SAND2007-3689C

Material Modeling Workshop
Sandia National Laboratories

Bill Scherzinger and Dan Hammerand
Solid Mechanics Department – 1524

Shane Schumacher
Thermal and Reactive Processes – 1516

Arne Gullerud
Computational Solid Mechanics and Structural Dynamics
1542

Workshop Overview

- Contact Information

Bill Scherzinger	Shane Schumacher
wmscher@sandia.gov	scschum@sandia.gov
505-284-4866	505-284-0610
Dan Hammerand	Arne Gullerud
dchamme@sandia.gov	asgulle@sandia.gov
505-844-8956	505-844-4326

Workshop Overview

- Day 1
  - Introduction/Overview (8:00 – 8:30)
  - Code Organization (8:30 – 9:00)
  - Presto (9:00 – 9:30)
  - Break
  - Numerical Issues (10:00 – 12:00)
  - Lunch
  - CTH (1:00 – 5:00)

Workshop Overview

- Day 2
  - LAME and Model Implementation (8:00 – 10:00)
  - Break
  - Running Presto (10:15 – 12:00)
  - Lunch
  - Examples (1:00 – 5:00)

Workshop Overview

- The schedule is flexible
  - What would you like to see?
  - What are your problems?
    - Current design/implementation is driven by **our** problems
    - Does the design work for your problems?
  - We want this to be interactive

Solid Mechanics Codes

- Sandia National Laboratories has a long history of mechanics code development
  - Shock Physics
  - Transient Dynamics
  - Quasi-Static

- Solve engineering problems for nuclear weapons program
  - Design, manufacturing and use (normal, abnormal and hostile environments)

Solid Mechanics Codes

- Types of codes
  - Description
    - Lagrangian (material)
    - Eulerian (spatial)
  - Physics
    - shock physics
    - transient dynamics
    - quasi-static
  - Numerical Method
    - finite difference
    - finite element

CTH is an Eulerian, shock physics, finite difference code

Presto is a Lagrangian, transient dynamic, finite element code

Adagio is a Lagrangian, quasi-static, finite element code

Solid Mechanics Codes

- Concentration will be on Lagrangian (material) descriptions for Presto
  - Natural description for constitutive models
- CTH – Shane Schumacher
  - Equation of state, strength and damage models
- Presto
  - Strength models
  - Equation of state and damage models also exist

