SANDIA REPORT

SAND2016-9978
Unlimited Release
Printed September, 2016

ROMULIS 1.01.00

Robert M. Lacayo, Matthew R. W. Brake

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

Sandia National Laboratories is a multi-mission laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

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 Rd
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: http://www.ntis.gov/search

SAND2016-9978
Unlimited Release
Printed September, 2016

ROMULIS 1.01.00

Robert M. Lacayo, Matthew R.W. Brake
Component Science and Mechanics
Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-0346

Abstract

The Reduced Order Modeling Unlimited Localized Interface Simulator (ROMULIS) is a set of toolbox scripts in MATLAB designed to perform nonlinear transient integration on a system of reduced order structural models that interact with each other at localized interfaces. ROMULIS is meant to provide a user-friendly interface for applying the latest developments in numerical techniques and modeling in structural dynamics analysis while also giving the freedom to implement new technologies from forthcoming research. This report documents how to use and interpret the toolbox scripts. The theory behind the code is given, followed by a manual for interacting with the scripts to perform simulations. Lastly, a high-level introduction that explains how the scripts interact with each other is given for aspiring developers.

CONTENTS

FIGURES

TABLES

1. INTRODUCTION

The Reduced Order Modeling Unlimited Localized Interface Simulator (ROMULIS) code is a set of MATLAB scripts designed to facilitate the modeling of dynamic subcomponents that interface with one another. The goal of ROMULIS is to model a nonlinear system that includes contact – whether from a frictional interface, impact event, or other localized phenomena – efficiently. Through the development of ROMULIS, much freedom has been given to the user to specify arbitrary contact models (some of which are built into ROMULIS). Additionally, the development of ROMULIS also allows for the use of novel and highly efficient nonlinear solvers.

The development of ROMULIS was preceded by a tool for modeling jointed structures using discontinuous basis functions [1]. This original tool took models developed in SIERRA [2] for each subcomponent, assembled them into a system connected by linear springs, calculated the "joint modes" of the system (which are represented by discontinuous basis functions), then used the joint modes to augment the set of linear basis functions in order to improve convergence. In the original tools, joints were then modeled using the four-parameter Iwan model [3]. This tool, unfortunately, was neither accessible nor easily modifiable to consider different structures. Thus, ROMULIS was developed as a sequel to this tool to make it possible to specify new geometries, contact models, boundary conditions, excitations, etc. Subsequently, ROMULIS grew out of several concurrent efforts at Sandia National Laboratories to develop a new class of integration techniques [4], improved normal contact models for point-to-point modeling [5], and more efficient algorithms for Iwan modeling [6].

While ROMULIS continues to be a work in progress, the ultimate goal is that it will enable the complete modeling of a nonlinear system composed of multiple subcomponents. The intention of ROMULIS is to assemble Craig-Bampton reduced order models [7, 8] that are exported from a finite element package, and to assign nonlinear constitutive models governing how the Craig-Bampton models interact at each interface (eventually via a graphical interface). The current capability of ROMULIS is focused on transient analyses, though future developments will focus on incorporating nonlinear quasi-static solvers for model calibration, frequency domain solvers for steady-state solutions, and data analysis techniques to post-process the results from both ROMULIS and experimental measurements.

This report documents ROMULIS version 1.01.00, compatible with SIERRA/SD release 4.38 and later, and is comprised of three major sections. Section 2 gives the theoretical background for ROMULIS, and it includes the formulation of Craig-Bampton reduced order models, the assembly of the system, and definitions for how ROMULIS applies internal forces due to nonlinear constitutive models. The user manual is provided in Section 3, which covers only those toolbox scripts in which the user needs to manipulate in order to run a simulation. Lastly, Section 4 gives an introduction to ROMULIS's major scripts to provide a developer with high-level knowledge of how the scripts interact with one another.

Following these sections are appendices that inform the user of ROMULIS's current library of implemented load functions and nonlinear constitutive models. The appendices are useful to help define loads and constitutive models on the assembled system in ROMULIS.

2. THEORY

2.1. Reduced-Order Systems and Craig-Bampton Reduction

ROMULIS includes the capability to solve transformed systems, whereby the physical degrees of freedom (DOF) are replaced and represented by a set of generalized coordinates. A common coordinate transformation technique used in structural dynamics involves model order-reduction, where a large number of DOF are governed by a smaller set of coordinates. For models with a large number of DOF, it is highly recommended that the full system be reduced before passing it into the ROMULIS. Using reduced-order models offers a number of benefits such as decreased computation time for the numerical solver, reduced memory usage, and well-sustained model fidelity depending on the method of reduction. Almost any linearly-transformed system will work with ROMULIS provided that the user can keep track of loads and initial conditions in terms of the generalized coordinates.

To assist with the bookkeeping, ROMULIS is coded to accept Craig-Bampton reduced systems [7,8] . Systems that are reduced using the Craig-Bampton method allow for the specification of a set of physical DOF whose coordinates remain the same after the transformation. In this way, the same values used for the internal loads and prescribed boundary conditions in the physical system are also valid in the generalized coordinates with minimal effort in projecting those values via coordinate transformation.

The Craig-Bampton reduction method belongs to a class of reduced-order modeling techniques known as component mode synthesis (CMS) [9], which transform the system using an assumed set of basis vectors that adequately describe possible spatial configurations of the physical DOF, similar to linear Eigen modes. Traditionally, CMS is used to reduce individual complex substructures that can then be coupled to each other at their interfaces using displacement compatibility equations. ROMULIS, however, uses a more generalized approach to coupling in that the connecting interfaces may be assigned a discrete, constitutive force model in addition to a constraint. With the Craig-Bampton method, the coupling of interface coordinates is more straightforward than other CMS techniques because the transformed interface coordinates are equivalent to those governing the original physical DOF on that interface before the transformation.

An overview of the theory in creating a Craig-Bampton reduced model is given as follows based on [9]. The starting assumption is that the dynamic response of a particular substructure can be represented linearly using a set of equations of the form

$$\mathbf{M}\ddot{\mathbf{u}} + \mathbf{C}\dot{\mathbf{u}} + \mathbf{K}\mathbf{u} = \mathbf{f}, \tag{1}$$

where **M**, **C**, and **K** represent the linear mass, damping, and stiffness matrices, respectively, as determined from finite element analysis. The vectors $\mathbf{u}$, $\dot{\mathbf{u}}$, and $\ddot{\mathbf{u}}$ are the displacement, velocity, and acceleration vectors, respectively, each containing *N* number of DOF, and **f** is the *N*-dimensional vector of applied loads.

The Craig-Bampton method begins by partitioning system into internal DOF and boundary DOF (denoted by subscripts *i* and *b,* respectively) as follows,

$$\begin{bmatrix} \mathbf{M}_{ii} & \mathbf{M}_{ib} \\ \mathbf{M}_{bi} & \mathbf{M}_{bb} \end{bmatrix} \begin{Bmatrix} \ddot{\mathbf{u}}_i \\ \ddot{\mathbf{u}}_b \end{Bmatrix} + \begin{bmatrix} \mathbf{C}_{ii} & \mathbf{C}_{ib} \\ \mathbf{C}_{bi} & \mathbf{C}_{bb} \end{bmatrix} \begin{Bmatrix} \dot{\mathbf{u}}_i \\ \dot{\mathbf{u}}_b \end{Bmatrix} + \begin{bmatrix} \mathbf{K}_{ii} & \mathbf{K}_{ib} \\ \mathbf{K}_{bi} & \mathbf{K}_{bb} \end{bmatrix} \begin{Bmatrix} \mathbf{u}_i \\ \mathbf{u}_b \end{Bmatrix} = \begin{Bmatrix} \mathbf{f}_i \\ \mathbf{f}_b \end{Bmatrix}. \tag{2}$$

In the context of the simulator, a boundary DOF refers to any DOF on which a boundary condition, an external force, or a coupling force is applied. All other DOF are assigned as internal DOF. Using these sub-partitions of the system matrices,

$$\boldsymbol{\psi}_{ib} = -\mathbf{K}_{ii}\mathbf{K}_{ib} \tag{3}$$

is defined as the constraint mode matrix. The matrix of fixed-interface mode shapes consisting of $N_k < N$ number of retained modes from the set of all possible mode shapes $\boldsymbol{\varphi}_j$ is $\boldsymbol{\Phi}_{ik}$, and is defined by satisfying the Eigenvalue-Eigenvector problem

$$\left(\mathbf{K}_{ii} - \lambda_j \mathbf{M}_{ii}\right)\boldsymbol{\varphi}_j = \mathbf{0}, \tag{4}$$

$$\boldsymbol{\Phi}_{ik} = \begin{bmatrix} \boldsymbol{\varphi}_1 & \boldsymbol{\varphi}_2 & \cdots & \boldsymbol{\varphi}_{N_k} \end{bmatrix}. \tag{5}$$

The Craig-Bampton coordinate transformation is then

$$\begin{Bmatrix} \mathbf{u}_i \\ \mathbf{u}_b \end{Bmatrix} = \boldsymbol{\Psi}_{CB} \begin{Bmatrix} \mathbf{q}_k \\ \mathbf{q}_b \end{Bmatrix} = \boldsymbol{\Psi}_{CB}\mathbf{q}, \qquad \boldsymbol{\Psi}_{CB} = \begin{bmatrix} \boldsymbol{\Phi}_{ik} & \boldsymbol{\psi}_{ib} \\ \mathbf{0}_{bk} & \mathbf{I}_{bb} \end{bmatrix}, \tag{6}$$

There are two features of the coordinate transformation worth noting. First, reduction is achieved by removing the "higher frequency" fixed interface modes from $\boldsymbol{\Phi}_{ik}$. Second, the $\mathbf{u}_b$ and $\mathbf{q}_b$ vectors, which denote the boundary DOF and constraint mode coordinates, are equivalent. The difference physically is that when a particular coordinate in $\mathbf{q}_b$ is displaced, the corresponding boundary DOF in $\mathbf{u}_b$ displaces by the same amount, and, at the same time, all nearby physical DOF displace statically in "response" to that boundary DOF. Due to the one-to-one equivalence of the $\mathbf{q}_b$ and $\mathbf{u}_b$ coordinates, the analyst can treat $\mathbf{q}_b$ as $\mathbf{u}_b$ to a certain extent. There is a limitation to this assumption when defining initial conditions, which is explored at the end of Section 2.2.

Substituting Eq. (6), the system in Eq. (1) transforms to

$$\hat{\mathbf{M}}\ddot{\mathbf{q}} + \hat{\mathbf{C}}\dot{\mathbf{q}} + \hat{\mathbf{K}}\mathbf{q} = \hat{\mathbf{f}}, \tag{7}$$

where $\hat{\mathbf{M}} = \boldsymbol{\Psi}_{CB}^{\mathrm{T}}\mathbf{M}\boldsymbol{\Psi}_{CB}$, $\hat{\mathbf{C}} = \boldsymbol{\Psi}_{CB}^{\mathrm{T}}\mathbf{C}\boldsymbol{\Psi}_{CB}$, $\mathbf{K} = \boldsymbol{\Psi}_{CB}^{\mathrm{T}}\mathbf{K}\boldsymbol{\Psi}_{CB}$, and $\hat{\mathbf{f}} = \boldsymbol{\Psi}_{CB}^{\mathrm{T}}\mathbf{f}$. Superscript T denotes the transpose operator.

ROMULIS itself does not have the capability to create Craig-Bampton reduced models. This function is outsourced to Sierra/SD, a massively-parallel finite element solver for structural dynamics, which generates the original system matrices from a mesh and transforms them using a Craig-Bampton reduction algorithm [10]. The reader is directed to Section 3.1.1 for

instructions on exporting the Craig-Bampton reduced system from Sierra/SD and importing them into ROMULIS. Otherwise the user can upload other reduced-order systems or even full-order system matrices into ROMULIS provided that the user's machine has ample memory for MATLAB to store matrices. In this case, the user will need to keep track of coordinate transformations when applying loads. Instructions on how to create a system and upload it into ROMULIS are given in Section 3.1.2.

2.2. System Assembly

Whether a physical system or a transformed system is uploaded, ROMULIS first assembles the system matrices from individual substructures into one global system for input into the numerical solver. The theory presented in this section follows commonly used matrix methods in finite element analysis for assembling the global system matrices and imposing load conditions [11].

Consider a global system that contains N_s number of substructures, the elements of which are denoted by superscript (s) for s = 1, 2, ⋯, N_s. Each substructure has a vector of generalized coordinate displacements $\mathbf{q}^{(s)}$ associated with a given mass matrix $\hat{\mathbf{M}}^{(s)}$, damping matrix $\hat{\mathbf{C}}^{(s)}$, stiffness matrix $\hat{\mathbf{K}}^{(s)}$, and force vector $\hat{\mathbf{f}}^{(s)}$. Since the coordinates are generalized, they can represent a physical system or a transformed system. Allowing matrix **D** to be substituted for **M**, **C**, or **K**, the global system matrices are then defined by arranging the substructure system matrices in block diagonals,

$$\tilde{\mathbf{D}} = \begin{bmatrix} \hat{\mathbf{D}}^{(1)} & & & \\ & \hat{\mathbf{D}}^{(2)} & & \\ & & \ddots & \\ & & & \hat{\mathbf{D}}^{(N_s)} \end{bmatrix}, \tag{8}$$

and arranging the coordinate vectors vertically,

$$\mathbf{p} = \begin{Bmatrix} \mathbf{q}^{(1)} \\ \mathbf{q}^{(2)} \\ \vdots \\ \mathbf{q}^{(N_s)} \end{Bmatrix}, \quad \tilde{\mathbf{f}} = \begin{Bmatrix} \hat{\mathbf{f}}^{(1)} \\ \hat{\mathbf{f}}^{(2)} \\ \vdots \\ \hat{\mathbf{f}}^{(N_s)} \end{Bmatrix}, \tag{9}$$

such that the following global equations of motion are formed

$$\tilde{\mathbf{M}}\ddot{\mathbf{p}} + \tilde{\mathbf{C}}\dot{\mathbf{p}} + \tilde{\mathbf{K}}\mathbf{p} = \tilde{\mathbf{f}}. \tag{10}$$

To prescribe boundary conditions, the coordinates are partitioned into active coordinates and constrained coordinates (denoted by subscripts *a* and *c,* respectively)

$$\begin{bmatrix} \tilde{\mathbf{M}}_{aa} & \tilde{\mathbf{M}}_{ac} \\ \tilde{\mathbf{M}}_{ca} & \tilde{\mathbf{M}}_{cc} \end{bmatrix} \begin{Bmatrix} \ddot{\mathbf{p}}_a \\ \ddot{\mathbf{p}}_c \end{Bmatrix} + \begin{bmatrix} \tilde{\mathbf{C}}_{aa} & \tilde{\mathbf{C}}_{ac} \\ \tilde{\mathbf{C}}_{ca} & \tilde{\mathbf{C}}_{cc} \end{bmatrix} \begin{Bmatrix} \dot{\mathbf{p}}_a \\ \dot{\mathbf{p}}_c \end{Bmatrix} + \begin{bmatrix} \tilde{\mathbf{K}}_{aa} & \tilde{\mathbf{K}}_{ac} \\ \tilde{\mathbf{K}}_{ca} & \tilde{\mathbf{K}}_{cc} \end{bmatrix} \begin{Bmatrix} \mathbf{p}_a \\ \mathbf{p}_c \end{Bmatrix} = \begin{Bmatrix} \tilde{\mathbf{f}}_a \\ \tilde{\mathbf{f}}_c \end{Bmatrix}, \tag{11}$$

where the coordinates $\mathbf{p}_c$ are the known prescribed displacements and the coordinates $\dot{\mathbf{p}}_c$ and $\ddot{\mathbf{p}}_c$ are their first and second time derivatives, respectively. In a Craig-Bampton reduced system, the constrained coordinates are the subset of the boundary coordinates defined in Eq. (2). The rest of the coordinates are assigned as active coordinates $\mathbf{p}_a$ because they are the unknowns for which the numerical integrator solves. The force vector $\tilde{\mathbf{f}}_c$ is also unknown, but not essential since only the top row of Eq. (11) is sufficient to determine the active coordinates. Rearranging the top row of Eq. (11) so that the known values lie on the right-hand side of the equation gives

$$\tilde{\mathbf{M}}_{aa}\ddot{\mathbf{p}}_a + \tilde{\mathbf{C}}_{aa}\dot{\mathbf{p}}_a + \tilde{\mathbf{K}}_{aa}\mathbf{p}_a = \tilde{\mathbf{f}}_a - \tilde{\mathbf{M}}_{ac}\ddot{\mathbf{p}}_c - \tilde{\mathbf{C}}_{ac}\dot{\mathbf{p}}_c - \tilde{\mathbf{K}}_{ac}\mathbf{p}_c . \tag{12}$$

Equation (12) is the final form of the system equations of motion that ROMULIS processes through an adaptive, fifth-order, implicit-explicit integrator [4].

In a Craig-Bampton reduced system, the equivalence of the constraint mode coordinates $\mathbf{q}_b{}^{(s)}$ and the boundary DOF $\mathbf{u}_b{}^{(s)}$ in a substructure can mislead the analyst into incorrectly defining the initial conditions due to prescribed displacements. In general, it cannot be assumed that the active coordinates, $\mathbf{p}_a$ and its derivatives, are independent of any prescribed motion applied on the boundary DOF because the motion of the boundary DOF do influence the fixed-interface modes, $\mathbf{q}_k{}^{(s)}$. This can be seen from taking a pseudo-inverse of the transformation in Eq. (6),

$$\mathbf{q}^{(s)} = \begin{Bmatrix} \mathbf{q}_k{}^{(s)} \\ \mathbf{q}_b{}^{(s)} \end{Bmatrix} = \boldsymbol{\Psi}_{CB}^{(s)\,-1} \begin{Bmatrix} \mathbf{u}_i{}^{(s)} \\ \mathbf{u}_b{}^{(s)} \end{Bmatrix} = \begin{bmatrix} \mathbf{G}_i{}^{(s)} & \mathbf{G}_b{}^{(s)} \end{bmatrix} \begin{Bmatrix} \mathbf{u}_i{}^{(s)} \\ \mathbf{u}_b{}^{(s)} \end{Bmatrix}. \tag{13}$$

Even after considering that motion is prescribed on only a subset of $\mathbf{u}_b{}^{(s)}$, it is not guaranteed that the top set of rows in matrix $\mathbf{G}_b{}^{(s)}$ are zeros so as to make $\mathbf{q}_k{}^{(s)}$ independent of $\mathbf{u}_b{}^{(s)}$. Therefore, when prescribing motion on $\mathbf{u}_b{}^{(s)}$, an initial condition must also be placed on the system using the transformation in Eq. (13) to get the correct starting state of $\mathbf{q}^{(s)}$ and $\dot{\mathbf{q}}^{(s)}$ (and $\mathbf{p}_a$ and $\dot{\mathbf{p}}_a$ by extension). ROMULIS calculates this initial condition automatically for Craig-Bampton reduced systems.

2.3. The Nonlinear Force Vector

Local nonlinearities are introduced into the system by coupling the boundary DOF between substructures with discrete constitutive models. In Eq. (12), the constitutive models manifest as a vector of nonlinear forces $\mathbf{f}_{NL}$ (expressed in physical coordinates) that, in addition to the vector

of time-varying loads $\mathbf{f}_{ext}$ (also in physical coordinates), comprise the vector of generalized forces applied on the active coordinates,

$$\tilde{\mathbf{f}}_a = \tilde{\boldsymbol{\Psi}}^{\mathrm{T}} \left(\mathbf{f}_{ext} + \mathbf{f}_{NL} \left(\tilde{\mathbf{u}}, \dot{\tilde{\mathbf{u}}} \right) \right), \tag{14}$$

where $\tilde{\boldsymbol{\Psi}}$ is a global transformation matrix formed by arranging the substructure transformation matrices (for example, $\boldsymbol{\Psi}_{CB}$ from Eq. (6)) in block diagonal form. The vectors $\mathbf{u}$ and $\dot{\mathbf{u}}$ are the global displacement and velocity vectors for the physical coordinates formed by arranging the substructure physical coordinates vertically similar to Eq. (9).

