default normal and tangential tolerances for overlap removal whenever possible.
- When working with lofted elements (shells, particles and beams) overlap removal may move the nodes of these elements, but will not change the lofted thickness of the elements.
- Overlap removal only removes overlaps, it does not close initial gaps. Overlap removal attempts to bring overlapping surfaces exactly into contact but in some cases may result in small initial gaps between parts.
- Overlap removal will not move nodes that are kinematically constrained (such as nodes with fixed displacement boundary conditions).
- When removing a gap at an interface, overlap removal will generally move both sides of that interface by equal amounts.
- A tied contact friction model will work without overlap removal. However, it is still preferable to use overlap removal with tied contact as exactly matching interfaces lead to more accurate constraints.
- When restarting an analysis, overlap removal is not applied to the restarted mesh. It is assumed that the restarted mesh is already in the proper configuration for contact.

8.9.5. Lofted Surface Options

Elements such as shells, beams, and particles are represented for contact with lofted geometry, which defines a geometric volume around these lower dimensional elements. This volume allows for contact to be unambiguously evaluated, so that situations such as stacked shells or edge on edge contact of shells can be handled naturally. Block skinning creates lofted geometry by default out of all shell, beam, and particle element blocks. Certain features of the lofted geometry may be controlled with the `SURFACE OPTIONS` command block.

```
BEGIN SURFACE OPTIONS
  SURFACE = <string list>surface_names
  REMOVE SURFACE = <string list>removed_surface_names
  BEAM RADIUS = <real>beam_radius
  LOFTED BEAM SHAPE = HEXAGON|OCTAGON|SQUARE(HEXAGON)
  PARTICLE RADIUS = <real>part_radius
  LOFTED PARTICLE SHAPE = CUBE|ICOSAHEDRON|OCTAHEDRON|TETRAHEDRON|
    PARENT_ELEMENT|SPHERE (ICOSAHEDRON)
  CONTACT SHELL THICKNESS =
    ACTUAL_THICKNESS|LET_CONTACT_CHOOSE(ACTUAL_THICKNESS)
  ALLOWABLE SHELL THICKNESS TO ELEMENT SIZE RATIOS =
    <real>lower_bound(0.1) TO <real>upper_bound(1.0)
  SHELL CONTACT LOFTING FACTOR = <real>loftFactor
  SHELL CONTACT THICKNESS = <real>thickness
```

```
  CONTACT SHELL THICKNESS SCALE FACTOR = <real>scale_factor
  COINCIDENT SHELL TREATMENT = <string>
    DISALLOW|IGNORE|BEST(BEST)
  COINCIDENT SHELL HEX TREATMENT = <string>
    DISALLOW|IGNORE|TAPERED(TAPERED)
END [SURFACE OPTIONS]
```

The SURFACE and REMOVE SURFACE command lines specify which contact surfaces this command block refers to. These command lines are optional; by default the commands in the surface options block apply to all contact surfaces. If the SURFACE command line is used by itself, only those surfaces listed will be affected. If the REMOVE SURFACE command line is used by itself, all contact surfaces except those listed will be affected. Note, the surface names used in this command block are the names of the contact surfaces, not the names of the sidesets or blocks used to form those contact surfaces. Also, lofting is not performed on contact surfaces defined from sidesets; a warning will be thrown and the sidesets will not be lofted. To loft shells, the contact surfaces must be defined by blocks.



Warning: To loft shells for contact, the contact surfaces must be defined by blocks.

8.9.5.1. Lofted Beams

By default, contact represents the cross section of lofted beam elements by a six sided prism. The radius of the vertexes of the hexagons is the square root of the beam cross sectional area divided by pi. That is, the beam cross sectional area is converted into a circle, and then the hexagon end cap of the beam roughly represents this circle. Note that for a circular beam, the contact representation of that beam will nearly match the actual beam geometry. However, for an I-Beam cross section, the shape of the contact representation may be significantly smaller than the actual cross section length scales. The representation of various beams in contact is shown in Figure 8-17.

If a T AXIS command is given in a beam section the orientation of the lofted contact beam prism will be informed by that t-axis. Specifically one of the flat sides of the prism will be placed perpendicular to the beam t-axis. If no t-axis is given in the beam section the orientation of the beam prism will be arbitrary.

Lofted beam elements are used to represent beams, springs, dampers, and other one dimensional elements.

The BEAM RADIUS command can be used to explicitly set the radius of the lofted beam elements, overriding the default use of the beam's cross sectional area.

The LOFTED BEAM SHAPE command changes how the beam is represented. Objects with fewer facets tend to be less accurate, but may run faster.

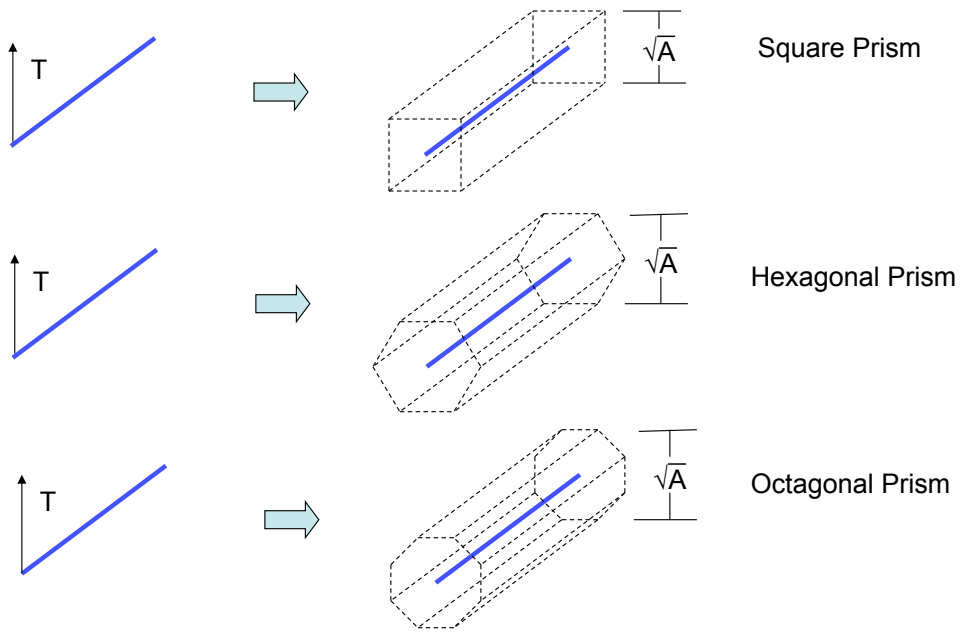


Figure 8-17. Lofted beam geometry

8.9.5.2. Lofted Particles

The representation of various particles in contact are shown in Figure 8-18. Lofted particle elements are used to represent mass particles, SPH particles, and other zero dimensional point elements.

The `PARTICLE RADIUS` command can be used to explicitly set the radius of the lofted particle elements. By default, lofted particles are represented as twenty-sided icosahedra having radii equal to half of the value of the element `sph_radius` field.

The `LOFTED PARTICLE SHAPE` can change how finely the particle sphere is represented. Objects with fewer facets tend to be less accurate, but may result in improved run times. Note, the `PARENT_ELEMENT` option is experimental and only available for hexahedral elements converted to particles at initialization.

The `LOFTED PARTICLE SHAPE = SPHERE` command lofts an analytic sphere about the center or each particle instead of a twenty-sided icosahedron. The surface of the sphere defined by the particle radius is used in the closest point projection calculation. The closest point projection routine is then computed as normal. This option is expected to perform better for both accuracy and performance due to the spherical representation.



Warning: The `LOFTED PARTICLE SHAPE = SPHERE` command is only implemented for node-face contact.



Warning: When using this option, the particle block must be defined as the side B in the interaction definition.

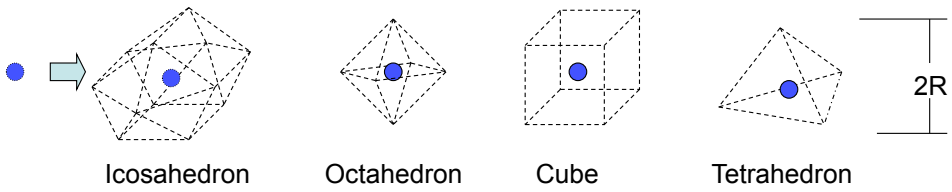


Figure 8-18. Lofted particle geometry



Warning: Self contact is not yet implemented for this option. Self contact will function, but contact will create an interaction between the side B analytic sphere and the side A icosahedron particle. Performance benefits are still expected with this option, but not as drastic as self analytic sphere contact.

8.9.5.3. Lofted Shells

Lofted shells are represented by prisms around each shell element. An example representation of shells in contact is shown in Figure 8-19. Lofted shells are used to represent shells, membranes, and other two dimensional elements. The lofted shell geometry is also based on the element section `LOFTING FACTOR` as shown in Figure 6-4.

The `CONTACT SHELL THICKNESS` command line controls whether lofted shell contact uses the actual thickness of shell elements or a pseudo thickness that is computed by the contact algorithm. The default option to this command, `ACTUAL_THICKNESS`, causes the actual element thickness to be used. The `LET_CONTACT_CHOOSE` option to the `CONTACT SHELL THICKNESS` command tells the contact algorithm to create lofted geometries of the shells that are more appropriate for contact.

Shell elements that are very thick or very thin in relation to their in-plane dimension can be problematic in contact. A shell is considered very thin when the shell thickness is less than ten percent of the shell element's in plane size. Thin shells may require a larger number of sub-steps in the contact algorithm to ensure that objects do not pass completely through the shell in a single step, causing contact to be missed. For thick shells, particularly near corners, the exterior geometry of these prisms may become jagged and not representative of the actual shell geometry. A shell may be considered very thick when the shell thickness is greater than the shell element's in plane size.

Shell elements that are thick in relation to their in-plane dimension can produce ill-defined geometries, as seen in Figure 8-20. The `LET_CONTACT_CHOOSE` option can help alleviate issues related to thick or thin shells. By default, when the `LET_CONTACT_CHOOSE` option is used, lofted geometry thicknesses are ensured to be within 0.1–1.0 times the in-plane dimension of the shell element. If the contact-appropriate shell thickness is computed to be outside of these bounds, the lofted shell thickness will be set to the upper or lower bound as appropriate.

The permissible bounds for the ratios of contact-appropriate thickness to actual thickness can be set using the `lower_bound` and `upper_bound` parameters in the

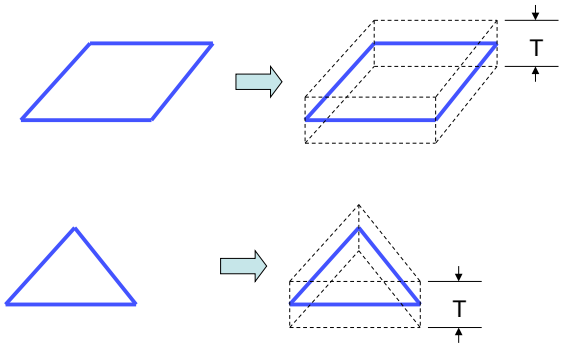


Figure 8-19. Lofted shell geometry

ALLOWABLE SHELL THICKNESS TO ELEMENT SIZE RATIOS command. The ALLOWABLE SHELL THICKNESS command is functional only for Dash contact.

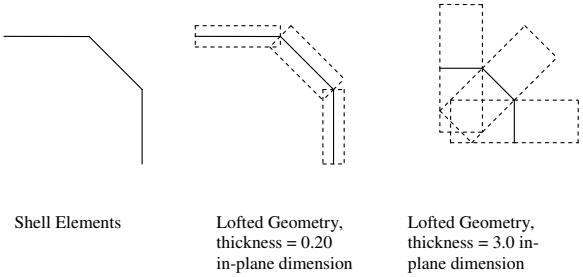


Figure 8-20. Example lofted geometries produced by shell lofting

A contact specific lofting factor for shells in can be set with the SHELL CONTACT LOFTING FACTOR command. For a description of shell lofting conventions see Section 6.2.5. By default contact uses the same lofting factor as does the shell element mechanics to loft the volume of the shell away from the shell element. The SHELL CONTACT LOFTING FACTOR will change the lofting factor for the shell in the contact algorithm, but not affect the lofting factor used for the element mechanics. This contact shell lofting factor can be used to properly fit together shell meshes without the algorithmic time step and stability limitations that can come from the lofting of the element mechanics.

A contact specific thickness for shells can be set with the SHELL CONTACT THICKNESS command. By default contact uses the same thickness as does the shell element mechanics. Note, this command overrides the ALLOWABLE SHELL THICKNESS TO ELEMENT SIZE RATIO option and explicitly sets an absolute contact thickness.

Additionally, the shell contact thickness can be set via the CONTACT SHELL THICKNESS SCALE FACTOR command. If this command is specified, the shell contact thickness well be set to the thickness used for the shell element mechanics multiplied by this scale factor. If both CONTACT SHELL THICKNESS SCALE FACTOR and SHELL CONTACT THICKNESS are specified for a surface, the SHELL CONTACT THICKNESS is used. Note, this command overrides the ALLOWABLE SHELL THICKNESS TO ELEMENT SIZE RATIO option and explicitly sets an absolute contact thickness.

8.9.5.4. Coincident Face Options

The COINCIDENT SHELL TREATMENT and COINCIDENT HEX SHELL TREATMENT command line identifies how elements that share the same nodes should be treated. This includes stacked shells and shell-clad faces of the hexahedral element. Note as Dash has more robust shell contact logic by default Dash contact allows coincident shells while acme disallows coincident shells. Generally Dash should be used with the default value of these two commands, other options are provided primarily for compatibility with ACME.

If the DISALLOW option is selected then any time that coincident contact faces are found the code will abort with an error message indicating which elements were found to be coincident. The DISALLOW option should be used if you do not want any coincident shells to be considered in the analysis. The option operates essentially as a check on the mesh.

If the IGNORE option is selected, any contact faces attached to coincident shells or coincident shell/hexahedral boundaries are ignored for contact.

The default BEST and TAPERED options indicate that dash should use its standard shell lofting logic on coincident shells. For stacked shells this effectively means the shell stack has a contact thickness governed by the thickest shell in the stack. For coincident shell/hexahedral boundaries contact will be done on the shell faces which may be lofted somewhat away from the hexahedral face.

8.9.6. Search Options

```
BEGIN SEARCH OPTIONS [<string>name]
  #
  # Search tolerances
  NORMAL TOLERANCE = <real>norm_tol
  TANGENTIAL TOLERANCE = <real>tang_tol
  CAPTURE TOLERANCE = <real>cap_tol
  TENSION RELEASE = <real>ten_release
  SLIP PENALTY = <real>slip_pen
END [SEARCH OPTIONS <string>name]
```

Implicit

Implicit

Implicit

Contact involves a search phase and an enforcement phase. The contact search algorithm used to detect interactions between contact surfaces is often the most computationally expensive part of an analysis. The user can exert some control over how the search phase is carried out via the SEARCH OPTIONS command block. By selecting different options in this command block, the user can make trade-offs between the accuracy of the search and computing time.

The SEARCH OPTIONS command block begins with the input line:

```
BEGIN SEARCH OPTIONS [<string>name]
```

and ends with:

```
END [SEARCH OPTIONS <string>name]
```

The `name` for the command block is optional.

Without a `SEARCH OPTIONS` command block, the default search with associated default search parameters is used for all contact pairs. If you want to override the default search method for all contact pairs, you should add a `SEARCH OPTIONS` command block. By adding a `SEARCH OPTIONS` command block, you establish a new set of global defaults for the search for all contact pairs.

The valid command lines within a `SEARCH OPTIONS` command block are described in Section 8.9.6 and Section 8.9.6.1. The values specified by these commands are applied by default to all interaction contact surfaces, unless overridden by a specific interaction definition.

8.9.6.1. Search Tolerances

```
NORMAL TOLERANCE = <real>norm_tol
TANGENTIAL TOLERANCE = <real>tang_tol
```

As indicated previously, the contact functionality in Sierra/SM uses a box defined around each face to locate nodes that may potentially contact the face. This box is defined by a tolerance normal to the face and another tolerance tangential to the face (see Figure 8-21). The code adds to these tolerances the maximum motion over a time step when identifying interactions. In the above command lines, the parameter `norm_tol` is the normal tolerance (defined on the `NORMAL TOLERANCE` command line) for the search box and the parameter `tang_tol` is the tangential tolerance (defined on the `TANGENTIAL TOLERANCE` command line) for the search box.

By default, Sierra/SM will automatically calculate normal and tangential tolerances roughly based on the characteristic length of a face multiplied by the value input by the search length scaling. The default search length scaling is 0.15. The automatic tolerances add the maximum motion over a time step just like the user defined tolerances. If you leave automatic search on and also specify normal and/or tangential tolerances with the `NORMAL TOLERANCE` and `TANGENTIAL TOLERANCE` command lines, the larger of the two (automatic or user specified) tolerances will be used. For example, suppose you specify a normal tolerance of 1.0×10^{-3} and the automatic tolerance computes a normal tolerance of 1.05×10^{-3}. Then Sierra/SM will use a normal tolerance of 1.05×10^{-3}.

Both of these tolerances are absolute distances in the same units as the analysis. The proper tolerances are problem dependent. If a normal or tangential tolerance is specified in the `SEARCH OPTIONS` command block, they apply to all interactions. These default search tolerances can be overwritten for a specific interaction by specifying a value for the normal tolerance and/or tangential tolerance for that interaction inside the `INTERACTION` command block (see Section 8.6.2).

Normal and tangential contact tolerances are generally robustly automatically set and manual setting of tolerances should be avoided. If specific normal and tangential tolerances are set the

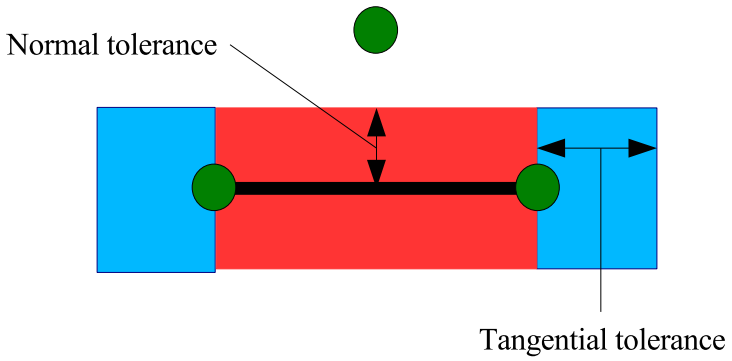


Figure 8-21. Illustration of normal and tangential tolerances

tolerances used should not exceed roughly half of the minimum characteristic face size. Tolerances beyond that length may cause instability in the contact solution.



8.9.7. Enforcement Options

```
BEGIN ENFORCEMENT OPTIONS [<string>name]
  MOMENTUM BALANCE ITERATIONS = <integer>num_iterations(5)
  NUM GEOMETRY UPDATE ITERATIONS =<integer>num_updates(1)
  ACCUMULATED GAP REMOVAL OPTION =
    REMOVE_ALL|REMOVE_DEFAULT| REMOVE_NONE(REMOVE_DEFAULT)
END [ENFORCEMENT OPTIONS <string>name]
```

Contact involves a search phase to create constraints and an enforcement phase to enforce those constraints. The user can exert some control over how the enforcement phase is carried out via the `ENFORCEMENT OPTIONS` command block. By selecting different options in this command block, the user can make trade-offs between solution accuracy and computing time. The `ENFORCEMENT OPTIONS` command block begins with the input line:

```
BEGIN ENFORCEMENT OPTIONS [<string>name]
```

and is terminated with the input line:

```
END [ENFORCEMENT OPTIONS <string>name]
```

The name for the command block, `name`, is optional

Only a single `ENFORCEMENT OPTIONS` command block is permitted within a `CONTACT DEFINITION` command block. Without an `ENFORCEMENT OPTIONS` command block, the default enforcement algorithm with associated default enforcement options is used for all contact pairs. If you want to override the defaults for enforcement for all contact pairs, you should add an `ENFORCEMENT OPTIONS` command block. By adding this command block, you establish a new set of global defaults for enforcement for all contact pairs.

In ACME contact enforcement, nodes are explicitly moved to a zero gap state at the beginning of the enforcement algorithm. Then, after the nodes are moved, ACME calculates the forces necessary to provide a momentum preserving net force. In ACME contact, the `MOMENTUM BALANCE ITERATIONS` command line specifies how many iterations to take inside of contact enforcement to calculate this net force.

In DASH contact enforcement, equal and opposite forces are always applied to the nodes, preserving the momentum by construction. DASH then iterates on the gap between the side A and side B surfaces, minimizing the gap as much as possible. In DASH contact, the `MOMENTUM BALANCE ITERATIONS` command line specifies how many iterations to take to incrementally modify the forces to drive the gap to zero. In implicit contact, AL updates are a similar concept to momentum balance iterations. Note that in implicit, contact enforcement will be called as many times as needed to get below the residual values specified in the control contact block within the solver. Once the implicit contact solution is converged, the model problem will be updated with the nodal forces and positions predicted in contact.

For explicit contact, the number of passes (iterations) is set by the value `num_iterations` in the `MOMENTUM BALANCE ITERATIONS` command line. The default value for the number of iterations is 5. This value is generally acceptable for satisfying the material interpenetration constraint. To get accurate results in a global sense in analyses that use friction, a value of 10 is more appropriate. To get accurate contact response at individual points in analyses with friction, a value of 20 or greater may be needed. Note that as the number of iterations increases, the expense of enforcement increases. Thus a user can balance execution speed and accuracy with this command line, though care must be taken to ensure that the appropriate level of accuracy is attained. This command line affects only the enforcement phase of the contact. A single search phase is used for contact detection, but the enforcement phase uses an iterative process. Note that when using momentum balance iterations, only the contact forces and nodal coordinates are updated on the nodes in contact. The internal and external forces are not re-computed, and the other nodes not in contact are not updated until the next explicit timestep.

The `NUM GEOMETRY UPDATE ITERATIONS` defines the number of times contact will be enforced, updating the position of the current coordinates each iteration. By default, explicit contact uses one geometry update iteration. In an explicit step, first nodes move as if they have zero contact force; contact constraints are defined in this deformed configuration. Next, contact force is computed to enforce the constraints, remove interpenetration, satisfy friction laws, and compute a new deformed end of step state from the combination of contact, internal, and external forces. With additional geometry update iterations, the constraint equations are re-found in this updated configuration and then enforced again. Geometry update iterations can improve accuracy and stability if there is substantial motion over a single explicit time step. This can occur in hypervelocity impact problems or if contact pressures are extremely high. However, geometry update iterations are very expensive. Taking two geometry update iterations will double the cost of contact.

The `ACCUMULATED GAP REMOVAL OPTION` controls how the explicit contact algorithm deals with either initial or accumulated gap in the frictionless and coulomb friction sliding interactions. Consider a mesh with initial overlap between contacting bodies. If the `OVERLAP REMOVAL` is not done then this initial overlap will be removed via standard contact forces. The

ACCUMULATED GAP REMOVAL OPTION command will affect this case as follows:

- The REMOVE_ALL option will remove all this existing gap over a single time step. The acceleration and force required to do this removal over the single explicit step may be very high, and may add a lot of energy to the system. The remove all option will tend to prevent any overlap from ever occurring in the mesh, but can lead to spuriously high energy and contact stability problems in some cases. Note, the REMOVE_ALL option was the default interaction behavior before the 4.38 code release.
- The REMOVE_NONE will not remove any of the old gap. The contact constraint zero gap point will be defined with respect to the initial configuration. Thus the initial gap will not be removed and the initial overlap will not add energy. However, with the REMOVE_NONE option it is possible for small errors in enforcement to accumulate gap in the interactions over time, this can lead to contact interactions being lost and bodies passing through one another.
- The REMOVE_DEFAULT uses a heuristic weighting to remove some of the preexisting gap. The portion of preexisting gap removed will depend on the current gap and gap rate. Generally enough gap will be removed to ensure that accumulation of error will not cause loss of the interaction while the energy gain from preexisting gap removal is minimized. Preexisting gaps will be removed over several time steps. The default option provides a good balance of stability and contact accuracy and should be used for most circumstances.

Note, in addition to initial overlap preexisting gap can occur in many other circumstances including element death, particle creation, non-exact contact enforcement, and a variety of geometric ambiguities that can occur at corners. Thus selection of the gap removal option will have some, though generally minimal, effect on most runs.

8.10. Contact-Specific Outputs

Contact variables can be output to provide information about enforcement of contact interactions. Currently, information on only one interaction at each node is provided. If a node has more than one interaction, the last one in its internal interaction list is reported.

Nodal contact variables that can be output are listed in Table 9-11. Where applicable, names of the equivalent variables in JAS3D are given in parentheses at the end of the description. The variables can be output in history files or results files; see Chapter 9 for more information on outputting nodal variables. Note that currently the variables cannot be calculated at output time so the first time they are output a request is made to calculate them. This means that the first output step where they are to appear the data will be all zero. A work around for this is to have at least one output step in which these variables appear before their values are needed.

It is no longer required to specify `COMPUTE CONTACT VARIABLES = ON` in the input file. Now, Sierra/SMwill determine if a contact variables is being requested for output and compute that variable by default at every output time (unless it is an accumulated contact variable). If you would like to compute contact variables at every step (to be used in user output for example), specify `COMPUTE CONTACT VARIABLES = EVERY_STEP`

The contact variables that are accumulated every step and cannot only be computed at the output times are the following: `CONTACT_ACCUMULATED_SLIP`, `CONTACT_ACCUMULATED_SLIP_VECTOR`, `CONTACT_FRICTIONAL_ENERGY`, and `CONTACT_FRICTIONAL_ENERGY_DENSITY`.

8.10.1. Contact Debugging

```
BEGIN DEBUG
  VISUALIZE CONTACT FACETS = OFF|ON(OFF)
  OUTPUT CONTACT SURFACES AND CONSTRAINTS = OFF|ON(OFF)
  OUTPUT LEVEL = <integer>level(0)
  WRITE SALINAS MPCS TO <string>file AT TIME <real>time
END
```

Note: This option is available only for Dash contact.

The debug contact block enables additional information to be output by the code to help debug what may be going wrong with a contact problem.

The `OUTPUT LEVEL` command controls the amount of information written to the log file. At the default level of zero, only minimal information is included. At level 1 or greater, more information is printed to the log file such as the composition of the different contact surfaces and which friction model is being used for which surface pairing.

The `VISUALIZE CONTACT FACETS` command in the `DEBUG` command block can be used to plot the actual contact facets and lofted contact geometries that contact is being evaluated on. When this command is set to `ON`, extra element blocks and node blocks will be added to the output mesh to represent the contact geometries. Note that use of this option will significantly increase the size

of the results file and moderately increase the run time. The naming convention used for these surfaces is to prepend the prefix `CSURF_` to the original contact surface name. For example, if a contact surface is produced by skinning `block_1`, then a side set will be created with the name `CSURF_block_1`. This is a Dash only option.

The `OUTPUT CONTACT SURFACES AND CONSTRAINTS` command block can be used to plot interactions into the mesh output file as sets of surfaces and nodes. Viewing the interactions can help debug contact problems. This feature only works with node_face contact interactions and only if the constraint can fit in the connectivity of a ten-noded tetrahedral element.

The `WRITE SALINAS MPCS` command can be used to write the contact constraints to a text file compatible with the Salinas input deck format. The constraints will be written to the specified file name `file` *once* at the first analysis time step after the specified target time `time`. All contact constraints (tied, glued, frictionless, frictional, etc.) are output in Salinas format as tied MPCs. The `WRITE SALINAS MPCS` command can be specified multiple times to output constraints at various times to various separate files.

8.11. Log File Contact Information

When contact is included in the analysis, pertinent information is written to the log file. Information on the name and size of each contact surface will be included in a table as shown:

```
-------- DASH SURFACES --------
 Contact Entity   Num Faces   Num Lofted   Num Points   MinCharLen   MaxCharLen
---------------- ----------- ------------ ------------ ------------ ------------
       pin_surf           0            0          287          0.0        0.012
        block_1         994            0            0         0.01         0.01
        block_2           0           24            0         0.02         0.03
   surface_1000           4            0            0         0.50         0.50
```

The above table will have one entry for each defined contact surface. The name of the contact surface is in the “Contact Entity” column. “Num Faces” is the number of non-lofted faces (such as triangles or quadrilaterals) contained in the surface. “Num Lofted” is the number of lofted objects (such as shells, beams, or particle elements) contained in the surface. “Num Points” is the number of nodes in the contact surface that are not part of any of the faces or lofted elements. `MinCharLen` and `MaxCharLen` are the minimum and maximum characteristic length of any face or lofted object in the contact surface.

The log file will also include a representation of the interaction matrix, for example, of the form:

```
------------ DASH INTERACTION WEIGHT MATRIX ------------
0000
1H00
00HH
00H0
```

The topmost row and leftmost column of the interaction matrix correspond to the first entry in the “Contact Entity” list. The bottom most row and the rightmost column of the interaction matrix corresponds to the last entry in the “Contact Entity” list. The entry at row I, column J of the matrix describes how contact surfaces I and J interact.

A value of “1” indicates that contact surface I has interactions with contact surface J where surface I is the side B surface and surface J is the side A surface. For node-face contact this means that the nodes of surface I contact the faces of surface J. For face-face contact this means that the faces of surface I interact with the faces of surface J and that the gap integral is performed on faces of surface I.

A value of “H” (short for half) indicates contact surface I interacts with contact surface J in symmetric contact. For node-face contact this means that the nodes of surface I contact the faces of surface J and also the nodes of surface J contact the faces of surface I. For face-face contact this means the faces of surface I contact the faces of surface J and the gap integral is performed simultaneously on both sides of the interface.

A value of “0” in both the IJ and JI position of the matrix indicates that surfaces I and J will not interact via contact.

Note that a surface can only interact with itself (the diagonal entries of the matrix) through self contact. If entry IJ of the matrix is “1” entry JI must be “0.” If entry IJ of the matrix is “H” than

entry JI must also be “H.”

8.12. Examples

This section has several example problems. The geometric configuration for each problem is presented together with a brief description and the appropriate command lines.

8.12.0.1. Example 1

The first example problem has two blocks that come into contact due to initial velocity conditions. Block 1 has an initial velocity equal to $v1$, and block 2 has an initial velocity equal to $v2$. The geometric configuration for this problem is shown in Figure 8-22.

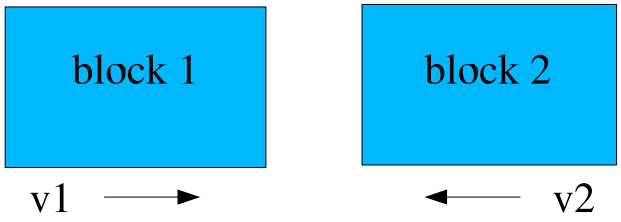


Figure 8-22. Problem with two blocks coming into contact

The simplest input for this problem will be named `EXAMPLE1` and is shown as follows:

```
BEGIN CONTACT DEFINITION EXAMPLE1
  #
  # define contact surfaces
  SKIN ALL BLOCKS = ON

  # set interactions
  BEGIN INTERACTION DEFAULTS
    GENERAL CONTACT = ON
  END

END
```

In this example, the `SKIN ALL BLOCKS` command line with parameter set to `ON` will create a surface named `block_1` from the exterior skin of `block_1` and a surface named `block_2` from the exterior skin of `block_2`. The `GENERAL CONTACT = ON` command defines that `block_1` can contact `block_2`. As no friction model was specified, the default frictionless model is used for this interaction.

Now consider the same problem (two blocks coming into contact) in which it is desired that the two blocks interact with a coulomb friction model. The input for this variation of the two block problem will be named `EXAMPLE1A` and is shown as follows:

```
BEGIN CONTACT DEFINITION EXAMPLE1A
```

```
  #
  # define contact surfaces
  SKIN ALL BLOCKS = ON

  # friction model definition
  BEGIN CONSTANT FRICTION MODEL rough
    FRICTION COEFFICIENT = 0.5
  END

  # set interactions
  BEGIN INTERACTION EX1A
    SURFACES = block_1 block_2
    FRICTION MODEL = rough
  END
END
```

As was the case for the `EXAMPLE1` command block, the `SKIN ALL BLOCKS` command line with its parameter set to `ON` will create a surface named `block_1` from the skin of `block_1` and a surface named `block_2` from the skinning of `block_2`.

In order to have frictional contact between the two blocks, a constant Coulomb friction model is specified with a `CONSTANT FRICTION MODEL` command block. We name this model `ROUGH` and specify the coefficient of friction as `0.5`.

In the `INTERACTION` command block, the friction model `ROUGH` is selected on the `FRICTION MODEL` command line. This model is then applied to the interaction of surfaces `block_1` and `block_2`.

8.12.0.2. Example 2

A second more complex example problem has three blocks that come into contact due to initial velocity conditions. Block 1 is a solid element brick block with initial velocity equal to $v1$. Block 3 is a solid element brick block with initial velocity $v3$. Block 2 is a shell element block that has zero initial velocity. There is a side set (Surface 3) on the top faces of block 3. The geometric configuration for this problem is shown in Figure 8-23. The contact input for this three-block problem is called `EXAMPLE2` and is shown as follows:

```
BEGIN CONTACT DEFINITION EXAMPLE2
  #
  # define contact surfaces
  CONTACT SURFACE b1 CONTAINS block_1
  CONTACT SURFACE b2 CONTAINS block_2
  CONTACT SURFACE s3 CONTAINS surface_3

  BEGIN CONSTANT FRICTION MODEL rough
    FRICTION COEFFICIENT = 0.5
  END
```

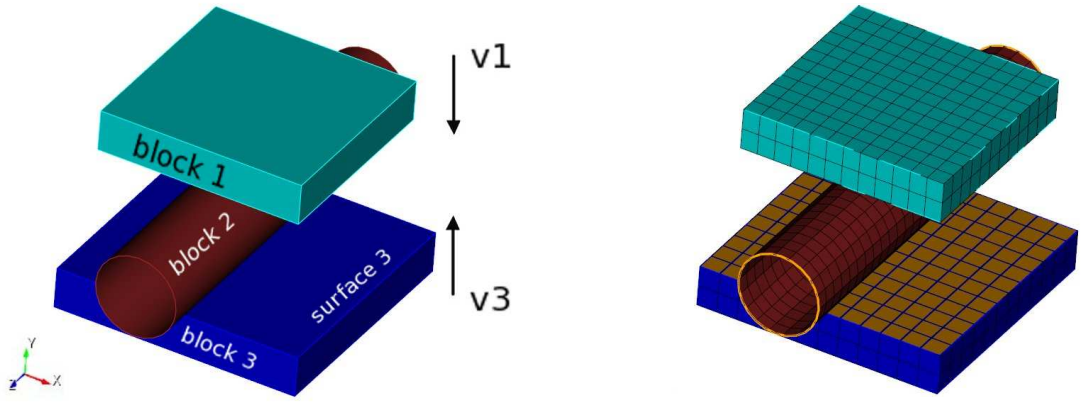


Figure 8-23. Problem with three blocks coming into contact.

```
  #
  # set interactions
  BEGIN INTERACTION DEFAULTS
    FRICTION MODEL = rough
    GENERAL CONTACT = ON
    SELF CONTACT = ON
  END

  # set specific interaction
  BEGIN INTERACTION
    SURFACES = b2 s3
    FRICTION MODEL = FRICTIONLESS
  END

END
```

In the `EXAMPLE2` command block, three surfaces have been defined. The first surface, `b1`, is obtained by skinning `block_1`. The second surface, `b2` is obtained by skinning `block_2`. The third surface, `s3`, is the user-defined side set `surface_3`.

In the `INTERACTION DEFAULTS` command block, we select the friction model `ROUGH` on the `FRICTION MODEL` command line. Both `GENERAL CONTACT` and `SELF CONTACT` are set to `ON`. For this problem the hollow cylinder, `block_2`, will undergo self-contact. Setting `GENERAL CONTACT` to `ON` will enforce contact between `b1` and `b2`, `b2` and `s3`, and `b1` and `s3`. Setting `SELF CONTACT` to `ON` will enable self-contact for all three of the surfaces.

For this particular example, we want to override some of the default setting for surface interactions which were previously set by the `INTERACTION DEFAULTS` command block. The `INTERACTION` command block shown in `EXAMPLE2` overrides these defaults and instead specifies surfaces `b2` and `s3` to have a frictionless interaction.

In Figure 8-24, the bodies are shown after deformation and contact. The second image shows a cross section cut through the center. Due to the lofting of the shell surface there is a small gap between the shell elements themselves and the compressing blocks, which is expected. The actual

contact geometry is shown in Figure 8-25. Note that as Block 1 is skinned, the entire exterior surface of block 1 is included in contact surface b1. The lofted geometry of the shell blocks is seen in this figure, this lofted geometry is both contacting with the blocks and experiencing self contact. Contact surface s3 contains only the top sideset on block 3, not the entire skin.

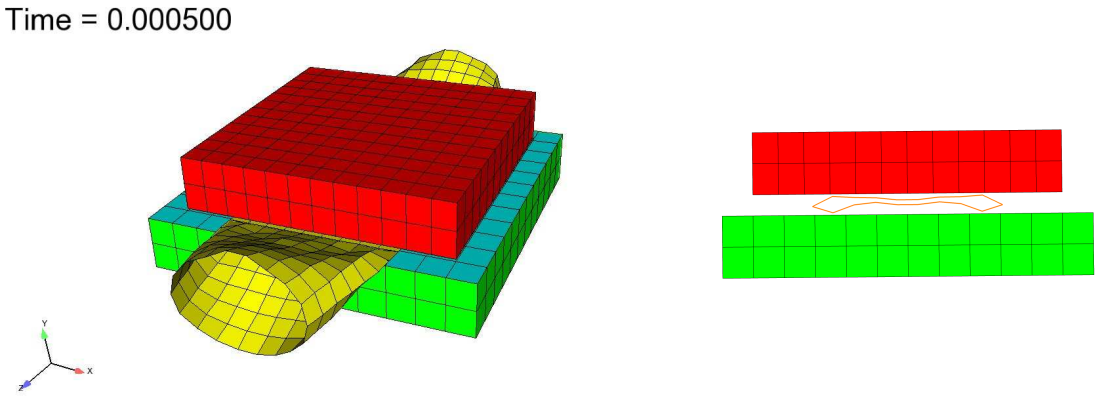


Figure 8-24. State of bodies after contact and deformation.

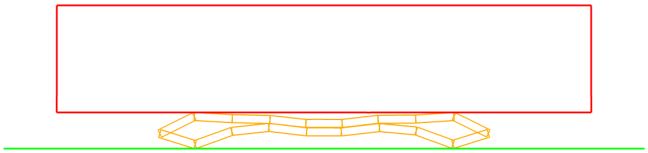


Figure 8-25. Deformed contact geometry.

8.13. Common Contact Algorithmic Issues

This sections describes some of the common problem issues that may be encountered when setting up contact.

Contact constraints are one type of constraint among many possible types of constraints. Other constraint types include MPCs 7.15, kinematic constraints 7, and welds 7.15.3. Contact constraints can potentially conflict with these other constraint types causing unexpected behavior. Refer to Appendix D for information on how conflicting constraints are handled.

Tied contact that occurs over a substantial distance is not guaranteed to preserve energy or angular momentum at the interface. The issue becomes more problematic as the distance between tied objects becomes a substantial fraction of the contact face size. It is recommended that default search tolerances are used so that only objects in close proximity are tied.

Lofted objects that are large compared to the element size are problematic. For example, lofting of shells that are thicker than they are wide or beams with a large radius compared to their element length.

The speed of contact is based primarily on the number of nodes and faces in the contact surfaces and, to a lesser extent, on the number of interactions specified. Choosing excessively large search tolerances will greatly increase the run time required by contact.

Contact performs best when objects in contact are on the same processor. It is strongly recommended to use a proximity based mesh decomposition method such as RCB or inertial for contact problems. If large motions occur during the problem solution, dynamic mesh rebalancing may be needed to maintain good performance, see Section 6.8.

The glued friction model is enforced by a velocity constraint. Over a very large number of steps, small integration errors can build up and potentially allow for some slip at the glued interface.

Large local mesh motions may cause inversion of the contact interface. It is recommended that shape metric based element death or other mechanisms be used to prevent the large errors that can occur when computing contact on tangled surfaces. Ensuring the nodal jacobian ratio of all elements is greater than zero will usually ensure the integrity of the contact surface definition.

Even when using frictionless interaction laws, contact interfaces will tend to be energy absorbers. This is due to contact interfaces behaving plastically rather than storing impact energy.

Initial overlap of contacting bodies may cause major instabilities in explicit contact runs. The remove initial overlap capability can be used to remove small overlap. Large overlaps must be removed by modifying the underlying geometry or running a preload step.

The implicit and explicit contact solves behave subtly different. If an implicit static preload is used and that state handed off to an explicit analysis, there will usually be a small equilibrium imbalance that needs to ring down after the transfer.

8.14. ACME Specific Commands

The Dash contact search is used by default and is the recommended methodology for all analyses. The ACME search is still maintained for backwards compatibility. this section lists ACME specific contact commands and behavior. The ACME search may be deprecated in the future leaving Dash as the only supported option.

8.14.1. ACME Specific Friction Models

The below friction models are available when using ACME search in an explicit dynamics analysis.



Warning: All ACME specific friction models will be deprecated in the future, please seek alternative modeling techniques and let sierra-help@sandia.gov know if this creates a capability gap.



8.14.1.1. Spring Weld Model

```
BEGIN SPRING WELD MODEL <string>name
  NORMAL DISPLACEMENT FUNCTION = <string>func_name
  NORMAL DISPLACEMENT SCALE FACTOR = <real>scale_factor(1.0)
  TANGENTIAL DISPLACEMENT FUNCTION = <string>func_name
  TANGENTIAL DISPLACEMENT SCALE FACTOR =
    <real>scale_factor(1.0)
  FAILURE ENVELOPE EXPONENT = <real>exponent(2.0)
  FAILURE DECAY CYCLES = <integer> num_cycles(1)
  FAILED MODEL = <string>failed_model_name|FRICTIONLESS
    (FRICTIONLESS)
 END [SPRING WELD MODEL <string>name]
```

The `SPRING WELD MODEL` command block defines a contact friction model that, when applied between two contact surfaces, connects a side B node to the nearest point of a corresponding side A face with a spring. The spring behavior is defined by a force-displacement curve in the normal and tangential directions. If the motion of the problem generates displacement between the side B node and its corresponding side A face and this motion is in purely the normal or tangential direction, the spring will fail once it passes the maximum displacement in the normal and tangential force-displacement curves, respectively. For displacements that include both normal and tangential components, the spring fails according to a failure criterion defined as the sum of the ratios of the normal and tangential components to their maximum values, raised to a power. If the criterion is greater than 1.0, the spring fails. Once the spring fails, its contact forces reduce over a number of load steps, and the contact evaluation reverts to another user-specified friction model (or frictionless contact if not specified).

In the above command block:

- The string `name` is a user-selected name for this friction model that is used to identify this model in the interaction definitions.
- The normal force-displacement curve is specified by the `NORMAL DISPLACEMENT FUNCTION` command line, where the string `func_name` is the name of a function defined in a `FUNCTION` command line in the SIERRA scope. This function can be scaled by the real value `scale_factor` in the `NORMAL DISPLACEMENT SCALE FACTOR` command line; the default for this factor is 1.0
- The tangential force-displacement curve is specified by the `TANGENTIAL DISPLACEMENT FUNCTION` command line, where the string `func_name` is the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the SIERRA scope. This function can be scaled by the real value `scale_factor` in the `TANGENTIAL DISPLACEMENT SCALE FACTOR` command line; the default for this factor is 1.0.
- The real value `exponent` in the `FAILURE ENVELOPE EXPONENT` command line specifies how normal and tangential failure criteria may be combined to yield failure of the weld, as described above. The default value for this exponent is 2.0.
- The `FAILURE DECAY CYCLES` command line describes how many cycles to ramp down the load in the spring weld after it fails through the integer value `num_cycles`. The default value for the number of decay cycles is 1.
- When the spring weld breaks, the friction model that contact reverts to when evaluating future node-face interactions between the surfaces is identified in the `FAILED MODEL` command line with the string `failed_model_name`. The friction model listed in this command must have been previously defined in the input file. The default value for the model used after failure is the frictionless model.

The `SPRING WELD MODEL` command block is similar to the Sierra/SM `SPOT WELD` command block, but permits greater flexibility in specifying a different friction model to be applied after failure.



8.14.1.2. Surface Weld Model

```
BEGIN SURFACE WELD MODEL <string>name
  NORMAL CAPACITY = <real>normal_cap
  TANGENTIAL CAPACITY = <real>tangential_cap
  FAILURE DECAY CYCLES = <integer>num_cycles(1)
  FAILED MODEL = <string>failed_model_name|FRICTIONLESS
    (FRICTIONLESS)
END [SURFACE WELD MODEL <string>name]
```

The `SURFACE WELD MODEL` command block defines a contact friction model that behaves identically to the `TIED MODEL` until a maximum force between the node and face of an interaction is reached in the normal direction or the tangential direction. Once this maximum force is reached, the tied contact "fails" and the friction model switches to a different friction model, as specified by the user.

In the above command block, the string `name` is a user-selected name for this friction model that is used to identify this model in the interaction definitions. The maximum allowed force in the

normal direction is specified by the real value normal_cap in the NORMAL CAPACITY command line. The maximum allowed force in the tangential direction is specified by the real value tangential_cap in the TANGENTIAL CAPACITY command line. There are no defaults for these values. The surface weld will break when either the specified normal or tangential capacity is reached. Once the model fails, the applied forces decrease to zero over a number of time steps defined through the integer value num_cycles in the FAILURE DECAY CYCLES command line. The default for num_cycles is 1. The friction model that should be used after the weld fails is identified in the FAILED MODEL command line with the string failed_model_name. The friction model designated in the FAILED MODEL command line must be defined within the CONTACT DEFINITION command block. The default model after failure is the frictionless contact model.



8.14.1.3. Area Weld Model

```
BEGIN AREA WELD MODEL <string>name
  NORMAL CAPACITY = <real>normal_cap
  TANGENTIAL CAPACITY = <real>tangential_cap
  FAILURE DECAY CYCLES = <integer> num_cycles(1)
  FAILED MODEL = <string>failed_model_name|FRICTIONLESS
    (FRICTIONLESS)
END [AREA WELD MODEL <string>name]
```

The AREA WELD MODEL command block defines a contact friction model that behaves identically to the TIED MODEL until a maximum traction between a node and face in an interaction is reached in the normal direction or the tangential direction. Once this maximum traction is reached, the tied contact "fails" and the friction model switches to a different friction model, as specified by the user. This model is identical to the SURFACE WELD MODEL command block, except that tractions are used instead of forces.

In the above command block, the string name is a user-selected name for this friction model that is used to identify this model in the interaction definitions. The maximum allowed traction in the normal direction is specified by the real value normal_cap in the NORMAL CAPACITY command line. The maximum allowed traction in the tangential direction is specified by the real value tangential_cap in the TANGENTIAL CAPACITY command line. There are no defaults for these values. The area weld will break when either the specified normal or tangential capacity is reached. Once the model fails, the applied tractions decrease to zero over a number of time steps defined through the integer value num_cycles in the FAILURE DECAY CYCLES command line. The default for num_cycles is 1. The friction model that should be used after the weld fails is identified in the FAILED MODEL command line with the string failed_model_name. The friction model designated in the FAILED MODEL command line must be defined within the CONTACT DEFINITION command block. The default model after failure is the frictionless contact model.



8.14.1.4. Adhesion Model

```
BEGIN ADHESION MODEL <string>name
```

```
  ADHESION FUNCTION = <string>func_name
  ADHESION SCALE FACTOR = <real>scale_factor(1.0)
END [ADHESION MODEL <string>name]
```

The ADHESION MODEL command block defines a friction model that behaves like frictionless contact when two surfaces are in contact, but computes an additional force between the surfaces when they are not touching. The value of the additional force is given by a user-specified function of force versus distance, where the distance is the distance between a node and the closest point on the opposing surface.

In the above command block, the string name is a user-selected name for this friction model that is used to identify this model in the interaction definitions. The force between surfaces that are not touching is given by the ADHESION FUNCTION command line, where the string func_name is the name of a function defined in a FUNCTION command block in the SIERRA scope. The values of this function are expected to be non-negative. The function can be scaled by the real value scale_factor in the ADHESION SCALE FACTOR command line; the default for this factor is 1.0. Because contact forces are typically only given to node-face interactions if they touching, the contact search requires appropriate tolerances when this model is used. The normal and tangential tolerances specified in the interaction definitions should be set to the maximum distance at which the adhesion model should be applying force. However, setting this distance to be large may cause excessive numbers of interactions to be identified in the search phase, causing the contact processing to be extremely slow and/or generate erroneous interactions.



8.14.1.5. Junction Model

```
BEGIN JUNCTION MODEL <string>name
  NORMAL TRACTION FUNCTION = <string>func_name
  NORMAL TRACTION SCALE FACTOR = <real>scale_factor(1.0)
  TANGENTIAL TRACTION FUNCTION = <string>func_name
  TANGENTIAL TRACTION SCALE FACTOR = <real>scale_factor(1.0)
  NORMAL CUTOFF DISTANCE FOR TANGENTIAL TRACTION = <real>distance
END [JUNCTION MODEL <string>name]
```

Note: This model is available only in explicit dynamics with ACME contact

The JUNCTION MODEL command block defines a model that prevents the interpenetration of contact surfaces and that also provides normal and tangential tractions to a node-face interaction when the surfaces are not touching. The normal tractions are defined as a function of the normal distance between the node and face of an interaction, while the tangential traction is given as a function of the relative tangential velocity. The tractions are defined by user-specified functions, and the tangential tractions from this model drop to zero once the normal distance between the node and the face exceeds a critical value. This friction model provides a simple way to model threaded connections, though the THREADED MODEL defined in Section 8.14.1.6 has more flexibility.

In the above command block, the string name is a user-selected name for this friction model that is used to identify this model in the interaction blocks. The normal traction curve is specified by

the NORMAL TRACTION FUNCTION command line, where the string func_name is the name of a function defined in a DEFINITION FOR FUNCTION command block in the SIERRA scope. This function defines a relation between the traction and the distance between the node and the face in the normal direction. This function can be scaled by the real value scale_factor in the NORMAL TRACTION SCALE FACTOR command line; the default for this factor is 1.0. Similarly, the tangential traction curve is specified by the TANGENTIAL TRACTION FUNCTION command line, where the string func_name is the name of a function defined in a FUNCTION command block in the SIERRA scope. This function defines a relation between the traction and the relative velocity of the node and face in the tangential direction. This function can be scaled by the real value scale_factor in the TANGENTIAL TRACTION SCALE FACTOR command line; the default for this factor is 1.0. Once the normal distance between a node and a face using this model reaches a critical distance, the tangential traction drops to zero; this distance is specified with the real value distance in the NORMAL CUTOFF DISTANCE FOR TANGENTIAL TRACTION command line.



8.14.1.6. Threaded Model

```
BEGIN THREADED MODEL <string>name
  NORMAL TRACTION FUNCTION = <string>func_name
  NORMAL TRACTION SCALE FACTOR = <real>scale_factor(1.0)
  TANGENTIAL TRACTION FUNCTION = <string>func_name
  TANGENTIAL TRACTION SCALE FACTOR = <real>scale_factor(1.0)
  TANGENTIAL TRACTION GAP FUNCTION = <string>func_name
  TANGENTIAL TRACTION GAP SCALE FACTOR = <real>scale_factor(1.0)
  NORMAL CAPACITY = <real>normal_cap
  TANGENTIAL CAPACITY = <real>tangential_cap
  FAILURE ENVELOPE EXPONENT = <real>exponent(2.0)
  FAILURE DECAY CYCLES = <integer>num_cycles(1)
  FAILED MODEL = <string>failed_model_name|FRICTIONLESS
    (FRICTIONLESS)
END [THREADED MODEL <string>name]
```

The THREADED MODEL command block defines a friction model that is designed to mimic a threaded interface. This model prevents interpenetration of contact surfaces, and also supplies additional tractions when the surfaces are not touching. Tensile tractions in the normal direction are given by a user-specified function of force versus distance between the node and face. Tensile tractions in the tangential direction are computed as the product of a traction tangential-displacement curve and a scaling curve that is a function of the normal displacement. Maximum normal and tangential tractions are input such that the model "fails" at a node-face interaction once they are reached. For interactions that include both normal and tangential displacements, the model failure is defined according to a failure criterion defined as the sum of the ratios of the normal and tangential traction components to their maximum capacity values, raised to a power. After failure, interactions shift to a different user-specified friction model.

In the above command block:

- The string name is a user-selected name for this friction model that is used to identify the

model in the interaction definitions.

- The traction-displacement relation in the normal direction is specified by the `NORMAL TRACTION FUNCTION` command line, where the string `func_name` is the name of a function defined in a `FUNCTION` command block in the SIERRA scope. This function can be scaled by the real value `scale_factor` in the `NORMAL TRACTION SCALE FACTOR` command line; the default for this factor is 1.0.
- The traction-displacement relation in the tangential direction is specified by two curves. The traction-displacement relation in the tangential direction when there is no displacement in the normal direction is defined by the `TANGENTIAL TRACTION FUNCTION` command line, where the string `func_name` is the name of a function defined in a `FUNCTION` command block in the SIERRA scope. This function can be scaled by the real value `scale_factor` in the `TANGENTIAL TRACTION SCALE FACTOR` command line; the default for this factor is 1.0. When the distance in the normal direction is greater than zero, the tangential traction is scaled by a function specified in the `TANGENTIAL TRACTION GAP FUNCTION` command line, where the string `func_name` is the name of a function defined in a `FUNCTION` command block in the SIERRA scope. This function defines a scaling factor as a function of the normal displacement. The function can be scaled by the real value `scale_factor` in the `TANGENTIAL TRACTION GAP SCALE FACTOR` command line; the default for this factor is 1.0.
- The threaded model "fails" once the normal and tangential tractions reach a critical capacity value. The normal capacity is specified by the real value `normal_cap` in the `NORMAL CAPACITY` command line. The tangential capacity is specified by the real value `tangential_cap` in the `TANGENTIAL CAPACITY` command line. There are no default values for these parameters. These capacities are defined for pure normal or tangential displacements. In cases where there is a combination of tangential and normal displacements, a failure curve is used to determine the combination of tangential and normal tractions that determines model failure. This curve is defined as the sum of the ratios of the normal and tangential traction components to their maximum capacity values, raised to a power. The power in the function is defined by the real value exponent in the `FAILURE ENVELOPE EXPONENT` command line. The default value of the exponent is 2.0. Once the model fails, the applied tractions decrease to zero over a number of time steps defined through the integer value `num_cycles` in the `FAILURE DECAY CYCLES` command line. The default for `num_cycles` is 1. When the model exceeds the designated capacity, the contact surfaces using this model switch to a different friction model as identified in the `FAILED MODEL` command line with the string `failed_model_name`. The friction model designated in the `FAILED MODEL` command line must be defined within the `CONTACT DEFINITION` command block. The default model is the frictionless model.



8.14.1.7. PV_Dependent Model

```
BEGIN PV_DEPENDENT MODEL <string>name
  STATIC COEFFICIENT = <real>stat_coeff
  DYNAMIC COEFFICIENT = <real>dyn_coeff
  VELOCITY DECAY = <real>vel_decay
```

```
  REFERENCE PRESSURE = <real>p_ref
  OFFSET PRESSURE = <real>p_off
  PRESSURE EXPONENT = <real>p_exp
END [PV_DEPENDENT MODEL <string> name]
```

The `PV_DEPENDENT MODEL` command block defines a friction model similar to a coulomb friction model, but which provides a frictional response that is dependent on the pressure and the velocity. The pressure-dependent portion of the model behaves similarly to the constant friction model except that the tangential traction is given by

$$\left[\frac{p + \text{p\_off}}{\text{p\_ref}}\right]^{\text{p\_exp}}. \tag{8.3}$$

The velocity-dependent part is given by

$$\left(\text{stat\_coeff} - \text{dyn\_coeff}\right) e^{\left(-\text{vel\_decay}\|v\|\right)} + \text{dyn\_coeff}. \tag{8.4}$$

The `PV_DEPENDENT MODEL` command block multiplies the pressure and velocity effects together.

In the above command block:

- The string `name` is a name assigned to this friction model that is used to identify the model in the interaction definitions.
- The real value `p_ref` in the pressure-dependent part given in Equation (8.3) is specified with the `REFERENCE PRESSURE` command line.
- The real value `p_off` in the pressure-dependent part given in Equation (8.3) is specified with the `OFFSET PRESSURE` command line.
- The real value `p_exp` in the pressure-dependent part given in Equation (8.3) is specified with the `PRESSURE EXPONENT` command line.
- The real value `stat_coeff` in the velocity-dependent part given in Equation (8.4) is specified with the `STATIC COEFFICIENT` command line.
- The real value `dyn_coeff` in the velocity-dependent part given in Equation (8.4) is specified with the `DYNAMIC COEFFICIENT` command line.
- The real value `vel_decay` in the velocity-dependent part given in Equation (8.4) is specified with the `VELOCITY DECAY` command line.



8.14.1.8. User Subroutine Friction Models

```
BEGIN USER SUBROUTINE MODEL <string>name
  INITIALIZE MODEL SUBROUTINE = <string>init_model_name
  INITIALIZE TIME STEP SUBROUTINE = <string>init_ts_name
  INITIALIZE NODE STATE DATA SUBROUTINE =
    <string>init_node_data_name
  LIMIT FORCE SUBROUTINE = <string>limit_force_name
  ACTIVE SUBROUTINE = <string>active_name
```

```
  INTERACTION TYPE SUBROUTINE = <string>interaction_name
  NUM INTEGER VALS = <int>num_int_vals
  INTEGER VALS = <int>int_vals
  NUM REAL VALS = <int>num_real_vals
  REAL VALS = <real>real_val
  NUM HISTORY VALS = <int>num_hist
  USER FUNCTION <int>id =
    <string>name SCALED BY <real>scale
END [USER SUBROUTINE MODEL <string>name]
```

The `USER SUBROUTINE MODEL` command blocks permit contact to use a user subroutine to define a friction model between surfaces. This capability is in a test phase at this time; please contact a Sierra/SM developer for more information.

In this command block:

- The string `name` is a user-specified name that is used to identify this model in the interaction definitions.
- The command line `INITIALIZE MODEL SUBROUTINE` specifies a user subroutine to initialize the friction model. The name of the subroutine is given by `init_model_name`.
- The command line `INITIALIZE TIME STEP SUBROUTINE` specifies a user subroutine to initialize the time step. The name of the subroutine is given by `init_ts_name`.
- The command line `INITIALIZE NODE STATE DATA SUBROUTINE` specifies a user subroutine to initialize the node state data. The name of the subroutine is given by `init_node_data_name`.
- The command line `LIMIT FORCE SUBROUTINE` specifies a user subroutine to provide the limit force for the friction model. The name of the subroutine is given by `limit_force_name`.
- The command line `ACTIVE SUBROUTINE` specifies a user subroutine to compute forces for an active node-face interaction. The name of the subroutine is given by `active_name`.
- The command line `INTERACTION TYPE SUBROUTINE` specifies a user subroutine to define the interaction type. The name of the subroutine is given by `interaction_name`.

For documentation of the remaining commands see the ACME manual.

8.14.2. Kinematic Partition

Kinematic partitioning is used only for explicit contact analysis. When computing two way contact between surface pairs ACME explicit enforcement uses a concept of kinematic partitioning to speed the convergence of the contact enforcement iterations.

The interaction defaults command block may contain the line

```
AUTOMATIC KINEMATIC PARTITION = <string>ON|OFF(OFF)
```

The interaction command block may contain the lines

```
AUTOMATIC KINEMATIC PARTITION = <string>ON|OFF(OFF)
KINEMATIC PARTITION = <real>kin_part(0.5)
```

If the AUTOMATIC KINEMATIC PARTITION command line is used, Sierra/SM will automatically compute the kinematic partition factors for pairs of surfaces. The automatic kinematic partitions are computed from the impedance of each surface based on nodal average density and wave speed. Automatic computation of kinematic partition factors provides the best approach to exact enforcement of symmetric contact of opposing surfaces provided that these surfaces have the same mesh resolution. If the mesh resolution is disparate, you can specify the coarser meshed body as side A, but generally it is better to use more contact iterations to deal with this case of a fine mesh contacting a coarse mesh.

For the interaction of any two surfaces, the sum of the partition factors for the surfaces must be 1.0. This is automatically taken care of when the AUTOMATIC KINEMATIC PARTITION command line is used. The default value for kinematic partition factors for all surfaces is 0.5.

The AUTOMATIC KINEMATIC PARTITION command line can be used to set the kinematic partitions for all interactions or to set the kinematic partitions for specific interactions. Thus the command line can appear in two different scopes:

1. The command line can be used within the INTERACTION DEFAULTS command block. In this case, all contact surface interactions defined in the command block will use the automatic kinematic partitioning scheme by default. This will override the default case that assigns a kinematic partition factor of 0.5 to all surfaces. For particular interactions, it is possible to override the use of the automatic kinematic partition factors by specifying kinematic partition values (with the KINEMATIC PARTITION command line) within the INTERACTION command blocks for those interactions.
2. The command line can be used inside an INTERACTION command block. If the automatic partitioning command line appears inside an INTERACTION command block, the kinematic partition factors for that particular interaction will be calculated by the automatic kinematic partition scheme.

Automatic kinematic partitioning is not currently operational for shell elements. If it is enabled, the kinematic partitioning factor is set to 0.5 for all contact interactions involving shell elements.

To provide accurate contact evaluation, Sierra/SM typically computes two-way contact between two surfaces, where interactions are defined between the nodes of the first surface and the faces of the second surface, and also between the nodes of the second surface and the faces of the first surface. If the two surfaces have penetrated each other by a distance δ, then each of the contact evaluations will compute forces to move the surface a distance δ, so that the total resulting displacement would be 2δ if both sets of contact computations were fully applied. The KINEMATIC PARTITION command line defines a kinematic partition scaling factor, kin_part, for the two contact computations so that the total contact motion is correct. The kinematic partition factor, kin_part, is a value between 0.0 and 1.0. The factor scales the relative motion of the first surface, where kin_part = 0.0 means the first surface will move none of δ, while kin_part = 1.0 means the surface moves all of δ. The second surface moves the portion of δ that

remains after the motion of the first surface (i.e., δ for second surface = 1.0 – `kin_part`). For instance, if `kin_part` is 0.2, the first surface will move 20% of the penetration distance (0.2δ in our example), and the second surface would move the remaining 80% (0.8δ in our example). The default value is 0.5, so that each surface would move half of the penetration distance. If `kin_part` is 0.0, the first surface does not move at all, and the second surface moves the full distance. This is exactly equivalent to a one-way contact definition, where the first surface is the side A and the second is the side B. If `kin_part` is 1.0, the second surface is the side A, and the first surface is the side B. Figure 8-26 illustrates how the kinematic partition factor varies from 0.0 to 1.0, with the specific example of `kin_part` being set to 0.2.

```
begin interaction
 surfaces = <first_surface> <second_surface>
 kinematic partition = 0.2
end
```

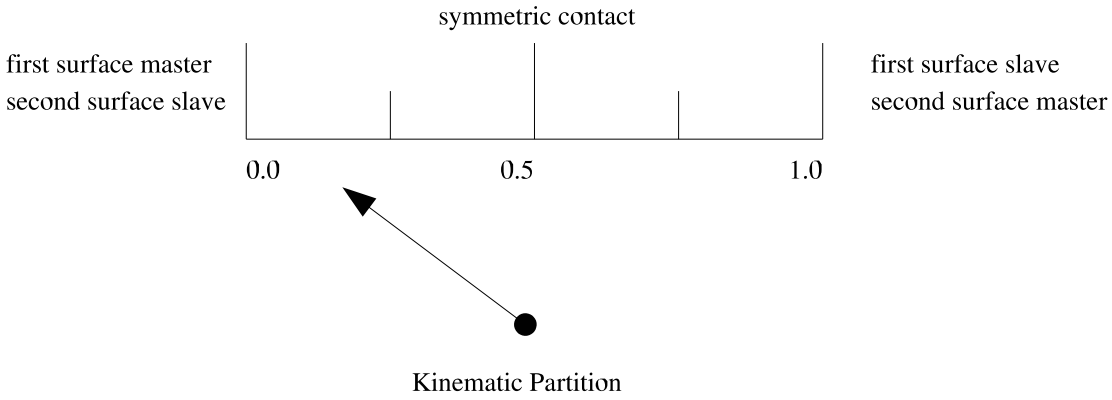


Figure 8-26. Illustration of kinematic partition values

The capability provided by the `KINEMATIC PARTITION` command line is important in cases where contact occurs between two materials of disparate stiffness. Physically, we would expect a material with a higher stiffness to have more of an effect in determining the position of the contact surface than a more compliant material. In this case, we want the softer material to move more of the distance, and thus it should have a higher kinematic partition factor. The appropriate kinematic partition factor can be determined in closed form; see the ACME contact library reference [1] for more information. Alternately, the `AUTOMATIC KINEMATIC PARTITION` capability can automatically calculate the proper kinematic partition based on the stiffness of the materials.

Another case where the kinematic partition factor has traditionally been used is when meshes with dissimilar resolutions contact each other. If an interaction is defined with a fine mesh as the side A surface and a coarse mesh as a side B surface, the contact algorithms will permit nodes on the side A surface to penetrate the side B surface. In these cases, such problems can be alleviated by making the coarse mesh the side A surface. However, the iterative approach implemented in the enforcement can also take care of this problem and is advised. If few iterations are chosen, an appropriate kinematic partition factor may be needed to prevent unintentional penetration due to mesh discretization.

For self-contact, the kinematic partition factor should be 0.5.

A kinematic partition factor cannot be defined for interactions that use the pure side A-side B syntax (see Section 8.6.2.1).

In general, it is best to use the automatic kinematic partition option to properly compute the kinematic partition for a pair of surfaces. However, in a few cases, side A-side B interactions are preferred. These cases consist of (1) interaction between a kinematically rigid surface and a deformable body, where the kinematically rigid surface should be the side A surface; and (2) contact between shells and solids, where the solid should be the side A surface.

8.14.3. ACME automated search tolerances

The methodology to compute default search tolerances can be set with the following commands in the `SEARCH OPTIONS` command blocks.

```
SEARCH TOLERANCE = AUTOMATIC|USER_DEFINED|DASH_AUTOMATIC(AUTOMATIC)
FACE MULTIPLIER = <real>face_multiplier(0.1)
```

By default both Dash and ACME contact searches define automated search tolerances. The search tolerance algorithms used by Dash and ACME contact search differ somewhat. It is possible to have ACME use the Dash tolerances (which are often more robust) with use of the `DASH_AUTOMATIC` option to the `SEARCH TOLERANCE` command. If the `USER_DEFINED` option is used with the `SEARCH TOLERANCE` command ACME will use no automated search tolerances at all and rely only on the search tolerance values given in the input deck. When the `USER_DEFINED` option is used the normal and tangential tolerances must be specified. If these tolerances are not specified, code execution will be terminated with an error. The default `AUTOMATIC` search tolerance method computes automatic search tolerances based on the minimum characteristic length of any faces in the contact surface multiplied by the `face_multiplier`.

The face size multiplier used by the ACME `AUTOMATIC` tolerancing method can be set for all interactions by use of the `FACE MULTIPLIER` command in `SEARCH OPTIONS` command block. The `FACE MULTIPLIER` command can also be used in individual `INTERACTION` command blocks.

The automatic tolerances add the maximum motion over a time step just like the user defined tolerances. If you leave automatic search on and also specify normal and/or tangential tolerances with the `NORMAL TOLERANCE` and `TANGENTIAL TOLERANCE` command lines, the larger of the two (automatic or user specified) tolerances will be used. For example, suppose you specify a normal tolerance of 1.0×10^{-3} and the automatic tolerancing computes a normal tolerance of 1.05×10^{-3}. Then Sierra/SM will use a normal tolerance of 1.05×10^{-3}.



8.14.4. Frictional behavior without Friction Model

ACME may define coulomb frictional behavior without explicitly defining a coulomb friction model with use of the following command in an `INTERACTION` command block.

```
FRICTION COEFFICIENT = <real>coeff
FRICTION COEFFICIENT FUNCTION = <string>coeff_func
```

The FRICTION COEFFICIENT command line allows setting a friction coefficient irrespective of the friction model used. The command line FRICTION COEFFICIENT FUNCTION provides a way for the friction coefficient to be set using a time history function from the input file. If the FRICTION COEFFICIENT line command is also present, the friction coefficient used by the code will be the FRICTION COEFFICIENT value multiplied by the value obtained by the function.



8.14.5. Shell Overlap Removal Iterations

For contact with lofted geometry ACME has commands to remove overlap. Because the thickness of a shell must be preserved when shell lofting is requested, removing the initial overlap between nested shells becomes an iterative process whereby shell locations are adjusted to remove the overlap. This process is approximate and may not remove all the overlap in all cases. It is advised to check the corrected mesh to make sure that the mesh modifications are acceptable. For ACME two additional input lines may be needed in the REMOVE INITIAL OVERLAP command block to robustly remove overlap.

```
SHELL OVERLAP ITERATIONS = <integer>max_iter(10)
SHELL OVERLAP TOLERANCE = <real>shell_over_tol(0.0)
```

- ACME Only: The SHELL OVERLAP ITERATIONS command line controls the maximum number of iterations used by the overlap removal mechanism to resolve nested shells. By default, the value of max_iter is 10. If the mesh has only a few layers of shells that may overlap, a value of 10 should suffice. However, if the mesh has a number of layers of shells that may overlap, this value may need to be larger.
- The SHELL OVERLAP TOLERANCE command line specifies an amount of overlap that is permitted to be left in the shell elements. This helps to limit the actual number of iterations required to remove the shell overlap, and to spread any remaining overlap over a number of shells instead of concentrating it all in a single shell. If the default value of 0.0 for the shell overlap tolerance is used, iteration continues until either all the overlap is removed or the maximum number of iterations is reached. If a nonzero value for the shell overlap tolerance is used, iteration continues until the tolerance is reached or the maximum number of iterations is reached. Note that the overlap removal process is only done once during an analysis, so a large number of iterations will only affect the first time step, not every time step.