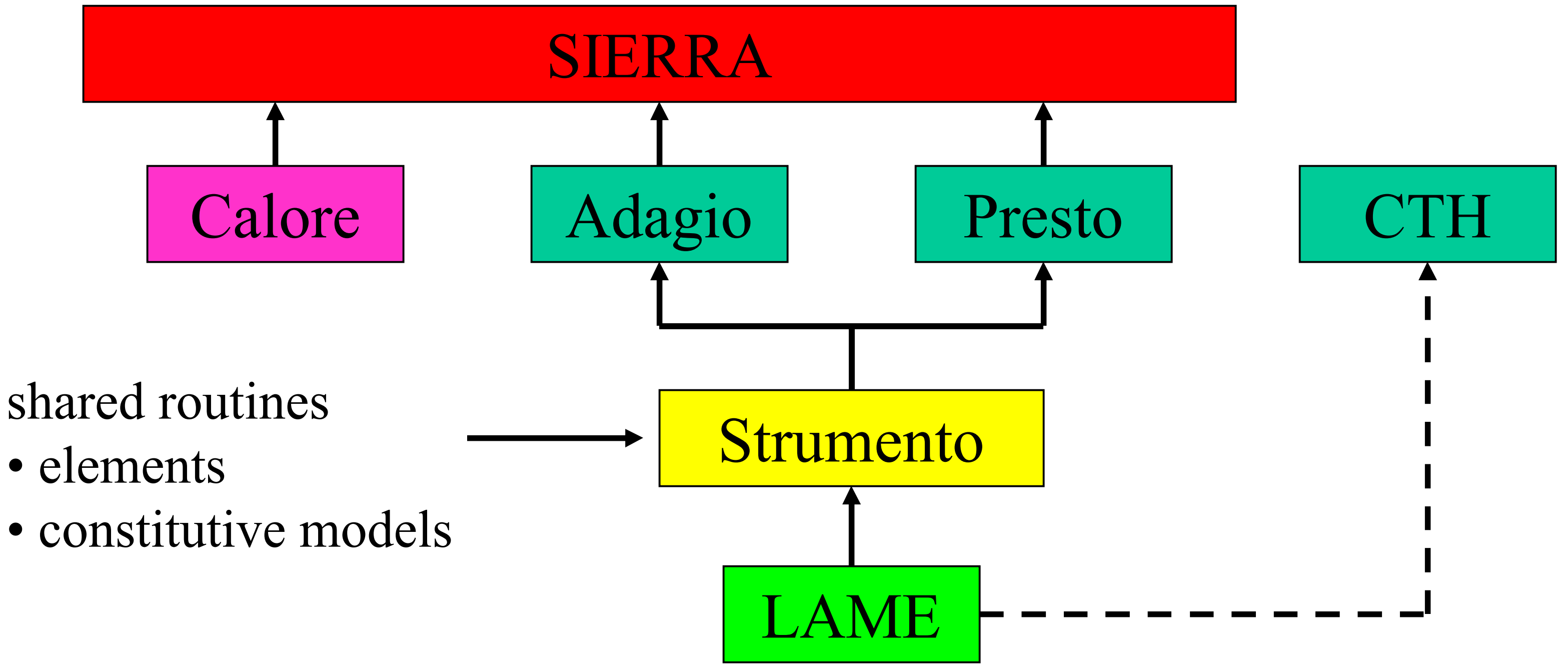
Solid Mechanics Codes

- Presto has a third party library for constitutive models
  - Particularly simple to do for a Lagrangian code
- Library of Advanced Materials for Engineering (LAME)
  - Gabriel Lamé – 1795-1870

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)} \quad ; \quad \mu = \frac{E}{2(1+\nu)}$$

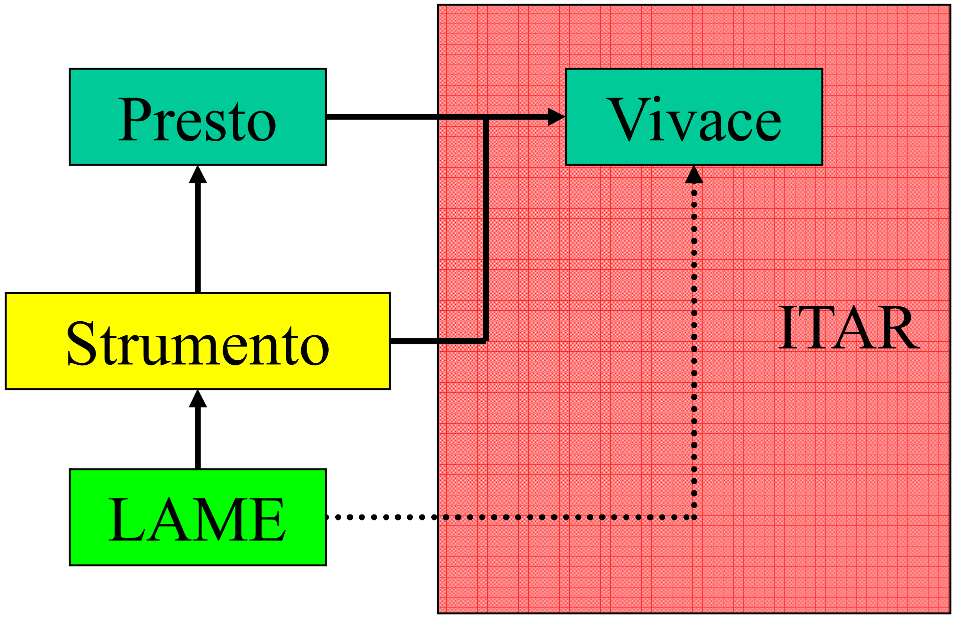
Solid Mechanics Codes

- LAME – Library of Advanced Materials for Engineering
  - Interfaces with Presto/Adagio through Strumento