In Craig-Bampton reduced systems, if the vector of nonlinear forces is applied on the boundary DOF only, then

$$\tilde{\boldsymbol{\Psi}}^{\mathrm{T}} \mathbf{f}_{NL} \left(\tilde{\mathbf{u}}, \dot{\tilde{\mathbf{u}}} \right) = \tilde{\boldsymbol{\Psi}}^{\mathrm{T}} \begin{Bmatrix} \mathbf{0}_i^{(1)} \\ \mathbf{f}_{NLb}^{(1)} \left(\tilde{\mathbf{u}}, \dot{\tilde{\mathbf{u}}} \right) \\ \mathbf{0}_i^{(2)} \\ \mathbf{f}_{NLb}^{(2)} \left(\tilde{\mathbf{u}}, \dot{\tilde{\mathbf{u}}} \right) \\ \vdots \\ \mathbf{0}_i^{(N_s)} \\ \mathbf{f}_{NLb}^{(N_s)} \left(\tilde{\mathbf{u}}, \dot{\tilde{\mathbf{u}}} \right) \end{Bmatrix} = \begin{Bmatrix} \mathbf{0}_k^{(1)} \\ \mathbf{f}_{NLb}^{(1)} \left(\tilde{\mathbf{u}}, \dot{\tilde{\mathbf{u}}} \right) \\ \mathbf{0}_k^{(2)} \\ \mathbf{f}_{NLb}^{(2)} \left(\tilde{\mathbf{u}}, \dot{\tilde{\mathbf{u}}} \right) \\ \vdots \\ \mathbf{0}_k^{(N_s)} \\ \mathbf{f}_{NLb}^{(N_s)} \left(\tilde{\mathbf{u}}, \dot{\tilde{\mathbf{u}}} \right) \end{Bmatrix}, \tag{15}$$

which shows that the nonlinear force in Craig-Bampton coordinates can be expressed directly in terms of the physical nonlinear forces. Furthermore, since the nonlinear forces also use the boundary DOF as the input states, then the equivalence of the $\mathbf{u}_b^{(s)}$ and $\mathbf{q}_b^{(s)}$ vectors means that $\mathbf{q}_b^{(s)}$ and $\dot{\mathbf{q}}_b^{(s)}$, which are subsets of $\mathbf{p}$ and $\dot{\mathbf{p}}$, can be used directly as inputs for the nonlinear force functions. ROMULIS takes advantage of this two-fold, direct formulation for the nonlinear force in Craig-Bampton systems to reduce the computational cost.

Each nonlinear constitutive model used to construct the $\mathbf{f}_{NL}$ vector in ROMULIS is a scalar force function. The function calculates the relative motion between two interfacing nodes in a finite element mesh, and returns a force value that must be applied equally and oppositely on the two nodes. Figure 1 illustrates a basic model for ensuring the correct signage for the values entered into $\mathbf{f}_{NL}$.

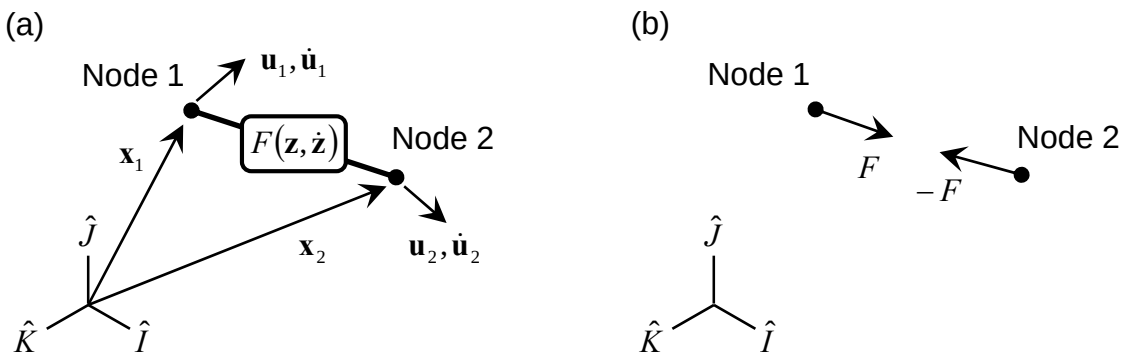

Figure 1: Schematic of (a) a nonlinear force model between two nodes with state vectors and (b) the resulting nonlinear force vectors applied on the nodes.

Figure 1(a) shows a force function between two nodes that have translational degrees of freedom in the three spatial directions. The two nodes have nominal position vectors, $\mathbf{x}_1$ and $\mathbf{x}_2$, relative to a point of origin on a global coordinate axis. The two nodes undergo elastic motion about these nominal positions, denoted by vectors $\mathbf{u}_1$ and $\mathbf{u}_2$ for displacement and by vectors $\dot{\mathbf{u}}_1$ and $\dot{\mathbf{u}}_2$ for velocity. ROMULIS defines the position of node 2 relative to node 1 as

$$\mathbf{z} = \mathbf{x}_2 + \mathbf{u}_2 - \mathbf{x}_1 - \mathbf{u}_1, \tag{16}$$

and the relative velocity between the two nodes is

$$\dot{\mathbf{z}} = \dot{\mathbf{u}}_2 - \dot{\mathbf{u}}_1. \tag{17}$$

Both $\mathbf{z}$ and $\dot{\mathbf{z}}$ are used as the state variable inputs for a scalar force function, $F(\mathbf{z}, \dot{\mathbf{z}})$, which returns a value that fills two entries in $\mathbf{f}_{NL}$ representing each node. With the relative motion defined in Eqs. (16) and (17), the entries associated with node 1 are given the exact value from F, and the entries associated with node 2 are given the oppositely signed value. This nodal sign convention applies for all nonlinear functions oriented in each of the three translational and rotational directions.

It is often the case that a nonlinear force function may not align with any of the three global coordinate directions, and thus would require a rotated frame of reference. In ROMULIS, the user must define two vectors, $\mathbf{r}_1$ and $\mathbf{r}_2$, in terms of the global coordinate basis, $\hat{I}$, $\hat{J}$, and $\hat{K}$, to help form the rotated basis, $\hat{i}$, $\hat{j}$, and $\hat{k}$. The vector $\mathbf{r}_1$ typically locates the nominal position of node 2 relative to node 1 (as in $\mathbf{r}_1 = \mathbf{x}_2 - \mathbf{x}_1$), but for some nonlinear force models it is more intuitive to define it as the outward normal vector for a surface at node 1. The direction of $\mathbf{r}_1$ becomes the new first coordinate direction $\hat{i}$ in the rotated frame. The vector $\mathbf{r}_2$ can be any vector perpendicular to $\mathbf{r}_1$ (often a surface-tangent direction), and it forms the second coordinate direction $\hat{j}$ in the rotated frame. The two input vectors, $\mathbf{r}_1$ and $\mathbf{r}_2$, are used to create a rotation transformation matrix as follows.

It is assumed that $\mathbf{r}_1$ and $\mathbf{r}_2$ are column vectors. Given that $\mathbf{r}_1$ points in the same direction as $\hat{i}$, it is normalized to have unit length to form the first basis vector,

$$\hat{\mathbf{e}}_1 = \frac{\mathbf{r}_1}{\|\mathbf{r}_1\|}. \tag{18}$$

Although the input for $\mathbf{r}_2$ should be orthogonal to $\mathbf{r}_1$, the simulator accounts for error by orthonormalizing $\mathbf{r}_2$ using the modified Gram-Schmidt process [9]. First, the vector projection of $\mathbf{r}_2$ onto $\hat{\mathbf{e}}_1$ is subtracted from $\mathbf{r}_2$,

$$\mathbf{n}_2 = \mathbf{r}_2 - (\mathbf{r}_2 \cdot \hat{\mathbf{e}}_1)\hat{\mathbf{e}}_1. \tag{19}$$

after which, the second basis vector is formed by normalizing $\mathbf{n}_2$,

$$\hat{\mathbf{e}}_2 = \frac{\mathbf{n}_2}{\|\mathbf{n}_2\|}. \tag{20}$$

The cross product of $\mathbf{e}_1 \times \mathbf{e}_2$ creates the third basis vector $\mathbf{e}_3$. The transformation from the rotated frame to the global frame is then defined as

$$\begin{Bmatrix} \hat{I} \\ \hat{J} \\ \hat{K} \end{Bmatrix} = \mathbf{R} \begin{Bmatrix} \hat{i} \\ \hat{j} \\ \hat{k} \end{Bmatrix}, \tag{21}$$

where $\mathbf{R} = \begin{bmatrix} \hat{\mathbf{e}}_1 & \hat{\mathbf{e}}_2 & \hat{\mathbf{e}}_3 \end{bmatrix}$ is the rotation matrix. This rotation is used to orient the global frame state vectors from Eqs. (16) and (17) to the rotated frame,

$$\mathbf{z}_r = \mathbf{R}^{\mathrm{T}} \mathbf{z}, \qquad \dot{\mathbf{z}}_r = \mathbf{R}^{\mathrm{T}} \dot{\mathbf{z}}. \tag{22}$$

The rotated state vectors, $\mathbf{z}_r$ and $\dot{\mathbf{z}}_r$, then become the inputs for the nonlinear force functions, F_i, F_j, and F_k, in the rotated frame. The outputs from these functions are then rotated back to the global frame with

$$\mathbf{f}_g = \mathbf{R} \begin{Bmatrix} F_i(\mathbf{z}_r, \dot{\mathbf{z}}_r) \\ F_j(\mathbf{z}_r, \dot{\mathbf{z}}_r) \\ F_k(\mathbf{z}_r, \dot{\mathbf{z}}_r) \end{Bmatrix}. \tag{23}$$

The values of the three components in $\mathbf{f}_g$ fill two entries each (six total) in the nonlinear force vector, $\mathbf{f}_{NL}$, whereby the coordinates associated with node 1 receive the exact force value, and those associated with node 2 are given the oppositely signed force value.

3. USER'S MANUAL

In setting up and simulating a system, ROMULIS focuses on the creation of three different MATLAB data files: a body file, a system file, and a response file. A body file contains information about a single, linear substructure, specifically its linear system matrices and nodeset information. A system file gathers the information from one or more body files to form a collection of substructures in a system. The system file also includes information about the interactions between the substructures, the loads applied, and the type of solution sought from the system. After the system is put through the solver, the solution is stored in the response file. In ROMULIS, these three data files are created by, and subsequently supplied to, a sequence of MATLAB scripts included the ROMULIS toolbox. A high-level schematic of this script sequence is shown in Fig. 2.

Performing a simulation with ROMULIS begins with the formation of at least one body file, the procedure for which is covered in Section 3.1. After one or more body files are created, they are passed into the *SystemSetup.m* script in the ROMULIS toolbox. In *SystemSetup*, the user specifies which body files to use in the system as well as the loads, interface models, and other conditions to apply on the system. The procedure for scripting *SystemSetup* is given in Section 3.2. The output for *SystemSetup* is the system file, which is subsequently passed into the *SerialExecuter.m* script. As is discussed in Section 3.4, the user then specifies the numerical integrator parameters in *SerialExecuter*, and finally runs the solver. The output for *SerialExecuter* is the response file.

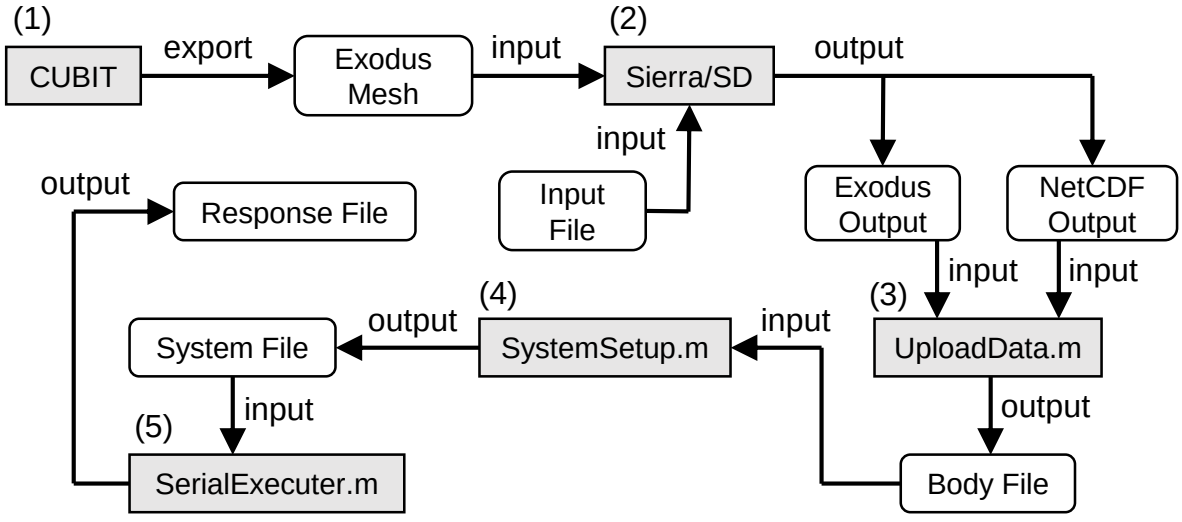

Figure 2: Flow chart of the programs used and the files accessed by the user in ROMULIS. The shaded boxes indicate software packages (CUBIT and Sierra/SD) or ROMULIS toolbox scripts (indicated by .m), and the white boxes are data files. All programs and files are utilized in sequence as numbered in parenthesis.

3.1. Creating a Body File

A body file can be created using one of two methods. In the first method, the user creates the substructure geometry using a mesh generator software package, and exports the mesh to Sierra/SD [10] to be reduced. The toolbox script *UploadData.m* then reads the output files from Sierra/SD, and automatically generates a body file from the output. Alternatively, the user can create a body file manually in MATLAB by specifying the required variables appropriately. Section 3.1.1 gives the procedure for importing data from Sierra/SD, and the procedure for creating one's own body file is given in Section 3.1.2.

3.1.1. Importing a Sierra/SD Finite Element Model

ROMULIS itself does not create reduced-order systems directly from a finite element mesh, but instead outsources this capability through Sierra/SD's Craig-Bampton reduction algorithm. Sierra/SD exports the reduced system in a data file, which the simulator uses as the basis for the body file. An overview of Craig-Bampton reduction theory is given in Section 2.1.

Creating a body file for a substructure that is reduced in Sierra/SD first requires the user to create the mesh geometry in a finite element modeling software package. The user is given free rein to create the geometry and mesh as desired provided that the final mesh is exported as a NetCDF Exodus file (extension *.e, *.exo) so that it is compatible with the solvers in Sierra/SD. The geometry and meshing software, CUBIT (also known as Trelis), is able to export meshes in Exodus format.

In creating the mesh, the following guidelines should be followed in anticipation for writing the relevant sections in the Sierra/SD input file, and for setting up the system in the *SystemSetup.m* script:

1) Try to minimize the number of elements in the finite element model. Although the simulator uses the reduced system for integration, it still relies on the Craig-Bampton transformation matrix to produce the correct initial state for prescribed boundary conditions. ROMULIS must store at least two instances of the entire transformation matrix in memory, so the size of the matrix is limited by the memory capacity of the user's machine. A good approximation of the memory requirement for a finite element model can be calculated as

$$\begin{aligned}\text{Memory} = 2 &\times (\#\text{ of nodes}) \times (6\text{ DOF per node}) \\ &\times (\#\text{ of modes in transformation}) \\ &\times (8\text{ bytes per matrix entry})\end{aligned}$$

   For instance, a model with 200,000 nodes that is reduced using 200 combined fixed-interface and constraint modes requires 3.84 Gb of available memory for ROMULIS to store the transformation matrix.

2) Assign all nodes of interest into nodesets. Nodes of interest include any node on which is applied an external load, a nonlinear force model, a prescribed boundary condition, or is

otherwise a response output node. In *SystemSetup*, the user prescribes loading conditions on nodesets, so every node belonging to a nodeset receives the same loading condition.

3) Nonlinear force models in ROMULIS are assigned to act between two nodesets, the master set and the slave set. In the current version of ROMULIS, there is no practical difference between the master and slave set, and the user can specify either nodeset as the master set in *SystemSetup*. There are, however, strict rules on nodal definitions in each set.
    a. Nodes that are specified as interface nodes must have either all three translational DOF, or have all three translational and all three rotational DOF.
    b. If one set contains multiple nodes, then the other set must contain either one node or the same number of nodes. In the case where both nodesets contain an equal number of nodes, each node in one set must have a coincident counterpart in the other set. The user is not required to order the nodes in one nodeset to align with the order of the coincident counterpart nodes in the other nodeset; ROMULIS has an internal algorithm that pairs coincident nodes and their degrees of freedom. A new instance of the same nonlinear force model is applied between each coincident node pair, so the user should beware of how many nonlinear models the system will contain and how that will affect the convergence of the integrator.
    c. If each nodeset contains one node, then the coincidence restriction is lifted. The two nodes can exist in different locations in space.

4) The solid elements (e.g. hexahedral and tetrahedral elements) in Sierra/SD do not have rotational degrees of freedom. The user should keep in mind how these undefined degrees of freedom may affect system definitions and adjust accordingly.

After exporting the mesh, the next step is to create the reduced structure from the mesh by running a Sierra/SD input file. The user can reference the necessary sections in the Sierra/SD user manual [10] to define the input file, but the following keywords shown in Table 1 must be included to output the reduced system.

When deciding how many fixed-interface modes to keep, a good rule to follow is to keep all modes associated with natural frequencies up to twice the value of the maximum frequency value of interest. This rule derives from the Nyquist criterion, which states that a time signal of desired fidelity can be constructed sufficiently using signals having a frequency corresponding to half the sampling period of that time signal. Current experimental measurement techniques produce valid response data with natural frequencies up to 4000 Hz, so it is recommended that the user keep modes up to a natural frequency 8000 Hz.

Table 1: The Sierra/SD input file keywords required for a Craig-Bampton reduction solution.

Keyword	Description

SOLUTION
  cbr
  nmodes = 21
END
	Invokes the Craig-Bampton reduction algorithm, and calculates the first 21 (example) fixed-interface modes.

OUTPUTS displacement END	Returns the fixed-interface and constraint mode shape vectors in the Exodus output file.

CBMODEL
  nodeset = '1,5,20:28'
  format = 'netcdf'
  file = 'ReducSys.ncf'
END
	Specifies the nodesets containing the boundary nodes, and returns the reduced system in the as-named NetCDF file, *ReducSys.ncf*.

HISTORY //Optional
  nodeset = '3,4'
  displacement
  //do not specify
  //coordinate keywords.
END
	OPTIONAL: Returns the rows in the transformation matrix associated with the nodes in nodesets 3 and 4 (example). Used for specifying force input and response output nodes.

If the input file is run successfully, Sierra/SD returns two output files: an Exodus output file (extension **-out.exo* by default), and a NetCDF file (extension **.ncf*). The Exodus output file contains the fixed-interface mode shapes and the constraint mode shapes that the simulator uses to form the Craig-Bampton transformation matrix. The reduced mass and stiffness matrices are stored in the NetCDF file. If the HISTORY keyword is specified in the Sierra/SD input file, then Sierra/SD additionally returns an Exodus history file (extension **.h*), but it is not used in subsequent steps.

The last step is to store the data from the two Sierra/SD output files into a MATLAB binary file using the *UploadData.m* function in the ROMULIS toolbox. *UploadData* requires no inputs, and can be called by opening the script in the MATLAB editor and running it, or by typing the function name in the MATLAB command window to execute it. On call, the function automatically requests the user to select the NetCDF file in a directory search, followed by a request to select the Exodus output file. It is very important to select the two files that are produced from the same Sierra/SD job run, and it is easy to mix up many output files found in the same directory. *UploadData* then searches the files for data relevant to the simulator, and stores them in a **.mat* file with the same name and directory location as the NetCDF file. The **.mat* file is the body file.

3.1.2. Writing a Body File Manually

Not all substructures require the detail provided by commercial finite element modelers, and the user may find it is more practical to define a body based on simpler elements or academic principles, such as a lumped-mass system. In this case, the user can create a body file directly by

supplying all the required variables into MATLAB and saving the data in a binary **.mat* file. Appendix A lists the required variables with their correct names, a description of their data formats, and their relationship to other variables. The variables listed in Appendix A are the same as those sought by the *UploadData.m* function for the imported Sierra/SD reduced systems.

The following guidelines should be observed when creating the required variables:

1) Each substructure in the system can only be modeled so as to produce linear mass, damping, and stiffness matrices.

2) All nonlinear models implemented in ROMULIS are of the discrete, force-constitutive type that couple only two nodes, or couple a node with an inertial point.

3) All nodes that receive an applied load, a nonlinear force model, or a boundary condition, or are output nodes of interest, must be assigned to a nodeset. More than one node can be assigned to a nodeset, but beware that any loading function specified for a nodeset will apply for all nodes in the set. See bullet number 3 in the list in Section 3.1.1 for additional restrictions.

3.2. Creating a System File

The entirety of the system file is defined using only the *SystemSetup.m* script. The user must open the script in the MATLAB editor, manipulate the variables according to the procedure below, and finally execute the script to produce a system file. The following subsections detail the procedure, and are ordered by appearance of the relevant code sections within *SystemSetup*.

3.2.1. Adding Substructures by Loading Body Files

The first section in *SystemSetup.m* defines all the substructures that belong to the system of interest. The user adds the substructures to a system by specifying the name and directory location of all relevant body files. Near lines 17 and 18 are two variables, *bodyFiles* and *bodyPaths*, both of which initialize cell arrays with curly braces. The names of the body files are entered as string arrays in between the curly braces following the *bodyFiles* variable, using commas to separate the different body files.

```
17-  bodyFiles = {'CubeModel','CubeModel','CrazySubstructure'};
```

Note that the same body file can be specified multiple times to create multiple instances of that substructure within the system. Also note not to include the **.mat* file extension to each file name when typing the string array.