The SHELL OVERLAP ITERATIONS and SHELL OVERLAP TOLERANCE commands have no meaning for analyses that do not have shell elements.

8.14.6. Surface Normal Smoothing

```
BEGIN SURFACE NORMAL SMOOTHING
  ANGLE = <real>angle_in_deg(60.0)
  DISTANCE = <real>distance(0.01)
```

```
  RESOLUTION = <string>NODE|EDGE(NODE)
END SURFACE NORMAL SMOOTHING
```

Surface normal smoothing is a feature that is primarily used in implicit contact.

Finite element discretization often results in models with faceted edges, while the true geometry of the part is actually smoothly curved. If the faces of adjacent finite elements on a surface have differing normals, the discontinuities at the edges between those faces can cause problems with contact. These discontinuities in the face normals are particularly troublesome with an implicit code such as Adagio, which uses an iterative solver to obtain a converged solution at every step. If a node is in contact near an edge with a normal discontinuity, the node may slide back and forth between the two neighboring faces during the iterations. Because the normal directions of the two faces differ, this can make it difficult to converge on a solution to this discontinuous contact problem.

Surface normal smoothing is a technique that creates a smooth variation in the normal near edges. The normal varies linearly from the value on one face to the value on the other face over a distance that spans the edge. A smoothly varying normal at the edge makes it easier for an iterative solver to obtain a converged solution in the case where a node has penetrated near the edge of a face.

Explicit dynamics does not use an iterative solver and thus does not encounter the difficulties associated with face normal discontinuities. Consequently, the `SURFACE NORMAL SMOOTHING` command block is not typically useful for explicit models, though is available in both explicit and implicit contact.

If the `SURFACE NORMAL SMOOTHING` command block is present, this feature is activated. There are three optional commands that can be used within this block to control the behavior of normal smoothing.

- The `ANGLE` command is used to control whether smoothing should occur between neighboring faces. If the angle between two faces is less than the specified angle (given in degrees), smoothing is activated between them. Otherwise, the discontinuity is considered to be a feature of the model rather than an artifact of meshing, and they are not smoothed. The default value for `angle` is 60.
- The `DISTANCE` command specifies the distance as a fraction of the face size over which smoothing should occur. The specified value can vary from 0 to 1. The default value for `distance` is 0.01.
- The `RESOLUTION` command specifies the method used to determine the smoothed normal direction. The default `NODE` option uses a node-based algorithm to fit a smooth curve, while the `EDGE` option uses an edge-based algorithm.



8.14.7. Angle for Multiple Interactions

```
MULTIPLE INTERACTIONS = <string>ON|OFF(ON)
MULTIPLE INTERACTIONS WITH ANGLE = <real>angle_in_deg(60.0)
```

When a node lies on the edge of a body, that node may need to support contact interactions with more than one face at the same time. For instance, see Figure 8-27. In Figure 8-27a, three blocks

are shown, with a single node identified. Through contact, this node can interact with both the block on the upper right and the block on the bottom. If the node only supports a single interaction, then it will be arbitrarily considered for contact between one of the blocks, but not the other, as in Figure 8-27b. In this case, contact enforcement will prevent penetration into the lower block, but may permit penetration into the upper right block. The proper way to deal with this case is shown in Figure 8-27c, where multiple interactions are considered at the node.

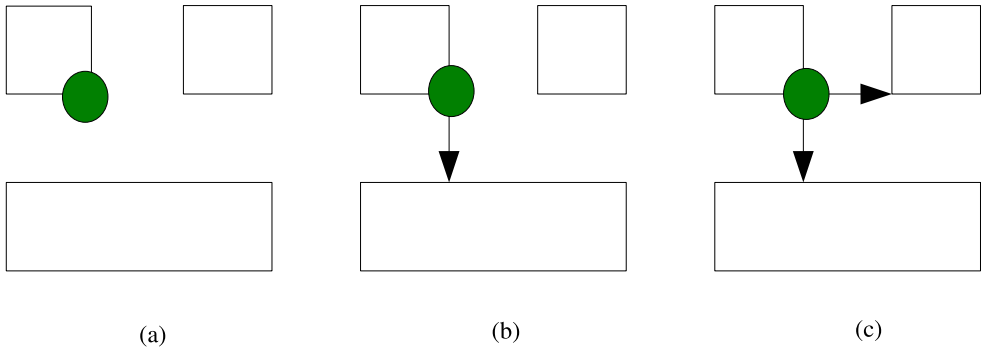


Figure 8-27. Illustrations of multiple interactions at a node: (a) initial configuration with node of interest identified (b) single interaction (c) multiple interactions

By default, Sierra/SM permits multiple interactions at a node. However, these multiple interactions may incur extra cost in the contact algorithm by increasing the number of interactions in enforcement. Also, a local search algorithm (see Section 8.9.6), which uses various contact tracking approaches, may operate more efficiently when the node can only have one interaction. Finally, multiple interactions may lead to instabilities that can be eliminated by switching to single interactions. For these reasons, the `MULTIPLE INTERACTIONS` command line allows the user to choose whether multiple interactions should be considered at a node. A value of `OFF` indicates that a node can have only one interaction. This value affects all interactions in a contact definition. Sierra/SM does not currently have the capability to force single interactions for some surface pairs while allowing multiple interactions for other surface pairs.

When the `MULTIPLE INTERACTIONS` command line is `ON`, the number of interactions that can be considered at a node is dependent on the measure of curvature of those faces that are connected to the node. If the angle between two faces on which the node is attached is small, then only one interaction is allowed. However, in cases where the angle between the faces is large enough such that they form a discrete corner, multiple interactions are considered. The contact algorithms can properly handle only a limited number of interactions per node (currently three), so it is generally feasible to properly define interactions at a node, e.g., at the corner of a block.

The critical angle for multiple interactions is set with the `MULTIPLE INTERACTIONS WITH ANGLE` command line, where `angle` is the angle over which an edge is considered sharp. If the angle between adjoining faces is greater than this critical angle, multiple interactions can be created. By default, this critical angle is 60 degrees, which works well for most analyses. This value can be changed in the contact input if needed.

8.14.8. ACME Specific Lofted Geometry Options

ACME allows additional options in the SEARCH OPTIONS command block.

```
LOFTING ALGORITHM = <string>ON|OFF(ON)
COINCIDENT SHELL TREATMENT = <string>DISALLOW|IGNORE|
  SIMPLE(DISALLOW)
COINCIDENT SHELL HEX TREATMENT = <string>DISALLOW|
  IGNORE|TAPERED|EMBEDDED(DISALLOW)
```

In ACME contact the contact on shell elements can occur on either the meshed shell geometry, ignoring any shell thickness, or on the "lofted" geometry, including the thickness of the shell.

The LOFTING ALGORITHM command line determines whether contact on a shell should be done on the lofted geometry or on the original shell geometry. If the value of the LOFTING ALGORITHM command line is set to ON, shell contact uses the lofted geometry; if it set to OFF, shell contact uses the original shell geometry.

The COINCIDENT SHELL TREATMENT command line identifies how shells that share the same nodes should be treated. If the DISALLOW option is selected, (the default), then any time that shells in contact are detected to share all the same nodes, the code will abort with an error message indicating which elements were found to be coincident. The DISALLOW option should be used if you do not want any coincident shells to be considered in the analysis. The option operates essentially as a check on the mesh. If the IGNORE option is selected, any contact faces attached to coincident shells are ignored for contact. This option is only provided as a backup approach if undiagnosed code problems arise from coincident shells. If such a case occurs, the IGNORE option may permit the user to continue with an analysis while the code team diagnoses the problem. The SIMPLE option enables coincident shells to be processed correctly. If lofting has been enabled and the SIMPLE option is selected, the thickness of the lofted coincident shell is taken as the largest thickness of all the coincident shells. If lofting is off and the SIMPLE option is selected, the coincident shell is treated as if only one of the shells is present.

The COINCIDENT SHELL HEX TREATMENT command line has a function similar to that of the COINCIDENT SHELL TREATMENT command line. The COINCIDENT SHELL HEX TREATMENT command line, however, identifies how shells that are fully coincident with the hexahedral elements are treated. If the DISALLOW option is selected (the default), then any time that a shell in contact is detected to share all the same nodes with the face of a continuum element, the code will abort with an error message indicating which elements were coincident. The DISALLOW option should be used if you do not want any shells coincident with hexahedra to be considered in the analysis. The option operates essentially as a check on the mesh. If the IGNORE option is selected, any contact faces attached to shells that are coincident with faces of continuum elements are ignored for contact. This option is only provided as a backup approach if undiagnosed code problems arise from coincident shells and continuum elements. If such a case occurs, the IGNORE option may permit the user to continue with an analysis while the code team diagnoses the problem. The TAPERED and EMBEDDED options permit shells that are coincident with faces of continuum elements to be processed in contact. The TAPERED option does two things: it includes for contact any faces that are on the free surface and ignores faces sandwiched between the shell

and the continuum element, and it automatically adjusts the lofting of the surfaces to provide a smooth transition between shells that are not coincident with the faces of the continuum elements and those that are coincident with the faces of the continuum elements. The EMBEDDED option includes for contact both free surface faces and those that are between the coincident shells and faces of the continuum elements; the option does not adjust thicknesses to make smooth transitions between shells that are not coincident with faces of continuum elements and those that are coincident with faces of continuum elements. In general, the TAPERED option is preferred; only use the EMBEDDED option if the TAPERED option causes a code problem.

8.14.9. ACME Specific Debugging Options

When using ACME the following commands may be placed in the BEGIN DEBUG command block.

```
DEBUG NODE <int>nodeId
START AT STEP = <int>step
STOP AT STEP = <int>step
TIMERS = OFF|ON
```

DEBUG NODE is used to output everything that happens in contact related to node with ID nodeId starting at step START AT STEP and ending at step STOP AT STEP. The TIMERS command may be used to turn on ACME specific runtime timing information.

8.15. Analytic Rigid Surfaces



Warning: The commands for analytic rigid surface (ARS) contact in this section are in development and have not been tested for a wide variety of applications. See the Development Manual for a description of ARS contact.

```
BEGIN INTERACTION
  FRICTION DIRECTION = LEGACY|OPPOSITE
  NAME = <string>NAME
  RESET ACCUMULATED = OFF|ON
  SLIP PREDICTOR = <real>val
  SLIP TOLERANCE = <real>val
  SST FRICTION EVALUATION = LEGACY|MEAN QUADRATURE
  TYING TOLERANCE = <real>val
END

  BEGIN SST FRICTION MODEL
    FRICTION SCALING = <real>
    MUVSSLIP FUNCTION = <string>
    PRESSURE EXPONENT = <real>
    PRESSURE REFERENCE = <real>
    SCALING FUNCTION = <string>
    XMAX = <real>
    XSTD = <real>
    YMAX = <real>
    YSTD = <real>
  END

  BEGIN TIME DEPENDENT FRICTION MODEL
    MU = <real>
    TIME FUNCTION = <string>
  END
```

8.16. Deprecated Options



Warning: The commands in this section should not be used. They are related to legacy capabilities and are slated for eventual removal from the code base. This includes syntax that has been changed or commands that have been removed.

The following contact definition block options have been renamed

```
# Use ``BEGIN SURFACE OPTIONS'' instead of
BEGIN SHELL LOFTING
  ....
END
```

The following are deprecated ACME options. The commands below either have no effect or have been deemed to cause severe robustness issues in most analyses.

```
REINITIALIZE CONTACT ON RESTART = OFF|ON(OFF)
SMOOTH EDGE <real>distance SURFACE <surf>

BEGIN ENFORCEMENT OPTIONS
  MOMENTUM BALANCE TOLERANCE = <real>val
END

BEGIN INTERACTION
  # These options to interaction behavior are deprecated
  INTERACTION BEHAVIOR = <string>TIED
END

BEGIN SEARCH OPTIONS
  GLOBAL SEARCH INCREMENT = <int>num_steps
  PARALLEL CONSISTENCY = OFF|ON
END
```

The following commands in the `BEGIN DASH OPTIONS` block are no longer available. The explicit contact solution tolerance is now set automatically and needs no input from a command line. The lofted sphere shape is now set in a `BEGIN SURFACE OPTIONS` command block. The lofted edges option was deemed not useful and this command has been removed from the code base.

```
BEGIN DASH OPTIONS
  GAP CONVERGENCE RATIO = <real>val
  LOFTED SPHERE REPRESENTATION = Cube|Icosahedron|Octahedron|Tetrahedron
  USE LOFTED SHELL EDGES = OFF|ON
END
```

References

[1] K.H. Brown, R.M. Summers, M.W. Glass, A.S. Gullerud, M.W. Heinstein, and R.E. Jones. ACME: Algorithms for contact in a multiphysics environment, API version, 1.0. Technical Report SAND2001-3318, Sandia National Laboratories, Albuquerque, NM, 2001. pdf.

[2] M.W. Heinstein and T.E. Voth. Contact enforcement for explicit transient dynamics, 2005.

[3] Sierra/SolidMechanics Team. Sierra/SolidMechanics VOTD Theory Manual. Technical Report In draft, Sandia National Laboratories, Albuquerque, NM, 2019.

9. OUTPUT

Sierra/SM produces a variety of output. This chapter discusses how to control the four major types of output:

- results output,
- history output,
- heartbeat output, and
- restart output.

The user is allowed to select a set of variables (internal, user-defined, or some combination thereof) for results output. If the user selects a nodal variable such as displacement for results output, the displacements for all the nodes in a model will be output to a results file. If the user selects an element variable such as stress for results output, the stress for all elements in the model that calculate this quantity (stress) will be output. The history output option lets the user select a specific set of information for output. For example, if the displacement at a particular node is critical, then the user may select only the displacement at that node as history output. The heartbeat output is similar to the history output except that the output is written to a text file instead of to a binary (exodusII [1]) file. The restart output is written so that any calculation can be halted at some arbitrary analysis time and then restarted at this time. The user has no control over what is written to the restart file. When a restart file is written, it must be a complete state description of the calculations at some given time. A restart file contains a great deal of information and is typically larger than a results file; thus the restart file output frequency must be chosen with care.

Section 9.1 describes the syntax for requesting output variables. Section 9.3 describes the results output. Included in the results output is a description of commands for user-defined output (Section 9.3.2). User-defined output lets the user post process analysis results as the code is running to produce a reduced set of output information. Section 9.4 describes the history output, Section 9.5 describes the heartbeat output, and Section 9.6 describes the restart output. All four types of output (results, history, heartbeat, and restart) can be synchronized for analyses with multiple regions. This scheduling functionality is discussed in Section 9.7. Section 9.8 describes the commands for performing output variable interpolation. Section 9.9 defines the output options for energy and rigid body global variables, which are defined at the region scope. Finally, in Section 9.10, there is a list of key variables. Unless otherwise noted, the command blocks and command lines discussed in Chapter 9 appear in the region scope.

9.1. Syntax for Requesting Variables

The basic method of requesting output is done by specifying the output type (nodal, element, or global), followed by the name of the variable. The original name will now appear in the output file, or the variable may be renamed using the `as` specifier.

```
var_type(element,nodal,global) <string>var_name
```

Variables may be accessed in the code either in whole or by component. Values at specific components or integration points of multi-component variables may be accessed via parenthesis syntax. Parenthesis syntax may be applied to results output, history output, element death, or any other command where variable names are specified. Values of single-component variables indexed in some other way may be accessed with underscore syntax, primarily applicable to rigid body fields as discussed in Section 9.1.4.

Parenthesis syntax is a variable name of the form:

```
<string>var_name(<index>component,<integer>integration_point)
```

For a variable named `var`, a variable name of the form `var(A,B)` asks for the `A` component of the variable at integration point `B`. If a variable is a vector, `x`, `y`, or `z` may be specified as the component. If a variable is 3x3 tensor, `xx`, `yy`, `zz`, `xy`, `xz`, `yz`, `yx`, `zx`, or `zy` may be specified as the component. Components for other types of variables may be requested through an integer index.

The characters `:` and `*` are wild cards if used for specifying either the component or the integration point. `var(:,B)` asks for all components of var at integration point B. `var(A,:)` asks for component A of var at all integration points. `var(:,:)` asks for all components of var at all integration points. `var` is shorthand for `var(:,:)`.

`var(A)` will behave slightly differently depending on the nature of the variable. If the variable has multiple components, then `Var(A)` is treated like `var(A,:)`. If a variable has one and only one component then it is assumed that A refers to the integration point number rather than the component number and `Var(A)` is treated like `var(1,A)`.

9.1.1. Example 1

Let stress be a tensor defined on a single integration point element and a displacement vector be defined at all model nodes. The following output variable specification:

```
element stress as str
nodal displacement as disp
```

asks for all the components of the stress tensor on elements and all components of the displacement vector on nodes. The code would write these variables to the output file:

```
str_xx
str_yy
str_zz
str_xy
str_xz
str_yz
disp_x
disp_y
disp_z
```

If only the `yy` component of stress is desired, either of the following could be used:

```
element stress(yy) as my_yy_str1
element stress(2) as my_yy_str2
```

If only the `z` component of displacement is desired either of the following could be specified:

```
nodal displacement(z) as my_z_disp1
nodal displacement(3) as my_z_disp2
```

Note, tensor index 2 corresponds to the `yy` component of the tensor and vector index 3 corresponds to the `z` component of the vector.

9.1.2. Example 2

Let `unrotated_stress` be a tensor defined at each of three integration points of an element (e.g., a shell element), and `eqps` a scalar material state variable also defined at each element integration point. To request all `unrotated_stress` tensor components and all `eqps` scalar data at all integration points the following could be specified:

```
element unrotated_stress as str
element eqps as eqps
```

Which would output the variables:

```
str_xx_1, str_yy_1, str_zz_1, str_xy_1, str_xz_1, str_yz_1
str_xx_2, str_yy_2, str_zz_2, str_xy_2, str_xz_2, str_yz_2
str_xx_3, str_yy_3, str_zz_3, str_xy_3, str_xz_3, str_yz_3
eqps_1, eqps_2, eqps_3
```

To request the `unrotated_stress` tensor and value of `eqps` at the second integration point, the following syntax can be used:

```
element unrotated_stress(:,2) as str_intg2
element eqps(2) as eqps_intg2
```

This would output the variables:

```
str_intg2_xx
str_intg2_yy
str_intg2_zz
str_intg2_xy
str_intg2_xz
str_intg2_yz
eqps_intg2
```

To request the `xy` component of `unrotated_stress` at all integration points any of the following could be used:

```
element  unrotated_stress(xy,*) as str_xy_all
element  unrotated_stress(xy,:) as str_xy_all
element  unrotated_stress(xy)   as str_xy_all
```

Any of the above would output:

```
str_xy_all_1
str_xy_all_2
str_xy_all_3
```

9.1.3. Other command blocks

The parenthesis syntax described above for results output can also be used in most other commands involving variable names. For example, to kill elements based on yy stress or z displacement the following could be specified:

```
begin element death
  criterion is element value of stress(yy) > 1000
  criterion is average nodal value of displacement(z) > 3.0
end
```

9.1.4. Rigid Body Variables

Rigid body variables are provided as separate components of the rigid body fields, which may be accessed by outputting the desired field component(s). To access the field component of rigid bodies, an underscore syntax is employed (the parenthesis syntax applies to multi-component fields and integration points). The underscore syntax takes the desired field component and appends it with an underscore and the desired rigid body name.

For example the following lines in a history output block (see Section 9.4)

```
global ax_rb1
global displz_rb4
```

would output to the history file the variables

```
ax_rb1
displz_rb4
```

which are the translational acceleration in the x-direction of rigid body `rb1` and the translational displacement in the z-direction of rigid body `rb4`, respectively (see Table 9-4).

9.2. General Output Database Commands

A number of commands that control the format of the output database are available in several types of output (results, restart, history, and/or heartbeat). These commands are described in detail below.

```
DATABASE NAME = <string>results_file_name
DATABASE TYPE = <string>database_type(exodusII)
OVERWRITE = <string>OFF|ON|TRUE|FALSE|YES|NO
  (ON|TRUE|YES)
TITLE <string>user_title
PROPERTY = <string>propName = <string>propValue
```

The first two command lines (`DATABASE NAME`, `DATABASE TYPE`) give pertinent information about the database file. The command line

```
DATABASE NAME = <string>results_file_name
```

gives the name of the results file with the string `results_file_name`. If the file is to appear in the current directory and is named `job.e`, this command line would appear as:

```
DATABASE NAME = job.e}
```

If the file is to be created in some other directory, the command line must include the path to that directory. If the directory does not exist, all necessary directories will be created automatically.

Two special, shorthand meta-characters may be used in the results file name specification in the input deck: `%P` represents the number of processors in use for the analysis, while `%B` represents the base name of the analysis input file. If the `%P` character is used, for example, a specified database name of `results-%P/job.e` for a job running on N processors would be expanded to `results-N/job.e`, and the actual results files would be of the form `results-N/job.e.N.i`, where $i = \{0000, 0001, \ldots, 1023\}$. If the `%B` character is used, for example, a results database name specified as `%B.e` in an input file named `my_analysis_run.i` and the , would yield a results output file named `my_analysis_run.e`. If no results database file name is given, the default file name is `%B.e`, the base name of the input file with the extension `.e`.

If the results file does not use the Exodus II format [1], the format for the results file must be specified using the command line:

```
DATABASE TYPE = <string>database_type(exodusII)
```

The Exodus II database is supported in Presto and Adagio. Exodus II is currently the only supported format, but other options may be added in the future.

The `OVERWRITE` command line can be used to prevent the overwriting of existing results files.

```
OVERWRITE = <string>OFF|ON|TRUE|FALSE|YES|NO (ON|TRUE|YES)
```

The `OVERWRITE` command line allows only a single value. The options `FALSE`, `NO`, or `OFF` will cause the code to terminate before existing results files can be overwritten, while `TRUE`, `YES`, or

ON will cause existing results files to be overwritten by new files (the default status). Suppose, for example, that an existing results is named file named job21.e, and suppose that an input file with a RESULTS OUTPUT command block containing

```
OVERWRITE = ON
DATABASE NAME = job21.e
```

is run. In this case the existing file job21.e will be overwritten.

Whether or not files are overwritten is also impacted by the use of the automatic read and write option for restart files described in Section 9.6.1.2. If using the automatic read and write option for restart files, the results files, like the restart files, are automatically managed. The automatic read and write option in restart adds extensions to file names and prevents the overwriting of any existing restart or results files. For the case of a user-controlled read and write of restart files (Section 9.6.1.3) or of no restart, however, the OVERWRITE command line is useful for preventing the overwriting of results files.

A title may be added to the results file with the TITLE command line. The specified user_title will be written to the 'title' field of the results file. Some of the programs that process the results file (such as various SEACAS programs [2]) can read and display this information.

The PROPERTY command can change some of the fundamental properties of the output database. For example, maximum variable name length, which, in Exodus II format, has a limit of 100 characters. Note, changing this properties is not recommended unless by direction of the development team.

9.3. Results Output

The results output capability allows the user to select some set of variables to be written to a file at various time intervals. All values for each selected variable will be written to the results file. The interval at which information is written can be changed throughout the analysis time. The name of the results file is set in the `RESULTS OUTPUT` command block.

9.3.1. Exodus Results Output File

```
BEGIN RESULTS OUTPUT <string>results_name
  #
  # Database Commands
  DATABASE NAME = <string>results_file_name
  DATABASE TYPE = <string>database_type(exodusII)
  OVERWRITE = <string>OFF|ON|TRUE|FALSE|YES|NO
    (ON|TRUE|YES)
  TITLE <string>user_title
  PROPERTY <string>propName = <string>propVal
  TIMESERIES NAME = <string>timeseries_file_name

  NODE VARIABLES = <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
    | NODAL VARIABLES = <string>variable_name
        [AS <string>dbase_variable_name] ...
        <string>variable_name [AS
        <string>dbase_variable_name]

  NODESET VARIABLES = <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
    | NODESET VARIABLES = <string>variable_name
        [AS <string>dbase_variable_name]
        INCLUDE|ON|EXCLUDE <string list>nodelist_names
        ... <string>variable_name
        [AS <string>dbase_variable_name] INCLUDE|ON|EXCLUDE
        <string list>nodelist_names

  SURFACE VARIABLES = <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
    | SURFACE VARIABLES = <string>variable_name
        [AS <string>dbase_variable_name]
```

```
      INCLUDE|ON|EXCLUDE <string list>surface_names
      ... <string>variable_name
      [AS <string>dbase_variable_name] INCLUDE|ON|EXCLUDE
      <string list>surface_names

FACE VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS
  <string>dbase_variable_name]
  | FACE VARIABLES = <string>variable_name
      [AS <string>dbase_variable_name]
      INCLUDE|ON|EXCLUDE <string list>surface_names
      ...  <string>variable_name
      [AS <string>dbase_variable_name] INCLUDE|ON|EXCLUDE
      <string list>surface_names

ELEMENT VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS
  <string>dbase_variable_name]
  | ELEMENT VARIABLES = <string>variable_name
      [AS <string>dbase_variable_name]
      INCLUDE|ON|EXCLUDE <string list>block_names
      ... <string>variable_name
      [AS dbase_variable_name] INCLUDE|ON|EXCLUDE
      <string list>block_names

GLOBAL VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS
  <string>dbase_variable_name]
OUTPUT MESH = REFINED|EXPOSED SURFACE (REFINED)
COMPONENT SEPARATOR CHARACTER = <string>character|NONE
INCLUDE = <string>list_of_included_entities
EXCLUDE = <string>list_of_excluded_entities
START TIME = <real>output_start_time
TIMESTEP ADJUSTMENT INTERVAL = <integer>steps
AT TIME <real>time_in INCREMENT =
  <real>time_increment_dt
AT STEP <integer>step_begin INCREMENT =
  <integer>step_increment
ADDITIONAL TIMES = <real>output_time1
  <real>output_time2 ...
ADDITIONAL STEPS = <integer>output_step1
  <integer>output_step2 ...
TERMINATION TIME = <real>termination_time_value
SYNCHRONIZE_OUTPUT
```

Explicit

```
  USE OUTPUT SCHEDULER <string>scheduler name
  OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
    SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
    SIGKILL|SIGILL|SIGSEGV
END [RESULTS OUTPUT <string>results_name]
```

See Section 9.2 for details on the general database commands such as `DATABASE NAME`, `DATABASE TYPE`, `OVERWRITE`, etc. The command `TIMESERIES NAME` optionally emits a time-series file containing the list of root output database filenames as they are written. This is useful for simulations on high numbers of processors with many mesh modifications that output a large number of files.

A results file, the results to be included in this file, and the frequency at which results are written may be specified using a `RESULTS OUTPUT` command block. The command block appears inside the region scope. More than one results file can be specified for an analysis; thus, for each results file, there will be one `RESULTS OUTPUT` command block. The command block begins with:

```
BEGIN RESULTS OUTPUT <string>results_name
```

and is terminated with:

```
END [RESULTS OUTPUT <string>results_name]
```

where `results_name` is a user-selected name for the command block. Nested within the `RESULTS OUTPUT` command block is a set of command lines, as shown in the block summary given above.

The other command lines that appear in the `RESULTS OUTPUT` command block determine the type and frequency of output information. Descriptions of these command lines follow in Section 9.3.1.1 through Section 9.3.1.12.

9.3.1.1. Output Nodal Variables

```
NODE VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS
  <string>dbase_variable_name]
  | NODAL VARIABLES = <string>variable_name
      [AS <string>dbase_variable_name] ...
      <string>variable_name [AS
      <string>dbase_variable_name]
```

Any nodal variable in Sierra/SM can be selected for output in the results file by using a command line in one of the two forms shown above. The only difference between the two forms is the use of `NODE VARIABLES` or `NODAL VARIABLES`. The string `variable_name` is the name of the nodal variable to output. The string `variable_name` can be either a variable listed in Section 9.10 or a user-defined variable (see Section 9.3.2 and Section 10.2.4). Any nodal variable requested for output is output for all nodes.

An alias can be specified for any of the nodal variables by using the AS specification. For example, to output the external forces in Sierra/SM, which are defined as force_external, with the alias f_ext, the command line

```
NODE VARIABLES = force_external AS f_ext
```

would be used. In this example, the external force is a vector quantity. For a vector quantity at a node, suffixes are appended to the variable name (or alias name) to denote each vector component. The results database would have three variable names associated with the external force: f_ext_x, f_ext_y, and f_ext_z. The component separator—an underscore in this example—may be changed using the COMPONENT SEPARATOR CHARACTER command line (see Section 9.3.1.7).

The NODE VARIABLES command line can be used an arbitrary number of times within a RESULTS OUTPUT command block. It is also possible to specify more than one nodal variable for output on a command line, as indicated by the ellipsis in the command line format. In the next example, two nodal variables are specified for output. Internal forces are defined as force_internal.

```
NODE VARIABLES = force_external force_internal
```

Aliases can be optionally specified for each of the variables in a single command line. For example, to output the alias f_ext for external forces and to output the alias f_int for internal forces, the command line

```
NODE VARIABLES = force_external AS f_ext
                 force_internal AS f_int
```

9.3.1.2. Output Node Set or Surface Variables

```
NODESET VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS
  <string>dbase_variable_name]
  | NODESET VARIABLES = <string>variable_name
      [AS <string>dbase_variable_name]
      INCLUDE|ON|EXCLUDE <string list>nodelist_names
      ... <string>variable_name
      [AS <string>dbase_variable_name] INCLUDE|ON|EXCLUDE
      <string list>nodelist_names
SURFACE VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS
  <string>dbase_variable_name]
  | SURFACE VARIABLES = <string>variable_name
      [AS <string>dbase_variable_name]
      INCLUDE|ON|EXCLUDE <string list>surface_names
      ... <string>variable_name
```

```
[AS <string>dbase_variable_name] INCLUDE|ON|EXCLUDE
<string list>surface_names
```

A nodal variable may be defined on a subset of the total set of nodes defining a model or a face variable may be defined on a subset of surfaces. A nodal variable defined only on a subset of nodes is called a node set variable. A face variable defined only on a subset of surfaces is called a surface variable.

The first form of this command line is as follows:

```
NODESET|SURFACE VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS <string>dbase_variable_name]
```

Here, the string `variable_name` is a variable associated with one or more node sets or surfaces. In this form, the variable is output for all node sets or surfaces associated with that variable.

An alias can be specified in the results file for any of the variables by using the `AS` option. Suppose, for example, the node set variable `force_nsetype` is to be output with the name `fnsetype` in the results file. The corresponding command line is:

```
NODESET VARIABLES = force_nsetype AS fnsetype
```

The `NODESET|SURFACE VARIABLES` command line can be used an arbitrary number of times within a `RESULTS OUTPUT` command block. It is also possible to specify more than one node set variable for output on a command line, as indicated by the ellipsis in the command line format. In the next example, two node set variables are specified for output. Here, the second node set variable is defined as `force_nsetype2`.

```
NODESET VARIABLES = force_nsetype force_nsetype2
```

Aliases can be optionally specified for each of the variables in a single command line. For example, to output the alias `fnsetype` for node set variable `force_nsetype` and also to output the alias `fnsetype2` for node set variable `force_nsetype2`, the command line

```
NODESET VARIABLES = force_nsetype AS fnsetype
                    force_nsetype2 AS fnsetype2
```

would be used.

The second form of the command line is as follows:

```
NODESET VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name]
  INCLUDE|ON|EXCLUDE <string list>nodelist_names
  ... <string>variable_name [AS <string>dbase_variable_name]
  INCLUDE|ON|EXCLUDE <string list>nodelist_names
SURFACE VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name]
```

```
INCLUDE|ON|EXCLUDE <string list>surface_names
... <string>variable_name [AS <string>dbase_variable_name]
INCLUDE|ON|EXCLUDE <string list>surface_names
```

This form of the NODESET|SURFACE VARIABLES command line is similar to the first, except that the user can control which node sets are used for output. The user can include a specific list of node sets or surfaces for output by using the INCLUDE option or the ON option. (The keyword INCLUDE is synonymous with the keyword ON.) Alternatively, the user can exclude a specific list of node sets for output by using the EXCLUDE option.

Suppose that the node set variable force_nsetype from the above example has been defined for nodelist_10, nodelist_11, nodelist_20, and nodelist_21. If we only want to output the node set variable for node sets nodelist_10, nodelist_11, and nodelist_21, then we could specify the NODESET command line as follows:

```
NODESET VARIABLES = force_nsetype AS fnsetype
   INCLUDE nodelist_10, nodelist_11, nodelist_21
```

(In the above command line, the keyword ON could be substituted for INCLUDE.) Alternatively, we could use the command line:

```
NODESET VARIABLES = force_nsetype AS fnsetype
   EXCLUDE nodelist_20
```

In the above command lines, an alias for a node set can be substituted for a node set identifier. For example, if center_case is an alias for nodelist_10, then the string center_case could be substituted for nodelist_10 in the above command lines. Because a node set identifier is a mesh entity, the alias for the node set identifier would be defined via an ALIAS command line in a FINITE ELEMENT MODEL command block.

The list of identifiers uses a comma to separate one node set identifier from the next node set identifier.

9.3.1.3. Output Face Variables

```
FACE VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS
  <string>dbase_variable_name]
  | FACE VARIABLES = <string>variable_name
      [AS <string>dbase_variable_name]
      INCLUDE|ON|EXCLUDE <string list>surface_names
      ...  <string>variable_name
      [AS <string>dbase_variable_name] INCLUDE|ON|EXCLUDE
      <string list>surface_names
```

A variable may be defined on some set of faces that constitute a surface. A variable defined on a set of faces is called a face variable. The FACE VARIABLES command line allows the user to

specify a face variable for output to the results file. Two forms of the FACE VARIABLES command line exist and either form will output a face variable. The first form of the command line is as follows:

```
FACE VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS
  <string>dbase_variable_name]
```

Here, the string variable_name is a face variable associated with one or more surfaces. In this form, the face variable is output for all surfaces associated with that face variable.

An alias can be specified in the results file for any face variable by using the AS option. For example, to output a face variable pressure_face with the name pressuref in the results file, the command line

```
FACE VARIABLES = pressure_face AS pressuref
```

would be used. The FACE VARIABLES command line can be used an arbitrary number of times within a RESULTS OUTPUT command block. It is also possible to specify more than one face variable for output on a command line, as indicated by the ellipsis in the command line format. In the next example, two face variables are specified for output. Here, the second face variable is defined as scalar_face2:

```
FACE VARIABLES = pressure_face scalar_face2
```

Aliases can be optionally specified for each of the variables in a single command line. For example, to output the alias pressuref for face variable pressure_face and also to output the alias scalarf2 for face variable scalar_face2, the command line

```
FACE VARIABLES = pressure_face AS pressuref
                 scalar_face2 AS scalarf2
```

The second form of the command line is as follows:

```
FACE VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name]
  INCLUDE|ON|EXCLUDE <string list>surface_names
  ...  <string>variable_name
  [AS <string>dbase_variable_name] INCLUDE|ON|EXCLUDE
  <string list>surface_names
```

This form of the FACE VARIABLES command line is similar to the first, except that the user can control which surfaces are used for output. The user can include a specific list of surfaces for output by using the INCLUDE option or the ON option. (The keyword INCLUDE is synonymous with the keyword ON.) Alternatively, the user can exclude a specific list of surfaces for output by using the EXCLUDE option.

Suppose that the face variable pressure_face from the above example has been defined for surface_10, surface_11, surface_20, and surface_21. If we only want to output the face variable for surface_10, surface_11, and surface_21, then we could specify the FACE VARIABLES command line as follows:

```
FACE VARIABLES = pressure_face AS pressuref
   INCLUDE surface_10, surface_11,
   surface_21
```

(In the above command line, the keyword ON could be substituted for INCLUDE.) Alternatively, we could use the command line:

```
FACE VARIABLES = pressure_face AS pressuref
   EXCLUDE surface_20
```

In the above command lines, an alias for a surface can be substituted for a surface identifier. For example, if center_case is an alias for surface_10, then the string center_case could be substituted for surface_10 in the above command lines. Because a surface identifier is a mesh entity, the alias for the surface identifier would be defined via an ALIAS command line in a FINITE ELEMENT MODEL command block.

The list of identifiers uses a comma to separate one surface identifier from the next surface identifier.

9.3.1.4. Output Element Variables

```
ELEMENT VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS
  <string>dbase_variable_name]
  | ELEMENT VARIABLES = <string>variable_name
      [AS <string>dbase_variable_name]
      INCLUDE|ON|EXCLUDE <string list>block_names
      ... <string>variable_name
      [AS dbase_variable_name] INCLUDE|ON|EXCLUDE
      <string list>block_names
```

Any element variable in Sierra/SM can be selected for output in the results file by using the ELEMENT VARIABLES command line. Two forms of the ELEMENT VARIABLES command line exist and either form will output an element variable. The first form of these command lines are as follows:

```
ELEMENT VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS
  <string>dbase_variable_name]
```

Here, the string variable_name is the name of the element variable to output. The string variable_name can be a variable listed in Section 9.10 or a user-defined variable (see Section 9.3.2 and Section 10.2.4).

In the first form of the ELEMENT VARIABLES command line, the element variable is output for all element blocks that have the element variable as a defined variable. For example, all the solid elements have stress as a defined variable. If output of the element variable stress were requested on a mesh consisting of hexahedral and tetrahedral elements, then stress would be output for all element blocks consisting of hexahedral and tetrahedral elements.

An alias can be specified for any of the element variables by using the AS specification. For example, to output the stress in Sierra/SM, which is defined as stress, with the alias str, the command line

```
ELEMENT VARIABLES = stress AS str
```

would be used. In this example, stress is a symmetric tensor quantity. For a symmetric tensor quantity, suffixes are appended to the variable name (or alias name) to denote each symmetric tensor component. The results database would have six variable names associated with the stress: stress_xx, stress_yy, stress_zz, stress_xy, stress_yz, and stress_zx. The tensor component separator, an underscore in this example, may be changed by using the COMPONENT SEPARATOR CHARACTER command line (see Section 9.3.1.7).

The ELEMENT VARIABLES command line can be used an arbitrary number of times within a RESULTS OUTPUT command block. It is also possible to specify more than one element variable for output on a command line, as indicated by the ellipsis in the command line format. In the following example, two element variables are specified for output. Here, the second element variable is defined as left_stretch.

```
ELEMENT VARIABLES = stress left_stretch
```

Aliases can be optionally specified for each of the variables in a single command line. For example, to output the alias str for element variable stress and also to output the alias strch for face variable left_stretch, the command line

```
ELEMENT VARIABLES = stress AS str
                    left_stretch AS strch
```

The second form of the command line is as follows:

```
ELEMENT VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name]
  INCLUDE|ON|EXCLUDE <string list>block_names
  ... <string>variable_name
  [AS dbase_variable_name] INCLUDE|ON|EXCLUDE
  <string list>block_names
```

This form of the ELEMENT VARIABLES command line is similar to the first, except that the user can control which element blocks are used for output. The user can include a specific list of element blocks for output by using the INCLUDE option or the ON option. (The keyword INCLUDE

is synonymous with the keyword ON.) Alternatively, the user can exclude a specific list of element blocks for output by using the EXCLUDE option.

Suppose the element variable stress from the above example exists for element blocks block_10, block_11, block_20, and block_21. If we only want to output the element variable for block_10, block_11, and block_21, then we could specify the ELEMENT VARIABLES command line as follows:

```
ELEMENT VARIABLES = stress AS str
INCLUDE block_10, block_11,
   block_21
```

(In the above command line, the keyword ON could be substituted for INCLUDE.) Alternatively, we could use the command line:

```
ELEMENT VARIABLES = stress AS str
EXCLUDE block_20
```

In the above command lines, an alias for an element block can be substituted for an element block identifier. For example, if center_case is an alias for block_10, then the string center_case could be substituted for block_10 in the above command lines. Because an element block identifier is a mesh entity, the alias for the element block identifier would be defined via an ALIAS command line in a FINITE ELEMENT MODEL command block. The list of identifiers uses a comma to separate one element block identifier from the next element block identifier.

For multi-integration point elements, quantities from the integration points are appended with a numerical index indicating the integration point. A suffix ranging from 1 to the number of integration points is attached to the quantity to indicate the corresponding integration point. The suffix is padded with leading zeros. If the number of integration points is less than 10, the suffix has the form _*i*, where *i* ranges from 1 to the number of integration points. If the number of integration points is greater than or equal to 10 and less than 100, the sequence of suffixes takes the form _01, _02, _03, and so forth. Finally, if the number of integration points is greater than or equal to 100, the sequence of suffixes takes the form _001, _002, _003, and so forth. As an example, if the von Mises stress is requested for a shell element with 15 integration points, then the quantities von_mises_01, von_mises_02, ..., von_mises_15 are output for the shell element.

In the above discussion concerning the format for output at multiple integration points, the underscore character preceding the integration point number can be replaced by another delimiter or the underscore character can be eliminated by use of the COMPONENT SEPARATOR CHARACTER command line (see Section 9.3.1.7).

Shell tensor quantities transform_shell_stress, transform_shell_strain and transform_shell_rate_of_deformation may be transformed to a user specified shell local co-rotational coordinate system (an in-plane coordinate system that rotates with the shell element) for output using the ORIENTATION shell section command. If no orientation is specified, these in-plane stresses and strains are output in the default orientation. See Section 6.2.5 for more details.

9.3.1.5. Subsetting of Output Mesh

A specified subset of the entities (element blocks, nodesets, surfaces) in the mesh can be output to the results database using the `INCLUDE` or `EXCLUDE` commands. The syntax is:

```
INCLUDE = <string>list_of_included_entities
EXCLUDE = <string>list_of_excluded_entities
```

Either command can appear multiple times within the results output block, but a specific entity cannot be specified in both. If the `INCLUDE` command is specified, the results database will only contain the listed entities of that type; if the `EXCLUDE` command is specified, the results database will contain all entities of the type listed except for the listed entities. If the model has surfaces or nodesets, only the portion of the surfaces or nodesets on the selected element blocks will be output.

9.3.1.6. Output Mesh Selection

```
OUTPUT MESH = REFINED|EXPOSED SURFACE (REFINED)
```

The `OUTPUT MESH` command provides a way to reduce the amount of data written to the results database. There are two options that can be selected:

`REFINED` Outputs the full 3D model in the current mesh configuration.

`EXPOSED SURFACE` Only output the element faces that make up the 'skin' of the finite element model; no internal nodes or elements will be written to the results database. The element results variables will be applied to the skin faces. If the mesh is visualized without any cutting planes, the display should look the same as if the original full mesh were visualized; however, the amount of data written to the output file can be much less than is needed if the full mesh were output.

9.3.1.7. Component Separator Character

```
COMPONENT SEPARATOR CHARACTER = <string>character|NONE
```

The component separator character is used to separate an output-variable base name from any suffixes. For example, the variable `stress` can have the suffixes `xx`, `yy`, etc. By default, the base name is separated from the suffixes with an underscore character so that we have `stress_xx`, `stress_yy`, etc. in the results output file.

The underscore can be replaced as the default separator by using the above command line. For example, to use a period (`.`) as the separator, the command line

```
COMPONENT SEPARATOR CHARACTER = .
```

would be used. For the `stress` output example, stress components would then appear in the results output file as `stress.xx`, `stress.yy`, etc. For shell element blocks, an integration point

suffix preceded, by default, with an underscore is also present; the above command line also resets this separator character to period (.). In the stress output example, in this case the results file would contain stress.xx.01, stress.xx.02, etc., for the shell elements. The separator can be eliminated with an empty string or NONE.

9.3.1.8. Output Global Variables

```
GLOBAL VARIABLES = <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS
  <string>dbase_variable_name]
```

Any global variable in Sierra/SM can be selected for output in the results file by using the GLOBAL VARIABLES command line. The string variable_name is the name of the global variable. The string variable_name can be either a variable listed in Section 9.10 or a user-defined variable (see Section 9.3.2 and Section 10.2.4).

Kinetic, external, hourglass, internal energies and momentum[1] can be requested as the sum over the entire mesh. For example, the total kinetic energy can be output as:

```
GLOBAL VARIABLES = kinetic_energy as ke
```

Block internal, strain, and hourglass energies can be output on a block-by-block basis by using a user output block such as the following:

```
BEGIN USER OUTPUT
  BLOCK = BLOCK_100
  COMPUTE GLOBAL BLOCK_100_IE AS SUM OF ELEMENT INTERNAL_ENERGY
  COMPUTE GLOBAL BLOCK_100_SE AS SUM OF ELEMENT STRAIN_ENERGY
  COMPUTE GLOBAL BLOCK_100_HGE AS SUM OF ELEMENT HOURGLASS_ENERGY
END
```

The sum of the block strain energy and the block hourglass block energy should equal the block internal energy.

With the AS keyword, the variable and an alias for this variable may be optionally specified in the results file. For example, to output the time steps in Sierra/SM (timestep) with the alias tstep, the command line

```
GLOBAL VARIABLES = timestep AS tstep
```

would be used. The GLOBAL VARIABLES command line can be used an arbitrary number of times within a RESULTS OUTPUT command block. It is also possible to specify more than one global variable for output on a command line. If output of the CPU time (cpu_time) is also desired for output with the alias cpu, the command line

[1] See the Sierra/SM 5.0 Theory Manual [3] for details on momentum and energy balance. Internal energy includes both elastic and dissipated energy and contact energy is included in the external energy. Significant error in the energy balance (see Section 3.1.5) often indicates the problem is under-resolved.

```
GLOBAL VARIABLES = timestep as tstep cpu_time as cpu
```

should be used.

9.3.1.9. Specifying Time Steps for Output

```
START TIME = <real>output_start_time
TERMINATION TIME = <real>output_termination_time
AT TIME <real>time_begin INCREMENT = <real>time_increment_dt
AT STEP <integer>step_begin INCREMENT = <integer>step_increment
ADDITIONAL TIMES = <real>output_time1 <real>output_time2 ...
ADDITIONAL STEPS = <integer>output_step1 <integer>output_step2 ...
TIMESTEP ADJUSTMENT INTERVAL = <integer>steps
```

Explicit

Several options are available to define output; output steps will be included in a results, history, or heartbeat output file. Note, these commands control only what time steps will be output, not what time steps will be computed. Thus requesting output at some time *N* will trigger an output of the next computed time step that is equal to or greater than time *N*. The analysis time stepping setup controls what time steps are computed, see Sections 3.1 and 4.10.

Using the `START TIME` command line, results are output to file beginning at time `output_start_time`. This command is optional, but if present, no results will be written before the `output_start_time`. If other commands set times for results output (e.g., `AT TIME`, `ADDITIONAL TIMES`) that are less than `output_start_time`, those times will be ignored, and results will not be written at those times.

If the `TERMINATION TIME` command is used, results will not be written to the results file after time `output_termination_time`. If other commands set times for results such (`AT TIME`, `ADDITIONAL TIMES`) that are greater than `termination_time_value`, those times will be ignored, and results will not be written at those times.

The `AT TIME` command will trigger output starting at the time specified by `time_begin` as long as this time is greater than `output_start_time` specified in a `START TIME` command, and then at every subsequent time increment `time_increment_dt` thereafter. Output will also occur at the end of the analysis even if it does not fall exactly at one of the specified intervals.

The `AT STEP` command will trigger output starting at the step specified by `step_begin` and then at every subsequent step increment `step_increment` thereafter. Output will also occur at the end of the analysis even if it does not fall exactly at a specified interval.

If both the `AT TIME` and the `AT STEP` commands are used time steps that match either condition will be output. For example the following would output time steps `0.0001`, `0.0002`, `0.0003`, etc., as well as every hundredth step.

```
at time 0.0, increment = 0.0001
at step 0, increment = 100
```

If multiple `AT TIME` or `AT STEP` commands are used the commands define different output frequencies at different time intervals. For example:

```
at time 0.00, increment = 0.25
at time 1.00, increment = 0.01
at time 1.05, increment = 0.25
```

From time `0.00` to `1.00` the steps will output at time interval `0.25`, at time `1.00` the output interval will switch to `0.01` and then the output interval witch switch back to `0.25` at time `1.05`. Specifically this will output the following time steps `0.00, 0.25, 0.50, 0.75, 1.00, 1.01, 1.02, 1.03, 1.04, 1.05, 1.30, 1.55, etc.`. Use of multiple `AT TIME` or `AT STEP` commands is often used to obtain high frequency of output at some critical analysis time while still limiting the overall size of the results file.

The `ADDITIONAL TIMES` command line can be used to specify a number of additional output times. If `output_start_time` is specified in a `START TIME` command, then no additional times that are less than `output_start_time` will output. Output will also occur at the end of the analysis even if it does not fall exactly at a specified interval.

The `ADDITIONAL STEPS` command line can be used to specify output at a number of other analysis steps. If `output_start_time` is specified in a `START TIME` command, then no additional steps at times less than `output_start_time` will output. Output will also occur at the end of the analysis even if it does not fall exactly at a specified interval.

The `TIMESTEP ADJUSTMENT INTERVAL` command line specifies that the output will be at exactly the times specified. To hit the output times exactly in an explicit, transient dynamics code, it is necessary to adjust the time step as the time approaches an output time. The integer value `steps` in the `TIMESTEP ADJUSTMENT INTERVAL` command line specifies the number of time steps to look ahead to adjust the time step. This command is only functional in explicit analysis. If this command line does not appear, results are output at times closest to the specified output times.



Examples The following examples demonstrate how time step output works for many cases. For this example assume the analysis time stepping is setup to compute the following steps:

step	1	2	3	4	5	6	7	8	9	10	11
time	0.0	0.5	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0

1.
```
at step 0, increment = 1
```

This will output all 11 computed time steps.

2.
```
at step 1, increment = 4
additional steps = 6 7
```

This will output every third time step, steps 6 and 7, and the final time step, i.e., steps 1, 5, 6, 7, 9, and 11.

3.
```
additional times 1.6
```

This will output at the next computed time t after the specified time $t_o = 1.6$, so output will be written only at step 5, at time $t = 2.0 > t_o$.

4. `at time 0.0, increment = 1.3`

This will output every 1.3 seconds of analysis times, i.e., times 0.0, 1.3, 2.6, 3.9, and 5.2. For a specified output time t_o not coincident with actual simulation time steps $\{t_s\}$, the next simulation time step $t \in \{t_s\}$, $t \geq t_o$ will be output. The actual output steps will then be as follows:

step	1	4	7	9
time	0.0	1.5	3.0	4.0

5.
```
at step 0, increment = 1
start time = 2.0
end time = 4.0
```

This will output at every step between the specified start and end times, i.e., at steps 5, 6, 7, 8, and 9.

9.3.1.10. Synchronize Output

```
SYNCHRONIZE OUTPUT
```

In an analysis with multiple regions, it is sometimes desirable to synchronize the output of results data between the regions. This can be done by adding the `SYNCHRONIZE OUTPUT` command line to the results output block. If a results block has this set, then it will write output whenever a previous region writes output. The ordering of regions is based on the order in the input file, algorithmic considerations, or by solution control specifications.

Although the `USE OUTPUT SCHEDULER` command line can also synchronize output between regions, the `SYNCHRONIZE OUTPUT` command line will synchronize the output with regions where the output frequency is not under the direct control of the Sierra IO system. Examples of this are typically coupled applications where one or more of the codes are not Sierra-based applications such as Alegra and CTH. A results block with `SYNCHRONIZE OUTPUT` specified will also synchronize its output with the output of the external code.

The `SYNCHRONIZE OUTPUT` command can be used with other output scheduling commands such as time-based or step-based output specifications.

9.3.1.11. Use Output Scheduler

```
USE OUTPUT SCHEDULER <string>scheduler_name
```

In an analysis with multiple regions, it can be difficult to synchronize output such as results files. To help synchronize output for analyses with multiple regions, an `OUTPUT SCHEDULER` command block may be specified at the SIERRA scope. The scheduler can then be referenced in the `RESULTS OUTPUT` command block via the `USE OUTPUT SCHEDULER` command line. The string `scheduler_name` must match a name used in an `OUTPUT SCHEDULER` command block.

See Section 9.7 for a description of using this command block and the `USE OUTPUT SCHEDULER` command line.

9.3.1.12. Write Results If System Error Encountered

```
OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
  SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
  SIGKILL|SIGILL|SIGSEGV
```

The `OUTPUT ON SIGNAL` command line is used to initiate the writing of a results file when the system encounters a system error. Only one error type in the list of error types should be entered for this command line. These system errors cause the code to terminate before the code can add any current results output (results output past the last results output time step) to the results output file. If the code encounters the specified error type during execution, a results file will be written before execution is terminated. This command line can also be used to force the writing of a results file at some point during execution of the code. Suppose the command line

```
OUTPUT ON SIGNAL = SIGUSR2
```

is included in the input file. While the code is running, a user can execute (from the keyboard) the system command line

`kill -s SIGUSR2` *pid*

to terminate execution and force the writing of a results file. In the above system command line, *pid* is the process identifier, an integer. The `OUTPUT ON SIGNAL` command line is primarily a debugging tool for code developers.

9.3.2. User-Defined Output

```
BEGIN FILTER <string>filter_name
  ACOEFF = <real_list>a_coeff
  BCOEFF = <real_list>b_coeff
  INTERPOLATION TIME STEP = <real>ts
END [FILTER]

BEGIN USER OUTPUT
  # mesh-entity set commands
  NODE SET|NODESET = <string list>nodelist_names
  SURFACE = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
  REMOVE SURFACE = <string list> surface_names
  REMOVE BLOCK = <string list>block_names
  #
  # compute result commands
```

```
PROCESS ELEMENTS = ACTIVE_ONLY|INACTIVE_ONLY|ALL(ACTIVE_ONLY)
COMPUTE GLOBAL <string>result_var_name
  AS SUM|MAX|MIN|MAX ABSOLUTE VALUE|L2NORM|VARIANCE
  OF NODAL|ELEMENT|FACE <string>source_var_name
COMPUTE GLOBAL <string>result_var_name
  AS AVERAGE
  OF NODAL|ELEMENT|FACE <string>source_var_name
  [WEIGHTED BY <string>weight_var_name]
COMPUTE GLOBAL|NODAL|ELEMENT <string>result_var_name
  AS NONLOCAL AVERAGE
  OF NODAL|ELEMENT <string>source_var_name
  OVER DOMAIN DEFINED BY RADIUS <real>r
  [AND POINT <real>x <real>y <real>z]
COMPUTE GLOBAL <string>result_var_name
  AS VOLUME INTEGRAL|SURFACE INTEGRAL
  OF NODAL|ELEMENT <string>source_var_name
COMPUTE GLOBAL <string>result_var_name
  AS CLOSEST DISTANCE TO <string> mesh_entity_name SEARCH KDTREE_NODE_FACE|N
COMPUTE GLOBAL <string>result_var_name
  AS FURTHEST DISTANCE TO <string> mesh_entity_name
COMPUTE GLOBAL <string>result_var_name
  AS MAX NORMAL|OPPOSING GAP BETWEEN <string> mesh_entity_name
COMPUTE GLOBAL <string>result_var_name
  AS SURFACE INTEGRAL OF ELEMENT ELEMENT_LOCAL_FORCE
  IN DIRECTION <real>dx <real>dy <real>dz
COMPUTE GLOBAL <string>result_var_name AS
  INTERNAL REACTION [VECTOR|SHEAR|NORMAL(VECTOR)]
  [AT POINT <real>px <real>py <real>pz]
  [IN DIRECTION <real>dx <real>dy <real>dz]
COMPUTE GLOBAL <string>result_var_name AS INTERPOLATION
  OF NODAL <string>source_var_name
  AT POINT <real>x <real>y <real>z

COMPUTE GLOBAL|NODAL|ELEMENT|FACE <string>result_var_name
  AS FUNCTION <string>function_name
COMPUTE GLOBAL <string>result_var_name
  FROM EXPRESSION "<string>expression"
COMPUTE GLOBAL <string>result_var_name
  AS INTEGRAL OVER TIME|LOG|EXP|MAGNITUDE OF GLOBAL
  <string>source_var_name

COMPUTE NODAL <string>result_var_name AS
  MAX OVER TIME|MIN OVER TIME|ABSOLUTE VALUE MAX OVER
  TIME|AVERAGE OVER TIME|STANDARD DEVIATION OVER TIME|
  LOG|EXP OF NODAL <string>source_var_name
COMPUTE NODAL <string>result_var_name AS
  END STEP VALUE OF NODAL VELOCITY
```

```
COMPUTE ELEMENT <string>result_var_name AS
  MAX OVER TIME|MIN OVER TIME|ABSOLUTE VALUE MAX OVER TIME|
  AVERAGE OVER TIME|STANDARD DEVIATION OVER TIME
  OF ELEMENT <string>source_var_name
COMPUTE ELEMENT <string>result_var_name AS
  MAX|MIN|VOLUME WEIGHTED AVERAGE|
   UNWEIGHTED AVERAGE|AVERAGE|LOG|EXP OF ELEMENT <string>source_var_name
COMPUTE ELEMENT|FACE <string>result_var_name AS
  MAX|MIN OF NODAL <string>source_var_name
COMPUTE ELEMENT|FACE <string>result_var_name AS
  AVERAGE OF NODAL <string>source_var_name
  [WEIGHTED BY <string>weight_var_name]


#
# user subroutine commands
NODE SET SUBROUTINE = <string>subroutine_name |
  SURFACE SUBROUTINE = <string>subroutine_name |
  ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
#
# copy command
COPY ELEMENT VARIABLE <string>ev_name
  TO NODAL VARIABLE <string>nv_name
COPY MAX|SUM|WEIGHTED SUM ELEMENT VARIABLE <string>ev_name
  TO NODAL VARIABLE <string>nv_name
#
# extrapolate command
EXTRAPOLATE ELEMENT VARIABLE <string>ev_name to NODAL VARIABLE
  <string>nv_name
EXTRAPOLATE PROJECTION TYPE = <string>projection_type
  CONSISTENT|ROW_SUM_LUMPED|PROPORTIONAL_LUMPED (CONSISTENT)
LINEAR SOLVER = <string>solver_name (DEFAULT_FETI_SOLVER)
#
# variable transformation command
TRANSFORM NODAL|ELEMENT VARIABLE <string>variable_name
  TO COORDINATE SYSTEM <string>coord_sys_name
  AS <string> transformed_name
  [ FROM MODEL|CURRENT(CURRENT) COORDINATES ]
# nonlocal averaging with additional options
```

```
  BEGIN NONLOCAL AVERAGE <string>name
    SOURCE VARIABLE = NODAL|ELEMENT <string>variable_name
    TARGET VARIABLE = NODAL|ELEMENT <string>variable_name
    RADIUS = <real>radius_value
    #options only for nodal nonlocal averaging
    DISTANCE WEIGHTING FUNCTION = <string>function_name
    DISTANCE ALGORITHM = EUCLIDEAN_DISTANCE |
                         GRAPH_DISTANCE | EUCLIDEAN_GRAPH
    WEIGHTING VARIABLE = <string>nodal_weighting_variable_name
    PRINT DEBUG INFORMATION FOR NODE = <int>node_id
    #option only for element nonlocal averaging
    SAMPLE POINTS IN SPHERE AT RADIAL INCREMENT
          <real>increment AND ANGLE <real>angle
  END
  #
  # data filtering
  FILTER <string>new_var FROM GLOBAL|NODAL|ELEMENT|FACE
    <string>source_var USING <string>filter_name
  #
  # compute for element death
  COMPUTE AT EVERY STEP
  #
  # fine-grained control of when output is computed
  COMPUTE AT HEARTBEAT|HISTORY|RESULTS OUTPUT STEPS
  #
  # additional command
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [USER OUTPUT]
```

Explicit



Warning: A new `USER OUTPUT` block should be created in the input deck when using a different type of command line, since user output operations are order-dependent. If placed in the same `USER OUTPUT` block, the order the commands are processed may be incorrect, causing errors in computed results.

The `USER OUTPUT` command block enables the user to generate specialized output information which is derived from analysis results such as element stresses, displacements, and velocities. `USER OUTPUT` post-processes analysis results *in situ*, as the code is running, to produce a reduced set of specialized output information. The `USER OUTPUT` command block offers an alternative to writing out large quantities of data and then post-processing them with an external code to produce customized output results.

As an example, the `USER OUTPUT` command block could be used to sum the contact forces in a particular global axis direction, on a certain surface in the mesh, to yield a net resultant contact force on that surface. In this instance, `USER OUTPUT` would reduce a potentially large set of detailed contact data to a single, meaningful value for a surface (or set of surfaces).



In an explicit analysis, the `USER OUTPUT` command block may also be used to define a user element failure criterion. In this case, the `COMPUTE AT EVERY STEP` command must be used to ensure that the variable is computed at every time step. See Section 9.3.2.9 on element death for additional details.

There are several options for calculating user-defined quantities:

Compute A single command line in the command block is used to compute reductions of variables on subsets of the mesh. This option makes use of one of the `COMPUTE` result command lines. For instance, the above example of the contact force is a case where we can accomplish the desired result by using the `COMPUTE GLOBAL` command line.

User subroutine The command block specifies a user subroutine to run immediately preceding output to calculate any desired variable. This option makes use of a `NODE SET`, `SURFACE`, or `ELEMENT BLOCK SUBROUTINE` command line.

Copy There is an option to copy an element variable for an element to the nodes associated with the element, via the `COPY ELEMENT VARIABLE` command line, made available primarily for creating results files for some of the post-processing tools used with Sierra/SM.

Only one of these three options may be used in a given `USER OUTPUT` command block.

For the compute result option, a user-defined variable is automatically generated. This user-defined variable is given whatever name the user selects for `results_var_name` in the above specification for any of the three compute result command lines. Parenthesis syntax (Section 9.1) may be used to define reductions on integration points or components of a variable. By default, a reduction operation operates on each integration point. For example, if the compute global command was used to average values of element stress it would average the values of stress at all integration points of all elements. If the command was used to average `stress(:,1)`, the result would only be the average of stress on the first integration points.

If the user subroutine or copy option is used, the user will need to define some type of user variable with the `USER VARIABLE` command block described in Section 10.2.4.

User-defined variables, whether they are generated via the compute result option or the `USER VARIABLE` command block, are not automatically written to a results or history file. If the user wants to output any user-defined variables, these variables must be referenced in a results or history output specification (see Section 9.3.1 and Section 9.4, which describe the output of variables to results files and history files, respectively).

The `USER OUTPUT` command block contains five groups of commands—mesh-entity set, compute result, user subroutine, copy, and compute for element death. Each of these command groups is independent of the others. In addition to the command lines in the five command groups, there is an additional command line: `ACTIVE PERIODS`. The following sections provide descriptions of the command groups and the `ACTIVE PERIODS` command line.

9.3.2.1. Mesh-Entity Set Commands

The `mesh-entity set commands` portion of the `USER OUTPUT` command block specifies the nodes, element faces, or elements associated with the variable to be output. This portion of the

command block can include some combination of the following command lines:

```
NODE SET|NODESET = <string list>nodelist_names
SURFACE = <string list>surface_names
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list> surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes, element faces, or elements. See Section 7.1.1 for more information about the use of these command lines for mesh entities. If no mesh-entity commands are defined for user output, the output will be computed for all blocks in the mesh.

9.3.2.2. Compute Result Commands

The compute result commands are used to compute new variables by performing operations on an existing variable. Several general forms of the compute results command are supported shown below.

```
# Compute global norms or reductions of nodal, face, or
# element quantities
PROCESS ELEMENTS = ACTIVE_ONLY|INACTIVE_ONLY|ALL(ACTIVE_ONLY)
COMPUTE GLOBAL <string>result_var_name AS
  SUM|MAX|MIN|MAX ABSOLUTE VALUE|L2NORM|VARIANCE
  OF NODAL|FACE|ELEMENT <string>source_var_name
COMPUTE GLOBAL <string>result_var_name
  AS AVERAGE
  OF NODAL|ELEMENT|FACE <string>source_var_name
  [WEIGHTED BY <string>weight_var_name]
COMPUTE GLOBAL <string>result_var_name
  AS NONLOCAL AVERAGE
  OF NODAL <string>source_var_name
  OVER DOMAIN DEFINED BY RADIUS <real>r
  AND POINT <real>x <real>y <real>z
COMPUTE GLOBAL <string>result_var_name AS
  VOLUME INTEGRAL|SURFACE INTEGRAL
  OF NODAL|ELEMENT <string>source_var_name
COMPUTE GLOBAL <string>result_var_name
  AS SURFACE INTEGRAL OF ELEMENT ELEMENT_LOCAL_FORCE
  IN DIRECTION <real>dx <real>dy <real>dz
COMPUTE GLOBAL <string>result_var_name
  AS CLOSEST|FURTHEST DISTANCE TO <string> mesh_entity_name
COMPUTE GLOBAL <string>result_var_name
  AS MAX NORMAL|OPPOSING GAP BETWEEN <string> mesh_entity_name
```

```
COMPUTE GLOBAL <string>result_var_name AS
  INTERNAL REACTION [VECTOR|SHEAR|NORMAL(VECTOR)]
  [AT POINT <real>px <real>py <real>pz]
  [IN DIRECTION <real>dx <real>dy <real>dz]
COMPUTE GLOBAL <string>result_var_name AS
  INTERNAL REACTION [VECTOR|SHEAR|NORMAL(VECTOR)]
  [AT POINT <real>px <real>py <real>pz]
  [IN DEFINED DIRECTION <string>dir_name]
COMPUTE GLOBAL <string>result_var_name AS INTERPOLATION
  OF NODAL <string>source_var_name
  AT POINT <real>x <real>y <real>z
```

The COMPUTE GLOBAL reduction result command computes a single global value or set of global values by performing a user-specified operation on the given nodal or element variable source_var_name. A single global value, for example, might be the maximum value of one of the stress components of all the elements in our specified set, while a set of global values would be the maximum value of each stress component over all elements in the specified set.

The PROCESS ELEMENTS command line may be optionally specified. By default, global reductions are performed only on the active element set, which excludes elements killed by element death or other processes. To consider all elements, only alive elements, or only dead elements, options ALL, ACTIVE_ONLY, or INACTIVE_ONLY, respectively, may be specified.