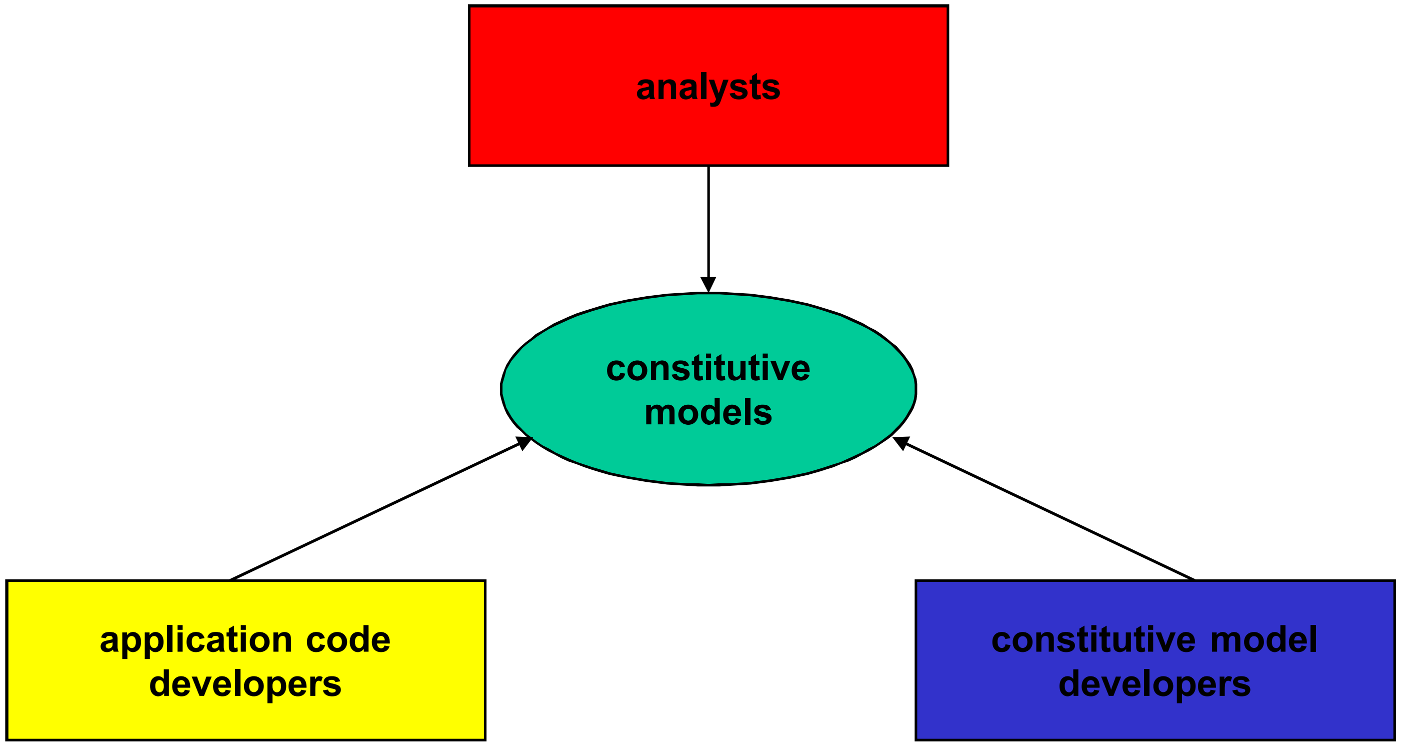
Solid Mechanics Codes

- Another twist...
  - Some aspects of Presto are export controlled
  - These are kept in a product called Vivace
  - Allows us to have a non-ITAR version of Presto



Solid Mechanics Codes

- Why was LAME created?



Solid Mechanics Codes

- Ease of implementation for model developers
- Constitutive models are independent of the host code (e.g. Presto)
- A single repository for constitutive models in Engineering Sciences

- LAME can be thought of as a logical extension of the MIG concept (Brannon and Wong)

Solid Mechanics Codes

- Is LAME finished? – NO
  - LAME is currently used by Presto and Adagio
  - LAME is **not** used by CTH – there is still work to do
  - LAME defines an interface – a host code must use that interface
  - The interface will be flexible

Solid Mechanics Codes

- What still needs to be addressed in LAME
  - Documentation
  - Support for F90
  - EOS model support
  - Structural model support (truss/beam, membrane/plate/shell)
  - Kinematics and thermal strains
  - Material model driver
  - Improved application code interface
  - Support for coordinate systems