The *bodyPaths* variable contains the cell array of strings that specify the directory where the body files specified in *bodyFiles* can be found. The directory can be reference relative to the current active folder in MATLAB, or starting from a top level file space, such as the *C:/* or */home* directory, as is displayed in MATLAB's directory address bar. The order of the directory strings **must** be consistent with the order of the substructures specified in *bodyFiles*.

```
18-  bodyPaths = {'../SimpleShapes/','../SimpleShapes/','C://Users/Myself/
       Documents/MATLAB/'};
```

In the above example, the body file *CubeModel* can be located by moving up one directory level from MATLAB's current folder, and moving into the /*SimpleShapes*/ directory from there. Similarly, the full path is supplied where the *CrazySubstructure* body file is found. Note that forward slashes separate each directory level, and each directory path **must** terminate with a forward slash. Although the user's operating system environment may use a different character for separating levels, MATLAB universally recognizes the forward slash as a separator.

After the body files and their locations are specified, the user must give a name to the system file that will be created. The name is entered as a string array next to the *File* variable near line 20. The user is recommended to choose a name that describes the substructures in the system and the type of loads they receive. Adding the *.mat extension is optional.

```
20-  File = 'CrazyCubeExc100N.mat';
```

At this point, the user should run **only** this first section (lines 1 to 48) of the *SystemSetup* script. This can be done by clicking the "Run Section" button in the Editor toolstrip, or by right-clicking anywhere in the section and selecting "Evaluate Current Section." MATLAB produces a number of figures equal to the number of substructures added to the *bodyPaths* variable. The figures plot the physical x, y, and z coordinate locations of all boundary nodes specified during Craig-Bampton reduction (or all physical nodes for non-Craig-Bampton systems). In addition, the figures display the nodeset to which those nodes belong. An example substructure figure is illustrated in Fig. 3.

Near the top of each figure, the name of the substructure is displayed along with a body number. The body number assigned to a substructure has the same value as the substructure's index location in the *bodyFiles* cell array. For example, the two *CubeModel* substructures listed above are named Body 1 and Body 2, while *CrazySubstructure* is named Body 3. The body number and nodeset information presented in these figures is useful for the sections that follow, so the user is encouraged to leave these figures open.

This first section in the SystemSetup script may also be used to load other files whose data may assist when specifying loads. The commented script near lines 23 to 26 provide example MATLAB binary files whose data is loaded into the workspace and assigned to variables.

```
25-  inData = load('Chatter/SinSweep/EP_SwptSin2excite.mat');
26-  w = inData.w; Xa = inData.Xa;
```

In the above example, the file EP_SwptSin2excite.mat contains Fourier coefficient data that may be used to prescribe a periodic-type motion for certain boundary nodes. This data is stored in the inData structure, and then the frequency (w), and acceleration Fourier coefficient (Xa) vectors are retrieved and stored in separate variables for convenience when the prescribed boundary is defined later in the script (see Section 3.2.6).

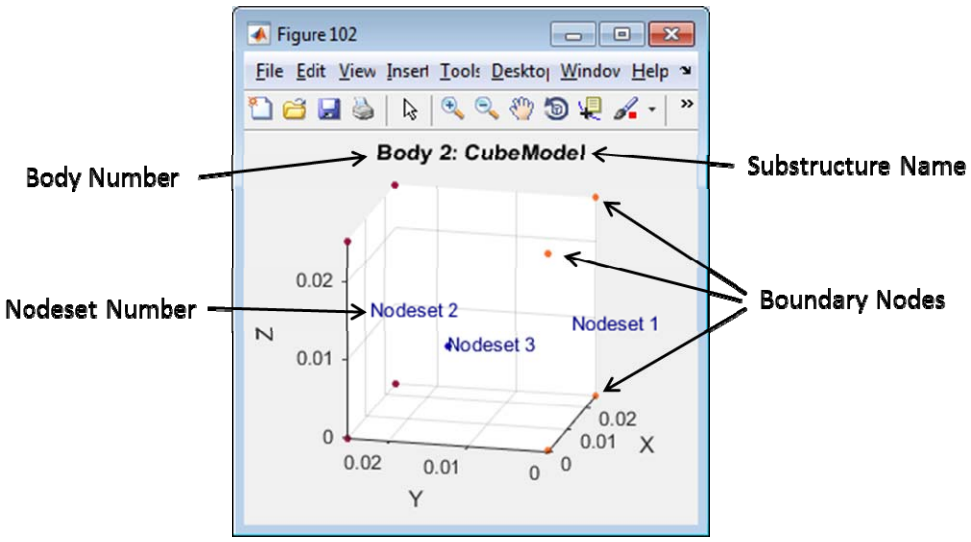

Figure 3: An example substructure figure produced from the *SystemSetup* script. Boundary nodes belonging to the same nodeset are highlighted with the same colored marker.

3.2.2. Applying Substructure Rigid Offsets

The second section in SystemSetup.m allows the user to adjust the nominal coordinate positions for each substructure by specifying a coordinate offset value in a particular direction. This offset is applied to all physical nodes in the substructure regardless of whether or not they belong to a nodeset. These offsets should not be confused with an initial condition, which is an offset on the displacement or velocity degrees of freedom. Rather, the offset translates the substructure rigidly in space.

The *Offset* variable near line 36 specifies coordinate offsets. *Offset* is a numeric array with three columns (when not empty):

1. The first column specifies the number of the body that receives the offset as referenced from the substructure figure (see Section 3.2.1).
2. The second column specifies the direction the offset is applied, with value “1” indicating the first coordinate direction (typically the x-translational direction), value “2” the second (y) direction, and value “3” the third (z) direction. As of this release, ROMULIS does not support offsets in rotational space.
3. The third column specifies the value of the offset.

Offsets may be specified for more substructures or directions by adding more rows to the *Offset* array. If no offset on any substructure is desired, the array may be left empty.

```
36-  Offset = [%Body,Direction (1=x,2=y,3=z),value];
37-             2,3,-0.67
38-             3,1,4.6];
```

In the above example, a -0.67 unit offset is applied in the third (z) coordinate direction on Body 2. In addition, Body 3 receives a 4.6 unit offset in the first (x) direction.

3.2.3. Applying Linear Damping

The third section in *SystemSetup.m* defines linear damping used in the unconstrained global system defined in Eq. (10). By default, ROMULIS uses the damping matrix provided in the body file[1] for each substructure to construct the global damping matrix. Alternatively, users may change the damping matrix by assigning a value to the *system.damping_type* variable near line 40 that matches any of the values in the switch-case statements near lines 41 to 48. Acceptable cases are as follows:

1. For zero damping, set the *damping_type* variable equal to "0."

```
40-  system.damping_type = 0;
```

2. For modal damping, set *damping_type* equal to "1" and enter a numerical array of modal damping ratio values for the *system.zetas* variable. If *N* number of damping ratios are supplied, the first *N* modes of the unconstrained, uncoupled system are given those damping ratios. Therefore, the user should be careful to account for the number of rigid body modes present among the linearly uncoupled substructures (in effect, the modes produced from the Eigenvalue problem using $\tilde{\mathbf{K}}$ and $\tilde{\mathbf{M}}$ from Eq. (10).

```
40-  system.damping_type = 1;
44-  system.zetas = [0.02,0.005,0.0012,0.0007,0.00027];
```

3. For proportional damping, enter a value of "3" for the *damping_type* variable, and select values for the mass proportion coefficient, *system.alpha*, and the stiffness proportion coefficient, *system.beta*. The damping matrix $\tilde{\mathbf{C}}$ is formed from the summation of

$$\tilde{\mathbf{C}} = \alpha\tilde{\mathbf{M}} + \beta\tilde{\mathbf{K}} \tag{24}$$

where α and β are the values in *system.alpha* and *system.beta*, respectively, and $\tilde{\mathbf{M}}$ and $\tilde{\mathbf{K}}$ are the same mass and stiffness matrices produced for Eq. (10).

```
40-  system.damping_type = 1;
46-  system.alpha = 0.01; system.beta = 0.0007;
```

Any other value listed for the *system.damping_type* variable defaults to the damping matrix provided in the substructure body files.

[1] Substructures processed through Sierra/SD can have a damping matrix by specifying the DAMPING keyword in the Sierra/SD input file. This will create a non-zero output for the reduced damping matrix for that substructure.

3.2.4. Defining Initial Conditions

Initial conditions may be defined in the fourth section of the *SystemSetup* script. The first parameter to specify is the starting time for the simulation through the *param.tini* variable near line 51.

```
51-  params.tini = 0;
```

The starting time is typically at zero. It may, however, be useful to start at a different time if, for example, the present system continues from a previous simulation and consistency must be maintained with any previously defined time-dependent loading functions. If continuing from a previous simulation, specifying the correct initial condition becomes especially important.

ROMULIS implements two different approaches to specifying initial conditions. In the first approach, the user specifies certain nodesets in the substructures to have non-zero initial conditions. Alternatively, the user supplies the full global state initial conditions either by typing out the full vector or by loading the vectors from a file. Both approaches have their benefits and pitfalls, and it may not be possible to know the correct initial condition without performing another simulation beforehand.

The *loading_ICs* variable near line 53 controls the approach taken. Setting the value of *loading_ICs* to "0" lets the user specify the first approach, which is to give non-zero initial conditions to certain nodesets in the system. This executes the second section of the if-statement that immediately follows, where the user defines the *ic_nodes* variable near line 60. The *ic_nodes* variable is a numerical array containing five columns (when not empty):

1. The first column specifies the number of the body where the nodeset of interest is found as referenced from the substructure figure (see Section 3.2.1).
2. The second column specifies nodeset number whose nodes receive the initial condition. The nodeset number may also be referenced from the substructure figure.
3. The third column specifies the direction the initial condition is applied. The values 1, 2, and 3 refer to the x, y and z translational directions, respectively, and the values 4, 5, and 6 are, respectively, the x, y and z rotational directions.
4. The fourth column holds the value of the velocity initial condition applied on all of the specified nodes.
5. The fifth column holds the value of the displacement initial condition.

Initial conditions may be specified for more nodesets and directions by adding additional rows to the *ic_nodes* array. If all zero-initial conditions are desired, the *ic_nodes* array is left empty.

```
53-  loading_ICs = 0;
60-    ic_nodes = [%Body, Nodeset, Coord, ICv, ICx];
61-                3, 2, 1, -.04, 0
62-                3, 2, 2,   .7, 1.1];
```

In the example above, Nodeset 2 in Body 3 is given -0.04 unit velocity initial condition in the x direction. The exact same nodeset is additionally given a 0.7 unit velocity and a 1.1 unit displacement initial condition in the y direction.

To specify initial conditions through global state vectors, set the value of the *loading_ICs* variable to "1." This executes only the first section of the if-statement that immediately follows, where the user must supply the initial displacement and velocity vectors for the *px0* and *pv0* variables, respectively. The vector format is a vertical numeric array. The ordering of the degrees of freedom in each substructure follows the ordering of columns in the substructure system matrices (see the *cbmap* variable description in Appendix A). The initial condition vectors from each substructure are then stacked in the global vector similarly to Eq. (9). The blocks are ordered by their corresponding substructure's appearance in the *bodyFiles* variable.

The drawback to specifying global initial condition vectors is that it may not be possible to know the initial state for all degrees of freedom. This is especially true for the fixed-interface modal coordinates in Craig-Bampton-reduced systems. It is more useful, however, to load in the displacement and velocity end state from a previous simulation, and use them as the initial condition for a new simulation, effectively continuing the previous simulation.

If continuing from a previous simulation that was conducted in ROMULIS, the end state displacement and velocity vectors can be found in the *endDisp* and *endVel* variables, respectively, stored in the output response file (see Section 3.4 for more information on response files). The response file can be uploaded into a structure variable in *SystemSetup* using MATLAB's *load* function. From there, the *endDisp* and *endVel* vectors can be retrieved from the structure and stored in the *px0* and *pv0* vectors, respectively.

```
53-  loading_ICs = 1;
55-    ICdata = load('../SimpleShapes/CrazyCubeExc100N-1-resp.mat');
56-    px0 = ICdata.endDisp;
57-    pv0 = ICdata.endVel;
```

If the previous simulation stored additional state variables in the *UD s*tructure (see Section 3.3.2), the end state for those variables should be retrieved for the present system as well. This can be done simply by uploading the *UD* structure from the response file of the previous simulation. Near line 58 is a commented command that stores this structure in the *UD* variable. Uncomment this line if the present system continues from a previous *UD* structure. Otherwise, leave it commented out. Be aware that this loaded *UD* structure will overwrite the *UD* structure that is initialized according to the specification of interface functions (Section 3.2.7), so any changes made to the interface functions from the previous simulation will not be represented in the current simulation.

```
58-  UD = ICdata.UD;
```

3.2.5. Assigning Time-Dependent External Loads

The user may specify any number of external loads to apply on the nodesets in the system. The external loads are force functions dependent on time only, and the functions are procured from a library of analytical equations presently implemented in ROMULIS. The user simply has

to associate a particular nodeset with a force function along with some parameters that define the shape of the function.

Within the *SystemSetup.m* script, external load assignments are controlled by the *f_nodes* variable near line 68. The *f_nodes* variable is a cell array (open and closed with curly braces) where every row in the cell array defines a new load function applied on a nodeset. Each row consists of three columns with the following entries:

1. The first column contains a horizontal numeric vector (open and closed with square brackets) filled with three integer entries.
    a. The first entry is the number of the body that holds the nodeset of interest as referenced from the substructure figure (see Fig. 3).
    b. The second entry is the nodeset number (also referenced from the substructure figure) indicating all nodes on which the forcing function is applied.
    c. The third entry specifies the coordinate direction the load is applied. The values 1, 2, and 3 refer to the x, y and z translational directions, respectively, and the values 4, 5, and 6 are, respectively, the x, y and z rotational directions.
2. The second cell column contains an integer value that denotes the type of load function (e.g. sinusoid, step, haversine, etc.). A list of values and their associated function type is provided in Appendix B.
3. The third cell column contains a vector array of parameter values that characterize the function (sinusoid amplitude and period, etc.). The number and ordering of the parameters depends on the function type, and this information is also provided in Appendix B.

Nodes that receive more than one loading function experience the sum contribution of all the functions applied on those nodes. If no external loads are desired, the *f_nodes* cell array can be left empty.

```
82-  f_nodes = { %[body,nodeset,coord],FModel,[Fparams]}
83-              [1,1,3],2,[-100,.001,0]
84-              [2,3,1],4,[100,.025,0,.5]};
```

The example above has two different force excitations. For the first, Body 1, Nodeset 1 is assigned load function 2, a haversine impulse, in the z coordinate direction. The parameters for the haversine, according to Appendix B, are that it reaches a peak value of 100 force units in the negative direction, lasts duration of 0.001 time units, and starts being applied at time 0 in the simulation. Similarly, for the second excitation, the degree of freedom associated with Body 2, Nodeset 3, and direction 1 has a sine wave excitation with a magnitude of 100 force units, a period of 0.025 time units, and 0 time shift, but does not start the function until time 0.5 in the simulation.

If the external load library lacks a desired function, the user may add additional functions by following the procedure in Section 3.5.1.

3.2.6. Prescribing Boundary Motion (Boundary Conditions)

If the motion of particular nodes is known, the user may prescribe that motion as a form of excitation (or constraint). Similar to external loads, prescribing motion involves assigning time-dependent functions to the nodes to describe their displacement, velocity, and acceleration states. Since each state is a time derivative (or antiderivative) of the others, knowing the function of one state determines the function of the others. ROMULIS includes a built-in library of analytical equations for the states.

In *SystemSetup*, the user must assign a function to the relevant nodes, and declare the parameters that characterize the shape of the function in addition to the state to which the function applies. The variable *bc_nodes* manages the boundary motion prescriptions. Like *f_nodes*, *bc_nodes* is a cell array (open and closed with curly braces) where every row in the cell array prescribes a new boundary function applied on a nodeset. Each row consists of four columns with the following entries.

1. The first column contains a horizontal numeric vector (open and closed with square brackets) filled with three integer entries.
    a. The first entry is the number of the body that holds the nodeset of interest as referenced from the substructure figure (see Fig. 3).
    b. The second entry is the nodeset number (also referenced from the substructure figure) indicating all nodes on which the forcing function is applied.
    c. The third entry specifies the coordinate direction the load is applied. The values 1, 2, and 3 refer to the x, y and z translational directions, respectively, and the values 4, 5, and 6 are, respectively, the x, y and z rotational directions.
2. The second cell column contains an integer value that defines the motion state for which the function applies. Value “1” signifies position, value “2” is velocity, and “3” is acceleration. In practice, a haversine function prescribed for acceleration antidifferentiates to a very different displacement function than a haversine set for displacement directly.
3. The third cell column contains another integer value that designates the type of time-dependent boundary function (e.g. sinusoid, step, haversine, etc.). A list of values and their associated boundary function type is provided in Appendix C.
4. The fourth cell column contains an array of parameter values that characterize the function type (sinusoid amplitude and period, etc.). The number and ordering of the parameters depends on the function type, and this information is also provided in Appendix C. Be aware that some boundary function parameter sets must be provided in another cell array as opposed to a numeric array.

The user should be very careful **not** to prescribe more than one boundary function on the same degree of freedom. During simulation, the contribution from each boundary function on the same degree of freedom is summed, but the initial condition for that degree of freedom reflects only the latest defined boundary function on that degree of freedom. Consequently, there will be inadvertent transient effects in the first time steps of the simulation.

The *bc_nodes* cell array can be left empty if no boundary constraints are desired.

```
 86-  bc_nodes = { %[body,nodeset,coord],State,BCModel,[BCparams]}
 87-              [1,1,3],1,0,[]
 88-              [2,3,1],3,0,[0,.6]
 89-              [3,1,3],3,4,{w,Xa,0,-.0567,.9}};
```

In the above example, the degree of freedom found on Body 1, Nodeset 1, and coordinate 3 is given a position boundary function. Function “0” refers to a zero function, which for position means that the degree of freedom is fixed. No parameters are required for a position zero function according to Appendix C. For the second boundary function, note that by changing the state to acceleration, the parameter set also changes even though the same zero function is used. This is because when the zero function is antidifferentiated to get velocity and displacement, the user must supply velocity and displacement offsets at time 0 to determine a unique solution to those functions.

The third boundary function above exemplifies one that uses a cell array that contains vectors in its parameter set. Function “4” refers to a time signal built up from Fourier coefficients, which requires the user to supply vector arrays for the complex Fourier coefficients and their corresponding frequencies. These vectors are supplied through the *w* and *Xa* variables, which the user must create somewhere in the *SystemSetup* script before *bc_nodes* is declared. A good way to create these variables is to load a file containing the relevant data, and then pass the data to the appropriate cell (see the final two paragraphs of Section 3.2.1).

The user may add more boundary functions to the ROMULIS library by following the procedure in Section 3.5.2.

3.2.7. Coupling Nodes with Nonlinear Constitutive Models

Arguably, the most complex section in the SystemSetup.m script manages the nonlinear elements in the simulation. All nonlinear elements are represented by discrete force-constitutive models that are applied between two nodesets in the system. The user designates one nodeset as the master, and the other nodeset as the slave. In the current version of ROMULIS, there is no practical difference between the master and slave designations, so the user may decide arbitrarily which nodeset to designate as master or slave. There are, however, strict checks in ROMULIS that limit the relationship between the nodes in the master set with those of the slave set. The rules are as follows:

1. All nodes in each set must have either all three translational degrees of freedom, or have all three translational and all three rotational degrees of freedom.
2. If one set contains multiple nodes, then the other set must contain either one node or the same number of nodes. In the case where both nodesets contain an equal number of nodes, each node in one set must have a coincident counterpart in the other set. The user is not required to order the nodes in one nodeset to align with the order of the coincident counterpart nodes in the other nodeset; ROMULIS has an internal algorithm that pairs coincident nodes and their degrees of freedom.

3. If each nodeset contains one node, then the coincidence restriction is lifted. The two nodes can exist in different locations in space.

All nonlinear constitutive models currently implemented in ROMULIS are those that couple only one node to another. In the case that the master and slave nodesets contain multiple nodes, ROMULIS builds a new instance of the same constitutive model between coincident pairs. The constitutive models in general observe the motion (velocity and displacement or position) of the slave node relative to a master node, and they return a force that is applied equally and oppositely on the nodes. Section 2.3 further explains how the relative motion between the master and the slave nodes help to formulate the nonlinear force vector.

When two nodes have been identified for coupling, all the degrees of freedom contained in the nodes are considered for the constitutive model. This accounts for any coordinate rotations that must occur to align a constitutive model in a desired direction, and it allows for coupling between degrees-of-freedom.

In ROMULIS, the assignment of nonlinear elements works by associating a constitutive model in each of the three translational (and rotational if applicable) directions in the rotated frame. Since most of the nonlinear elements implemented in the library describe surface interaction between different substructures, the names of the rotated frame directions likewise relate to surface orientations. The first rotated direction (the direction to which the x axis rotates) is known as the “surface normal” direction, while the second and third coordinate directions (the rotated y and z axes) are the “first tangent” and “second tangent” directions. For nonlinear models applied between nodesets with multiple nodes, the user should be aware that ROMULIS assumes that all nodes have the same surface orientation for their nonlinear elements.