```
# Store current value of an XY function into a global quantity
COMPUTE GLOBAL|NODAL|ELEMENT|FACE <string>result_var_name
  AS FUNCTION <string>function_name

# Store current value of a scalar expression into a global quantity
COMPUTE GLOBAL <string>result_var_name
  FROM EXPRESSION "<string>expression"

# Compute a global quantity from the time history of an existing
# global quantity
COMPUTE GLOBAL <string>result_var_name
  AS INTEGRAL OVER TIME OF GLOBAL <string>source_var_name

# Compute a global quantity from another global quantity
COMPUTE GLOBAL <string>result_var_name
  AS LOG|EXP|MAGNITUDE OF GLOBAL <string>source_var_name


# Compute an element quantity by operating on quantities stored
# at the element integration points
COMPUTE ELEMENT <string>result_var_name AS
  MAX|MIN|AVERAGE|SUM|LOG|EXP OF ELEMENT
  <string>source_var_name
COMPUTE ELEMENT <string>result_var_name
  AS NONLOCAL AVERAGE
```

```
  OF ELEMENT <string>source_var_name
  OVER DOMAIN DEFINED BY RADIUS <real>r

# Compute an element or face quantity by operating on quantities
# stored at nodes attached to the element or face
COMPUTE ELEMENT|FACE <string>result_var_name AS
  MAX|MIN|SUM OF NODAL <string>source_var_name
COMPUTE ELEMENT|FACE <string>result_var_name AS
  AVERAGE OF NODAL <string>source_var_name
  [WEIGHTED BY <string>weight_var_name]

# Compute a nodal quantity from the time history of an existing
# nodal quantity
COMPUTE NODAL <string>result_var_name AS
  MAX OVER TIME|MIN OVER TIME|ABSOLUTE VALUE MAX OVER TIME|
  AVERAGE OVER TIME|STANDARD DEVIATION OVER TIME
  OF NODAL <string>source_var_name
```

Explicit

```
COMPUTE NODAL <string>result_var_name AS
  END STEP VALUE OF NODAL VELOCITY

# Compute a nodal quantity from another nodal quantity
COMPUTE NODAL <string>result_var_name AS
  LOG|EXP OF NODAL <string>source_var_name
COMPUTE NODAL <string>result_var_name
  AS NONLOCAL AVERAGE
  OF NODAL <string>source_var_name
  OVER DOMAIN DEFINED BY RADIUS <real>r

# Compute an element quantity from the time history of an existing
# element quantity
COMPUTE ELEMENT <string>result_var_name AS
  MAX OVER TIME|MIN OVER TIME|ABSOLUTE VALUE MAX OVER TIME|
  AVERAGE OVER TIME|STANDARD DEVIATION OVER TIME
  OF ELEMENT <string>source_var_name
```

The `COMPUTE NODAL` and `COMPUTE ELEMENT` variable commands compute a new nodal or element field or user variable by operating on an existing field, for example, to compute the maximum stress in each element over the course of an analysis.

The `AS FUNCTION` option uses a time history function `function_name` to compute a global, element, node, or face variable. The function can be either a simple time history function or a more complex analytic function with custom arguments. In the analytic function, a `GLOBAL` variable may only be defined as a function of other `GLOBAL` variables. `NODAL`, `ELEMENT`, and `FACE` variables may be a function of `GLOBAL` variables and other `NODAL`, `ELEMENT`, or `FACE` variables, respectively; for instance, face variables may not depend on node variables.

The `FROM EXPRESSION` option computes a new global variable by evaluating an expression

involving other global variables. For example

```
COMPUTE GLOBAL energyNorm FROM EXPRESSION
  "max(abs(KineticEnergy), abs(InternalEnergy))"
```

would compute a new global variable called `energyNorm`, which is the maximum absolute value of either kinetic or internal energy at a given time.

The `INTEGRAL OVER TIME` option computes a new global variable by computing the time integral of an existing global variable from the analysis start time up until the current time.

In the above command lines, the following definitions apply:

- The string `source_var_name` is the name of the variable used to compute the result variable. (See Section 9.10 for a listing of available code variables).
- The string `result_var_name` is the name of a new variable in which the computed results are to be stored. To output its value, `result_var_name` may be specified in `RESULTS`, `HEARTBEAT`, and/or `HISTORY OUTPUT` blocks. The result variable may also be used as a source variable in other `USER OUTPUT` command blocks.
- The `SUM`, `AVERAGE`, `MAX`, `MIN`, and `MAX ABSOLUTE VALUE` global reduction commands compute the sum, average, maximum value, minimum value, and maximum absolute value of the given field, respectively, over all included mesh entities. The `AVERAGE` command may also take an optional `WEIGHTED BY` field to computed a weighted average. The weighted by field may be given any scalar field defined on the same entities as the source variable. Mostly commonly this weighted by field would be used to compute a mass or volume weighted average value.
- The `NONLOCAL AVERAGE` computes weighted averages over all nodes or elements inside spherical domains with size defined by the `RADIUS`. The search for points inside each domain is performed once during initialization, so the domains are defined in initial model coordinates. A `GLOBAL result_var_name` will perform nonlocal averaging over a single domain and requires a `POINT` to define the center of this domain. A `NODAL` or `ELEMENT result_var_name` will perform nonlocal averaging at every included mesh entity by creating a domain with the specified radius at every mesh entity. A `NODAL result_var_name` requires a `NODAL source_var_name` and an `ELEMENT result_var_name` requires an `ELEMENT source_var_name`. A nonlocal average over a nodal `source_var_name` is mass-weighted and a nonlocal average over an element `source_var_name` is volume-weighted.
- The `L2NORM` global reduction computes the vector L_2-norm, or Euclidean norm, of the given field on all included entities. The vector L_2-norm computes the square root of the sum of the squares of all field values. It is important to note that this output command does *not* compute a volume-integrated L_2-norm.
- The `VARIANCE` global reduction computes the variance over all included mesh entities, defined as the average of the squared differences from the mean.
- The `LOG` and `EXP` commands calculate the natural logarithm and exponential, respectively, of the variable on all included mesh entities. Logarithm and exponential are valid for scalars, symmetric tensors, and full tensors. A logarithmic or exponential mapping,

respectively, is performed on symmetric and full tensors rather than a component-wise scalar logarithm or exponential.



Warning: The `LOG` command is only defined for non-zero scalars and certain types of tensors (see [4] for details).

- The `MAGNITUDE` command calculates the magnitude of a global variable, computed as the square root of the sum of the squares of the variable components. For a scalar value the magnitude is the absolute value, and for a vector value it is the vector length.
- The `INTERNAL REACTION` command is a special user output reduction that computes the internal reaction on an interior plane of the model. To use this command a set of elements or blocks must be specified in the user output command block. The options `AT POINT` and `IN DIRECTION` optionally define a point on the plane and the plane normal, respectively, given in initial model coordinates. The direction may also be specified with the `IN DEFINED DIRECTION` command and a user-defined direction as described in Section 2.1.7. If not specified, the body's center of mass and the eigenvector corresponding to the minimum eigenvalue of the inertia tensor are used to define the plane.
  The reaction is computed as the net internal force across the specified plane. The default output is a global 3-dimensional vector variable with components in the global Cartesian coordinate system. The computed internal reaction force vector points as would be expected from a free body diagram. By default, the `VECTOR` option is used and the entire force vector is output.
  Optionally, the `SHEAR` and `NORMAL` modifiers can be used to resolve the internal reaction force vector onto the plane or plane normal direction, respectively. These specialty commands are intended for use with preloading structures or fasteners, but may be applied in more general cases. The `SHEAR` option projects the internal reaction force onto the cut plane and returns a single positive value that is the magnitude of the in-plane shear force. The `NORMAL` option projects the internal reaction force onto the given normal direction and returns a signed value that is positive for tensile loading and negative for compressive loading.
  The current algorithm used for the `INTERNAL REACTION` calculation evaluates the internal forces on a set of elements that intersect the cut plane. The given plane serves to divide the selected body into two parts, while the actual contour of the cut body is dictated by the discretization. Thus, there should be sufficient elements in the selected body so that there are elements on both sides of the plane.

  Warning: The `INTERNAL REACTION` command may return unexpected results if only a single layer of elements is provided.

- The `AS INTERPOLATION` command can be used in two different ways. In the first case, nodal variables can be interpolated to a specific point which stores the value in a global variable. The interpolation point coordinates are provided in initial model coordinates. However, the interpolation is performed using the containing element's shape functions, so the point moves with the deformation of the model in a Lagrangian manner. Eight-noded hexahedral and four-noded linear and ten-noded quadratic tetrahedral solid elements are

currently supported. The analysis will fail if a point outside of the model is given; the log file will describe the error, including the point that caused it.
In the second case, element variable values are created from nodal variable values. If the the receiving element field has the same number of values as the element has integration points, then an interpolation to the integration points is performed. If the receiving element field has one value, then an average of the element's nodal quantities is performed. All other cases result in an error. Eight-noded hexahedral, four-noded linear tetrahedral, and ten-noded quadratic tetrahedral solid elements are currently supported.
To interpolate an element variable at a point, it must first be projected to a nodal variable via the `EXTRAPOLATE` user output command. The projected nodal variable may then be interpolated to a point as usual. It is important to note that the `EXTRAPOLATE` command will yield a continuous projected nodal field across element block boundaries; to circumvent this limitation for inherently discontinuous element fields at block boundaries, such as stress at material interfaces, separate `EXTRAPOLATE` commands should be provided per element block, and the projected block-specific nodal variables may then be interpolated to points of interest. For example,

```
BEGIN USER OUTPUT
  BLOCK = block_1
  EXTRAPOLATE ELEMENT VARIABLE stress(xx) TO
    NODAL VARIABLE stress_b1
END USER OUTPUT

BEGIN USER OUTPUT
  BLOCK = block_2
  EXTRAPOLATE ELEMENT VARIABLE stress(xx) TO
    NODAL VARIABLE stress_b2
END USER OUTPUT

BEGIN USER OUTPUT
  INCLUDE ALL BLOCKS
  # interpolate to a point inside block_1
  COMPUTE GLOBAL stress_pt1 AS INTERPOLATION OF
    NODAL stress_b1 AT POINT 0.0 0.0 0.0
  # interpolate to a point inside block_2
  COMPUTE GLOBAL stress_pt2 AS INTERPOLATION OF
    NODAL stress_b2 AT POINT 5.0 0.0 0.0
END USER OUTPUT
```

For additional guidance on the `EXTRAPOLATE` user output command, consult Section 9.3.2.5.

- The `CLOSEST|FURTHEST DISTANCE` global reduction computes the closest or furthest distance between two mesh objects. This computation will take the mesh entities defined via the `BLOCK`, `SURFACE`, or `NODESET` commands (source) and compute the distance to the mesh entity defined on the compute line (target). If the target object is a block or a sideset, a closest point projection from the nodes in the source object to the faces in the target will be computed. If the target object is a nodeset, a point-to-point distance calculation will be

performed between the nodes in the source object to the nodes in the target object. See example syntax below for further details. For the closest distance calculation, an additional argument `SEARCH KDTREE_NODE_FACE|NODE_FACE|NODE_NODE|MIXED` may be specified. The default search type uses the `KDTREE_NODE_FACE` search which first does a coarse search using a k-d tree structure, then computes the closest point projections to the faces found in the coarse search. The `NODE_FACE|NODE_NODE|MIXED` options put all entities defined as the target object on all processors, then equally distribute all source entities across all processors. The k-d tree search should be the fastest option in almost all cases, but the other options could be faster when running with a large number of processors.

```
#Node-Face closest point projection is used
BEGIN USER OUTPUT
  BLOCK = block_1
  COMPUTE GLOBAL dist_b1_to_b2 AS CLOSEST DISTANCE TO block_2
END USER OUTPUT

#Node-Node distance calculation is used
BEGIN USER OUTPUT
  BLOCK = block_1
  COMPUTE GLOBAL dist_b1_to_ns1 AS CLOSEST DISTANCE TO nodelist_1
END USER OUTPUT
```

- The `MAX NORMAL|OPPOSING GAP` global reduction computes the max gap between two mesh objects. This computation will take the mesh entities defined via the `BLOCK`, `SURFACE`, or `NODESET` commands (source) and compute the furthest gap to the mesh entity defined on the compute line (target).
- If the `MAX NORMAL` option is used, a ray search is performed from the nodes of the source object to the faces of the target object, where the rays are formed using the node normals of the source object. If a ray is found to intersect a face on the target object, a closest point projection is taken from the node to the face, and the shortest distance between the node to the face is saved for each node. The maximum value of all closest distances found is returned as the max normal gap. This algorithm is useful when determining the gap between long objects and short objects (such as a pin in a hole).
- If the `MAX OPPOSING` option is used, the node normals for all nodes in the source object and the target object are computed. Then, the closest distance between each node on the source object to each node on the target object is computed; if the normals of the two nodes point in the same direction, that node is not saved as the closest distance. After all closest distances are computed between the source and target, the max value of all of the closest distances is returned as the max opposing gap. This algorithm is useful when it is possible that node normals on the source do not intersect any faces on the target (e.g. large deformation/motion). Example use cases are shown below in Figures 9-1 - 9-4.
- The `VOLUME INTEGRAL` global reduction is valid for nodal or element fields. The volume integral of an element field is computed as $\sum_{i=1}^{N} f_i w_i$ where N is the number of quadrature points, f_i are the field values at the integration points, and w_i are the effective quadrature point volumes. If a uniform gradient element is used, $N = 1$, and w_1 represents the element volume. The volume integral of a nodal field is computed in the same manner, but nodal field values must first be interpolated to the quadrature points of each element using the

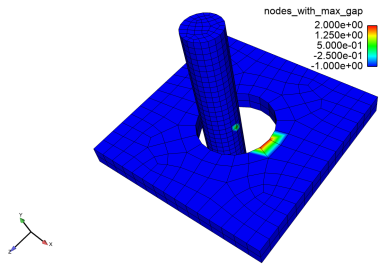


Figure 9-1. Pin-In-Hole Results for Max Normal Gap

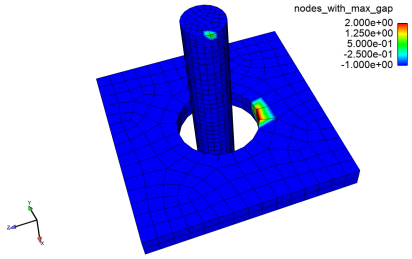


Figure 9-2. Pin-In-Hole Results for Max Opposing Gap

element shape functions. The quadrature points for integration of a nodal field are always taken from the fully integrated element, even if a uniform gradient element is used in the analysis; this method improves integration accuracy and is not likely to affect performance. Volume weighting is based on the reference (initial) quadrature point volumes and not the deformed mesh volumes.



Warning: Volume integration is currently only compatible with solid element types.

- The `SURFACE INTEGRAL` global reduction is nearly identical to the `VOLUME INTEGRAL` option. However, in this case, integration is only performed over surfaces of a given element, so a side set must be specified in the containing user output command block. Generic surface integration currently only supports variables of `NODAL` type; however, `SURFACE INTEGRAL` may also be used as a specialized user output for computing the net internal force over a sideset. The command operates on the internal force on the side set in the specified direction, yielding the net internal force magnitude in that direction. Note that this capability is only compatible with the element variable `element_local_force`.
- Five different methods are available for computing nodal or element variables over all time steps in the analysis: `MAX OVER TIME`, `MIN OVER TIME`, `ABSOLUTE VALUE MAX OVER TIME`, `AVERAGE OVER TIME`, or `STANDARD DEVIATION OVER TIME` which respectively compute the maximum, minimum, absolute maximum, average, or standard deviation of values in the source variable over all time.
- ***Explicit*** The `AS END STEP VALUE OF` command gives the user the option to output the end-step velocity. In the central difference time integrator implementation, velocities are typically stored at the half-step; i.e., at a given time t_n, the default velocity output yields

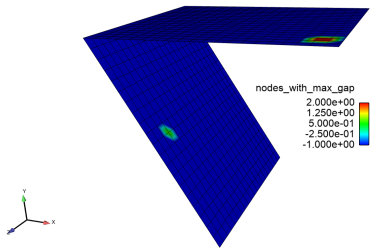


Figure 9-3. Large Motion Results for Max Normal Gap

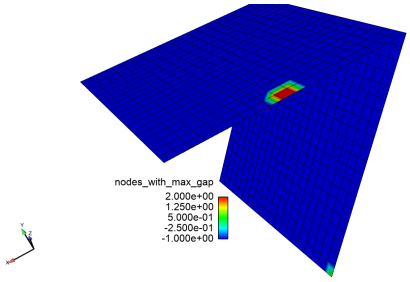


Figure 9-4. Large Motion Results for Max Opposing Gap

values at $t_n - 0.5\Delta t_n$ rather than t_n. This user output option moves the velocities to t_n and stores them in `result_var_name`.

Usage guidelines for the above `COMPUTE` commands are as follows:

- If both the source variable and target variable are element variables, and `MAX`, `MIN`, `VOLUME WEIGHTED AVERAGE`, `UNWEIGHTED AVERAGE`, or `AVERAGE` is used, then the operation computes a single element variable based on a reduction of the values at all integration points for that element. `VOLUME WEIGHTED AVERAGE` returns the volumetric weighted average over the element integration points. `UNWEIGHTED AVERAGE` and `AVERAGE` return the arithmetic average over the element integration points.
- Note that global reduction methods such `SUM`, `AVERAGE`, `MAX`, `MIN`, and `MAX ABSOLUTE VALUE` are only performed on active nodes and elements; this behavior can be changed with the `PROCESS ELEMENTS` command.
- The source variable used to compute a global variable must be either a nodal quantity or an element quantity, as specified by the `NODAL` or `ELEMENT` option.
- The source variable used to compute a nodal or element variable must be of the same type as the result variable.
- Standard component syntax may also be used in the `source_var_name` to specify operations on sub-components of a given variable (e.g., `stress(xx)`).

Examples Compute the total x-direction reaction force on `surface_1`:

```
SURFACE = surface_1
COMPUTE GLOBAL wall_x_reaction AS SUM OF NODAL reaction(x)
```

Compute the maximum force seen by each spot weld at the spot weld connection node.

```
COMPUTE NODAL max_spotn AS MAXIMUM OVER TIME OF NODAL #
    spot_weld_normal_force
```

Compute the volumetric average value of `eqps` in an element with multiple integration points. This is the sum of integration point values weighted by the relative integration point volumes.

```
COMPUTE ELEMENT eqps_avg AS VOLUME WEIGHTED AVERAGE OF ELEMENT eqps
```

Compute the approximate element centroid as the average the element coordinates:

```
COMPUTE ELEMENT elem_cent AS AVERAGE OF NODAL coordinates
```

Compute the total mass of elements removed by element death or other element removal processes.

```
COMPUTE GLOBAL removedMass AS SUM OF ELEMENT element_mass
PROCESS ELEMENTS = INACTIVE_ONLY
```

9.3.2.3. User Subroutine Commands

If the user subroutine option is invoked, the user-defined output quantities will be calculated by a subroutine written by the user explicitly for this purpose. The subroutine will be called by Sierra/SM at the appropriate time to perform the calculations. User subroutines allow for more generality in computing user-defined results than the `COMPUTE GLOBAL` command line. A subroutine could be used, for example, in case an analytic solution exists for a given problem, and it is desired to compute the difference between the analytic value and a corresponding computed value throughout an analysis. A full detailed description of user subroutines is given in Chapter 10.

The following command lines are related to the user subroutine option:

```
NODE SET SUBROUTINE = <string>subroutine_name |
  SURFACE SUBROUTINE = <string>subroutine_name |
  ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

The user subroutine option is invoked by using the NODE SET SUBROUTINE command line, the SURFACE SUBROUTINE command line, or the ELEMENT BLOCK SUBROUTINE command line. The particular command line selected depends on the mesh-entity variable type for which the result quantities are being calculated. For example, variables associated with nodes would be calculated by using a NODE SET SUBROUTINE command line, variables associated with faces by using a SURFACE SUBROUTINE command line, and variables associated with elements by using the ELEMENT BLOCK SUBROUTINE command line. The string subroutine_name is the name of a FORTRAN subroutine written by the user. A user subroutine in the USER OUTPUT command block returns no values. Instead, it performs its operations directly with commands such as aupst_put_nodal_var, aupst_put_elem_var, and aupst_put_global_var. See Chapter 10 for further discussion of these various put commands.

Following the selected command line (NODE SET SUBROUTINE, SURFACE SUBROUTINE, or ELEMENT BLOCK SUBROUTINE) are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 10.2.2 and consist of SUBROUTINE DEBUGGING OFF, SUBROUTINE DEBUGGING ON, SUBROUTINE REAL PARAMETER, SUBROUTINE INTEGER PARAMETER, and SUBROUTINE STRING PARAMETER. Examples of using these command lines are provided in Chapter 10.

Importantly, to implement the user subroutine option and output the calculated information, it is necessary to

1. create the user-defined variable with a USER VARIABLE command block;
2. calculate the results for the user-defined variable in the user subroutine; and
3. write the results for the user-defined variable to an output file by referencing it in a RESULTS OUTPUT command block and/or a HISTORY OUTPUT command block and/or a HEARTBEAT OUTPUT command block. In the RESULTS OUTPUT command block, a NODAL, ELEMENT, or GLOBAL command line would be used, depending on the USER VARIABLE command block definition. Similarly, in the HISTORY OUTPUT or HEARTBEAT OUTPUT command block, the appropriate form of the variable command line should be used.

9.3.2.4. Copy Command

```
COPY ELEMENT VARIABLE <string>ev_name TO NODAL VARIABLE
  <string>nv_name
COPY MAX ELEMENT VARIABLE <string>ev_name TO NODAL VARIABLE
  <string>nv_name
COPY SUM ELEMENT VARIABLE <string>ev_name TO NODAL VARIABLE
  <string>nv_name
COPY WEIGHTED SUM ELEMENT VARIABLE <string>ev_name TO NODAL
  VARIABLE <string>nv_name
```

The COPY command line copies the value of an element variable to a node associated with the element. The element variable to be copied is specified by ev_name; the name of the nodal

variable to which the value is being transferred is nv_name.

The COPY ELEMENT VARIABLE version of the command sets the nodal value as the average of the values in the connected elements. An example use would be creating a smoothed nodal Von Mises stress.

The MAX option sets the nodal value as the maximum value in any connected element. The SUM option sets the nodal value as the sum of the value in any connected element. The WEIGHTED SUM option takes the element value and divides it equally to the nodes of the element then sums up the nodal values. An example use of weighted sum would be to calculate a nodal volume from the weighted sum of element volumes. For the weighted sum the sum of the resultant quantity on all nodes will be equal to the sum of the source quantity on all elements.

9.3.2.5. Extrapolate Command

```
EXTRAPOLATE ELEMENT VARIABLE <string>ev_name TO NODAL VARIABLE
  <string>nv_name
```

The EXTRAPOLATE ELEMENT VARIABLE command line projects element variables to the nodes via L_2 projection. The element variable to be extrapolated is specified by ev_name. The nodal variable to which the element variable is projected is a user-defined variable denoted nv_name.

The L_2 projection operates on the inverse of the consistent mass matrix (by default) corresponding to the specified block(s). The extrapolation option is available for most solid elements such as hexahedra, tetrahedra, and quadrilateral cohesive zone elements.

The projection type performed by the extrapolate command may be modified by the EXTRAPOLATE PROJECTION TYPE command. A consistent projection is the default and does not modify the projection matrix. Selection of either a row-sum or proportional lumping scheme will cause the projection matrix to be modified before any extrapolation operations are performed. Lumped projections may be useful when a consistent projection results in undesirable negative values (for example, negative plastic strain or von Mises stress), or oscillations in the projected field.



Warning: Because the extrapolation operates on element variables, the NODE SET = command line is *not* compatible with user output extrapolation operations; either specific element blocks or INCLUDE ALL BLOCKS must be specified.

L_2 projection using the uniform gradient, fully integrated, and composite tetrahedral formulations of the 10-noded tetrahedron yields spurious nodal values at the 6 edge nodes. Instead, each 10-noded tetrahedron is converted to an equivalent 4-noded, linear tetrahedron based at the corner nodes during the projection phase. The extrapolated corner node values are then interpolated to the higher-order edge nodes of the original 10-noded tetrahedron.

The extrapolation can be restricted to specific element blocks with the BLOCK = command line. A *global* extrapolation linear system will be shared among all user output command blocks in which INCLUDE ALL BLOCKS is specified. However, a separate *local* extrapolation linear system will be created for each user output command block specifying individual blocks.



Warning: Global extrapolation should be avoided on heterogeneous sets of elements such as combinations of hexahedra and cohesive zones due to inconsistencies in their respective topological formulations.



Warning: Specifying the same nodal variable name for multiple element blocks and/or multiple user output extrapolation operations may lead to inconsistencies in nodal extrapolated quantities at any shared block boundaries. When extrapolating the same element variable (e.g., `stress`) to nodes of multiple blocks, it is recommended to specify a unique nodal output variable name on each block (e.g., `nodal_stress_block_1`, `nodal_stress_block_2`, etc.).

A particular linear solver may be selected for the L_2 projection using the `LINEAR SOLVER =` user output line command. The default is `DEFAULT_FETI_SOLVER`.

9.3.2.6. Variable Transformation Command

```
TRANSFORM NODAL|ELEMENT VARIABLE <string>variable_name
  TO COORDINATE SYSTEM <string>coord_sys_name
  AS <string> transformed_name
  [ FROM MODEL|CURRENT(CURRENT) COORDINATES ]
```

The `TRANSFORM NODAL|ELEMENT VARIABLE` command line transforms a nodal vector (displacement, velocity, acceleration, etc.) or an element tensor (stress, strain, etc.) from components in the global coordinate system to components in the coordinate system defined in a `BEGIN COORDINATE SYSTEM` command block having the name `coord_sys_name` (see Section 2.1.9). The transformed variables will be output to the results file as `transformed_name`.

By default most vectors and tensors are rotated into a new local X'Y'Z' Cartesian coordinate system. As an example consider transforming a nodal force to a cylindrical coordinate system. A local Cartesian X'Y'Z' coordinate system will be computed at the node location where X' aligns with the cylindrical radial direction, Y' aligns with the cylindrical hoop direction, and Z' aligns with the cylindrical Z direction. The force will then be transformed into this new Cartesian coordinate system in a way that is length preserving as shown in Figure 9-5.

The nodal fields `coordinates` and `model_coordinates` are treated specially. These positional vectors are transformed to new positions in the target coordinate system. This special position transformation only applies to these two specific nodal field names. For cylindrical coordinate system the coordinate vectors are transformed into R Θ Z coordinates where Θ is in radians as shown in Figure 9-6.



Warning: This command cannot be used to transform shell element tensors which are not computed in the global coordinate system. Output of shell stress and strain components in a user-defined, local co-rotational coordinate system are obtained with the element variables `transform_shell_stress` and `transform_shell_strain` as described in Section 9.3.1.

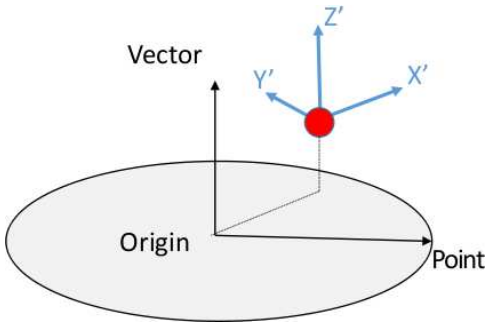


Figure 9-5. Default Cylindrical Transformation

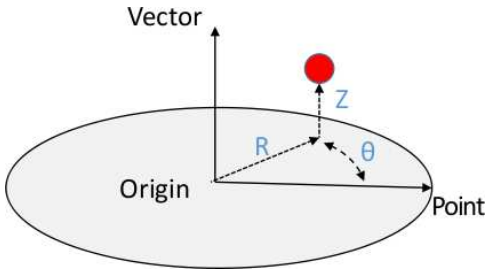


Figure 9-6. Cylindrical System Transformation of Coordinates



Warning: This command cannot be used with derived output variables. The derived output variables are variables such as `log_strain`, `biot_strain` and `green_lagrange_strain` which are not internally stored but are computed before being output to the results file

9.3.2.7. Data Filtering Commands

```
BEGIN FILTER <string>filter_name
  ACOEFF = <real_list>a_coeff
  BCOEFF = <real_list>b_coeff
  INTERPOLATION TIME STEP = <real>ts
END [FILTER]

BEGIN USER OUTPUT
  FILTER <string>new_var FROM GLOBAL|NODAL|ELEMENT|FACE
    <string>source_var USING <string>filter_name
END
```

The user output `FILTER` command creates a new variable `new_var` by performing an on-the-fly frequency filter of the named element, nodal or face variable `source_var`. The `BEGIN FILTER` command block defines a filter with the name `filter_name`. The filter defined by the begin filter command block is then referenced by the `USER OUTPUT` filter command.

In the `BEGIN FILTER` command block the A and B filtering coefficients are defined with the `ACOEFF` and `BCOEFF` command lines. The filter must define at least one A and one B coefficient,

there is no maximum number coefficients and the length of A and B do not need to match.

Filter operations assume that the data given at a constant time step. The explicit dynamics time step will tend to vary during the analysis. The `INTERPOLATION TIME STEP` command is used to linearly interpolate the data being produced at a non-constant time step down to some specified constant time step. The interpolation time step must be larger than zero and ideally should be specified to be smaller than the smallest time step with which the computations will iterate.

One way to obtain the filtering coefficients is with MATLAB. The following is an example of defining a third order Butterworth filter with a pass frequency of 100Hz at data interpolated to a time step of $1.0e-5$ seconds. The filter is then used to filter acceleration histories of the nodes to 100Hz. The MATLAB code below will give the desired filtering coefficients. The full 16-digit precision of the coefficients returned by MATLAB should be used. If truncated precision numbers are used, the filters can potentially be unstable.

```
clear;
format long e;
passFrequency = 100;
interp_ts = 1.0e-5;
butterCoeff = 2.0*interp_ts*passFrequency;
[bcoeff,acoeff] = butter(3,butterCoeff);
acoeff
bcoeff
```

The computed filtering coefficients can be used in a `USER OUTPUT` block as show below. If the analysis time step always remains above $1.0e-5$ the filter will be valid. If the analysis time step drops below $1.0e-5$ there could be aliasing issues, and a smaller interpolation time step should be specified.

```
BEGIN FILTER filt_100Hz
  ACOEFF = 1.000000000000000e+00 -2.987433650055722e+00 $
           2.974946132665442e+00 -9.875122361107358e-01
  BCOEFF = 3.081237301416628e-08  9.243711904249885e-08 $
           9.243711904249885e-08  3.081237301416628e-08
  INTERPOLATION TIME STEP = 1.0e-5
END
BEGIN USER OUTPUT
  FILTER ax100Hz FROM NODAL acceleration(x) USING filt_100Hz
  FILTER ay100Hz FROM NODAL acceleration(y) USING filt_100Hz
  FILTER az100Hz FROM NODAL acceleration(z) USING filt_100Hz
END
```

9.3.2.8. Nonlocal Averaging

```
BEGIN NONLOCAL AVERAGE <string>name
  SOURCE VARIABLE = NODAL|ELEMENT <string>variable_name
```

```
  TARGET VARIABLE = NODAL|ELEMENT <string>variable_name
  RADIUS = <real>radius_value
  #options only for nodal nonlocal averaging
  DISTANCE WEIGHTING FUNCTION = <string>function_name
  DISTANCE ALGORITHM = EUCLIDEAN_DISTANCE |
                       GRAPH_DISTANCE | EUCLIDEAN_GRAPH
  WEIGHTING VARIABLE = <string>nodal_weighting_variable_name
  PRINT DEBUG INFORMATION FOR NODE = <int>node_id
  #option only for element nonlocal averaging
  SAMPLE POINTS IN SPHERE AT RADIAL INCREMENT
          <real>increment AND ANGLE <real>angle
END
```

The `NONLOCAL AVERAGE` computes weighted averages over all nodes or elements inside spherical domains with size defined by the `RADIUS`. The search for points inside each domain is performed once during initialization, so the domains are defined in initial model coordinates. A `GLOBAL result_var_name` will perform nonlocal averaging over a single domain and requires a `POINT` to define the center of this domain. A `NODAL` or `ELEMENT result_var_name` will perform nonlocal averaging at every included mesh entity by creating a domain with the specified radius at every mesh entity. A `NODAL result_var_name` requires a `NODAL source_var_name` and an `ELEMENT result_var_name` requires an `ELEMENT source_var_name`.

By default, a nonlocal average over a nodal source variable is mass-weighted and a nonlocal average over an element source variable is volume-weighted. Additionally, a `DISTANCE WEIGHTING FUNCTION` can be used to specify the weighting used for nodal variables. The formula for the distance weighting is

$$nonlocalValue = \frac{\sum_{n=1}^{N} x(n)w(d(n))}{\sum_{n=1}^{N} w(d(n))} \tag{9.1}$$

where x(n) is the source value at node n, d(n) is the distance node n is away from the target node, and w(d(n)) is the weight function evaluated at d(n). An example input deck setup that uses the weighting function is shown below.

```
BEGIN FUNCTION include_everything_but_me
  TYPE IS PIECEWISE LINEAR
  BEGIN VALUES
    0.0      0.0
    1.0E-6   1.0
    3.0      1.0
  END VALUES
END

BEGIN NONLOCAL AVERAGE avg1
  SOURCE VARIABLE = NODAL contact_status
  TARGET VARIABLE = NODAL averaged_contact_status
  RADIUS = 2.0
```

```
  DISTANCE WEIGHTING FUNCTION = include_everything_but_me
END
```

A WEIGHTING VARIABLE may also be specified and used in combination with the DISTANCE WEIGHTING FUNCTION. The weighting variable may be any nodal variable used in Sierra/SM.

There are currently three different distance algorithms that can be used for nonlocal averaging: EUCLIDEAN_DISTANCE, GRAPH_DISTANCE, and EUCLIDEAN_GRAPH.

The EUCLIDEAN_DISTANCE option simply computes the distance between the source and target entity. No considerations are taken with respect to the mesh graph in this method.

The GRAPH_DISTANCE option starts at the source entity and walks the edges of the mesh to find connected target entities. As the mesh is walked, the graph distance is computed. If the graph distance from the source entity to the target entity is less than the given radius, the target entity is included in the kernel.

The EUCLIDEAN_GRAPH option walks the mesh and only finds entities within the sphere that are connected to the source entity. This algorithm first does a EUCLIDEAN search to get a set of nodes within the sphere. Then, the mesh is walked from the source node, and grabs all nodes connected to the source node, and recursively continues. Nodes outside of the euclidean set are not included and stop the recursion.

For nodal nonlocal averaging, debugging information may be printed to the logfile with the PRINT DEBUG INFORMATION FOR NODE command. Debugging information will be printed for all nodes specified in the line command.

For element nonlocal averaging, lower order integration may be specified via the SAMPLE POINTS IN SPHERE AT RADIAL INCREMENT <real>increment AND ANGLE <real>angle command. Instead of including every element that lies within the domain of the sphere, the elements are only included if a point specified by the above line command lies within the element. This line command will significantly improve performance of the algorithm, but in general, may be less accurate than including every element in the domain.

9.3.2.9. Compute at Every Step Command

```
COMPUTE AT EVERY STEP
```

If this command line appears in the USER OUTPUT command block, a user-defined variable in the command block will be written at every time step. Section 10.2.4 discusses user-defined variables.



In explicit dynamics analyses, user-defined variables can be used as criteria for element death. In this case, the COMPUTE AT EVERY STEP command must be used to force those variables to be computed at every time step. See Section 6.5 for more information on element death.

9.3.2.10. Additional Commands

Active and Inactive Periods Commands

```
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines determine when the `USER OUTPUT` command block is either active or inactive. See Section 2.6 for more information about this optional command line.

Compute at Output Steps Commands

```
COMPUTE AT HEARTBEAT|HISTORY|RESULTS OUTPUT STEPS
```

This command line specifies to compute the current user output block at only heartbeat, history, or results output steps. For instance, if a particular `USER OUTPUT` command block is computationally intensive, but the results of the user output computation are only needed at relatively infrequent results output intervals, specifying `COMPUTE AT RESULTS OUTPUT STEPS` would exclude the user output block from being computed at history and heartbeat output steps and could result in a significant simulation runtime savings.

If this command line is not specified anywhere in a `USER OUTPUT` command block, the block will be computed at all output steps. It is admissible to specify multiple `COMPUTE AT OUTPUT STEPS` command lines per `USER OUTPUT` command block, for instance, if it is desired to compute at both history and heartbeat output steps.

9.4. History Output

```
BEGIN HISTORY OUTPUT <string>history_name
  #
  # Database Commands
  DATABASE NAME = <string>history_file_name
  DATABASE TYPE = <string>database_type(exodusII)
  OVERWRITE = <string>OFF|ON|TRUE|FALSE|YES|NO
    (ON|TRUE|YES)
  TITLE <string>user_title
  PROPERTY = <string>propName = <string>propValue
  #
  # for global variables
  GLOBAL <string>variable_name
    [AS <string>history_variable_name]
  #
  # for mesh entity - node, edge, face,
  # element - variables
  NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
    AT NODE|NODAL|EDGE|FACE|ELEMENT <integer>entity_id
    AS <string>history_variable_name
  # for edge or face variables
  EDGE|FACE <string>variable_name
    AT EDGE|FACE <integer>entity_id.local_face
    AS <string>history_variable_name
  #
  # for nearest point output of mesh entity - node,
  # edge, face, element - variables
  NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
    NEAREST LOCATION <real>global_x,
      <real>global_y>, <real>global_z
    AS <string>history_variable_name
  START TIME = <real>output_start_time
  TIMESTEP ADJUSTMENT INTERVAL = <integer>steps
  AT TIME <real>time_begin INCREMENT =
    <real>time_increment_dt
  ADDITIONAL TIMES = <real>output_time1
    <real>output_time2 ...
  AT STEP <integer>step_begin INCREMENT =
    <integer>step_increment
  ADDITIONAL STEPS = <integer>output_step1
    <integer>output_step2 ...
  TERMINATION TIME = <real>termination_time_value
  SYNCHRONIZE_OUTPUT
  USE OUTPUT SCHEDULER <string>scheduler_name
  OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
    SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
```

Explicit

```
    SIGKILL|SIGILL|SIGSEGV
END [HISTORY OUTPUT <string>history_name]
```

A history file gives nodal variable results (displacements, forces, etc.) for specific nodes, edge variable results for specific edges, face variable results for specific faces, element results (stress, strain, etc.) for specific elements, and global results at specified times. A history file, the results to be included in this file, and the frequency at which results are written may be specified by a `HISTORY OUTPUT` command block. The command block appears inside the region scope. Typically, node, element, and occasionally face variables may be included.

More than one history file can be specified for an analysis. For each history file will correspond to one `HISTORY OUTPUT` command block in the input file. The command block for a history file description begins with:

```
BEGIN HISTORY OUTPUT <string>history_name
```

and is terminated with:

```
END [HISTORY OUTPUT <string>history_name]
```

where `history_name` is a user-selected name for the command block. Nested within the `HISTORY OUTPUT` command block are a set of command lines, as shown in the block summary given above.

See Section 9.2 for details on the general database commands such as `DATABASE NAME`, `DATABASE TYPE`, `OVERWRITE`, etc.

The other command lines that appear in the `HISTORY OUTPUT` command block determine the type and frequency of output information. Descriptions of these command lines follow in Section 9.4.1 through Section 9.4.5. The command lines for controlling the frequency of history output (in Section 9.4.1 through Section 9.4.5) are the same as those for controlling the frequency of results output. These frequency-related command lines are repeated here for convenience.

9.4.1. Output Variables

The `GLOBAL`, `NODE` (or `NODAL`), `EDGE`, `FACE`, or `ELEMENT` command line is used to select variables for output in the history file. One of several types of variables can be selected for output. The form of the command line varies depending on the variable type selected for output.

9.4.1.1. Global Output Variables

```
GLOBAL <string>variable_name
  [AS <string>history_variable_name]
```

This form of the command line lets the user select any global variable for output in the history file. The variable is selected with the string `variable_name`. The string `variable_name` is the

name of the global variable and can be either a variable listed in Section 9.10 or a user-defined variable (see Section 9.3.2 and Section 10.2.4).

A name, `history_variable_name`, can also be specified for the selected entity following the `AS` keyword. For example, suppose it is desired to output the CPU time (`cpu_time`) as `cpu` in the history file. The command line to accomplish this would be:

```
GLOBAL cpu_time AS cpu
```

The specification of an alias is optional for output of a global variable.

9.4.1.2. Mesh Entity Output Variables

```
NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
  AT NODE|NODAL|EDGE|FACE|ELEMENT <integer>entity_id
  AS <string>history_variable_name
```

This form of the command line allows the user to select any nodal, edge, face, or element variable for a specific mesh entity for output in the history file. For example, this form of the command line will pick the displacement at a specific node and output the displacement to the history file using a specified alias.

For this form of the command line, the mesh entity type is set to `NODE` (or `NODAL`), `EDGE`, `FACE`, or `ELEMENT` depending on the variable (set by `variable_name`) to be output. If the mesh entity type is set to `NODE` (or `NODAL`), `EDGE`, or `FACE`, the string `variable_name` can be either a variable listed in Section 9.10 or a user-defined variable (see Section 9.3.2 and Section 10.2.4). If the mesh entity type is set to `ELEMENT`, the string `variable_name` can be a variable listed in Section 9.10 or a user-defined variable (see Section 9.3.2 and Section 10.2.4.

Selection of a mesh entity follows the `AT` keyword. A mesh entity type (`NODE` [or `NODAL`], `EDGE`, `FACE`, or `ELEMENT`) must be specified, followed by the integer identifier, `entity_id`, for the mesh entity. A name, `history_variable_name`, for the selected entity must also be specified following the `AS` keyword. For example, output of accelerations at node 88 would be specified using the command line:

```
NODE ACCELERATION AT NODE 88 AS accel_88
```

where `accel_88` is the name used for this history variable in the history file.

For face and edge variables, an additional method of specifying the face or edge entity is provided. This method selects the face or edge by specifying the element that the face or edge is a member of and the local face or edge number on that element. To use this method, the element id and the face ordinal are specified as `id.ordinal`. The numbering of an elements faces and edges is specified in the ExodusII users manual [1]. Note that the selected face or edge must exist in the mesh database as a face or edge on a sideset, or the face or edge must be created internally by the application.

As an example, the command line to obtain the pressure for the third face of element 42 for the history file would be:

```
FACE PRESSURE AT FACE 42.3 AS PRES_42.3
```

Either the keyword NODE or NODAL can be used for nodal quantities.

As an example, the command line to obtain the von Mises stress for element 1024 for the history file would be:

```
ELEMENT VON_MISES AT ELEMENT 1024 AS GM_1024
```

Material state variables may be output in the same way. If an element has the state variable 'EQPS' then the value of 'EQPS' could be obtained on the element with a statement of the form:

```
ELEMENT EQPS AT ELEMENT 1024 AS eqps_1024
```

Additionally, the at element command supports the standard variable component syntax. For example:

```
ELEMENT STRESS(XX) AT ELEMENT 1024 AS S_XX_1024
```

where S_XX_1024 is the name that will be used for this history variable in the history file.

9.4.1.3. Nearest Point Output Variables

```
NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
  NEAREST LOCATION <real>global_x,
    real<global_y>, real<global_z>
  AS <string>history_variable_name
```

This form of the command line allows the user to select any nodal, edge, face, or element variable for output in the history file using a nearest point criterion. The command line described in this subsection is an alternative to the command line described in the preceding section, Section 9.4.1.2, for obtaining history output. The command line in this section or the command line in Section 9.4.1.2 produces history files with variable information. The difference between the two is how the variable information is selected.

For the above form of the command line, the mesh entity type is set to NODE (or NODAL), EDGE, FACE, or ELEMENT depending on the variable (set by variable_name) to be output. If the mesh entity type is set to NODE (or NODAL), EDGE, or FACE, the string variable_name can be either a variable listed in Section 9.10 or a user-defined variable (see Section 9.3.2 and Section 10.2.4). If the mesh entity type is set to ELEMENT, the string variable_name can be a variable listed in Section 9.10 or a user-defined variable (see Section 9.3.2 and Section 10.2.4).

The specific mesh entity used for output is determined by global coordinates specified by the NEAREST LOCATION keyword and its associated input parameters—global_x, global_y, global_z. The specific mesh entity chosen for output is as follows:

- If the mesh entity has been set to NODE (or NODAL), the node in the mesh selected for output is the node whose initial position is nearest the input global X, Y, and Z coordinates specified with the parameters global_x, global_y, and global_z.
- If the mesh entity has been set to EDGE, the edge in the mesh selected for output is the edge

with a center point (the average location of the two end points of the edge) whose initial position is nearest the input global X, Y, and Z coordinates specified with the parameters `global_x`, `global_y`, and `global_z`.

- If the mesh entity has been set to `FACE`, the face in the mesh selected for output is the face with a centroid whose initial position is nearest the input global X, Y, and Z coordinates specified with the parameters `global_x`, `global_y`, and `global_z`.
- If the mesh entity has been set to `ELEMENT`, the element in the mesh selected for output is the element with a centroid whose initial position is nearest the input global X, Y, and Z coordinates specified with the parameters `global_x`, `global_y`, and `global_z`.

In all the above cases, the original model coordinates are used when selecting the nearest entity, not the current coordinates.

A name, `history_variable_name`, must be specified for the selected entity following the `AS` keyword. As an example, to output the accelerations at a node closest to the point with global coordinates (1012.0, 54.86, 103.3141), the command line

```
NODE ACCELERATION
     NEAREST LOCATION 1012.0, 54.86, 103.3141 AS accel_near
```

would be used, where `accel_near` is the name used for this history variable in the history file. Either the keyword `NODE` or `NODAL` can be used for nodal quantities.

9.4.2. Outputting History Data on a Node Set

It is commonly desired to output history data on a single-node node set. If a mesh file is slightly modified, the node and element numbers will completely change. The node associated with a node set, however, remains the same, i.e., the node in the node set retains the same initial geometric location with the same connectivity to other elements, even when its node number changes. Therefore, we might want to specify the history output for a node set with a single node rather than with the global identifier for a node. This can easily be accomplished, as follows:

```
begin user output
  node set = nodelist_1
  compute global disp_ns_1 as average of nodal displacement
end

begin history output
  global disp_ns_1
end
```

If `nodelist_1` contains only a single node, the history output variable `disp_ns_1` will contain the displacement for the single node in the node set. If `nodelist_1` contains multiple nodes, the average displacement of the nodes will be output.

9.4.2.1. Specifying Time Steps for Output

The same output step control commands available for results output are also available for history output. See Section 9.3.1.9.

9.4.3. Synchronize Output

```
SYNCHRONIZE OUTPUT
```

The `SYNCHRONIZE OUTPUT` command can be used in the same manner it is used in the exodus output command block. See Section 9.3.1.10 for more details about this command.

9.4.4. Use Output Scheduler

```
USE OUTPUT SCHEDULER <string>scheduler_name
```

The `USE OUTPUT SCHEDULER` command can be used in the same manner it is used in the exodus output command block. See Section 9.3.1.11 for more details about this command.

9.4.5. Write History If System Error Encountered

```
OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
  SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
  SIGKILL|SIGILL|SIGSEGV
```

The `OUTPUT ON SIGNAL` command line is used to initiate the writing of a history file when the system encounters a type of system error. Only one error type in the list of error types should be entered for this command line. These system errors cause the code to terminate before the code can add any current history output (history output past the last history output time step) to the history file. If the code encounters the specified error type during execution, a history file will be written before execution is terminated. This command line can also be used to force the writing of a history file at some point during execution of the code. Suppose the command line

```
OUTPUT ON SIGNAL = SIGUSR2
```

is included in the input file. While the code is running, a user can execute (from the keyboard) the system command line

`kill -s SIGUSR2` *pid*

to terminate execution and force the writing of a results file. In the above system command line, *pid* is the process identifier, an integer. The `OUTPUT ON SIGNAL` command line is primarily a debugging tool for code developers.

9.5. Heartbeat Output

```
BEGIN HEARTBEAT OUTPUT <string>heartbeat_name
  # Can also use predefined streams "cout", "stdout",
  # "cerr", "clog", "log", "output", or "outputP0"
  STREAM NAME = <string>heartbeat_file_name
  APPEND = ON|OFF (OFF)
  #
  # Specify whether heartbeat file will be in spyhis (cth)
  # format, or default format
  FORMAT = SPYHIS|ORIGINAL|CSV(CSV)
  #
  # for global variables
  GLOBAL <string>variable_name
    [AS <string>heartbeat_variable_name]
  #
  # for mesh entity - node, edge, face,
  # element - variables
  NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
    AT NODE|NODAL|EDGE|FACE|ELEMENT <integer>entity_id
    AS <string>heartbeat_variable_name
  #
  # for nearest point output of mesh entity - node,
  # edge, face, element - variables
  NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
    NEAREST LOCATION <real>global_x,
      <real>global_y>, <real>global_z
    AS <string>heartbeat_variable_name
  # for edge or face variables
  EDGE|FACE <string>variable_name
    AT EDGE|FACE <integer>entity_id.local_face
    AS <string>heartbeat_variable_name
  START TIME = <real>output_start_time
  TIMESTEP ADJUSTMENT INTERVAL = <integer>steps
  AT TIME <real>time_begin INCREMENT =
    <real>time_increment_dt
  ADDITIONAL TIMES = <real>output_time1
    <real>output_time2 ...
  AT STEP <integer>step_begin INCREMENT =
    <integer>step_increment
  ADDITIONAL STEPS = <integer>output_step1
    <integer>output_step2 ...
  TERMINATION TIME = <real>termination_time_value
  SYNCHRONIZE_OUTPUT
  USE OUTPUT SCHEDULER <string>scheduler_name
  OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
    SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
```

Explicit

```
    SIGKILL|SIGILL|SIGSEGV
  PRECISION = <integer>precision
  LABELS = <string>OFF|ON
  LEGEND = <string>OFF|ON
  TIMESTAMP FORMAT <string>timestamp_format
  MONITOR = <string>RESULTS|RESTART|HISTORY

END [HEARTBEAT OUTPUT <string>heartbeat_name]
```

The heartbeat output is text output file that gives:

- nodal variable results (displacements, forces, etc.) for specific nodes,
- edge variable results for specific edges,
- face variable results for specific faces,
- element results (stress, strain, etc.) for specific elements, and
- global results

at specified times.



Known Issue: user-defined variables (see Section 10.2.4) are not currently supported with heartbeat output.

The output is written as text instead of the binary history output. A heartbeat file, the results to be included in this file, the formatting of the output, and the frequency at which results are written may be specified using a `HEARTBEAT OUTPUT` command block. The command block appears inside the region scope. Heartbeat output may typically involve global, node, and element variables, and, on some occasions, face variables.

More than one heartbeat file can be specified for an analysis. For each heartbeat file, there will be one `HEARTBEAT OUTPUT` command block. The command block for a heartbeat file description begins with

```
BEGIN HEARTBEAT OUTPUT <string>heartbeat_name
```

and is terminated with

```
END [HEARTBEAT OUTPUT <string>heartbeat_name]
```

where `heartbeat_name` is a user-selected name for the command block. Nested within the `HEARTBEAT OUTPUT` command block are a set of command lines, as shown in the block summary given above. The first command line listed (`STREAM NAME`) gives pertinent information about the heartbeat file. The command line

```
STREAM NAME = <string>heartbeat_file_name
```

gives the name of the heartbeat file with the string `heartbeat_file_name`. A file that already exists is overwritten. If the heartbeat file is to appear in the current directory and is named `job.h`, this command line would appear as

```
STREAM NAME = job.h
```

If the heartbeat file is to be created in some other directory, the command line would have to show the absolute path to that directory. If the directory does not exist yet, all necessary directories are automatically created.

In addition to specifying a specific file name, there are several predefined streams that can be specified. The predefined streams are:

- `cout` or `stdout` specifies standard output;
- `cerr`, `stderr`, `clog`, or `log` specifies standard error;
- `output` or `outputP0` specifies Sierra's standard output which is redirected to the file specified by the `-o` option on the command line.

Two meta-characters can appear in the name of the heartbeat file. If the `%P` character is found in the name, it will be replaced with the number processors being used for the run. For example, if, in a given analysis on 1024 processors, the name `heartbeat-%P/job.h` is used, then the name would be expanded to `heartbeat-1024/job.h`. The other recognized meta-character is `%B` replaced with the base name of the input file containing the input commands. For example, if the commands are in the file `my_analysis_run.i` and the heartbeat stream name is specified as `%B.h`, then the heartbeat data would be written to the file `my_analysis_run.h`. If the stream name command line is omitted, the default file name will be `%B.hrt` which is the base name of the input file with the extension `.hrt`

The other command lines that appear in the `HEARTBEAT OUTPUT` command block determine the type, frequency, and format of output information. Descriptions of these command lines follow in Section 9.5.1 through Section 9.5.8. The command lines for controlling the frequency of heartbeat output (in Section 9.3.1.9) are the same as those for controlling the frequency of results and history output. The `APPEND` command can be used to cause heartbeat information output by the current run to append to an existing file name rather than overwrite the existing file.

9.5.1. Output Variables

The `GLOBAL`, `NODE` (or `NODAL`), `EDGE`, `FACE`, or `ELEMENT` command line is used to select variables for output in the heartbeat file. One of several types of variables can be selected for output. The form of the command line varies depending on the variable type selected for output.

9.5.1.1. Global Output Variables

```
GLOBAL <string>variable_name
  [AS <string>heartbeat_variable_name]
```

This form of the command allows the user to select any global variable for output in the heartbeat file. The variable is selected with the string `variable_name`. The string `variable_name` is the name of the global variable and can be either a variable listed in Section 9.10 or a user-defined variable (see Section 9.3.2 and Section 10.2.4). The `variable_name` can also specify `time`,

timestep, or step to output the current simulation time, time step, or execution step, respectively.

A name, heartbeat_variable_name, for the selected entity can also be optionally specified following the AS keyword. For example, the CPU time (cpu_time) could be output as cpu in the heartbeat file using the command:

```
GLOBAL cpu_time AS cpu
```

9.5.1.2. Mesh Entity Output Variables

```
NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
  AT NODE|NODAL|EDGE|FACE|ELEMENT <integer>entity_id
  AS <string>heartbeat_variable_name
```

This form of the command allows the user to select any nodal, edge, face, or element variable for a specific mesh entity for output in the heartbeat file. For example, this command line will pick the displacement at a specific node and output the displacement to the heartbeat file using a chosen alias.

For this form of the command, the mesh entity type is set to NODE (or NODAL), EDGE, FACE, or ELEMENT depending on the variable (set by variable_name) to be output. If the mesh entity type is set to NODE (or NODAL), EDGE, or FACE, the string variable_name can be either a variable listed in Section 9.10 or a user-defined variable (see Section 9.3.2 and Section 10.2.4). If the mesh entity type is set to ELEMENT, the string variable_name can be a variable listed in Section 9.10 or a user-defined variable (see Section 9.3.2 and Section 10.2.4).

Selection of a specific mesh entity follows the AT keyword. A mesh entity type (NODE [or NODAL], EDGE, FACE, or ELEMENT) is specified, followed by the specific integer identifier, entity_id, for the mesh entity. A name, heartbeat_variable_name, for the selected entity can be specified following the AS keyword. For example, the accelerations at node 88 can be output using the command line:

```
NODE ACCELERATION AT NODE 88 AS accel_88
```

where accel_88 is the name used for this heartbeat variable in the heartbeat file.

For face and edge variables, an additional method of specifying the specific face or edge entity is provided. This method selects the face or edge by specifying the element that the face or edge is a member of and the local face or edge number on that element. To use this method, the element id and the face ordinal are specified as id.ordinal. The numbering of an elements faces and edges is specified in the Exodus II users manual [1]. Note that the selected face or edge must exist in the mesh database as a face or edge on a sideset, or the face or edge must be created internally by the application.

As an example, the command line to obtain the pressure for the third face of element 42 for the heartbeat file would be:

```
FACE PRESSURE AT FACE 42.3 AS PRES_42.3
```

Either the keyword NODE or NODAL can be used for nodal quantities.

As an example, the command line to obtain the von Mises stress for element 1024 for the heartbeat file would be:

```
ELEMENT VON_MISES AT ELEMENT 1024 AS vm_1024
```

where vm_1024 is the name used for this heartbeat variable in the heartbeat file.

9.5.1.3. Nearest Point Output Variables

```
NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
  NEAREST LOCATION <real>global_x,
    real<global_y>, real<global_z>
  AS <string>heartbeat_variable_name
```

This form of the command allows the user to select any nodal, edge, face, or element variable for output in the heartbeat file using a nearest point criterion. The command line described in this subsection is an alternative to the command line described in the preceding section, Section 9.5.1.2, for obtaining heartbeat output. The command line in this section or the command line in Section 9.5.1.2 produces heartbeat files with variable information. The difference in these two command lines (Section 9.5.1.3 and Section 9.5.1.2) is how the variable information is selected.

For the above form of the command, the mesh entity type is set to NODE (or NODAL), EDGE, FACE, or ELEMENT depending on the variable (set by variable_name) to be output. If the mesh entity type is set to NODE (or NODAL), EDGE, or FACE, the string variable_name can be either a variable listed in Section 9.10 or a user-defined variable (see Section 9.3.2 and Section 10.2.4). If the mesh entity type is set to ELEMENT, the string variable_name can be a variable listed in Section 9.10 or a user-defined variable (see Section 9.3.2 and Section 10.2.4).

The specific mesh entity used for output is determined by global coordinates specified by the NEAREST LOCATION keyword and its associated input parameters—global_x, global_y, global_z. The specific mesh entity chosen for output is as follows:

- If the mesh entity has been set to NODE (or NODAL), the node in the mesh selected for output is the node whose initial position is nearest the input global X, Y, and Z coordinates specified with the parameters global_x, global_y, and global_z.
- If the mesh entity has been set to EDGE, the edge in the mesh selected for output is the edge with a center point (the average location of the two end points of the edge) whose initial position is nearest the input global X, Y, and Z coordinates specified with the parameters global_x, global_y, and global_z.
- If the mesh entity has been set to FACE, the face in the mesh selected for output is the face with a centroid whose initial position is nearest the input global X, Y, and Z coordinates specified with the parameters global_x, global_y, and global_z.
- If the mesh entity has been set to ELEMENT, the element in the mesh selected for output is the element with a centroid whose initial position is nearest the input global X, Y, and Z coordinates specified with the parameters global_x, global_y, and global_z.

In all the above cases, the original model coordinates are used when selecting the nearest entity, not the current coordinates.

A name, `heartbeat_variable_name`, for the selected entity may be specified following the `AS` keyword. As an example, accelerations at a node closest to the point with global coordinates (1012.0, 54.86, 103.3141) may be output with the command line:

```
NODE ACCELERATION
    NEAREST LOCATION 1012.0, 54.86, 103.3141 AS accel_near
```

where `accel_near` is the name used for this heartbeat variable in the heartbeat file.

Either the keyword `NODE` or `NODAL` can be used for nodal quantities.

9.5.2. Outputting Heartbeat Data on a Node Set

It is commonly desired to output heartbeat data on a single-node node set. If a mesh file is slightly modified, the node and element numbers will completely change. The node associated with a node set, however, remains the same, i.e., the node in the node set retains the same initial geometric location with the same connectivity to other elements even when its node number changes. Therefore, we might want to specify the heartbeat output for a node set with a single node rather than with the global identifier for a node. This can easily be accomplished, as follows:

```
 begin user output
   node set = nodelist_1
   compute global disp_ns_1 as average of nodal displacement
 end

begin heartbeat output
  global disp_ns_1
end
```

If `nodelist_1` contains only a single node, the heartbeat output variable `disp_ns_1` will contain the displacement for the single node in the node set. If `nodelist_1` contains multiple nodes, the average displacement of the nodes will be output.

9.5.3. Specifying Time Steps for Output

The same output step control commands available for results output are also available for heartbeat output. See Section 9.3.1.9.

9.5.4. Synchronize Output

```
SYNCHRONIZE OUTPUT
```

The SYNCHRONIZE OUTPUT command can be used in the same manner it is used in the exodus output command block. See Section 9.3.1.10 for more details about this command.

9.5.5. Use Output Scheduler

```
USE OUTPUT SCHEDULER <string>scheduler_name
```

The USE OUTPUT SCHEDULER command can be used in the same manner it is used in the exodus output command block. See Section 9.3.1.11 for more details about this command.

9.5.6. Write Heartbeat On Signal

```
OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
  SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
  SIGKILL|SIGILL|SIGSEGV
```

The OUTPUT ON SIGNAL command line is used to initiate the writing of a heartbeat file when the system encounters the specified signal. The signal can either occur as the result of a system error, or the user can explicitly send the specified signal to the application (See the system documentation man pages for "signal" or "kill" for more information). Only one signal type in the list of signal types should be entered for this command line. These signals cause the code to terminate before the code can add any current heartbeat output (heartbeat output past the last heartbeat output time step) to the heartbeat file. If the code encounters the specified error type during execution, a heartbeat file will be written before execution is terminated. This command line can also be used to force the writing of a heartbeat file at some point during execution of the code. Suppose the command line

```
OUTPUT ON SIGNAL = SIGUSR2
```

is included in the input file. While the code is running, a user can execute (from the keyboard) the system command line

```
kill -s SIGUSR2 pid
```

to force the writing of a results file. In the above system command line, *pid* is the process identifier, which is an integer. The OUTPUT ON SIGNAL command line is primarily a debugging tool for code developers.

9.5.7. Heartbeat Output Formatting Commands

There are several command lines for the heartbeat section that modify the formatting of the heartbeat text output. The default output for the heartbeat data consists of a line beginning with a timestamp showing the current wall-clock time followed by multiple columns of data, for example:

```
Begin HeartBeat Region_1_Heartbeat
  Stream Name = output
  At Step 0, Increment = 10

  precision is 5

  global step
  global timestep as dt
  global time
  global total_energy as te
End
```

```
+[12:18:51] step=240, dt=3.13933e-04, time=7.56578e-02, te=4.02795e-06
+[12:18:51] step=250, dt=3.13933e-04, time=7.87971e-02, te=1.32125e-06
+[12:18:51] step=260, dt=3.13933e-04, time=8.19365e-02, te=6.88142e-07
+[12:18:51] step=270, dt=3.13933e-04, time=8.50758e-02, te=3.93574e-06
+[12:18:52] step=280, dt=3.13933e-04, time=8.82151e-02, te=7.46609e-06
+[12:18:52] step=290, dt=3.13933e-04, time=9.13545e-02, te=1.03856e-05
+[12:18:52] step=300, dt=3.13933e-04, time=9.44938e-02, te=1.36822e-05
+[12:18:52] step=310, dt=3.13933e-04, time=9.76331e-02, te=1.64630e-05
```

The above example begins each line with a timestamp followed by five labeled data columns. The precision of the real data is 5. This example does not have a legend. This format can be modified with these commands.

9.5.7.1. CTH SpyHis output format

```
FORMAT = SPYHIS|ORIGINAL|CSV(CSV)
```

The `FORMAT` described the output format for the heartbeat file

- `FORMAT=SPYHIS` specifies a particular spaced format that can be processed with the CTH SpyHis application as post-processor for time-history data.
- `FORMAT=ORIGINAL` is a legacy format option. This format is similar to the default `CSV` option, but does not have a comment after the time stamp column.
- `FORMAT=CSV` specifies a comma separated value format. This is the default format.

9.5.7.2. Specify floating point precision

```
PRECISION = <integer>precision
```

By default, the real data is written with a precision of 5 which gives 5 digits following the decimal point. This can be altered with the `PRECISION` command. If the command line `PRECISION = 2` is specified, then the above data would look like:

```
Begin HeartBeat Region_1_Heartbeat
  ...
```

```
        precision = 2
        ...
      End

+[12:18:51] step=240, dt=3.14e-04, time=7.57e-02, te=4.03e-06
+[12:18:51] step=250, dt=3.14e-04, time=7.88e-02, te=1.32e-06
+[12:18:51] step=260, dt=3.14e-04, time=8.19e-02, te=6.88e-07
```

The precision applies to all real data; it is not possible to specify a different precision for each variable.

9.5.7.3. Specify Labeling of Heartbeat Data

```
LABELS = <string>OFF|ON
```

The above example shows the default output which consists of a label and the data separated by '=.' The existence of the labels is controlled with the `LABELS` command. If `LABELS = OFF` is specified, then the above data would look like:

```
    Begin HeartBeat Region_1_Heartbeat
      ...
      labels = off
      precision = 2
      ...
    End

+[12:17:37] 240, 3.14e-04, 7.57e-02, 4.03e-06
+[12:17:37] 250, 3.14e-04, 7.88e-02, 1.32e-06
+[12:17:38] 260, 3.14e-04, 8.19e-02, 6.88e-07
```

9.5.7.4. Specify Existence of Legend for Heartbeat Data

```
LEGEND = <string>OFF|ON
```

Outputting data without labels can make it easier to work with the data in a spreadsheet program or other data manipulation program, but with no labels, it is difficult to determine what the data represents. The `LEGEND` output will print a line at the beginning of the heartbeat output identifying the data in each column. For example:

```
    Begin HeartBeat Region_1_Heartbeat
      ...
      legend = on
      labels = off
      precision = 2
      ...
    End
```

```
+[12:17:37] Legend: step, dt, time, te
+[12:17:37] 240, 3.14e-04, 7.57e-02, 4.03e-06
+[12:17:37] 250, 3.14e-04, 7.88e-02, 1.32e-06
+[12:17:38] 260, 3.14e-04, 8.19e-02, 6.88e-07
```

9.5.7.5. Specify format of timestamp

```
TIMESTAMP FORMAT <string>"timestamp_format"
```

Each line of the heartbeat output is preceded by a timestamp which shows the wall-clock time at the time that the line was output. This can be useful to verify that the code is still running and producing output and to determine how fast the code is running. The default timestamp is in the format `[12:34:56]` specified by the format `[%H:%M:%S]`. If a different format is desired, it can be specified with the `TIMESTAMP FORMAT` command line. The format must be surrounded by double or single quotes, and the format is defined to be the string between the first single or double quote and the last matching quote type. If a modified format is desired, the documentation for the UNIX `strftime` provides some details on time format specification. The example below shows a timestamp format delimited by `[` and `]`. The timestamp consists of an ISO-8601 date format followed by the current time.

```
...
     timestamp format "{%F %H:%M:%S}"
...
+{2008-03-17 09:26:17} 2212, 1.34244e-06, 2.96948e-03, 2.96948e-03
+{2008-03-17 09:26:17} 2213, 1.34244e-06, 2.97082e-03, 2.97082e-03
+{2008-03-17 09:26:17} 2214, 1.34244e-06, 2.97216e-03, 2.97216e-03
+{2008-03-17 09:26:17} 2215, 1.34244e-06, 2.97350e-03, 2.97350e-03
+{2008-03-17 09:26:17} 2216, 1.34244e-06, 2.97485e-03, 2.97485e-03
```

9.5.8. Monitor Output Events

```
MONITOR = <string>RESULTS|RESTART|HISTORY
```

It is sometimes a benefit to know when the code has written a new set of data to one of the other output files (restart output, history output, or results output). The heartbeat output will report this data if the `MONITOR` command line is specified. Each time output is performed to any of the monitored output types, a line will be written to the heartbeat file specifying the timestamp, the simulation time and step, and the label name of the output type. For example:

```
begin results output my_results
  at step 0, increment = 10
...
end results output results

begin heartbeat data hb
  stream name = stdout
```

```
    monitor = results
    labels = off
    legend = on
    timestamp format "%F %H:%M:%S "
    at step 0, increment = 2
    global step
    global timestep as dt
    global time
    element spring_engineering_strain at \#
        element 1 as sp1
end
```

Will give the output:

```
....
+2008-03-17 10:03:22  718, 1.34244e-06, 9.63871e-04, 9.63871e-04
-2008-03-17 10:03:22  Results data written at time = 0.00096656,
step = 720. my_results
+2008-03-17 10:03:22  720, 1.34244e-06, 9.66556e-04, 9.66556e-04
+2008-03-17 10:03:22  722, 1.34244e-06, 9.69241e-04, 9.69241e-04
+2008-03-17 10:03:22  724, 1.34244e-06, 9.71926e-04, 9.71926e-04
+2008-03-17 10:03:22  726, 1.34244e-06, 9.74611e-04, 9.74611e-04
+2008-03-17 10:03:22  728, 1.34244e-06, 9.77296e-04, 9.77296e-04
-2008-03-17 10:03:22  Results data written at time = 0.00097998,
step = 730. my_results
+2008-03-17 10:03:22  730, 1.34244e-06, 9.79981e-04, 9.79981e-04
....
```

9.6. Restart Data

```
BEGIN RESTART DATA <string>restart_name
  #
  # Database Commands
  DATABASE NAME = <string>restart_file
  INPUT DATABASE NAME = <string>restart_input_file
  OUTPUT DATABASE NAME = <string>restart_output_file
  DATABASE TYPE = <string>database_type(exodusII)
  OVERWRITE = <string>OFF|ON|TRUE|FALSE|YES|NO
    (ON|TRUE|YES)
  PROPERTY = <string>propName = <string>propValue

  RESTART TIME = <real>time
  RESTART = AUTOMATIC


  START TIME = <real>restart_start_time
  TIMESTEP ADJUSTMENT INTERVAL = <integer>steps
  AT TIME <real>time_begin INCREMENT =
    <real>time_increment_dt
  ADDITIONAL TIMES = <real>output_time1
    <real>output_time2 ...
  AT STEP <integer>step_begin INCREMENT =
    <integer>step_increment
  ADDITIONAL STEPS = <integer>output_step1
    <integer>output_step2 ...
  AT WALL TIME <real>time_begin[s|m|h|d] INCREMENT =
     <real>time_increment_dt[s|m|h|d]
  TERMINATION TIME = <real>termination_time_value
  OVERLAY COUNT = <integer>overlay_count
  CYCLE COUNT = <integer>cycle_count
  FILE CYCLE COUNT = <integer>file_cycle_count
  SYNCHRONIZE_OUTPUT
  USE OUTPUT SCHEDULER <string>scheduler_name
  COMPONENT SEPARATOR CHARACTER = <string>character|NONE
  OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
    SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
    SIGKILL|SIGILL|SIGSEGV
  OPTIONAL
  DECOMPOSITION METHOD = RCB|RIB|LINEAR|HSFC|BLOCK|CYCLIC|
    RANDOM|KWAY|GEOM_KWAY|KWAY_GEOM|METIS_SFC|EXTERNAL

END [RESTART DATA <string>restart_name]
```

Explicit

The restart capability in Sierra/SM allows a simulation to resume after the analysis has terminated. Restart can be used to break a long analysis into several smaller runs so the user can

examine intermediate results before proceeding. In case of abnormal termination, such as a machine outage, restart may be used to avoid rerunning the entire simulation. If a restart file has been written at various intervals throughout the analysis up to the point where the abnormal termination has occurred, the analysis can be restarted at a time before the abnormal termination, and the simulation can continue.

In general, the restart capability assumes the simulation input parameters are the same before and after the restart. Therefore, it is highly recommended that parameters such as boundary conditions, element types, and material models remain the same for at least the first step of each 'restart run.'

It is also highly discouraged to restart from an implicit simulation directly into an explicit simulation or vice versa. If an implicit/explicit restart is desired, refer to Section 9.6.8 below.

Many input parameters may be modified in a restart, such as hourglass control or solver parameters. If an abnormal termination is encountered due to, for example, element inversion or an implicit solver convergence problem, it is recommended to modify these parameters to allow the simulation to continue past the abnormal termination time previously encountered.

The amount of data written at a restart time is large, since it contains a complete description of the state for the problem at that time. The restart data includes not only information such as displacement, velocity, and acceleration, but also information such as element stresses and all state variables for the material model associated with each element. Careful consideration should be used to choose the increment for writing restart files (so that storage space is not completely filled) and to determine if it is necessary to overwrite previous restart files.

The time at which to restart or automatic restart can be specified within the Sierra scope (see Section 2.1.3.1 for more details), but the same commands are available within the `RESTART DATA` command block as well.

In order to perform a basic restart, it is required to specify (i) the restart database name to read or write and (ii) either automatic restart or the time for restart to take place, for example:

```
BEGIN RESTART DATA RESTART_DATA
  DATABASE NAME = g.rsout
  Restart = auto
END RESTART DATA RESTART_DATA
```

Note that if this command is used, restart data will only be written at the last step of the analysis or if Sierra/SM detects an internal error such as element inversion. To specify more frequent times to write restart data, the `AT TIME ...` or `AT STEP ...` command should be used.