A final general note about specifying nonlinear constitutive models is that some nonlinear elements require additional state variables that must be updated with each time step. Internally, these additional variables are all stored in a structure, called *UD*, which in itself contains a cell array called *Space*. Each cell in *Space* stores the state variables for the particular nonlinear element to which the cell is assigned. The *UD* structure is passed between the force function and the user-output function between each time step, so the user can, for example, construct the user-output function to call the *UD* structure and keep track of the evolution of certain constitutive models as time progresses.

The *int_nodes* variable in *SystemSetup* manages all nonlinear element specifications. Like *f_nodes* and *bc_nodes*, *int_nodes* is a cell array where each row designates a new nonlinear constitutive model, while the columns designate relevant nodesets and functions pertaining to that model. When supplying values into the columns, the user has two different options for specifying nonlinear functions. The first is the usual method of referencing a library of commonly used functions implemented in ROMULIS. For the second option, the user may create a new MATLAB function that can be passed into ROMULIS. If no nonlinear elements are needed, the *int_nodes* array can be left empty.

3.2.7.1. Referencing Constitutive Models from a Library of Functions

For the first option of referencing the library of constitutive equations, the user must supply values for six columns in *int_nodes*. The rules for filling the columns are as follows:

1. The first and second cell columns designate master nodeset and the slave nodeset, respectively. Each column contains a vector array with two entries.
   a. The first entry is the body number for the substructure that holds the nodeset of interest. The body number may be referenced from the substructure figure (see Fig. 3).
   b. The second entry is the number assigned to the nodeset of interest. The nodeset number may also be referenced from the substructure figure.

   Preference is left to the user as to which node is designated the master node (keeping in mind that the constitutive models define motion relative to the master node).

   If the user desires to couple a node to a fixed point relative to the local frame, then that fixed point can be designated to either the master or the slave nodeset. This can be done by filling in either the first or second columns with the "0" value rather than a body-nodeset array. When ROMULIS parses this value, it adds a fixed point to the system in the same location in space as the node coupled to it. Keep in mind that whether the fixed point is the master or the slave, the motion of the joint is still observed as slave relative to master.
2. The third cell column contains a numeric array of length 3 defining the vector (using global x, y, and z coordinates) oriented along the surface-normal direction in the rotated frame from the perspective of the master node. This vector typically locates the slave node relative to the master node, though in some cases it is more intuitive to define the vector as the surface-normal direction at the location of the master node. It is not necessary to normalize the vector by any rule.
3. The fourth cell column is a horizontal numeric array of length 3 defining the vector (using global coordinates) oriented along a surface-tangent axis in the rotated frame. This vector should be perpendicular to the vector defining the surface-normal direction, but ROMULIS subtracts any surface-normal component from the tangent vector just in case.
4. The fifth cell column defines a numeric array with 6 entries containing integer values (except '1') representing the force-constitutive functions to be oriented in the three translational and three rotational directions. The first three entries represent the surface-normal, first tangent, and second tangent translational directions, ordered respectively. The fourth through sixth entries represent the rotational directions of the same ordering. A list of acceptable integer values and constitutive functions they represent is given in Appendix D. If no constitutive function is desired for a certain direction, the corresponding entry may be filled with a '0' value. If a node does not have rotational degrees of freedom, then the fourth through sixth entries are ignored.
5. The sixth cell column contains a nested cell array of length 6. Each cell array contains a numeric array of parameter values associated with the functions given in the fifth cell column of *int_nodes*. The first three cells give parameter arrays for the normal, first

tangent, and second tangent functions, ordered respectively, and the last three cells give parameter arrays for the rotation functions of the same order. For each parameter array, the number and order of the parameters depends on the function type, and this information is also provided in Appendix D. If a node does not have rotational degrees of freedom, then the rotation parameter cells are ignored.

```
77-  int_nodes = {
78-    0,[3,2],[1;1;0],[-1;1;0],[5,7,7,2,0,0], ...
79-        {[100e9,.07],[.6,1e-4,300],[.6,1e-4,300],[1e6,0],[],[]};
80-    [1,11],[2,21],[.866,.5,0],[-.5,.866,0],[0,9,9,0,0,0], ...
81-        {[],[9,5e4,-.7,.07],[9,5e4,-.7,.07],[],[],[]}
82-  };
```

The above example depicts two sets of coupled nodes. For the first set, the nodes in Body 3, Nodeset 2 (the slave nodeset) are each coupled to ground (the master nodeset) using a Hertz contact element in the normal direction, Coulomb friction elements in both tangential directions, and a linear spring and dashpot in the surface-normal rotation direction. The rotated frame is such that the global frame is rotated 45° about the positive z-axis. The Hertz contact element has an effective modulus of 100×10^9 units, and an effective radius of 0.07 units. Both of the Coulomb friction elements in the two tangential directions have a friction coefficient of 0.6, which saturates at a velocity of 10^{-4} units, and a normal force parameter of 300 units. The rotation spring is given a stiffness of 10^6, and the dashpot has zero damping, which effectively removes the dashpot from this model.

The second set couples a master nodeset on Body 1 to a slave nodeset on Body 2. The normal translation direction and all rotational directions have no constitutive elements, but the two tangent directions hold Segalman four-parameter Iwan models in translation. The rotated frame is such that the global frame is rotated 30° about the positive z axis. Both of the Iwan models are given the same parameterization.

The user may implement more force-constitutive functions in the ROMULIS library by following the procedure in Section 3.5.3.

3.2.7.2. Supplying a User-Created Constitutive Model

On occasion, an interface condition between two nodes cannot be captured by simply assigning nonlinear elements in the three coordinate directions. In this case, the user has the option to upload a custom force-constitutive MATLAB function. Creating a custom function gives the user freedom in defining the nonlinearity based on a limited set of state inputs for the two nodes, but the output must always be the force vectors that are applied on the two nodes. Section 3.3.1 contains more information on how to create a custom force constitutive function.

To upload a user-created constitutive function, the user must modify a row in the *int_nodes* variable differently than when referencing a library function. The six cell entries in that row are filled in as follows.

1. The first and second cell columns define the master node and slave node, respectively. These entries are filled in the exact same way as for library functions. See rules 1 and 2

under the Referencing Constitutive Models from a Library of Functions section for the details.

2. The third and fourth cell columns are empty arrays. For library functions, the user provides vectors to define the surface normal and tangent directions to rotate the frame of reference as desired. For user-created functions, the nodal degrees of freedom that are passed into the function remain in the global frame, so the user must code up any coordinate rotations in the function as necessary.
3. The fifth cell column contains a single, numeric '1'.
4. The sixth cell column contains another cell array with two cells.
    a. The first cell entry contains a string array listing the directory where the function can be found. The syntax for listing the directory is the same for listing directory in the *bodyPaths* variable.
    b. The second cell entry lists the name of the function in string characters. Do not include the **.m* extension with the function name.

```
91-  int_nodes = {
92-        [2 1],[3,2],[],[],1,{'../SimpleShapes/Funcs/','Lanyard'}
93-    };
```

The above example couples the master node in Body 2, Nodeset 1 with the slave node in Body 3, Nodeset 2 using the user-created function, *Lanyard*.

3.2.8. Identifying Output Degrees of Freedom

When the numerical integrator finishes simulating the transient response for the system, ROMULIS saves the response only for the nodesets specified in the *out_nodes* variable near line 86 (in Section 3.2.8) of *SystemSetup.m*. If the response at all degrees of freedom is desired, the *out_nodes* cell array is left empty. Otherwise, *out_nodes* is a cell array with multiple rows and only one column. Each row specifies a new set of output nodes of interest, and the single cell entry in that row contains a numeric array with three elements.

1. The first value in the array lists the number of the body that holds the nodeset of interest. The body number may be referenced from the substructure figure (see Fig. 3).
2. The second value is the number assigned to the nodeset of interest. The nodeset number may also be referenced from the substructure figure.
3. The third value represents the coordinate direction of interest. The values 1, 2, and 3 refer to the x, y and z translational directions, respectively, and the values 4, 5, and 6 are, respectively, the x, y and z rotational directions.

```
86-  out_nodes = { %[body,nodeset,coord]}
87-                [4,1,2]
88-                [2,3,1]};
```

In the above example, the response of the degrees of freedom located on Body 4, Nodeset 1 in the y-direction are of interest as are the x-direction degrees of freedom on Body 2, Nodeset 3.

3.2.9. Locating the User Output Function

If the user has constructed an output function to supplement the IMEX integrator [4], then the name of the user-output function can be supplied in string characters to the *UO_File{1,1}* variable. Make sure **not** to include the **.m* file extension with the file name. The user must then provide the directory where the function is located in string characters to the *UO_File{1,2}* variable. Take care to terminate the string with a forward slash character.

```
91-  UO_File{1,1} = 'UO_Iwan4_BRBr2';
92-  UO_File{1,2} = 'UO Functions/';
```

If no output function is desired, these *UO_File* variables must be empty arrays.

If continuing from a previous simulation that uses the same user output function, then the UD structure from the previous simulation's response file must be uploaded to preserve any recorded variables. The response file and its *UD* structure may be loaded into *SystemSetup* by following the procedure at the end of Section 3.2.4. Beware that this loaded *UD* structure overwrites the original *UD* structure that would apply for a fresh simulation. Therefore, any changes made to the nonlinear constitutive functions from a previous simulation will not reference the correct *UD* structure for storing and updating internal state variables.

3.2.10. Running SystemSetup

After the user completes Sections 3.2.1 to 3.2.9, the *SystemSetup.m* script may finally be run. MATLAB takes the data provided in *SystemSetup*, processes it in the *Initialization* script, and stores the processed data in a system file. The system file is given the name provided in the *File* variable in Section 3.2.1, and is stored in the same directory as the first body file uploaded in the *bodyPaths* variable. The system file is ready to be uploaded into the integrator following the procedure in Section 3.4.

3.3. Creating Supplementary Functions

3.3.1. User-Created Nonlinear Constitutive Function

If the ROMULIS library of nonlinear constitutive models (see Appendix D) does not contain a desired function, or if the interaction between two nodes cannot be modeled simply by attaching nonlinear elements in the three directions, then the user may define a custom constitutive model by creating a new MATLAB function and passing it into the simulator. This section describes the variables that are supplied to the function in addition to those that are sent out. The user is directed to Section 3.2.6 for information on how to upload the new function into ROMULIS after it is made.

Creating a new function in MATLAB starts with the function declaration. The user may give any name to the function, but it must always have eight input arguments and three output arguments.

```
1-  function [Fm,Fs,Space] = FuncName(t,um,us,vm,vs,xm,xs,Space)
```

The input arguments are the variables available to the user to help construct the force-constitutive model. They consist mainly of the motion state for the master and slave degrees of freedom in addition to a structure array that stores and updates variables with each time step. The input arguments are further detailed in Table 2.

At times, a nonlinear constitutive model will require additional state variables (besides displacement and velocity) to be updated with each time step. These variables can be stored in the *Space* structure. The *Space* structure is a MATLAB structure array that contains the fields given in Table 3.

The *init* field is the key to add and initialize new variables that will be used consistently in the function. It is used to enter an if-statement, located at the beginning of the function, where the user can declare more variables in the *Space* structure before the simulation begins. Additional states and other variables that are updated and passed between time steps must be declared in this if-statement. The if-statement is set up as follows:

```
 2-  if isfield(Space,'init') % Declare and initialize new variables.
 3-      % Remove init field to skip if-statement in later steps (required).
 4-      Space = rmfield(Space,'init');
 5-      % Initialize Iwan model parameters (example).
 6-      Space.Fs = 100;   % Slip force
 7-      Space.Kt = 1e5;   % Tangent stiffness
 8-      Space.chi = -0.7; % log slope of energy dissipation
 9-      Space.beta = 3;   % Ratio of slip term to power-law term
10-      Space.y0 = zeros(100,1); % initialize states of 100 sliders
11-  end
```

Table 2: Descriptions for the input arguments required in a user-created nonlinear constitutive function.

Argument	Description
t	The time value at the present time step.
um	A vertical array containing the current displacement values for the degrees of freedom in the master nodeset.
us	A vertical array containing the current displacement values for the degrees of freedom in the slave nodeset.
vm	A vertical array containing the velocity values in the current time step for the degrees of freedom in the master node. The number and order of the degrees of freedom are the same as in *um*.
vs	A vertical array containing the velocity values in the current time step for the degrees of freedom in the slave node. The number and order of the degrees of freedom are the same as in *us*.
xm	A vertical array containing the resting coordinate positions for the master degrees of freedom. The number and order of the degrees of freedom are the same as in *um*.
xs	A vertical array containing the resting coordinate positions for the slave degrees of freedom. The number and order of the degrees of freedom are the same as in *us*.
Space	A structure array containing degree of freedom identity information. Meant to store additional variables that will be updated with each time step. *Space* must always be supplied as an input argument regardless of whether or not it is used in the function.

Table 3: Descriptions for the fields in the *Space* structure array.

Field	Description
model	Contains the string 'User-created function'. Helps the user identify the *Space* array in the *UD* structure.
init	A Boolean field used to invoke an if-statement to allow the user to initialize additional variables in the *Space* structure on the first time step.
dof_map_m	An *N*×3 numeric array where *N* is the number of rows in *um* from Table 2. Each row in *dof_map_m* identifies the body, node, and direction for the degree of freedom corresponding to the same row in *um*. Column 1 gives the body number (see Fig. 3), column 2 gives node number, and column three gives the direction number (1 = x, 2 = y, 3 = z, 4 = rx, 5 = ry, 6 = rz). Helps the user identify the degree of freedom associated with each row in *um*.
dof_map_s	An *N*×3 numeric array where *N* is the number of rows in *um* from Table 2. Each row in *dof_map_s* identifies the body, node, and direction for the degree of freedom corresponding to the same row in *us*. Column 1 gives the body number (see Fig. 3), column 2 gives node number, and column three gives the direction number (1 = x, 2 = y, 3 = z, 4 = rx, 5 = ry, 6 = rz). Helps the user identify the degree of freedom associated with each row in *us*.

Table 4: Descriptions for the output arguments required in a custom nonlinear constitutive function.

Argument	Description
Fm	The vector of forces applied on the master degrees of freedom. The number and ordering of the degrees of freedom is the same as that of *um* in Table 2. Some degrees of freedom in *um* may be repeated (see *Space.dof_map_m*). Make sure that the last of the repeated entries in *Fm* holds the correct force value. All repeated entries before the last can be ignored.
Fs	The vector of forces applied on the slave degrees of freedom. The number and ordering of the degrees of freedom is the same as that of *us* in Table 2. Some degrees of freedom in *us* may be repeated (see *Space.dof_map_s*). Make sure that the last of the repeated entries in *Fs* holds the correct force value. All repeated entries before the last can be ignored.
Space	Updated *Space* structure. This must always be supplied regardless of whether or not any changes were made to it.

The three output arguments are the force vectors applied on the master degrees of freedom and slave degrees of freedom, as well as the updated *Space* structure. Typically, a nonlinear element applies equal and opposite reaction forces on the master and slave nodes, but the custom function gives the user the freedom to differ the force values to satisfy the constitutive model. The values of the components in both the master and slave force vectors are all added to the global force vector on the right-hand side of Eq. (12), so the user must keep track of which values are directed in the positive directions as well as in the negative directions. Table 4 describes the output arguments in more detail.

Given the input arguments listed in Table 2, the user has complete freedom to write the rest of the function so as to produce the two output force vectors listed in Table 4. In scripting this function, the user is expected to know the order of the degrees of freedom provided in the input state vectors. The *dof_map_m* and *dof_map_s* fields provided in the *Space* structure can assist with identifying these degrees of freedom. The *Space* structure is provided in the system file after successfully running the *SystemSetup* script. Within the *system* structure in the system file is the *Int_loc* field, which is a cell array containing four columns and as many rows as in the *int_nodes* array from the *SystemSetup* script. After locating the same row in *Int_loc* as the row the user-created function is defined in *int_nodes*, the fourth cell in this row contains an integer value. This integer references the cell index of the *SpaceCells* cell array found in the *UD_init* field in the *system* structure. This cell within *SpaceCells* contains the *dof_map_m* and *dof_map_s* associated with the user-created function.

3.3.2. User Output Function

The user output function is a user-supplied MATLAB function that IMEX, the numerical integrator, recognizes as a subroutine in its algorithm. The user output function is called in between time steps to perform some manner of processing on results of the just-completed time step. The major use of user output function in ROMULIS is to record certain variables after each time step, whether to be accessed for post-processing after integration, or to be fed back into the

integrator for the next time step. At the end of a simulation, all recorded variables are supplied to the response file for the user to access.

The user may give any name to the output function, but every user output function must have one output argument and six input arguments as follows:

```
1-  function UD = MyOutputFunc(t,dt,u,v,UD,flag)
```

The descriptions for each input argument are given in Table 5.

The entire point of the user output function is to initialize and update *UD*, the user output structure. *UD* is a structure that can record all variables of interest to the user in addition to those that are normally given by the time integrator. *UD* is the only variable that is passed between all supplementary functions in IMEX, including the force function. Therefore, any variables (such as the internal states of a nonlinear constitutive function) that are updated and recorded in *UD* within the force function may also be accessed in the user output function.

The *UD* structure contains variables that help the user process the raw displacement and velocity vector inputs to the user output function, and it holds data from the force constitutive functions implemented in the system. Table 6 gives descriptions for the fields that ROMULIS creates in the *UD* structure before starting numerical integration.

Typically, an initialization control statement immediately follows the function declaration. In the initialization control statement, the user creates additional variables that are of interest and are typically updated as the integrator progresses though each time step. The control statement checks the *init* field in the *UD* structure for whether it has a value of '1' (which is always the case on the first time step). After executing the commands in the control statement, the value of *init* is switched to '0' so that subsequent time steps do not reinitialize the variables.

```
2-  if UD.init == 1 % initialize variables on the first time step.
3-      UD.timevec = []; % records time.
4-      UD.vvec = [];    % records velocity input to nonlinear
5-                       %   constitutive function.
6-      UD.Fvec = [];    % records force output from nonlinear
7-                       %   constitutive function.
8-      UD.init = 0; % change init so that variables aren't reinitialized.
9-  end
```

In the example above, three variables were initialized: a vector for recording the current time, a vector for recording the relative motion between the master and slave node for a nonlinear constitutive function to be identified later, and a vector for recording the force output from said nonlinear function.

Table 5: Descriptions for the output arguments required in a custom nonlinear constitutive function.

Argument	Description
t	The time value at the current time step
dt	The size of the current time step.
u	The vector of displacements for active degrees of freedom in the current time step. The order of the degrees of freedom is given by *dofmap* field in the *UD* structure (see Table 6).
v	The vector of velocities for active degrees of freedom in the current time step. The order of the degrees of freedom is given by *dofmap* field in the *UD* structure (see Table 6).
UD	The user output structure.
flag	A Boolean value set to '1'.

Table 6: Descriptions of the variables found in the *UD* structure from a user output function.

Variable	Description
init	A Boolean field (value set to '1'). Most commonly used to enter a control statement at the beginning of a simulation in order to initialize variables in *UD* to be used as the integrator computes.
dofmap	A vector having three columns and the same number of rows as the displacement and velocity vectors that are supplied as input arguments to the user output function. Each row in *dofmap* is associated with the degree of freedom in the same row of the displacement and velocity vectors. The first column gives the body number for that degree of freedom, the second column gives the node number, and the third column gives the direction number.
coords	The vector of the nominal resting coordinate positions of all active degrees of freedom. The order of degrees of freedom is the same as in *dofmap*.
SpaceCells	An array of cells where each cell contains a structure with fields pertaining to a nonlinear element. The fields are exactly those of the *Space* structure described in Table 3.

The *SpaceCells* variable in *UD* carries the recorded data from each interface model. *SpaceCells* is a cell array where each cell contains a structure of parameters relating to a particular nonlinear model. Each structure contains exactly the fields described in Table 3 for the *Space* structure, plus other fields as needed by the nonlinear function. The nonlinear functions in ROMULIS's library (see Appendix D) have four additional variables added their *Space* structure as described in Table 7.

Table 7: Descriptions for the additional fields added to the *Space* structure array for each nonlinear constitutive element in ROMULIS's library.

Field	Description
param_set	Contains the array of parameters that the user assigns to the particular nonlinear constitutive element thought the *int_nodes* variable in the *SystemSetup* script.
u	The relative displacement between the master and slave degrees of freedom as observed by the nonlinear constitutive element for the current time step. This value is left as '0' for elements that do not require displacement input.
v	The relative velocity between the master and slave degrees of freedom as observed by the nonlinear constitutive element for the current time step. This value is left as '0' for elements that do not require velocity input.
F	The force output from the constitutive element for the current time step.

In the user output function, the *Space* structure for a desired nonlinear model may be retrieved by calling *UD.SpaceCells*{*i*}, where *i* may be substituted for the appropriate cell index value. Likewise, *UD.SpaceCells*{*i*} must be updated or overwritten if any changes are made to the fields in its Space structure (not recommended in most cases).

```
10-  UD.timevec = [UD.timevec; t]; % Add current time to time vector
11-  SpaceCoulomb = UD.SpaceCells{3}; % Retrieve Space structure for the
12-                                   % Coulomb friction model.
13-  UD.vvec = [UD.vvec; SpaceCoulomb.v]; % record current velocity.
14-  UD.Fvec = [UD.Fvec; SpaceCoulomb.F]; % records current force output.
```

The user is expected to know which *Space* cell indices pertain to what functions, and the system file can help identify the *Space* cells after the *SystemSetup* script is run successfully. Within the *system* structure in the system file is the *Int_loc* field, which is a cell array containing four columns and as many rows as in the *int_nodes* array from the *SystemSetup* script. After locating the same row in *Int_loc* as the row the user-created function is defined in *int_nodes*, the fourth cell in this row contains an array of integers. There are always six columns in this array, each representing the model oriented in one of the 6 directions (the 1st column is surface-normal translation, 2nd and 3rd columns are surface-tangent translations, 4th column is normal rotation, 5th and 6th are tangent rotations). Each integer in this array references the correct cell index of the *SpaceCells* array for its particular model.

If the index array contains more than one row (i.e. multiple pairs of nodes were specified to have the same instance of a particular constitutive element), then an additional search can be made to identify the one or more pairs of nodes that are of interest. After identifying all possible *SpaceCell* indices that could pertain to the element of interest, observe those cells from the *SpaceCells* array found in the *UD_init* field in the *system* structure. Each cell contains a *dof_map_m* and *dof_map_s* field, which are mapping vectors for the applicable master and slave degrees of freedom, respectively. Each vector has three elements. The first entry contains the body number for the degree of freedom, the second entry is the node number, and the third entry is the direction number (1=normal translation, 2=tangent translation, etc.).

3.4. Running a Simulation

After a system file has been created, it is ready to upload into the time integrator using the *SerialExecuter* script. With *SerialExecuter*, the user simply specifies all parameters to tweak the IMEX2 time integrator [4,12], and directs the script to the directory where the system file is located. Upon running the script, *SerialExecuter* calls upon a number of subfunctions to assemble the system, set up the force function, and ultimately run the integrator. After numerical integration is complete, *SerialExecuter* collects the response solution and any user output data, and stores them in the response file.

Upon opening the *SerialExecuter* script in the MATLAB editor, the first step to complete is to locate a system file. Near lines 14 and 15 are the two variables *filepath* and *inFile*. The path to the directory locating the system file is typed as a string array in the filepath variable. Remember to include the forward slash, '/', to terminate the path name. The name of the system file is typed as a string array for the inFile variable. The *.mat extension must be included after the name.

```
14-  filepath = 'C:/Users/user/Documents/SimpleShapes/';
15-  inFile = 'CrazyCubeExc100N.mat';
```

Next, the time period for simulation is specified. In anticipation for a case where a user may specify a very long simulation such that the RAM needed to store such time histories exceeds the capacity given to the machine, *SerialExecuter* was designed to divide the long simulation into smaller simulations that run one after another until the desired time period is completed. Each time segment produces its own response file, and, in this way, ROMULIS effectively pauses the simulation to store the time histories into the hard drive before clearing RAM to continue the next segment of the simulation.

The user defines two parameters: The ending time for the entire simulation, and the time period per simulation segment. When the length of the entire simulation is divided by the time period of each segment, this gives the total number of simulation segments that *SerialExecuter* computes. The ending time for the simulation is supplied to the *t_end* variable near line 21, and the period per segment is given for the *SimT* variable on line 22. The simulation starting time is defined in *SystemSetup* (see Section 3.2.4), and that value is carried over to *SerialExecuter*.

```
21-  t_end = 0.1;
22-  SimT = .02;
```

In the above example, the total simulation period is 0.1 (given that the starting time is at 0). At 0.02 time units per segment, this simulation computes five time segments, and ultimately produces five response files.

The final variables of *SerialExecuter* that the user can adjust are the IMEX2 time integrator parameters. These include specification of the minimum and maximum time step sizes, and the relative tolerance for convergence. These same parameters can be referenced in [4] for a similar algorithm, and they are repeated in Table 8. In *SerialExecuter*, these variables are stored directly in the *params* structure. If the user does not wish to specify a new value for a parameter, that parameter must be commented out.

Table 8: Descriptions for the IMEX time step parameters. The following variables are fields in the *params* structure.

Field	Description (default value)
dt	Initial time step size (1e-6).
dtmin	Minimum allowable step size (1e-12).
dtmax	Maximum allowable step size (1e-3).
tol	Relative tolerance for convergence (1e-3).
output_res	Resolution of the output time histories (*SimT* * 0.001)
output_flag	Boolean for displaying progress dots in command window (1).
cutoff	Real time to elapse before integrator quits (3e6 seconds).
method	Order of the explicit integrator (5). Possible values are 4 and 5 for IMEX2.

The *SerialExecuter* script can now be run.

Upon running SerialExecuter, the MATLAB command window will display some preprocessing operations and then proceed with numerical integration of the first time segment. If *output_flag* is set to '1', the command window will also display a dot '.' to show the progress of the integrator. Each dot represents the completion of 3000 time steps (the precise number is dependent on the parameters passed into the IMEX2 algorithm, specifically *output_res*). After 150,000 time steps are completed (specifically, 50 dots), the command window will terminate the line of dots with the value of the latest time step completed, and then start a new line of 50 dots.

```
Assembling global system...     Done.
Partitioning constrained DOF... Done.
Writing Force_Func.m ...        Done.

Reuploading search path to update Force_Func.m ...Done.

Performing numerical integration...
.................................................t = 0.0100108 s
.................................................t = 0.0199989 s
..............|
```

When a simulation segment has completed integration, the command window will display the total real time it took to complete the simulation, and then declare that post-processing is complete.

```
Done. Elapsed Time is 767.8143 seconds.
Post-Processing...Done.
```

When this last line appears, a response file for that segment should appear in the same directory where the system file is found. The response file name is the same as that of the system file, except that it has been appended with a ' –r–*resp.mat'*. Here, the r is a placeholder for the segment number. In the above example, the first response file that is produced is called

CrazyCubeExc100N-1-resp.mat, then *CrazyCubeExc100N-2-resp.mat*, and so on up to *CrazyCubeExc100N-5-resp.mat*.

Each response file is a MATLAB binary file containing the eight variables listed in Table 9.

Table 9: Descriptions for the variables in a response file.

Variable	Description (default value)
time	Time vector.
Out_map	A three-column mapping array where each row corresponds to an output degree of freedom. The first column gives the body number, the second column gives the node number, and the third column gives the direction number (1=x translation, 2=y trans, 3=z trans, 4=x rotation, 5=y rot, 6=z rot).
Disp	Displacement time histories for the output degrees of freedom. Each row corresponds to the degree of freedom referenced in the same row of Out_map.
Vel	Velocity time histories for the output degrees of freedom. Each row corresponds to the degree of freedom referenced in the same row of Out_map.
Coord	A vertical vector of the nominal coordinate position for each output degree of freedom. Each row corresponds to the degree of freedom referenced in the same row of Out_map.
UD	User output structure (see Section 3.3.2).
endDisp	The global vector of displacements for the full assembled system at the final time step. Useful as an initial condition in a subsequent simulation.
endVel	The global vector of velocities for the full assembled system at the final time step. Useful as an initial condition in a subsequent simulation.

3.5. Adding Functions to the ROMULIS Library

The user may implement new functions to supplement ROMULIS's existing library of functions. This process essentially requires the user to write new lines of code into one or more of the toolbox subfunctions that help create the force function. However, the code for implementing new functions may not be intuitive to even an experienced MATLAB user, so some explanation is given here as to what the code is trying to accomplish.

From a traditional MATLAB coding perspective, a function would normally be referenced with a function handle that is stored in a variable to be passed into the greater force function. Such an approach creates overhead in the MATLAB runtime such that processing power, and therefore time, must be dedicated to search for the handle in memory, locate the function, run the function, and finally return the output and clear locally-allocated memory. As this process may be repeated hundreds of thousands of times in a simulation, that overhead time accumulates and slows the simulation substantially.

The approach that ROMULIS takes to circumvent the overhead is simply to write the function directly into the force function as needed. This is a concept known as

metaprogramming, where the ROMULIS code itself automatically writes a new script for the force function to be used recursively as code at a later point. Normally, MATLAB is not an ideal language suited to metaprogramming, but since the force function is written and updated once rather than thousands of times in any simulation, the practical benefits are regained.

The tools that MATLAB provides for writing new scripts are the basic input/output file commands. In effect, ROMULIS writes commands as string arrays into a MATLAB m-file that becomes the script for the force function. Hence, in the subfunctions discussed in Sections 3.5.1 through 3.5.3, the user writes the commands of the function as string arrays rather than actual executable lines.

This metaprogramming approach has the additional benefit of writing the force function economically. That is, the force function contains only the commands that are absolutely necessary to resolve both applied and internal loads on the system. Had the force function been a traditional script in the line of programming, it would require a complex, convoluted assembly of nested control statements designed to parse through user input on loading conditions, and return the correct force value. This in itself creates more overhead in each iteration that the integrator completes. With metaprogramming, the parsing is left to the subfunctions that write the force function all before the integrator is ever called.

More to the point, in addition to writing function commands as string arrays in these subfunctions, the user must also navigate their arrays of nested control statements, which the following subsections will explain.

3.5.1. External Load Functions

New external load functions (dependent on time only) can be implemented by adding code to the *Load_Library* function. *Load_Library* contains a single control statement, a switch-case statement (begins on line 29), which observes the integer held in the *model* variable. The model variable holds the number assigned to the load function as discussed in Section 3.2.5 and in Appendix B. In essence, the load number is compared to the integer described in each of the case statements. When a match between the case number and the load function number is found, then the contents of that case statement is followed.

At present, there are six implemented cases, case 1 through case 6, which should not be altered in order to maintain consistency with Appendix B. A new load function is added as a subsequent case. As an example, a linear, ramp-up type load function is added for the sake of this documentation. This function is assigned load number 7, and, and, hence, is typed under a new case 7 in the control statement. To characterize this function, the parameters *m*, the slope of the ramp-up, and t_0, the time in the simulation that the force starts ramping up, are assigned. Then the output of this force function can be expressed analytically as

$$F = \begin{cases} 0, & t \le t_0, \\ m(t - t_0), & t > t_0. \end{cases} \tag{25}$$

In *Load_Library*, this function is typed as a string array into the *F_line* variable. Note that each line of string commands terminates with a newline character (\n) so that MATLAB knows to begin the next set of strings on a new line in the force function.

```
 29-  switch model
   ⋮       ⋮
 74-       case 7 % linear ramp-up force
 75-         F_line = [
 76-           'm = Fp{',ii,'}(1);\n'... % slope parameter
 77-           't0 = Fp{',ii,'}(2);\n'... % time parameter
 78-           'F = m*(t-t0).*(t>t0);\n' % Eq. (25)
 79-         ]; % close F_line
   ⋮       ⋮
 82-  end
```

The first lines of strings in *F_line* are commands that retrieve the parameter values for this function. The parameter values for all load functions are passed into the force function through the *Fp* input argument. *Fp* is a cell array where each cell contains an array of parameter values as directly copied from the rows of *f_nodes* in *SystemSetup* (see Section 3.2.5). The variable *ii* in *Load_Library* contains an integer as a string that references the correct cell entry, so it is interjected into the *Fp* string declaration above.

The integer in parenthesis following the bracketed *ii* references the correct entry in the numeric array to retrieve the parameter value. Therefore, when the parameter array is entered into *f_nodes* in *SystemSetup*, the array must be a vector of length at least 2, with the slope parameter in the first entry, and the time parameter in the second entry. If the user desires the supplied parameters to be vectors rather than single numbers, then the parameter array can be supplied as its own cell array. The integers would then be enclosed in brackets rather than in parenthesis.

The last string line in *F_line* writes the force function itself based on Eq. (25). The value of the force must be assigned to the variable *F*. In addition, the variable *t* is the current time. It is supplied by the force function itself, so there is no need to initialize *t* in *F_line*.

Upon closing F_line, the user will have successfully implemented a new load function.

3.5.2. Prescribed Boundary Functions

The *Boundary_Library* function in the ROMULIS toolbox holds the library of presently implemented functions for prescribed boundaries. There are two switch-case statements, a parent and child, of which the user needs to keep track. The parent switch-case (begins on line 97) observes the *model* variable, which holds the number assigned to the boundary function. The child-switch case examines the *state* variable, which holds a values of '1' if the base function applies to displacement, '2' if the base function applies to velocity, or '3' if the base function applies to acceleration. More information on both of these number assignments is given in Section 3.2.7 and in Appendix C. When the *Boundary_Library* function is called, it matches the *model* and *state* numbers to user input in order to return the correct boundary function.

The library currently has five *model* cases implemented, and each *model* case (except case 5) has three *state* cases. All of these cases should not be altered to maintain consistency with Appendix C, but new boundary functions should be added as subsequent *model* cases after number 5. In the following procedure, the ramp-up function is added to the library as a demonstration. This function is assigned function number 6, and, hence, is typed under a new *model* case 6 in the control statement. To characterize this function, the parameters *m*, the slope of the ramp-up, and t_0, the time in the simulation that the force starts ramping up, are assigned. If the base function is given to displacement, *x*, then, after differentiating to velocity *v* and acceleration *a*, the three states are expressed analytically as

$$x = \begin{cases} 0, & t \le t_0, \\ m(t - t_0), & t > t_0, \end{cases} \tag{26}$$

$$v = \begin{cases} 0, & t \le t_0, \\ m, & t > t_0, \end{cases} \tag{27}$$

$$a = 0. \tag{28}$$

Equations (26) through (28) are typed under the *state* case 1. If the base function is given to velocity, then *state* case 2 is given the equations

$$x = \begin{cases} x_0, & t \le t_0, \\ \frac{m}{2}(t - t_0)^2 + x_0, & t > t_0, \end{cases} \tag{29}$$

$$v = \begin{cases} 0, & t \le t_0, \\ m(t - t_0), & t > t_0, \end{cases} \tag{30}$$

$$a = \begin{cases} 0, & t \le t_0, \\ m, & t > t_0. \end{cases} \tag{31}$$

Note that Eq. (29) has another parameter x_0, the initial displacement at time zero, which appeared as a constant after integrating the velocity function into displacement. Under *state* case 3, the equations are

$$x = \begin{cases} v_0(t - t_0) + x_0, & t \le t_0, \\ \frac{m}{6}(t - t_0)^3 + v_0(t - t_0) + x_0, & t > t_0, \end{cases} \tag{32}$$

$$v = \begin{cases} v_0, & t \le t_0, \\ \frac{m}{2}(t - t_0)^2 + v_0, & t > t_0, \end{cases} \tag{33}$$

$$a = \begin{cases} 0, & t \le t_0, \\ m(t - t_0), & t > t_0. \end{cases} \tag{34}$$

Again noting the addition of v_0, The initial velocity at time zero.

Boundary_Library serves a dual purpose of computing state vectors from these equations in addition to typing them out as strings for the force function, so each *state* case must provide for both types of outputs. The displacement, velocity, and acceleration vectors are assigned to the *x*, *v*, and *a* variables, respectively, and the text output is given to the *bc_line* variable. Note that each line of string commands terminates with a newline character (\n) so that MATLAB knows to begin the next set of strings on a new line in the force function.

```
 97-  switch model
   :    :
362-    case 6 % linear ramp-up motion
363-      switch state % check state
364-        case 1 % displacement base function
365-          m = params(1);  % get slope parameter
366-          t0 = params(2); % get time parameter
367-          x = m*(t-t0).*(t>t0); % Eq. (26)
368-          v = m*(t>t0);         % Eq. (27)
369-          a = t*0;              % Eq. (28)
370-          bc_line = [
371-          'm = BCp{',ii,'}(1);\n'...   % slope parameter
372-          't0 = BCp{',ii,'}(2);\n'...  % time parameter
373-          'x = m*(t-t0).*(t>t0);\n'... % Eq. (26)
374-          'v = m*(t>t0);\n'...         % Eq. (27)
375-          'a = t*0;\n'];               % Eq. (28)
376-        case 2 % velocity base function
377-          m = params(1);  % get slope parameter
378-          t0 = params(2); % get time parameter
379-          x0 = params(3); % get initial displacement parameter
380-          x = m/2*(t-t0).^2.*(t>t0) + x0; % Eq. (29)
381-          v = m*(t-t0).*(t>t0);           % Eq. (30)
382-          a = m.*(t>t0);                  % Eq. (31)
383-          bc_line = [
384-          'm = BCp{',ii,'}(1);\n'...   % slope parameter
385-          't0 = BCp{',ii,'}(2);\n'...  % time parameter
386-          'x0 = BCp{',ii,'}(3);\n'...  % init disp parameter
387-          'x = m/2*(t-t0).^2.*(t>t0)+x0;\n'... % Eq. (29)
388-          'v = m*(t-t0).*(t>t0);\n'...         % Eq. (30)
389-          'a = m*(t>t0);\n'];                  % Eq. (31)
390-        case 3 % acceleration base function
391-          m = params(1);  % get slope parameter
392-          t0 = params(2); % get time parameter
393-          x0 = params(3); % get initial displacement parameter
394-          v0 = params(4); % get initial velocity parameter
395-          x = m/6*(t-t0).^3.*(t>t0)+v0*(t-t0)+x0; % Eq. (32)
396-          v = m/2*(t-t0).^2.*(t>t0)+v0;           % Eq. (33)
397-          a = m*(t-t0).*(t>t0);                   % Eq. (34)
398-          bc_line = [
399-          'm = BCp{',ii,'}(1);\n'...   % slope parameter
400-          't0 = BCp{',ii,'}(2);\n'...  % time parameter
```

```
401-          'x0 = BCp{',ii,'}(3);\n'...  % init disp parameter
402-          'v0 = BCp{',ii,'}(4);\n'...  % init vel parameter
403-          'x = m/6*(t-t0).^3.*(t>t0)+v0(t-t0)+x0;\n'... % Eq. (32)
404-          'v = m/2*(t-t0).^2.*(t>t0)+v0;\n'...          % Eq. (33)
405-          'a = m*(t-t0).*(t>t0);\n'...                  % Eq. (34)
406-    end
   ⋮      ⋮
413-  end
```

The *params* variable that is passed into *Boundary_Library* is the array of parameter values for the boundary function. It is the same parameter array that the user supplies in the particular row of *bc_nodes* in *SystemSetup* (see Section 3.2.7).

For each *state* case, the first lines of strings in *bc_line* are commands that retrieve the parameter values for the relevant set of functions. The parameter values for all boundary functions are passed into the force function through the *BCp* input argument. *BCp* is a cell array where each cell contains an array of parameter values as directly copied from the rows of *bc_nodes* in *SystemSetup*. The variable *ii* in *Boundary_Library* contains an integer as a string that references the correct cell entry, so it is interjected into the *BCp* string declaration above.

The integer in parenthesis following the bracketed *ii* references the correct entry in the numeric array to retrieve the parameter value. Therefore, when the parameter array is entered into *bc_nodes* in *SystemSetup*, the array must be a vector of length at least 2, with the slope parameter in the first entry, and the time parameter in the second entry. For the displacement *state* case, these only these two parameters are required. For the velocity and acceleration *state* cases, additional variables are added to the array, so the user must always be aware of what parameters are required for a particular *model* and *state*. If the user desires the supplied parameters to be vectors rather than single numbers, then the parameter array can be supplied as its own cell array. The integers would then be enclosed in brackets rather than in parenthesis.

The latter string lines in *bc_line* write the boundary functions. The value of the displacement, velocity, and acceleration must be assigned to the variables *x*, *v*, and *a*, respectively. The variable *t* is the current time, and is supplied by the force function itself.

The user has finished implementing a new boundary function when the *bc_line* text output and the *x*, *v*, and *a* vectors are created for each *state* case.

3.5.3. Nonlinear Constitutive Functions

The procedure for adding new nonlinear constitutive functions to the ROMULIS library spans several scripts due to their complexity. In addition, based on the restrictions that ROMULIS places on nodes coupled with a nonlinear model (see Section 3.2.8), this procedure also assumes that the new function to implement is one-dimensional; that is, it couples one degree of freedom to only one other. To use functions that couple more than two degrees of freedom, it is better to upload a user-created nonlinear function (see Section 3.3.1) since more freedom is given as to how those degrees of freedom can be coupled.

As an example, this procedure implements the constitutive function, *Jenkins.m*, which has input arguments *u*, the relative displacement between degrees of freedom, *v*, the relative velocity,

and *Space*, a structure containing parameters as well as an internal state variable. Its output arguments are *F*, the force applied, and *Space*, which is the same input structure but with an updated internal state variable. As there are currently nine implemented constitutive functions in the library, this function is assigned function number 10.