```
BEGIN RESTART DATA RESTART_DATA
  DATABASE NAME = g.rsout
  Restart = auto
  At Time 1e-4 Increment = 1e-5
END RESTART DATA RESTART_DATA
```

For more complex uses of restart, please read below for further details.

9.6.1. Restart Options

The RESTART DATA command block begins with the input line:

```
BEGIN RESTART DATA <string>restart_name}
```

and is terminated with:

```
END [RESTART DATA <string>restart_name]
```

where restart_name is a user-selected name for the RESTART DATA command block.

Nested within the RESTART DATA command block are a set of command lines, as shown in the block summary given above.

```
DATABASE NAME = <string>restart_file
INPUT DATABASE NAME = <string>restart_input_file
OUTPUT DATABASE NAME = <string>restart_output_file
DATABASE TYPE = <string>database_type(exodusII)
OPTIONAL
```

The database name commands are used to specify the names of the files to read restart data from or write restart data to. If the OUTPUT DATABASE NAME command is used, restart data will be written to the filename specified. If the INPUT DATABASE NAME command is used, restart data will be read from the file with the specified filename. If the file does not exist at the specified path, the simulation will start from the beginning. If the DATABASE NAME command is used, the filename specified will be used for both writing out restart data and reading in restart data. Either the DATABASE NAME command should be used alone or the INPUT DATABASE NAME command line/OUTPUT DATABASE NAME command line should be used, but not both. The DATABASE NAME command is more automated than specifying individual input and output database names and is the preferred method. Section 9.6.1.2 through Section 9.6.1.5 shows how this command line is used in specific instances.

See Section 9.2 for more details on the general database commands such as DATABASE NAME, DATABASE TYPE, OVERWRITE, etc.

In coupled physics analyses with multiple regions, only a subset of the regions may have a restart database associated with them. The OPTIONAL command (Section 9.6.1) is used to tell the application that it is acceptable to restart the analysis even though a region does not have an associated restart database. This is only allowed in analyses containing multiple regions; if there is only a single region, it must have a restart database to restart.

9.6.1.1. What Times to Write

Several options exist to write restart at specific points in the analysis.

- The AT TIME and ADDITIONAL TIMES commands define restart in terms of analysis time
- The AT STEP and ADDITIONAL STEPS commands define restart in terms of analysis step number

- The `AT WALL TIME` command defines restart in terms of wall clock runtime. An example of the expected format is

  ```
  AT WALL TIME 3600s INCREMENT = 60m
  ```

  This example would write the first restart after 3600 seconds of runtime; then additional restart data will be written every 60 minutes thereafter. The units of time must be specified by `s` (seconds), `m` (minutes), `h` (hours), or `d` (days).

Note that restart can only be written at the end of a completed step. If a restart time t_s which does not coincide with a simulation time step t_i is specified, restart data will be written at the next analysis step $t_{n+1} \geq t_s$. For example, if an analysis has steps at times 0.1 and 0.2, asking for restart at time 0.125 will write the restart data at time 0.2 (the end of the next completed step). Additionally, restart will only be written at the end of a completed implicit loadstep.

9.6.1.2. Automatic Read and Write of Restart Files

Restart files can be tracked in an automated fashion by using the `RESTART = AUTO` command. Automatic restart can best be explained by an example. For the below example, assume the problem is run on two processors and all files are in the current working directory.

The automated restart option will not only manage the restart files to prevent overwriting, it will also manage the results files and history files to prevent overwriting. In the example given, assume the results file is defined with the command line

```
DATABASE NAME = rslt.e
```

where this command will generate the file `rslt.e`. Also assume history output is defined with the command line

```
DATABASE NAME = hist.h
```

to generate the file `hist.h`.

For all runs in this example, the command line

```
RESTART = AUTOMATIC
```

should be in either the SIERRA scope or the `RESTART DATA` command block. The `RESTART TIME` command line should not be used with `RESTART = AUTOMATIC`. For additional documentation of the `RESTART TIME` command, refer to Section 2.1.3.1.

Finally, assume the `RESTART DATA` command block is defined as follows:

```
BEGIN RESTART DATA RESTART_DATA
  DATABASE NAME = g.rsout
  AT TIME 0.0 INCREMENT = 0.25E-3
  RESTART = AUTO     #Optionally placed here or SIERRA scope
END RESTART DATA RESTART_DATA
```

In this block, the DATABASE NAME command line specifies a root file name for the restart file. The AT TIME command line gives the time and the increment for which restart information will be written.

Assume the first job is launched; the first run will generate the following output files:

```
# restart files
g.rsout.2.0
g.rsout.2.1
# results files
rslt.e.2.0
rslt.e.2.1
# history files
hist.h.2.0
hist.h.2.1
```

For the above files, the .2.0 and .2.1 extensions designate how the mesh was decomposed across the two processors.

Now, it is desired to start a second simulation using the restart data from the first, at the termination time of the previous run. The restart information in the input file will remain the same. In the TIME CONTROL command block, the only line that needs to be changed is the TERMINATION TIME command line. Even if the START TIME remains at 0.0, this time will be superseded by the last output time in the restart file.

The second job may then be launched; when it is completed, the following output files will be in the current working directory:

```
# restart files
g.rsout.2.0
g.rsout.2.1
g.rsout-s0002.2.0
g.rsout-s0002.2.1
# results files
rslt.e.2.0
rslt.e.2.1
rslt.e-s0002.2.0
rslt.e-s0002.2.1
# history files
hist.h.2.0
hist.h.2.1
hist.h-s0002.2.0
hist.h-s0002.2.1
```

Files that were previously written remain untouched; new files with the extension -s0002 contain the output data associated with the second run.

The process outlined above can be continued as long as necessary. As long as the RESTART = AUTO command remains, new output files with the -s* extension will continue to be generated and the previous files will *not* be overwritten.

9.6.1.3. User-Controlled Read and Write of Restart Files

Restart times and input/output database names may be specified using a combination of the `RESTART TIME` command line in the SIERRA scope and the `INPUT DATABASE NAME` and `OUTPUT DATABASE NAME` command lines in the `RESTART DATA` command block, allowing the user fine-grained control of restart data.

Note, however, that restart information is always automatically written upon element inversion.

User-controlled restart can best be explained by an example. We will use a two-processor example and assume all files will be in our current directory. In this example, the creation of new restart files is managed by the user so as not to overwrite existing restart files, analogously to the automated option. Unlike the automated option for restart, this controlled use of restart requires the user to manage restart file names manually so as to prevent overwriting previously generated files. The user must also manage the creation of new results and history output files so as not to overwrite existing files by creating new results/history files for each run in the sequence of restart runs requires changing the `DATABASE NAME` command line in the `RESULTS OUTPUT` and `HISTORY OUTPUT` command blocks. Explicit examples of `DATABASE NAME` command line usage in the `RESULTS OUTPUT` and `HISTORY OUTPUT` command blocks are omitted, since it closely parallels the pattern managing restart file names.

For the first run in the restart sequence, only a `RESTART DATA` command block is present in the region; there is no restart-related command line in the SIERRA scope of the input file. However, a

```
START TIME = 0.0
```

command line in a `TIME STEPPING` command block (within the `TIME CONTROL` command block) and a

```
TERMINATION TIME = 2.5E-3
```

command line within the `TIME CONTROL` command block are present to set the limits for the begin and end times. The `RESTART DATA` command block in the example input file is as follows:

```
BEGIN RESTART DATA RESTART_DATA
  OUTPUT DATABASE NAME = RS1.rsout
  AT TIME 0.0 INCREMENT = 0.5E-3
END RESTART DATA RESTART_DATA
```

In the first run, the restart option will generate the restart files:

```
RS1.rsout.2.0
RS1.rsout.2.1
```

The `.2.0` and `.2.1` file name extensions correspond to each processor's mesh file in a two-processor run. If our mesh file is `mesh.g`, then our mesh files on the individual processors

will be `mesh.g.2.0` and `mesh.g.2.1`. All restart information in the above files appears at time intervals of 0.5×10^{-3}, and the last restart information is written at time 2.5×10^{-3}.

For the second run in the sequence of restart runs, it is desired to start at the previous termination time, 2.5×10^{-3}, and terminate at time 5.0×10^{-3}. To do this, the command line

```
RESTART TIME = 2.5E-3
```

must be added to the SIERRA scope, and the termination time must be set using the command line

```
TERMINATION TIME = 5.0E-3
```

within the `TIME CONTROL` command block.

The actual start time for the second run in our analysis is now set by the restart time set on the `RESTART TIME` command line, 2.5×10^{-3}. The command line `START TIME = 0.0` in the `TIME STEPPING` command block is now superseded as the actual starting time for the second run by the restart commands. Any `START TIME` command line in a `TIME STEPPING` command block is still valid for defining time stepping blocks (used to set activation periods), but the restart process sets the actual start time for our analysis. This pattern of control for setting the actual start time holds for any run in the sequence of restart runs.

The `RESTART DATA` command block must also be modified to read:

```
BEGIN RESTART DATA RESTART_DATA
  INPUT DATABASE NAME = RS1.rsout
  OUTPUT DATABASE NAME = RS2.rsout
  AT TIME 0.0 INCREMENT = 0.5E-3
END RESTART DATA RESTART_DATA
```

For the second run, the files

```
RS1.rsout.2.0
RS1.rsout.2.1
```

are read, and the files

```
RS2.rsout.2.0
RS2.rsout.2.1
```

are written. All restart information in `RS2.rsout.2.0` and `RS2.rsout.2.1` appears at time intervals of 0.5×10^{-3}, restart information is written from times 2.5×10^{-3} to 5.0×10^{-3}. The restart files from the first run of the restart sequence have been preserved, since distinct input and output database names, `RS2.rsout` and `RS1.rsout`, respectively, have been specified.

Now, a third run is conducted in the restart sequence, starting at time 4.5×10^{-3} and terminating at time 8.5×10^{-3}. The second run terminated at time 5.0×10^{-3}; thus, a restart time must be manually specified:

```
RESTART TIME = 4.5E-3
```

and the TERMINATION TIME command line within the TIME CONTROL command block must be specified:

```
TERMINATION TIME = 8.5E-3
```

The RESTART DATA command block should also read:

```
BEGIN RESTART DATA RESTART_DATA
  INPUT DATABASE NAME = RS2.rsout
  OUTPUT DATABASE NAME = RS3.rsout
  AT TIME 0.0, INCREMENT = 0.5E-3
END RESTART DATA RESTART_DATA
```

The third run will read restart files:

```
RS2.rsout.2.0
RS2.rsout.2.1
```

and write restart data to files:

```
RS3.rsout.2.0
RS3.rsout.2.1
```

All restart information in RS3.rsout.2.0 and RS3.rsout.2.1 appears at time intervals of 0.5×10^{-3}, and restart information is written from times 4.5×10^{-3} to 8.5×10^{-3}. The restart files from the first and second runs of the restart sequence have been preserved, since distinct input and output database names have been specified.

9.6.1.4. Overwriting Restart Files

By default, every time an analysis is restarted, that output file the restart data is written to will be overwritten. As previously stated, it is best to have a separate restart file (or files for parallel runs) associated with each run in a sequence of restart runs if storage space permits. To not overwrite a restart file, the OVERWRITE = OFF command should be used. If this command is used and a restart file already exists with the name given in the DATABASE NAME command line, Sierra/SM will throw an error on start-up. To fix this error, the name in the DATABASE NAME command should be changed.

The below example demonstrates how the restart data will be written:

For the first run, suppose the termination time is set as

```
TERMINATION TIME = 1.0E-3
```

and the RESTART DATA command block is as follows:

```
BEGIN RESTART DATA
  DATABASE NAME = RS.out
  AT TIME 0.0 INTERVAL = 0.25E-3
END RESTART DATA
```

The above command block will output restart data to the file `RS.out` at intervals of 0.25×10^{-3} from time 0.0 through the termination time.

For the second run, suppose the termination time is set as:

```
TERMINATION TIME = 2.0E-3
```

and the restart data block is:

```
BEGIN RESTART DATA
  DATABASE NAME = RS.out
  AT TIME 0.0 INTERVAL = 0.25E-3
  RESTART TIME = 1.0E-3
END RESTART DATA
```

For this second run, restart information is read from the `RS.out` file(s) at time 1.0×10^{-3}. Then, these files are overwritten with new restart data generated starting at time 1.0×10^{-3} at intervals of 0.25×10^{-3}.

This pattern will continue for all analyses run in the set of restart runs.

9.6.1.5. Recovering from a Corrupted Restart

When performing an analysis during a sequence of restart runs, it is sometimes possible that a restart file becomes corrupted (due to a system crash for example). If the `RESTART=AUTO` option is exercised, restart will detect the corrupted entry and find the closest previous valid entry which can be used for restart.

A manual recovery can also be performed using the `RESTART TIME` command. Using this method, the user must select a restart time before the file corruption occurred. The `INPUT DATABASE NAME` and `OUTPUT DATABASE NAME` command lines should be used to avoid overwriting previous restart files. Additionally, the filenames should also be changed in the results and history command blocks to avoid overwriting those files.

9.6.2. Component Separator Character

```
COMPONENT SEPARATOR CHARACTER = <string>character|NONE
```

The `COMPONENT SEPARATOR CHARACTER` command can be used in the same manner it is used in the exodus output command block. See Section 9.3.1.7 for more details about this command.

9.6.3. Specifying Time Steps for Output

The same output step control commands available for results output are also available for restart output. See Section 9.3.1.9.

9.6.4. Reducing Restart File Size

As stated in the introduction, the amount of data written for restart output is significant; therefore, restart files can become large. The commands in this section may be used to limit restart file size. These commands may be used individually or collectively.

9.6.4.1. Overlay Count

```
OVERLAY COUNT = <integer>overlay_count
```

The `OVERLAY COUNT` command specifies the number of restart output times that will be overlaid on top of the current step before advancing to the next step. For example, suppose the `overlay_count` parameter $n_o = 2$, and restart data is set to be written every $\Delta t_s = 0.1$ second. At time 0.1 seconds, restart step 1 will be written to the output restart database. At time 0.2 seconds, restart information will be written over the step 1 information, which originally contained restart information at 0.1 seconds; similarly, step 1 information will once again be overwritten at time 0.3 seconds. At time 0.4 seconds, step 1 will now be complete and data will be written to step 2 At time 0.5 seconds, restart information will be written over the step 2 information, which originally contained information at 0.4 seconds. This process then continues throughout the simulation.

9.6.4.2. Cycle Count

```
CYCLE COUNT = <integer>cycle_count
FILE CYCLE COUNT = <integer>file_cycle_count
```

The `CYCLE COUNT` command line specifies the number of restart steps that will be written to the output restart database before previously written steps are overwritten. For example, suppose the `cycle_count` parameter $n_c = 5$, and restart data is set to be written every $\Delta t_s = 0.1$ second. The restart system writes information to the output restart database at times 0.1, 0.2, …, 0.5 seconds. The information at step 1 (originally written at time 0.1) is overwritten with information at time 0.6. At time 0.7, step 2 is overwritten, etc. At time 0.8, the output restart database will contain information at times 0.6, 0.7, 0.8, 0.4, and 0.5 seconds—time will not necessarily be monotonically increasing on a restart database that uses `CYCLE COUNT`. If it is desired to only keep the last step available on the output restart database, `CYCLE_COUNT` should be set to 1.

9.6.4.3. Example

As stated previously, the `CYCLE COUNT` and `OVERLAY COUNT` command lines can be used at the same time. The following example is used to explain that scenario:

It is assumed the overlay count $n_o = 2$ and the cycle count $n_c = 5$. Information is written to the output restart database time step every $\Delta t_s = 0.1$ second. The times at which information is written to the output restart database are $t_n = n\Delta t_s$, $n = 1, 2, \ldots, N$ seconds, where n is an output step and N is the number of steps. The overlay command will result in information at times $t_i = i(n_o + 1)\Delta t_s$, $i = 1, 2, \ldots, n_c$ seconds written at steps i on the output restart database—in the example, times 0.3, 0.6, 0.9, 1.2, and 1.5, at steps 1, 2, 3, 4, and 5. For $t_i > n_c(n_o + 1)\Delta t_s$, or 1.5 seconds, the cycle command will take effect because $n_c = 5$ different time steps are currently written on the output restart database. The overlay procedure then resumes at time $t = (1 + n_c(n_o + 1))\Delta t_s$: information at subsequent times 1.6, 1.7, and 1.8 seconds overwrite the information at step 1, which previously held information at time 0.3 second. Information at times 1.9, 2.0, and 2.1 seconds overwrites the information at step 2, etc. For any output step n, its corresponding restart output database step number n_s is given by

$$n_s = 1 + \frac{(n-1) \mod n_c(n_o+1)}{n_o + 1},$$

where $i \mod j$ represents the modulo operator, or the remainder of i/j. The quotient is evaluated using integer arithmetic (any fractional remainder is discarded). For example, if $n = 4$, $n_c = 5$, and $n_o = 2$, then $(4-1) \mod 5(2+1) = 3$ and $n_s = 1 + 3/3 = 2$.

The `FILE CYCLE COUNT` command is related to `CYCLE COUNT`; using this option, each restart dump will be written to a separate file suffixed with A, B, etc., where the count specifies how many separate files are written before the cycle repeats. For example, if `FILE CYCLE COUNT = 3` would cycle through `file-A.rs`, `file-B.rs`, `file-C.rs`, `file-A.rs`, etc. The maximum value for the cycle count is 26.

9.6.5. Synchronize Output

```
SYNCHRONIZE OUTPUT
```

The `SYNCHRONIZE OUTPUT` command can be used in the same manner it is used in the exodus output command block. See Section 9.3.1.10 for more details about this command.

9.6.6. Use Output Scheduler

```
USE OUTPUT SCHEDULER <string>scheduler_name
```

The `USE OUTPUT SCHEDULER` command can be used in the same manner it is used in the exodus output command block. See Section 9.3.1.11 for more details about this command.

9.6.7. Write Restart If System Error Encountered

```
OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
  SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
  SIGKILL|SIGILL|SIGSEGV
```

The OUTPUT ON SIGNAL command line is used to initiate the writing of a restart file when the computer system encounters an error. Only one error type in the list of error types should be entered for this command line. These system errors cause the code to terminate before the code can add any current restart output (restart output past the last restart output time step) to the restart file. If the code encounters the specified error type during execution, a restart file will be written before execution is terminated. This command line can also be used to force the writing of a restart file at some point during execution of the code. Suppose the command line

```
OUTPUT ON SIGNAL = SIGUSR2
```

is included in the input file. While the code is running, a user can execute (from the keyboard) the system command line

```
kill -s SIGUSR2 {pid}
```

to terminate execution and force the writing of a results file. In the above system command line, pid is the process identifier, an integer.

The most useful application of the command line is to send a signal via a system command line to write a restart file. The OUTPUT ON SIGNAL command line is primarily a debugging tool for code developers.

9.6.8. Restarting an Implicit/Explicit Analysis

Restarting from an implicit analysis directly into an explicit analysis (or vice versa) is not expected to work correctly; Sierra/SM uses different default settings for the two simulation types, and internal modifications are required when transitioning from a simulation which does not consider mass/inertial effects (implicit quasistatics) to one that does (explicit dynamics), or vice versa. To start an explicit simulation from a restart file generated by an implicit simulation, for example, the following procedure should be followed:

Suppose the input file used to generate the implicit restart file is called implicit.i, and the file that contains the input commands for the explicit simulation is called explicit.i. Before restarting into the explicit simulation, the contents of the adagio procedure in implicit.i must be copied and pasted into the procedure before the presto procedure in explicit.i—the procedures should have different names. If using the RESTART=AUTO option, the restart block in the implicit procedure may remain unchanged. If manually specifying the time at which to restart, the RESTART TIME should be set to the termination time of the implicit procedure (prior time, if desired). A procedural transfer block should then be added to the adagio procedure performing the explicit dynamics portion of the run. The simplest and most conventional procedural transfer block for this purpose is as follows:

```
BEGIN PROCEDURAL TRANSFER trans
  INCLUDE ALL BLOCKS
END
```

For more information on procedural transfers, see Section 11.2. This pattern should always be followed when restarting from different analysis types (e.g., Implicit Quasistatics to Implicit

Dynamics, Explicit Dynamics to Implicit Quasistatics, Explicit Dynamics to Implicit Dynamics, etc.).

9.7. Output Scheduler

In an analysis with multiple regions, it can be difficult to synchronize output such as results files, history files, and restart files. To help synchronize output for analyses with multiple regions, an `OUTPUT SCHEDULER` command block may be defined at the SIERRA scope. This scheduler can then be referenced

- in the `RESULTS OUTPUT` command block to control the output of results information,
- in the `HISTORY OUTPUT` command block to control the output of history information,
- in the `HEARTBEAT OUTPUT` command block to control the output of heartbeat information, and/or
- in the `RESTART DATA` command block to control the writing of restart files.

The `OUTPUT SCHEDULER` command block is defined in the SIERRA scope and is referenced by a `USE OUTPUT SCHEDULER` command line inside the appropriate output block. Section 9.7.1 describes the `OUTPUT SCHEDULER` command block, and Section 9.7.2 illustrates how this block is referenced with the `USE OUTPUT SCHEDULER` command line.

9.7.1. Output Scheduler Command Block

Explicit

```
BEGIN OUTPUT SCHEDULER <string>scheduler_name
  START TIME = <real>output_start_time
  TIMESTEP ADJUSTMENT INTERVAL = <integer>steps
  AT TIME <real>time_begin INCREMENT =
    <real>time_increment_dt
  ADDITIONAL TIMES = <real>output_time1
    <real>output_time2 ...
  AT STEP <integer>step_begin INCREMENT =
    <integer>step_increment
  ADDITIONAL STEPS = <integer>output_step1
    <integer>output_step2 ...
  TERMINATION TIME = <real>termination_time_value
END [OUTPUT SCHEDULER <string>scheduler_name]
```

An output scheduler is defined with a command block in the SIERRA scope. The `OUTPUT SCHEDULER` command block begins with the input line:

```
BEGIN OUTPUT SCHEDULER <string>scheduler_name
```

and is terminated with the line:

```
END OUTPUT SCHEDULER <string>scheduler_name
```

where `scheduler_name` is a user-defined name for the command block. All the normal scheduling command lines are valid in an `OUTPUT SCHEDULER` command block.

9.7.1.1. Specifying Time Steps for Output

The same output step control commands available for results output are also available in the output scheduler. See Section 9.3.1.9.

9.7.2. Example of Using the Output Scheduler

Once an output scheduler has been defined via the `OUTPUT SCHEDULER` command block, it can be used by inserting a `USE OUTPUT SCHEDULER` command line in any of the command blocks: `RESULTS OUTPUT`, `HISTORY OUTPUT`, and `RESTART DATA`. The next paragraph provides an example of using output schedulers.

In the SIERRA scope, we define two output schedulers, `Timer` and `Every_Step`:

```
BEGIN OUTPUT SCHEDULER Timer
  AT TIME 0.0 INCREMENT = 10.0e-6
  TIME STEP ADJUSTMENT INTERVAL = 4
END OUTPUT SCHEDULER Timer
#
BEGIN OUTPUT SCHEDULER Every_Step
  AT STEP 0 INCREMENT = 1
END OUTPUT SCHEDULER Every_Step
```

With the `USE OUTPUT SCHEDULER` command, we reference the scheduler named `Timer` for results output:

```
BEGIN RESULTS OUTPUT Out_Region_1
  .
  USE OUTPUT SCHEDULER Timer
  .
END RESULTS OUTPUT Out_Region_1
```

With the `USE OUTPUT SCHEDULER` command, we reference the scheduler named `Every_STEP` for history output:

```
BEGIN HISTORY OUTPUT Out_Region_2
  .
  USE OUTPUT SCHEDULER Every_Step
  .
END HISTORY OUTPUT Out_Region_2
```

9.8. Variable Interpolation

The variable interpolation command block is used to transfer data from one block or surface to another. Generally the transferred data is output data that will be used for a subsequent analysis. As an example the variable interpolation command block was created primarily to aid acoustic analysis where the derivative of void element volume is mapped from the structural Sierra/SM mesh to a background acoustic mesh for use in Salinas.

```
BEGIN VARIABLE INTERPOLATION [<string>var_interp_name]
  TRANSFER TYPE = INTERPOLATE FROM NEAREST FACE|
    SUM TO NEAREST ELEMENT
  SOURCE VARIABLE = ELEMENT|NODAL|
    SURFACE NORMAL NODAL <string>source_var_name
  SOURCE ELEMENT BLOCK = <string>source_block
  SOURCE SURFACE = <string>source_surface
  TARGET VARIABLE = ELEMENT|FACE|NODAL
    <string>target_var_name
  TARGET DERIVATIVE VARIABLE = ELEMENT|FACE|NODAL
    <string>target_deriv_var_name
  TARGET SURFACE = <string>target_surface
  SEARCH TOLERANCE = <real>search_tol
  PROXIMITY SEARCH TYPE = RAY SEARCH|SPHERE SEARCH
  RAY SEARCH DIRECTION = <real>vecx <real>vecy <real>vecz
  ONE WAY RAY SEARCH DIRECTION = <real>vecx <real>vecy <real>vecz
  RAY SEARCH DIRECTION = ORTHOGONAL TO LINE <real>p1x
    <real>p1y <real>p1z <real>p2x <real>p2y <real>p2z
  RAY SEARCH DIRECTION = TARGET SURFACE NORMAL

  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [VARIABLE INTERPOLATION <string>var_interp_name]
```

The command `TRANSFER TYPE` can be set to either `INTERPOLATE FROM NEAREST FACE` or `SUM TO NEAREST ELEMENT`. The `TRANSFER TYPE` restricts the commands that can be used in conjunction with it.

When `TRANSFER TYPE` is set to `INTERPOLATE FROM NEAREST FACE` the available commands are restricted to:

```
BEGIN VARIABLE INTERPOLATION [<string>var_interp_name]
  TRANSFER TYPE = INTERPOLATE FROM NEAREST FACE
  SOURCE VARIABLE = NODAL|SURFACE NORMAL NODAL
    <string>source_var_name
  SOURCE SURFACE = <string>source_surface
  TARGET VARIABLE = NODAL <string>target_var_name
  TARGET DERIVATIVE VARIABLE = NODAL <string>target_deriv_var_name
```

```
  TARGET SURFACE = <string>target_surface
  SEARCH TOLERANCE = <real>search_tol
  PROXIMITY SEARCH TYPE = RAY SEARCH|SPHERE SEARCH
  RAY SEARCH DIRECTION = <real>vecx <real>vecy <real>vecz
  ONE WAY RAY SEARCH DIRECTION = <real>vecx <real>vecy <real>vecz
  RAY SEARCH DIRECTION = ORTHOGONAL TO LINE <real>p1x
    <real>p1y <real>p1z <real>p2x <real>p2y <real>p2z
  RAY SEARCH DIRECTION = TARGET SURFACE NORMAL
END [VARIABLE INTERPOLATION <string>var_interp_name]
```

For `TRANSFER TYPE` = `INTERPOLATE FROM NEAREST FACE`, a given point on the target surface is mapped to the closest point on the source surface. At that closet source point the source variable is interpolated. The interpolated value is then copied from the interpolated source point to the target point. Optionally if the `SURFACE NORMAL NODAL` command is being used and the variable being transferred is a vector then only the surface normal component of that variable will be transferred. This is useful for several applications involving transferring solid mesh quantities such as velocity to a reference mesh modeling a fluid (e.g., air). In such applications only the surface normal vibration is relevant to the air acoustic solve.

When `TRANSFER TYPE` is set to `SUM TO NEAREST ELEMENT` the available commands are restricted to:

```
BEGIN VARIABLE INTERPOLATION [<string>var_interp_name]
  TRANSFER TYPE = SUM TO NEAREST ELEMENT
  SOURCE VARIABLE = ELEMENT <string>source_var_name
  SOURCE ELEMENT BLOCK = <string>source_block
  TARGET VARIABLE = ELEMENT <string>target_var_name
  TARGET DERIVATIVE VARIABLE = ELEMENT
    <string>target_deriv_var_name
  TARGET SURFACE = <string>target_surface
  SEARCH TOLERANCE = <real>search_tol
  PROXIMITY SEARCH TYPE = RAY SEARCH|SPHERE SEARCH
  RAY SEARCH DIRECTION = <real>vecx <real>vecy <real>vecz
  ONE WAY RAY SEARCH DIRECTION = <real>vecx <real>vecy <real>vecz
  RAY SEARCH DIRECTION = ORTHOGONAL TO LINE <real>p1x
    <real>p1y <real>p1z <real>p2x <real>p2y <real>p2z
END [VARIABLE INTERPOLATION <string>var_interp_name]
```

For `TRANSFER TYPE` = `SUM TO NEAREST ELEMENT`, a given element in the source block computes its centroid. Then the closest face to that centroid on the target surface is found. The source element value is summed to the element attached to that target face.

The command `SOURCE VARIABLE` specifies the variable type, `ELEMENT`, `GLOBAL`, or `NODAL`, and the name of the source variable `source_var_name`.

The command `SOURCE SURFACE` specifies the name of the source side set `source_surface`.

The command `TARGET VARIABLE` specifies the variable type, `ELEMENT`, `FACE`, or `NODAL`, and the name of the target variable `target_var_name`. The target variable must exist. If necessary, a

target variable can be created using the BEGIN USER VARIABLE command block (Section 10.2.4).

The command TARGET DERIVATIVE VARIABLE can be used to optionally compute the time derivative of the target variable. This command specifies the type of variable, ELEMENT, FACE, or NODAL, and the name of the variable to store the derivative, target_deriv_var_name. The target derivative variable must exist. If necessary, a target variable can be created using the BEGIN USER VARIABLE command block (Section 10.2.4).

The command TARGET SURFACE specifies the name of the target side set target_surface.

The command SEARCH TOLERANCE specifies a search distance for use in the proximity search.

The command PROXIMITY SEARCH TYPE can be set to either RAY SEARCH or SPHERE SEARCH. When PROXIMITY SEARCH TYPE = SPHERE SEARCH the SEARCH TOLERANCE is used as the radius for closest point searches. When PROXIMITY SEARCH TYPE = RAY SEARCH the SEARCH TOLERANCE and any RAY SEARCH DIRECTION are used together to form rays with lengths equal to the SEARCH TOLERANCE. If the RAY SEARCH DIRECTION command is used then the rays will extend in both the positive and negative directions. If the ONE WAY RAY SEARCH DIRECTION is used the rays will extend in only the positive direction. The one way rays are currently only available with the explicitly defined search direction vector. Faces penetrated by a ray are included in the set of faces for determining the closest point.

The commands RAY SEARCH DIRECTION and ONE WAY RAY SEARCH DIRECTION specify a search direction for the PROXIMITY SEARCH = RAY SEARCH DIRECTION case. A RAY SEARCH DIRECTION or ONE WAY RAY SEARCH DIRECTION can be specified directly with three values, vecx, vecy, and vecz. Alternatively, RAY SEARCH DIRECTION can be calculated through options ORTHOGONAL TO LINE or TARGET SURFACE NORMAL. For ORTHOGONAL TO LINE the search direction is the vector between a given point and the closest point on the infinite line specified with the two points p1x, p1y, p1z and p2x, p2y, p2z. For TARGET SURFACE NORMAL the search direction is the nodal normal vector calculated from the TARGET SURFACE.

9.8.1. Variable Interpolation Examples

The variable interpolation commands are complicated and best explained by the following example of a likely use case for this capability.

Example 1: Transferring volume infusion term to acoustic analysis. The example geometry is shown in Figure 9-7a run with the below syntax.

```
BEGIN VARIABLE INTERPOLATION
  TRANSFER TYPE = SUM TO NEAREST ELEMENT
  SOURCE VARIABLE = ELEMENT volume_first_derivative
  SOURCE ELEMENT BLOCK = block_1
  TARGET VARIABLE = ELEMENT volume_change_rate
```

```
  TARGET SURFACE = surface_2
  SEARCH TOLERANCE = 1.0e-3
END [VARIABLE INTERPOLATION <string>var_interp_name]
```

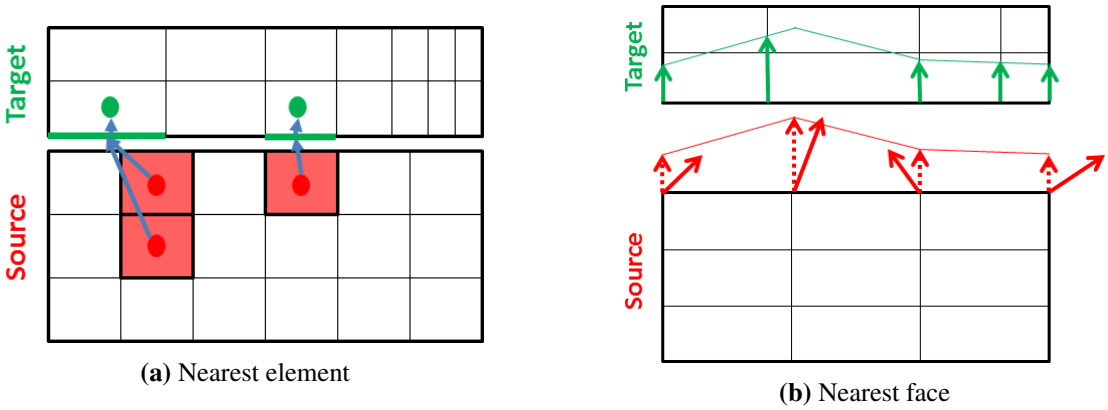
The volume derivatives are computed void elements (red) in source block one. At the centroid of each of these elements the closest facet on the target surface `surface_2` is found (green). The volume derivatives are then summed to the elements attached to the target surface faces (following blue arrows). In this way the change of volume of structural void elements can be mapped to gas source term in a subsequent Salinas acoustic analysis. A similar boundary condition can be used to map structural energy source terms for a subsequent thermal analysis.

Example 2: Transferring surface motion for acoustic analysis. The example geometry is shown in Figure 9-7b run with the below syntax.

```
BEGIN VARIABLE INTERPOLATION [<string>var_interp_name]
  TRANSFER TYPE = INTERPOLATE FROM NEAREST FACE
  SOURCE VARIABLE = SURFACE NORMAL NODAL velocity
  SOURCE SURFACE = surface_1
  TARGET VARIABLE = NODAL surface_motion
  TARGET SURFACE = surface_2
  SEARCH TOLERANCE = 1.0e-3
  PROXIMITY SEARCH TYPE = RAY SEARCH|SPHERE SEARCH
END [VARIABLE INTERPOLATION <string>var_interp_name]
```

The source surface velocities (solid red arrows) are first restricted to the surface normal (dotted red errors). These normal velocities define a velocity distribution on the source source (thin red line). The velocity distribution at each node of the target surface (green arrows) found by spatial interpolation of the source surface normal velocity. In this way the solid normal surface motion can be mapped as a boundary condition on a subsequent Salinas acoustic analysis at the exterior fluid domain. A similar boundary condition can be used to drive a variety of one way interface based fluid/structure or other code-to-code couplings.

Note that a general acoustic source mapping will include both a volume source term and a surface motion boundary condition defined in two variable interpolation blocks.



(a) Nearest element

(b) Nearest face

Figure 9-7. Variable interpolation options.

9.9. Global Output Options

These commands exist at the region scope to control the output of default global variables:

```
GLOBAL ENERGY REPORTING = EXACT|APPROXIMATE|OFF (EXACT
EXTENSIVE GLOBAL VARS   PROBLEMSIZE|CORESOLVER|MLSOLVER|CONTACT|
                         ALMATERIAL|CONSTRAINTMESH|ENERGYBALANCE|
                         NODAL|ELEMENT|ALL
                         [IN <output_name: string>]
EXTENSIVE RIGID BODY VARS OUTPUT = OFF|HISTORY|RESULTS|ALL|
                                   DEFAULT(DEFAULT)
```

Through the `GLOBAL ENERGY REPORTING` command line Sierra/SM offers three reporting options for global energy variables: `EXACT`, `APPROXIMATE`, and `OFF`. The `EXACT` and `APPROXIMATE` reporting options use different algorithms for tracking the global values of external energy, internal energy, contact energy, and hourglass energy. In many cases, the `APPROXIMATE` reporting option will provide a modest performance improvement with a negligible effect on the reported energy values. The `OFF` option will result in a further performance improvement and will report all energy values as zero.

The `GLOBAL ENERGY REPORTING` command has no effect on the analysis itself; the energy values calculated are used only for reporting purposes.

The command `EXTENSIVE GLOBAL VARS` requests that a set of global variables (dictated by the choice of the option) be written to the output file(s). An output results file may optionally be specified to which to write the variables; otherwise they will be written to all output files. Note, by default energy and momentum variables are written to all output files. Regardless of the option set here any global variable may be requested specifically by name in an output block.

The line command `EXTENSIVE RIGID BODY VARS OUTPUT` controls the default output set for global rigid body variables. Regardless of the option choice here, global rigid body variables may be output by name in the history or results output blocks. See Table 9-4 for a list of available variables. The `EXTENSIVE RIGID BODY VARS OUTPUT` options are: `OFF` to specify no default rigid body global variable output; `HISTORY` to specify default rigid body global variable output to the history file(s) only; `RESULTS` to specify default rigid body global variable output to the results file(s) only; `ALL` to specify default rigid body global variable output to both the history and results files. This option defaults to `DEFAULT` by default if the analysis contains less than ten rigid bodies then the rigid body variables will be written to results and history. If the analysis has more than ten rigid bodies then no rigid body global variable will be output by default. For an analysis with several rigid bodies it may a significant file size and performance benefit to not output the rigid body variables due to the large number of such variables.

9.10. Variables

This section lists commonly used variables that the user can be output to the results, history, or restart files. This includes global, nodal, and element variables. Material model specific output variables are described in the material model chapter 5.

9.10.1. Global, Nodal, Face, and Element Variables

This section lists commonly used variables. The variables are presented in tables based on use as follows:

- Table 9-1 Energy Variables
- Table 9-2 Other Global Variables for All Analyses
- Table 9-3 Global Variables for AL Control Contact
- Table 9-4 Global Variables for Rigid Bodies
- Table 9-5 Nodal Variables for All Analyses
- Table 9-6 Nodal Variables for Implicit Analyses
- Table 9-7 Nodal Variables for Shells
- Table 9-8 Nodal Variables for Spot Welds
- Table 9-9 Nodal Variables for Line Welds
- Table 9-10 Nodal Variables for MPCs
- Table 9-11 Nodal Variables for Contact
- Table 9-12 Element Variables for All Elements
- Table 9-13 Element Variables for Solid Elements
- Table 9-14 Element Variables for Membranes
- Table 9-15 Element Variables for Shells
- Table 9-16 Element Variables for Trusses
- Table 9-17 Element Variables for Cohesive Elements
- Table 9-18 Element Variables for Nonlinear Beams
- Table 9-19 Element Variables for Springs
- Table 9-21 Face Variables

The tables provide the following information about each variable:

Variable Name. The variable can be accessed via this name in an results output block, user output block, user subroutine, etc.

Type. This is the intrinsic variable type. The various types are listed as follows:

- `Integer`: A single integer value.
- `Integer[]`: An array of integer values.
- `Real`: A single floating point value.
- `Real[]`: An array of floating point values.
- `Vector_2D`: A two component vector value. Values will be output as `name_x` and `name_y`.
- `Vector_3D`: A three component vector value. Values will be output as `name_x`, `name_y`, and `name_z`.
- `SymTen33`: A symmetric 3x3 tensor with six unique components `name_xx`, `name_yy`,

name_zz, name_xy, name_yz, name_zx.

- FullTen36: A full 3x3 tensor with nine unique components name_xx, name_yy, name_zz, name_xy, name_yz, name_zx, name_yx, name_zy, and name_xz.

Derived. Standard variables are stored in the mesh database. A variable designed with a yes in this column is a derived variables. Derived variables are computed on the fly as a function of other variables. For example unrotated_stress is a standard variable while von_mises is a derived norm of the unrotated_stress field. Usually standard and derived variables can be used interchangeably. There are some distinctions. Derived variables cannot be set in initial conditions. Additional derived variables must be included in a BEGIN DERIVED OUTPUT command block if it is to be transferred to another procedure or region as described in Section 6.7.

Refer to Section 9.3.1.4 for details on output in the context of elements with multiple integration points.

For node-based tetrahedra the "element" variables are stored at both the elements and at the nodes; more details are available in Section 6.2.1.1. The tables of various types of variables follow.

Table 9-1. Global Energy Variables

Variable Name	Type	Description
contact_energy	Real	The energy added or subtracted from the system by contact forces. Includes frictional loses and plastic contact. Computed as the integral of contact_force dot velocity
external_energy	Real	The energy added or subtracted from the system by external forces (force, pressure, kinematic BCs, etc.) Computed as integral of force_external dot velocity
internal_energy	Real	Energy stored by material and hourglass forces. Computed as integral of internal+force dot velocity
kinetic_energy	Real	Total kinetic energy of model. Computed as sum of nodal kinetic energies
hourglass_energy	Real	The internal energy due to hourglass forces. Will be present in under-integrated elements such as the uniform gradient hexahedron. hourglass_energy is a component of internal_energy
stiffness_hourglass_energy	Real	The internal energy due to hourglass stiffness. Currently only available for the uniform gradient hexahedron, fiber shell, and fiber membrane elements

continued on next page...

. . . continued from previous page

Variable Name	Type	Description
viscous_hourglass_energy	Real	The internal energy due to hourglass viscosity. Currently only available for the uniform gradient hexahedron, fiber shell, and fiber membrane elements
strain_energy	Real	Total strain energy accumulated in all elements. Integral of stress dot strain. strain_energy is a component of internal_energy
deposited_energy	Real	Total energy deposited by PRESCRIBED ENERGY DEPOSITION boundary conditions. Generally only relevant to equation of state based material models
external_viscous_energy	Real	Viscous energy lost due to external viscous forces. At present, this includes velocity damping and mass damping (Section 7.15.5.2)
internal_viscous_energy	Real	Viscous energy lost due to internal viscous forces. At present, this only includes stiffness damping (Section 7.15.5.2)
block<blockID>_external_viscous_energy	Real	Viscous energy lost due to external viscous forces for a specific block, where <blockID> is replaced by the ID of the block requested.
block<blockID>_internal_viscous_energy	Real	Viscous energy lost due to internal viscous forces for a specific block, where <blockID> is replaced by the ID of the block requested.

Table 9-2. Other Global Variables For All Analyses

Variable Name	Type	Comments
momentum	Vector_3D	Total model momentum sum. Computed as mass dot velocity
angular_momentum	Vector_3D	Total model angular momentum sum. Includes angular momentum of both the translational and rotational degrees of freedom.
momentum_ block<blockID>_x	Real	Momentum sum in the x direction for each block, i.e., momentum_block1_x for blockID 1
momentum_ block<blockID>_y	Real	Momentum sum in the y direction for each block, i.e., momentum_block1_y for blockID 1
momentum_ block<blockID>_z	Real	Momentum sum in the z direction for each block, i.e., momentum_block1_z for blockID 1
stepcount	Real	Step count (1, 2, ..., N)
timestep	Real	Current time step
timestep_element	Real	Time step from element estimator
timestep_nodal	Real	Time step from nodal estimator
timestep_material	Real	Time step from material model
timestep_lanczos	Real	Time step from Lanczos estimator
timestep_powermethod	Real	Time step from power method estimator
wall_clock_time	Real	Accumulated wall clock run time in seconds
wall_clock_time_per_ step	Real	Wall clock time for last time step in seconds
cpu_time	Real	Accumulated CPU time in seconds
cpu_time_per_step	Real	CPU time for last time step in seconds
ghost_count	Real	Total number of off-processor nodes and elements at any given time. If ghost count is growing substantially over time it may indicate that the analysis could be sped up via an automated rebalance.

Table 9-3. Global Variables for Augmented Lagrange Control Contact (See Section 4.5.2)

Variable Name	Type	Comments
num_interactions	Int	Number of possible interactions that were found in contact search
num_released	Int	Number of interactions that are in a released state
num_captured	Int	Number of interactions that are in a captured state
num_dubious	Int	Number of interactions that are in a dubious state
max_normal_gap	Real	Maximum normal gap among all captured and dubious interactions
max_normal_relative_gap	Real	Maximum normal gap divided by interaction tolerance among all captured and dubious interactions
max_tangential_gap	Real	Maximum tangential gap among all captured and dubious interactions
LM_norm	Real	Magnitude of the Lagrange multiplier
LM_last_relative_change	Real	Relative change in LM_norm over the last control contact iteration

Table 9-4. Global Variables for Rigid Bodies. (See Section 9.9 for default output options.)

Variable Name	Type	Comments
a<x|y|z>_<rb_name>	Real	Translational acceleration
vel<x|y|z>_<rb_name>	Real	Translational velocity
displ<x|y|z>_<rb__name>	Real	Translational displacement
rota<x|y|z>_<rb__name>	Real	Rotational acceleration
rotv<x|y|z>_<rb__name>	Real	Rotational velocity
rotd<x|y|z>_<rb__name>	Real	Rotational displacement
react<x|y|z>_<rb_name>	Real	Translational reaction
rreact<x|y|z>_<rb__name>	Real	Rotational reaction
qvec<1|2|3|4>_<rb__name>	Real	Unit quaternion
Example: ax_rb1 is the x translational acceleration for the rigid body specified in a BEGIN RIGID BODY rb1 block where rb1 is a name chosen by the user.		

Table 9-5. Nodal Variables for All Analyses

Variable Name	Type	Comments
model_coordinates	Vector_3D	Original coordinates of nodes
coordinates	Vector_3D	Current coordinates of nodes
displacement	Vector_3D	Total displacement, generally coordinates minus model_coordinates
velocity	Vector_3D	Velocity of nodes, derivative of displacement
acceleration	Vector_3D	Acceleration of nodes, derivative of velocity
force_internal	Vector_3D	Force produced by element and material response. Also includes hourglass and other element stabilization forces.
force_external	Vector_3D	Force produced by external loading. Force, pressure, contact force, as well as reaction forces of boundary conditions.
force_inertial	Vector_3D	Force that is equivalent to the nodal mass times the nodal acceleration (Implicit only)
force external_transferred	Vector_3D	Force transferred from another physics (coupled problems only)
force_contact	Vector_3D	Force produced by contact
reaction	Vector_3D	Force produced by kinematic boundary conditions such as fixed displacement
mass	Real	Mass of each node
nodal_time_step	Real	Nodal stable time step (explicit control modes, coarse mesh only)
node_id	Integer	Global ID of each node
quaternion	Real	Current quaternion (rigid body reference nodes only)
node_id	Int	Exodus global id of the node



Table 9-6. Nodal Variables for Implicit Analyses

Variable Name	Type	Comments
displacement_increment	Vector_3D	Change in displacement over the current step
residual	Vector_3D	Nodal free body force imbalance at current time step

Table 9-7. Nodal Variables for Shells and Beams

Variable Name	Type	Comments
rotational_displacement	Vector_3D	Total spin of each node
rotational_velocity	Vector_3D	Spin rate of each node, derivative of rotational_displacement
rotational_acceleration	Vector_3D	Spin acceleration of each node, derivative of rotational_velocity
moment_internal	Vector_3D	Nodal moment from element and material response
moment_external	Vector_3D	Applied external moment, for example from a "PRESCRIBED MOMENT" boundary condition
moment external_ transferred	Vector_3D	Moment transferred from another physics (coupled problems only)
moment_inertial	Vector_3D	Nodal moment from nodal rotational mass times the nodal rotational acceleration (Implicit Only)
rotational_reaction	Vector_3D	Applied moment to enforce a kinematic boundary condition such as "PRESCRIBED ROTATION"
rotational_mass	Real	Nodal inertia. A single value, nodal inertia assumed spherical.



Table 9-8. Nodal Variables for Spot Welds

Variable Name	Type	Comments
spot_weld_parametric_coordinates	Vector_2D	Parametric coordinates defining node location on face
spot_weld_normal_force_at_death	Real	Value of force normal to face when spot weld breaks
spot_weld_tangential_force_at_death	Real	Value of force tangential to face when spot weld breaks
spot_weld_death_flag	Integer	alive = 0, dead = FAILURE DECAY CYCLES (default is 10), -1 = no spot weld constructed at this node
spot_weld_scale_factor	Real	Nodal influence area of current node
spot_weld_normal_displacement	Real	Current displacement of weld normal to face
spot_weld_tangential_displacement	Real	Current displacement of weld tangential to face
spot_weld_normal_force	Real	Current force of weld normal to face
spot_weld_tangential_force	Real	Current force of weld tangential to face
spot_weld_stiffness	Real	Current stiffness of weld
spot_weld_norm_stiffness	Real	Current stiffness of weld normal to face
spot_weld_tang_stiffness	Real	Current stiffness of weld tangential to face
spot_weld_initial_offset	Vector_3D	The initial offset of the spot weld node from the spot weld surface. Does not change over time, only output if IGNORE INITIAL OFFSET = YES is specified at input.
spot_weld_initial_normal	Vector_3D	The initial normal of the spot weld surface at the point of interaction. Only output if IGNORE INITIAL OFFSET = YES is specified at input.



Table 9-9. Nodal Variables for Line Welds

Variable Name	Type	Comments
line_weld_force_applied	Vector_3D	The force applied to the nodes of the beam and the face of the shell it is connected to
line_weld_moment_applied	Vector_3D	The moment applied to the nodes of the beam and the face of the shell it is connected to



Table 9-10. Nodal Variables for MPCs

Variable Name	Type	Comments
force_constraint	Vector_3D	The force applied to the nodes in the MPC
moment_constraint	Vector_3D	The moment applied to the nodes in the MPC
constraint_mass	Real	The mass used to compute the force applied to the node
constraint_rotational_mass	Real	The rotational mass used to compute the moment applied to the nodes with rotational DOFs
status_constraint_flag	Integer	0 = node not in MPC, 1 = Node is in one or more MPCs and among which it is always a side A MPC node, 2 = Node is in one or more MPCs and it is a side B node in at least one MPC

Table 9-11. Nodal Variables for Contact (See Section 8.10)

Variable	Type	Description
contact_strength	Real	Strength of contact interactions at the node, computed as nodal contact_area divided by surface_area. Nominal range is between 0.0 (no contact) and 1.0 (fully in contact). Values above 1.0 may be reported in some circumstances. Generally contact strength values between 0.85 and 1.15 will be seen in fully overlapped zones and the contact strength will fade to zero near the edge of contact zones.
contact_status	Real	Status of the interactions at the node. Possible values are as follows: 0.0 = Node is not a contact node (not in a defined contact surface) 0.5 = Node is not in contact For explicit contact: 1.0 = Node is in contact For implicit contact: 1.0 = Node is in contact and is slipping -1.0 = Node is in contact and is sticking (celement).
contact_friction_index	Int	Records what friction model is being used by a node. The index number at the node can be compared to a table printed to the log file to determine what friction model name and type a given node is using. Note, a node in multiple interactions will report the friction model associated with the strongest interaction.
contact_normal_direction	Vector_3D	Direction of the constraint. This is, in general, the normal of the face in the interaction (cdirnor).
force_contact	Vector_3D	Force applied by contact.
contact_tangential_direction	Vector_3D	Direction of the contact tangential force (cdirtan).
contact_normal_force_magnitude	Real	Magnitude of the contact force at the node in the direction normal to the contact face. Magnitude of contact_normal_direction.

continued on next page...

. . . continued from previous page

Variable	Type	Description
contact_tangential_force_magnitude	Real	Magnitude of the contact force at the node in the plane of the contact face. Magnitude of contact_tangential_direction.
contact_normal_traction_magnitude	Real	Contact traction normal to the contact face. contact_normal_force_magnitude divided by contact_area. If there are multiple interactions for this node, the traction only for the last interaction is given. Negative for compression tractions, positive for tensile tractions.
contact_tangential_traction_magnitude	Real	Traction in the plane of the contact face. contact_traction_force_magnitude divided by contact_area. If there are multiple interactions for this node, the traction only for the last interaction is given. This value is generally negative as it opposes slip.
contact_incremental_slip_magnitude	Real	Magnitude of incremental slip over the current time step (cdtan).
contact_incremental_slip_direction	Vector_3D	Normalized direction of incremental slip over the current time step (cdirislp).
contact_accumulated_slip	Real	Magnitude of tangential slip accumulated over the entire analysis. This is the distance along the slip path, and not the magnitude of contact_accumulated_slip_vector (cstan).
contact_accumulated_slip_vector	Vector_3D	Total accumulated tangential slip over the entire analysis (cdirslp).
contact_frictional_energy	Real	Accumulated amount of frictional energy dissipated over the entire analysis.
contact_frictional_energy_density	Real	Accumulated amount of frictional energy dissipated over the entire analysis, divided by the nodal surface area (cetan).
contact_area	Real	Contact area for the node. This is the tributary area around the node for this interaction. For multiple interactions, the reported area is the area associated with the last interaction (carea).
surface_area	Real	Exposed surface area for the node. This is the tributary area around the node.

continued on next page. . .

. . . continued from previous page

Variable	Type	Description
contact_normal_gap	Real	Magnitude of gap in the direction normal to the face (cgnor).
contact_tangential_gap	Real	Magnitude of gap in the direction tangent to the face (only applicable for compliant friction models) (cgtan).
removed_overlap	Vector_3D	Direction and magnitude of removed overlap (only applicable for explicit).
contact_energy	Real	Work done by contact on a node by node basis. Positive values indicate contact added energy to the node, negative subtracted energy from the node.

Table 9-12. Element Variables for All Elements. *indicates a derived variable; refer to Section 6.7 for additional details.

Variable	Type	Comments
diagonal_ratio	Real	See Section 2.5
element_mass	Real	Total mass of element
element_id	Integer	Global ID of the element
perimeter_ratio	Real	See Section 2.5
solid_angle	Real	See Section 2.5
timestep	Real	Critical time step for the element. The element in the model with the smallest time step controls the analysis time step.
von_mises*	Real	Von Mises stress norm
hydrostatic_stress*	Real	One-third the trace of the stress tensor
fluid_pressure*	Real	Negative of hydrostatic_stress
stress_invariant_1*	Real	I_1, Trace of the stress tensor
stress_invariant_2*	Real	I_2, Second invariant of the stress tensor
stress_invariant_3*	Real	I_3, Third invariant of the stress tensor
stress_invariant_j2*	Real	J_2, Second invariant of the deviatoric stress tensor
stress_invariant_j3*	Real	J_3, Third invariant of the deviatoric stress tensor
triaxiality*	Real	Ratio of hydrostatic pressure to Von Mises stress
lode_angle*	Real	$\frac{1}{3}\arccos(\frac{3}{2}\sqrt{3}(J_3/J_2^{3/2}))$
principal_stresses*	Vector_3D	All three eigenvalues of the stress tensor sorted smallest to largest
max_principal_stress*	Real	Largest eigenvalue of the stress tensor
intermediate_principal_stress*	Real	Middle eigenvalue of the stress tensor
min_principal_stress*	Real	Smallest eigenvalue of the stress tensor
max_shear_stress*	Real	Maximum shear stress from Mohr's circle
octahedral_shear_stress*	Real	Octahedral shear norm of the stress tensor
internal_energy	Real	Total internal energy for the element
hourglass_energy	Real	Total hourglass energy for the element
stiffness_hourglass_energy	Real	The internal energy due to hourglass stiffness. Currently only available for the uniform gradient hexahedron, fiber shell, and fiber membrane elements.
viscous_hourglass_energy	Real	The internal energy due to hourglass viscosity. Currently only available for the uniform gradient hexahedron, fiber shell, and fiber membrane elements.

continued on next page...

. . . continued from previous page

Variable	Type	Comments
nonmaterial_strain_3d	SymTen33	Total thermal and artificial strain (engineering strain) applied to the element
nonmaterial_strain_rate_3d	SymTen33	Current nonmaterial log strain rate applied to the element
thermal_strain_rate_3d	SymTen33	Current thermal log strain rate applied to the element
artificial_strain_rate_3d	SymTen33	Current artificial log strain rate applied to the element
processor_id	Int	Parallel processor rank that owns the element
element_id	Int	Exodus global ID of the element
material_direction_1*	Vector_3D	Material direction 1 for orthotropic materials
material_direction_2*	Vector_3D	Material direction 2 for orthotropic materials
material_direction_3*	Vector_3D	Material direction 3 for orthotropic materials
strain_energy_density	Real	Volumetric material strain energy. Integral of material stress dotted with material strain. Note the strain_energy_density resultant removes the energy associated with thermal strains and artificial bulk viscosity that are included in the deformation_energy_density
strain_energy	Real	Total strain energy, strain energy times element volume

Table 9-13. Element Variables for Solid Elements

Variable	Type	Comments
aspect_ratio	Real	Tetrahedra only. See Section 2.5
inradius	Real	Tetrahedra only. See Section 2.5
normalized_inradius	Real	10-node tetrahedra only. See Section 2.5
dilmod	Real	Dilatational modulus
nodal_jacobian_ratio	Real	Hex and Quad only. See Section 2.5
scaled_jacobian	Real	Hex only. See Section 2.5
element_shape	Real	Element shape quality metric. See Section 2.5
rate_of_deformation	SymTen33	Unrotated rate of deformation tensor, **D** (Hexahedra and node-based tetrahedra only)
principal_rates_of_deformation	Vector_3D	Principal rates of deformation (stretching)
max_principal_rate_of_deformation	Real	Maximum principal rate of deformation (stretching)
intermediate_principal_rate_of_deformation	Real	Intermediate principal rate of deformation (stretching)
min_principal_rate_of_deformation	Real	Minimum principal rate of deformation (stretching)
max_principal_rate_of_deformation_direction	Vector_3D	Direction of the maximum principal rate of deformation (stretching)
intermediate_principal_rate_of_deformation_direction	Vector_3D	Direction of the intermediate principal rate of deformation (stretching)
min_principal_rate_of_deformation_direction	Vector_3D	Direction of the minimum principal rate of deformation (stretching)
deformation_gradient	FullTen36	Deformation gradient tensor, **F**
right_stretch	SymTen33	Right stretch tensor, **U**
left_stretch	SymTen33	Left stretch tensor, **V**
rotation	FullTen36	Rotation tensor, **R**
shrmod	Real	Shear modulus
stress	SymTen33	Cauchy stress tensor in global configuration. This is a single volume averaged value for multiple integration point elements.
cauchy_stress	SymTen33	Cauchy stress tensor in the global configuration. This value is present at each integration point.
unrotated_stress	SymTen33	Cauchy stress in the unrotated material configuration. This value is present at each integration point.

continued on next page...

. . . continued from previous page

Variable	Type	Comments
max_principal_stress_direction	Vector_3D	Eigenvector of the Cauchy stress tensor corresponding to the largest eigenvalue
intermediate_principal_stress_direction	Vector_3D	Eigenvector of the Cauchy stress tensor corresponding to the middle eigenvalue
min_principal_stress_direction	Vector_3D	Eigenvector of the Cauchy stress tensor corresponding to the smallest eigenvalue
green_lagrange_strain	SymTen33	Green-Lagrange strain tensor, $\mathbf{E} = (1/2)(\mathbf{F}^T \cdot \mathbf{F} - \mathbf{I})$
principal_green_lagrange_strains	Vector_3D	All three eigenvalues of the Green-Lagrange strain tensor sorted smallest to largest
max_principal_green_lagrange_strain	Real	Largest eigenvalue of the Green-Lagrange strain tensor
intermediate_principal_green_lagrange_strain	Real	Middle eigenvalue of the Green-Lagrange strain tensor
min_principal_green_lagrange_strain	Real	Smallest eigenvalue of the Green-Lagrange strain tensor
max_principal_green_lagrange_strain_direction	Vector_3D	Eigenvector of the Green-Lagrange strain tensor corresponding to the largest eigenvalue
intermediate_principal_green_lagrange_strain_direction	Vector_3D	Eigenvector of the Green-Lagrange strain tensor corresponding to the middle eigenvalue
min_principal_green_lagrange_strain_direction	Vector_3D	Eigenvector of the Green-Lagrange strain tensor corresponding to the smallest eigenvalue
green_lagrange_strain_rate	SymTen33	Green-Lagrange strain rate tensor, $\dot{\mathbf{E}} = \mathbf{F}^T\mathbf{DF}$
principal_green_lagrange_strain_rates	Vector_3D	All three eigenvalues of the Green-Lagrange strain rate tensor sorted smallest to largest
max_principal_green_lagrange_strain_rate	Real	Largest eigenvalue of the Green-Lagrange strain rate tensor
intermediate_principal_green_lagrange_strain_rate	Real	Middle eigenvalue of the Green-Lagrange strain rate tensor
min_principal_green_lagrange_strain_rate	Real	Smallest eigenvalue of the Green-Lagrange strain rate tensor
max_principal_green_lagrange_strain_rate_direction	Vector_3D	Eigenvector of the Green-Lagrange strain rate tensor corresponding to the largest eigenvalue

continued on next page. . .

. . . continued from previous page

Variable	Type	Comments
intermediate_principal_green_lagrange_strain_rate_direction	Vector_3D	Eigenvector of the Green-Lagrange strain rate tensor corresponding to the middle eigenvalue
min_principal_green_lagrange_strain_rate_direction	Vector_3D	Eigenvector of the Green-Lagrange strain rate tensor corresponding to the smallest eigenvalue
biot_strain	SymTen33	Biot Strain tensor, $\mathbf{U} - \mathbf{I}$
log_strain	SymTen33	Log strain tensor, $\ln \mathbf{V}$
unrotated_log_strain	SymTen33	Log strain tensor in unrotated configuration, $\ln \mathbf{U}$
effective_log_strain	Real	Effective log strain
log_strain_invariant_1	Real	Trace of the log strain tensor
log_strain_invariant_2	Real	Second invariant of the log strain tensor
log_strain_invariant_3	Real	Third invariant of the log strain tensor
max_principal_log_strain	Real	Largest eigenvalue of the log strain tensor
intermediate_principal_log_strain	Real	Middle eigenvalue of the log strain tensor
min_principal_log_strain	Real	Smallest eigenvalue of the log strain tensor
max_principal_strain_direction	Vector_3D	Eigenvector of the log strain tensor corresponding to the largest eigenvalue
intermediate_principal_strain_direction	Vector_3D	Eigenvector of the log strain tensor corresponding to the middle eigenvalue
min_principal_strain_direction	Vector_3D	Eigenvector of the log strain tensor corresponding to the smallest eigenvalue
max_shear_log_strain	Real	Maximum shear log strain from Mohr's circle
octahedral_shear_log_strain	Real	Octahedral strain norm of the log strain tensor
volume	Real	Element volume
internal_energy_density	Real	Element total internal energy divided by current element volume
hourglass_energy_density	Real	Element hourglass energy divided by current element volume
deformation_energy_density	Real	internal_energy_density minus hourglass_energy_density. Equivalent to the integral of total element stress dotted with total element deformation.
iplocation	Vector_3D	Location of each element integration points in the original model coordinates
centroid	Vector_3D	Location element mass-centroid in the original model coordinates

Table 9-14. Element Variables for Membranes

Variable Name	Type	Comments
memb_stress	SymTen33	Cauchy membrane stress tensor in the global configuration.
element_area	Real	Surface area of the element
element_thickness	Real	Thickness provided to element

Table 9-15. Element Variables for Shells. *indicates a derived variable; consult Section 6.7 for additional details. †indicates a variable which is available for analytic integration; refer to Section 6.2.5.

Variable Name	Type	Comments
memb_stress*,†	SymTen33	Cauchy stress at mid plane in global X, Y, and Z coordinates.
bottom_stress*,†	SymTen33	Cauchy stress at bottom integration point in global X, Y, and Z coordinates
top_stress*,†	SymTen33	Cauchy stress at top integration point in global X, Y, and Z coordinates
unrotated_stress	SymTen33	Cauchy stress in the unrotated material configuration. This value is present at each integration point. ZZ component is undefined.
transform_shell_stress*	SymTen21	In-plane shell stress
strain	SymTen33	Integrated strain at mid plane in local shell coordinate system
effective_strain*	Real	Effective strain norm
strain_invariant_1*	Real	Trace of the strain tensor
strain_invariant_2*	Real	Second invariant of the strain tensor
strain_invariant_3*	Real	Third invariant of the strain tensor
max_principal_strain*	Real	Largest eigenvalue of the strain tensor
intermediate_principal_strain*	Real	Middle eigenvalue of the strain tensor
min_principal_strain*	Real	Smallest eigenvalue of the strain tensor
max_shear_strain*	Real	Maximum shear strain from Mohr's circle
octahedral_shear_strain*	Real	Octahedral strain norm of the strain tensor
transform_shell_strain*	Real	In-plane shell strain
element_area	Real	Element surface area
element_thickness	Real	Element current thickness
rate_of_deformation	SymTen33	Rate of deformation (stretching) tensor
thickness_to_length_ratio*,†	Real	Ratio of shell thickness to in-plane element size. Shells with ratios beyond 1.0 may encounter stability issues

Table 9-16. Element Variables for Trusses

Variable Name	Type	Comments
init_length	Real	Initial length of the truss element
truss_strain	Real	Axial strain in the truss element
unrotated_stress	SymTen33	Axial Cauchy stress is stored in unrotated_stress_xx. All other components are zero. See Section 6.2.9 for more details.
truss_force	Real	Axial force in the truss

Table 9-17. Element Variables for Cohesive Elements

Variable Name	Type	Comments
cse_traction	Vector_3D	_z component is normal direction. _x and _y components are tangential directions.
cse_separation	Vector_3D	_z component is normal direction. _x and _y components are tangential directions.
cse_initial_trac	Vector_3D	Available only if traction initialization is used. _z component is normal direction. _x and _y components are tangential directions.
cse_activated	Integer	For intrinsic elements
cse_initial_area	Real	Initial area of the cohesive element
cse_avg_normal_dir	Vector_3D	Averaged normal direction over the element integration point locations
cse_avg_t1_dir	Vector_3D	Averaged tangent 1 direction over the element integration point locations
cse_avg_t2_dir	Vector_3D	Averaged tangent 2 direction over the element integration point locations
cse_local_force	Vector_3D	The element local force z is normal force, x and y are tangential forces.

Table 9-18. Element Variables for Nonlinear Beams

Variable Name	Type	Comments
beam_strain_axial	Real	Axial strain at each integration point.
beam_strain_shear	Real	Shear torsional strain at each integration point.
beam_strain_inc	Vector_2D	Strain increments at each integration point. Axial strain increments are _x and shear torsional strain increments are _y.
beam_stress_axial	Real	Axial Cauchy stress at each integration point.
beam_stress_shear	Real	Shear circumferential torsional stress at each integration point.
unrotated_stress	SymTen33	Cauchy stresses at each integration point. Only the _xx and _xy values per integration point contain actual stress values. Axial stresses are in _xx and circumferential torsional shear stresses are in _xy. See Section 6.2.8 for more details.
beam_axial_force	Real	Axial force at midpoint.
beam_transverse_force_s	Real	Transverse shear in *s*-direction at midpoint.
beam_transverse_force_t	Real	Transverse shear in *t*-direction at midpoint.
beam_moment_r	Real	Torsion at midpoint.
beam_moment_s	Real	Moment about *s*-direction at midpoint.
beam_moment_t	Real	Moment about *t*-direction at midpoint.
beam_avg_rate_of_def	Real	Average rate of deformation over all integration point.
current_area	Real	Current area of the beam cross-section.



Table 9-19. Element Variables for Springs

Variable Name	Type	Comments
spring_force	Real	Magnitude of the internal spring force.
spring_engineering_strain	Real	Change in length over initial length $\frac{dL}{L_0}$.
spring_init_length	Real	Initial spring length, L_0.



Table 9-20. Element Variables for Line Welds

Variable Name	Type	Comments
line_weld_force	Vector_3D	The force (in units force-per-unit-length) the weld is applying in the global XYZ coordinate system
line_weld_moment	Vector_3D	The moment (in units moment-per-unit-length) the weld is applying in the global XYZ coordinate system
line_weld_force_rst	Vector_3D	The force (in units force-per-unit-length) the weld is applying in the weld local RST coordinate system
line_weld_moment_rst	Vector_3D	The moment (in units moment-per-unit-length) the weld is applying in the weld local RST coordinate system
line_weld_weld_active	Integer	A flag on the beam elements of the line weld that states the weld is active or not 1 is active 0 is inactive due to the weld reaching the failure criterion, -1 is inactive due to the beam element not being found, -2 is inactive due to the connecting faces not being found, -3 is inactive due to the shells the weld is connecting not being found, -4 is inactive for any other reason
line_weld_initial_weld_length	Real	The original length of the beam element of the line weld. This length is used in computing the nodal forces to apply
line_weld_death_flag	Integer	A flag on the beam elements of the line weld that designates that portion of the weld has died (0 is alive; 1 is dead)
line_weld_death_step	Integer	The current death step of a beam element that is in the process of dying
line_weld_force_at_death	Real	A real array of size 12 that is the 3 force and moment components for each node of the beam. This value is zero as long as the beam element is alive, and is set to the current nodal forces/moments once the failure criteria is reached

Table 9-21. Face Variables

Variable Name	Type	Comments
`pressure`	`Real`	Pressure applied to faces via a pressure boundary condition

9.10.2. Variables for Material Models

State variables can be output from the material models. Most of the materials, with the exception of simple models such as the elastic model, have state variables. The state variables for material models in LAMÉ are accessible directly by name. For instance, the equivalent plastic strain variable is accessible by the name `EQPS` for all elastic-plastic material models. State variables can also be accessed by index.

Available LAMÉ state variable names for a material will also be listed in the log file from a run that uses the material model.

Tables of available material state variables for commonly used material models are provided in the Materials Chapter 5. Note, some models produce slightly different state variable sets for solids versus shells versus beams as described in the material model documentation.

9.10.3. Energy Output Variables

There are several variables available for monitoring energy throughout an analysis. Global energy variables will generally satisfy the relationship

$$IE + KE = EE, \tag{9.2}$$

with IE, KE, and EE denoting internal, kinetic, and external energies, respectively. Internal and external energies are generally computed incrementally through numerical integration of either internal or external nodal force vector dotted with the incremental nodal displacement vector. Kinetic energy is computed as $1/2m(\mathbf{v} \cdot \mathbf{v})$ with m as nodal mass (including mass scaling) and $\mathbf{v}$ as nodal velocity.

Internal energies come from the following sources:

- Strain energy due to material stresses (`strain_energy`). Note that dissipated energy due to plastic deformation or material viscosity is also included, so `strain_energy` may be nonzero even in an unloaded state.
- Hourglass forces including hourglass viscosity and hourglass stiffness (`hourglass_energy`)
- Bulk viscosity forces including linear and quadratic terms (not active in implicit analyses by default). No output terms are currently available to monitor energy due to bulk viscosity directly.

External energies come from the following sources (may not be exhaustive):

- Applied external forces via boundary conditions (pressures, forces, moments, gravity, tractions, etc.).
- Contact forces (`contact_energy`); expected to be nonzero only for frictional contact.
- Viscous damping due to Rayleigh damping (mass or stiffness proportional) or due to velocity damping.
- ARS damping forces.

Energy added to induce thermal and artificial strains is not explicitly accounted for, so simulations that use these features may experience energy imbalance issues. For example, a fully-constrained

body experiencing thermal expansion will have a nonzero stress state and thus nonzero strain energy, but zero `internal_energy` due to the fact that no work has been done due to internal or external forces (nodal displacements are zero). This inconsistency is because strain energy is computed from the inner product of Cauchy stress and rate of deformation, but internal energy is computed from the inner product of nodal force and displacement (zero).

Note that a few hyperelastic material models output the value of their strain energy potential function directly into the `strain_energy` and `strain_energy_density` variables.

Note that global energies outputs are generally computed as a summation of element or nodal energy quantities, and thus element blocks designated as `INACTIVE` will not contribute to the energy outputs.

Additional information may be found in the theory and verification manuals.



Known Issue: Strain energy may not be consistent with internal energy in analyses with artificial or thermal strain.



Known Issue: Strain energy reported by element blocks using both selective deviatoric elements and a hyper- elastic material model may differ slightly from internal energy. This is due to a known issue with how the selective-deviatoric element volume averages element quantities. The Total Lagrange element formulation with volume averaging of the Jacobian may provide a better alternative in cases where this is an issue.

9.11. Solution Verification

Solution verification is the capability to enable the code to check a computed solution versus some reference solution and abort the run if an anomaly is detected. Solution verification is primarily used by the code development team to create automated verification tests for various capabilities. Solution verification can be of use to the code end user to help write tests to verify that a user written subroutine or expression is behaving as expected. Solution verification could also be used to stop a run that has entered an unphysical or unneeded regime.