The first script to modify from the toolbox is *InterfaceNodes*. This script performs a syntax check on *int_nodes* array from *SystemSetup*, and it also initializes the variables in the *Space* structure (see Table 7). As is done for external loads and boundary loads, a new case statement is added to the switch statement that checks the function number. This switch-case statement begins on line 272, and it contains the nine function cases (plus a null case) described in Appendix D. Case 10 is added for the new Jenkins model.

```
272-  switch Int_loc{j,3}(k)
   ⋮      ⋮
529-      case 10 % Jenkins element
530-        % Perform syntax checks first.
531-        % Check that the number of nodes in slave set is equal to the
532-        %   number of nodes in the master set.
533-        if num_nodes_m~=num_nodes_s
534-          error(['int_nodes row ',num2str(j),', direction ' ...
535-            num2str(k),'. Jenkins element requires as many slave ' ...
536-            'nodes as there are master nodes.']);
537-        end
538-        % Check that the user provided three parameters to int_nodes
539-        if length(Int_params{j,k})~=3
540-          error(['int_nodes row ',num2str(j),', direction ' ...
541-            num2str(k),'. Jenkins element requires 3 parameters.']);
542-        end
543-        % Initialize Space structure.
544-        % Grab function parameters.
545-        Space.k = Int_params{j,k}(1); % spring stiffness.
546-        Space.mu = Int_params{j,k}(2); % friction coefficient.
547-        Space.Fn = Int_params{j,k}(2); % normal force.
548-        % Initialize internal state.
549-        Space.xin = 0;
550-        % Add ROMULIS Space variables.
551-        Space.u = 0; % displacement input to function.
552-        Space.v = 0; % velocity input to function.
553-        Space.F = 0; % Force output
554-        for i = 1:num_space % run through each node pair in the set.
555-          Nspace = Nspace + 1; % increment SpaceCells index.
556-          UD_ind(i,k) = Nspace; % record index
557-          % Add Space to SpaceCells
558-          UD.SpaceCells{NSpace} = Space;
559-          % Add connectivity maps for master and slave nodesets
560-          connectivity = Int_loc{j,1}(i,:);
561-          body = G2L(connectivity(1),1);
562-          node_and_dof = cbmap{body}(G2L(connectivity(1),2),:);
563-          UD.SpaceCells{Nspace}.dof_map_m = [body,node_and_dof];
564-          body = G2L(connectivity(2),1);
565-          node_and_dof = cbmap{body}(G2L(connectivity(2),2),:);
566-          UD.SpaceCells{Nspace}.dof_map_s = [body,node_and_dof];
567-        end
   ⋮      ⋮
```

```
572-  end
```

Lines 530 through 542 perform syntax checks on user input. The *num_nodes_m* variable contains the number of nodes that are in the master nodeset that the user specifies for this function, and *num_nodes_s* contains the number of slave nodes. The *num_space* variable seen on line 554 contains the greater of these two values. The variable *j* indexes the row number from *int_nodes* where the function is specified, and *k* indexes the direction number.

Lines 545 through 549 initialize the *Space* variables that are applicable to the nonlinear function itself. The function parameters are obtained from the *Int_params* cell array, which has six columns (one for each surface-oriented direction) and the same number of rows as in *int_nodes*. Each cell contains the array of parameters applicable to the model of the corresponding row and direction in *int_nodes*.

Lines 551 through 567 are variables used within ROMULIS. These lines are required code.

After filling the above lines for the *InterfaceNodes* script, the next script to manipulate is *Write_Force_Func*. Starting on line 100 is a declaration of the *bols* structure, which contains eight fields holding Boolean values. Each Boolean field corresponds with one of the currently implemented nonlinear function, and the value of the Boolean determines whether ROMULIS writes the actual constitutive function as a subfunction to the parent force function. Here, another arbitrarily-named Boolean field is added for the Jenkins function, with value set to zero.

```
107-  bols.Jenk = 0;
```

Scrolling down to the very end of Write_Force_Func, an if-statement is added just before the *fclose* function that will append the nonlinear function to the parent force function when *bols.Jenk* = 1. If the user has already typed up the script in a separate function, the following lines are sufficient to copy the function.

```
321-  if bols.Jenk
322-      % type the directory and function name in Ffile
323-      Ffile = 'C:/Users/user/Documents/NonlinFunc/Jenkins.m';
324-      textout = typeFile(Ffile);
325-      fprintf(fileID,[textout,'\n\n']);
326-  end
```

The final script to manipulate is *InterfaceDeclaration*. Here, the commands that call the nonlinear function are typed to the force function. As before, a switch-case statement that checks the function number begins on line 10. Case 10 is added for the Jenkins function.

```
 10-  switch Model
   :    :
159-    case 10
160-      bols.Jenk = 1;
161-      rel_disp = 1;
162-      rel_vel = 1;
163-      ind = num2str(UD_ind(pairind,jj));
```

```
164-      F_direc = [
165-        'Space = UD.SpaceCells{',ind,'};\n'...
166-        '[f,Space] = Jenkins(ur(',direc,'),',vr(',direc,'),Space);\n'...
167-        'Fr(',direc,') = f;\n'...
168-        'UD.SpaceCells{',ind,'} = Space;\n'...
169-        'UD.SpaceCells{',ind,'}.u = ur(',direc,');\n'...
170-        'UD.SpaceCells{',ind,'}.v = vr(',direc,');\n'...
171-        'UD.SpaceCells{',ind,'}.F = f;\n'];
   ⋮    ⋮
175-  end
```

Line 160 in the example above switches the value of the *Jenkins* function Boolean to '1' so that Write_Force_Func appends the *Jenkins* function to the end of the force function. Lines 161 and 162 change the value of two additional Boolean variables. These Boolean variables let the parser know to include commands that retrieve the state variables required of the nonlinear function. There are three such Boolean variables: *rel_disp*, *rel_pos*, and *rel_vel*. The *rel_disp* variable sets retrieves the relative displacement between the master and slave nodes, while *rel_pos* retrieves the relative position. The *rel_vel* Boolean retrieves relative velocity. By default, the value of these Booleans is set to zero, and if any of these state variables are required, the user need only change their value to '1'.

The distinction between relative position and relative displacement is that relative position measures the difference between the two degrees of freedom in space. That is, relative position is relative displacement plus the nominal coordinate location in space. Mathematically, relative position is $\mathbf{z}$ in Eq. (16), whereas relative displacement is just $\mathbf{u}_2 - \mathbf{u}_1$. Given the nature of contacting interfaces, most constitutive models would require relative position over relative displacement.

Line 163 retrieves the index of the *SpaceCells* cell associated with the nonlinear function. The text output is stored in the *F_direc* variable. Note that each line of string commands terminates with a newline character (\n) so that MATLAB knows to begin the next set of strings on a new line in the force function.

The command in line 165, retrieves the function's *Space* structure from *UD* structure passed into the force function. On line 166, the function itself is called. The inputs and outputs must be exactly as required by the nonlinear function submitted in the *Write_Force_Func* script. If relative displacement is required by the function, the variable *ur* is supplied by the force function. If relative position is required, the variable *zr* is supplied, and the variable *vr* is supplied for relative velocity. *ur*, *zr*, and *vr* are the rotated, relative state vectors with three elements representing the normal, and two tangent degrees of freedom. The *direc* variable holds the index of the correct degree of freedom required of the function.

Line 167 updates the rotated force vector, which will be re-rotated back to the global frame, and line 168 updates the *Space* cell in *UD.* Lastly, Lines 169 to 171 are optional for storing the state inputs and force outputs for recording in the user output function (see Section 3.3.2).

By following this procedure in its entirety, the user will have successfully implemented a new nonlinear constitutive model in the ROMULIS library.

4. A DEVELOPER'S INTRODUCTION TO THE TOOLBOX SCRIPTS

ROMULIS consists of twenty MATLAB scripts that are written to process user input regarding a structural system towards the ultimate goal of producing a transient response. On the whole, the scripts could be divided into three major task groups performed in sequence based on their intended order of use.

1) System setup – scripts include but not limited to
   - *UploadData.m*
   - *SystemSetup.m*
   - *Initialization.m*
   - *SerialExecuter.m*
2) Integrator preparation – scripts include but not limited to
   - *InterfaceDynamics.m*
   - *Write_Force_Func.m*
   - *Boundary_Library.m*
   - *Load_Library.m*
   - *Interface_Library.m*
   - *typeFile.m*
3) Numerical integration – scripts include
   - *IMEX_2a_Romulis.m*
   - *RKNG4.m*
   - *RKNG5.m*

The most important of these tasks is numerical integration, which uses a second order (in terms of the type of equation solved), fifth-order accurate, implicit-explicit (IMEX) adaptive solver to calculate the solution to the system equations of motion defined in Eq (12). The solver comprises three scripts: the *IMEX_2a_Romulis* function as the core function, and the *RKNG4* and *RKNG5* functions as supporting functions.

The *IMEX_2a_Romulis* function works similarly to MATLAB's own suite of ordinary differential equation solvers (e.g. *ode45*) in that it accepts a definition of the relationship between state variables and their derivatives in addition to a specification of time interval and initial condition parameters. *IMEX_2a_Romulis* differs in its ability to solve a stiff, nonlinear system of equations more efficiently than the built-in solvers in MATLAB. The simulator calls this function within the *InterfaceDynamics* function on line 247 as follows:

```
200-  [eta,eta2,t,f_out,UD]=IMEX_2a_Romulis(
      @(t,q,qv,UD)Force_Func(t,q,qv,Mats,BC_params,F_params,Int_params,UD),
      M_aa,C_aa,K_aa,params.tini,params.simTime,params,UD_init );
```

Line 200 shows that *IMEX_2a_Romulis* accepts seven inputs as described in Table 10. After completing numerical integration, *IMEX_2a_Romulis* outputs the five arguments listed in Table 11.

Table 10: Descriptions on the input arguments for the *IMEX_2a_Romulis* function.

Argument	Description
@Force_Func	A handle referencing a function that dictates all loads applied on the active coordinates. This function defines the entire right-hand side of Eq. (12). The handle defines time (*t*), state (*q*, *qv*), and output function data (*UD*) variables as implied arguments to the function, but the function itself requires four more arguments, *Mats*, *BC_params*, *F_params*, and *Int_params*, which define various load function vectors and parameters.
M_aa	The system mass matrix ($\tilde{\mathbf{M}}_{aa}$ in Eq. (12)).
C_aa	The system damping matrix ($\tilde{\mathbf{C}}_{aa}$ in Eq. (12)).
K_aa	The system stiffness matrix ($\tilde{\mathbf{K}}_{aa}$ in Eq. (12)).
params.tini	Simulation starting time.
params.simTime	Simulation ending time.
params	A structure containing variables to define the time step properties and supplementary functions to the IMEX integrator.
UD_init	Initialization structure for *UD* (see section 3.3.2 for more information).

Table 11: Descriptions on the output arguments from the *IMEX_2a_Romulis* function.

Argument	Description
eta	The displacement time history solution to the system ($\mathbf{p}_a$ in Eq. (12))..
eta2	The velocity time history solution ($\dot{\mathbf{p}}_a$ in Eq. (12)).
t	The time vector associated with the entries in *eta* and *eta2*.
f_out	The time history of applied loads on the system. This is the value of the right-hand side of Eq. (12) at each time step.
UD	A structure containing variables calculated in a user output function (see section 3.3.2 for more information).

Under the objective of retrieving these output arguments, the remaining MATLAB scripts in the toolbox are dedicated to collecting user input on the nature of the system, and processing the input to write the *Force_Func* function and develop the *M_aa*, *C_aa, K_aa*, and *params* input arguments. While forming the system, ROMULIS creates several data files that record the inputs so that the user can reference the setup of the system without having to restart from the beginning. The flow chart in Fig. 3 illustrates the sequence of the data files and the prominent toolbox scripts.

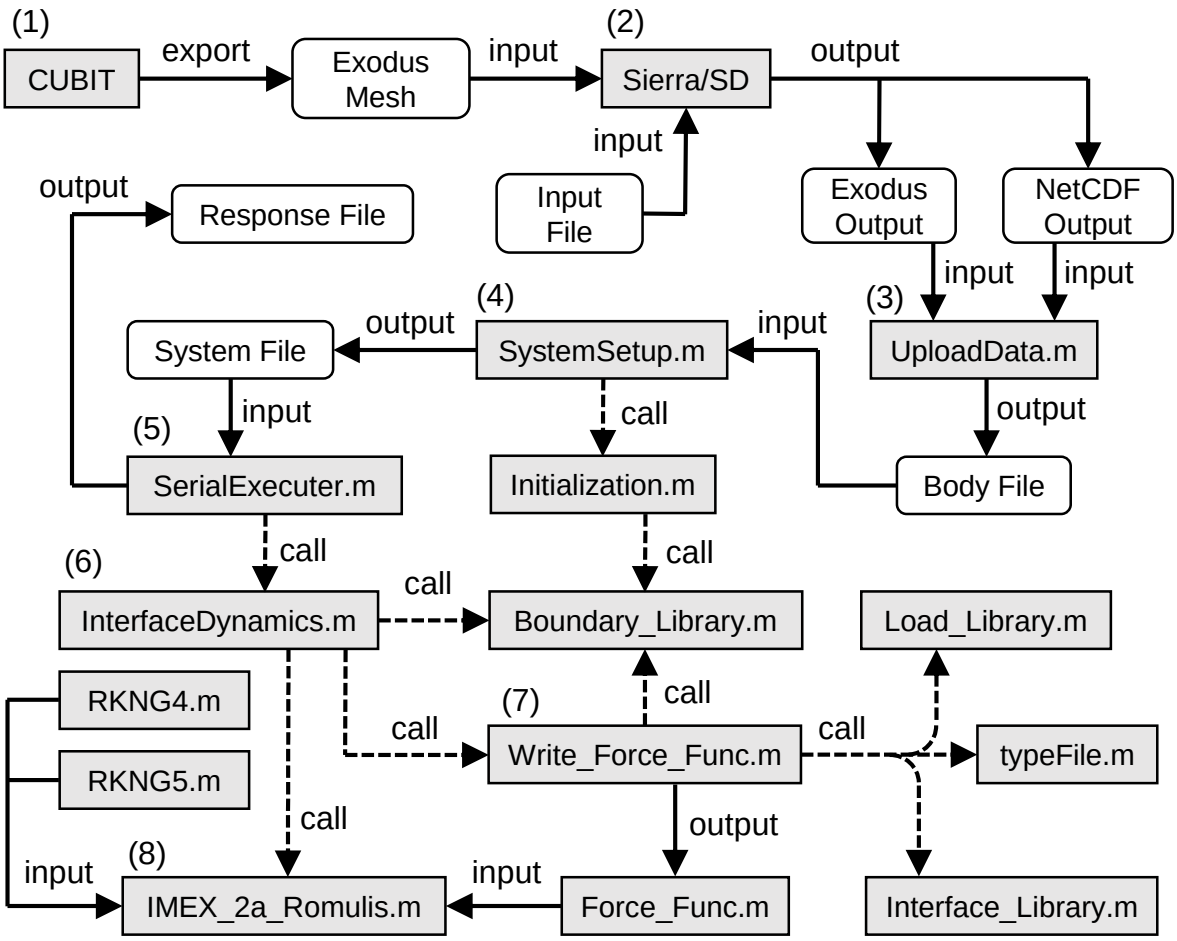

Figure 4: Complete flow chart of the programs used and the files accessed throughout ROMULIS. The shaded boxes indicate software packages or ROMULIS toolbox scripts, and the white boxes are data files. All programs and files are utilized in sequence as numbered in parenthesis.

The *UploadData*, *SystemSetup*, and *SerialExecuter* scripts are the only scripts in the toolbox with which the user interacts directly, and are detailed in Sections 3.1, 3.2, and 3.4, respectively. When the user runs the *SystemSetup* script, the loads and other inputs on the system are processed in the *Initialization* script. The *Initialization* script primarily maps the local body-nodeset-direction specifications into global degrees of freedom for all loads specified by the user, but it also calculates the global state vectors for the initial condition. The *Initialization* script in turn calls the *Boundary_Library* function in order to calculate the initial state for all constrained degrees of freedom.

When *SerialExecuter* is run, it launches the simulation by first calling on the *InterfaceDynamics* function. The main task for *InterfaceDynamics* is to process the data in the system file to produce the variables required for the IMEX integrator. It first loads each of the individual body files to gather the substructure matrices, and then assembles the matrices into global system matrices. *InterfaceDynamics* then partitions the global matrices into active degrees

of freedom (DOF) and constrained DOF, and separates the constrained DOF for boundary conditions. The active DOF are then input into the IMEX integrator.

InterfaceDynamics then uses the loading and nonlinear model data provided by the user in SystemSetup to write a new force function input to the IMEX integrator. The force function, *Force_Func.m*, is created through metaprogramming techniques, so it contains only the commands that are absolutely necessary to build the force vector. This is to improve the efficiency of *Force_Func* so that it executes as quickly and saves on computational cost.

The hub of the metaprogramming scripts is *Write_Force_Func.m*, to which the *InterfaceDynamics* function passes the load and nonlinear model data. *Write_Force_Func* examines the data and calls the *Load_Library*, *Boundary_Library*, *Interface_Library,* and *typeFile* functions to help write the force function. Each of these subfunctions contains an elaborate collection of nested control statements that lead to a certain text output depending on what load options that the user specifies for the system. The *Load_Library* function specializes in text outputs for external loads, *Boundary_Library* specializes in prescribed boundary motion, and *Interface_Library* focuses on the nonlinear constitutive functions. The *typeFile* function copies user-created custom constitutive functions (see Section 3.3.1) into *Force_Func*. When *Write_Force_Func* finishes writing a new *Force_Func* file, *InterfaceDynamics* forcibly reloads the MATLAB search path so that the runtime recognizes the changes made to *Force_Func*.

With the force function updated and the input variables to *IMEX_2a* created, *InterfaceDynamics* finally runs the time integrator to retrieve the response for the active DOF. Then the response is transformed back to the generalized coordinates, and Boundary_Library is called one last time to create response vectors for the constrained coordinates. The final response vectors are sent back to SerialExecuter for output into a response file.

All toolbox scripts contain documentation of their own to explain its coding process, so a new developer is encouraged to follow along with the commands in each script to learn how ROMULIS works.

REFERENCES

[1] Segalman, D. J. (2007). "Model reduction of systems with localized nonlinearities." *Journal of Computational and Nonlinear Dynamics* 2(3), pp. 249-266. doi: 10.1115/1.2727495

[2] Edwards, H. C. (2002). "Sierra framework version 3: core services theory and design." SAND2002-3616. Sandia National Laboratories: Albuquerque, NM.

[3] Segalman, D. J. (2005). "A four-parameter Iwan model for lap-type joints." *Journal of Applied Mechanics* 72(5), pp. 752-760. doi: 10.1115/1.1989354

[4] Brake, M. R. W. (2013). "IMEX-a: an adaptive, fifth order implicit-explicit integration scheme," SAND2013-4299. Sandia National Laboratories: Albuquerque, New Mexico.

[5] Brake, M. R. W. (2015). "An analytical elastic plastic contact model with strain hardening and frictional effects for normal and oblique impacts." *International Journal of Solids and Structures* 62, pp. 104-123. doi: 10.1016/j.ijsolstr.2015.02.018

[6] Brake, M. R. W. "A reduced Iwan model that includes pinning for bolted joint mechanics." Submitted to *Nonlinear Dynamics*.

[7] Bampton, M. C. C., and Craig, R. R, "Coupling of substructures for dynamic analyses," *AIAA Journal*, 6(7), 1968, pp. 1313-1319. doi: 10.2514/3.45435

[8] de Klerk, D., Rixen, D. J., and Voormeeren S. N. (2008). "General framework for dynamic substructuring: history, review and classification techniques." *AIAA Journal* 46(5), pp. 1169-1181. doi: 10.2514/1.33274

[9] Craig, R. R., and Kurdila, A. J., *Fundamentals of Structural Dynamics*, 2nd ed., Wiley: Hoboken, New Jersey, 2006.

[10] "Sierra Structural Dynamics–User's Notes," Sandia National Laboratories: Albuquerque, New Mexico, 2015, SAND2015-9132.

[11] Cook, R. D., Malkus, D. S., Plesha, M. E., and Witt, R. J., *Concepts and Applications of Finite Element Analysis*, 4th ed., Wiley: New York, 2002.

[12] Johnson, R., and Brake, M. R. W., "Stability of a Second-Order Adaptive Implicit-Explicit (IMEX) Integration Scheme," *ZAMM – Journal of Applied Mathematics and Mechanics,* under review.

APPENDIX A: MATLAB VARIABLES REQUIRED IN A BODY FILE

To qualify as a body file, the following variables must be named in a MATLAB binary file (extension **.mat*). Each variable should contain the data type as described. Beware that the variables are case-sensitive.

M

An array of double values. The linear substructure mass matrix.

C

An array of double values. The substructure damping matrix. The order of the degrees of freedom must be consistent with M.

K

An array of double values. The substructure stiffness matrix. The order of the degrees of freedom must be consistent with M.

cbmap

An array of double or integer values. For Craig-Bampton reduced substructures, this array maps the fixed-interface mode and constraint mode coordinates to the node number and coordinate direction of their corresponding boundary degree of freedom. For all other systems, cbmap maps all degrees of freedom to their node number and coordinate direction. Each row in cbmap references a particular degree of freedom (or modal coordinate in Craig-Bampton systems). The first column lists the associated node number for each degree of freedom, and the second column lists its coordinate direction. The value '1' refers to the x-translation direction, '2' refers to the y-translation direction, '3' is z-translation, '4' is x-rotation, '5' is y-rotation, and '6' is z-rotation. Note that for Craig-Bampton systems, the node number and coordinate direction are both zero for the fixed-interface modal coordinates. The order of the rows must be consistent with the order of the degrees of freedom in M, C, and K. Hence, there are as many rows in cbmap as there are columns in M, C, and K.

Example:

```
cbmap = [0 0; 0 0; 0 0;
         9 1; 9 2; 9 3; 9 4; 9 5;9 6;
         212 1; 212 2; 212 3;
         33 1; 33 2; 33 3];
```

G

An array of double values. For Craig-Bampton reduced systems, this is matrix $G_b^{(s)}$ from Eq. (13). The rows are ordered to match the order of degrees of freedom in the rows of cbmap. For all other systems, G is the identity matrix having the same size as M.

OutMap

An array of double or integer values. A column vector of node numbers associated with the nodes on which an external load is applied, or for which the response output is desired. If no input or output nodes are desired in the present substructure, then this array is left empty.

Example:

```
OutMap = [4;5];
```

OTM

An array of double values. In a Craig-Bampton system, OTM are the rows of the Craig-Bampton transformation matrix ($\boldsymbol{\Psi}_{CB}$ in Eq. (6)) associated with the nodes listed in OutMap. Each node is associated with six rows drawn from the transformation matrix, one for each of the three translation and three rotation degrees of freedom. The rows for each node are ordered as x-translation, y-translation, z-translation, x-rotation, y-rotation, and z-rotation, and each block of six rows are ordered in block rows consistent with the order of the node numbers in OutMap. The columns are ordered consistently with the order of degrees of freedom in the rows of cbmap. There must always be six rows associated with a node. If a node does not have certain degrees of freedom (particularly rotation for solid elements), then that row contains all zeros. For all other systems, each row in OTM is an array of zeros except for a one in the column associated with the same degree of freedom in cbmap. If OutMap is an empty array, then OTM is also an empty array.

Nodes

An array of double or integer values. A vertical vector of all node numbers used in cbmap and OutMap. The order of nodes is not important, but node numbers should not be repeated.

Example:

```
Nodes = [9;212;33;4;5];
```

coords

An array of double values. Lists the x, y, and z coordinate locations of the nodes in Nodes. Each row in coords contains three columns. The first column gives the x coordinate location, the second column gives the y-coordinate location, and the third column is the z coordinate location. The order of the rows is consistent with the corresponding node order in Nodes.

NSnodes

A one-dimensional cell array where each cell contains an array of double or integer values. Each cell corresponds to a unique nodeset specified for this substructure, and each cell contains a vertical vector of the node numbers associated with that nodeset. The order of nodes in each nodeset is not important, and the cells need not follow a particular order either.

Example:

```
NSnodes = {[212;33],[9],[4;5]};
```

NSnums

An array of double or integer values. A vertical vector of numbers assigned to each nodeset. The order of nodeset numbers must be consistent with the order of the nodesets represented by the corresponding cell in NSnodes.

Example:

```
NSnums = [10;1,3];
```

APPENDIX B: THE LIBRARY OF EXTERNAL LOAD FUNCTIONS

This appendix defines the external force functions currently implemented in ROMULIS. External loads are specified on a system by filling in the *f_nodes* cell array in *SystemSetup* according to the script syntax described in Section 3.2.5. Each row in *f_nodes* is specified as shown in Fig. B. The implemented function numbers and associated parameter sets are summarized in Table B, and detailed thereafter.

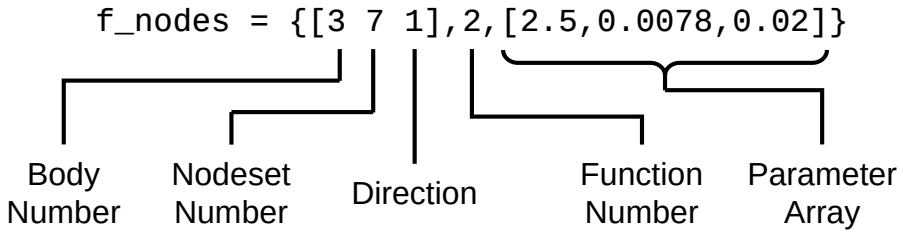

Figure B: Script syntax for the *f_nodes* variable with syntax labels.

Table B: Summary of implemented external force functions.

Function Number	Function Description	Parameter Array
0	Zero Function	[ ]
1	Step (Constant) Function	$[A,t_0]$
2	Haversine Impulse Function	$[A,T,t_0]$
3	Sinusoid Function with Smooth Start	$[A,T,t_0]$
4	Sinusoid Function	$[A,T,t_\phi,t_0]$
5	Step Function with Smooth Ramp-Up	$[A,T,t_0]$
6	Sine Sweep Function (Exponential Frequency Sweep)	$[A,f_0,c,r,t_0]$

B.0. The Zero Function

A zero external load function may be applied on a nodeset by specifying a value of '0,' or any number not listed in Table B, as the function number in *f_nodes*. This applies a zero-load force on the specified nodes for the duration of the simulation in accordance with the time function,

$$f(t) = 0\,. \tag{B-1}$$

The zero function does not require a parameter set, so the parameter array may be listed as an empty numeric array. While not particularly effective during a simulation, the zero function is more useful as a placeholder function while building the *f_nodes* cell array. In practice, the user may "turn off" any of the other implemented functions for a simulation by changing the function number to '0.' This allows the parameter array and the nodeset specification for the original function to be kept since ROMULIS ignores any row in *f_nodes* characterized with a zero function.

B.1. The Step Function

The step function is based on the Heaviside step function where the applied load is described by

$$f(t) = \begin{cases} 0, & t < t_0, \\ A, & t \geq t_0. \end{cases} \tag{B-2}$$

where A is the step amplitude, and t_0 is the time at which the step occurs. Function number 1 specifies the step function in *f_nodes*, and its parameter set is the numeric array $[A, t_0]$. The step function may also be used to apply a constant load on the nodes in a nodeset by setting the value of t_0 equal to the value for the simulation start time.

B.2. The Haversine Impulse Function

Function number 2 calls a haversine impulse function, which is useful for creating smooth impact loads. The form of the haversine impulse load is essentially a vertically-shifted cosine signal taken over one period of the wave. The equation ROMULIS uses is

$$f(t) = \begin{cases} \dfrac{A}{2}\left(1 - \cos\left(\dfrac{2\pi}{T}(t - t_0)\right)\right), & t_0 < t < T + t_0 \\ 0, & \text{otherwise}, \end{cases} \tag{B-3}$$

with the peak force value the impulse reaches A, the time duration of the impulse T, and the time in the simulation where the impulse starts to build, t_0. These three values also make up the parameter numeric array for this function, and are ordered as $[A, T, t_0]$.

B.3. The Sinusoid Function with a Smooth Start

The smooth-start sine wave distinguishes itself from the regular sine wave in that the first quarter-period of the waveform is replaced with a half-haversine. In this way, the slope of the force function ramps up from zero in the beginning as opposed to the slope abruptly changing as is done in the regular sine wave. If A is the waveform amplitude, T is the waveform period, and t_0 is the time in the simulation at which the waveform begins, then the equations take the form of

$$f(t) = \begin{cases} 0, & t \leq t_0, \\ \dfrac{A}{2}\left(1 - \cos\left(\dfrac{4\pi}{T}(t - t_0)\right)\right), & t_0 < t < \dfrac{T}{4} + t_0, \\ A\sin\left(\dfrac{2\pi}{T}(t - t_0)\right), & t \geq \dfrac{T}{4} + t_0. \end{cases} \tag{B-4}$$

This function may be called in *f_nodes* by assigning the function number 3, and specifying the parameter array $[A, T, t_0]$.

B.4. The Sinusoid Function

Specifying function number 4 in *f_nodes* activates a sine forcing function. The sine wave may be characterized with the parameter array $[A, T, t_\phi, t_0]$, where A is the amplitude of the waveform, T is the waveform period, and t_ϕ is a time offset from zero that defines the phase lead for the excitation signal. The t_0 parameter defines the time in the simulator that the signal "turns on," and should not be mistaken with the time that the signal starts at zero and ramps up to its first peak. Hence, the user should beware of activating the waveform at a point that creates a jump discontinuity in the forcing signal. With these parameters, the analytical form of the signal is

$$f(t) = \begin{cases} 0, & t < t_0, \\ A\sin\left(\dfrac{2\pi}{T}(t - t_\phi)\right), & t \geq t_0. \end{cases} \tag{B-5}$$

B.5. The Step Function with Smooth Ramp-Up

The step function with a smooth ramp-up includes a half-haversine function that eliminates the jump discontinuity in the original Heaviside function. The smooth ramp up helps alleviate any transient effects that would otherwise occur with the discontinuity present. The time function for the force takes the form

$$f(t) = \begin{cases} 0, & t \leq t_0 \\ \dfrac{A}{2}\left(1 - \cos\left(\dfrac{\pi}{T}(t - t_0)\right)\right), & t_0 < t < T + t_0, \\ A & t \geq T + t_0, \end{cases} \tag{B-6}$$

where A is the step amplitude, T is the time allotted for the ramp-up, and t_0 is the time in the simulation where the ramp-up period begins. Specifying function number 5 in *f_nodes* calls the step function with a smooth ramp-up, and its parameter set is the array $[A, T, t_0]$.

B.6. The Swept Sine Function with Exponential Frequency Sweeping

The swept sine function implemented in ROMULIS creates a forcing waveform signal with constant amplitude and exponentially increasing (or decreasing) frequency. Specifying function number 6 in *f_nodes* calls the following function for the force,

$$f(t) = \begin{cases} 0, & t \leq t_0, \\ A\sin\left(\dfrac{2\pi f_0}{cr}\left(\exp\left(cr\left(t - t_0\right)\right) - 1\right)\left(t - t_0\right)\right), & t > t_0. \end{cases} \tag{B-7}$$

The parameter set, ordered in *f_nodes* as $[A, f_0, c, r, t_0]$, contains several variables. The variable A is the force amplitude for the signal, f_0 is the frequency at the start of the sweep, and c is the logarithmic base coefficient whose value depends on the units for the rate of frequency sweep (e.g. log(2) for octave rate, log(10)/20 for decibel rate, and 1 for exponential rate). The variable r is the rate of frequency sweep, which is negative for a downward frequency sweep, and t_0 is the time in the simulator that the sweep signal begins.

APPENDIX C: THE LIBRARY OF BOUNDARY MOTION FUNCTIONS

This appendix defines the prescribed boundary functions currently implemented in ROMULIS. External loads are specified on a system by filling in the *bc_nodes* cell array in *SystemSetup* according to the script syntax described in Section 3.2.6. Each row in *bc_nodes* is specified as shown in Fig. C. The implemented function numbers and associated parameter sets are summarized in Table C. In the equations that follow, the variable x refers to prescribed displacement, v is prescribed velocity, and a is prescribed acceleration.

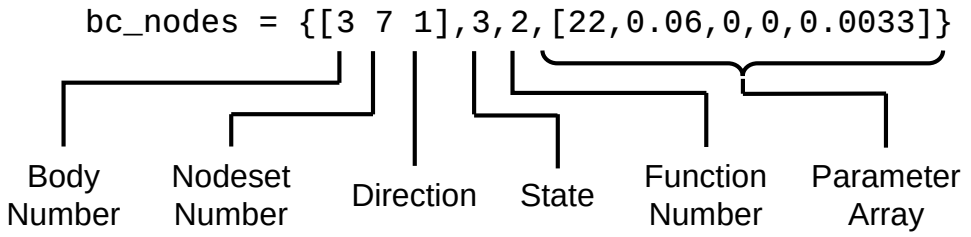

Figure C: Script syntax for the *bc_nodes* variable with syntax labels.

Table C: Summary of implemented boundary motion functions.

Function Number	Function Description	State	Parameter Array
0	Zero Function	1	[ ]
		2	$[x_0]$
		3	$[v_0,x_0]$
1	Haversine Impulse	1	$[A,T,t_0]$
		2	$[A,T,t_0,x_0]$
		3	$[A,T,t_0,v_0,x_0]$
2	Sinusoid with Smooth Start	1	$[A,T,t_0]$
		2	$[A,T,t_0,x_0]$
		3	$[A,T,t_0,v_0,x_0]$
3	Sinusoid	1	$[A,T,t_\phi,t_0]$
		2	$[A,T,t_\phi,t_0,x_0]$
		3	$[A,T,t_\phi,t_0,v_0,x_0]$
4	Fourier Coefficients	1	$\{\omega,X,t_{end}\}$
		2	$\{\omega,X,x_0,t_{end}\}$
		3	$\{\omega,X,v_0,x_0,t_{end}\}$
5	Discrete Time Input	1, 2, or 3	$\{t,u,v,a\}$

C.0. The Zero Function

The zero function (function number 0) for displacement applies fixed boundary conditions for the specified nodeset. The boundary state equations for the zero function in displacement (state value 1) are

$$x(t) = 0, \tag{C-1}$$
$$v(t) = 0, \tag{C-2}$$
$$a(t) = 0, \tag{C-3}$$

No parameters are necessary, so the parameter array may be left empty.

The zero function in velocity applies a constant displacement value on the specified set of nodes. The state equations for the velocity zero function (state value 2) are

$$x(t) = x_0, \tag{C-4}$$
$$v(t) = 0, \tag{C-5}$$
$$a(t) = 0, \tag{C-6}$$

The parameter array requires only x_0, which defines a the displacement value at time zero in the simulation.

The zero function in acceleration applies a constant velocity on the specified set of nodes. The state equations for the acceleration zero function (state value 3) are

$$x(t) = v_0 t + x_0, \tag{C-7}$$
$$v(t) = v_0, \tag{C-8}$$
$$a(t) = 0, \tag{C-9}$$

The parameter array requires values for v_0, the velocity and time zero in the simulation, and x_0, the displacement at time zero.

C.1. The Haversine Impulse Function

A haversine impulse function (function number 1) prescribes a smooth, peak-type motion behavior in the form of a vertically shifted cosine wave applied over one period of the wave. In displacement (state value 1), the state equations are

$$x(t) = \begin{cases} \frac{A}{2}\left(1 - \cos\left(\frac{2\pi}{T}\left(t - t_0\right)\right)\right), & t_0 < t < T + t_0, \\ 0, & \text{otherwise}, \end{cases} \tag{C-10}$$

$$v(t) = \begin{cases} \frac{A\pi}{T}\sin\left(\frac{2\pi}{T}\left(t - t_0\right)\right), & t_0 < t < T + t_0, \\ 0, & \text{otherwise}, \end{cases} \tag{C-11}$$

$$a(t) = \begin{cases} \frac{2A\pi^2}{T^2}\cos\left(\frac{2\pi}{T}\left(t - t_0\right)\right), & t_0 < t < T + t_0, \\ 0, & \text{otherwise}. \end{cases} \tag{C-12}$$

The displacement state parameter array requires *A*, the displacement peak value, *T*, the impulse duration, and t_0, the time in the simulation when the impulse starts.

The velocity haversine impulse equations are

$$x(t) = \begin{cases} x_0, & t \le t_0 \\ \frac{A}{2}\left(t - t_0 - \frac{T}{2\pi}\sin\left(\frac{2\pi}{T}\left(t - t_0\right)\right)\right) + x_0, & t_0 < t < T + t_0, \\ \frac{AT}{2} + x_0, & t \ge T + t_0, \end{cases} \tag{C-13}$$

$$v(t) = \begin{cases} \frac{A}{2}\left(1 - \cos\left(\frac{2\pi}{T}\left(t - t_0\right)\right)\right), & t_0 < t < T + t_0, \\ 0, & \text{otherwise}, \end{cases} \tag{C-14}$$

$$a(t) = \begin{cases} \frac{A\pi}{T}\sin\left(\frac{2\pi}{T}\left(t - t_0\right)\right), & t_0 < t < T + t_0, \\ 0, & \text{otherwise}. \end{cases} \tag{C-15}$$

The parameters for the velocity haversine function are the same as those for displacement, except that A represents the velocity peak value, and that the displacement value, x_0, at time 0 is added to the array.

The acceleration haversine impulse state equations are

$$x(t) = \begin{cases} v_0 t + x_0, & t \le t_0 \\ \frac{A}{2}\left(\frac{1}{2}\left(t - t_0\right)^2 - \frac{T^2}{4\pi^2}\left(\cos\left(\frac{2\pi}{T}\left(t - t_0\right)\right) - 1\right)\right) + v_0 t + x_0, & t_0 < t < T + t_0, \\ \frac{AT}{2}\left(-\frac{T}{2} + t - t_0\right) + v_0 t + x_0, & t \ge T + t_0, \end{cases} \tag{C-16}$$

$$v(t) = \begin{cases} v_0, & t \le t_0 \\ \frac{A}{2}\left(t - t_0 - \frac{T}{2\pi}\sin\left(\frac{2\pi}{T}\left(t - t_0\right)\right)\right) + v_0, & t_0 < t < T + t_0, \\ \frac{AT}{2} + v_0, & t \ge T + t_0, \end{cases} \tag{C-17}$$

$$a(t) = \begin{cases} \frac{A}{2}\left(1 - \cos\left(\frac{2\pi}{T}\left(t - t_0\right)\right)\right), & t_0 < t < T + t_0, \\ 0, & \text{otherwise}, \end{cases} \tag{C-18}$$

The parameters for the acceleration haversine function are the same as those for velocity, except that A represents the acceleration peak value, and that the velocity value, v_0, at time 0 is added to the array.

C.2. The Sinusoid Function with a Smooth Start

The smooth-start sinusoid function (function number 2) is identical to a regular sine wave function, except that the first quarter-period of the first wave is replaced with a half-haversine function. The displacement state equations are

$$x(t) = \begin{cases} 0, & t \le t_0, \\ \frac{A}{2}\left(1 - \cos\left(\frac{4\pi}{T}\left(t - t_0\right)\right)\right), & t_0 < t < \frac{T}{4} + t_0, \\ A\sin\left(\frac{2\pi}{T}\left(t - t_0\right)\right), & t \ge \frac{T}{4} + t_0, \end{cases} \tag{C-19}$$

$$v(t)=\begin{cases}0, & t\le t_0,\\ \frac{2\pi A}{T}\sin\left(\frac{4\pi}{T}\left(t-t_0\right)\right), & t_0<t<\frac{T}{4}+t_0,\\ \frac{2\pi A}{T}\cos\left(\frac{2\pi}{T}\left(t-t_0\right)\right), & t\ge\frac{T}{4}+t_0,\end{cases} \tag{C-20}$$

$$a(t)=\begin{cases}0, & t\le t_0,\\ \frac{8\pi^2 A}{T^2}\cos\left(\frac{4\pi}{T}\left(t-t_0\right)\right), & t_0<t<\frac{T}{4}+t_0,\\ \frac{4\pi^2 A}{T^2}\sin\left(\frac{2\pi}{T}\left(t-t_0\right)\right), & t\ge\frac{T}{4}+t_0,\end{cases} \tag{C-21}$$

The parameter array requires A, the displacement amplitude, T, the waveform period, and t_0, the time in the simulation that the waveform starts.

The velocity state equations are

$$x(t)=\begin{cases}x_0, & t\le t_0,\\ \frac{A}{2}\left(t-t_0-\frac{T}{4\pi}\sin\left(\frac{4\pi}{T}\left(t-t_0\right)\right)\right)+x_0, & t_0<t<\frac{T}{4}+t_0,\\ \frac{AT}{8}-\frac{AT}{2\pi}\cos\left(\frac{2\pi}{T}\left(t-t_0\right)\right)+x_0, & t\ge\frac{T}{4}+t_0,\end{cases} \tag{C-22}$$

$$v(t)=\begin{cases}0, & t\le t_0,\\ \frac{A}{2}\left(1-\cos\left(\frac{4\pi}{T}\left(t-t_0\right)\right)\right), & t_0<t<\frac{T}{4}+t_0,\\ A\sin\left(\frac{2\pi}{T}\left(t-t_0\right)\right), & t\ge\frac{T}{4}+t_0,\end{cases} \tag{C-23}$$

$$a(t)=\begin{cases}0, & t\le t_0,\\ \frac{2\pi A}{T}\sin\left(\frac{4\pi}{T}\left(t-t_0\right)\right), & t_0<t<\frac{T}{4}+t_0,\\ \frac{2\pi A}{T}\cos\left(\frac{2\pi}{T}\left(t-t_0\right)\right), & t\ge\frac{T}{4}+t_0,\end{cases} \tag{C-24}$$

The parameters for the velocity smooth sine function are the same as those for displacement, except that A represents the velocity amplitude, and that the displacement value, x_0, at $t = 0$ is added to the array.