```
BEGIN SOLUTION VERIFICATION
  VERIFY GLOBAL <string>gVar <|<=|=|>|>=
    <real>gVal [PLUS OR MINUS <real>Tol]
  VERIFY GLOBAL <string>gVar <|<=|=|>|>=
    GLOBAL <string>gVar2 [PLUS OR MINUS <real>Tol]
  VERIFY GLOBAL <string>gVar <|<=|=|>|>=
    FUNCTION <string>funcName
  VERIFY NODAL  <string>nVar
    AT NODE <integer>nodeNum <|<=|=|>|>=
    FUNCTION <string>funcName
  VERIFY NODAL  <string>nVar
    AT NODE <integer>nodeNum <|<=|=|>|>=
    <real>val
  VERIFY ELEMENT <string>eVar
    AT ELEMENT <integer>elemNum <|<=|=|>|>=
    <real>val
  VERIFY ELEMENT <string>eVar
    AT ELEMENT <integer>elemNum <|<=|=|>|>=
    FUNCTION <string>funcName

  # Absolute tolerance
  TOLERANCE = <real>absTol
  # Relative tolerance
  RELATIVE TOLERANCE = <real>relTol
  # Time tolerance
  TIME TOLERANCE = <real>timeTol

  SKIP TIMES = <real>time0 TO <real>time1

  COMPLETION FILE = <string>fileName
END
```

The various forms of the `VERIFY GLOBAL`, `VERIFY NODAL`, `VERIFY ELEMENT` commands check that the named variable has some relation to a defined constant, time history function, or other global variable. For example if creating and debugging a user subroutine based displacement function it may be helpful to ensure that the displacement at some node tracks with the expected analytic displacement given in a time history function. The global variables used in

solution verification are often created via a `USER OUTPUT` based command block. If at any time step at which the solution is verified the `VERIFY` statement fails, the analysis will immediately abort and report an error.

Tolerances of how close a verification needs to be acceptable can be specified. The `TOLERANCE` and `PLUS OR MINUS` verify option both specify absolute tolerances. For example the command:

```
VERIFY GLOBAL disp1 = 1.0 PLUS OR MINUS 0.1
```

will verify successfully as long as disp1 is between 0.9 and 1.1.

The `RELATIVE TOLERANCE` command specifies an acceptable relative error between values being verified. A verification is successful if:

$$\frac{|gVal - gVar|}{max(|gVal|, |gVar|)} < relTol \tag{9.3}$$

The `TIME TOLERANCE` is used with the time history function based verify commands. A verification will pass if the `gVar` is within tolerance to any value of the function that would be computed at the current time plus or minus the `timeTol`. The time tolerance command is useful for verifying functions that have sharp jumps in time.

The `SKIP TIMES` command is used to skip verification over the defined time periods. The skip times command can be used to focus the verification on only the time periods where the solution is known.

The `COMPLETION FILE` command causes solution verification to output a status file named `fileName` at successful completion of the code run. Existence of the status file indicates that the run completed as expected. If the run dies due to failing a solution verification, runs into a queue time limit, or other unexpected error, the status file will not be written.

9.12. Identify Fragments

The IDENTIFY FRAGMENTS capability provides data on unique contiguous mesh fragments in a simulation. Fragments are identified based on mesh connectivity. The fragment identification algorithm accounts for topological changes including element death, element separation in an XFEM simulation. Fragment identification element field elem_fragment_id and nodal field node_fragment_id indicate the fragment number for each element and node in the simulation, respectively, and may be output in a results block, allowing for fragment visualization and other post-processing operations. Fragment numbers range from 1 to the total number of fragments, sorted in order of increasing fragment mass. Additional fragment data, including fragment mass and momentum, may be written to an optional output text file.

A nodal vector field fragment_force is provided to compute force and moment on a per-fragment basis. If using an applied pressure (Section 7.6.1), the fragment_force field can be used as the external force contribution output name. In a Zapotec coupled analysis the field force_to_apply might coupled with fragment_force to report force and moments on the fragments. Otherwise, fragment_force can be populated through user output methods.

Fragment identification is controlled by the BEGIN IDENTIFY FRAGMENTS block in the region scope:

```
BEGIN IDENTIFY FRAGMENTS
   OUTPUT FILE = <string>output_filename
   INCREMENT = <real>increment
   MINIMUM FRAGMENT SIZE = <int>min_element_threshold
   START TIME = <real>
END [IDENTIFY FRAGMENTS]
```

The optional OUTPUT FILE command triggers output to a text file named output_filename containing fragment data. The data written to the text file include the mass, momentum, center of mass, moment of inertia, angular momentum, and names of element blocks present in each fragment. The optional MINIMUM FRAGMENT SIZE command groups fragments containing fewer than min_element_threshold elements into a single fragment with ID 0. This feature is useful, for example, in a brittle fracture simulation that results in a significant number of isolated elements. By default, the fragment identification algorithm is executed once per output step. If the optional INCREMENT command is included in the IDENTIFY FRAGMENTS block, the fragment identification algorithm is executed only at the corresponding simulation times. For example, if INCREMENT is assigned a value of 0.23, the fragment identification algorithm will be executed only when the simulation time reaches a multiple of 0.23. Limiting the number of times the fragment identification algorithm is executed reduces the associated computational expense. The optional command START TIME controls when the fragment data will start to be printed to the output file.



Warning: Fragment identification is performed based only on node-to-element connectivity. If two elements share a node, those two elements are in the same fragment. Other mechanisms to connect elements, such as tied contact, MPCs, spot welds, etc., are not considered in fragment identification at this time.

9.13. Detect Breach

The breach detection capability enables automatic detection of a breach creating an open path between two user-defined surfaces via element. This capability can be used to predict when venting of a pressure vessel begins due to an open crack. The `BEGIN DETECT BREACH` block is defined in the region scope with the following command block:

```
BEGIN DETECT BREACH <string>breachName
  SURFACE1 = <string list>surface1_names
  SURFACE2 = <string list>surface2_names
END [DETECT BREACH]
```

The `breachName` string is required, as it defines a global variable with the same name that initializes to 0 and changes to 1 when breach is detected. Within the `DETECT BREACH` block, two line command are required: `surface1 = sideset1 sideset2 ...` and `surface2 = sideset3 sideset4 ...`. These command lines set the "inner" and "outer" surfaces for detecting breach between. An arbitrary number of detect breach blocks are allowed.

The algorithm is implemented by identifying clusters of dead elements. If two dead elements share a node, then they are considered to be in the same cluster. (An alternative implementation could consider dead elements to be in the same cluster only when they share a face; however, that is not the version currently implemented.) If a cluster of dead elements has an adjacent face in both surface1 and surface2, then we say breach has occurred between those surfaces(see Figure 9-8).

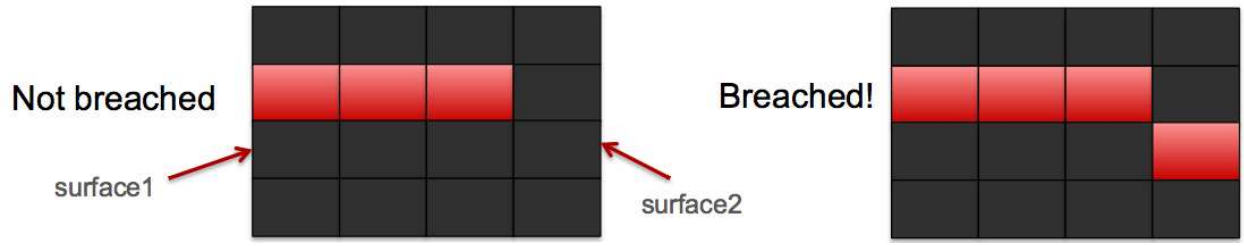


Figure 9-8. Breach detection between surfaces 1 and 2. Dead elements in red.

The death cluster id number can be output as either an element or nodal variable named `dead_element_cluster_id`. The nodal variable version is only used for implementing the clustering algorithm, but the element variable `dead_element_cluster_id` can be output and used by analyst scripts to compute quantities of interest for the dead element clusters such as the volume, minimum cross-sectional area, moments of inertia, etc.



Warning: This capability is currently only tested for bulk 3D elements such as hexahedra and tetrahedra. The capability will not work with shells. Additionally, the capability has no knowledge of tied contacts; thus a breach can only propagate between mesh-contiguous elements.



Warning: If using detect breach variables such as the `dead_element_cluster_id` element variable in a user output you must add "process elements = all" to your user output block.

References

[1] L.A. Schoof and V.R. Yarberry. EXODUS II: A finite element data model. Technical Report SAND92-2137, Sandia National Laboratories, Albuquerque, NM, September 1994. pdf. See also documentation available at EXODUS II Sourceforge page. link.

[2] G.D. Sjaardema. APREPRO: An algebraic preprocessor for parameterizing finite element analyses. Technical Report SAND92-2291, Sandia National Laboratories, Albuquerque, NM, 1999. pdf.

[3] Sierra/SolidMechanics Team. Sierra/SolidMechanics VOTD Theory Manual. Technical Report In draft, Sandia National Laboratories, Albuquerque, NM, 2019.

[4] A. Mota, W. Sun, J.T. Ostien, J.W. Foulk III, and K.N. Long. Lie-group interpolation and variational recovery for internal variables. *Computational Mechanics*, 52(6):1281–1299, 2013.

10. USER SUBROUTINES

User-defined subroutines is a functionality shared by Adagio and Presto. This chapter discusses when and how to use user-defined subroutines. There are examples of user-defined subroutines in the latter part of this chapter. Some of the examples are specific to a given analysis type, i.e., they are applicable to Presto rather than Adagio or vice versa. All examples, regardless of their applicability, do provide important information about how to use the command options available for user-defined subroutines.

In the introductory part of Chapter 10, we first describe, in general, possible applications for the user subroutine functionality in Sierra/SM. Then, again in general, we describe the various pieces and steps that are required by the user to implement a user subroutine. Subsequently, we focus on various aspects of implementing the user subroutine functionality. Section 10.1 describes the details of the user subroutine. Section 10.2 describes the command lines associated with user subroutines that appear in the Sierra/SM input file. In Section 10.3, we explain how to build and use a version of Sierra/SM that incorporates your user subroutine. Finally, Section 10.4 provides examples of actual user subroutines, and Section 10.5 lists some subroutines that are now in the standard user library.

Applications. User subroutines are primarily intended as complex function evaluators that are to be used in conjunction with an existing Sierra/SM capability (boundary conditions, element death, user output, etc.). Some Examples of possible uses of user subroutines are as follows:

- Element death determined by a complex function based on a set of physical parameters and element stress.
- Complex time and space history dependency for a displacement boundary condition.
- Computation the total contact force acting on a given surface and then use of that force to compute a specialized post processing output.
- Transformation of element stress to a arbitrary user defined local coordinate system.
- Computation of cavity volume of an enclosed sideset.
- Aerodynamic pressure based on velocity and surface normal applied to a specified surface.

Some capability exists for using mesh connectivity. It is possible to compute an element quantity based on values at the element nodes.

Some difficulties might occur in parallel applications. If computations for element A depend on quantities in element B and elements A and B are on different processors, then the computations for A may not have access to quantities in element B. For most computations in user subroutines, however, this should not be a problem.

Implementing completely new capabilities, particularly if these capabilities involve parallel computing, may be difficult or impossible with user subroutines.

General Pieces and Steps. A number of pieces and steps are required to make use of user subroutines. Here, we present a brief description of the pieces and steps that a user will need for user subroutines without going into detail. The details are discussed in later parts of this chapter.

1. You must first determine whether your application fits in the user subroutine format. This can be done by considering the above requirements and examining the description of commands for functionality in Sierra/SM. For example, the basic kinematic boundary conditions and force conditions allow for the use of user subroutines. The description of these commands includes a discussion of how a user subroutine could be applied and what command line will invoke a user subroutine.
2. If you determine that your application can make use of the user subroutine functionality in Sierra/SM, you will then need to write the subroutine. The parts of the subroutine that interface to Sierra/SM have specified formats. The details of these interfaces are described in later sections. One part of the subroutine with a specified format is the call list. Other parts of the subroutine with a specified format are code that will do the following:

   - Read parameters from the Sierra/SM input file
   - Access a variety of information—field variables, analysis time, etc.—from Sierra/SM
   - Store computed quantities

   Parameters are values they may be passed from the Sierra/SM input file to the user subroutine. Suppose that the spatial variation for some quantity in the user subroutine uses some characteristic length and the user wishes to examine results generated by using several different values of the characteristic length. By setting up the characteristic length as a parameter, the value for the parameter in the user subroutine can easily be changed by changing the value for the parameter in the input file. This lets the user change the value for a variable inside the user subroutine without having to recompile the user subroutine.

   The portion of your subroutine not built on the Sierra/SM specifications will reflect your specific application. The code to implement your application may include a loop over nodes that prescribes a displacement based on the current time for the analysis and the spatial location of the node.
3. After you write the user subroutine, you will need to have a command line in your input file that tells Sierra/SM you want to use the user subroutine you have written. For example, if your user subroutine is a specialized prescribed displacement boundary condition, then inside a `PRESCRIBED DISPLACEMENT` command block, you will have a command line of the form

   ```
   NODE SET SUBROUTINE = <string>subroutine_name
   ```

   that provides the name of your user subroutine.
4. Following the invocation of the user subroutine, there may be command lines for various parameters associated with the user subroutine. There may also be some additional command lines in other sections of the code required for your application. For example, you may have to add command lines in the region scope that will create an internal variable associated with a computed quantity so that the computed quantity can be written to the

results file.

5. Once you have constructed the user subroutine, which is a FORTRAN file, and the Sierra/SM input file, you can build an executable version of Sierra/SM that will run your user subroutine. Your Sierra/SM run will then incorporate the functionality you have created in your user subroutine.

Figure 10-1 presents a high-level overview of the various components that work together to implement the user subroutine functionality. The two main components needed for user subroutines, which are commands in the Sierra/SM input file and the actual user subroutine, are represented by the two columns in Figure 10-1.

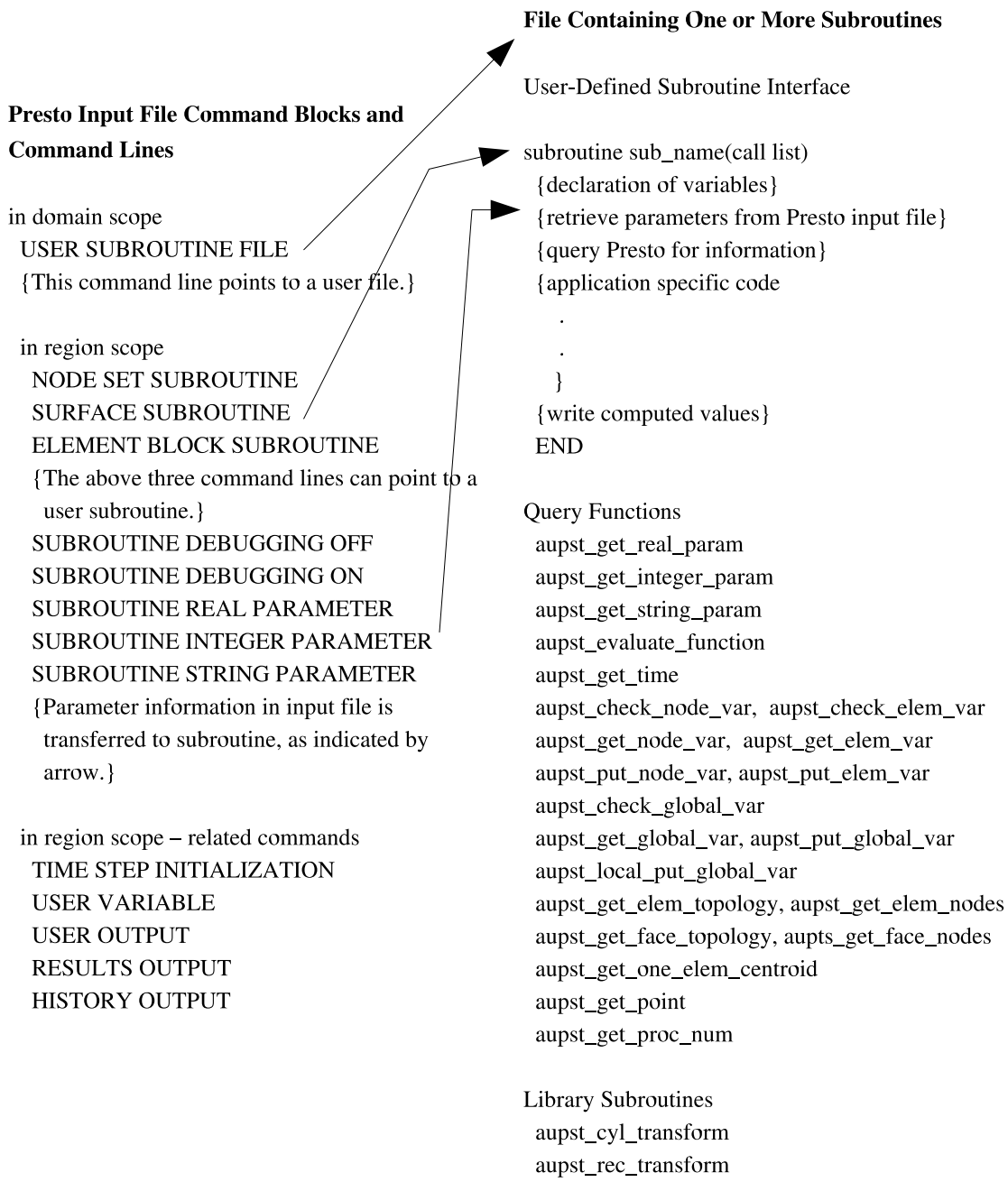


Figure 10-1. Overview of components required to implement user subroutine functionality, excluding compilation and execution commands.

10.1. Programming

Currently, user subroutines are only supported in FORTRAN 77. Any subroutine that can be compiled with a FORTRAN 77 compiler on the target execution machine can be used. The user should be aware that some computers support different FORTRAN language extensions than

others. (In the future, other languages such as FORTRAN 90, C, and C++ may be supported.)

User subroutine variable types must interface directly with the matching variable types used in the main Sierra/SM code. Thus, the FORTRAN 77 subroutines should use only integer, double precision, or character types for any data used in the interface or in any query function. Using the wrong data type may yield unpredictable results. The methods used to pass character types from Sierra/SM to FORTRAN user subroutines can be machine-dependent, but generally this functionality works well.

The basic structure for the user subroutine is as follows:

```
subroutine sub_name(call list)
{declaration of variables}
{retrieve parameters from Sierra/SM input file}
{query Sierra/SM for information}
{application-specific code
   .
   .
}
{write computed values}
END
```

In general, the user will begin the subroutine with variable declarations. After the variable declarations, the user can then query the Sierra/SM input file for parameters. Additional Sierra/SM information such as field variables or element topology can then be retrieved from Sierra/SM. Once the user has collected all the information for the application, the application-specific portion of the code can be written. After the application-specific code is complete, the user may store computed values.

Section 10.1.1 through Section 10.1.3 describe in detail the format for the interfaces to Sierra/SM that will allow the user to make the subroutine call, retrieve information from Sierra/SM, and write computed values. In these sections, mesh entities can be a node, an element face, or an element.

10.1.1. Subroutine Interface

The following interface is used for all user subroutines:

```
subroutine sub_name(int  num_objects,
           int  num_values,
           real evaluation_time,
           int  object_ids[],
           real output_values[],
           int  output_flags[],
           int  error_code)
```

The name of the user subroutine, `sub_name`, is selected by the user. Avoid names for the subroutine that are longer than 10 characters. This may cause build problems on some systems.

A detailed description of the input and output parameters is provided in Table 10-1 and Table 10-2.

Table 10-1. Subroutine Input Parameters

Input Parameter	Data Type	Parameter Description
num_objects	Integer	Number of input mesh entities. For example, if the subroutine is a node set subroutine, this would be the number of nodes on which the subroutine will operate.
num_values	Integer	Number of return values. This is the number of values per mesh entity.
evaluation_time	Real	Time at which the subroutine should be evaluated. This may vary slightly from the current analysis time. For example, in explicit dynamics analyses, velocities are evaluated one-half time step ahead.
object_ids (num_objects)	Integer	Array of mesh-entity identification numbers. The array has a length of num_objects. The input numbers are the global numbers of the input objects. The object identification numbers can be used to query information about a mesh entity.

10.1.2. Query Functions

Sierra/SM follows a design philosophy for user subroutines that a minimal amount of information should be passed through the call list. Additional information may be queried from within the subroutine. A user subroutine may query a wide variety of information from Sierra/SM.

10.1.2.1. Parameter Query

A number of user subroutine parameters may be set as described in Section 10.2.2.3. These subroutine parameters can be obtained from the Sierra/SM input file via the query functions listed below.

```
aupst_get_real_param(string var_name, real var_value,
                     int error_code)
```

Table 10-2. Subroutine Output Parameters

Output Parameter	Data Type	Parameter Description
output_values (num_values, num_objects)	Integer	Array of output values computed by the subroutine. The number of output values will be either the number of mesh entities or some multiple of the number of mesh entities. For example, if there were six nodes (num_objects equals 6) and one value was to be computed per node, the length of output_values would be 6. Similarly, if there were six nodes (num_objects equals 6) and three values were to be computed for each node (as for acceleration, which has X-, Y-, and Z-components), the length of output_values would be 18.
output_flags (num_objects)	Integer	Array of returned flags for each set of data values. When used, this array will generally have a length of num_objects. The usage of the flags depends on subroutine type; the flags are currently used only for element death and for kinematic boundary conditions. For the kinematic boundary conditions (displacement, velocity, acceleration) a flag of –1 means ignore the constraint, a flag of 0 means set the absolute constraint value, and a flag of 1 means set the constraint with direction and distance.
error_code	Integer	Error code returned by the user subroutine. A value of 0 indicates no errors. Any value other than zero is an error. If the return value is nonzero, Sierra/SM will report the error code and terminate the analysis.

```
aupst_get_integer_param(string var_name, int var_value,
                        int error_code)

aupst_get_string_param(string var_name, string var_value,
                       int error_code)
```

All three of these subroutine calls are tied to a corresponding "parameter" command line that will appear in the Sierra/SM input file. The parameter command lines are described in Section 10.2.2.3. These command lines are named based on the type of value they store, i.e., `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and `SUBROUTINE STRING PARAMETER`.

We will use the example of a real parameter to show how the subroutine call works in conjunction with the `SUBROUTINE REAL PARAMETER` command line. Suppose we have a real parameter radius that is set to a value of 2.75 on the `SUBROUTINE REAL PARAMETER` command line:

```
SUBROUTINE REAL PARAMETER: radius = 2.75
```

Also suppose we have a call to `aupst_get_real_parameter` in the user subroutine:

```
call aupst_get_real_parameter("radius",cyl_radius,error_code)
```

In the call to `aupst_get_real_parameter`, we have `var_name` set to `radius` and `var_value` defined as the real FORTRAN variable `cyl_radius`. The call to `aupst_get_real_parameter` will assign the value 2.75 to the FORTRAN variable `cyl_radius`. A similar pattern is followed for integer and string parameters.

The arguments for the parameter-related query functions are described in Table 10-3, Table 10-4, and Table 10-5. The function is repeated before each table for quick reference.

```
aupst_get_real_param(string var_name, real var_value,
                     int error_code)
```

Table 10-3. aupst_get_real_param Arguments

Parameter	Usage	Data Type	Description
var_name	Input	String	Name of a real-valued subroutine parameter, as defined in the Sierra/SM input file via the `SUBROUTINE REAL PARAMETER` command line.
var_value	Output	Real	Name of a real variable to be used in the FORTRAN subroutine. The FORTRAN variable var_value will be set to the value specified by the `SUBROUTINE REAL PARAMETER` command line.
error_code	Output	Integer	Error code indicating status of retrieving the parameter value from the input file. If the retrieval is successful, error_code is set to 0. If the parameter is not found or is the wrong type, error_code is set to a value other than 0.

```
aupst_get_integer_param(string var_name, int var_value,
                        int error_code)
```

Table 10-4. aupst_get_integer_param Arguments

Parameter	Usage	Data Type	Description
var_name	Input	String	Name of an integer-valued subroutine parameter, as defined in the Sierra/SM input file via the `SUBROUTINE INTEGER PARAMETER` command line.
var_value	Output	Integer	Name of an integer variable to be used in the FORTRAN subroutine. The FORTRAN variable var_value will be set to the value specified by the `SUBROUTINE INTEGER PARAMETER` command line.
error_code	Output	Integer	Error code indicating status of retrieving the parameter value from the input file. If the retrieval is successful, error_code is set to 0. If the parameter is not found or is the wrong type, error_code is set to a value other than 0.

```
aupst_get_string_param(string var_name, string var_value,
                       int error_code)
```

Table 10-5. aupst_get_string_param Arguments

Parameter	Usage	Data Type	Description
var_name	Input	String	Name of a string-valued subroutine parameter, as defined in the Sierra/SM input file via the `SUBROUTINE STRING PARAMETER` command line.
var_value	Output	String	Name of a string variable to be used in the FORTRAN subroutine. The FORTRAN variable var_value will be set to the value specified by the `SUBROUTINE STRING PARAMETER` command line.
error_code	Output	Integer	Error code indicating status of retrieving the parameter value from the input file. If the retrieval is successful, error_code is set to 0. If the parameter is not found or is the wrong type, error_code is set to a value other than 0.

10.1.2.2. Function Data Query

The function data query routine listed below may be used for extracting data from a function that is defined in a `FUNCTION` command block in the Sierra/SM input file. This query allows the user to directly access information stored in a function defined in the Sierra/SM input file.

```
aupst_evaluate_function(string func_name, real input_times[],
                        int num_times, real output_data[])
```

The arguments for this function are described in Table 10-6.

10.1.2.3. Time Query

The time query function can be used to determine the current analysis time. This is the time associated with the new time step. This time may not be equivalent to the `evaluation_time` argument passed into the subroutine (see Section 10.1.1, Table 10-1) as some boundary conditions need to be evaluated at different times than others. The parameter of the time query function listed below is given in Table 10-7.

Table 10-6. aupst_evaluate_function Arguments

Parameter	Usage	Data Type	Description
func_name	Input	String	Name of the function to look up.
input_times (num_times)	Input	Real	Array of times used to extract values of the function.
num_times	Input	Integer	Length of the array input_times.
output_data (num_times)	Output	Real	Array of output values of the named function at the specified times.

```
aupst_get_time(real time)
```

Table 10-7. aupst_get_time Argument

Parameter	Usage	Data Type	Description
time	Output	Real	Current analysis time.

10.1.2.4. Field Variables

Field variables (displacements, stresses, etc.) may be defined on groups of mesh entities. Many queries are available for getting and putting field variables. These queries involve passing in a set of mesh-entity identification numbers to receive field values on the mesh entities. There are query functions to check for the existence and size of a field, functions to retrieve the field values, and functions to store new variables in a field. The field query functions listed below can be used to extract any nodal or element variable field.

```
aupst_check_node_var(int num_nodes, int num_components,
                     int node_ids[], string var_name,
                     int error_code)

aupst_check_elem_var(int num_elems, int num_components,
                     int elem_ids[], string var_name,
                     int error_code)

aupst_get_node_var(int num_nodes, int num_components,
                   int node_ids[], real return_data[],
                   string var_name, int error_code)

aupst_get_elem_var(int num_elems, int num_components,
                   int elem_ids[], real return_data[],
                   string var_name, int error_code)
```

```
aupst_get_elem_var_offset(int num_elems, int num_components,
                          int offset, int elem_ids[],
                          real return_data[], string var_name,
                          int error_code)

aupst_put_node_var(int num_nodes, int num_components,
                   int node_ids[], real new_data[],
                   string var_name, int error_code)

aupst_put_elem_var(int num_elems, int num_components,
                   int elem_ids[], real new_data[],
                   string var_name, int error_code)

aupst_put_elem_var_offset(int num_elems, int num_components,
                          int offset, int elem_ids[],
                          real new_data[], string var_name,
                          int error_code)
```

The arrays where data are stored are static arrays. These arrays of a set size will be declared at the beginning of a user subroutine. The query functions to check for the existence and size of a field can be used to ensure that the size of the array of information being returned from Sierra/SM does not exceed the size of the array allocated by the user.

The arguments to field query functions are defined in Table 10-8 through Table 10-15. The function is repeated before each table for quick reference.

The `var_name` strings may be the name of any variable that could be used for code output and additionally may include the component syntax described in 9.1.

```
aupst_check_node_var(int num_nodes, int num_components,
                     int node_ids[], string var_name,
                     int error_code)
```

Table 10-8. aupst_check_node_var Arguments

Parameter	Usage	Data Type	Description
num_nodes	Input	Integer	Number of nodes used to extract field information.
num_components	Output	Integer	Number of components in the field information. A displacement field at a node has three components, for example.
node_ids (num_nodes)	Input	Integer	Array of size num_nodes listing the node identification number for each node where field information will be retrieved.
var_name	Input	String	Name of the field variable. The field variable must be a defined Sierra/SM variable.
error_code	Output	Integer	Error code indicating status of retrieving the field. If the retrieval is successful, error_code is set to 0. If a nonzero value is returned for error_code, the field variable does not exist or is not defined on one or more of the input nodes.

```
aupst_check_elem_var(int num_elems, int num_components,
                     int elem_ids[], string var_name,
                     int error_code)
```

Table 10-9. aupst_check_elem_var Arguments

Parameter	Usage	Data Type	Description
num_elems	Input	Integer	Number of elements used to extract field information.
num_components	Output	Integer	Number of components in the field information. A stress field for an eight-node hexahedron element has six components, for example.
elem_ids (num_elems)	Input	Integer	Array of size num_elems listing the element identification number for each element where field information will be retrieved.
var_name	Input	String	Name of the field variable. The field variable must be a defined Sierra/SM variable.
error_code	Output	Integer	Error code indicating status of retrieving the field. If the retrieval is successful, error_code is set to 0. If a nonzero value is returned for error_code, the field variable does not exist or is not defined on one or more of the input nodes.

```
aupst_get_node_var(int num_nodes, int num_components,
                   int node_ids[], real return_data[],
                   string var_name, int error_code)
```

Table 10-10. aupst_get_node_var Arguments

Parameter	Usage	Data Type	Description
num_nodes	Input	Integer	Number of nodes used to extract field information.
num_components	Input	Integer	Number of components in the field information. A displacement field at a node has three components, for example.
node_ids (num_nodes)	Input	Integer	Array of size num_nodes listing the node identification number for each node where field information will be retrieved.
return_data (num_components, num_nodes)	Output	Real	Array of size num_components × num_nodes containing the field data at each node.
var_name	Input	String	Name of the field variable. The field variable must be a defined Sierra/SM variable.
error_code	Output	Integer	Error code indicating status of retrieving the field. If the retrieval is successful, error_code is set to 0. If a nonzero value is returned for error_code, the field variable does not exist or is not defined on one or more of the input nodes.

```
aupst_get_elem_var(int num_elems, int num_components,
                   int elem_ids[], real return_data[],
                   string var_name, int error_code)
```

Table 10-11. aupst_get_elem_var Arguments

Parameter	Usage	Data Type	Description
num_elems	Input	Integer	Number of elements used to extract field information.
num_components	Input	Integer	Number of components in the field information. A stress field for an eight-node hexahedron element has six components, for example.
elem_ids (num_elems)	Input	Integer	Array of size num_elems listing the element identification number for each element where field information will be retrieved.
return_data (num_components, num_elems)	Output	Real	Array of size num_components × num_elems containing the field data for each element.
var_name	Input	String	Name of the field variable. The field variable must be a defined Sierra/SM variable.
error_code	Output	Integer	Error code indicating status of retrieving the field. If the retrieval is successful, error_code is set to 0. If a nonzero value is returned for error_code, the field variable does not exist or is not defined on one or more of the input nodes.

```
aupst_get_elem_var_offset(int num_elems, int num_components,
                          int offset, int elem_ids[],
                          real return_data[], string var_name,
                          int error_code)
```

Table 10-12. aupst_get_elem_var_offset Arguments

Parameter	Usage	Data Type	Description
num_elems	Input	Integer	Number of elements used to extract field information.
num_components	Input	Integer	Number of components in the field information. A stress field for an eight-node hexahedron element has six components, for example.
offset	Input	Integer	Offset into var_name field variable at which to get data.
elem_ids (num_elems)	Input	Integer	Array of size num_elems listing the element identification number for each element where field information will be retrieved.
return_data (num_components, num_elems)	Output	Real	Array of size num_components × num_elems containing the field data for each element.
var_name	Input	String	Name of the field variable. The field variable must be a defined Sierra/SM variable.
error_code	Output	Integer	Error code indicating status of retrieving the field. If the retrieval is successful, error_code is set to 0. If a nonzero value is returned for error_code, the field variable does not exist or is not defined on one or more of the input nodes.

```
aupst_put_node_var(int num_nodes, int num_components,
                   int node_ids[], real new_data[],
                   string var_name, int error_code)
```

Table 10-13. aupst_put_node_var Arguments

Parameter	Usage	Data Type	Description
num_nodes	Input	Integer	Number of nodes for which the user will specify the field data.
num_components	Input	Integer	Number of components in the field information. A displacement field at a node has three components, for example.
node_ids (num_nodes)	Input	Integer	Array of size num_nodes listing the node identification number for each node where field information will be retrieved.
new_data (num_components, num_nodes)	Input	Real	Array of size num_components × num_nodes containing the new data for the field.
var_name	Input	String	Name of the field variable. The field variable must be a defined Sierra/SM variable.
error_code	Output	Integer	Error code indicating status of retrieving the field. If the retrieval is successful, error_code is set to 0. If a nonzero value is returned for error_code, the field variable does not exist or is not defined on one or more of the input nodes.

```
aupst_put_elem_var(int num_elems, int num_components,
                   int elem_ids[], real new_data[],
                   string var_name, int error_code)
```

Table 10-14. aupst_put_elem_var Arguments

Parameter	Usage	Data Type	Description
num_elems	Input	Integer	Number of elements for which the user will specify the field data.
num_components	Input	Integer	Number of components in the field information. A stress field for an eight-node hexahedron element has six components, for example.
elem_ids (num_elems)	Input	Integer	Array of size num_elems listing the element identification number for each element where field information will be retrieved.
new_data (num_components, num_elems)	Input	Real	Array of size num_components × num_elems containing the new data for the field.
var_name	Input	String	Name of the field variable. The field variable must be a defined Sierra/SM variable.
error_code	Output	Integer	Error code indicating status of retrieving the field. If the retrieval is successful, error_code is set to 0. If a nonzero value is returned for error_code, the field variable does not exist or is not defined on one or more of the input nodes.

```
aupst_put_elem_var_offset(int num_elems, int num_components,
                          int offset, int elem_ids[],
                          real new_data[], string var_name,
                          int error_code)
```

Table 10-15. aupst_put_elem_var_offset Arguments

Parameter	Usage	Data Type	Description
num_elems	Input	Integer	Number of elements for which the user will specify the field data.
num_components	Input	Integer	Number of components in the field information. A stress field for an eight-node hexahedron element has six components, for example.
offset	Input	Integer	Offset into var_name field variable at which to put data.
elem_ids (num_elems)	Input	Integer	Array of size num_elems listing the element identification number for each element where field information will be retrieved.
new_data (num_components, num_elems)	Input	Real	Array of size num_components × num_elems containing the new data for the field.
var_name	Input	String	Name of the field variable. The field variable must be a defined Sierra/SM variable.
error_code	Output	Integer	Error code indicating status of retrieving the field. If the retrieval is successful, error_code is set to 0. If a nonzero value is returned for error_code, the field variable does not exist or is not defined on one or more of the input nodes.

10.1.2.5. Global Variables

Global variables may be extracted or set from user subroutines. A global variable has a single value for a given region. Global variables have limited support for parallel operations. The subroutine `aupst_put_global_var` attempts to immediately modify and optionally perform a parallel reduction on the value of a global variable. `aupst_put_global_var` can specify the variable reduction type “sum”, “min”, “max”, and “none.” If the “none” option is passed to the routine no global reduction operation will take place, only the processor local copy of the global variable will be modified. If any option other than “none” is used care must be taken to call

`aupst_put_global_var` on all processors at the same time with the same arguments. Failure to call the routine from all processors will result in the code hanging. For some types of subroutines this is not possible or reliable. For example, a boundary condition subroutine may not be called at all on a processor that contains no nodes in the set of nodes assigned to the boundary condition. It is recommended that the `aupst_put_global_var` function only be used in conjunction with a user subroutine referenced in a `USER OUTPUT` command block (Section 9.3.2).

Only user-defined global variables may be modified by the user subroutine (see Section 10.2.4). However, any global variable that exists on the region may be checked or extracted. The following subroutine calls pertain to global variables:

```
aupst_check_global_var(int num_comp, string var_name
                       int error_code)

aupst_get_global_var(int num_comp, real return_data,
                     string var_name, int error_code)

aupst_put_global_var(int num_comp, real input_data,
                     string var_name,
                     string reduction_type,
                     int error_code)
```

The arguments for subroutine calls pertaining to global variables are defined in Table 10-16 through Table 10-18. The call is repeated before each table for quick reference.

```
aupst_check_global_var(int num_comp, string var_name
                       int error_code)
```

Table 10-16. aupst_check_global_var Arguments

Parameter	Usage	Data Type	Description
num_comp	Output	Integer	Number of components of the global variable.
var_name	Input	String	Name of the global variable.
error_code	Output	Integer	Error code indicating status of accessing the global variable. If there is no error in accessing this variable, error_code is set to 0. A nonzero value of error_code means the global variable does not exist or in some way cannot be accessed.

```
aupst_get_global_var(int num_comp, real return_data,
                     string var_name, int error_code)
```

Table 10-17. aupst_get_global_var Arguments

Parameter	Usage	Data Type	Description
num_comp	Input	Integer	Number of components of the global variable.
return_data	Output	Real	Value of the global variable.
var_name	Input	String	Name of the global variable.
error_code	Output	Integer	Error code indicating status of accessing the global variable. If there is no error in accessing this variable, error_code is set to 0. A nonzero value of error_code means the global variable does not exist or in some way cannot be accessed.

```
aupst_put_global_var(int num_comp, real input_data,
                     string var_name,
                     string reduction_type,
                     int error_code)
```

Table 10-18. aupst_put_global_var Arguments

Parameter	Usage	Data Type	Description
num_comp	Input	Integer	Number of components of the global variable.
input_data	Input	Real	New value of the global variable.
var_name	Input	String	Name of the global variable.
reduction_type	Input	String	Type of parallel reduction to perform on the variable. Options are “sum”, “min”, “max”, and “none.”
error_code	Output	Integer	Error code indicating status of accessing the global variable. If there is no error in accessing this variable, error_code is set to 0. A nonzero value of error_code means the global variable does not exist, in some way cannot be accessed, or may not be overwritten.

10.1.2.6. Topology Extraction

The element and surface subroutines operate on groups of elements or faces. The elements and faces may have a variety of topologies. Topology queries can be used to get topological data about elements and faces. The topology of an object is represented by an integer. The integer is formed from a function of the number of dimensions, vertices, and nodes of an object. The topology of an object is given by:

```
topology = num_node + 100 * num_vert + 10000 * num_dim
```

In a FORTRAN routine, the number of nodes can easily be extracted with the mod function:

```
num_node = mod(topo,100)
num_vert = mod(topo / 100, 100)
num_dim  = mod(topo / 10000, 100)
```

Table 10-19 lists the topologies currently in use by Sierra/SM.

The following topology query functions are available in Sierra/SM:

Table 10-19. Topologies Used by Sierra/SM

Topology	Element / Face Type
00101	One-node particle
10202	Two-node beam, truss, or damper
20404	Four-node quadrilateral
20303	Three-node triangle
20304	Four-node triangle
20306	Six-node triangle
30404	Four-node tetrahedron
30410	Ten-node tetrahedron
30808	Eight-node hexahedron
30606	Six-node wedge
30505	Five-node pyramid

```
aupst_get_elem_topology(int num_elems, int elem_ids[],
                        int topology[], int error_code)

aupst_get_elem_nodes(int num_elems, int elem_ids[],
                     int elem_node_ids[], int error_code)

aupst_get_face_topology(int num_faces, int face_ids[],
                        int topology[], int error_code)

aupst_get_face_nodes(int num_faces, int face_ids[],
                     int face_node_ids[], int error_code)
```

The arguments for the topology extraction functions are defined in Table 10-20 through Table 10-23. The function is repeated before each table for quick reference.

```
aupst_get_elem_topology(int num_elems, int elem_ids[],
                        int topology[], int error_code)
```

Table 10-20. aupst_get_elem_topology Arguments

Parameter	Usage	Data Type	Description
num_elems	Input	Integer	Number of elements from which the topology will be extracted.
elem_ids (num_elems)	Input	Integer	Array of length num_elems listing the element identification for each element from which the topology will be extracted.
topology (num_elems)	Output	Integer	Array of length num_elems that has the topology for each element. See Table 8.18.
error_code	Output	Integer	Error code indicating status of retrieving the element identification numbers. If the retrieval is successful, error_code is set to 0. A nonzero value is returned for error_code if one of the element identification numbers is not valid.

```
aupst_get_elem_nodes(int num_elems, int elem_ids[],
                     int elem_node_ids[], int error_code)
```

Table 10-21. aupst_get_elem_nodes Arguments

Parameter	Usage	Data Type	Description
num_elems	Input	Integer	Number of elements from which the topology will be extracted.
elem_ids (num_elems)	Input	Integer	Array of length num_elems listing the element identification for each element from which the topology will be extracted.
elem_node_ids (number of nodes for element type × num_elems)	Output	Integer	Array containing the node identification numbers for each element requested. The length of the array is the total number of nodes contained in all elements. If the elements are eight-node hexahedra, then the number of nodes will be 8 × num_elems. The first set of eight entries in the array will be the eight nodes defining the first element. The second set of eight entries will be the eight nodes defining the second element, and so on.
error_code	Output	Integer	Error code indicating status of retrieving the element identification numbers. If the retrieval is successful, error_code is set to 0. A nonzero value is returned for error_code if one of the element identification numbers is not valid.

```
aupst_get_face_topology(int num_faces, int face_ids[],
                        int topology[], int error_code)
```

Table 10-22. aupst_get_face_topology Arguments

Parameter	Usage	Data Type	Description
num_faces	Input	Integer	Number of faces from which the topology will be extracted.
face_ids (num_faces)	Input	Integer	Array of length num_faces listing the face identification for each face from which the topology will be extracted.
topology (num_faces)	Output	Integer	Array of length num_faces containing the output topologies of each face.
error_code	Output	Integer	Error code indicating status of retrieving the face identification numbers. If the retrieval is successful, error_code is set to 0. A nonzero value is returned for error_code if one of the face identification numbers is not valid.

```
aupst_get_face_nodes(int num_faces, int face_ids[],
                     int face_node_ids[], int error_code)
```

Table 10-23. aupst_get_face_nodes Arguments

Parameter	Usage	Data Type	Description
num_faces	Input	Integer	Number of faces from which the topology will be extracted.
face_ids (num_faces)	Input	Integer	Array of length num_faces listing the face identification for each face from which the topology will be extracted.
face_node_ids (number of nodes for face type × num_faces)	Output	Integer	Array containing the node identification numbers for each face requested. The length of the array is the total number of nodes contained in all faces. If the faces are four-node quadrilaterals, then the number of nodes will be 4 × num_faces. The first set of four entries in the array will be the four nodes defining the first face. The second set of four entries will be the four nodes defining the second face, and so on.
error_code	Output	Integer	Error code indicating status of retrieving the face identification numbers. If the retrieval is successful, error_code is set to 0. A nonzero value is returned for error_code if one of the face identification numbers is not valid.

10.1.3. Miscellaneous Query Functions

Many miscellaneous query functions are available for computing some commonly used quantities.

```
aupst_get_elem_centroid(int num_elems, int elem_ids[],
                     real centroids, int error_code)

aupst_get_point(string point_name, real point_coords,
                     int error_code)

aupst_get_proc_num(proc_num)
```

The arguments for the miscellaneous query functions are defined in Table 10-24 through Table 10-26. The function is repeated before each table for quick reference.

```
aupst_get_elem_centroid(int num_elems, int elem_ids[],
                        real centroids[], int error_code)
```

Table 10-24. aupst_get_elem_centroid Arguments

Parameter	Usage	Data Type	Description
num_elems	Input	Integer	Number of elements for which to extract the topology.
elem_ids (num_elems)	Input	Integer	Array of length num_elems listing the element identification for each element for which the centroid will be computed.
centroids (3, num_elems)	Output	Real	Array of length 3 × num_elems containing the centroid of each element.
error_code	Output	Integer	Error code indicating status of retrieving the element identification numbers. If the retrieval is successful, error_code is set to 0. A nonzero value is returned for error_code if one of the element identification numbers is not valid.

```
aupst_get_point(string point_name, real point_coords,
                    int error_code)
```

Table 10-25. aupst_get_point Arguments

Parameter	Usage	Data Type	Description
point_name	Input	String	SIERRA name for a given point.
point_coords (3)	Output	Real	Array of length 3 containing the x, y, and z coordinates of the point.
error_code	Output	Integer	Error code indicating status of retrieving the point. If the retrieval is successful, error_code is set to 0. A nonzero value is returned for error_code if the point cannot be found

```
aupst_get_proc_num(proc_num)
```

Table 10-26. aupst_get_proc_num Arguments

Parameter	Usage	Data Type	Description
proc_num	Output	Integer	Processor number of the calling process. This number can be used for informational purposes. A common example is that output could only be written by a single processor, e.g., processor 0, rather than by all processors.

10.2. Command File

In addition to the actual user subroutine, you will need to add command lines to your input file to make use of your user subroutine. This section describes the command lines that are used in conjunction with user subroutines. This section also describes two additional command blocks, `TIME STEP INITIALIZATION` and `USER VARIABLE`. The `TIME STEP INITIALIZATION` command block lets you execute a user subroutine at the beginning of a time step as opposed to some later time. The `USER VARIABLE` command block can be used in conjunction with user subroutines or for user-defined output.

10.2.1. Subroutine Identification

As described in Section 2.1.4, there is one command line associated with the user subroutine functionality that must be provided in the SIERRA scope:

```
USER SUBROUTINE FILE = <string>file_name
```

The named file may contain one or more user subroutines. The file must have an extension of ".F", as in blast.F.

10.2.2. User Subroutine Command Lines

```
{begin command block}
  NODE SET SUBROUTINE = <string>subroutine name |
    SURFACE SUBROUTINE = <string>subroutine_name |
    ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
  SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
  SUBROUTINE REAL PARAMETER: <string>param_name
      = <real>param_value
  SUBROUTINE INTEGER PARAMETER: <string>param_name
      = <integer>param_value
  SUBROUTINE STRING PARAMETER: <string>param_name
      = <string>param_value
{end command block}
```

A number of user subroutine command lines will appear in some Sierra/SM command block. User subroutine commands can appear in boundary condition, element death (only for explicit dynamics), user output, and state initialization command blocks. The possible command lines are shown above. The following sections describe the command lines related to user subroutines.

10.2.2.1. Type

User subroutines are currently available in three general types: node set, surface, and element.

Node set subroutines operate on groups of nodes. The command line for defining a node set subroutine is:

```
NODE SET SUBROUTINE = <string>subroutine_name
```

where `subroutine_name` is the name of the user subroutine. The name is case sensitive. A node set subroutine will operate on all nodes contained in an associated mechanics instance.

Surface subroutines work on groups of surfaces. A surface may be an external face of a solid element or the face of a shell element associated with either the positive or negative normal for the surface of the shell. The command line for defining a surface subroutine is:

```
SURFACE SUBROUTINE = <string>subroutine_name
```

Element block subroutines work on groups of elements. The command line for defining an element block subroutine is:

```
ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
```

An element may be a solid element such as a hexahedron or a two-dimensional element such as a shell.

Different Sierra/SM features may accept one or more types of user subroutines. Only one subroutine is allowed per command block.

10.2.2.2. Debugging

Subroutines may be run in a special debugging mode to help catch memory errors. For example, there is a potential for a user subroutine to write outside of its allotted data space by writing beyond the bounds of an input or output array. Generally, this causes Sierra/SM to crash, but it also has the potential to introduce other hard-to-trace bugs into the Sierra/SM analysis. Subroutines run in debug mode require more memory and more processing time than subroutines not run in debug mode.

Subroutine debugging is on by default in debug executables. It can be turned off with the following command line:

```
SUBROUTINE DEBUGGING OFF
```

Subroutine debugging is off by default in optimized executables. It can be turned on with the following command line:

```
SUBROUTINE DEBUGGING ON
```

10.2.2.3. Parameters

All user subroutines have the ability to use parameters. Parameters are defined in the input file and are quickly accessible by the user subroutine during run time. Parameters are a way of making a single user subroutine much more versatile. For example, a user subroutine could be written to define a periodic loading on a structure. A parameter for the subroutine could be defined specifying the frequency of the function. In this way, the same subroutine can be used in different parts of the model, and the subroutine behavior can be modified without recompiling the

program. These command lines are placed within the scope of the command block in which the user subroutine is specified.

Real-valued parameters can be stored with the following command line:

```
SUBROUTINE REAL PARAMETER: <string>param_name
   = <real>param_value
```

Integer-valued parameters can be stored with the following command line:

```
SUBROUTINE INTEGER PARAMETER: <string>param_name
   = <integer>param_value
```

String-valued parameters can be stored with the following command line:

```
SUBROUTINE STRING PARAMETER: <string>param_name
   = <string>param_value
```

Any number of subroutine parameters may be defined. The subroutine parameters may be defined in any order within the command block. The user subroutine may request the values of the parameters but is not required to use them or even have any knowledge of their existence. An example of subroutine usage with parameters is as follows:

```
BEGIN PRESSURE
  SURFACE = surface_1
  SURFACE SUBROUTINE = blast_pressure
  SUBROUTINE REAL PARAMETER: blast_time = 1.2
  SUBROUTINE REAL PARAMETER: blast_power = 1.3e+07
  SUBROUTINE STRING PARAMETER: formulation = alpha
  SUBROUTINE INTEGER PARAMETER: decay_exponent = 2
  SUBROUTINE DEBUGGING ON
END PRESSURE
```

In the above example, four parameters are associated with the subroutine `blast_pressure`. Two of the parameters are real (`blast_time` and `blast_power`), one of the parameters is a string (`formulation`), and one of the parameters is an integer (`decay_exponent`). To access the parameters in the user subroutine, the user will need to include interface calls described in previous sections.

10.2.3. Time Step Initialization

```
BEGIN TIME STEP INITIALIZATION
  # mesh-entity set commands
  NODE SET|NODESET = <string list>nodelist_names
  SURFACE = <string list>surface_names
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE NODE SET = <string list>nodelist_names
```

```
    REMOVE SURFACE = <string list> surface_names
    REMOVE BLOCK = <string list>block_names
    #
    # external database commands
    READ VARIABLE = <string> mesh_var_name
    COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
      MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
    TYPE = <string>NODE|ELEMENT
      [<string>REAL|INTEGER LENGTH = <integer>length]|
      [<string>SYM_TENSOR|FULL_TENSOR|VECTOR]
    #
    # user subroutine commands
    NODE SET SUBROUTINE = <string>subroutine_name |
      SURFACE SUBROUTINE = <string>sub_name |
      ELEMENT BLOCK SUBROUTINE = <string>sub_name
    SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
    SUBROUTINE REAL PARAMETER: <string>param_name
      = <real>param_value
    SUBROUTINE INTEGER PARAMETER: <string>param_name
      = <integer>param_value
    SUBROUTINE STRING PARAMETER: <string>param_name
      = <string>param_value
    #
    # additional command
    ACTIVE PERIODS = <string list>period_names
    INACTIVE PERIODS = <string list>period_names

END TIME STEP INITIALIZATION
```

The `TIME STEP INITIALIZATION` command block, which appears in the region scope, is used to control variables while an analysis is running. This can be done by reading a variable from a separate mesh file or by running a user subroutine. These variables can then be used to compute quantities used by other command types. For example, if the traction on a surface was dependent on the area, the time step initialization subroutine could be used to calculate the area, and that area could be stored and later read when calculating the traction. The user initialization subroutine will pass the specified mesh objects to the subroutine for use in calculating some value.

The `TIME STEP INITIALIZATION` command block contains two groups of commands—mesh entity set and user subroutine. In addition to the command lines in the these command groups, there is an additional command line: `ACTIVE PERIODS` or `INACTIVE PERIODS`. Following are descriptions of the different command groups and the `ACTIVE PERIODS` or `INACTIVE PERIODS` command line.

10.2.3.1. Mesh-Entity Set Commands

The `mesh-entity set commands` portion of the `TIME STEP INITIALIZATION` command block specifies the nodes, element faces, or elements associated with the particular subroutine run

at the beginning of the applicable time steps. This portion of the command block can include some combination of the following command lines:

```
NODE SET|NODESET = <string list>nodelist_names
SURFACE = <string list>surface_names
BLOCK = <string list>block_names
INCLUDE ALL BLOCKS
REMOVE NODE SET = <string list>nodelist_names
REMOVE SURFACE = <string list> surface_names
REMOVE BLOCK = <string list>block_names
```

These command lines, taken collectively, constitute a set of Boolean operators for constructing a set of nodes, element faces, or elements. See Section 6.1 for more information about the use of these command lines for mesh entities. There must be at least one `NODE SET|NODESET`, `SURFACE`, `BLOCK`, or `INCLUDE ALL BLOCKS` command line in the command block.

10.2.3.2. External Mesh Database Commands

```
READ VARIABLE = <string>var_name
COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
  MAP_BY_PROXIMITY|MAP_BY_ID(MAP_BY_PROXIMITY)
TYPE = <string>NODE|ELEMENT
    [<string>REAL|INTEGER LENGTH = <integer>length]|
    [<string>SYM_TENSOR|FULL_TENSOR|VECTOR]
```



Warning: At this time the `COPY VARIABLE` command can only be used with nodal variables. Neither the `COPY VARIABLE` nor `READ VARIABLE` commands are compatible with global variables.

If the external database option is used, the requested variable is read from an external mesh database. The variables are read from a finite element model defined via the `FINITE ELEMENT MODEL` command block described in Section 6.1. The finite element model can either be the model used by the region for its mesh definition as specified with the `USE FINITE ELEMENT MODEL` command (see Section 2.3), or it can be a different (but compatible) model.The `TYPE` command specifies the type of data the field is associated with. Once `TYPE` and one of the other commands are specified, the variable will show up on the new exodus mesh with the same name.

See Section 7.3.5 for more details.

10.2.3.3. User Subroutine Commands

The following command lines are related to the user subroutine specification:

```
NODE SET SUBROUTINE = <string>subroutine_name |
  SURFACE SUBROUTINE = <string>subroutine_name |
  ELEMENT BLOCK SUBROUTINE = <string>subroutine_name
SUBROUTINE DEBUGGING OFF | SUBROUTINE DEBUGGING ON
SUBROUTINE REAL PARAMETER: <string>param_name
  = <real>param_value
SUBROUTINE INTEGER PARAMETER: <string>param_name
  = <integer>param_value
SUBROUTINE STRING PARAMETER: <string>param_name
  = <string>param_value
```

Only one of the first three command lines listed above can be specified in the command block. The particular command line selected depends on the mesh-entity type of the variable being initialized. For example, variables associated with nodes would be initialized if you are using the `NODE SET SUBROUTINE` command line, variables associated with faces if you are using the `SURFACE SUBROUTINE` command line, and variables associated with elements if you are using the `ELEMENT BLOCK SUBROUTINE` command line. The string `subroutine_name` is the name of a FORTRAN subroutine that is written by the user.

Following the selected subroutine command line are other command lines that may be used to implement the user subroutine option. These command lines are described in Section 10.2.2 and consist of `SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`, `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and `SUBROUTINE STRING PARAMETER`. Examples of using these command lines are provided throughout Chapter 10.

10.2.3.4. Additional Command

The `ACTIVE PERIODS` or `INACTIVE PERIODS` command line can optionally appear in the `TIME STEP INITIALIZATION` command block:

```
ACTIVE PERIODS = <string list>period_names

INACTIVE PERIODS = <string list>period_names
```

The `ACTIVE PERIODS` or `INACTIVE PERIODS` command line is used to activate or deactivate the running of the user subroutine at the beginning of every time step for certain time periods. See Section 2.6 for more information about this optional command line.

10.2.4. User Variables

```
BEGIN USER VARIABLE <string>var_name
  BLOCK = <string list>block_names
  TYPE = <string>NODE|ELEMENT|GLOBAL
    [<string>REAL|INTEGER LENGTH = <integer>length]|
    [<string>SYM_TENSOR|FULL_TENSOR|VECTOR]
```

```
  GLOBAL OPERATOR = <string>SUM|MIN|MAX
  INITIAL VALUE = <real list>values
  INITIAL VARIATION = <real list>values LINEAR DISTRIBUTION
  USE WITH RESTART
END [USER VARIABLE <string>var_name]
```

The USER VARIABLE command block is used to create a user-defined variable. This kind of variable may be used for scratch space in a user subroutine or for some user-defined output. A user-defined variable may be output to the results file or the history file just like any other defined variable; i.e., a user-defined variable once defined by the USER VARIABLE command block can be specified in a USER OUTPUT command block, a RESULTS OUTPUT command block, and a HISTORY OUTPUT command block.

User-defined variables are associated with mesh entities: a node variable will exist on the nodes, an element variable will exist on the elements, and a global variable will have a single value for the entire model. By default, the variable is registered on the entire model, but the BLOCK command can be used to register the variable on a subset of the model. The BLOCK command enables registering a variable with different numbers of components between blocks, such as an element variable with LENGTH = 1 on a block with mean-quadrature linear hexahedra and LENGTH = 8 on another block with fully-integrated linear hexahedra.

If the user-defined variable functionality is used in conjunction with restart, the USE WITH RESTART command line must be included.

The USER VARIABLE command block is placed within a Sierra/SM region. The command block begins with the input line:

```
BEGIN USER VARIABLE <string>var_name
```

and ends with the input line:

```
END [USER VARIABLE <string>var_name]
```

where var_name is a user-selected name for the variable. In the above command block:

- A user-defined variable has an associated type that is specified by the TYPE command line, which itself contains several parameters. The TYPE command line is required.
    1. The variable must be a nodal quantity, an element quantity, or a global quantity. The options NODE, ELEMENT, and GLOBAL determine whether the variable will be a nodal, element, or global quantity. One of these options must appear on the TYPE command line.
    2. The user-defined variable can be either an integer or a real, as specified by the INTEGER or REAL option.
    3. The length of the variable must be set by using one of the options SYM_TENSOR, FULL_TENSOR, VECTOR, or LENGTH = <integer>length. If the LENGTH option is used, the user must specify whether the variable is an integer number or a real number by using the INTEGER or REAL option. If the SYM_TENSOR option is used, the variable has six real components. If the FULL_TENSOR is used, the variable has nine

real components. If the VECTOR option is used, the variable has three real components. The three options SYM_TENSOR, FULL_TENSOR, and VECTOR all imply real numbers, and thus the REAL option need not be included in the command line when one of these three options is specified.

Some examples of the TYPE command line follow:

```
type = global real length = 1
type = element tensor
type = element real length = 3
type = node sym_tensor
type = node vector
```

- If you use the GLOBAL option on the TYPE command line, a global variable is created, and this global variable must be given an associated reduction type, which is specified by the GLOBAL OPERATOR command line. The reduction type tells Sierra/SM how to reduce the individual values stored on each processor to a mesh global value. Global reductions are performed at the end of each time step. The SUM operator sums all processor variable contributions. The MAX operator takes the maximum value among all processor variable contributions. The MIN operator takes the minimum value among all processor variable contributions.
- One or more initial values may be specified for the user-defined variable in the INITIAL VALUE command line. The number of initial values specified should be the same as the length of the variable, as specified in the TYPE command line either explicitly via the LENGTH option or implicitly via the SYM_TENSOR, FULL_TENSOR, or VECTOR option. The initial values will be copied to the variable space on every mesh object on which the variable is defined. Only real type variables may be given initial values at this time.
- The initial value of the user variable can be given some random distribution by use of the INITIAL VARIATION command line. If the INITIAL VARIATION command is used the INITIAL VALUE command must also be used. In addition the number of values specified in the initial variation command must exactly match the number of values specified in the initial value command. When the initial variation command is used the initial values of the variable will be set as initial value plus or minus some random factor times the initial variation. Currently the only random distribution supported is the linear distribution. With linear distribution the random values will be distributed evenly from initial value minus variation to initial value plus variation.
- All intrinsic type options such as REAL, INTEGER, SYM_TENSOR, FULL_TENSOR, VECTOR and the LENGTH option can be used with any of the mesh entity options (NODE, ELEMENT, GLOBAL).
- As indicated previously, if the user-defined variable functionality is used in conjunction with restart, the USE WITH RESTART command line must be included.

10.3. Compilation and Execution

Running a code with user subroutines is a two-step process. First, an executable version of Sierra/SM that recognizes the user subroutines must be created. This executable must then be used for a Sierra/SM analysis with an input file that incorporates the proper user subroutine command lines.

How the above two steps are carried out is site- and installation-specific, but an example demonstrating this process on various systems at Sandia using SIERRA—a general code framework and code management system at Sandia—command lines is provided.

For the first step, you will need the user subroutine, in a FORTRAN file, and a Sierra/SM input file that makes use of the user subroutine. You will use a system command line of the general form shown below.

```
sierra --make adagio -i <string>input_file_name
```

Suppose that you have a subdirectory in your area called `test` and you wish to incorporate a user subroutine called `blast_load`. The actual user subroutine will be in a file called `blast_load.F`, and the associated input file will be called `blast_load_1.i`. Both of these files will be in the directory `test`. In the input file, you will have the following command line in the SIERRA scope:

```
USER SUBROUTINE FILE = blast_load.F
```

You will also have some subset of the command lines described in the previous section in your Sierra/SM input file. The specific form of the system command line to execute the first step of the user subroutine process is shown below.

```
sierra --make adagio -i blast_load_1.i
```

The above command will create a local version of Sierra/SM in a local directory named `UserSubsProject`. The system command line to run the local version of Presto is shown below.

```
sierra ./UserSubsProject/bin/adagio -i <string>input_file_name
```

The specific form of the system command line you will execute in the subdirectory test is shown below.

```
sierra ./UserSubsProject/bin/adagio -i blast_load_1.i
```

The second command line runs Sierra/SM using `blast_load_1.i` as an input file and uses the user subroutines in the process. Again, all of this is a site-specific example. You must determine how Sierra/SM is set up at your installation to determine what system command lines are necessary to build Sierra/SM with user subroutines and then use this version of Sierra/SM.

10.4. Examples

10.4.1. Pressure as a Function of Space and Time

(The following example provides functionality—a blast load on a surface—more applicable to Presto than Adagio. It is included in both manuals as it is instructive in the general use of a user-defined subroutine.)

The following code is an example of a user subroutine to compute blast pressures on a group of faces. The blast pressure simulates a blast occurring at a specified position and time. The blast wave radiates out from the center of the blast and dissipates over time. This subroutine is included in the input file as follows:

```
#In the SIERRA scope:
user subroutine file = blast_load.F

#In the region scope:
begin pressure
  surface = surface_1
  surface subroutine = blast_load
  subroutine real parameter: pos_x = 5.0
  subroutine real parameter: pos_y = 5.0
  subroutine real parameter: pos_z = 1.6
  subroutine real parameter: wave_speed = 1.5e+02
  subroutine real parameter: blast_time = 0.0
  subroutine real parameter: blast_energy = 1.0e+09
  subroutine real parameter: blast_wave_width = 0.75
end pressure
```

The FORTRAN 77 subroutine listing follows. It would be possible to increase the speed of this subroutine by calling the topology functions (see Section 10.1.2.6) on groups of elements, though this would increase subroutine complexity.