The acceleration state equations are

$$x(t)=\begin{cases}v_0t+x_0, & t\le t_0,\\ \frac{A}{2}\left(\frac{1}{2}\left(t-t_0\right)-\frac{T^2}{16\pi^2}\left(\cos\left(\frac{4\pi}{T}\left(t-t_0\right)\right)-1\right)\right)+v_0t+x_0, & t_0<t<\frac{T}{4}+t_0,\\ -\frac{AT^2}{4\pi^2}\left(\sin\left(\frac{2\pi}{T}\left(t-t_0\right)\right)-\frac{2}{15}\right)+\frac{AT}{8}\left(t-t_0\right)+v_0t+x_0, & t\ge\frac{T}{4}+t_0,\end{cases} \tag{C-25}$$

$$v(t)=\begin{cases}v_0, & t\le t_0,\\ \frac{A}{2}\left(t-t_0-\frac{T}{4\pi}\sin\left(\frac{4\pi}{T}\left(t-t_0\right)\right)\right)+v_0, & t_0<t<\frac{T}{4}+t_0,\\ \frac{AT}{8}-\frac{AT}{2\pi}\cos\left(\frac{2\pi}{T}\left(t-t_0\right)\right)+v_0, & t\ge\frac{T}{4}+t_0,\end{cases} \tag{C-26}$$

$$a(t) = \begin{cases} 0, & t \le t_0, \\ \frac{A}{2}\left(1 - \cos\left(\frac{4\pi}{T}\left(t - t_0\right)\right)\right), & t_0 < t < \frac{T}{4} + t_0, \\ A\sin\left(\frac{2\pi}{T}\left(t - t_0\right)\right), & t \ge \frac{T}{4} + t_0, \end{cases} \tag{C-27}$$

The parameters for the acceleration smooth sine function are the same as those for velocity, except that A represents the acceleration amplitude, and that the velocity, v_0, at $t = 0$ is added to the array.

C.3. The Sinusoid Function

Boundary motion may be prescribed to follow a sine function (function number 3). If the fundamental sine function is attributed to the displacement (state number 1), the state equations are

$$x(t) = \begin{cases} 0, & t < t_0, \\ A\sin\left(\frac{2\pi}{T}\left(t - t_\phi\right)\right), & t \ge t_0, \end{cases} \tag{C-28}$$

$$v(t) = \begin{cases} 0, & t < t_0, \\ \frac{2\pi A}{T}\cos\left(\frac{2\pi}{T}\left(t - t_\phi\right)\right), & t \ge t_0, \end{cases} \tag{C-29}$$

$$a(t) = \begin{cases} 0, & t < t_0, \\ -\frac{4\pi^2 A}{T^2}\sin\left(\frac{2\pi}{T}\left(t - t_\phi\right)\right), & t \ge t_0. \end{cases} \tag{C-30}$$

The displacement sine function parameter array includes A, the displacement amplitude, T, the wave period, t_ϕ, the time shift for the phase offset, and t_0, the time that the signal "turns on." The user should beware of jump discontinuities that occur in the displacement signal when setting t_0 to a time that the signal is not a zero crossing.

Applying the sinusoid function to velocity (state number 2) invokes the equations

$$x(t) = \begin{cases} x_0, & t < t_0, \\ -\frac{AT}{2\pi}\cos\left(\frac{2\pi}{T}\left(t - t_\phi\right)\right) + x_0, & t \ge t_0, \end{cases} \tag{C-31}$$

$$v(t) = \begin{cases} 0, & t < t_0, \\ A\sin\left(\frac{2\pi}{T}\left(t - t_\phi\right)\right), & t \ge t_0, \end{cases} \tag{C-32}$$

$$a(t) = \begin{cases} 0, & t < t_0, \\ \frac{2\pi A}{T}\cos\left(\frac{2\pi}{T}\left(t - t_\phi\right)\right), & t \ge t_0. \end{cases} \tag{C-33}$$

The parameters in the velocity sine function are the same as those for displacement except that A represents the velocity amplitude. The x_0 parameter, representing the displacement at time 0, is added to the parameter array.

The acceleration sine function (state number "3") uses the equations

$$x(t) = \begin{cases} v_0 t + x_0, & t < t_0, \\ -\frac{AT^2}{4\pi^2}\sin\left(\frac{2\pi}{T}\left(t - t_\phi\right)\right) + v_0 t + x_0, & t \geq t_0, \end{cases} \tag{C-34}$$

$$v(t) = \begin{cases} v_0, & t < t_0, \\ -\frac{AT}{2\pi}\cos\left(\frac{2\pi}{T}\left(t - t_\phi\right)\right) + v_0, & t \geq t_0, \end{cases} \tag{C-35}$$

$$a(t) = \begin{cases} 0, & t < t_0, \\ A\sin\left(\frac{2\pi}{T}\left(t - t_\phi\right)\right), & t \geq t_0. \end{cases} \tag{C-36}$$

The parameters in the velocity sine function are the same as those for displacement except that A represents the velocity amplitude. The v_0 parameter, representing the velocity at time 0, is added to the parameter array.

C.4. A Time Function Constructed from Fourier Coefficients

The time histories for boundary motion may be constructed from a superposition of many sinusoidal functions using complex Fourier coefficients. Rather than passing in a numeric array of parameters, the user supplies a cell array that contains the vector of Fourier coefficients and their corresponding frequencies. If the Fourier coefficients option (function number 4) is selected for boundary displacements (state value 1), then parameter cell array to supply is $\{\omega, X, t_{\text{end}}\}$, where ω is the vector of radian frequencies, X is the vector of displacement complex Fourier coefficients, and t_{end} contains a single value denoting the time to "turn off" the signal. The displacement state equations follow the form

$$x(t) = \begin{cases} \sum_k \text{Re}\left[X_k \exp\left(-\text{i}\omega_k t\right)\right], & t \leq t_{\text{end}}, \\ 0, & t > t_{\text{end}}, \end{cases} \tag{C-37}$$

$$v(t) = \begin{cases} \sum_k \text{Re}\left[\text{i}\omega_k X_k \exp\left(-\text{i}\omega_k t\right)\right], & t \leq t_{\text{end}}, \\ 0, & t > t_{\text{end}}, \end{cases} \tag{C-38}$$

$$a(t) = \begin{cases} \sum_k \text{Re}\left[-\omega_k^2 X_k \exp\left(-\text{i}\omega_k t\right)\right], & t \leq t_{\text{end}}, \\ 0, & t > t_{\text{end}}, \end{cases} \tag{C-39}$$

where ω_k and X_k represent the k^{th} entry in their respective vectors, and i is the imaginary number.

Similarly, the velocity time histories are constructed through

$$x(t) = \begin{cases} x_0 + \sum_k \mathrm{Re}\left[\frac{X_k}{\mathrm{i}\omega_k}\exp(-\mathrm{i}\omega_k t)\right], & t \le t_{\text{end}}, \\ 0, & t > t_{\text{end}}, \end{cases} \tag{C-40}$$

$$v(t) = \begin{cases} \sum_k \mathrm{Re}\left[X_k \exp(-\mathrm{i}\omega_k t)\right], & t \le t_{\text{end}}, \\ 0, & t > t_{\text{end}}, \end{cases} \tag{C-41}$$

$$a(t) = \begin{cases} \sum_k \mathrm{Re}\left[\mathrm{i}\omega_k X_k \exp(-\mathrm{i}\omega_k t)\right], & t \le t_{\text{end}}, \\ 0, & t > t_{\text{end}}, \end{cases} \tag{C-42}$$

where X in this case is the vector of velocity Fourier coefficients. The parameter cell array also includes an additional parameter, x_0, which is the value of the displacement at time 0.

The acceleration Fourier coefficient time histories are constructed through

$$x(t) = \begin{cases} v_0 t + x_0 + \sum_k \mathrm{Re}\left[-\frac{X_k}{\omega_k^2}\exp(-\mathrm{i}\omega_k t)\right], & t \le t_{\text{end}}, \\ 0, & t > t_{\text{end}}, \end{cases} \tag{C-43}$$

$$v(t) = \begin{cases} v_0 + \sum_k \mathrm{Re}\left[\frac{X_k}{\mathrm{i}\omega_k}\exp(-\mathrm{i}\omega_k t)\right], & t \le t_{\text{end}}, \\ 0, & t > t_{\text{end}}, \end{cases} \tag{C-44}$$

$$a(t) = \begin{cases} \sum_k \mathrm{Re}\left[X_k \exp(-\mathrm{i}\omega_k t)\right], & t \le t_{\text{end}}, \\ 0, & t > t_{\text{end}}, \end{cases} \tag{C-45}$$

where X in this case is the vector of acceleration Fourier coefficients. The parameter cell array also includes another parameter, v_0, which is the value of the velocity at time 0.

If the user has time history data, where a waveform signal is represented in discrete time points, then the ω and X vectors can be created using MATLAB's *fft* function. Given that the signal, x, consists of an even N number time points with uniform time spacing Δt, then ω is created through

$$\omega = \frac{1}{N}\frac{2\pi}{\Delta t}\left(0:1:\tfrac{N}{2}\right). \tag{C-46}$$

The term $\left(0:1:\frac{N}{2}\right)$ is MATLAB syntax for a vector that starts with a value of zero, and each subsequent entry increments by 1 until the value $N/2$ is reached. The X vector may then be constructed with the following MATLAB commands.

```
1-  Xfft = fft(x);
2-  X = 2/N*Xfft(1:N/2+1);
```

Both the ω and X vectors can be passed into ROMULIS following the procedure near the end of Section 3.2.1.

C.5. A Time Function Constructed from Discrete Time History Data

The user may supply discrete time vectors describing the displacement, velocity, and acceleration for a set of nodes. This option is called using function number 5 and specifying the cell array $\{t,u,v,a\}$. The variable t is a discrete time vector, and the variables u, v, and a are the discrete vectors for the displacement, velocity, and acceleration, respectively whose values correspond to the time values in a. All vectors can be passed into ROMULIS following the procedure near the end of Section 3.2.1.

For the state value, either a 1, 2, or 3 may be specified. ROMULIS uses linear interpolation on the time vectors to extract the state values for a particular time step. The user should take care that the displacement, velocity, and acceleration vectors are appropriate derivatives and antiderivatives of each other to avoid convergence issues while the simulation computes.

APPENDIX D: THE LIBRARY OF FORCE-CONSTITUTIVE FUNCTIONS

All nonlinear consitutive functions currently implemented in ROMULIS are defined in this appendix. The functions are specified on a system by filling in the *int_nodes* cell array in *SystemSetup* according to the script syntax described in Section 3.2.7. Each row in *int_nodes* is specified as shown in Fig. D, except when passing in a user-created function. The implemented function numbers and associated parameter sets are summarized in Table D. Detailed descriptions of each function are given thereafter.

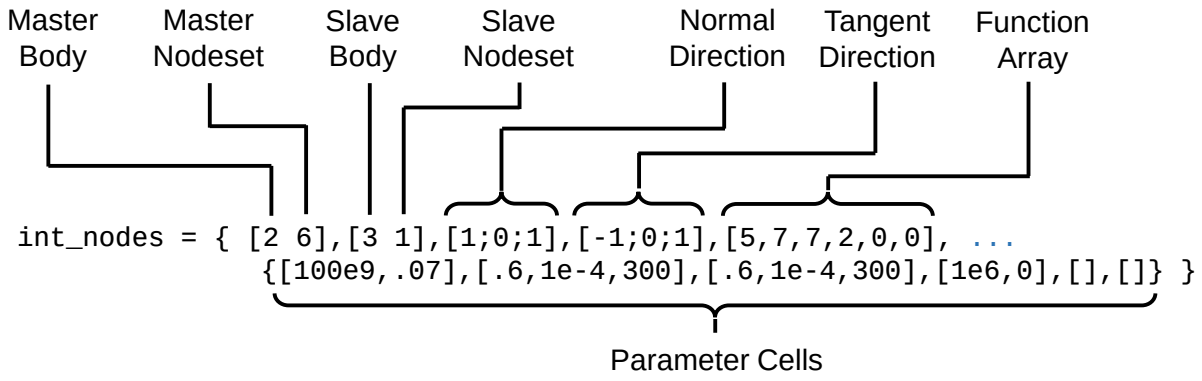

Figure D: Script syntax for the *int_nodes* variable with syntax labels.

Table D: Summary of implemented constitutive functions.

Function Number	Description	Parameter Array
0	Null element	[ ]
1	User-created function	See Section 3.2.7
2	Discrete spring and dashpot	[k,c]
3	Penalty spring	[k,L]
4	Hyperbolic penalty spring	[k,b,c,L]
5	Hertz contact element	[E,R]
6	Brake's mixed elastic-plastic contact with strain hardening	[E,R,Y,H_B,w,v]
7	Coulomb friction element	[μ,v_s,F_N]
8	Brake's RIPP joint element	[F_S,K_T,χ,β,θ,K_P,δ_P,m,C_v,v_s,s]
9	Segalman's four-parameter Iwan element	[F_S,K_T,χ,β]

D.0. The Null Element

Specifying a function number of 0 informs ROMULIS to ignore any coupling between the specified degrees of freedom. Null elements are useful as a placeholder function whereby the user may "turn off" another nonlinear element by changing the function number to "0." This allows the parameter array and the nodeset specification for the original element to be kept since

ROMULIS ignores those arrays for a null element. Otherwise, null elements do not require any parameters, so the parameter array may be left as an empty numeric array.

D.1. User-Created Force-Constitutive Function

Refer to Section 3.3.1 for information on creating a custom, force-constitutive function in MATLAB.

D.2. Discrete Spring and Dashpot

Degrees of freedom that are coupled with function number 2 are given a discrete linear spring and viscous damper. The constitutive equation is defined in the linear sense where the reaction force F applied oppositely on both degrees of freedom is

$$F = ku + cv, \tag{D-1}$$

where u and v are the displacement and velocity, respectively, of the slave degree of freedom relative to the master degree of freedom. The parameter k is the spring stiffness, and the parameter c is the damping coefficient. Both parameters are required in the parameter array $[k,c]$, and the user may turn off either the spring or the damper by setting its parameter value to zero.

D.3. Penalty Spring

Function number 3 adds a discrete penalty spring between two degrees of freedom. In a penalty model, the spring activates only when the position of the slave degree of freedom is negative relative to the position of the master degree of freedom plus some spring length, L. The force-constitutive equation is

$$F = \begin{cases} k(x - L), & x < L, \\ 0, & x \geq L, \end{cases} \tag{D-2}$$

for x is the absolute position of the slave degree of freedom in space relative to the absolute position of the master degree of freedom. The penalty spring element requires the parameter array $[k,L]$, where k is the penalty stiffness of the spring.

D.4. The Hyperbolic Penalty Spring

The hyperbolic penalty spring element is useful for representing force-displacement curves whose derivative can be well-fitted with the hyperbolic tangent function,

$$\frac{\mathrm{d}F}{\mathrm{d}x} = \frac{k}{2}\left(\tanh\left(b\left(x - c\right)\right) + 1\right). \tag{D-3}$$

The parameter k represents the far-field stiffness, b relates to the stiffness saturation rate, and c is a horizontal shift. After reversing the sign of the relative position x to apply for the penalty regime only, and adding a spring length parameter, L, Eq. (D-3) integrates to

$$F = \begin{cases} \frac{k}{2}\left(\frac{1}{b}\log\left(\frac{\cosh\left(b\left(-x+L-c\right)\right)}{\cosh\left(bc\right)}\right)-x+L\right), & x < L, \\ 0, & x \geq 0. \end{cases} \tag{D-4}$$

A good giveaway that a force-displacement curve follows a hyperbolic function is if the force curve is relatively flat near zero displacement, but then the slope gradually increases until it becomes virtually constant after a certain displacement. The hyperbolic penalty spring requires the parameter array [*k,b,c,L*].

D.5. Hertz Contact Element

Hertz contact defines elastic impact between two spherical, metallic bodies [1]. The parameter set requires two variables, [*E*,*R*], which are derived parameters. The effective elastic modulus, *E*, can be calculated from the Young's moduli, E_1 and E_2, and the Poisson's ratios, ν_1 and ν_2, of the two impacting materials through

$$E = \left(\frac{1-\nu_1^2}{E_1}+\frac{1-\nu_2^2}{E_2}\right)^{-1}. \tag{D-5}$$

Similarly, the effective radius of curvature, *R*, is derived from the radii of curvature, R_1 and R_2, of the two impacting surfaces by

$$R = \left(\frac{1}{R_1}+\frac{1}{R_2}\right)^{-1}. \tag{D-6}$$

With these two parameters, the constitutive equation for Hertz contact between spherical bodies is

$$F = \begin{cases} \frac{4}{3}\sqrt{R}E\delta^{3/2}, & \delta > 0, \\ 0, & \delta \leq 0, \end{cases} \tag{D-7}$$

Where δ is the penetration distance between the slave degree of freedom and the master degree of freedom.

D.6. Brake's Mixed Elastic-Plastic Contact with Strain Hardening

A more sophisticated contact model developed by Matthew Brake accounts for the energy lost due to plastic deformation occurring between two spherical, ductile bodies. Depending on the depth of penetration between the two contact points, the force applied on the points develops in the elastic regime until one of the two materials yields. After which, the force enters a mix of the elastic and plastic regimes, and accounts for strain-hardening effects. The user is directed to [2] for more information.

The required parameters for this function are [E,R,Y,H_B,w,v]. Here, E is the effective modulus calculated as in Eq. (D-5), and R is the effective radius of curvature calculated as in Eq. (D-6). The parameters Y, w, and v are the yield strength, strain hardening exponent, and Poisson's ratio, respectively, for the more compliant material. The effective Brinell hardness, H_B, is calculated from the Brinell hardness values, H_1 and H_2, for the two materials as

$$H_B = \left(\frac{2}{H_1} + \frac{2}{H_1} \right)^{-1}. \tag{D-8}$$

D.7. Coulomb Friction Element

Function number 7 couples the master and slave degrees of freedom with a Coulomb friction element. The constitutive equation for the element follows a dynamic model where the transition region from negative force to positive force near zero relative velocity is represented with a smooth, continuous function (as opposed to a jump discontinuity at zero velocity) as in

$$F = \mu \left| F_N \right| \tanh\left(5 \frac{v}{v_{sat}} \right). \tag{D-9}$$

The parameter μ is the friction coefficient, F_N is the normal force applied on the element, and v_{sat} is a velocity saturation parameter defining the bounds of the window around zero velocity where the transition from positive force to negative force begins. All three parameters, [μ,v_{sat},F_N] are required of the function.

D.8. Brake's RIPP Joint Element

Matthew Brake's reduced Iwan plus pinning (RIPP) element is used to model friction behavior in a bolted joint. The RIPP joint is based on a closed-form, reduced solution of the Iwan model under the assumption that the distribution of friction sliders after load reversal is a scaled version of the original distribution of sliders before reversal. The model also includes pinning stiffness due to the presence of the bolt. The user is referred to [3] for more information on the formulation behind the RIPP joint element.

The RIPP joint element requires eleven parameters, [F_S,K_T,χ,β,θ,K_P,δ_P,m,C_v,v_s,s]. F_S is the joint slip force, K_T is the joint tangent stiffness, χ relates to the power-law slope of energy dissipation in the joint, β relates to the ratio of the tangent stiffness and the stiffness just before the joint enters macroslip, and θ is the ratio of dynamic friction to static friction. The parameter, m, contains an integer that determines the type of Iwan model used for the joint; the value '1' activates Segalman's/Mignolet's model [4, 5], '2' activates an Iwan model with a uniform distribution of slider strengths, and '3' activates the Iwan-Stribeck model. K_P is the pinning stiffness, δ_P is the distance at which pinning engages. In the Iwan-Stribeck model, C_v is the viscous damping coefficient due to lubrication, v_s is the velocity scaling parameter for transition, and s is the exponent for transition. These last three parameters can be left as zero if not using the Iwan-Stribeck model.

D.9. Segalman's Four-Parameter Iwan Element

The predecessor to the RIPP joint is the four-parameter Iwan model formulated by Segalman [4]. This Iwan model is comprised of a statistical distribution of discrete friction sliders that approximate a power-law energy dissipation relationship from a lap joint in the micro-slip regime. In ROMULIS, the Iwan element requires four parameters, $[F_S,K_T,\chi,\beta]$, where F_S is the element's slip force at the onset of macroslip, K_T is the tangent stiffness of the joint in microslip, χ relates to the slope of the power-law, and β relates to the ratio of the tangent stiffness and the stiffness of the element just before macroslip.

References

[1] Johnson, K. L., *Contact Mechanics*, Cambridge University Press: Cambridge, United Kingdom, 1985.

[2] Brake, M. R. W., "An analytical elastic-plastic contact model with strain hardening and frictional effects for normal and oblique impacts," Sandia National Laboratories: Albuquerque, New Mexico, 2014, SAND2014-4365J.

[3] Brake, M. R. W., "A reduced Iwan model that includes pinning for bolted joint mechanics," Sandia National Laboratories: Albuquerque, New Mexico, 2015, SAND2015-8624C.

[4] Segalman, D. J., "A four-parameter Iwan model for lap-type joints," *Journal of Applied Mechanics*, 72(5), 2005, pp. 752-760. doi: 10.1115/1.1989354

[5] Mignolet, M. P., Song, P., Wang, X. Q., "A stochastic Iwan-type model for joint behavior variability modeling," *Journal of Sound and Vibration*, 349, 2015, pp. 289-298. doi: 10.1016/j.jsv.2015.03.032

DISTRIBUTION

4	Robert M. Lacayo Department of Engineering Physics 1500 Engineering Drive, 534 ERB Madison, WI 53706-1609		(electronic copy)
1	MS0346	Org 1556	1556 (electronic copy)
1	MS0840	James M. Redmond	1550 (electronic copy)
1	MS0899	Technical Library	9536 (electronic copy)


Sandia National Laboratories