```
c
c  Subroutine to simulate a blast load on a surface
c
      subroutine blast_load(num_faces, num_vals,
     &  eval_time, faceID, pressure, flags, err_code)

      implicit none
c
c  Subroutine input arguments
c
      integer num_faces
      double precision eval_time
      integer faceID(num_faces)
```

```
      integer num_vals

c
c  Subroutine output arguments
c
      double precision pressure(num_vals, num_faces)
      integer flags(num_faces)
      integer err_code
c
c  Variables to hold the subroutine parameters
c
      double precision pos_x, pos_y, pos_z, wave_speed,
     &                 blast_time, blast_energy,
     &                 blast_wave_width
c
c  Local variables
c
      integer iface, inode
      integer cur_face_id, face_topo, num_nodes
      integer num_comp_check
      double precision dist, blast_o_rad, blast_i_rad
      double precision blast_volume, blast_pressure
      integer query_error
      double precision face_center(3)
c
c  Create some static variables to hold queried
c  information.  Assume no face has more than 10
c  nodes
c
      double precision face_nodes(10)
      double precision face_coords(3, 10)
c
c  Extract the subroutine parameters
c
      call aupst_get_real_param("pos_x",pos_x,query_error)
      call aupst_get_real_param("pos_y",pos_y,query_error)
      call aupst_get_real_param("pos_z",pos_z,query_error)
      call aupst_get_real_param("wave_speed",wave_speed,
     &                          query_error)
      call aupst_get_real_param("blast_energy",
     &                          blast_energy,query_error)
      call aupst_get_real_param("blast_time",
     &                          blast_time,query_error)
      call aupst_get_real_param("blast_wave_width",
     &                  blast_wave_width, query_error)
c
c  Determine the outer radius of the blast wave
```

```
c
      blast_o_rad = (eval_time - blast_time) * wave_speed
      if(blast_o_rad .le. 0.0) return;
c
c  Determine the inner radius of the blast wave
c
      blast_i_rad = blast_o_rad - blast_wave_width
      if(blast_i_rad .le. 0.0) blast_i_rad = 0.0
c
c  Determine the total volume the blast wave occupies
c
      blast_volume = 3.1415 * (4.0/3.0) *
     &               (blast_o_rad**2 - blast_i_rad**2)
c
c  Determine the total pressure on faces inside the
c  blast wave
c
      blast_pressure = blast_energy / blast_volume
c
c  Loop over all faces in the set
c
      do iface = 1, num_faces
c
c  Extract the topology of the current face
c
        cur_face_id = faceID(iface)
        call aupst_get_face_topology(1, cur_face_id,
     &                               face_topo, query_error)
c
c  Determine the number of nodes of the current face
c
        num_nodes = mod(face_topo,100)
c
c  Extract the node ids for nodes contained in the current
c  face
c
        call aupst_get_face_nodes(1, cur_face_id,
     &                             face_nodes, query_error)
c
c  Extract the nodal coordinates of the face nodes
c
        call aupst_get_node_var(num_nodes, 3, face_nodes,
     &          face_coords, "coordinates", query_error)
c
c  Compute the centroid of the face
c
        face_center(1) = 0.0
```

```
         face_center(2) = 0.0
         face_center(3) = 0.0
         do inode = 1, num_nodes
           face_center(1) = face_center(1) +
     &                      face_coords(1,inode)
           face_center(2) = face_center(2) +
     &                      face_coords(2,inode)
           face_center(3) = face_center(3) +
     &                      face_coords(3,inode)
         enddo
         face_center(1) = face_center(1)/num_nodes
         face_center(2) = face_center(2)/num_nodes
         face_center(3) = face_center(3)/num_nodes
c
c  Determine the distance from the current face
c  to the blast center
c
         dist = sqrt((face_center(1) - pos_x)**2 +
     &               (face_center(2) - pos_y)**2 +
     &               (face_center(3) - pos_z)**2)
c
c  Apply pressure to the current face if it falls within
c  the blast wave
c
         if(dist .ge. blast_i_rad .and.
     &      dist .le. blast_o_rad) then
           pressure(1,iface) = blast_pressure
         else
           pressure(1,iface) = 0.0
         endif
       enddo
       err_code = 0
       end
```

10.4.2. Error Between a Computed and an Analytic Solution

The following code is a user subroutine to compute the error between Sierra/SM-computed results and results from an analytic manufactured solution. This subroutine is called by a `USER OUTPUT` command block immediately before producing an output Exodus file. The error for the mesh is computed by taking the squared difference between the computed and analytic displacements at every node. Finally, a global sum of the error is produced along with the square root norm of the error.

This user subroutine requires a user variable, which is defined in the Sierra/SM input file. The command block for the user variable specified in this user subroutine is as follows:

```
begin user variable conv_error
```

```
 type = global real length = 1
  global operator = sum
  initial value = 0.0
end user variable conv_error
```

The subroutine is called in the Sierra/SM input file as follows:

```
begin user output
  node set = nodelist_10
  node set subroutine = conv0_error
  subroutine real parameter: char_length = 1.0
  subroutine real parameter: char_time   = 1.0e-3
  subroutine real parameter: x_offset    = 0.0
  subroutine real parameter: y_offset    = 0.0
  subroutine real parameter: z_offset    = 0.0
  subroutine real parameter: t_offset    = 0.0
  subroutine real parameter: u0          = 0.01
  subroutine real parameter: v0          = 0.02
  subroutine real parameter: w0          = 0.03
  subroutine real parameter: alpha       = 1.0
  subroutine real parameter: youngs_modulus = 10.0e6
  subroutine real parameter: poissons_ratio = 0.3
  subroutine real parameter: density        = 0.0002588
  subroutine real parameter: num_nodes      = 125.0
end user output
```

The FORTRAN listing for the subroutine is as follows:

```
      subroutine conv0_error(num_nodes, num_vals,
     &  eval_time, nodeID, values, flags, ierror)
      implicit none

      integer num_nodes
      integer num_vals
      double precision eval_time
      integer nodeID(num_nodes)
      double precision values(1)
      integer flags(1)
      integer ierror
c
c     Local vars
c
      integer inode
      integer error_code
      double precision clength, ctime, xoff, yoff, zoff, toff
      double precision zero, one, two, three, four, nine
```

```
      double precision mod_coords(3,3000)
      double precision cdispl(3,3000)
      integer num_comp_check
      double precision expat
      double precision x, y, z, t
      double precision u0, v0, w0, alpha
      double precision pi
      double precision half
      double precision mdisplx, mdisply, mdisplz
      double precision xdiff, ydiff, zdiff
      double precision conv_error
      double precision numnod

      pi    = 3.141592654
      half  = 0.5
      zero  = 0.0
      one   = 1.0
      two   = 2.0
      three = 3.0
      four  = 4.0
      nine  = 9.0
c
c  Check that the nodal coordinates will fit into the
c  statically allocated array
c
      if(num_nodes .gt. 3000) then
        write(6,*) ERROR in sphere disp, cdispl,
     &  num_nodes exceeds static array size
        ierror = 1
        return
      endif
c
c  Extract the model coordinates for all nodes
c
      call aupst_check_node_var(num_nodes, num_comp_check,
     &                          nodeID, "model_coordinates",
     &                          ierror)
      if(ierror .ne. 0) return
      if(num_comp_check .ne. 3) return
      call aupst_get_node_var(num_nodes, num_comp_check,
     &       nodeID, mod_coords, "model_coordinates",
     &       ierror)
c
c  Extract the computed displacements for all nodes
c
      call aupst_check_node_var(num_nodes, num_comp_check,
     &                          nodeID, "displacement",
```

```
     &                            ierror)
      if(ierror .ne. 0) return
      if(num_comp_check .ne. 3) return
      call aupst_get_node_var(num_nodes, num_comp_check,
     &       nodeID, cdispl, "displacement",
     &       ierror)
c
c  Extract the subroutine parameters.
c
      call aupst_get_real_param("char_length",
     &                          clength,error_code)
      call aupst_get_real_param("char_time",
     &                          ctime,error_code)
      call aupst_get_real_param("x_offset",xoff,error_code)
      call aupst_get_real_param("y_offset",yoff,error_code)
      call aupst_get_real_param("z_offset",zoff,error_code)
      call aupst_get_real_param("t_offset",toff,error_code)
      call aupst_get_real_param("u0",u0,error_code)
      call aupst_get_real_param("v0",v0,error_code)
      call aupst_get_real_param("w0",w0,error_code)
      call aupst_get_real_param("alpha",alpha,error_code)
      call aupst_get_real_param("num_nodes",
     &                          numnod,error_code)
c
c  Calculate a solution scaling factor
c
      expat = half * ( one - cos( pi * eval_time / ctime ) )
c
c  Compute the expected solution at each node and do a
c  sum of the differences from the analytic solution
c
      conv_error = zero
      do inode = 1, num_nodes
c
c  Set the displacement value from the manufactured solution
c
        x = ( mod_coords(1,inode) - xoff ) / clength
        y = ( mod_coords(2,inode) - yoff ) / clength
        z = ( mod_coords(3,inode) - zoff ) / clength
c
        mdisplx = u0 * sin(x) * cos(two*y) * cos(three*z)
     *               * expat
        mdisply = v0 * cos(three*x) * sin(y) * cos(two*z)
     *               * expat
        mdisplz = w0 * cos(two*x) * cos(three*y) * sin(z)
     *               * expat
c
```

```
         xdiff = mdisplx - cdispl(1,inode)
         ydiff = mdisply - cdispl(2,inode)
         zdiff = mdisplz - cdispl(3,inode)
         conv_error = conv_error + xdiff*xdiff
     *                           + ydiff*ydiff
     *                           + zdiff*zdiff
c
      enddo
c
      ierror = 0
c
c  Do a parallel sum of the squared errors and extract
c  the total summed value on all processors
c
      call aupst_put_global_var(1,conv_error,
     &                          "conv_error","sum",ierror)
      call aupst_get_global_var(1,conv_error,
     &                          "conv_error",ierror)
c
c  Take the square root of the errors and store that as
c  the net error norm
c
      conv_error = sqrt(conv_error) / sqrt(numnod)
      call aupst_put_global_var(1,conv_error,
     &                          "conv_error","none",ierror)
c
      return
      end
```

10.4.3. Transform Output Stresses to a Cylindrical Coordinate System

The following code is a user subroutine to transform element stresses in global *x*, *y*, and *z* coordinates to a global cylindrical coordinate system. This subroutine could be used to transform the relatively meaningless shell stress in *x*, *y*, and *z* coordinates to more meaningful tangential, hoop, and radial stresses. The subroutine is called from a `USER OUTPUT` command block. It reads in the old stresses, transforms them, and writes them back out to a user-created scratch variable, defined via a `USER VARIABLE` command block, for output.

```
begin user variable cyl_stress
  type = element sym_tensor length = 1
  initial value = 0.0 0.0 0.0 0.0 0.0 0.0
end user variable

begin user output
  block = block_1
  element block subroutine = aupst_cyl_transform
```

```
  subroutine string parameter: origin_point = Point_O
  subroutine string parameter: z_point      = Point_Z
  subroutine string parameter: xz_point     = Point_XZ
  subroutine string parameter: input_stress = memb_stress
  subroutine string parameter: output_stress = cyl_stress
end user output
```

The FORTRAN listing for the subroutine is as follows:

```
      subroutine aupst_cyl_transform(num_elems, num_vals,
     *  eval_time, elemID, values, flags, ierror)
      implicit none
#include<framewk/Fmwk_type_sizes_decl.par>
#include<framewk/Fmwk_type_sizes.par>
c
c  Subroutine Arguments
c
c  num_elems: Input: Number of elements to calculate on
c  num_vals : Input: Ignored
c  eval_time: Input: Time at which to evaluate the stress.
c  elemID   : Input: Global sierra IDs of the input elements
c  values   : I/O  : Ignored, stress will be stored manually
c  flags    : I/O  : Ignored
c  ierror   :Output: Returns non-zero if an error occurs
c
      integer num_elems
      integer num_vals
      double precision eval_time
      integer elemID(num_elems)
      double precision values(1)
      integer flags(1)
      integer ierror
c
c  Fortran cannot dynamically allocate memory, thus worksets
c  will be iterated over by chucks each of size chunk_size.
c
      integer chunk_size
      parameter (chunk_size = 100)
      integer chunk_ids(chunk_size)
c
c  Subroutine parameter data
c
      character*80     origin_point_name
      double precision origin_point(3)
      character*80     z_point_name
      double precision z_point(3)
      character*80     xz_point_name
```

```
      double precision xz_point(3)
      character*80     input_stress_name
      character*80     output_stress_name
c
c  Local element data for centroids and rotation vectors
c
      double precision cent(3)
      double precision centerline_pos(3)
      double precision dot_prod
      double precision z_vec(3)
      double precision r_vec(3)
      double precision theta_vec(3)
      double precision rotation_tensor(9)
c
c  Chunk data storage
c
      double precision elem_centroid(3, chunk_size)
      double precision input_stress_val(6, chunk_size)
      double precision output_stress_val(6, chunk_size)
c
c  Simple iteration variables
c
      integer error_code
      integer ichunk, ielem
      integer zero_elem, nel
c
c  Extract the current subroutine parameters.  origin_point
c  is the origin of the coordinate system
c  z_point is a point on the z axis of the coordinate system
c  xz_point is a point on the xz plane
c
      call aupst_get_string_param("origin_point",
     &                            origin_point_name,
     &                            error_code)
      call aupst_get_string_param("z_point",
     &                            z_point_name,
     &                            error_code)
      call aupst_get_string_param("xz_point",
     &                            xz_point_name,
     &                            error_code)
      call aupst_get_string_param("input_stress",
     &                            input_stress_name,
     &                            error_code)
      call aupst_get_string_param("output_stress",
     &                            output_stress_name,
     &                            error_code)
c
```

```
c  Use the point names to look up the coordinates of each
c  relevant point
c
      call aupst_get_point(origin_point_name, origin_point,
     &                     error_code)
      call aupst_get_point(z_point_name, z_point,
     &                     error_code)
      call aupst_get_point(xz_point_name, xz_point,
     &                     error_code)
c
c  Compute the z axis vector
c
      z_vec(1) = z_point(1) - origin_point(1)
      z_vec(2) = z_point(2) - origin_point(2)
      z_vec(3) = z_point(3) - origin_point(3)
c
c  Transform z_vec into a unit vector, abort if it is invalid
c
      call aupst_unitize_vector(z_vec, ierror)
      if(ierror .ne. 0) return
c
c  Loop over chunks of the data arrays
c
      do ichunk = 1, (num_elems/chunk_size + 1)
c
c  Determine the first and last element number for the
c  current chunk of elements
c
       zero_elem = (ichunk-1) * chunk_size
       if((zero_elem + chunk_size) .gt. num_elems) then
        nel = num_elems - zero_elem
       else
        nel = chunk_size
       endif
c
c  Copy the elemIDs for all elems in the current chunk to a
c  temporary array
c
       do ielem = 1, nel
       chunk_ids(ielem) = elemID(zero_elem + ielem)
       enddo
c
c  Extract the element centroids and stresses
c
       call aupst_get_elem_centroid(nel, chunk_ids,
     &                              elem_centroid,
     &                              ierror)
```

```
      call aupst_get_elem_var(nel, 6, chunk_ids,
     &                        input_stress_val,
     &                        input_stress_name, ierror)
c
c  Loop over each element in the current chunk
c
      do ielem = 1, nel
c
c  Find the closest point on the cylinder centerline axis
c  to the element centroid
c
        cent(1) = elem_centroid(1, ielem) - origin_point(1)
        cent(2) = elem_centroid(2, ielem) - origin_point(2)
        cent(3) = elem_centroid(3, ielem) - origin_point(3)
        dot_prod = cent(1) * z_vec(1) +
     &             cent(2) * z_vec(2) +
     &             cent(3) * z_vec(3)
        centerline_pos(1) = z_vec(1) * dot_prod
        centerline_pos(2) = z_vec(2) * dot_prod
        centerline_pos(3) = z_vec(3) * dot_prod
c
c  Compute the current normal radial vector
c
        r_vec(1) = cent(1) - centerline_pos(1)
        r_vec(2) = cent(2) - centerline_pos(2)
        r_vec(3) = cent(3) - centerline_pos(3)
        call aupst_unitize_vector(r_vec, ierror)
        if(ierror .ne. 0) return
c
c  Compute the current hoop vector
c
        theta_vec(1) = z_vec(2)*r_vec(3) - r_vec(2)*z_vec(3)
        theta_vec(2) = z_vec(3)*r_vec(1) - r_vec(3)*z_vec(1)
        theta_vec(3) = z_vec(1)*r_vec(2) - r_vec(1)*z_vec(2)
c
c  The r, theta, and z vectors describe the new stress
c  coordinate system, Transform the input stress tensor
c  in x,y,z coords to the output stress tensor in r, theta,
c  and z coords use the unit vectors to create a rotation
c  tensor
c
        rotation_tensor(k_f36xx) = r_vec(1)
        rotation_tensor(k_f36yx) = r_vec(2)
        rotation_tensor(k_f36zx) = r_vec(3)
        rotation_tensor(k_f36xy) = theta_vec(1)
        rotation_tensor(k_f36yy) = theta_vec(2)
        rotation_tensor(k_f36zy) = theta_vec(3)
```

```
        rotation_tensor(k_f36xz) = z_vec(1)
        rotation_tensor(k_f36yz) = z_vec(2)
        rotation_tensor(k_f36zz) = z_vec(3)
c
c  Rotate the current stress tensor to the new configuration
c
        call fmth_rotate_symten33(1, 1, 0, rotation_tensor,
     &                            input_stress_val(1,ielem),
     &                           output_stress_val(1,ielem))
       enddo
c
c  Store the new stress
c
        call aupst_put_elem_var(nel, 6, chunk_ids,
     &                           output_stress_val,
     &                           output_stress_name, ierror)
      enddo
      ierror = 0
      end
```

10.5. User Subroutine Library

Many user subroutines are used commonly and have been permanently incorporated into the code. These subroutines are used just like any other subroutines, but they do not need to be manually compiled into the code. (The user need be concerned only about the Sierra/SM command lines.) Additionally the subroutines listed here are regularly tested with Sierra/SM and formally maintained by the development team.

This section describes the usage of each of these user subroutine library subroutines..

10.5.1. aupst_preload_solver

Author: Nathan Crane

Note, use of the 'aupst_preload_solver' subroutine is also documented in the example manual chapter 'Automated Adaptive Preloading.'

Purpose:

This subroutine is used to solve simple inverse problems for preload conditions determining the value of a working variable to achieve some target condition. Examples where this subroutine has been successfully used include: apply the target strain in bolts to yield specified clamping force in each bolt, find the force that must be applied to a pin to reach a target displacement in that pin, and to find the displacement at a fixture to yield a target contact force on a body.

There are a number of restrictions on the relations that can be effectively solved by this subroutine. The first is that the relationship is monotonic. For example adding more strain to bolt increases the clamping force. If the relationship is not monotonic the solution may stall at a local maximum. The second restriction is that the relationship is at least somewhat smooth. The relationship between changes in the working variable to the target condition does not need to be linear, but should be at least nominally C^0 continuous. Third the relationship must be somewhat strong. Again as an example changing the artificial strain in a bolt strongly affects its clamping force.

The subroutine itself is generally called by a `USER OUTPUT` command block and often needs to be combined with a number of other commands as shown in this example:

```
#  Drive element preload as a variable set by
#  aupst_preload_solver subroutine
begin function bolt_preload
  type is analytic
  expression variable: v = element applied_strain
  evaluate expression = "v"
end
#  Internal variables needed by subroutine
begin user variable applied_strain
  type = element real length = 1
```

```
  use with restart
end
begin user variable bolt_preload_state
  type = element real length = 12
  use with restart
end
#  Use the 'working' variable to apply strain to bolt
begin artificial strain
  block = block_201
  r function = sierra_constant_function_zero
  s function = sierra_constant_function_zero
  t function = bolt_preload
end
#  Compute the currently obtained clamping force
#  on the bolt head.
begin user output
  surface = surface_200
  compute global bolt_force as sum of nodal force_contact(z)
end
# Solver for preload.  Update the working variable to try
# to make the target_variable equal to the target value
begin user output
  block = block_201
  subroutine real parameter:   target_value     = 1.0e+7
  subroutine real parameter:   initial_guess    = -3.06e-4
  subroutine real parameter:   iteration_time   = 5.0e-5
  subroutine string parameter: target_variable  = global bolt_force
  subroutine string parameter: working_variable = element applied_strain
  subroutine string parameter: state_variable   = element bolt_preload_state
  element block subroutine = aupst_preload_solver
  compute at every step
end
```

Parameters:

The meaning of some of the parameters are shown visually in Figure 10-2. For the bolt preload example the target artificial strain starts low and monotonically asymptotes up to the correct solution over several predictor corrector iteration cycles.

Subroutine Operation:

The subroutine drives a predictor correct iteration in time. Each predictor corrector cycle is performed over the input deck defined 'iteration_time.' The step cycle is shown in Figure 10-3 the value of the working variable produced by the subroutine is shown in blue and an example value of the response of the target variable shown in yellow. During the first and second quarter of the cycle the working variable is slowly increased from its old value to the new value. This ramp up is done with a smooth half cycloidal loading ramp. On the first cycle this new value will be half of the initial guess. During the third quarter of the cycle the working variable is held constant. Ideally damping terms will settle the value of the target variable to some relatively constant value

Parameter Name	Type	Description
target_value	Real	The value that the target_variable is trying to reach. In this example the target clamping force in the bolt
initial_guess	Real	An initial guess for the working_variable. In this example the initial guess for artificial strain.
iteration_time	Real	Amount of analysis time each predictor-corrector iteration takes
target_variable	String	Type and name of the target_variable. In this example the current clamping force in the bolt as computed by the sum of contact force on the bolt head.
working_variable	String	Type and name of the working_variable. In this example the artificial strain magnitude applied to each element of the bolt
state_variable	String	Type and name of state variables used internally by the subroutine. State variable type must match working variable type.

Table 10-27. Required aupst_preload_solver Parameters

during this time. In the fourth quarter of the load cycle the working variable is again held constant and the target variable is evaluated. The evaluation is the time average of the target variable over this fourth of the cycle time. Finally, at the end of the cycle, the new value for the next cycle working variable is defined based on the tangent of the response from the previous to the current iteration. The new working variable value is set such that the error in the target variable versus the target value will be roughly halved. This one half safety factor is used in an attempt to converge the solution monotonically from below. Additionally, regardless of the function tangent computed, the working variable can change by no more than the initial guess each iteration. This is an additional safety factor to prevent overshooting the solution.

It is important to consider that these predictor-corrector iterations only make sense if the target variable reaches a reasonably stable value by the fourth part of each iteration to make a reliable next prediction. For explicit transient dynamics these preload iterations should usually encompass several thousand time steps, but the actual number of steps needed may vary.

The preload solver subroutine will also operate with implicit statics (adagio). One consideration in this case is that each preload iteration cycle will take a minimum of three time steps: a ramp-up step, a hold step, and an evaluation step.

State Variables:

The subroutine state variables are used primarily for the internal operation of the subroutine. These variables can be also be post processed to understand how the subroutine is operating. The description of each variable is shown in the below Table 10-28. Depending on how the subroutine

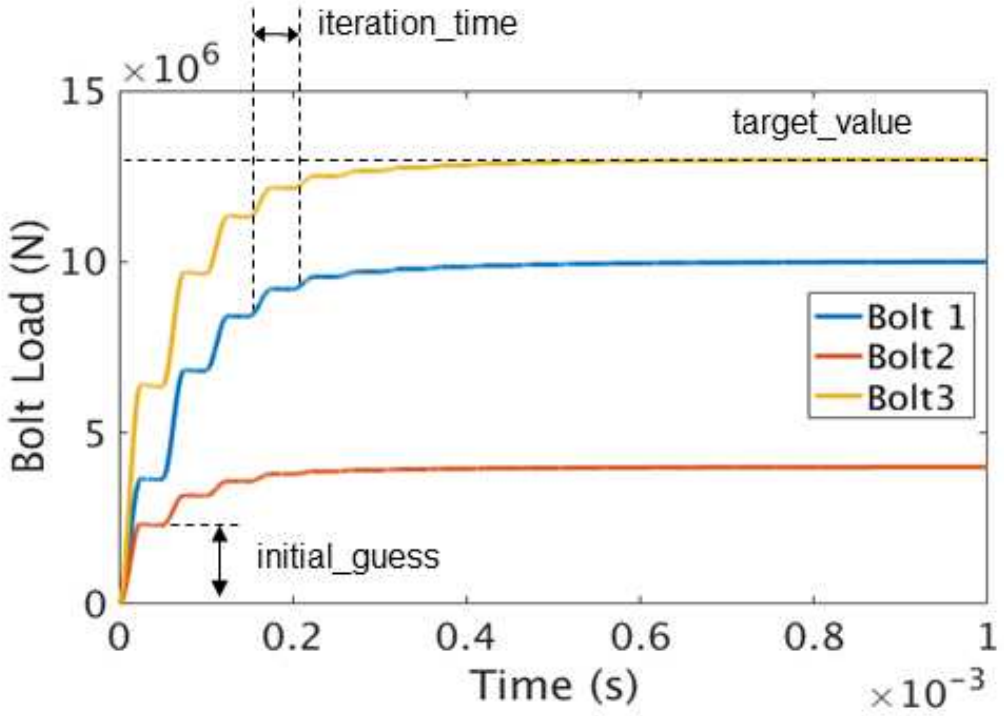


Figure 10-2. Example of Preload Iteration

is used these state variables could be stored in a global variable or there could be a set of state variables at each node or element that is being loaded. The twelfth state variable, which indicates the current error in the preload solution can be particularly useful to look at.

Usage Guidelines and Warnings:

Pay special care to the 'use with restart' and 'compute at every step' commands present in the example input blocks. The 'use with restart' flag ensures the state variables are properly stored in any restart database ensuring the subroutine works correctly with a restart analysis. The 'compute at every step' command ensures the preload solver correctly updates each an every step. Due to its presence in the 'user output' command block without this command the preload solver would only evaluate at time steps where output is performed, likely yielding strange results in the analysis.

The iteration time must be long enough that the target variable can re-equilibrate to some relatively constant value in time in the target variable evaluation part of the cycle. The subroutine will compare previously found target variable values to current target variable values to make a new prediction for the working variable. If the target variable has a large dynamic oscillation that causes its value to change unrelated to the working variable change this prediction will be poor and the solver will not function correctly.

The analysis must run for enough time that a sufficient number of predictor corrector preload cycles are taken to converge the target variable to the target value. How many cycles are required

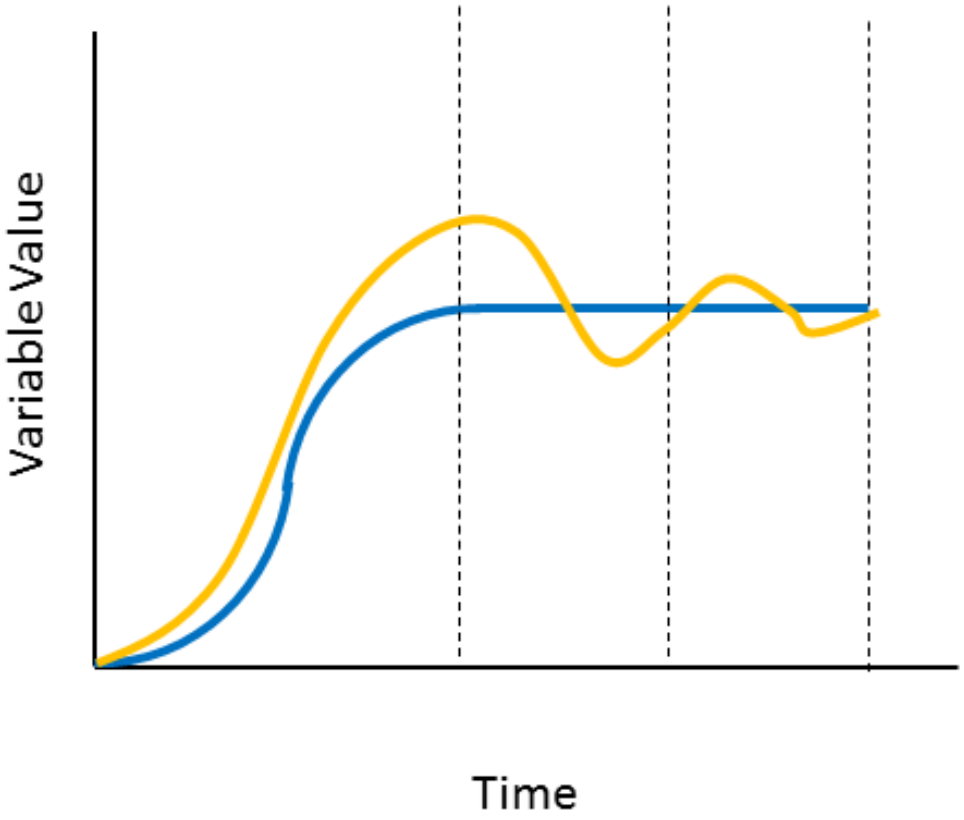


Figure 10-3. Anatomy of the Preload Iteration Cycle

will depend on how close the initial guess is. However, due to conservative assumptions in the solver usually at least 10 cycles will be required even if the initial guess is good.

This subroutine uses a simple predictor corrector approach and can only solve for conditions where the target versus working variable response is both monotonic and relatively smooth (i.e., at least C^0 continuous).

It is strongly recommended that results be investigated both to ensure an acceptable error bound has been reached and that a reasonable history of the working variable was applied. For example greatly overshooting the target on the initial guess could cause spurious yield which would then be locked into the solution.

A few different combinations of variable types may be used. The working and state variable must be the same type. The target variable must be either a global or the same type as the working and state variables.

State Number	Description
1	Start time of current preload iteration
2	End time of current preload iteration
3	Value of working variable at start of current preload iteration
4	Value of working variable at end of current preload iteration
5	Iteration Phase. 1=ramp up (1st and 2nd quarter), 2=hold (3rd quarter), 3=hold/evaluate (4th quarter)
6 7	Variables 6 and 7 are used to compute a running average of the target variable during the hold/evaluate phase
8	Initial (very first time) value of the working variable
9	Initial (very first time) value of the target variable
10	The time the subroutine was last evaluated
11	The target variable that was computed in the previous preload iteration
12	The relative solution error in target value versus variable error starts at 1.0 and ideally reaches 0.0

Table 10-28. aupst_preload_solver Subroutine State Variables

10.5.2. aupst_torque_calc

Author: Nathan Crane

Purpose:

This subroutine computes the net torque applied to a body by a set of forces. The torque is defined around an axis defined by two points. The output torque is given as a single value about this axis with a sign following the right hand rule. This subroutine is generally called by a `USER OUTPUT` command block. For example:

```
begin user output
  node set = nodelist_1
  compute global ns1 as average of nodal coordinates
end
begin user output
  node set = nodelist_2
  compute global ns2 as average of nodal coordinates
end
begin user variable moment1
  type = global real length = 1
  global operator = sum
end
begin user output
```

```
  surface = surface_1    #Contains nodes over which to do
                         # moment summation
  node set subroutine = aupst_torque_calc
  subroutine string parameter: axis_node_1 = ns1
  subroutine string parameter: axis_node_2 = ns2
  subroutine string parameter: force_field = reaction
  subroutine string parameter: output_var = moment1
end
```

Parameters:

Parameter Name	Type	Description
axis_node_1	String	Name of global variable containing location of first axis node
axis_node_2	String	Name of global variable containing location of second axis node
force_field	String	Name of field containing force values to sum into the torque
output_var	String	Name of output global variable to store the computed moment

Table 10-29. Required aupst_torque_calc Parameters

10.5.3. aupst_trans_rot_velocity

Author: Nathan Crane

Purpose:

This subroutine sets an initial velocity on a part that is the combination of an arbitrary rigid body spin rate superimposed on an arbitrary translational velocity. This subroutine is generally called by a `INITIAL VELOCITY` command block. For example:

```
begin initial velocity
   block = block_2
   node set subroutine = aupst_trans_rot_velocity
   subroutine real parameter: center_x = 0.0
   subroutine real parameter: center_y = 0.0
   subroutine real parameter: center_z = 2.0

   subroutine real parameter: rot_dir_x = 1.0
   subroutine real parameter: rot_dir_y = 1.0
   subroutine real parameter: rot_dir_z = 1.0
```

```
  subroutine real parameter: angular_velocity = 628.31853

  subroutine real parameter: tran_dir_x = 1.0
  subroutine real parameter: tran_dir_y = 1.0
  subroutine real parameter: tran_dir_z = 1.0

  subroutine real parameter: translational_velocity = 173.20508
end
```

Parameters:

Parameter Name	Type	Description
center_x, center_y, center_z	Real	Coordinates of center of rotation
rot_dir_x, rot_dir_y, rot_dir_z	Real	Direction of axis of rotation
tran_dir_x, tan_dir_y, tan_dir_z	Real	Direction of translational velocity
angular_velocity	Real	Spin rate in radians per time
translational_velocity	Real	translational velocity in distance per time

Table 10-30. Required aupst_trans_rot_velocity Parameters

10.5.4. aupst_cyl_transform

Author: Nathan Crane

Purpose:

This subroutine transforms element stresses from a global rectangular coordinate system to a local cylindrical coordinate system. This subroutine is generally called by a `USER OUTPUT` command block. For example:

```
begin user output
  block = block_1
  element block subroutine = aupst_cyl_transform
  subroutine string parameter: origin_point = Point_O
  subroutine string parameter: z_point      = Point_Z
  subroutine string parameter: xz_point     = Point_XZ
  subroutine string parameter: input_stress = memb_stress
  subroutine string parameter: output_stress = cyl_stress
end user output
```

Requirements:

This subroutine requires a tensor variable to store the cylindrical stress into a variable for each element. The variable is created by the following command block in the Sierra/SM region:

```
begin user variable cyl_stress
   type = element sym_tensor length = 1
   initial value = 0.0 0.0 0.0 0.0 0.0 0.0
end user variable
```

Parameters:

Parameter Name	Usage	Description
origin_point	String	Name of the point at the cylinder origin.
z_point	String	Point on the cylinder axis.
xz_point	String	Point on the line that passes through theta = 0 on the cylinder.
input_stress	String	Name of the Sierra/SM internal input stress tensor variable.
output_stress	String	Name of the Sierra/SM internal output stress tensor variable.

10.5.5. aupst_rec_transform

Author: Daniel Hammerand

Purpose:

This subroutine transforms element stresses from a global rectangular coordinate system to a different local rectangular coordinate system. This subroutine is generally called by a `USER OUTPUT` command block. For example:

```
begin user output
  block = block_1
  element block subroutine = aupst_rec_transform
  subroutine string parameter: origin_point = Point_O
  subroutine string parameter: z_point      = Point_Z
  subroutine string parameter: xz_point     = Point_XZ
  subroutine string parameter: input_stress = memb_stress
  subroutine string parameter: output_stress = new_stress
end user output
```

Requirements:

This subroutine requires a tensor variable to store the new stress into a variable for each element. The variable is created by the following command block in the Sierra/SM region:

```
begin user variable new_stress
  type = element sym_tensor length = 1
  initial value = 0.0 0.0 0.0 0.0 0.0 0.0
end user variable
```

Parameters:

Parameter Name	Usage	Description
origin_point	String	Name of the point at the cylinder origin.
z_point	String	Point on the cylinder axis.
xz_point	String	Point on the line that passes through theta = 0 on the cylinder.
input_stress	String	Name of the Sierra/SM internal input stress tensor variable.
output_stress	String	Name of the Sierra/SM internal output stress tensor variable.

10.5.6. copy_data

Author: Jason Hales

Purpose:

This subroutine copies data from one variable to another with offsets given for both variables. This subroutine is generally called by a `USER OUTPUT` command block. For example:

```
begin user output
  block = block_1
  element block subroutine = copy_data
  subroutine integer parameter: source_offset = 4
  subroutine string parameter:  source_name = stress
  subroutine integer parameter: destination_offset = 1
  subroutine string parameter:  destination_name = uservarxy
end user output
```

Requirements:

This subroutine requires that the source and destination fields exist and have lengths at least as great as the values supplied as offsets. The fields used may be defined by the user as variables. In this example, the variable is created by the following command block in the Sierra/SM region:

```
begin user variable uservarxy
  type = element real length = 1
  initial value = 0.0
end user variable
```

Parameters:

Parameter Name	Usage	Description
source_offset	Integer	The offset into the source variable.
source_name	String	The name of the source variable.
destination_offset	Integer	The offset into the destination variable.
destination_name	String	The name of the destination variable.

10.5.7. trace

Author: Jason Hales

Purpose:

This subroutine computes the trace of a tensor. This subroutine is generally called by a `USER OUTPUT` command block. For example:

```
begin user output
  block = block_1
  element block subroutine = trace
  subroutine string parameter:  source_name = log_strain
  subroutine string parameter:  destination_name = uvarbulkstrain
end user output
```

Requirements:

This subroutine requires that the source and destination fields exist. The source field should have a length of six. The destination field should have a length of one. The destination field will typically be defined by the user as a variable. In this example, the variable is created by the following command block in the Sierra/SM region:

```
begin user variable uvarbulkstrain
  type = element real length = 1
  initial value = 0.0
end user variable
```

Parameters:

Parameter Name	Usage	Description
source_name	String	The name of the source variable.
destination_name	String	The name of the destination variable.

10.5.8. aupst_cavity_volume

Author: Nathan Crane

Purpose:

Compute the volume of an enclosed side set. This subroutine is generally called by a `USER OUTPUT` command block. For example:

```
begin user output
  SURFACE = surface_4000
  SURFACE SUBROUTINE = aupst_cavity_volume
  subroutine string parameter: outputVarName = s4000_vol

  SUBROUTINE REAL PARAMETER: cavitySymmetryPoint_x = 0.0
  SUBROUTINE REAL PARAMETER: cavitySymmetryPoint_y = 0.0
  SUBROUTINE REAL PARAMETER: cavitySymmetryPoint_z = 0.0
  SUBROUTINE REAL PARAMETER: volumeScale           = 2.0

  compute at every step
end
```

Availability:

Compiled into Adagio 4.24 and later.

Compatible with Presto 2.2 / Adagio 2.2 and later.

Parameters:

Name	Type	Description
outputVarName	String	Name of global variable in which to store the computed volume.
cavitySymmetryPoint_x	Real	Optional: X coordinate of a point on the intersection of all model symmetry planes. Needed if the cavity is not fully enclosed due to open symmetry planes.
cavitySymmetryPoint_y	Real	Optional: Y coordinate of a point on the intersection of all model symmetry planes. Needed if the cavity is not fully enclosed due to open symmetry planes.
cavitySymmetryPoint_z	Real	Optional: Z coordinate of a point on the intersection of all model symmetry planes. Needed if the cavity is not fully enclosed due to open symmetry planes.
volumeScale	Real	Optional: Value to multiple the final cavity volume by. Defaults to 1.0. May be used to adjust for symmetry boundary conditions.

11. TRANSFERS

It is sometimes desirable to chain two or more analyses (procedures) together. A common example of this is the need to preload an assembly quasistatically, then subject that assembly to a loading environment best suited to an explicit transient dynamics analysis. The displacements and stresses produced by the quasistatic preload become initial conditions for the transient dynamics event. These displacements and stresses must therefore be transferred from the initial to the subsequent analysis.

This chapter reviews the concept of transfers in SIERRA and outlines the syntax required to perform a transfer of information between procedures.

11.1. SIERRA Transfers

Applications built on the SIERRA computational framework share underlying data structures. This makes it convenient to couple applications together using transfers.

The coupling available through SIERRA is of two types. The first is what is called intra-procedural coupling. In this case, multiple regions within a single procedure share data. This enables multiphysics analysis such as thermal-mechanical coupling. The details of this type of coupling, along with the syntax to support it, will not be covered here. Coupled codes such as Arpeggio use this type of coupling.

The second type of coupling is inter-procedural coupling. Here, the result from one procedure is handed to the next procedure. This is the type of coupling used when moving from one analysis stage to another in Adagio and Presto. When using this type of coupling, the two procedures generally have only one region each.

11.2. Inter-procedural Transfers

The inter-procedural transfers used by Adagio and Presto can transfer data from one or all blocks of the preceding or sending procedure to the subsequent or receiving procedure. The commands to control the transfers should appear at the procedure scope in the receiving procedure.

When using inter-procedural transfers, irrespective of the transfer type chosen, all global variables are copied by default from the sending procedure to the receiving procedure, and all appropriate element data is transferred from the sending to the receiving elements. Nodal data is transferred based on the type of analysis done in the two procedures. For example, if the first procedure is a quasistatic analysis and the second is an explicit transient dynamics analysis, the displacements of the nodes will be transferred but not their velocities.



Warning: Transfer of data for node-based tetrahedra is currently not fully supported. A coupled analysis with node-based tetrahedra will only be correct if the tetrahedral elements are not deformed in the first (sending) procedure, in which case initialization in the second (receiving) procedure is appropriate.

The available commands are given below.

```
BEGIN PROCEDURAL TRANSFER [<string>name]

  BLOCK = <string list>block_name
  INCLUDE ALL BLOCKS
  REMOVE BLOCK = <string list>block_name

  ADDITIONAL ELEMENT VARIABLES = <string list>addElemVar
  ELEMENT VARIABLES = <string list>elemVar
  ELEMENT LIE VARIABLES = <string list>elemLieVar
  NODE VARIABLES =  <string list>nodeVar
  GLOBAL VARIABLES = <string list>globalVar

  ABSOLUTE SEARCH TOLERANCE = <real>tolerance (1.0e-5)

  BEGIN INTERPOLATION TRANSFER [<string>name]

    BLOCK BY BLOCK
    NEAREST ELEMENT COPY

    SEND BLOCKS = <string list>block_name
    SEND COORDINATES = ORIGINAL|CURRENT(ORIGINAL)

    RECEIVE BLOCKS = <string list>block_name
    RECEIVE COORDINATES = ORIGINAL

    TRANSFORMATION TYPE = NONE|RIGIDBODY|REFLECT|WEDGE23D
```

```
    # ADDITIONAL REFLECT TRANSFER SPECIFICATION
    REFLECT PLANE = XY|YZ|ZX

    # ADDITIONAL WEDGE23D TRANSFER SPECIFICATION
    W23D ANGLE = <real>wedgeAngle
    W23D PLANE = XY|XZ|YX|YZ|ZX|ZY
    W23D REVOLVE ANGLE = <real>Angle

    NODAL TRANSFER TYPE = INTERPOLATE|GLOBAL_INTERPOLATION
                          (GLOBAL_INTERPOLATION)

  END [INTERPOLATION TRANSFER <string>name]

  BEGIN L2_PROJECTION TRANSFER [<string>name]

    BLOCK BY BLOCK

    SEND BLOCKS = <string list>block_name
    SEND COORDINATES = ORIGINAL|CURRENT(ORIGINAL)

    RECEIVE BLOCKS = <string list>block_name
    RECEIVE COORDINATES = ORIGINAL

    TRANSFORMATION TYPE = ELEMENT2NODES|ELEMENT2ELEMENT

    LINEAR SOLVER = <string>solver_name(DEFAULT_FETI_SOLVER)

    PROJECTION TYPE = CONSISTENT|ROW_SUM_LUMPED|
       PROPORTIONAL_LUMPED(CONSISTENT)

    NODAL TRANSFER TYPE = INTERPOLATE|GLOBAL_INTERPOLATION
                          (INTERPOLATE)

  END [L2_PROJECTION TRANSFER <string>name]
END [PROCEDURAL TRANSFER <string>name]
```

If the *same* finite element model is used in the sending and receiving regions, data may be *copied* directly from the sending to the receiving region. In this case, use of the `BLOCK`, `INCLUDE ALL BLOCKS`, and `REMOVE BLOCK` command lines alone will suffice to copy data for all blocks except those that are rigid bodies.

If *different* finite element models are used in the sending and receiving regions, or it is desired to transfer data for rigid bodies, an `INTERPOLATION TRANSFER` or `L2_PROJECTION TRANSFER` block must be used, and the `BLOCK`, `INCLUDE ALL BLOCKS`, and `REMOVE BLOCK` command lines in the `PROCEDURAL TRANSFER` block should *not* be used. If these commands, appropriate only for the direct *copy* of data, are used when not applicable, their behavior is undefined.

11.2.1. Copying Data with Inter-procedural Transfers

To copy data from one or more blocks in a sending region to the matching blocks in the receiving region, use these line commands:

```
BLOCK = <string list>block_name
INCLUDE ALL BLOCKS
REMOVE BLOCK = <string list>block_name
```

The first of these, the `BLOCK` line command, is used to list blocks that should be copied. If it is desired to copy data for all blocks, use the `INCLUDE ALL BLOCKS` line command. Finally, one or more blocks may be removed from the set of all blocks with the `REMOVE BLOCK` line command. Either the `BLOCK` or `INCLUDE ALL BLOCKS` line command must appear for copy transfers.

11.2.2. Selecting Variables for Transfer

By default, the code determines the variables to transfer: all global variables, and the sets of node and element variables necessary to initialize a new procedure where the previous procedure finished. Additional variables may be added to the default set with the `ADDITIONAL ELEMENT VARIABLES` command.

Alternatively, a manual subgroup of variables may be transferred with a combination of the `ELEMENT VARIABLES`, `ELEMENT LIE VARIABLES`, `NODE VARIABLES`, and `GLOBAL VARIABLES`. If any of these commands are used then all default variable transfers will be deactivated and all desired transfer variables *must be manually specified*.

The `ELEMENT LIE VARIABLES` designation indicates that variables should undergo a specialized mapping for interpolation to preserve their spaces. Specifically, a logarithmic mapping of the scalar, vector, or tensor field is performed before any transfer operation is initiated. The logarithmically mapped fields are transferred and interpolated to the elements of the receiving mesh, after which the values are exponentially mapped. This transfer method is more accurate, and appropriate for only variables that belong to Lie groups. In practice, these variables include rotations, deformation gradients, and stretches. The `ELEMENT LIE VARIABLES` option is valid only when an L_2 projection transfer is specified.



Warning: Manual selection of element variables is only functional with L_2 projection transfer.

11.2.3. Additional Transfer Options

Additional procedural transfer options include the ability to specify the `absolute search tolerance`. The `absolute search tolerance` is the absolute amount that the search bounding box should be expanded in each of the six global directions (+/- x, y, and z) for variable transfers. This tolerance is applied to the sending region to determine which

sending mesh objects (nodes or elements) contain, or are nearby, each receiving mesh object. If the bounding box search returns zero or multiple sending mesh objects, then a more expensive inverse isoparametric mapping search routine is invoked. The default `absolute search tolerance` is 1.0e-5.

11.2.4. Interpolating Data with Interpolation Transfers

When different finite element models are used by the sending and receiving regions, or when it is desired to transfer data for rigid bodies, the `INTERPOLATION TRANSFER` or `L2_PROJECTION TRANSFER` must be used. The `INTERPOLATION TRANSFER` uses a local interpolation patch. The `L2_PROJECTION TRANSFER` relies on extrapolation of element fields to nodes and interpolation of those fields back to elements, which involves solving a global minimization problem. Generally, the `L2_PROJECTION TRANSFER` is more accurate. For both the `INTERPOLATION TRANSFER` and `L2_PROJECTION TRANSFER`, nodal variables are interpolated from the sending mesh to the receiving mesh.

The `TRANSFORMATION TYPE` command inside of `INTERPOLATION TRANSFER` has the following options:

`NONE` (default) Do interpolation with no special transformation.

`RIGIDBODY` Transfer the rigid body information of each rigid body separately. Specify one and only one block in the `SEND BLOCKS` and `RECEIVE BLOCKS` line commands.

`REFLECT` Transform from a half-symmetric sending model to a full receiving model.

`WEDGE23D` Transform from a sending axisymmetric wedge model to a receiving full 3D cylinder model.

The `TRANSFORMATION TYPE` command inside of `L2_PROJECTION TRANSFER` has the following options:

`ELEMENT2NODES` Project element variables on the sending model to nodal variables on the receiving model.

`ELEMENT2ELEMENT` Project element variables from the elements to the nodes of the sending model, transfer these projected nodal values from the sending to the receiving model via the normal interpolation transfer, then interpolate these nodal values to element variables on the receiving model. For continuous fields, this is generally the most accurate transfer method. This transfer type is combined with Lie algebra for variables belonging to Lie groups to preserve spaces of such variables.

There is no default for transformation type in `L2_PROJECTION TRANSFER` and one of the options must be specified.

When transferring data for non-rigid body blocks, the only required line commands are the `SEND BLOCKS` and `RECEIVE BLOCKS` line commands. These each list one or more blocks to be included in the transfer.

Interpolation transfers move data from sending to receiving meshes by performing searches and interpolating values using shape functions. Interpolation transfers will give the most accurate

results when the sending and receiving meshes are similar. If the sending and receiving meshes do not occupy the same 3D space, for instance, transferred data is likely to be highly inaccurate.

In some instances, the sending and receiving meshes are highly similar such that there is a one-to-one correspondence between the list of sending blocks and the list of receiving blocks. In other words, it may be desired to send data from a given block to a corresponding block, from another block to its pair, and so forth. If this is the case, use of the `BLOCK BY BLOCK` line command will cause a separate transfer to be created for each pair of sending and receiving blocks. This is useful to ensure that data from a single block will be sent to one and only one receiving block. This could also be accomplished by listing multiple transfer blocks in the input file. Note that if using the `BLOCK BY BLOCK` line command, the names of the blocks listed in `SEND BLOCKS` and `RECEIVE BLOCKS` must be identical.

For certain use cases, such as remeshing for large deformations, it is desired to remesh on the deformed (current) configuration and adopt that as the new reference (model) configuration. For these use cases, use the command `SEND COORDINATES = CURRENT` instead of the default `SEND COORDINATES = ORIGINAL` so that the current configuration of the sending mesh is mapped to the model configuration of the receiving mesh when performing the search used by the transfers. Displacements are not transferred for this use case since deformations are assumed to be accounted for in the receiving mesh, i.e., the mesh was generated on the deformed configuration).



Warning: Adoption of a new reference configuration on the receiving mesh is currently only compatible with midpoint strain increment element formulations and hypo-elastic material models. All other element formulations and hyper-elastic materials require a multiplicative decomposition of the deformation gradient: the deformation gradient mapped from the sending mesh pre-multiplied by the deformation gradient due to the subsequent deformation applied on the receiving mesh. Such a multiplicative decomposition is not currently supported in transfers at this time.

11.2.4.1. Nodal variable interpolation transfer

The location of each receiving node in the receiving mesh is used to find a corresponding element in the sending mesh. The parametric coordinates of the coordinates associated with the receiving node within (or near) the sending element are calculated and subsequently used in conjunction with the element shape functions to interpolate the sending element nodal data onto each receiving node.

Nodal variables are always transferred using an `INTERPOLATION TRANSFER`. The `NODAL TRANSFER TYPE` command determines the send and receive sets over which nodal variables interpolate and has the following options:

`INTERPOLATE` (default for `L2_PROJECTION TRANSFER`) Nodal variables are transferred between the send and receive blocks defined within the same transfer.

GLOBAL_INTERPOLATION (default for INTERPOLATION TRANSFER) Nodal variables are transferred between all send and receive blocks in the mesh.

11.2.4.2. Element variable interpolation transfer

In an INTERPOLATION TRANSFER block, the center of each receiving element will be located within (or near) the closest element in the sending mesh. A patch of elements in the sending mesh centered around the element containing the receiving element center is created. The receiving element value is then computed by interpolating the data in this patch of sending elements, using a least squares approach, to the point associated with the receiving element.

When the NEAREST ELEMENT COPY line command is used, data is sent directly from the closest sending element to the receiving element, forgoing the patch construction and least squares interpolation. A NEAREST ELEMENT COPY transfer is expected to be faster but potentially less accurate than an INTERPOLATION TRANSFER.

11.2.4.3. Reflection (REFLECT) transfer

Simulation results from a half-symmetry model may be transferred onto a full geometry receiving mesh using the REFLECT interpolation transfer. The reflection plane must be on one of the global coordinate planes defined XY, YZ, or ZX. This plane should be additionally specified with the REFLECT PLANE = XY|YZ|ZX command.

11.2.4.4. Wedge-to-3D (WEDGE23D) transfer

Simulation results from an axisymmetric wedge submodel may be transferred onto a full geometry using the WEDGE23D interpolation transfer. The WEDGE23D transfer will transforms vector and tensor quantities as appropriate during the interpolation. One of the wedge faces should lie on a global coordinate plane to use this transfer type. Several additional commands may be used to further specify the transfer:

- W23D ANGLE The wedge angle should be provided in degrees. This command is used to determine how the submodel data is rotated and interpolated onto the full model.
- W23D PLANE The location of the wedge and the cylindrical axis used to revolve the wedge during reconstruction is specified as one of XY|XZ|YX|YZ|ZX|ZY. The first letter identifies the cylindrical axis. For example: if the wedge submodel is meshed with one of the wedge faces on the YX plane, and the wedge should be revolved around the y-axis to obtain the full geometry, then one should select the YX option rather than the XY option.
- W23D REVOLVE ANGLE The revolve angle command specifies the total angle, in degrees, of the target geometry. This value should be an integral multiple of the W23D ANGLE. In practice, this might be useful for reconstructing a quarter-symmetry or half-symmetry model from a smaller wedge.

11.2.4.5. Element variable L_2 projection transfer

In an `L2_PROJECTION` transfer block, element variables at the numerical quadrature points of the sending mesh are first projected, via a global least squares solution, to the nodes of the sending mesh. A nodal interpolation transfer (described above) is then performed to transfer data from the nodes of the sending mesh to nodes of the receiving mesh. Finally, nodal data on the receiving mesh is interpolated to the quadrature points of the receiving mesh. A L_2 projection transfer is expected to yield more accurate element fields than an interpolation transfer.

For each block specified in `SEND BLOCKS`, a separate linear system, based on the consistent mass matrix (refer to Section 9.3.2.5 for details), is assembled and solved. Each element variable is projected to a nodal variable unique to the sending block so that potentially discontinuous fields at neighboring block boundaries are handled consistently.

In L_2 projection transfers based on higher-order element topologies such as ten-noded tetrahedra, the topology is transformed into an equivalent linear topology for extrapolation. Extrapolated nodal values are then linearly interpolated to the high-order nodes. This strategy has been shown to avoid spurious oscillations in the extrapolated quantities.

A particular linear solver may be selected for the L_2 projection using the `LINEAR SOLVER` user output command. The default is "DEFAULT_FETI_SOLVER." Care should be taken to specify appropriate solver parameters and tolerances for the desired accuracy of the projection. Loose tolerances can significantly degrade the mapped solution. Generally, transfers consume a small percentage of the total run time, so a tight residual tolerance (e.g., 10^{-12}) is recommended.

The formation of the L2 projection matrix can be controlled via the `PROJECTION TYPE` command. The default value `CONSISTENT` constructs an L2 projection matrix through a consistent evaluation of the shape functions, which must subsequently be used with a linear solver to obtain the nodal values. The alternative options `ROW_SUM_LUMPED` and `PROPORTIONAL_LUMPED` use lumping schemes (row-sum lumping or diagonal proportional lumping schemes) to diagonalize the projection matrix. As a result, no iterative linear solve is necessary and the solver provided by `LINEAR SOLVER` is not used. This may be useful in specific cases where the iterative linear solver fails in parallel solutions.

11.3. Automated Implicit/Explicit Hand-off

Explicit/Implicit hand off analyses are a common use case for Sierra/SM. For example, a component may be preloaded in an implicit quasistatic procedure; then, the preloaded component is subjected to an impact loading in an explicit dynamics procedure. Multi procedure analysis may be setup manually by defining multiple analysis regions, procedures, time controls, and transfer blocks. However, this manual process is time consuming and error prone.

The same functionality described above can be accomplished with the much simpler syntax below. The following example time block will perform an implicit to explicit hand off analysis.

```
BEGIN SM PROCEDURE myProcedure
  BEGIN TIME CONTROL
    BEGIN TIME STEPPING BLOCK preload
      START TIME = 0.0
      BEGIN PARAMETERS FOR SM REGION myRegion
        NUMBER OF TIME STEPS = 5
        TIME INTEGRATION = implicit_statics
        ADVANCE ON SOLVER FAILURE = current_procedure
      END
    END
    BEGIN TIME STEPPING BLOCK impact
      START TIME = 0.1
      BEGIN PARAMETERS FOR SM REGION myRegion
        TIME INTEGRATION = explicit_dynamics
      END
    END
    termination time = 0.1001
  END
  BEGIN SM REGION myRegion
    ...Include region commands as normal here...
  END
END
```

The automatic time period capability is triggered by using the `BEGIN SM PROCEDURE` and `BEGIN SM REGION` command lines rather than using `BEGIN PRESTO REGION` or `BEGIN ADAGIO REGION` procedures and regions. When these `SM` commands are used, the input deck is automatically run though a preprocessor to turn the single region multi time period input deck into an equivalent multi-procedure analysis. This preprocessed input deck is then automatically run by the sierra script. If the input deck originally has a name such as 'plateImpact.i' the preprocessed input deck will be saved to 'plateImpact.sm.i.' The modified input deck will also be echoed in the run log file.

The input deck preprocessing currently does the following operations to aid in setting up multiple region analysis.

- Time periods with `TIME_INTEGRATION` of `implicit_statics` will be converted into implicit adagio regions. Time periods with the `TIME_INTEGRATION` of

explicit_dynamics will be converted into explicit presto regions. The appropriate begin and end times for each procedure and region will be automatically determined from the time period blocks defined in the 'sm' region. Additionally, the appropriate transfer operations between the regions will be automatically defined. The analysis procedures may switch back and forth between time integration schemes as many times as needed.

- The optional ADVANCE ON SOLVER FAILURE command can be used within an implicit analysis to define the course of action if the solver fails. The current_procedure option will kill the current procedure and start the next procedure using the last converged state of the solution. If the current_time_period option is used, the simulation will continue to the next time stepping block of the same procedure (if present). Keep in mind that the next time stepping block should not have a START TIME, since it will likely conflict with the last converged time of the previous time stepping block. The default option is no_advancement, which will terminate the simulation upon solver failure.
- Boundary conditions will be appropriately placed in each region based on their active periods.
- Each solution procedure will define its own results, history, and restart files. Output database names for results, history, and restart files will be automatically prepended with time period names to be unique for each analysis region. This file name modification prevents the output from one region overwriting the output of a previous region. If in the above example a results output file in the sm region is given the name 'myResult.e' the analysis will output two separate results files 'preload_myResult.e' and 'impact_myResult.e.'
- If solution termination is used to dynamically swap between time periods and procedures these swaps are setup to function correctly with the multi-procedure analysis during the preprocessing.

The SM region preprocessing assumes all solution procedures use the same mesh, blocks, and material definitions. Additionally, transfers are always setup to transfer all possible analysis state data from one solution procedure to the next. More complex multi-region analyses cases can always be setup manually. For such manual setups, the automatically generated preprocessed 'SM' input deck may still provide a good initial starting point for setting up a more complex input deck.

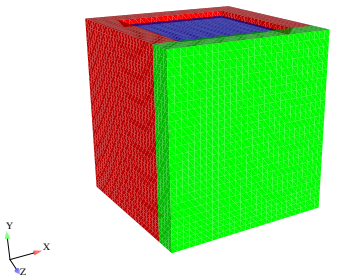
11.3.1. Known Issues and Limitations

Only capabilities that can be used in both implicit and explicit runs may be present in the SM region. For example this precludes use of SPH particles as this element formulation is only available in explicit analyses.

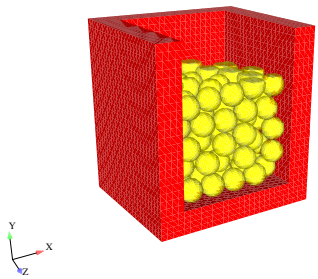
Hand-offs using an SM region cannot be run coupled to another physics code through solution control. An example would be an arpeggio (aria/adagio) coupled analysis.

APPENDIX A. Explicit Dynamic Example Problem

This chapter provides an example problem to illustrate the construction of an input file for an analysis. The example problem consists of 124 spheres made of lead enclosed in a steel box. The steel box has an open top into which a steel plate is placed (see Figure A-1). A prescribed velocity is then applied on the steel plate, pushing it into the box and crushing the spheres contained within using frictionless contact. This problem is a severe test for the contact algorithms as the spheres crush into a nearly solid block. See Figure A-2 for results of this problem.

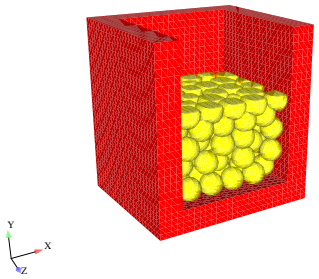


(a) Undeformed Mesh

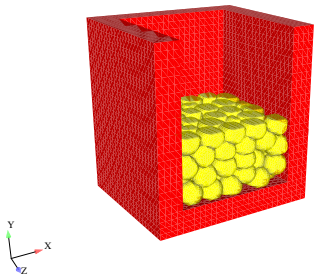


(b) Initial Crush of Spheres

Figure A-1. Mesh for example problem: (a) box (red and green surfaces) with plate in top (blue surface) and (b) mesh with blue and green surfaces removed to show internal spheres (yellow) with initial crush.



(a) Additional Crush of Spheres



(b) Final Deformed Configuration

Figure A-2. Mesh with blue and green surfaces removed to show internal spheres (yellow) after initial crush shown in Figure A-1 (b).

The input file is described below, with comments to explain every few lines. Following the description, the full input file is listed again. Most of the key words in this example are all lowercase, which is different from the convention we have used to describe the command lines in this document. However, all the lowercase usage in the next example is an acceptable format in Presto.

The input file starts with a begin sierra statement (i.e., the first line of the `SIERRA` command block), as is required for all input files:

```
begin sierra crush_124_spheres
```

We now need to define the functions used with this problem. The boundary conditions require a function for the initial velocity, as follows:

```
begin function constant_velocity
  type is piecewise linear
  ordinate is velocity
  abscissa is time
  begin values
    0.0 30.0
    1.0 30.0
  end values
end
```

To define the boundary conditions, we need to define the direction for the initial velocity—this is in the *y*-direction. We could also choose to specify the Y component for the initial condition, but this input file uses directions.

```
define direction y_axis with vector 0.0 1.0 0.0
```

Next we define the material models used for this analysis. This problem has two materials: steel for the box, and lead for the spheres. Both materials use the elastic-plastic material model (denoted as `elastic_plastic`).

```
begin material steel
  density = 7871.966988

  begin parameters for model elastic_plastic
    youngs modulus = 1.999479615e+11
    poissons ratio = 0.33333
    yield stress = 275790291.7
    hardening modulus = 275790291.7
    beta = 1.0
  end parameters for model elastic_plastic

end material steel

begin material lead
  density = 11253.30062

  begin parameters for model elastic_plastic
```

```
    youngs modulus = 1.378951459e+10
    poissons ratio = 0.44
    yield stress = 13789514.59
    hardening modulus = 0.0
    beta = 1.0
  end parameters for model elastic_plastic

end material lead
```

Now we define the finite element mesh. This includes specification of the file that contains the mesh, and a list of all the element blocks we will use from the mesh and the material associated with each block. The name of the file is `crush_124_spheres.g`. The specification of the database type is optional—ExodusII is the default. Currently, each element block must be defined individually. For this particular problem, all the spheres are the same element block. Each sphere is a distinct geometry entity, but all spheres constitute one element block in the Exodus II database. The three element blocks that make up the box and lid all reference the same material description. The material description is *not* repeated three times. The material description for steel appears once and is then referenced three times.

```
begin finite element model mesh1
  Database Name = crush_124_spheres.g
  Database Type = exodusII

  begin parameters for block block_1
    material = steel
    model =  elastic_plastic
  end parameters for block block_1

  begin parameters for block block_2
    material = steel
    model =  elastic_plastic
  end parameters for block block_2

  begin parameters for block block_3
    material = steel
    model =  elastic_plastic
  end parameters for block block_3

  begin parameters for block block_4
    material = lead
    model =  elastic_plastic
  end parameters for block block_4

end finite element model mesh1
```

As an alternative to referencing the material description for steel three times as done above, you could define multiple element blocks simultaneously on the same command line. Thus, the three

element block specifications with the material `steel` could be consolidated into one, as follows:

```
begin parameters for block block_1 block_2 block_3
  material = steel
  model =  elastic_plastic
end parameters for block block_1 block_2 block_3
```

At this point we have finished specifying physics-independent quantities. We now want to set up the Presto procedure and region, along with the `time control` command block. We start by defining the beginning of the procedure scope, the `time control` command block, and the beginning of the region scope. Only one `time stepping block` command block is needed for this analysis. The termination time is set to 7×10^{-4}.

```
begin presto procedure Apst_Procedure

begin time control
  begin time stepping block p1
    start time = 0.0
    begin parameters for presto region presto
      time step scale factor = 1.0
      time step increase factor = 2.0
      step interval = 25
    end parameters for presto region presto
  end time stepping block p1

  termination time = 7.0e-4
end time control

  begin presto region presto
```

Next we associate the finite element model we defined above (mesh1) with this presto region.

```
  use finite element model mesh1
```

Now we define the boundary conditions on the problem. We prescribe the velocity on the top surface of the box (`nodelist_100`) to crush the spheres, and we confine the bottom surface of the box (`nodelist_200`) not to move. Although we use node sets to define these boundary conditions, we could have used the corresponding side sets.

```
  begin prescribed velocity
    node set = nodelist_100
    direction = y_axis
    function = constant_velocity
    scale factor = -1.0
```

```
  end

  begin fixed displacement
    node set = nodelist_200
    components = Y
  end
```

Now we define the contact for this problem. For this problem, we want all four element blocks to be able to contact each other. In this case, we define the same contact characteristics for all interactions. Since no friction model is defined in the block below, the contact defaults to frictionless contact. (Numerous options exist to control the contact algorithm. The options you choose will affect contact algorithm efficiency and solution accuracy. See Chapter 8 to determine how to set input for the `CONTACT DEFINITION` command block to obtain the best level of efficiency and accuracy for your particular problem.)

```
  begin contact definition
    skin all blocks = on
    begin interaction defaults
      general contact = on
      self contact = on
    end
  end
```

Now we define what variables we want in the results file and how often we want this file to be written. Here we request files written every 7×10^{-6} sec of analysis time. This will result in results output at one hundred time steps (plus the zero time step) since the termination time is set to 7×10^{-4} sec. The output file will be called `crush_124_spheres.e`, and it will be an Exodus II file (the database type command is optional; it defaults to ExodusII). The variables we are requesting are the displacements and external forces at the nodes, the rotated stresses for the elements, the time step increment, and the kinetic energy.

```
  begin Results Output output_presto
    Database Name = crush_124_spheres.e
    Database Type = exodusII
    At Time 0.0, Increment = 7.0e-6
    nodal displacement as displ
    nodal force_external as fext
    element stress as stress
    global KineticEnergy as KE
    global timestep
  end
```

Now we end the presto region, presto procedure, and sierra blocks to complete the input file.

```
    end presto region presto
  end presto procedure Apst_Procedure
end sierra crush_124_spheres
```

Here is the resulting full input file for this problem:

```
begin sierra crush_124_spheres
  begin function constant_velocity
    type is piecewise linear
    ordinate is velocity
    abscissa is time
    begin values
      0.0 30.0
      1.0 30.0
    end values
  end function constant_velocity
  define direction y_axis with vector 0.0 1.0 0.0

  begin material steel
    density = 7871.966988

    begin parameters for model elastic_plastic
      youngs modulus = 1.999479615e+11
      poissons ratio = 0.33333
      yield stress = 275790291.7
      hardening modulus = 275790291.7
      beta = 1.0
    end parameters for model elastic_plastic

  end material steel

  begin material lead
    density = 11253.30062

    begin parameters for model elastic_plastic
      youngs modulus = 1.378951459e+10
      poissons ratio = 0.44
      yield stress = 13789514.59
      hardening modulus = 0.0
      beta = 1.0
    end parameters for model elastic_plastic

  end material lead

  begin finite element model mesh1
    Database Name = crush_124_spheres.g
    Database Type = exodusII

    begin parameters for block block_1
      material = steel
      model =  elastic_plastic
    end parameters for block block_1
```

```
  begin parameters for block block_2
    material = steel
    model =  elastic_plastic
  end parameters for block block_2

  begin parameters for block block_3
    material = steel
    model =  elastic_plastic
  end parameters for block block_3

  begin parameters for block block_4
    material = lead
    model =  elastic_plastic
  end parameters for block block_4

end finite element model mesh1

begin presto procedure Apst_Procedure

  begin time control
    begin time stepping block p1
      start time = 0.0
      begin parameters for presto region presto
        time step scale factor = 1.0
        time step increase factor = 2.0
        step interval = 25
      end parameters for presto region presto
    end time stepping block p1

    termination time = 7.0e-4
  end time control

  begin presto region presto

    use finite element model mesh1

    begin prescribed velocity
      node set = nodelist_100
      direction = y_axis
      function = constant_velocity
      scale factor = -1.0
    end prescribed velocity

    begin fixed displacement
      node set = nodelist_200
      components = Y
    end fixed displacement
```

```
      begin contact definition
        skin all blocks = on
        begin interaction defaults
          general contact = on
          self contact = on
        end
      end

      begin Results Output output_presto
        Database Name = crush_124_spheres.e
        Database Type = exodusII
        At Time 0.0, Increment = 7.0e-6
        nodal displacement as displ
        nodal force_external as fext
        element stress as stress
        global KineticEnergy as KE
        global timestep
      end results output output_presto

    end presto region presto
  end presto procedure Apst_Procedure
end sierra crush_124_spheres
```

APPENDIX B. Implicit Quasistatic Example Problem

This appendix provides an example problem to illustrate the construction of an input file for an analysis. The problem is modeled after a pencil/eraser that is pressed against and rubbed across a tablet. The pencil, eraser and tablet are represented by blocks 1, 2 and 3 respectively. Block 4 is used to apply kinematics directly to the eraser via tied contact between block 4 and block 2. Block 1 is superfluous except when viewed together with blocks 1, 2 and 3. Then they closely resemble a pencil with an eraser on a tablet. The problem demonstrates the overall structure of an input file and includes the use of both tied and frictional contact simultaneously, the multilevel solver, and a linear solver for preconditioning. A schematic of the problem is shown in Figure B-1 and the mesh is shown in Figure B-2.

The problem kinematics and loading are described briefly here. The loading has two phases: preload and sliding. In the preload phase, the tablet (block 3) is kinematically fixed in the x-y plane and a vertical load (in the z-direction) is applied to the tablet which presses the tablet against the eraser (block 2). The eraser is kinematically fixed to block 4 via tied contact (see Figure B-2). Block 4 is held fixed in all three coordinate directions for the compression phase. The initial configuration and deformations of the preload phase are shown in the first two snapshots on the left in Figure B-1. In the sliding phase, block 4 is kinematically prescribed to move along the x-direction, thus sliding or dragging the eraser along the tablet (see Figure B-1) while the tablet force is held constant.

The input file is described below, with comments to explain every few lines. Following the description, the full input file is listed again. All character strings in the input file are presented in lowercase, which is an acceptable format in Adagio.

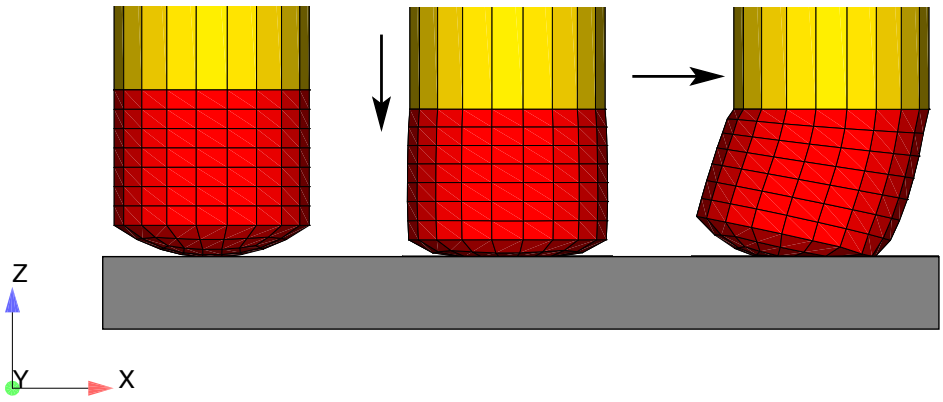


Figure B-1. Eraser schematic; Block 1 (yellow) is pencil, block 2 (red) is eraser, block 3 (gray) is tablet, block 4 (not shown) is tied to eraser and has the kinematics applied to it.

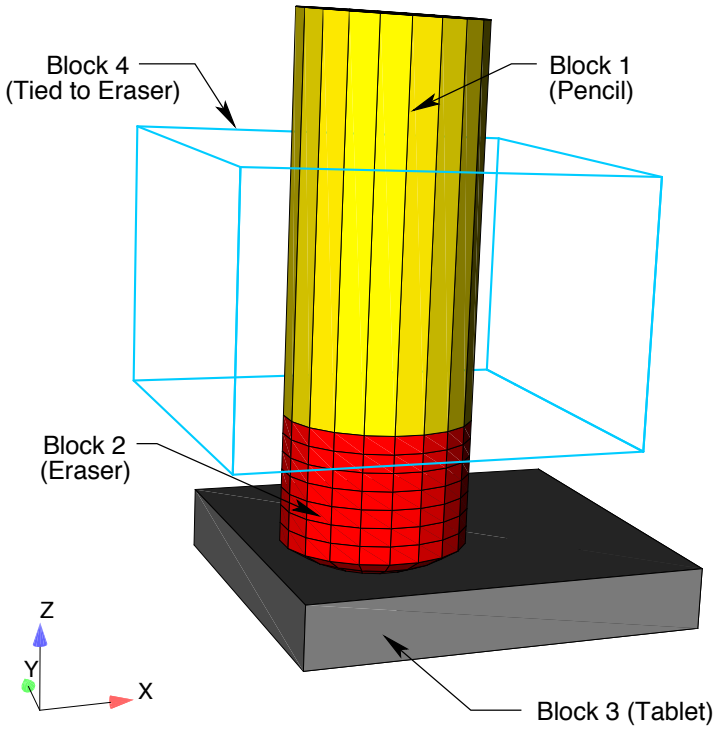


Figure B-2. Complete eraser mesh

The input file starts with a begin sierra statement, as is required for all input files:

```
begin sierra eraser
```

We begin by defining vectors corresponding to the coordinate axes. These vectors/directions will be used to define the input for boundary condition blocks that come later.

```
define direction X with vector 1.0 0.0 0.0
define direction Y with vector 0.0 1.0 0.0
define direction Z with vector 0.0 0.0 1.0
```

We now need to define the functions used for this problem. The boundary conditions require a function for the tablet force and the sliding motion. Both the tablet force and sliding motion are prescribed as functions of time.

```
begin function slide
  type is piecewise linear
  begin values
    0.0       0.0
    1.0       0.0
    2.0       1.0
```

```
    3.0      1.0
  end values
end function slide

begin function tablet_force
  type is piecewise linear
  begin values
    0.0      0.0
    1.0      1.0
    3.0      1.0
  end values
end function tablet_force

begin function zero
  type is constant
  begin values
    0.0
  end values
end function zero
```

The tablet force ramps up over the time interval (0.0,1.0) and then is held constant over the interval (1.0,3.0). The sliding function is zero over the time interval (0.0,1.0) and then it linearly increases over the interval (1.0,2.0). Next, we define the material properties and models that are used in this problem. In this example, we define two sets of material properties: one is called stiff (pencil, tablet, and block 4), and the second is called soft (eraser). We use a linear elastic constitutive model for both cases.

```
begin material stiff
  density = 1.0
  begin parameters for model elastic
    youngs modulus = 1.e5
    poissons ratio = 0.3
  end parameters for model elastic
end material stiff

begin material soft
  density = 1.0
  begin parameters for model elastic
    youngs modulus = 1000.
    poissons ratio = 0.3
  end parameters for model elastic
end material soft
```

Now, we define the finite element mesh. This includes specification of the file that contains the mesh, a list of all the element blocks we will use from the mesh and the material associated with each block. The name of the file is `eraser.g`. The specification of the database type is

optional-ExodusII is the default. Currently, each element block must be defined individually. The tablet, pencil and block 4 all reference the same material description (stiff). The material description is not repeated three times. The material description for stiff appears once and is then referenced three times.

```
begin finite element model mesh1
  database name = eraser.g
  database type = exodusII
  begin parameters for block block_1 #Pencil
    material = stiff
    model = elastic
  end
  begin parameters for block block_2 #Eraser
    material = soft
    model = elastic
  end
  begin parameters for block block_3 #Tablet
    material = stiff
    model = elastic
  end
  begin parameters for block block_4 #dummy block
    material = stiff
    model = elastic
  end
end finite element model mesh1
```

At this point we have finished specifying physics-independent quantities. We now want to set up the Adagio procedure and region, along with the time control command block. We start by defining the beginning of the procedure scope, the time control command block, and the beginning of the region scope. Three time stepping command blocks are used in this analysis. The termination time is set to 1.8. Having multiple time blocks is useful because we can make some solver options a function of the time block.

```
begin adagio procedure agio_procedure
  begin time control
    begin time stepping block preload
      start time = 0.0
      begin parameters for adagio region adagio
        number of time steps = 5
      end parameters for adagio region adagio
    end time stepping block preload
    begin time stepping block slide_1
      start time = 1.0
      begin parameters for adagio region adagio
        number of time steps = 1
      end parameters for adagio region adagio
```

```
  end time stepping block slide_1
  begin time stepping block slide_2
    start time = 1.1
    begin parameters for adagio region adagio
      number of time steps = 10
    end parameters for adagio region adagio
  end time stepping block slide_2
  termination time = 1.8
end time control

begin adagio region adagio
```

Next we associate the finite element model we defined above (`mesh1`) with this Adagio region.

```
  use finite element model mesh1
```

Now we define the kinematic boundary conditions. First, we prescribe the displacement of surface_200 which is part of block 4. We fix this surface in both the x and z directions and prescribe the horizontal displacement along the X-direction using the previously defined functions `zero` and `slide`.

```
  ### movement of pencil prescribed by block 4
  begin prescribed displacement
    surface = surface_200
    direction = X
    function = slide
    scale factor = 3.0
  end prescribed displacement
  begin fixed displacement
    surface = surface_200
    components = Y Z
  end fixed displacement
```

Next, we fix the tablet and prevent it from moving in the X-Y plane. We do this on all tablet nodes (nodelist_111 and nodelist_112).

```
  ### Constraints on tablet
  begin fixed displacement
    node set = nodelist_111
    components = X Y
  end fixed displacement
  begin fixed displacement
    node set = nodelist_112
    components = X Y
  end fixed displacement
```

Finally, we prescribe preload force on the tablet which compresses the tablet against the eraser.

```
### Tablet force
begin prescribed force
  node set = nodelist_111
  direction = Z
  function = tablet_force
  scale factor = 100.0
end prescribed force
```

We now define the contact for this problem. Here we need to define tied contact between block 4 and the eraser. In addition, we have frictional sliding contact between the tablet and the eraser.

```
### block 4 tied to eraser ###
begin contact definition
  #
  # define contact surfaces.
  contact surface surf_200 contains surface_200
  contact surface surf_110 contains surface_110
  contact surface surf_11  contains surface_11 # Tablet
  contact surface surf_10  contains surface_10 # Eraser
  #
  begin interaction
    surfaces = surf_200 surf_110
    friction model = tied
  end interaction

  begin interaction
    surfaces = surf_11 surf_10
    friction model = fric
  end interaction

  begin constant friction model fric
    friction coefficient = 0.8
  end

end contact definition
```

Now we define what variables we want in the output file and how often we want the output file to be written. The output file will be called eraser.e, and it will be an ExodusII file (the database type command is optional; it defaults to ExodusII). The variables we are requesting are the displacements, velocities, and contact diagnostics at the nodes, and stresses and strains at the elements.

```
begin results output output_adagio
  database name = eraser.e
```

```
  database type = exodusII
  at step 0, increment = 1
  nodal displacement as displ
  nodal velocity as vel
  nodal contact_tangential_direction as cont_tan
  nodal contact_normal_direction as cont_norm
  nodal contact_accumulated_slip_vector as cont_slip_vec
  nodal contact_status as cont_stat
  nodal contact_normal_traction_magnitude as norm_trac_mag
  nodal contact_tangential_traction_magnitude as tan_trac_mag
  nodal contact_incremental_slip_magnitude as cont_Islip_mag
  nodal contact_accumulated_slip as cont_Aslip
  nodal contact_frictional_energy_density as cont_fric_E
  nodal contact_area
  element stress as stress
  element log_strain as strain
end results output output_adagio
```

The final part of the input deck is the Adagio solver commands. Because we have sliding contact, we must use the solver command block. Nested inside the solver block, we define the control contact block as well as the nonlinear conjugate gradient (CG) solver block.

```
begin solver

  begin loadstep predictor
    type = secant
    slip scale factor = 0.0
  end loadstep predictor

  level 1 predictor = none

  begin control contact name
    target relative residual = 1e-3
    Maximum Iterations       = 200
    maximum slip iterations  = 1
    maximum line search step = 1.00
    minimum line search step = 0.50
    slip nodes               = all_nodal
  end control contact name

  begin cg
    target relative residual     = 1.e-4
    minimum residual improvement = 0.5
    begin full tangent preconditioner
      linear solver = feti
    end full tangent preconditioner
  end cg
```

```
    end solver
```

After defining the Adagio region and procedure blocks, we close them using the lines:

```
      end adagio region adagio
    end adagio procedure agio_procedure
```

The final detail that we need to define for this file is the linear solver. In the above solver command block, we included the `full tangent preconditioner` block, which has a nested command line `linear solver = feti`. This command line refers to a linear solver called `feti` that must be defined outside the "Procedure" scope but within the "Sierra" scope and can come at the top of the file before the "Procedure" definition or after it as is the case here. The input `feti` on this line command is a user defined string that can have any useful label. The following command block defines the linear solver labeled `feti`. Note that the command block has the label `feti` at the end of the `Begin` line and that this label is referred to from within the above `full tangent preconditioner` command block. The FETI parameters are all set to reasonable values for most problems, so there is no need to set any of them, but if it were necessary to set any of them to non-default values, that would be done within the `feti equation solver` block.

```
    begin feti equation solver feti
      # use default settings
    end feti equation solver feti
```

Finally, we finish the input file and close the sierra command block:

```
  end sierra eraser
```

APPENDIX C. Consistent Units

This appendix describes common consistent sets of units. Maintaining a consistent set of units is crucial in using Sierra/SMwhen entering material properties and interpreting results. The only variables that have intrinsic units are rotations, which are in radians. All other variables depend on the consistent set of units that the user uses in inputting the material properties and dimensioning the geometry.

A consistent set of units is made by picking the base units. When using the International System (SI), these are length, mass, and time, while in the English unit systems, these base units are length, force, and time. All other units are then derived from these base units. Table C-1 provides several examples of commonly used consistent sets of units. In general, the names of the unit systems in this table are taken from the names of the base units. For example, CGS stands for (centimeters, grams, seconds) and IPS stands for (inches, pounds, seconds).

Table C-1. Consistent Unit Sets

Unit	Unit System				
	SI	CGS	IPS	FPS	MMTS
Mass	kg	g	$\frac{lb \cdot s^2}{in}$	$slug$	$tonne$
Length	m	cm	in	ft	mm
Time	s	s	s	s	s
Density	$\frac{kg}{m^3}$	$\frac{g}{cm^3}$	$\frac{lb \cdot s^2}{in^4}$	$\frac{slug}{ft^3}$	$\frac{tonne}{mm^3}$
Force	N	dyn	lb	lb	N
Pressure	Pa	$\frac{dyn}{cm^2}$	psi	psf	MPa
Moment	$N \cdot m$	$dyn \cdot cm$	$in \cdot lb$	$ft \cdot lb$	$N \cdot mm$
Temperature	K	K	$^\circ R$	$^\circ R$	K
Energy	J	erg	$lb \cdot in$	$lb \cdot ft$	mJ
Velocity	$\frac{m}{s}$	$\frac{cm}{s}$	$\frac{in}{s}$	$\frac{ft}{s}$	$\frac{mm}{s}$
Acceleration	$\frac{m}{s^2}$	$\frac{cm}{s^2}$	$\frac{in}{s^2}$	$\frac{ft}{s^2}$	$\frac{mm}{s^2}$

One of the most common mistakes related to consistent units comes in when entering density. For example, in the IPS system, a common error is to enter the density of stainless steel as 0.289

lb/in^3, when it should be entered as 7.48e-4 $lb \cdot s^2/in$. The weight per unit volume should be divided by the gravitational constant (386.4 in/s^2 in this case) to obtain a mass per unit volume.

APPENDIX D. Constraint Enforcement Hierarchy

When a node has multiple constraints, they are enforced in a specific order. Table D-1 shows the order of enforcement of the various types of constraints.

Table D-1. Constraint Enforcement Order

1	Contact
2	MPC
3	Kinematic
4	Rigid Body

If any of the constraints are in conflict, the last constraint enforced will override previously enforced constraints. For example, if a kinematic boundary condition and a MPC are both active on a node and conflict with each other, the MPC will be enforced first, followed by the kinematic boundary condition. As a result, the kinematic boundary condition will be enforced and will override the MPC.

Additionally, a node may have multiple instances of one type of constraint. For example a node could have multiple conflicting kinematic constraints. For such cases generally the last constraint defined in the input deck will be enforced overriding any conflicting earlier constraints. Only the part of previous constraints orthogonal to the last constraint will be enforced.

Example 1:

```
BEGIN INITIAL VELOCITY
  NODE SET = nodelist_1
  COMPONENT = X
  MAGNITUDE = 0.1
END
BEGIN INITIAL VELOCITY
  NODE SET = nodelist_1
  COMPONENT = Y
  MAGNITUDE = 0.2
END
```

The two velocity boundary conditions are orthogonal thus combine normally, each node in nodelist_1 will be given a velocity of (0.1, 0.2. 0.0).

Example 2:

```
DEFINE DIRECTION angle WITH VECTOR 1 1 0
BEGIN INITIAL VELOCITY
  NODE SET = nodelist_1
  DIRECTION = angle
  MAGNITUDE = 1.0
END
BEGIN INITIAL VELOCITY
  NODE SET = nodelist_1
  COMPONENT = X
  MAGNITUDE = 0.5
END
```

The two velocity boundary conditions are not orthogonal. In evaluating the first boundary condition the nodes will be given a velocity of (0.707107, 0.707107, 0.0). Next the second boundary condition is evaluating overriding the non-orthogonal portion first resulting in a final nodal velocity of (0.5, 0.707107, 0.0).

APPENDIX E. Improving Explicit Dynamics Performance

In general, the default settings in Sierra/SM explicit dynamics are chosen for robustness and accuracy over speed. When appropriate, defaults are changed in new releases as new options are shown to be sufficiently robust and accurate for a wide range of problems. In this appendix, we describe several approaches that can help to improve the execution speed, though potentially at the expense of robustness or accuracy.

Increase the critical time step

The critical time step is generally controlled by three factors: element size, material stiffness, and material density. Smaller elements drive the critical time step down, as do stiffer materials. Denser materials have the opposite effect of increasing the critical time step.

Improve the mesh quality The critical time step for explicit dynamics is often set by the smallest element. To identify the particular element which is limiting the simulation time step, the critical time step of each element can be output for plotting using the element variable `timestep`, see Table 9-12.

Mass scaling This is a fairly common approach used by many commercial finite element codes to increase the time step. A range of options are provided in Sierra/SM to apply mass scaling dynamically throughout the analysis runtime, see Section 3.3.1. However, mass scaling can affect the solution in critical areas and must be used with caution.

Lanczos time step This is an alternative method for computing the critical time step. The default element-based time step is sometimes more than 30% smaller than the true critical time step, especially if element sizes and material properties vary significantly and suddenly throughout the domain. The Lanczos algorithm computes a scaling factor which can bring the time step closer to the true critical value; this can result in a significant reduction in analysis time. However, for some analyses the Lanczos method does not provide a significantly improved time step, and computing the Lanczos time step estimate is expensive and thus should be done only occasionally during the analysis. And in a few problems, this approach can lead to unstable solutions. See Section 3.2.1.5 for further details.

Node-based time step This is another approach to improve the estimate of the critical time step and often leads to larger time steps than the default element-based time step. Compared to the Lanczos estimate, the node-based method may still underestimate the critical time step but requires less overhead to compute. See Section 3.2.3.1 for further details.

Improve contact performance

Appropriate contact tolerances It is almost always best (for both performance and robustness) to use the default contact search tolerances. Larger search tolerances will only slow down the simulation.

Mesh rebalancing Use mesh rebalancing to avoid significant loss of performance for problems with large deformations and evolving contact surface interactions see Section 6.8.

Mesh decomposition The initial decomposition of the mesh can have a sizable effect on the parallel performance of a model. For example, a Recursive Coordinate Bisection (RCB) decomposition can sometimes lead to improved load balancing compared to the default decomposition; see also the SEACAS manuals [1, 2, 3].

Self contact off Not using self contact can help decrease the total time needed for tracking and reduce the number of contact interactions to assess. If possible, specify the needed contact interactions without using self contact. See Chapter 8 for more information.

Improve computational efficiency

Reduce the number of side sets, node sets and blocks A large number of side sets, node sets and/or blocks adversely affects the memory layout and can lead to both larger memory usage and longer execution times. Try to define boundary conditions using as few side sets/node sets as possible. Also try to remove node sets, sides sets and blocks which are not explicitly used in the input deck.

Optimal number of mesh partitions If the mesh is partitioned such that there are less than around 1000 elements per processor, the parallel performance may become significantly compromised. Try to aim for 2000-10000 elements per processor. This recommendation may change as computer architectures evolve.

Less frequent checks for element death Element death can lead to significant slowdowns if a large number of criteria are applied every time step. To reduce this effect, assess element death only occasionally, see Section 6.5.3. Also, if possible try to limit the areas that have both element death and contact.

Use the uniform-gradient hex8 element Whenever performance improvements are made to the elements in Sierra/SM they are generally first applied to the uniform-gradient hex8 element. Other element types may also be upgraded or addressed only as needs arise.

Use an alternative material model Some material models have large numbers of function look ups (for instance the orthotropic crush model). These models typically have poor performance. Elastic models are typically significantly faster but may not be appropriate for many analyses.

The Sierra/SM team is interested in assisting with any performance issues. Please contact them if you have questions, example performance timings, or would like help in getting better performance out of a Sierra/SM explicit dynamics simulation.

References

[1] T.A. Laursen, S.W. Attaway, and R.I. Zadoks. SEACAS theory manuals: Part I. problem formulation in nonlinear solid mechanics. Technical Report SAND98-1760/1, Sandia National Laboratories, Albuquerque, NM, 1998. pdf.

[2] T.A. Laursen, S.W. Attaway, and R.I. Zadoks. SEACAS theory manuals: Part II. nonlinear continuum mechanics. Technical Report SAND98-1760/2, Sandia National Laboratories, Albuquerque, NM, 1998. pdf.

[3] T.A. Laursen, S.W. Attaway, and R.I. Zadoks. SEACAS theory manuals: Part III. finite element analysis in nonlinear solid mechanics. Technical Report SAND98-1760/3, Sandia National Laboratories, Albuquerque, NM, 1999. pdf.

APPENDIX F. Troubleshooting Guide for Implicit Convergence

Implicit simulations often encounter convergence problems that are difficult to diagnose and fix. Below is a list of commonly used techniques for obtaining convergence in implicit problems.

When using a multi-level solver such as control contact, two distinct convergence issues are often encountered: 1) the model problem (i.e., the level 0 solver) fails to converge, and 2) control contact (i.e., the level 1 solver) fails to converge. The following suggestions may help to obtain convergence in either case:

- **Add additional constraints to remove rigid body modes.** If the problem is depending on contact to remove rigid body modes, it can be difficult for the solver to converge due to poor conditioning of the system. If there is a way to remove the rigid body modes without relying on contact then these problems are better posed and easier for the solver to handle.
- **Add tangent diagonal scaling.** If it is not possible to add additional constraints to remove rigid body modes then tangent diagonal scale can avoid some of the ill-conditioning of the problem. Because this option only modifies the preconditioner, it will not affect the final converged solution significantly. However, if too much tangent diagonal scaling is used then the solver convergence rate may slow down. See Section 4.2.2 for further discussion.
- **Set a reasonably small number of maximum iterations and a high acceptable residual for the model problem.** This is useful because the contact constraints for the first model problem solve may be over-constrained or otherwise invalid. These inconsistencies are resolved after the first contact iteration, and setting the maximum iterations and a high acceptable residual allows the simulation to get past this hurdle. Adjusting these parameters will not affect the final converged solution since the overall solution will still be checked against the convergence tolerances of the level 1 control contact solve.
- **Try the full tangent preconditioner.** The full tangent preconditioner often makes problems converge in significantly fewer iterations and is typically more efficient than other available preconditioners.
- **Take smaller step sizes.** This can help reduce the amount of nonlinearity introduced in the load step, making the problem easier to solve.
- **Turn off load step predictor.** This can help because the interactions are defined in the initial predicted configuration. Predicting the initial configuration can be very difficult and no prediction is always better than a bad prediction. See Section 4.9 for details.
- **Turn off adaptive penalty.** The adaptive penalty algorithm attempts to improve the convergence rate by adjusting the penalty parameters but is sometimes unsuccessful. Without adaptive penalty convergence is typically slower but more robust.
- **Turn off adaptive time stepping.** Adaptive time stepping can mask larger issues by pushing past the real issue and later spawning a fictitious issue.
- **If using frictionless contact, try a frictional model.** Frictional models have tangential stiffness that prevents sliding modes, making the model problem easier to solve.

- **Solve the model problem to a tighter target tolerance.** The penalty kernel for augmented Lagrange comes from the model problem solve. So, if the model problem is not solved to a sufficiently small tolerance then there is little hope that the Lagrange multiplier will converge with an acceptable residual tolerance.
- **Try a different preconditioner.** The full-tangent preconditioner is sensitive to rigid body modes but can handle body forces better than the nodal preconditioner. Alternatively, the nodal preconditioner is insensitive to rigid body modes but is very slow to converge body forces.
- **Use iteration plot to diagnose problems.** This is useful to see if a rigid body mode is present. It can also be helpful to determine where contact is iterating excessively to diagnose problems which would be difficult to address without a visual aid. See Section 4.1.1 for further discussion.
- **Switch contact constraint formulation.** The node-face contact constraint formulation is better able to handle conflicting constraints but may miss contact in corners or introduce force concentrations that make the problem difficult to solve. Alternatively, face-face constraints give smoother forces but can have more difficulty with constraint conflicts.
- **Avoid constraint conflicts with contact.** Kinematic boundary conditions, MPCs and prescribed rigid bodies can all conflict with contact constraint definitions (see Appendix D). When this occurs convergence can be very poor because the penalty algorithm does not see the other constraints and assumes it can freely move the constrained nodes/faces.
- **Use default normal and tangential search tolerance.** Bad search tolerances can cause the solver to miss constraints or attempt to enforce bad constraints. The default values are generally robust.
- **Use block skinning instead of side-set definitions.** Block skinning gives contact search more information to work with when making projections to define constraints. This extra information can help the search cull incorrect contact constraints.
- **Adjust the AL penalty.** If contact constraints appear to be under-enforced, increasing the AL penalty parameter may help. In particular, for cases in which a very stiff material contacts a very soft material, increasing the AL penalty parameter can sometimes improve convergence. Conversely, a high AL penalty parameter may be over enforcing the constraints, resulting in an ill-conditioned system. If this is the case, decreasing the AL penalty may help, see Section 8.6.1.5.
- **Set a Lagrange tolerance.** This is useful if the maximum gap is acceptably small and the residual in the model problem is acceptably small, but the residual imbalance introduced by a small Lagrange multiplier change prevents control contact convergence, see Section 4.5.1.6.
- **Use Control Damped Solve.** This is useful in problems that have rigid body motion before contact is seated, problems where contact causes rebound or bouncing to occur, problems that involve sliding contact, and other difficult implicit problems. Control Damped Solve adds artificial springs to the nodes where the solve difficulty is occurring, and the spring force is slowly decreased near zero so the springs do not introduce error into the problem. See Section 4.8 for more details.

APPENDIX G. User-Defined Model Initialization

Multiple methods exist to define the initial state of an analysis model. When initializing the state of a new Sierra/SM analysis from a previous analysis, restart is the recommended method (see Section 9.6). However, in some cases a restart initialization is not possible; for instance, if the initial stress/strain state of the model is coming from an external source such as another code, experimental data, or an analytic solution. Another example is when new blocks are added to an existing analysis and need initial stress/strain/material state that is defined in a secondary analysis.

The methodology of manually initializing a model is sometimes called "poor man's restart." Generally many node and element fields need to be set to give a fully consistent initial state. The required field set is different for different element types as shown in Table G-1.

Material state needs to be initialized in a manner that is consistent with other element kinematic quantities, as well. Several elastic materials have no state data, but most materials have a state data array at each element. The name of the material state data field is usually defined by `lame_state_[model_name]`, where `model_name` is the name of the material model. For example the name of the state field for the viscoplastic material is `lame_state_viscoplastic` and the name of the state field for the soil foam model is `lame_state_soil_foam`.

Material state variables also often have aliases, which causes some confusion when using them for restart and initialization purposes. When writing out material state in a preload analysis for initialization in a later analysis output, the following syntax is strongly recommended:

```
BEGIN RESULTS OUTPUT
  ...
  ELEMENT VARIABLES = lame_state_viscoplastic(:) as \#
    state_viscoplastic
  ...
END
```

The '(:)' operator will prevent the use of material state alias names; the material state will be output as a single contiguous field which can then be read back in a single operation. An initialization block like the following should then be used to read back in the material state:

```
BEGIN INITIAL CONDITION
  INITIALIZE VARIABLE NAME = lame_state_viscoplastic(:)
  VARIABLE TYPE = ELEMENT
  READ VARIABLE = state_viscoplastic
END
```

In addition to element- and material-specific field data, some boundary-specific conditions may have additional initialization requirements.

Table G-1. Required Variables For Consistent Model Initialization

Element	Nodes	Formulation	Element Variables	Nodal Variables
Beam	2	Rod	beam_tshear_rs, beam_tshear_rt, beam_t_axis, cross_sectional_area, left_stretch, unrotated_stress	acceleration, displacement, velocity, rotational_acceleration, rotational_displacement, rotational_velocity
		Cylindrical Joint	cylj_rst_theta1, cylj_rst_theta2, hg_resistance, left_stretch, unrotated_stress	displacement, velocity
Shell	3	Orig_Tri_Shell, C0_Tri_Shell	element_thickness, unrotated_stress	acceleration, displacement, velocity, rotational_acceleration, rotational_displacement, rotational_velocity
	4	BL_Shell (Integration rule = Trapezoid)	membranehourglass, shearhourglass, unrotated_stress	acceleration, displacement, velocity, rotational_acceleration, rotational_displacement, rotational_velocity
		BT_Shell (Integration rule = Trapezoid)	hg_resistance_q4, unrotated_stress	acceleration, displacement, velocity, rotational_acceleration, rotational_displacement, rotational_velocity
Hex	8	Uniform Gradient	rotation, left_stretch, hg_resistance, unrotated_stress	displacement, acceleration, velocity
		Fully Integrated	unrotated_stress	displacement, velocity
Tet	4	Uniform Gradient	left_stretch, unrotated_stress	displacement, acceleration, velocity
		Fully Integrated	left_stretch, unrotated_stress	displacement, acceleration, velocity
	10	Composite_Tet	left_stretch, rotation, unrotated_stress	displacement, velocity
Particle	1	SPH	calc_dens, element_density, element_mass, left_stretch, rotation, total_stress, unrotated_stress	coordinates, acceleration, displacement, velocity, rotational_acceleration, rotational_displacement, rotational_velocity, force_internal

APPENDIX H. Other Scripts Maintained by Development Team

In addition to the large Sierra executables the Sierra/SM development team also maintains several pre- and post-processing scripts that can be helpful for analyst use. These scripts are installed with the Sierra tools in the directory `contrib/testTools/adagio/`. Many of the scripts in this directory are used for regression test utilities and not intended for general use. The scripts that are tested for general use are as follows:

`cth_tracers.py` a utility for mapping pressures from a CTH analysis onto a solid mechanics analysis.

`material_fitting.py` a utility to aid fitting of elastic plastic material constants from test data.

`rotateModel.py` a utility to rotate an exodus results file along with all of its data output fields.

`spheregen.py` a utility to convert a solid element model into spheres for use with particle element formulations.

`wedgeTo3d.py` a utility to translate the mesh and results from an axisymmetric wedge model into a full 3D model.

APPENDIX I. Pre-defined User Options

Many of the material models and boundary conditions use functions, directions, or points to define variables in the code. To reduce the size of the input file, some of the common user options have been defined.

Table I-1. Pre-defined functions, directions, and points that can be used in a SIERRA SM analysis

Option	Name	Description
Function	SIERRA_CONSTANT_FUNCTION_ZERO	Output zero at all times
	SIERRA_CONSTANT_FUNCTION_ONE	Output one at all times
	SIERRA_LINEAR_RAMP_FUNCTION	Output simulation time
	SIERRA_COS_RAMP_FUNCTION	Output cosine ramp starting at beginning of simulation and ending at termination time
Direction	SIERRA_DIRECTION_X	Defined direction in (1,0,0)
	SIERRA_DIRECTION_Y	Defined direction in (0,1,0)
	SIERRA_DIRECTION_Z	Defined direction in (0,0,1)
	SIERRA_DIRECTION_NEG_X	Defined direction in (-1,0,0)
	SIERRA_DIRECTION_NEG_Y	Defined direction in (0,-1,0)
	SIERRA_DIRECTION_NEG_Z	Defined direction in (0,0,-1)
Point	SIERRA_POINT_ORIGIN	Defined point at (0,0,0)

APPENDIX J. Known Issues

The following is a list of known issues in Sierra/SM as of the most recent code release.

Section 4.3: Deactivation of element blocks (see Section 6.1.7.9) does not currently work in conjunction with the full tangent preconditioner in Adagio. To use this capability, one of the nodal preconditioners must be used.

Section 6.2.1: For problems with large rotations, the hourglass energies are known to spike exponentially with increases in total rotation. This behavior is observed for both midpoint and strongly objective strain incrementation and for both incremental and total hourglass formulations. The large rotation issue is a known limitation in solid mechanics for all non-hyperelastic material models.

Section 6.2.13: For problems with rotations, the SPH algorithm generates rotational spring like forces on the boundary because of known deficiencies in the algorithm in distinguishing between rigid body rotation and deformation. The SPH algorithm does not conserve rotational or angular momentum.

Section 6.2.14: Superelements are not compatible with several modeling capabilities. They cannot be used with element death. They cannot be used with node-based, power method, or Lanczos critical time step estimation methods. They are also not compatible with some preconditioners (such as FETI) for implicit solutions.

Section 7.4.2.2: If a prescribed displacement with the `CYLINDRICAL AXIS` option is applied to nodes that fall on the axis, it will have no effect. Separate boundary conditions should be applied to those nodes to fix them in the plane normal to the axis.

Section 7.4.3.2: If a prescribed velocity with the `CYLINDRICAL AXIS` option is applied to nodes that fall on the axis, it will have no effect. Separate boundary conditions should be applied to those nodes to fix them in the plane normal to the axis.

Section 8.9.6: Attempting to use `GLOBAL SEARCH INCREMENT` with a value greater than 1, especially in a problem that contains shells and/or restart, will, in most cases, cause code failure. A `GLOBAL SEARCH INCREMENT` value greater than 1 will, under the best circumstances, give only a marginal improvement in speed.

Section 9.5: User defined variables (see Section 10.2.4) are not currently supported with heartbeat output.

Section 8: Contact of one rigid body against another is currently inexact. This includes contact of actual meshed rigid bodies, analytic surfaces, or incompressible lofted objects (particles, contact through the thickness of shells, or contact through the radius of beams.) As these contact systems have no elastic response, the contact solver may need many iterations to solve these systems or may fail due to an unsolvable system. This could lead to energy conservation issues in the result

of contact between these objects.

Section 4.3: Although multiple linear-solver packages are available for use as preconditioners for solving implicit solid mechanics problems, we strongly recommend that the FETI equation solver or plain CG be used in production analyses. The FETI equation solver is actively maintained and tested by the code development team. The `FETI EQUATION SOLVER` command block is documented in Section 4.4. Other solvers may not be thoroughly tested and may have compatibility problems with some code capabilities such as MPCs and contact.

Many of the more advanced constraint types are not fully compatible with one another. Thus if a single node is involved in multiple types of constraints the resultant behavior may be unstable, particularly if the constraints are not fully orthogonal. Currently supported types of constraints, involve kinematic (such as prescribed velocity), rigid body, periodic, MPC, contact, and spot weld. Kinematic constraints are compatible with contact. MPC constraints are somewhat compatible with contact. However, most other combinations have limited compatibility. Thus for example, a node with both a contact constraint and a periodic constraint may lead to model instability.

J.1. Epic Material Model Issues

Epic (MMM) materials will have problems when used with several of the following capabilities.

- Converting MMM elements to SPH particles
- Given a set of MMM SPH particles adding (via particle conversion) or subtracting (via element death) from that set. Changing the contents of the particle set will cause a redistribution of volume among the integration points, not just addition or subtraction of volume at the changed particle.
- Nodal based tetrahedra, and non-local continuum elements have the same non-local issue as SPH particles.
- Mapping of a solution from one mesh to a different mesh will cause a similar problem.

The problem with the above is that many of the epic material models explicitly store the integration point volume as a material state value; the change of this volume is used to compute an incremental strain rate for equation of state calculations. Any method that yields changes in material point volume unassociated with material deformation will cause these models to have severe accuracy and stability problems. If, for instance, on conversion to particles, a new particle returns a volume of 90% of the old element, the material interprets this as a 10% volumetric compression over a single nanoseconds-long time step; the material can view this as a monstrously large shock. This can cause the material point to introduce an immense amount of artificial energy to the system and drive the model immediately unstable.

In order to avoid these problems when converting MMM materials to a particles, convert them to a standard material type (such as elastic plastic). Additionally do not use the MMM materials when using any of the non-local or mesh dynamic methods listed above.

J.2. SPH Particle Method Time Step Stability Issues

Many particle problems that are unstable at the default time step can be made stable if they are run at a manually reduced time step. There are a number of related issues: SPH time step calculation is not that robust to begin with, particularly in high volumetric compression. If over a single time step, two particles can move more that about one half particle radius length, which can cause an instability. Effectively, this will cause the SPH material to invert. This can be an issue with high velocity impact problems. The relative particle velocities are not taken into account in the critical time step calculation, so the code has no knowledge when this may be occurring.

If this problem occurs, it may be fixed by dropping the global time step scaling factor (say from 1.0 to 0.75 to 0.5) and examining if stability can be obtained. Generally if stability cannot be obtained at a factor of 0.5, there is something other than time step driving the instability.

J.3. Particle Creation Issues

If two SPH particles happen to be created extremely close to one another (for example, less than 10“%” of a particle radius), the particles will fly apart violently, adding significant artificial energy to the system. The closer the particles are, the large the energy will be. If two particles are created exactly on top of one another, the error is infinite. This excessively close particle creation can happen with particle conversion or embedded particles. If the source elements for the particles are heavily deformed and in close proximity, new particles can be created close by to existing particles. Additionally contact and/or particle repulsion may be imperfect in high velocity impact cases, allowing bodies to temporarily occupy the same space. If a newly converted particle happens to be placed on top of an existing particle, a major influx of energy may ensue, causing instability.

Possible input deck fixes include

- reducing the time step,
- reducing the potential for contact or particle repulsion errors,
- adding a “failsafe” death criterion to detect and remove particles that have gone unstable (for example, when these types of errors occur, the problem elements often have astronomically high velocities. Establishing a death criterion of velocity that exceeds, for instance, 10 times the initial impact velocity may detect and immediately remove the problem elements, preventing the stability error from propagating to the rest of the analysis.), or
- creating more robustly enforced contact (the use of more contact momentum balance iterations may help prevent materials—and thus newly created particles—from occupying the same space).

J.4. Bad Element Shapes Yielding Invalid Contact Geometry

Elements will continue to return some material response as long as they maintain positive volume. However, long before the point of absolute element inversion, it is possible for the contact geometry of elements to become corrupted. For example concave and locally inverted elements may have exterior faces that point the wrong way. An incorrectly pointed face can cause major contact errors to occur either adding artificial energy to the system or preventing solver convergence. These types of extreme deformation can be common in high velocity impacts. Use of a `nodal_jacobian_ratio <= 0.0` death criterion can help avoid this issue. For eight-noded hexahedral elements, the contact geometry for an element is guaranteed to be valid at least until this shape metric reduces to zero.

Index

2G SCALING
in Incompressible Solid material model, 343

A
in Barlat Plasticity material model, 274
in Hosford Plasticity material model, 254
in Low Density Foam material model, 319
A0
in Soil and Crushable Foam material model, 312
A1
in Soil and Crushable Foam material model, 312
in Swanson material model, 357
in Viscoelastic Swanson material model, 360
A2
in Soil and Crushable Foam material model, 312
ABSCISSA
in function, 59
ABSOLUTE SEARCH TOLERANCE
in Procedural Transfer, 917, 919
ACCEPTABLE PRESSURE INCREMENT
in Control Stiffness, 182
description of, 183
ACCEPTABLE RELATIVE CONTACT RESIDUAL
in Control Contact, 171
description of, 173
ACCEPTABLE RELATIVE PRESSURE INCREMENT
in Control Stiffness, 182
description of, 183
ACCEPTABLE RELATIVE RESIDUAL
in CG, 141
description of, 143
in Control Contact, 171
description of, 173
in Control Stiffness, 182
description of, 183
ACCEPTABLE RELATIVE SDEV INCREMENT
in Control Stiffness, 182
description of, 183
ACCEPTABLE RELATIVE STRAIN INCREMENT
in Control Stiffness, 182
description of, 183
ACCEPTABLE RELATIVE STRESS INCREMENT
in Control Stiffness, 182
description of, 183
ACCEPTABLE RESIDUAL
in CG, 141
description of, 143
in Control Contact, 171
description of, 173
in Control Stiffness, 182
description of, 183
ACCEPTABLE SDEV INCREMENT
in Control Stiffness, 182
description of, 183
ACCEPTABLE STRESS INCREMENT
in Control Stiffness, 182
description of, 183
ACTIVE FOR PROCEDURE
in Finite Element Model – in Parameters For Block, 389
description of, 397
ACTIVE PERIODS, 94
description of, 94
in Adaptive Time Stepping
description of, 203
in Cavity Expansion, 607
in Centripetal Force, 588
in Contact Definition
use of, 687
in Element Death, 477
usage in, 491
in Fixed Displacement, 519
usage in, 520
in Fixed Rotation, 536
usage in, 538
in Fluid Pressure, 600
usage in, 602
in Gravity, 586
in Implicit Dynamics, 205
description of, 205
in Inertia Relief, 604
usage in, 605
in Line Weld, 617
in Mass Scaling, 129
usage in, 131
in Periodic, 534
in Pore Pressure, 596
usage in, 599
in Prescribed Acceleration, 530
usage in, 534
in Prescribed Displacement, 521

usage in, 525
in Prescribed Force, 572
usage in, 576
in Prescribed Moment, 577
usage in, 580
in Prescribed Rotation, 538
usage in, 543
in Prescribed Rotational Velocity, 543
usage in, 547
in Prescribed Temperature, 591
usage in, 595
in Prescribed Velocity, 526
usage in, 530
in Pressure, 557
usage in, 563
in REFERENCE AXIS ROTATION
usage in, 551
in Silent Boundary, 609
in Spot Weld, 610
in Time Step Initialization, 882
usage in, 885
in Traction, 566
usage in, 570
in User Output, 757
description of, 779
in Viscous Damping, 622
usage in, 625
ADAGIO PROCEDURE, 85
ADAGIO REGION
in Adagio Procedure, 85
description of, 87
ADAPTIVE STRATEGY
in Full Tangent Preconditioner, 155
description of, 163
ADAPTIVE TIME STEPPING
in Adagio Region, 199
ADDITIONAL MASKING FIELD
in Pressure, 557
description of, 562
ADDITIONAL STEPS
in Heartbeat Output, 786
in History Output, 780
in Output Scheduler, 810
in Restart Data, 797
in Results Output, 742
description of, 754
ADDITIONAL TIMES
in Heartbeat Output, 786
in History Output, 780
in Output Scheduler, 810
in Restart Data, 797
in Results Output, 742
description of, 754
AGGRESSIVE CONTACT CLEANUP
in Element Death, 477
in Element Death
description of, 487
AL PENALTY
in Contact Definition – in Interaction, 676
in Contact Definition – in Interaction Defaults, 673
description of, 676
ALGORITHM DEBUG LEVEL
in Zoltan Parameters, 498
as default, 498
ALIAS
in Finite Element Model, 383
description of, 387
ALLOWABLE MASS INCREASE RATIO
in Mass Scaling, 129
description of, 131
ALPHA
in Elastic 3D Orthotropic material model, 323
in Hill Plasticity material model, 263
in Hyperfoam material model, 321
in Implicit Dynamics, 205
description of, 206
ANGULAR VELOCITY
in Initial Velocity, 553
description of, 555
in Rigid Body command block, 469
AREA
in Damper Section, 441
in Truss Section, 438
AT DISCONTINUITY EVALUATE TO
in function, 59
AT STEP
in Heartbeat Output, 786
in History Output, 780
in Output Scheduler, 810
in Restart Data, 797
in Results Output, 742
description of, 754
AT TIME
in Heartbeat Output, 786
in History Output, 780
in Output Scheduler, 810
in Restart Data, 797
in Results Output, 742
description of, 754
aupst_check_elem_var, 860
aupst_check_global_var, 870
aupst_check_node_var, 860
aupst_cyl_transform, 909
aupst_evaluate_function, 859
aupst_get_elem_centroid, 877
aupst_get_elem_nodes, 872
aupst_get_elem_topology, 872

aupst_get_elem_var, 860
aupst_get_elem_var_offset, 860
aupst_get_face_nodes, 872
aupst_get_face_topology, 872
aupst_get_global_var, 870
aupst_get_integer_param, 854
aupst_get_node_var, 860
aupst_get_point, 877
aupst_get_proc_num, 877
aupst_get_real_param, 854
aupst_get_string_param, 854
aupst_get_time, 859
aupst_put_elem_var, 860
aupst_put_elem_var_offset, 860
aupst_put_global_var, 870
aupst_put_node_var, 860
aupst_rec_transform, 911
AXIS OFFSET
in Beam Section, 428
AXIS OFFSET GLOBAL
in Beam Section, 428
AXIS OFFSET VARIABLE
in Beam Section, 428
AXIS STRETCHING
in Coordinate System, 77

B
in Low Density Foam material model, 319
B1
in Swanson material model, 357
in Viscoelastic Swanson material model, 360
BACK STRESS DYNAMIC RECOVERY CONSTANT
in BCJ_MEM material model, 292
BACK STRESS DYNAMIC RECOVERY TEMPERATURE DEPENDENCE
in BCJ_MEM material model, 292
BACK STRESS HARDENING CONSTANT
in BCJ_MEM material model, 292
BACK STRESS HARDENING TEMPERATURE DEPENDENCE
in BCJ_MEM material model, 292
BALANCE PROBE
in CG, 141
description of, 147
in Full Tangent Preconditioner, 155
description of, 157
BEAM SECTION, 428
BEGIN INTERACTION SPECIFIC OUTPUT
in Contact Definition – in Interaction
description of, 681
BENDING HOURGLASS STIFFNESS
in Finite Element Model – in Parameters For Block, 389
description of, 393
BENDING HOURGLASS VISCOSITY
in Finite Element Model – in Parameters For Block, 389
description of, 393
BETA
in Elastic-Plastic material model, 230
in Foam Plasticity material model, 315
in Implicit Dynamics, 205
description of, 206
in Multilinear Elastic-Plastic Hardening Model material model, 240
in Multilinear Elastic-Plastic Hardening Model with Failure material model, 244
in Thermoelastic Plastic material model, 250
BETA METHOD
in CG, 141
description of, 151
BIOT'S COEFFICIENT
about, 212
BLOCK
description of, 503
in Centripetal Force, 588
in Contact Definition – in Contact Surface (block)
usage in, 650
in Element Death, 477
description of, 479
in Fixed Displacement, 519
usage in, 520
in Fixed Rotation, 536
usage in, 537
in Gravity, 586
usage in, 586
in Inertia Relief, 604
usage in, 604
in Initial Condition, 509
usage in, 510
in Initial Temperature
usage in, 590
in Initial Velocity, 553
usage in, 554
in Line Weld, 617
in Mass Properties, 475
usage in, 475
in Mass Scaling, 129
usage in, 130
in Pore Pressure, 596
usage in, 597
in Prescribed Acceleration, 530
usage in, 532
in Prescribed Displacement, 521
usage in, 522
in Prescribed Force, 572

usage in, 573
in Prescribed Moment, 577
usage in, 578
in Prescribed Rotation, 538
usage in, 540
in Prescribed Rotational Velocity, 543
usage in, 545
in Prescribed Temperature, 591
usage in, 592
in Prescribed Velocity, 526
usage in, 527
in Pressure, 557
in Procedural Transfer, 917
description of, 919
in Reference Axis, 548
in Reference Axis Rotation
usage in, 549
in Time Step Initialization, 882
usage in, 884
in User Output, 757
usage in, 762
in Viscous Damping, 622
usage in, 623
BLOCK BY BLOCK
in Procedural Transfer, 917
description of, 920
BULK FUNCTION
in Mooney-Rivlin material model, 352
BULK GLASSY 0
in Universal Polymer Material Model, 346
BULK MODULUS
in Barlat Plasticity material model, 274
in BCJ material model, 289
in Ductile Fracture material model, 237
in Elastic 3D Orthotropic material model, 323
in Elastic Fracture material model, 228
in Elastic material model, 223
in Elastic-Plastic material model, 230
in Elastic-Plastic Power-Law Hardening material model, 234
in Foam Plasticity material model, 315
in Hill Plasticity material model, 263
in Hosford Plasticity material model, 254
in Hyperfoam material model, 321
in Incompressible Solid material model, 343
in Johnson–Cook material model, 286
in Low Density Foam material model, 319
in MD Viscoplastic material model, 308
in Mooney-Rivlin material model, 352
in Multilinear Elastic-Plastic Hardening Model material model, 240
in Multilinear Elastic-Plastic Hardening Model with Failure material model, 244
in Neo-Hookean material model, 226
in Orthotropic Crush material model, 329
in Orthotropic Rate material model, 333
in Plane Stress Rate Plasticity material model, 336
in Power Law Creep material model, 301
in Soil and Crushable Foam material model, 312
in Stiff Elastic material model, 355
in Swanson material model, 357
in Thermoelastic material model, 225
in Thermoelastic Plastic material model, 250
in Viscoelastic Swanson material model, 360
in Viscoplastic material model, 303
in Wire Mesh material model, 327
BULK RUBBERY 0
in Universal Polymer Material Model, 346

C
in Low Density Foam material model, 319
C01
in Mooney-Rivlin material model, 352
C01 FUNCTION
in Mooney-Rivlin material model, 352
C1
in BCJ material model, 289
in Swanson material model, 357
in Viscoelastic Swanson material model, 360
C10
in BCJ material model, 289
in Mooney-Rivlin material model, 352
C10 FUNCTION
in Mooney-Rivlin material model, 352
C11
in BCJ material model, 289
C12
in BCJ material model, 289
C13
in BCJ material model, 289
C14
in BCJ material model, 289
C15
in BCJ material model, 289
C16
in BCJ material model, 289
C17
in BCJ material model, 289
C18
in BCJ material model, 289
C19
in BCJ material model, 289
C2
in BCJ material model, 289
C20
in BCJ material model, 289
C3

in BCJ material model, 289
C4
in BCJ material model, 289
C5
in BCJ material model, 289
C6
in BCJ material model, 289
C7
in BCJ material model, 289
C8
in BCJ material model, 289
C9
in BCJ material model, 289
CAPTURE TOLERANCE
in Contact Definition – in Interaction, 676
description of, 680
CAVITY EXPANSION, 607
CENTRIPETAL FORCE, 588
CG, 141
in Adagio Region, 141
in Solver, 135
CHECK GEOMETRY
in Zoltan Parameters, 498
as default, 498
CHECK STEP INTERVAL
in Element Death, 477
description of, 485
CHECK TIME INTERVAL
in Element Death, 477
description of, 485
CLOCK C3
in Universal Polymer Material Model, 346
CLOCK C4
in Universal Polymer Material Model, 346
COARSE SOLVER
in FETI Equation Solver, 165
description of, 168
COHESIVE MATERIAL
in Element Death, 477
COHESIVE MODEL
in Element Death, 477
COHESIVE SECTION, 409
in Element Death, 477
COHESIVE ZONE INITIALIZATION METHOD
in Element Death, 477
COHESIVE ZONE MODEL
in Contact Definition
description of, 665
COMMUNICATION RATIO THRESHOLD
in Rebalance, 495
description of, 496
COMPONENT
in Fixed Displacement, 519
description of, 520, 538
in Fixed Rotation, 536
description of, 538
in Inertia Relief, 604
in Initial Velocity, 553
description of, 555
in Periodic, 534
in Prescribed Acceleration, 530
description of, 532
in Prescribed Displacement, 521
description of, 523
in Prescribed Force, 572
description of, 573
in Prescribed Moment, 577
description of, 579
in Prescribed Rotational Velocity, 543
description of, 545
in Prescribed Velocity, 526
description of, 528
COMPONENT SEPARATOR CHARACTER
in Finite Element Model, 383
description of, 388
in Restart Data, 797
description of, 805
in Results Output, 742
description of, 752
COMPONENTS
in Fixed Displacement, 519
description of, 520, 538
in Fixed Rotation, 536
description of, 538
in MPC, 630
in Prescribed Acceleration, 530
in Prescribed Displacement, 521
in Prescribed Velocity, 526
in TIED MPC, 627
COMPUTE AT EVERY STEP, 757
in User Output
description of, 778
COMPUTE AT HEARTBEAT OUTPUT STEPS, 757
COMPUTE AT HISTORY OUTPUT STEPS, 757
COMPUTE AT RESULTS OUTPUT STEPS, 757
COMPUTE ELEMENT
in User Output, 757
COMPUTE GLOBAL
in User Output, 757
COMPUTE INTERACTION CONTACT STATUS
in Contact Definition – in Interaction
description of, 681
COMPUTE INTERACTION FORCE CONTACT
in Contact Definition – in Interaction
description of, 681
COMPUTE INTERACTION MASK
in Contact Definition – in Interaction
description of, 681

COMPUTE INTERACTION RELATIVE VELOCITY
in Contact Definition – in Interaction
description of, 681
COMPUTE NODAL
in User Output, 757
CONDITIONING
in Full Tangent Preconditioner, 155
description of, 157
CONICAL COORDINATE SYSTEM
in Coordinate System, 77
CONNECTION BLOCK
in RBE3, 633
CONNECTION NODE SET
in RBE3, 633
CONNECTION NODES
in RBE3, 633
CONNECTION SURFACE
in RBE3, 633
CONSTANT FRICTION MODEL
in Contact Definition
description of, 661
CONSTANT SPHERE RADIUS
in Particle Section, 444
CONSTRAINT FORMULATION
in Contact Definition – in Interaction, 676
in Contact Definition – in Interaction Defaults, 673
description of, 676
CONTACT LOFTING FACTOR
in Membrane Section, 425
in Shell Section, 414
CONTACT NODE SET
in Contact Definition
description of, 650
CONTACT SURFACE
in Contact Definition
description of, 649
CONTACT SURFACE (block)
in Contact Definition
description of, 650
CONTACT THICKNESS
in Membrane Section, 425
in Shell Section, 414
CONTACT TIMESTEP
in Implicit Dynamics, 205
description of, 206
CONTACT VARIABLES
in Contact Definition
description of, 707
CONTROL CONTACT
in Solver, 135
description of, 171
CONTROL DAMPED SOLVE
in Solver, 135
description of, 189
CONTROL FAILURE
in Solver, 135
description of, 186
CONTROL STIFFNESS
in Solver, 135
description of, 182
COORDINATE SYSTEM, 77
in Elastic 3D Orthotropic material model, 323
in Hill Plasticity material model, 263
COPY ELEMENT VARIABLE
in User Output, 757
description of, 772
COPY VARIABLE
in Initial Condition, 509
in Pore Pressure, 596
description of, 598
in Prescribed Acceleration, 530
description of, 534
in Prescribed Displacement, 521
description of, 525
in Prescribed Rotation, 538
description of, 542
in Prescribed Rotational Velocity, 543
description of, 547
in Prescribed Temperature, 591
description of, 593
in Prescribed Velocity, 526
description of, 529
in Pressure, 557
description of, 561
in Time Step Initialization
description of, 884
copy_data, 912
CP12
in Barlat Plasticity material model, 274
CP13
in Barlat Plasticity material model, 274
CP21
in Barlat Plasticity material model, 274
CP23
in Barlat Plasticity material model, 274
CP31
in Barlat Plasticity material model, 274
CP32
in Barlat Plasticity material model, 274
CP44
in Barlat Plasticity material model, 274
CP55
in Barlat Plasticity material model, 274
CP66
in Barlat Plasticity material model, 274
CPP12

in Barlat Plasticity material model, 274
CPP13
in Barlat Plasticity material model, 274
CPP21
in Barlat Plasticity material model, 274
CPP23
in Barlat Plasticity material model, 274
CPP31
in Barlat Plasticity material model, 274
CPP32
in Barlat Plasticity material model, 274
CPP44
in Barlat Plasticity material model, 274
CPP55
in Barlat Plasticity material model, 274
CPP66
in Barlat Plasticity material model, 274
CREATE VISUALIZATION ELEMENTS
in Spot Weld, 610
CREEP CONSTANT
in Power Law Creep material model, 301
in Viscoplastic material model, 303
CREEP EXPONENT
in Power Law Creep material model, 301
in Viscoplastic material model, 303
CRITERION IS
in Element Death – for Always True, 477
description of, 483
in Element Death – for Element Value Of, 477
description of, 481
in Element Death – for Global Value Of, 477
description of, 482
in Element Death – for Nodal Value Of, 477
description of, 480
CRITICAL BIAXIALITY RATIO
in Multilinear Elastic-Plastic Hardening Model with Failure material model, 244
CRITICAL CRACK OPENING STRAIN
in Ductile Fracture material model, 237
in Elastic Fracture material model, 228
in Multilinear Elastic-Plastic Hardening Model with Failure material model, 244
CRITICAL NORMAL GAP
in Contact Definition – in Cohesive Zone Model
description of, 665
CRITICAL TANGENTIAL GAP
in Contact Definition – in Cohesive Zone Model
description of, 665
CRITICAL TEARING PARAMETER
in Ductile Fracture material model, 237
in Multilinear Elastic-Plastic Hardening Model with Failure material model, 244
CRUSH XX
in Orthotropic Crush material model, 329
CRUSH XY
in Orthotropic Crush material model, 329
CRUSH YY
in Orthotropic Crush material model, 329
CRUSH YZ
in Orthotropic Crush material model, 329
CRUSH ZX
in Orthotropic Crush material model, 329
CRUSH ZZ
in Orthotropic Crush material model, 329
CUT OFF STRAIN
in Swanson material model, 357
in Viscoelastic Swanson material model, 360
CUTBACK FACTOR
in Adaptive Time Stepping
description of, 201
CYCLE COUNT
in Restart Data, 797
description of, 806
CYLINDRICAL AXIS
in Centripetal Force, 588
in Initial Velocity, 553
description of, 555
in Prescribed Displacement, 521
description of, 523
in Prescribed Velocity, 526
description of, 528
in Rigid Body command block, 469
CYLINDRICAL COORDINATE SYSTEM
in Coordinate System, 77

DAMAGE EXPONENT
in BCJ material model, 289
in BCJ_MEM material model, 292
DAMPER SECTION, 441
DAMPING MATRIX
in Superelement Section, 449
description of, 450
DATABASE NAME
in Finite Element Model, 383
description of, 386
in History Output, 780
in Restart Data, 797
description of, 799
in Results Output, 742
DATABASE TYPE
in Finite Element Model, 383
description of, 386
in History Output, 780
in Restart Data, 797
description of, 799
in Results Output, 742
DEATH METHOD
in Element Death, 477

in Element Death
description of, 487
DEATH ON ILL DEFINED CONTACT
in Element Death
description of, 486
DEATH ON INVERSION
in Element Death, 477
in Element Death
description of, 486
DEATH PROXIMITY
in Element Death, 477
description of, 489
DEATH START TIME
in Element Death, 477
description of, 485
DEATH STEPS
in Element Death, 477
description of, 486
DEBUG ITERATION PLOT
in Contact Definition – in Remove Initial Overlap
description of, 693
DEBUG MEMORY
in Zoltan Parameters, 498
DEBUG PROCESSOR NUMBER
in Zoltan Parameters, 498
DECAY COEFFICIENT
in Plane Stress Rate Plasticity material model, 336
DEFINE AXIS
with point and direction, 71
with two points, 71
DEFINE DIRECTION, 71
DEFINE POINT, 40, 71
DENSITY
in Fluid Pressure, 600
usage in, 601
DENSITY FOR PLASTIC DISSIPATION CALCULATIONS
in BCJ_MEM material model, 292
DENSITY FORMULATION
in Particle Section, 444
DENSITY FUNCTION
in Fluid Pressure, 600
usage in, 601
DENSITY SCALE FACTOR
in Finite Element Model – in Parameters For Block, 389
DEPTH
in Fluid Pressure, 600
usage in, 601
DEPTH FUNCTION
in Fluid Pressure, 600
usage in, 601
DERIVED LOG STRAIN OUTPUT, 495
DERIVED OUTPUT, 495
DERIVED STRAIN OUTPUT, 495
DETECT BREACH, 846
DETERMINISTIC DECOMPOSITION
in Zoltan Parameters, 498
DEVIATORIC PARAMETER
in Membrane Section, 425
in Solid Section, 404
DIFFERENTIATE EXPRESSION
in Function, 59
DIRECTION
in Gravity, 586
in Initial Velocity, 553
description of, 555
in Prescribed Acceleration, 530
description of, 532
in Prescribed Displacement, 521
description of, 523
in Prescribed Force, 572
description of, 573
in Prescribed Moment, 577
description of, 579
in Prescribed Rotation, 538
description of, 541
in Prescribed Rotational Velocity, 543
description of, 545
in Prescribed Velocity, 526
description of, 528
in Rigid Body command block, 469
in Traction, 566
description of, 567
DIRECTION FOR ROTATION
in Elastic 3D Orthotropic material model, 323
in Hill Plasticity material model, 263
DISTANCE TYPE
in Initial Condition, 509
DISTRIBUTE REFERENCE NODE MASS
in RBE3, 633
DRILLING STIFFNESS FACTOR
in Shell Section, 414

EDGE
in Heartbeat Output, 786
in Heartbeat Output – for mesh entities variables
description of, 789
in Heartbeat Output – for nearest point variables
description of, 790
in History Output, 780
about, 781, 788
in History Output – for mesh entities variables
description of, 782
in History Output – for nearest point variables
description of, 783
in Results Output, 742

EFFECTIVE MODULI MODEL
in Finite Element Model – in Parameters For Block, 389
description of, 395
EIGENVALUE CONVERGENCE TOLERANCE
in Lanczos Parameters, 118
in Power Method Parameters, 124
ELEMENT
in Heartbeat Output, 786
in Heartbeat Output – for mesh entities variables
description of, 789
in Heartbeat Output – for nearest point variables
description of, 790
in History Output, 780
about, 781, 788
in History Output – for mesh entities variables
description of, 782
in History Output – for nearest point variables
description of, 783
in Inertia Relief, 604
usage in, 604
ELEMENT BLOCK SUBROUTINE
as user subroutine command line, 880
description of, 881
in Element Death, 477
description of, 484
in Initial Condition, 509
description of, 516
in Pore Pressure, 596
in Time Step Initialization, 882
description of, 884
in User Output, 757
description of, 771
ELEMENT DEATH, 477
example of, 492
ELEMENT GROUPING TYPE
in Rebalance, 495
ELEMENT LIE VARIABLES
in Procedural Transfer, 917
description of, 919
ELEMENT NUMERICAL FORMULATION
in Finite Element Model – in Parameters For Block, 389
description of, 396
ELEMENT VARIABLES
in Procedural Transfer, 917
description of, 919
in Results Output, 742
description of, 749
ELLIPSOIDAL COORDINATE SYSTEM
in Coordinate System, 77
EMBEDDED BLOCKS
in Submodel
usage in, 632
ENCLOSING BLOCKS
in Submodel
usage in, 632
ENFORCEMENT
in Contact Definition
use of, 693
ENFORCEMENT OPTIONS
in Contact Definition
description of Options, 704
EQUATION MPC, 630
EVALUATE EXPRESSION
in Function, 59
evaluate expression
in Function
examples of, 64
rules and options for composing, 63
EX
in Orthotropic Crush material model, 329
EXCLUDE
in Results Output, 742, 752
EXPANSION RADIUS
in Cavity Expansion, 607
EXTERNAL FORCE CONTRIBUTION OUTPUT NAME, 557
in Inertia Relief
description of, 605
in Pressure
description of, 563
EXTRAPOLATE ELEMENT VARIABLE
in User Output, 757
description of, 773
EXTRAPOLATE PROJECTION TYPE
description of, 773
in User Output, 757
EY
in Orthotropic Crush material model, 329
EZ
in Orthotropic Crush material model, 329

F1 1
in Universal Polymer Material Model, 346
F2 1
in Universal Polymer Material Model, 346
FACE
in Heartbeat Output, 786
in Heartbeat Output – for mesh entities variables
description of, 789
in Heartbeat Output – for nearest point variables
description of, 790
in History Output, 780
about, 781, 788
in History Output – for mesh entities variables
description of, 782
in History Output – for nearest point variables

description of, 783
FACE VARIABLES
in Results Output, 742
description of, 747
FACTOR
in BCJ material model, 289
FAILURE DECAY CYCLES
in Line Weld, 617
in Spot Weld, 610
FAILURE ENVELOPE EXPONENT
in Line Weld, 617
in Spot Weld, 610
FAILURE FUNCTION
in Spot Weld, 610
FETI EQUATION SOLVER, 165
FIBER ORIENTATION
in Shell Section, 414
FIELD VARIABLE
in Pressure, 557
description of, 562
FILE
in Superelement Section, 449
description of, 450
FILLER VOL FRACTION
in Universal Polymer Material Model, 346
FINAL RADIUS MULTIPLICATION FACTOR
in Particle Section, 444
FINITE ELEMENT MODEL, 383
FIXED DISPLACEMENT, 519
FIXED ROTATION, 536
FLANGE THICKNESS
in Beam Section, 428
FLOW RULE COEFFICIENT CONSTANT
in BCJ_MEM material model, 292
FLOW RULE COEFFICIENT TEMPERATURE DEPENDENCE
in BCJ_MEM material model, 292
FLOW RULE EXPONENT CONSTANT
in BCJ_MEM material model, 292
FLOW RULE EXPONENT TEMPERATURE DEPENDENCE
in BCJ_MEM material model, 292
FLUID PRESSURE, 600
FLUID SURFACE NORMAL
in Fluid Pressure, 600
usage in, 601
FORCE SCALE FACTOR
in Centripetal Force, 588
FORCE STRAIN FUNCTION
in Spring Section, 440
FORMAT
in Heartbeat Output, 786
FORMULATION
in Membrane Section, 425
in Particle Section, 444
in Shell Section, 414
in Solid Section, 404
FRACTURE CURVES
in Plane Stress Rate Plasticity material model, 336
FRACTURE STRAIN RATES
in Plane Stress Rate Plasticity material model, 336
FREE SURFACE
in Cavity Expansion, 607
FREE SURFACE EFFECT COEFFICIENTS
in Cavity Expansion – in Layer, 607
description of, 609
FRICTION MODEL
in Contact Definition – in Interaction, 676
description of, 680
in Contact Definition – in Interaction Defaults, 673
description of, 675
FRICTIONLESS MODEL
in Contact Definition
description of, 661
FULL TANGENT PRECONDITIONER
in CG, 141
description of, 147, 155
FUNCTION, 59
in Gravity, 586
in Initial Condition, 509
in Periodic, 534
in Pore Pressure, 596
description of, 597
in Prescribed Acceleration, 530
description of, 532
in Prescribed Displacement, 521
description of, 523
in Prescribed Force, 572
description of, 573
in Prescribed Moment, 577
description of, 579
in Prescribed Rotation, 538
description of, 541
in Prescribed Rotational Velocity, 543
description of, 545
in Prescribed Temperature, 591
description of, 592
in Prescribed Velocity, 526
description of, 528
in Pressure, 557
description of, 559
in Traction, 566
description of, 567
usage for thermal strains, 216

GAMMA
in Implicit Dynamics, 205
description of, 206
GENERAL CONTACT
in Contact Definition – in Interaction Defaults, 673
GLOBAL
in Heartbeat Output – for global variables
description of, 788
in History Output – for global variables
description of, 781
GLOBAL OPERATOR
in User Variable, 885
GLOBAL OUTPUT OPTIONS
description of, 817
GLOBAL VARIABLES
in Procedural Transfer, 917
in Results Output, 742
description of, 753
GLUED MODEL
in Contact Definition
description of, 664
GLUED NORMAL MODEL
in Contact Definition
description of, 665
GRAVITATIONAL CONSTANT
in Fluid Pressure, 600
usage in, 601
in Gravity, 586
GRAVITY, 586
GROWTH FACTOR
in Adaptive Time Stepping
description of, 201
GXY
in Orthotropic Crush material model, 329
GYZ
in Orthotropic Crush material model, 329
GZX
in Orthotropic Crush material model, 329

HARDENING CONSTANT
in Barlat Plasticity material model, 274
in Ductile Fracture material model, 237
in Elastic-Plastic Power-Law Hardening material model, 234
in Hill Plasticity material model, 263
in Hosford Plasticity material model, 254
in Plane Stress Rate Plasticity material model, 336
HARDENING EXPONENT
in Barlat Plasticity material model, 274
in Ductile Fracture material model, 237
in Elastic-Plastic Power-Law Hardening material model, 234
in Hill Plasticity material model, 263
in Hosford Plasticity material model, 254
in Plane Stress Rate Plasticity material model, 336
HARDENING FUNCTION
in Barlat Plasticity material model, 274
in Hill Plasticity material model, 263
in Hosford Plasticity material model, 254
in Multilinear Elastic-Plastic Hardening Model material model, 240
in Multilinear Elastic-Plastic Hardening Model with Failure material model, 244
in Plane Stress Rate Plasticity material model, 336
HARDENING FUNCTIONS
in Thermoelastic Plastic material model, 250
HARDENING MODULUS
in Barlat Plasticity material model, 274
in Elastic-Plastic material model, 230
in Hill Plasticity material model, 263
in Hosford Plasticity material model, 254
in Plane Stress Rate Plasticity material model, 336
in Thermoelastic Plastic material model, 250
HEARTBEAT OUTPUT, 786
HEIGHT
in Beam Section, 428
HISTORY OUTPUT, 780
HOURGLASS FORMULATION
in Solid Section, 404
HOURGLASS INCREMENT
in Solid Section, 404
HOURGLASS ROTATION
in Solid Section, 404
HOURGLASS STIFFNESS
in Finite Element Model – in Parameters For Block, 389
description of, 393
HOURGLASS VISCOSITY
in Finite Element Model – in Parameters For Block, 389
description of, 393
HYDRO EXPONENT
in Foam Plasticity material model, 315
HYDRO HARDENING
in Foam Plasticity material model, 315
HYDRO STRENGTH
in Foam Plasticity material model, 315

IDENTIFY FRAGMENTS, 844
IGNORE INITIAL OFFSET
in Spot Weld, 610
IMBALANCE TOLERANCE
in Zoltan Parameters, 498

IMPLICIT DAMAGE SOLVER NUMBER OF ITERATIONS
in BCJ_MEM material model, 292
IMPLICIT DAMAGE SOLVER RESIDUAL TOLERANCE
in BCJ_MEM material model, 292
IMPLICIT DYNAMICS, 205
INACTIVE FOR PROCEDURE
in Finite Element Model – in Parameters For Block, 389
INACTIVE PERIODS, 94
description of, 94
in Cavity Expansion, 607
in Centripetal Force, 588
in Element Death, 477
usage in, 491
in Fixed Displacement, 519
usage in, 520
in Fixed Rotation, 536
usage in, 538
in Fluid Pressure, 600
usage in, 602
in Gravity, 586
in Inertia Relief, 604
usage in, 605
in Line Weld, 617
in Mass Scaling, 129
usage in, 131
in Periodic, 534
in Pore Pressure, 596
usage in, 599
in Prescribed Acceleration, 530
usage in, 534
in Prescribed Displacement, 521
usage in, 525
in Prescribed Force, 572
usage in, 576
in Prescribed Moment, 577
usage in, 580
in Prescribed Rotation, 538
usage in, 543
in Prescribed Rotational Velocity, 543
usage in, 547
in Prescribed Temperature, 591
usage in, 595
in Prescribed Velocity, 526
usage in, 530
in Pressure, 557
in REFERENCE AXIS ROTATION
usage in, 551
in Silent Boundary, 609
in Spot Weld, 610
in Traction, 566
usage in, 570
in Viscous Damping, 622
usage in, 625
INCLUDE
in Results Output, 742, 752
INCLUDE ALL BLOCKS
description of, 503
in Centripetal Force, 588
in Element Death, 477
description of, 479
in Finite Element Model, 383
description of, 391
in Finite Element Model – in Parameters For Block, 389
in Fixed Displacement, 519
usage in, 520
in Fixed Rotation, 536
usage in, 537
in Gravity, 586
usage in, 586
in Inertia Relief, 604
usage in, 604
in Initial Condition, 509
usage in, 510
in Initial Temperature
usage in, 590
in Initial Velocity, 553
usage in, 554
in Mass Properties, 475
usage in, 475
in Mass Scaling, 129
usage in, 130
in Pore Pressure, 596
usage in, 597
in Prescribed Acceleration, 530
usage in, 532
in Prescribed Displacement, 521
usage in, 522
in Prescribed Force, 572
usage in, 573
in Prescribed Moment, 577
usage in, 578
in Prescribed Rotation, 538
usage in, 540
in Prescribed Rotational Velocity, 543
usage in, 545
in Prescribed Temperature, 591
usage in, 592
in Prescribed Velocity, 526
usage in, 527
in Pressure, 557
in Procedural Transfer, 917
description of, 919
in Time Step Initialization, 882
usage in, 884

in User Output, 757
usage in, 762
in Viscous Damping, 622
usage in, 623
INCLUDE BLOCKS WITH MATERIAL
in Element Death, 477
description of, 479
INCLUDE NODES IN
in Rigid Body command block, 469
INCREASE OVER STEPS
in Lanczos Parameters, 118
in Power Method Parameters, 124
INCREMENT INTERVAL
in Node Based Time Step Parameters, 128
INERTIA
in Rigid Body command block, 469
INERTIA RELIEF, 604
INITIAL ALPHA_XX
in BCJ material model, 289
in BCJ_MEM material model, 292
INITIAL ALPHA_XY
in BCJ material model, 289
in BCJ_MEM material model, 292
INITIAL ALPHA_XZ
in BCJ material model, 289
in BCJ_MEM material model, 292
INITIAL ALPHA_YY
in BCJ material model, 289
in BCJ_MEM material model, 292
INITIAL ALPHA_YZ
in BCJ material model, 289
in BCJ_MEM material model, 292
INITIAL ALPHA_ZZ
in BCJ material model, 289
in BCJ_MEM material model, 292
INITIAL CONDITION, 509
INITIAL DAMAGE
in BCJ material model, 289
in BCJ_MEM material model, 292
INITIAL KAPPA
in BCJ material model, 289
in BCJ_MEM material model, 292
INITIAL LOAD
in Truss Section, 438
INITIAL MESH MODIFICATION, 507
INITIAL TEMPERATURE, 589
INITIAL TEMPERATURE FOR UNCOUPLED ADIABATIC HEATING
in BCJ_MEM material model, 292
INITIAL TIME STEP
in Parameters For Presto Region, 105
description of, 106
INITIAL VALUE
in User Variable, 885
INITIAL VELOCITY, 553
INITIAL ZETA
in BCJ_MEM material model, 292
INITIALIZE VARIABLE NAME
in Initial Condition, 509
description of, 511
INPUT DATABASE NAME
in Restart Data, 797
description of, 799
INTEGRATION METHOD
in Pressure, 557
INTEGRATION RULE
in Shell Section, 414
INTERACTION
in Contact Definition
description of, 676
INTERACTION BEHAVIOR
in Contact Definition – in Interaction, 676
description of, 679
in Contact Definition – in Interaction Defaults, 673
description of, 675
INTERACTION DEFAULTS
in Contact Definition
description of, 673
INTERFACE MATERIAL
in Contact Definition – in Interaction
description of, 680
INTERPOLATION TRANSFER
in Procedural Transfer, 917
description of, 920
ISOTROPIC DYNAMIC RECOVERY CONSTANT
in BCJ_MEM material model, 292
ISOTROPIC DYNAMIC RECOVERY TEMPERATURE DEPENDENCE
in BCJ_MEM material model, 292
ISOTROPIC HARDENING CONSTANT
in BCJ_MEM material model, 292
ISOTROPIC HARDENING SHEAR COEFFICIENT
in BCJ_MEM material model, 292
ISOTROPIC HARDENING TEMPERATURE DEPENDENCE
in BCJ_MEM material model, 292
ISOTROPIC STATIC RECOVERY CONSTANT
in BCJ_MEM material model, 292
ISOTROPIC STATIC RECOVERY SINH CONSTANT
in BCJ_MEM material model, 292
ISOTROPIC STATIC RECOVERY SINH TEMPERATURE DEPENDENCE
in BCJ_MEM material model, 292
ISOTROPIC STATIC RECOVERY TEMPERATURE DEPENDENCE
in BCJ_MEM material model, 292

ITERATION PLOT
in CG, 141
description of, 153
in Control Contact, 171
in Control Damped Solve, 189
in Control Failure, 186
in Control Stiffness, 182
ITERATION PLOT OUTPUT BLOCKS
in CG, 141
description of, 153
in Control Contact, 171
in Control Damped Solve, 189
in Control Failure, 186
in Control Stiffness, 182
ITERATION PRINT
in CG, 141
description of, 153
in Full Tangent Preconditioner, 155
description of, 159
ITERATION RESET
in CG, 141
description of, 151
ITERATION UPDATE
in Full Tangent Preconditioner, 155
description of, 159
ITERATION WINDOW
in Adaptive Time Stepping
description of, 201

K SCALING
in Incompressible Solid material model, 343
KEEP CUTS
in Zoltan Parameters, 498

L FUNCTION
in Orthotropic Rate material model, 333
L2_PROJECTION TRANSFER
in Procedural Transfer, 917
description of, 920
LABELS
in Heartbeat Output, 786
LAGRANGE ACCEPTABLE GAP
in Control Contact, 171
description of, 175
LAGRANGE ACCEPTABLE RELATIVE GAP
in Control Contact, 171
description of, 175
LAGRANGE ADAPTIVE PENALTY
in Control Contact, 171
description of, 176
LAGRANGE ADAPTIVE PENALTY GROWTH FACTOR
in Control Contact, 171
description of, 176
LAGRANGE ADAPTIVE PENALTY REDUCTION FACTOR
in Control Contact, 171
description of, 176
LAGRANGE ADAPTIVE PENALTY THRESHOLD
in Control Contact, 171
description of, 176
LAGRANGE MAXIMUM PENALTY MULTIPLIER
in Control Contact, 171
description of, 176
LAGRANGE MAXIMUM UPDATES
in Control Contact, 171
description of, 177
LAGRANGE NODAL STIFFNESS MULTIPLIER
in Control Contact, 171
description of, 177
LAGRANGE TOLERANCE
in Control Contact, 171
description of, 177
LAMBDA
in Barlat Plasticity material model, 274
in BCJ material model, 289
in Ductile Fracture material model, 237
in Elastic 3D Orthotropic material model, 323
in Elastic Fracture material model, 228
in Elastic material model, 223
in Elastic-Plastic material model, 230
in Elastic-Plastic Power-Law Hardening material model, 234
in Foam Plasticity material model, 315
in Hill Plasticity material model, 263
in Hosford Plasticity material model, 254
in Hyperfoam material model, 321
in Incompressible Solid material model, 343
in Johnson–Cook material model, 286
in Low Density Foam material model, 319
in MD Viscoplastic material model, 308
in Multilinear Elastic-Plastic Hardening Model material model, 240
in Multilinear Elastic-Plastic Hardening Model with Failure material model, 244
in Neo-Hookean material model, 226
in Orthotropic Crush material model, 329
in Orthotropic Rate material model, 333
in Plane Stress Rate Plasticity material model, 336
in Power Law Creep material model, 301
in Soil and Crushable Foam material model, 312
in Stiff Elastic material model, 355
in Swanson material model, 357
in Thermoelastic material model, 225
in Thermoelastic Plastic material model, 250
in Viscoelastic Swanson material model, 360
in Viscoplastic material model, 303

in Wire Mesh material model, 327
LANCZOS PARAMETERS, 118
LAYER
in Cavity Expansion, 607
description of, 608
LAYER SURFACE
in Cavity Expansion – in Layer, 607
description of, 608
LEGEND
in Heartbeat Output, 786
LEVEL
in Control Contact, 171
in Control Failure, 186, 189
in Control Stiffness, 182
LEVEL 1 PREDICTOR
in Solver, 135
description of, 195
LINE SEARCH
in CG, 141
description of, 150
LINE WELD, 617
LINEAR BULK VISCOSITY
in Finite Element Model – in Parameters For Block, 389
description of, 393
LINEAR SOLVER
in Full Tangent Preconditioner, 155
description of, 157
in Procedural Transfer, 917
description of, 920
LOAD BALANCING METHOD
in Zoltan Parameters, 498
as default, 498
LOAD RATIO THRESHOLD
in Rebalance, 495
LOADSTEP PREDICTOR
in Solver, 193
LOCAL SOLVER
in FETI Equation Solver, 165
description of, 168
LOCALIZATION SECTION, 410
LOCK RCB DIRECTIONS
in Zoltan Parameters, 498
LOFTING FACTOR
in Membrane Section, 425
in Shell Section, 414
LOFTING FACTOR MESH VARIABLE
in Membrane Section, 425
in Shell Section, 414
LOG TIME INCREMENT
in Universal Polymer Material Model, 346
LUDERS STRAIN
in Ductile Fracture material model, 237
in Elastic-Plastic Power-Law Hardening material model, 234
LW FUNCTION
in Orthotropic Rate material model, 333
LX
in Orthotropic Rate material model, 333
LY
in Orthotropic Rate material model, 333
LZ
in Orthotropic Rate material model, 333

MAGNITUDE
in Initial Condition, 509
description of, 511
in Initial Velocity, 553
description of, 555
in Rigid Body command block, 469
MAP
in Superelement Section, 449
description of, 450
MASKING METHOD
in Pressure, 557
description of, 562
MASS
in Rigid Body command block, 469
MASS DAMPING COEFFICIENT
in Viscous Damping, 622
description of, 623
MASS LUMPING METHOD
in Solid Section, 404
MASS MATRIX
in Superelement Section, 449
description of, 450
MASS PROPERTIES, 475
MASS SCALING, 129
MATERIAL, 209
in Finite Element Model – in Parameters For Block, 389
description of, 391
MATERIAL CRITERION
in Element Death, 477
description of, 483
MAX POISSONS RATIO
in Incompressible Solid material model, 343
in Mooney-Rivlin material model, 352
in Swanson material model, 357
in Viscoelastic Swanson material model, 360
MAX STRESS
in Elastic Fracture material model, 228
MAX SUBINCREMENTS
in Power Law Creep material model, 301
in Viscoplastic material model, 303
MAXIMUM CHANGE IN DAMAGE ALLOWED PER TIMESTEP

in BCJ_MEM material model, 292
MAXIMUM FAILURE CUTBACKS
in Adaptive Time Stepping
description of, 202
MAXIMUM ITERATIONS
in CG, 141
description of, 143
in Control Contact, 171
description of, 173
in Control Damped Solve, 189
description of, 190
in Control Failure, 186
description of, 187
in Control Stiffness, 182
description of, 183
in FETI Equation Solver, 165
description of, 166
MAXIMUM ITERATIONS FOR LOADSTEP
in Full Tangent Preconditioner, 155
description of, 159
MAXIMUM ITERATIONS FOR MODELPROBLEM
in Full Tangent Preconditioner, 155
description of, 159
MAXIMUM MULTIPLIER
in Adaptive Time Stepping
description of, 202
MAXIMUM ORTHOGONALIZATION
in FETI Equation Solver, 165
description of, 167
MAXIMUM RESETS FOR LOADSTEP
in Full Tangent Preconditioner, 155
description of, 159
MAXIMUM RESETS FOR MODELPROBLEM
in Full Tangent Preconditioner, 155
description of, 159
MAXIMUM SMOOTHING ITERATIONS
in Full Tangent Preconditioner, 155
MAXIMUM STEP LENGTH
in CG, 141
in CG
description of, 151
MEMBRANE HOURGLASS STIFFNESS
in Finite Element Model – in Parameters For Block, 389
description of, 393
MEMBRANE HOURGLASS VISCOSITY
in Finite Element Model – in Parameters For Block, 389
description of, 393
MEMBRANE SECTION, 425
MESH QUALITY, 90
METHOD
in Adaptive Time Stepping
description of, 200
MINIMUM CONVERGENCE RATE
in Full Tangent Preconditioner, 155
description of, 163
MINIMUM ITERATIONS
in CG, 141
description of, 143
in Control Contact, 171
description of, 173
in Control Stiffness, 182
description of, 183
MINIMUM MULTIPLIER
in Adaptive Time Stepping
description of, 202
MINIMUM REFERENCE
in CG, 141
description of, 143
MINIMUM RESIDUAL IMPROVEMENT
in CG, 141
description of, 143
MINIMUM SMOOTHING ITERATIONS
in Full Tangent Preconditioner, 155
MINIMUM STEP LENGTH
in CG, 141
in CG
description of, 151
MISORIENTATION VARIABLE HARDENING CONSTANT
in BCJ_MEM material model, 292
MISORIENTATION VARIABLE HARDENING EXPONENT
in BCJ_MEM material model, 292
MODEL
in Finite Element Model – in Parameters For Block, 389
description of, 391
MODULUS FUNCTION
in Orthotropic Rate material model, 333
MODULUS LLLL
in Orthotropic Rate material model, 333
MODULUS LLWW
in Orthotropic Rate material model, 333
MODULUS LWLW
in Orthotropic Rate material model, 333
MODULUS TLTL
in Orthotropic Rate material model, 333
MODULUS TTLL
in Orthotropic Rate material model, 333
MODULUS TTTT
in Orthotropic Rate material model, 333
MODULUS TTWW
in Orthotropic Rate material model, 333
MODULUS WTWT
in Orthotropic Rate material model, 333

MODULUS WWWW
in Orthotropic Rate material model, 333
MOMENTUM BALANCE ITERATIONS
in Contact Definition – in Enforcement Options, 704
description of, 704
MONAGHAN EPSILON
in Particle Section, 444
MONAGHAN N
in Particle Section, 444
MONITOR
in Heartbeat Output, 786
description of, 795
MOUNT TYPE
in User Mount Section, 453
MOVE BLOCK
in Initial Mesh Modification, 507

N
in Hyperfoam material model, 321
NAIR
in Low Density Foam material model, 319
NEAREST ELEMENT COPY
in Procedural Transfer, 917
description of, 920
NEW MATERIAL
in Particle Conversion, 489
NODAL
in Heartbeat Output, 786
in Heartbeat Output – for mesh entities variables
description of, 789
in Heartbeat Output – for nearest point variables
description of, 790
in History Output, 780
about, 781, 788
in History Output – for mesh entities variables
description of, 782
in History Output – for nearest point variables
description of, 783
NODAL DIAGONAL SCALE
in CG, 141
description of, 147
NODAL DIAGONAL SHIFT
in CG, 141
description of, 147
NODAL PRECONDITIONER METHOD
in Full Tangent Preconditioner, 155
description of, 157
NODAL PROBE FACTOR
in CG, 141
description of, 147
NODAL VARIABLES
in Results Output, 742
description of, 744
NODE
in Heartbeat Output, 786
in Heartbeat Output – for mesh entities variables
description of, 789
in Heartbeat Output – for nearest point variables
description of, 790
in History Output, 780
about, 781, 788
in History Output – for mesh entities variables
description of, 782
in History Output – for nearest point variables
description of, 783
in Inertia Relief, 604
usage in, 604
NODE BASED ALPHA FACTOR
in Solid Section, 404
NODE BASED BETA FACTOR
in Solid Section, 404
NODE BASED STABILIZATION METHOD
in Solid Section, 404
NODE BASED TIME STEP PARAMETERS, 128
NODE SET
description of, 503
in Centripetal Force, 588
in Contact Definition – in Contact Surface (block)
usage in, 650
in Fixed Displacement, 519
usage in, 520
in Fixed Rotation, 536
usage in, 537
in Gravity, 586
usage in, 586
in Heartbeat Output, 786
in Heartbeat Output – for mesh entities variables
description of, 789
in Heartbeat Output – for nearest point variables
description of, 790
in History Output, 780
about, 781, 788
in History Output – for mesh entities variables
description of, 782
in History Output – for nearest point variables
description of, 783
in Inertia Relief, 604
usage in, 604
in Initial Condition, 509
usage in, 510
in Initial Velocity, 553
usage in, 554
in Mass Scaling, 129
usage in, 130
in Prescribed Acceleration, 530
usage in, 532

in Prescribed Displacement, 521
usage in, 522
in Prescribed Force, 572
usage in, 573
in Prescribed Moment, 577
usage in, 578
in Prescribed Rotation, 538
usage in, 540
in Prescribed Rotational Velocity, 543
usage in, 545
in Prescribed Velocity, 526
usage in, 527
in Spot Weld, 610
in Time Step Initialization, 882
usage in, 884
in User Output, 757
usage in, 762
NODE SET SUBROUTINE
as user subroutine command line, 880
description of, 880
in Initial Condition, 509
description of, 516
in Initial Velocity, 553
description of, 556
in Pore Pressure
description of, 598
in Prescribed Acceleration, 530
description of, 533
in Prescribed Displacement, 521
description of, 524
in Prescribed Force, 572
description of, 574
in Prescribed Moment, 577
description of, 579
in Prescribed Rotation, 538
description of, 542
in Prescribed Rotational Velocity, 543
description of, 546
in Prescribed Temperature, 591
description of, 593
in Prescribed Velocity, 526
description of, 529
in Pressure
description of, 560
in Time Step Initialization, 882
description of, 884
in Traction, 566
description of, 568
in User Output, 757
description of, 771
NODE SET VARIABLES
in Results Output, 742
description of, 745
NODE SETS TO DEFINE BODY AXIS
in Cavity Expansion, 607
NODE VARIABLES
in Procedural Transfer, 917
description of, 919
in Results Output, 742
description of, 744
NODESET
description of, 503
in Centripetal Force, 588
in Contact Definition – in Contact Surface (block)
usage in, 650
in Fixed Displacement, 519
usage in, 520
in Fixed Rotation, 536
usage in, 537
in Gravity, 586
usage in, 586
in Heartbeat Output, 786
in Heartbeat Output – for mesh entities variables
description of, 789
in Heartbeat Output – for nearest point variables
description of, 790
in History Output, 780
about, 781, 788
in History Output – for mesh entities variables
description of, 782
in History Output – for nearest point variables
description of, 783
in Inertia Relief, 604
usage in, 604
in Initial Condition, 509
usage in, 510
in Initial Velocity, 553
usage in, 554
in Mass Scaling, 129
usage in, 130
in Prescribed Acceleration, 530
usage in, 532
in Prescribed Displacement, 521
usage in, 522
in Prescribed Force, 572
usage in, 573
in Prescribed Moment, 577
usage in, 578
in Prescribed Rotation, 538
usage in, 540
in Prescribed Rotational Velocity, 543
usage in, 545
in Prescribed Velocity, 526
usage in, 527
in Spot Weld, 610
in Time Step Initialization, 882
usage in, 884

in User Output, 757
usage in, 762
NODESET VARIABLES
in Results Output, 742
description of, 745
NORMAL DISPLACEMENT FUNCTION
in Spot Weld, 610
NORMAL DISPLACEMENT SCALE FACTOR
in Spot Weld, 610
NORMAL TOLERANCE
in Contact Definition – in Interaction, 676
description of, 680
in Contact Definition – in Search Options, 702
description of, 703
NUM LOCAL SUBDOMAINS
in FETI Equation Solver, 165
description of, 168
NUMBER EIGENVALUES
in Lanczos Parameters, 118
NUMBER ITERATIONS
in Power Method Parameters, 124
NUMBER OF INTEGRATION POINTS
in Cohesive Section, 409
in Localization Section, 410
in Shell Section, 414
NUMBER OF TIME STEPS
in Parameters For Adagio Region, 197, 198
NUMERICAL SHIFT FUNCTION
in Viscoelastic Swanson material model, 360

OBJECT TYPE
in Pressure, 557
description of, 561
OFFSET MESH VARIABLE
in Shell Section, 414
OMIT BLOCK
in Finite Element Model, 383
description of, 387
OPTIONAL
in Restart Data, 797
description of, 799
ORDINATE
in function, 59
ORIENTATION, 73
in Shell Section, 414
ORIGIN
in Coordinate System, 77
ORIGIN NODESET
in Coordinate System, 77
ORTHOGONALITY MEASURE FOR RESET
in CG, 141
description of, 151
OUTPUT DATABASE NAME
in Restart Data, 797
description of, 799
OUTPUT MESH
EXPOSED SURFACE, 752
in Results Output, 742
description of, 752
REFINED, 752
OUTPUT ON SIGNAL
in Heartbeat Output, 786
description of, 792
in History Output, 780
description of, 785
in Restart Data, 797
description of, 807
in Results Output, 742
description of, 757
OUTPUT SCHEDULER, 810
example of, 811
use of, 810
OVER ALLOCATE MEMORY
in Zoltan Parameters, 498
as default, 498
OVERLAP ITERATIONS
in Contact Definition – in Remove Initial Overlap
description of, 693
OVERLAP NORMAL TOLERANCE
in Contact Definition – in Interaction, 676
description of, 680
in Contact Definition – in Remove Initial Overlap
description of, 693
OVERLAP TANGENTIAL TOLERANCE
in Contact Definition – in Interaction, 676
description of, 680
in Contact Definition – in Remove Initial Overlap
description of, 693
OVERLAY COUNT
in Restart Data, 797
description of, 806
OVERWRITE
in History Output, 780
in Restart Data, 797
in Results Output, 742

P0
in Low Density Foam material model, 319
P1
in Swanson material model, 357
in Viscoelastic Swanson material model, 360
PARAM-STRING debugMask VALUE
in FETI Equation Solver, 165
PARAM-STRING precision VALUE
in FETI Equation Solver, 165
description of, 167
PARAMETERS
in User Mount Section, 453

PARAMETERS FOR ADAGIO REGION
in Time Stepping Block, 196
PARAMETERS FOR BLOCK
in Finite Element Model, 383
about, 390
listing of, 389
PARAMETERS FOR MODEL
in material command blocks
description of, 210
PARAMETERS FOR MODEL BARLAT_PLASTICITY
in Barlat Plasticity material model, 274
PARAMETERS FOR MODEL BCJ
in BCJ material model, 289
PARAMETERS FOR MODEL BCJ_MEM
in BCJ_MEM material model, 292
PARAMETERS FOR MODEL DUCTILE FRACTURE
in Ductile Fracture material model, 237
PARAMETERS FOR MODEL ELASTIC
in Elastic material model, 223
in Neo-Hookean material model, 226
in Thermoelastic material model, 225
PARAMETERS FOR MODEL ELASTIC_3D_ORTHOTROPIC
in Elastic 3D Orthotropic material model, 323
PARAMETERS FOR MODEL ELASTIC_FRACTURE
in Elastic Fracture material model, 228
PARAMETERS FOR MODEL ELASTIC_PLASTIC
in Elastic-Plastic material model, 230
PARAMETERS FOR MODEL EP_POWER_HARD
in Elastic-Plastic Power-Law Hardening material model, 234
PARAMETERS FOR MODEL FOAM_PLASTICITY
in Foam Plasticity material model, 315
PARAMETERS FOR MODEL HILL_PLASTICITY
in Hill Plasticity material model, 263
PARAMETERS FOR MODEL HOSFORD_PLASTICITY
in Hosford Plasticity material model, 254
PARAMETERS FOR MODEL HYPERFOAM
in Hyperfoam material model, 321
PARAMETERS FOR MODEL INCOMPRESSIBLE_SOLID
in Incompressible Solid material model, 343
PARAMETERS FOR MODEL JOHNSON_COOK
in Johnson–Cook material model, 286
PARAMETERS FOR MODEL LOW_DENSITY_FOAM
in Low Density Foam material model, 319
PARAMETERS FOR MODEL MD_VISCOPLASTIC
in MD Viscoplastic material model, 308
PARAMETERS FOR MODEL ML_EP_FAIL
in Multilinear Elastic-Plastic Hardening Model material model, 240
in Multilinear Elastic-Plastic Hardening Model with Failure material mode, 244
PARAMETERS FOR MODEL MOONEY_RIVLIN
in Mooney-Rivlin material model, 352
PARAMETERS FOR MODEL ORTHOTROPIC_CRUSH
in Orthotropic Crush material model, 329
PARAMETERS FOR MODEL ORTHOTROPIC_RATE
in Orthotropic Rate material model, 333
PARAMETERS FOR MODEL PLANE_STRESS_RATE_PLASTICITY
in Plane Stress Rate Plasticity material model, 336
PARAMETERS FOR MODEL POWER_LAW_CREEP
in Power Law Creep material model, 301
in Viscoplastic material model, 303
PARAMETERS FOR MODEL SOIL_FOAM
in Soil and Crushable Foam material model, 312
PARAMETERS FOR MODEL STIFF_ELASTIC
in Stiff Elastic material model, 355
PARAMETERS FOR MODEL SWANSON
in Swanson material model, 357
PARAMETERS FOR MODEL THERMO ELASTIC PLASTIC
in Thermoelastic Plastic material model, 250
PARAMETERS FOR MODEL UNIVERSAL_POLYMER
in Universal Polymer material model, 346
PARAMETERS FOR MODEL VISCOELASTIC_SWANSON
in Viscoelastic Swanson material model, 360
PARAMETERS FOR MODEL WIRE_MESH
in Wire Mesh material model, 327
PARAMETERS FOR PRESTO REGION
in Time Stepping Block, 104
PARAMETERS FOR PRESTO REGION
in Time Stepping Block
contents of, 105
PARAMETERS VARIABLE
in User Mount Section, 453
PARTICLE CONVERSION
in Element Death, 477
description of, 489
PARTICLE EMBEDDING, 493
PARTICLE SECTION, 444
in Particle Conversion, 489
PENALTY FACTOR
in Full Tangent Preconditioner, 155

description of, 157
PERIOD
in Truss Section, 438
PERIODIC, 534
PHI
in Foam Plasticity material model, 315
in Low Density Foam material model, 319
PLASTIC DISSIPATION FACTOR
in BCJ_MEM material model, 292
POINT A
in Orientation, 73
POINT B
in Orientation, 73
POINT INERTIA
in Rigid Body command block, 469
POINT MASS
in Rigid Body command block, 469
POINT MASS SECTION, 442
POINT ON AXIS
in Periodic, 534
POISSON
in Hyperfoam material model, 321
POISSONS RATIO
in Barlat Plasticity material model, 274
in BCJ material model, 289
in BCJ_MEM model, 292
in Ductile Fracture material model, 237
in Elastic 3D Orthotropic material model, 323
in Elastic Fracture material model, 228
in Elastic material model, 223
in Elastic-Plastic material model, 230
in Elastic-Plastic Power-Law Hardening material model, 234
in Foam Plasticity material model, 315
in Hill Plasticity material model, 263
in Hosford Plasticity material model, 254
in Hyperfoam material model, 321
in Incompressible Solid material model, 343
in Johnson–Cook material model, 286
in Low Density Foam material model, 319
in MD Viscoplastic material model, 308
in Multilinear Elastic-Plastic Hardening Model material model, 240
in Multilinear Elastic-Plastic Hardening Model with Failure material model, 244
in Neo-Hookean material model, 226
in Orthotropic Crush material model, 329
in Orthotropic Rate material model, 333
in Plane Stress Rate Plasticity material model, 336
in Power Law Creep material model, 301
in Soil and Crushable Foam material model, 312
in Stiff Elastic material model, 355
in Swanson material model, 357
in Thermoelastic material model, 225
in Thermoelastic Plastic material model, 250
in Viscoelastic Swanson material model, 360
in Viscoplastic material model, 303
in Wire Mesh material model, 327
POISSONS RATIO AB
in Elastic 3D Orthotropic material model, 323
POISSONS RATIO BC
in Elastic 3D Orthotropic material model, 323
POISSONS RATIO CA
in Elastic 3D Orthotropic material model, 323
POISSONS RATIO FUNCTION
in BCJ material model, 289
in BCJ_MEM material model, 292
in Multilinear Elastic-Plastic Hardening Model material model, 240
in Multilinear Elastic-Plastic Hardening Model with Failure material model, 244
in Thermoelastic material model, 225
in Thermoelastic Plastic material model, 250
PORE PRESSURE, 596
POWER METHOD PARAMETERS, 124
PRECISION
in Heartbeat Output, 786
PRECONDITIONER
in CG, 141
description of, 147
PRECONDITIONER ITERATION UPDATE
in CG, 141
description of, 147
PRECONDITIONING METHOD
in FETI Equation Solver, 165
description of, 167
PREDICTOR TYPE
in Loadstep Predictor
description of, 193
PRESCRIBED ACCELERATION, 530
PRESCRIBED DISPLACEMENT, 521
PRESCRIBED FORCE, 572
PRESCRIBED MOMENT, 577
PRESCRIBED QUANTITY
in Periodic, 534
PRESCRIBED ROTATION, 538
PRESCRIBED ROTATIONAL Velocity, 543
PRESCRIBED TEMPERATURE, 591
usage for thermal strains, 216
PRESCRIBED VELOCITY, 526
PRESSURE, 557
PRESSURE COEFFICIENTS
in Cavity Expansion – in Layer, 607
description of, 608
PRESSURE CUTOFF
in Soil and Crushable Foam material model, 312
PRESSURE FUNCTION

in Soil and Crushable Foam material model, 312
PRESSURE INTEGRATION OPTION
in BCJ_MEM material model, 292
PRESTO PROCEDURE, 85
description of, 86
PRESTO REGION
in Presto Procedure, 85
description of, 87
PRINT BANNER INTERVAL FOR SOLUTION CONTROL
in Parameters For Presto Region, 105
PRINT BANNER INTERVAL FOR SOLUTION CONTROL
in Parameters For Presto Region
description of, 108
PROBE FACTOR
in Full Tangent Preconditioner, 155
description of, 157
PROBLEM DIMENSION
in Particle Section, 444
PROCEDURAL TRANSFER, 917
PROJECTION TYPE
in Procedural Transfer
description of, 920
PRONY SHEAR 1
in Viscoelastic Swanson material model, 360
PRONY SHEAR 10
in Viscoelastic Swanson material model, 360
PRONY SHEAR 2
in Viscoelastic Swanson material model, 360
PRONY SHEAR 3
in Viscoelastic Swanson material model, 360
PRONY SHEAR 4
in Viscoelastic Swanson material model, 360
PRONY SHEAR 5
in Viscoelastic Swanson material model, 360
PRONY SHEAR 6
in Viscoelastic Swanson material model, 360
PRONY SHEAR 7
in Viscoelastic Swanson material model, 360
PRONY SHEAR 8
in Viscoelastic Swanson material model, 360
PRONY SHEAR 9
in Viscoelastic Swanson material model, 360
PRONY SHEAR INFINITY
in Viscoelastic Swanson material model, 360
PUSHBACK FACTOR
in Contact Definition – in Interaction, 676
description of, 682

Q1
in Swanson material model, 357
in Viscoelastic Swanson material model, 360
QUADRATIC BULK VISCOSITY
in Finite Element Model – in Parameters For Block, 389
description of, 393

R DISPLACEMENT FUNCTION
in Line Weld, 617
R DISPLACEMENT SCALE FACTOR
in Line Weld, 617
R ROTATION FUNCTION
in Line Weld, 617
R ROTATION SCALE FACTOR
in Line Weld, 617
R VECTOR READ VARIABLE
in Shell Section, 414
R1
in Swanson material model, 357
in Viscoelastic Swanson material model, 360
R11
in Hill Plasticity material model, 263
R12
in Hill Plasticity material model, 263
R22
in Hill Plasticity material model, 263
R23
in Hill Plasticity material model, 263
R31
in Hill Plasticity material model, 263
R33
in Hill Plasticity material model, 263
RADIAL AXIS
in Prescribed Displacement, 521
description of, 523
in Prescribed Velocity, 526
description of, 528
RADIUS MESH VARIABLE
in Particle Section, 444
RADIUS TIME STEP
in Particle Section, 444
RATE CONSTANT
in Plane Stress Rate Plasticity material model, 336
RATE FUNCTION
in Orthotropic Rate material model, 333
in Plane Stress Rate Plasticity material model, 336
RATE INDEPENDENT YIELD CONSTANT
in BCJ_MEM material model, 292
RATE INDEPENDENT YIELD SHEAR COEFFICIENT
in BCJ_MEM material model, 292
RATE INDEPENDENT YIELD TEMPERATURE DEPENDENCE
in BCJ_MEM material model, 292

RATE INDEPENDENT YIELD TEMPERATURE DEPENDENCE2
in BCJ_MEM material model, 292
RATE INDEPENDENT YIELD TEMPERATURE DEPENDENCE3
in BCJ_MEM material model, 292
RATE INDEPENDENT YIELD TEMPERATURE DEPENDENCE4
in BCJ_MEM material model, 292
RBE3, 633
READ VARIABLE
in Initial Condition, 509
in Pore Pressure, 596
description of, 598
in Prescribed Acceleration, 530
description of, 534
in Prescribed Displacement, 521
description of, 525
in Prescribed Rotation, 538
description of, 542
in Prescribed Rotational Velocity, 543
description of, 547
in Prescribed Temperature, 591
description of, 593
in Prescribed Velocity, 526
description of, 529
in Pressure, 557
description of, 561
in Time Step Initialization
description of, 884
REBALANCE, 495
REBALANCE STEP INTERVAL
in Rebalance, 495
REBALANCE STRATEGY
in Rebalance
description of, 496
REBALANCE TYPE
in Rebalance, 495
description of, 496
RECEIVE BLOCKS
in Procedural Transfer, 917
description of, 920
RECEIVE COORDINATES
in Procedural Transfer, 917
description of, 920
RECEIVE FROM TRANSFER
in Pore Pressure, 596
in Prescribed Temperature, 591
RECTANGULAR COORDINATE SYSTEM
in Coordinate System, 77
RECTILINEAR RCB BLOCKS
in Zoltan Parameters, 498
REFERENCE
in CG, 141
description of, 143
in Control Stiffness, 182
description of, 183
REFERENCE AXIS, 548
in Beam Section, 428
in Periodic, 534
REFERENCE AXIS X FUNCTION
in Reference Axis, 548
description of, 550
REFERENCE AXIS Y FUNCTION
in Reference Axis, 548
description of, 550
REFERENCE AXIS Z FUNCTION
in Reference Axis, 548
description of, 550
REFERENCE BLOCK
in RBE3, 633
REFERENCE LOCATION
in Rigid Body command block, 469
REFERENCE NODE SET
in RBE3, 633
REFERENCE NODES
in RBE3, 633
REFERENCE POINT
in Fluid Pressure, 600
usage in, 601
REFERENCE STRAIN
in Incompressible Solid material model, 343
in Mooney-Rivlin material model, 352
in Stiff Elastic material model, 355
in Swanson material model, 357
in Viscoelastic Swanson material model, 360
REFERENCE SURFACE
in RBE3, 633
REFERENCE TEMPERATURE
in Universal Polymer Material Model, 346
RELAX TIME 1
in Universal Polymer Material Model, 346
REMOVE BLOCK
description of, 505
in Centripetal Force, 588
in Contact Definition –in Contact Surface (block)
usage in, 650
in Element Death, 477
description of, 479
in Finite Element Model, 383
description of, 392
in Finite Element Model – in Parameters For Block, 389
in Fixed Displacement, 519
usage in, 520
in Fixed Rotation, 536
usage in, 537
in Gravity, 586

usage in, 586
in Inertia Relief, 604
usage in, 604
in Initial Condition, 509
usage in, 510
in Initial Temperature
usage in, 590
in Initial Velocity, 553
usage in, 554
in Line Weld, 617
in Mass Properties, 475
usage in, 475
in Mass Scaling, 129
usage in, 130
in Pore Pressure, 596
usage in, 597
in Prescribed Acceleration, 530
usage in, 532
in Prescribed Displacement, 521
usage in, 522
in Prescribed Force, 572
usage in, 573
in Prescribed Moment, 577
usage in, 578
in Prescribed Rotation, 538
usage in, 540
in Prescribed Rotational Velocity, 543
usage in, 545
in Prescribed Temperature, 591
usage in, 592
in Prescribed Velocity, 526
usage in, 527
in Procedural Transfer, 917
description of, 919
in Time Step Initialization, 882
usage in, 884
in User Output, 757
usage in, 762
in Viscous Damping, 622
usage in, 623
REMOVE GAPS AND OVERLAPS
in MPC, 630
in TIED MPC, 627
REMOVE INITIAL OVERLAP
in Contact Definition
description of, 693
REMOVE NODE SET
description of, 505
in Centripetal Force, 588
in Contact Definition – in Contact Surface (block)
usage in, 650
in Fixed Displacement, 519
usage in, 520
in Fixed Rotation, 536
usage in, 537
in Gravity, 586
usage in, 586
in Inertia Relief, 604
usage in, 604
in Initial Condition, 509
usage in, 510
in Initial Velocity, 553
usage in, 554
in Mass Scaling, 129
usage in, 130
in Prescribed Acceleration, 530
usage in, 532
in Prescribed Displacement, 521
usage in, 522
in Prescribed Force, 572
usage in, 573
in Prescribed Moment, 577
usage in, 578
in Prescribed Rotation, 538
usage in, 540
in Prescribed Rotational Velocity, 543
usage in, 545
in Prescribed Velocity, 526
usage in, 527
in Spot Weld, 610
in Time Step Initialization, 882
usage in, 884
in User Output, 757
usage in, 762
REMOVE SURFACE
description of, 505
in Cavity Expansion, 607
in Centripetal Force, 588
in Contact Definition – in Contact Surface (block)
usage in, 650
in Fixed Displacement, 519
usage in, 520
in Fixed Rotation, 536
usage in, 537
in Gravity, 586
usage in, 586
in Inertia Relief, 604
usage in, 604
in Initial Condition, 509
usage in, 510
in Initial Velocity, 553
usage in, 554
in Line Weld , 617
in Mass Scaling, 129
usage in, 130
in Prescribed Acceleration, 530

usage in, 532
in Prescribed Displacement, 521
usage in, 522
in Prescribed Force, 572
usage in, 573
in Prescribed Moment, 577
usage in, 578
in Prescribed Rotation, 538
usage in, 540
in Prescribed Rotational Velocity, 543
usage in, 545
in Prescribed Velocity, 526
usage in, 527
in Pressure, 557
in Silent Boundary, 609
in Spot Weld, 610
in Time Step Initialization, 882
usage in, 884
in Traction, 566
in User Output, 757
usage in, 762
RENUMBER PARTITIONS
in Zoltan Parameters, 498
RESET AT NEW PERIOD
in Adaptive Time Stepping
description of, 203
RESET LIMITS
in CG, 141
description of, 151
RESIDUAL NORM TOLERANCE
in FETI Equation Solver, 165
description of, 166
RESIDUAL NORM TYPE
in CG, 141
description of, 143
RESIDUAL ROUNDOFF TOLERANCE
in CG
description of, 143
in CG, 141
RESOLVE MULTIPLE MPCS, 630
RESTART, 58
about, 58
RESTART DATA, 797
about, 58
about auto read and write, 800
about overwriting, 804
about recovering, 805
about user-controlled read and write, 802
Implicit/Explicit Restart, 808
RESTART TIME, 58
about, 58
RESULTS OUTPUT, 742
REUSE CUTS
in Zoltan Parameters, 498
as default, 498
RHO
in BCJ material model, 289
RIGID BODIES FROM ATTRIBUTES
in Beam Section, 428
in Membrane Section, 425
in Point Mass Section, 442
in Shell Section, 414
in Solid Section, 404
in Truss Section, 438
RIGID BODY
in Fixed Displacement, 519
as command block, 469
in Beam Section, 428
in Fixed Displacement
usage in, 520
in Fixed Rotation, 536
usage in, 537
in Initial Velocity, 553
usage in, 554
in Membrane Section, 425
in Point Mass Section, 442
in Prescribed Acceleration, 530
usage in, 532
in Prescribed Displacement, 521
usage in, 522
in Prescribed Force, 572
usage in, 573
in Prescribed Moment, 577
usage in, 578
in Prescribed Rotation, 538
usage in, 540
in Prescribed Rotational Velocity, 543
usage in, 545
in Prescribed Velocity, 526
usage in, 527
in Reference Axis, 548
in Reference Axis Rotation
usage in, 549
in Shell Section, 414
in Solid Section, 404
in Truss Section, 438
ROTATE BLOCK
in Initial Mesh Modification, 507
ROTATION
in Reference Axis, 548
in Reference Axis Rotation
description of, 550
ROTATION ABOUT
in Orientation, 73
ROTATIONAL SCALE FACTOR
in Centripetal Force, 588
ROTATIONAL VELOCITY
in Reference Axis, 548

in Reference Axis Rotation
description of, 550
ROTATIONAL VELOCITY FUNCTION
in Centripetal Force, 588
RVE COORDINATE SYSTEM
in Solid Section, 404

S AXIS
in User Mount Section, 453
S AXIS VARIABLE
in User Mount Section, 453
S DISPLACEMENT FUNCTION
in Line Weld, 617
S DISPLACEMENT SCALE FACTOR
in Line Weld, 617
SCALAR SOURCE VARIABLE
in Pressure, 557
description of, 562
SCALE FACTOR
in Gravity, 586
in Initial Condition, 509
description of, 517
in Lanczos Parameters, 118
in Loadstep Predictor, 193
description of, 194
in Periodic, 534
in Pore Pressure, 596
description of, 599
in Power Method Parameters, 124
in Prescribed Acceleration, 530
description of, 534
in Prescribed Displacement, 521
description of, 525
in Prescribed Force, 572
description of, 576
in Prescribed Moment, 577
description of, 580
in Prescribed Rotation, 538
description of, 543
in Prescribed Rotational Velocity, 543
description of, 547
in Prescribed Temperature, 591
description of, 595
in Prescribed Velocity, 526
description of, 530
in Pressure, 557
description of, 563
in Reference Axis, 548
in REFERENCE AXIS ROTATION
usage in, 551
in Stiff Elastic material model, 355
in Traction, 566
description of, 570
SCALING FUNCTION
in Incompressible Solid material model, 343
SEARCH CONFIGURATION
in TIED MPC, 627
SEARCH OPTIONS
in Contact Definition
description of, 702
SEARCH TOLERANCE
in Contact Definition – in Search Options, 702
description of, 703
in Line Weld, 617
in Periodic, 534
in Spot Weld, 610
in TIED MPC, 627
SECOND ALPHA
in Elastic 3D Orthotropic material model, 323
in Hill Plasticity material model, 263
SECOND DIRECTION FOR ROTATION
in Elastic 3D Orthotropic material model, 323
in Hill Plasticity material model, 263
SECOND SURFACE
in Spot Weld, 610
SECTION
in Beam Section, 428
in Finite Element Model – in Parameters For Block, 389
description of, 392
general overview, 390
Section command blocks
about, 401
SELF CONTACT
in Contact Definition – in Interaction Defaults, 673
description of, 674
SEMI IMPLICIT PLASTIC STRAIN SOLVER NUMBER OF ITERATIONS
in BCJ_MEM material model, 292
SEMI IMPLICIT PLASTIC STRAIN SOLVER RESIDUAL TOLERANCE
in BCJ_MEM material model, 292
SEND BLOCKS
in Procedural Transfer, 917
description of, 920
SEND COORDINATES
in Procedural Transfer, 917
description of, 920
SET RCB DIRECTIONS
in Zoltan Parameters, 498
SHEAR
in Hyperfoam material model, 321
SHEAR EXPONENT
in Foam Plasticity material model, 315
SHEAR GLASSY 0
in Universal Polymer Material Model, 346
SHEAR HARDENING

in Foam Plasticity material model, 315
SHEAR MODULUS
in Barlat Plasticity material model, 274
in BCJ material model, 289
in Ductile Fracture material model, 237
in Elastic 3D Orthotropic material model, 323
in Elastic Fracture material model, 228
in Elastic material model, 223
in Elastic-Plastic material model, 230
in Elastic-Plastic Power-Law Hardening material model, 234
in Foam Plasticity material model, 315
in Hill Plasticity material model, 263
in Hosford Plasticity material model, 254
in Hyperfoam material model, 321
in Incompressible Solid material model, 343
in Johnson–Cook material model, 286
in Low Density Foam material model, 319
in MD Viscoplastic material model, 308
in Multilinear Elastic-Plastic Hardening Model material model, 240
in Multilinear Elastic-Plastic Hardening Model with Failure material model, 244
in Neo-Hookean material model, 226
in Orthotropic Crush material model, 329
in Orthotropic Rate material model, 333
in Plane Stress Rate Plasticity material model, 336
in Power Law Creep material model, 301
in Soil and Crushable Foam material model, 312
in Stiff Elastic material model, 355
in Thermoelastic material model, 225
in Thermoelastic Plastic material model, 250
in Viscoplastic material model, 303
in Wire Mesh material model, 327
SHEAR MODULUS AB
in Elastic 3D Orthotropic material model, 323
SHEAR MODULUS BC
in Elastic 3D Orthotropic material model, 323
SHEAR MODULUS CA
in Elastic 3D Orthotropic material model, 323
SHEAR RELAX TIME 1
in Viscoelastic Swanson material model, 360
SHEAR RELAX TIME 10
in Viscoelastic Swanson material model, 360
SHEAR RELAX TIME 2
in Viscoelastic Swanson material model, 360
SHEAR RELAX TIME 3
in Viscoelastic Swanson material model, 360
SHEAR RELAX TIME 4
in Viscoelastic Swanson material model, 360
SHEAR RELAX TIME 5
in Viscoelastic Swanson material model, 360
SHEAR RELAX TIME 6
in Viscoelastic Swanson material model, 360
SHEAR RELAX TIME 7
in Viscoelastic Swanson material model, 360
SHEAR RELAX TIME 8
in Viscoelastic Swanson material model, 360
SHEAR RELAX TIME 9
in Viscoelastic Swanson material model, 360
SHEAR RUBBERY 0
in Universal Polymer Material Model, 346
SHEAR STRENGTH
in Foam Plasticity material model, 315
SHELL SECTION, 414
SIDE A
in Contact Definition – in Interaction, 676
description of, 677
in Periodic, 534
SIDE A BLOCK
in MPC, 630
SIDE A NODE SET
in MPC, 630
SIDE A NODES
in MPC, 630
SIDE A SURFACE
in MPC, 630
SIDE B
in Contact Definition – in Interaction, 676
description of, 677
in Periodic, 534
SIDE B BLOCK
in MPC, 630
SIDE B NODE SET
in MPC, 630
SIDE B NODES
in MPC, 630
SIDE B SURFACE
in MPC, 630
SIERRA, 57
SILENT BOUNDARY, 609
SKIN ALL BLOCKS
in Contact Definition
description of, 652
SLIP SCALE FACTOR
in Loadstep Predictor, 193
description of, 194
SMALL NUMBER OF ITERATIONS
in Full Tangent Preconditioner, 155
description of, 159
SMALL STRAIN
in Lanczos Parameters, 118
in Power Method Parameters, 124
SMOOTHING STEPS
in Full Tangent Preconditioner, 155
SOLID SECTION, 404
SOLVER, 135

SPECIFIC HEAT
in BCJ material model, 289
SPECIFIC HEAT FOR PLASTIC DISSIPATION CALCULATIONS
in BCJ_MEM material model, 292
SPECTRUM END TIME
in Universal Polymer Material Model, 346
SPECTRUM STAT TIME
in Universal Polymer Material Model, 346
SPH ALPHAQ PARAMETER
in Particle Section, 444
SPH BETAQ PARAMETER
in Particle Section, 444
SPH DECOUPLE STRAINS
in SPH Options, 446
SPH OPTIONS, 446
SPH SYMMETRY PLANE
in SPH Options, 446
SPHERE INITIAL RADIUS
in Particle Section, 444
SPHERICAL COORDINATE SYSTEM
in Coordinate System, 77
SPOT WELD, 610
SPRING SECTION, 440
STABLE TIME STEP ESTIMATE
in Shell Section, 414
STAGNATION THRESHOLD
in Full Tangent Preconditioner, 155
description of, 163
START TIME
in Heartbeat Output, 786
in History Output, 780
in Output Scheduler, 810
in Restart Data, 797
in Results Output, 742
description of, 754
in Time Stepping Block, 104, 196
STARTING VECTOR
in Lanczos Parameters, 118
in Power Method Parameters, 124
STEP INTERVAL
in Node Based Time Step Parameters, 128
in Parameters For Presto Region, 105
description of, 107
in Rebalance
description of, 496
STIFFNESS DAMPING COEFFICIENT
in Viscous Damping, 622
description of, 623
STIFFNESS MATRIX
in Superelement Section, 449
description of, 450
STRAIN INCREMENTATION
in Solid Section, 404
STREAM NAME
in Heartbeat Output, 786
STRESS FREE TEMPERATURE
in Universal Polymer Material Model, 346
STRESS MEASURE
in Solid Section, 404
STRUCTURE NAME
in Mass Properties, 475
description of, 476
Submodel, 632
SUBROUTINE DEBUGGING OFF
as user subroutine command line, 880
description of, 881
in Element Death, 477
usage in, 484
in Initial Condition, 509
usage in, 516
in Initial Velocity, 553
usage in, 556
in Pore Pressure, 596
usage in, 598
in Prescribed Acceleration, 530
usage in, 533
in Prescribed Displacement, 521
usage in, 524
in Prescribed Force, 572
usage in, 574
in Prescribed Moment, 577
usage in, 580
in Prescribed Rotation, 538
usage in, 542
in Prescribed Rotational Velocity, 543
usage in, 547
in Prescribed Temperature, 591
usage in, 593
in Prescribed Velocity, 526
usage in, 529
in Pressure, 557
usage in, 560
in Time Step Initialization, 882
usage in, 885
in Traction, 566
usage in, 568
in User Output, 757
usage in, 772
SUBROUTINE DEBUGGING ON
as user subroutine command line, 880
description of, 881
in Element Death, 477
usage in, 484
in Initial Condition, 509
usage in, 516
in Initial Velocity, 553
usage in, 556

in Pore Pressure, 596
usage in, 598
in Prescribed Acceleration, 530
usage in, 533
in Prescribed Displacement, 521
usage in, 524
in Prescribed Force, 572
usage in, 574
in Prescribed Moment, 577
usage in, 580
in Prescribed Rotation, 538
usage in, 542
in Prescribed Rotational Velocity, 543
usage in, 547
in Prescribed Temperature, 591
usage in, 593
in Prescribed Velocity, 526
usage in, 529
in Pressure, 557
usage in, 560
in Time Step Initialization, 882
usage in, 885
in Traction, 566
usage in, 568
in User Output, 757
usage in, 772
SUBROUTINE INTEGER PARAMETER
as user subroutine command line, 880
description of, 881
in Element Death, 477
usage in, 484
in Initial Condition, 509
usage in, 516
in Initial Velocity, 553
usage in, 556
in Pore Pressure, 596
usage in, 598
in Prescribed Acceleration, 530
usage in, 533
in Prescribed Displacement, 521
usage in, 524
in Prescribed Force, 572
usage in, 574
in Prescribed Moment, 577
usage in, 580
in Prescribed Rotation, 538
usage in, 542
in Prescribed Rotational Velocity, 543
usage in, 547
in Prescribed Temperature, 591
usage in, 593
in Prescribed Velocity, 526
usage in, 529
in Pressure, 557
usage in, 560
in Time Step Initialization, 882
usage in, 885
in Traction, 566
usage in, 568
in User Output, 757
usage in, 772
SUBROUTINE REAL PARAMETER
as user subroutine command line, 880
description of, 881
in Element Death, 477
usage in, 484
in Initial Condition, 509
usage in, 516
in Initial Velocity, 553
usage in, 556
in Pore Pressure, 596
usage in, 598
in Prescribed Acceleration, 530
usage in, 533
in Prescribed Displacement, 521
usage in, 524
in Prescribed Force, 572
usage in, 574
in Prescribed Moment, 577
usage in, 580
in Prescribed Rotation, 538
usage in, 542
in Prescribed Rotational Velocity, 543
usage in, 547
in Prescribed Temperature, 591
usage in, 593
in Prescribed Velocity, 526
usage in, 529
in Pressure, 557
usage in, 560
in Time Step Initialization, 882
usage in, 885
in Traction, 566
usage in, 568
in User Output, 757
usage in, 772
usage with query function, 856
SUBROUTINE STRING PARAMETER
as user subroutine command line, 880
description of, 881
in Element Death, 477
usage in, 484
in Initial Condition, 509
usage in, 516
in Initial Velocity, 553
usage in, 556
in Pore Pressure, 596
usage in, 598

in Prescribed Acceleration, 530
usage in, 533
in Prescribed Displacement, 521
usage in, 524
in Prescribed Force, 572
usage in, 574
in Prescribed Moment, 577
usage in, 580
in Prescribed Rotation, 538
usage in, 542
in Prescribed Rotational Velocity, 543
usage in, 547
in Prescribed Temperature, 591
usage in, 593
in Prescribed Velocity, 526
usage in, 529
in Pressure, 557
usage in, 560
in Time Step Initialization, 882
usage in, 885
in Traction, 566
usage in, 568
in User Output, 757
usage in, 772
Subsetting
in Results Output, 752
SUMMARY OUTPUT STEP INTERVAL
in Element Death, 477
description of, 485
SUMMARY OUTPUT TIME INTERVAL
in Element Death, 477
description of, 485
SUPERELEMENT SECTION, 449
Support, 54
SURFACE
description of, 49, 503
in Cavity Expansion , 607
in Centripetal Force, 588
in Contact Definition – in Contact Surface(block)
usage in, 650
in Fixed Displacement, 519
usage in, 520
in Fixed Rotation, 536
usage in, 537
in Fluid Pressure, 600
in Gravity, 586
usage in, 586
in Inertia Relief, 604
usage in, 604
in Initial Condition, 509
usage in, 510
in Initial Velocity, 553
usage in, 554
in Line Weld, 617
in Mass Scaling, 129
usage in, 130
in Prescribed Acceleration, 530
usage in, 532
in Prescribed Displacement, 521
usage in, 522
in Prescribed Force, 572
usage in, 573
in Prescribed Moment, 577
usage in, 578
in Prescribed Rotation, 538
usage in, 540
in Prescribed Rotational Velocity, 543
usage in, 545
in Prescribed Velocity, 526
usage in, 527
in Pressure, 557
in Silent Boundary, 609
in Spot Weld, 610
in Time Step Initialization, 882
usage in, 884
in Traction, 566
in User Output, 757
usage in, 762
SURFACE EFFECT
in Cavity Expansion – in Layer, 607
description of, 609
SURFACE SUBROUTINE
as user subroutine command line, 880
description of, 881
in Initial Condition, 509
description of, 516
in Pressure, 557
description of, 560
in Time Step Initialization, 882
description of, 884
in User Output, 757
description of, 771
SURFACE VARIABLES
in Results Output, 742
SURFACES
in Contact Definition – in Interaction, 676
description of, 677
in Contact Definition – in Interaction Defaults, 673
description of, 674
SYNCHRONIZE OUTPUT
in Heartbeat Output, 786
description of, 791
in History Output, 780
description of, 785
in Restart Output, 797
description of, 807
in Results Output, 742

description of, 756
SYSTEM
in Orientation, 73

T AXIS
in Beam Section, 428
in User Mount Section, 453
T AXIS VARIABLE
in User Mount Section, 453
T DISPLACEMENT FUNCTION
in Line Weld, 617
T DISPLACEMENT SCALE FACTOR
in Line Weld, 617
T FUNCTION
in Orthotropic Rate material model, 333
T VECTOR READ VARIABLE
in Shell Section, 414
TANGENT DIAGONAL SCALE
in Full Tangent Preconditioner, 155
description of, 157
TANGENT DIAGONAL SHIFT
in Full Tangent Preconditioner, 155
description of, 157
TANGENTIAL DISPLACEMENT FUNCTION
in Spot Weld, 610
TANGENTIAL DISPLACEMENT SCALE FACTOR
in Spot Weld, 610
TANGENTIAL TOLERANCE
in Contact Definition – in Interaction, 676
description of, 680
in Contact Definition – in Search Options, 702
description of, 703
TARGET E
in Incompressible Solid material model, 343
in Mooney-Rivlin material model, 352
in Swanson material model, 357
in Viscoelastic Swanson material model, 360
TARGET E FUNCTION
in Mooney-Rivlin material model, 352
in Swanson material model, 357
in Viscoelastic Swanson material model, 360
TARGET ITERATIONS
in Adaptive Time Stepping
description of, 200
TARGET PRESSURE INCREMENT
in Control Stiffness, 182
description of, 183
TARGET RELATIVE CONTACT RESIDUAL
in Control Contact, 171
description of, 173
TARGET RELATIVE PRESSURE INCREMENT
in Control Stiffness, 182
description of, 183
TARGET RELATIVE RESIDUAL
in CG, 141
description of, 143
in Control Contact, 171
description of, 173
in Control Stiffness, 182
description of, 183
TARGET RELATIVE SDEV INCREMENT
in Control Stiffness, 182
description of, 183
TARGET RELATIVE STRAIN INCREMENT
in Control Stiffness, 182
description of, 183
TARGET RELATIVE STRESS INCREMENT
in Control Stiffness, 182
description of, 183
TARGET RESIDUAL
in CG, 141
description of, 143
in Control Contact, 171
description of, 173
in Control Stiffness, 182
description of, 183
TARGET SDEV INCREMENT
in Control Stiffness, 182
description of, 183
TARGET SMOOTHING RELATIVE RESIDUAL
in Full Tangent Preconditioner, 155
TARGET SMOOTHING RESIDUAL
in Full Tangent Preconditioner, 155
TARGET STRESS INCREMENT
in Control Stiffness, 182
description of, 183
TARGET TIME STEP
in Mass Scaling, 129
description of, 131
TEMPERATURE OPTION
in BCJ_MEM material model, 292
TEMPERATURE TYPE
in Prescribed Temperature
description of, 593
TEMPERATURES
in Thermoelastic Plastic material model, 250
TEMPO
in BCJ material model, 289
TENSION
in Wire Mesh material model, 327
TENSION RELEASE
in Contact Definition – in Interaction, 676
description of, 683
TENSION RELEASE FUNCTION
in Contact Definition – in Interaction, 676
description of, 683
TERMINATION TIME
in Heartbeat Output, 786

in History Output, 780
in Output Scheduler, 810
in Restart Data, 797
in Results Output, 742
description of, 754
in Time Control, 104, 196
THERMAL CONSTANT
in Power Law Creep material model, 301
in Viscoplastic material model, 303
THERMAL EXPANSION FUNCTION
in Mooney-Rivlin material model, 352
in Swanson material model, 357
in Viscoelastic Swanson material model, 360
THERMAL STRAIN AA FUNCTION
in Elastic 3D Orthotropic material model, 323
THERMAL STRAIN BB FUNCTION
in Elastic 3D Orthotropic material model, 323
THERMAL STRAIN CC FUNCTION
in Elastic 3D Orthotropic material model, 323
THETA
in Periodic, 534
THETA OPT
in BCJ material model, 289
THICKNESS
in Membrane Section, 425
in Shell Section, 414
THICKNESS MESH VARIABLE
in Membrane Section, 425
in Shell Section, 414
THICKNESS SCALE FACTOR
in Membrane Section, 425
in Shell Section, 414
THICKNESS TIME STEP
in Membrane Section, 425
in Shell Section, 414
TIED FACES
in TIED MPC, 627
TIED MODEL
in Contact Definition
description of, 663
TIED MPC, 627
TIED NODES
in TIED MPC, 627
TIME
in Initial Condition, 509
in Pore Pressure, 596
description of, 598
in Prescribed Acceleration, 530
description of, 534
in Prescribed Displacement, 521
description of, 525
in Prescribed Rotation, 538
description of, 542
in Prescribed Rotational Velocity, 543
description of, 547
in Prescribed Temperature, 591
description of, 593
in Prescribed Velocity, 526
description of, 529
in Pressure, 557
description of, 561
TIME CONTROL
example of, 108, 203
in Adagio Procedure, 85
contents and description of, 196
overview, 196
in Presto Procedure
general layout of, 102
in Presto Procedure, 85
contents and description of, 104
overview, 101
usage of, 102
in Procedure
about, 86
TIME INCREMENT
in Parameters For Adagio Region, 197, 198
TIME INCREMENT FUNCTION
in Parameters For Adagio Region, 197, 198
TIME STEP INCREASE FACTOR
in Parameters For Presto Region, 105
in Parameters For Presto Region
description of, 106
usage of, 103
TIME STEP INITIALIZATION, 882
TIME STEP LIMIT
in Node Based Time Step Parameters, 128
TIME STEP SCALE FACTOR
in Parameters For Presto Region, 105
in Parameters For Presto Region
description of, 106
usage of, 103
with Element Numerical Formulation, 397
TIME STEPPING BLOCK
in Time Control, 104
TIME STEPPING BLOCK
in Time Control, 196
description of, 104, 197
TIMESTEP ADJUSTMENT INTERVAL
in Heartbeat Output, 786
in History Output, 780
in Output Scheduler, 810
in Restart Data, 797
in Results Output, 742
description of, 754
TIMESTEP FORMAT
in Heartbeat Output, 786
TIP RADIUS
in Cavity Expansion, 607

TITLE, 57
in Heartbeat Output, 786
in History Output, 780
in Results Output, 742
TL FUNCTION
in Orthotropic Rate material model, 333
TOLERANCE
in Plane Stress Rate Plasticity material model, 336
TORQUE
in Reference Axis, 548
in Reference Axis Rotation
description of, 550
TRACTION, 566
TRACTION DISPLACEMENT FUNCTION
in Contact Definition – in Cohesive Zone Model
description of, 665
TRACTION DISPLACEMENT SCALE FACTOR
in Contact Definition – in Cohesive Zone Model
description of, 665
TRACTION INTEGRATION METHOD
in Traction, 566
Transfers, 916
TRANSFORMATION TYPE
in Procedural Transfer, 917
description of, 920
TRANSVERSE SHEAR HOURGLASS STIFFNESS
in Finite Element Model – in Parameters For Block, 389
description of, 393
TRANSVERSE SHEAR HOURGLASS VISCOSITY
in Finite Element Model – in Parameters For Block, 389
description of, 393
TRUSS SECTION, 438
TX
in Orthotropic Rate material model, 333
TY
in Orthotropic Rate material model, 333
TYPE
in Function, 59
in Loadstep Predictor, 193
in Time Step Initialization
description of, 884
in User Variable, 885
TZ
in Orthotropic Rate material model, 333

UPDATE ON TIME STEP CHANGE
in Lanczos Parameters, 118
in Power Method Parameters, 124
UPDATE STEP INTERVAL
in Lanczos Parameters, 118
in Power Method Parameters, 124
USE FINITE ELEMENT MODEL, 89
USE HHT INTEGRATION
in Implicit Dynamics, 205
description of, 205
USE OUTPUT SCHEDULER
example of, 811
in Heartbeat Output, 786
description of, 792
in History Output, 780
description of, 785
in Restart Data, 797
description of, 807
in Results Output, 742
description of, 756
USE WITH RESTART
in User Variable, 885
USER CRITERIA GLOBAL
in Adaptive Time Stepping
description of, 203
USER INTEGRATION RULE
in Shell Section, 414
USER MOUNT SECTION, 453
USER OUTPUT, 757
example of, 893
USER RATE DEPENDENCE
in Plane Stress Rate Plasticity material model, 336
USER SUBROUTINE FILE, 59
example of, 888
usage in context, 880
User Subroutines
aupst_check_elem_var, 860
aupst_check_global_var, 870
aupst_check_node_var, 860
aupst_cyl_transform, 909
aupst_evaluate_function, 859
aupst_get_elem_centroid, 877
aupst_get_elem_nodes, 872
aupst_get_elem_topology, 872
aupst_get_elem_var, 860
aupst_get_elem_var_offset, 860
aupst_get_face_nodes, 872
aupst_get_face_topology, 872
aupst_get_global_var, 870
aupst_get_integer_param, 854
aupst_get_node_var, 860
aupst_get_point, 877
aupst_get_proc_num, 877
aupst_get_real_param, 854
aupst_get_string_param, 854
aupst_get_time, 859
aupst_put_elem_var, 860
aupst_put_elem_var_offset, 860
aupst_put_global_var, 870

aupst_put_node_var, 860
aupst_rec_transform, 911
copy_data, 912
ELEMENT BLOCK SUBROUTINE, 880
HEARTBEAT OUTPUT, 786
HISTORY OUTPUT, 780
NODE SET SUBROUTINE, 880
RESULTS OUTPUT, 742
SUBROUTINE DEBUGGING OFF, 880
SUBROUTINE DEBUGGING ON, 880
SUBROUTINE INTEGER PARAMETER, 880
SUBROUTINE REAL PARAMETER, 880
SUBROUTINE STRING PARAMETER, 880
SURFACE SUBROUTINE, 880
USER OUTPUT, 757
USER VARIABLE, 885
USER VARIABLE, 885

VALUES
in Function, 59
VARIABLE TYPE
in Initial Condition, 509
description of, 511
VECTOR SCALE
in Lanczos Parameters, 118
in Power Method Parameters, 124
VELOCITY DAMPING COEFFICIENT
in Pressure, 557
description of, 560
in Viscous Damping
description of, 623
VISCOUS DAMPING, 622
VMIN
in Orthotropic Crush material model, 329
Void Elements, 405
VOLCTE GLASSY 0
in Universal Polymer Material Model, 346
VOLCTE RUBBERY 0
in Universal Polymer Material Model, 346
VOLUME
in Point Mass Section, 442
VOLUMETRIC EFFECTS OF DAMAGE OPTION
in BCJ_MEM material model, 292
VOLUMETRIC SEARCH TOLERANCE
in TIED MPC, 627

W FUNCTION
in Orthotropic Rate material model, 333
WALL THICKNESS
in Beam Section, 428
WEIBULL
in Initial Condition, 509
description of, 512
WEIBULL MEDIAN
in Initial Condition, 509
WEIBULL SCALE
in Initial Condition, 509
WEIBULL SCALING EXPONENT SCALE
in Initial Condition, 509
WEIBULL SCALING FIELD NAME
in Initial Condition, 509
WEIBULL SCALING FIELD TYPE
in Initial Condition, 509
WEIBULL SCALING REFERENCE VALUE
in Initial Condition, 509
WEIBULL SEED
in Initial Condition, 509
WEIBULL SHAPE
in Initial Condition, 509
WEIGHTS
in RBE3, 633
WIDTH
in Beam Section, 428
WLF C1
in Universal Polymer Material Model, 346
WLF C2
in Universal Polymer Material Model, 346
WLF COEF C1
in Viscoelastic Swanson material model, 360
WLF COEF C2
in Viscoelastic Swanson material model, 360
WLF TREF
in Viscoelastic Swanson material model, 360
WT FUNCTION
in Orthotropic Rate material model, 333
WWBETA 1
in Universal Polymer Material Model, 346
WWBETA 2
in Universal Polymer Material Model, 346
WWTAU 1
in Universal Polymer Material Model, 346
WWTAU 2
in Universal Polymer Material Model, 346

XZ POINT
in Coordinate System, 77
XZ POINT NODESET
in Coordinate System, 77

YIELD CURVES
in Plane Stress Rate Plasticity material model, 336
YIELD FUNCTION
in Wire Mesh material model, 327
YIELD STRAIN RATES
in Plane Stress Rate Plasticity material model, 336
YIELD STRESS

in Ductile Fracture material model, 237
in Elastic-Plastic material model, 230
in Elastic-Plastic Power-Law Hardening material model, 234
in Johnson–Cook material model, 286
in Multilinear Elastic-Plastic Hardening Model material model, 240
in Multilinear Elastic-Plastic Hardening Model with Failure material model, 244
in Orthotropic Crush material model, 329
in Orthotropic Rate material model, 333
in Plane Stress Rate Plasticity material model, 336
in Thermoelastic Plastic material model, 250
YIELD STRESS FUNCTION
in Multilinear Elastic-Plastic Hardening Model material model, 240
in Multilinear Elastic-Plastic Hardening Model with Failure material model, 244
in Thermoelastic Plastic material model, 250
YOUNGS MODULUS
in Barlat Plasticity material model, 274
in BCJ material model, 289
in BCJ_MEM material model, 292
in Ductile Fracture material model, 237
in Elastic 3D Orthotropic material model, 323
in Elastic Fracture material model, 228
in Elastic material model, 223
in Elastic-Plastic material model, 230
in Elastic-Plastic Power-Law Hardening material model, 234
in Foam Plasticity material model, 315
in Hill Plasticity material model, 263
in Hosford Plasticity material model, 254
in Hyperfoam material model, 321
in Incompressible Solid material model, 343
in Johnson–Cook material model, 286
in Low Density Foam material model, 319
in MD Viscoplastic material model, 308
in Multilinear Elastic-Plastic Hardening Model material model, 240
in Multilinear Elastic-Plastic Hardening Model with Failure material model, 244
in Neo-Hookean material model, 226
in Orthotropic Crush material model, 329
in Orthotropic Rate material model, 333
in Plane Stress Rate Plasticity material model, 336
in Power Law Creep material model, 301
in Soil and Crushable Foam material model, 312
in Stiff Elastic material model, 355
in Swanson material model, 357
in Thermoelastic material model, 225
in Thermoelastic Plastic material model, 250
in Viscoelastic Swanson material model, 360
in Viscoplastic material model, 303
in Wire Mesh material model, 327
YOUNGS MODULUS AA
in Elastic 3D Orthotropic material model, 323
YOUNGS MODULUS BB
in Elastic 3D Orthotropic material model, 323
YOUNGS MODULUS CC
in Elastic 3D Orthotropic material model, 323
YOUNGS MODULUS FUNCTION
in BCJ material model, 289
in BCJ_MEM material model, 292
in Multilinear Elastic-Plastic Hardening Model material model, 240
in Multilinear Elastic-Plastic Hardening Model with Failure material model, 244
in Thermoelastic material model, 225
in Thermoelastic Plastic material model, 250

Z POINT
in Coordinate System, 77
Z POINT NODESET
in Coordinate System, 77
ZERO LENGTH R AXIS
in User Mount Section, 453
ZOLTAN DEBUG LEVEL
in Zoltan Parameters, 498
as default, 498
ZOLTAN PARAMETERS
as command block, 498
with defaults, 498
as command line in Rebalance, 495
description of, 498

DISTRIBUTION

Email— [redacted]

Name	Org.	Sandia Email Address
Technical Library	1911	sanddocs@sandia.gov



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.