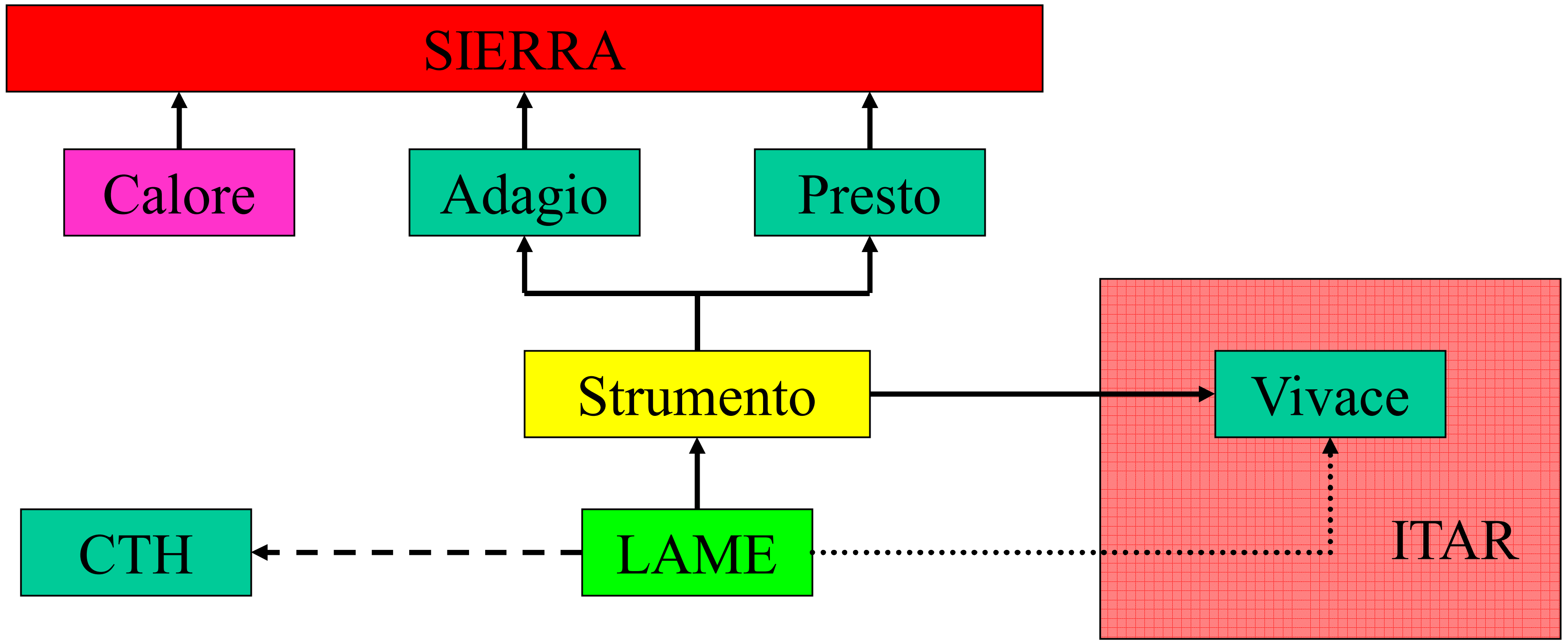
Material Modeling Workshop
Sandia National Laboratories

Bill Scherzinger and Dan Hammerand
Solid Mechanics Department – 1524

Shane Schumacher
Thermal and Reactive Processes – 1516

Arne Gullerud
Computational Solid Mechanics and Structural Dynamics
1542

Code Overview



Code Overview

- There are a number of code products that appear
  - Sierra
  - Presto / Adagio
  - Strumento
  - LAME
  - Vivace
  - CTH

Code Overview

- Sierra – framework
  - data structures
  - parallel issues
  - load balancing
  - re-meshing
  - a lot of CS

- Constitutive modeling is many layers below Sierra framework

Code Overview

- Presto / Adagio
  - Solid mechanics codes
    - transient dynamics ($\mathbf{f} = m\mathbf{a}$ - hyperbolic)
    - quasi-static ($\mathbf{f} = \mathbf{0}$ - elliptic)
  - Timescales
    - Presto – µs – ms
    - Adagio – s – decades
  - Solution procedures are different for both codes
    - Presto – integrate using central difference
    - Adagio – iterative solvers
- Both codes use constitutive models

Code Overview

- Strumento
  - Some common code to Presto / Adagio
    - Elements
    - Constitutive models
    - Effective moduli
  - What's not in strumento
    - Contact

- Constitutive models in strumento are being put into LAME (Interface to LAME)

Code Overview

- LAME
  - Constitutive models are being placed here
  - Simplifies model implementation
    - Most constitutive modelers are “part-time” code developers
    - Sierra code development environment is difficult for “part-time” developers (C++/OOP, dynamic code development environment)
  - Mitigate code development problems with a well designed code library

- LAME provides a stable platform for implementation of constitutive models

Code Overview

- Vivace
  - International Traffic in Arms Regulations (ITAR)
    - State Department – Arms Export Control
  - EOS models are subject to ITAR
  - We want ITAR and non-ITAR versions of Presto
  - Vivace contains our ITAR code
    - We have similar code products for other customers (e.g. CRADA partners)

- Vivace will be able to support LAME models

Code Overview

- CTH
  - Eulerian, finite difference, shock physics code
  - In the process of being coupled with Presto
  - Shane Schumacher

Fitting the Pieces Together

- Except for CTH, all of these code pieces are under the SNTools (Sierra/Nevada tools) code development environment
- SNTools has "systems"
  - Sierra is a system
- The systems have "products"
  - Presto, Adagio, Strumento, LAME and Vivace are products
  - CTH is not a product of the Sierra system (or any system)

Fitting the Pieces Together

- Products in the Sierra system have dependencies
  - On other products
  - On Third Party Libraries

- Presto depends on the following products:
  - contact, equationsolver, FETI-DP, framework, imprint, lame, MPIH, strumento and utility

Fitting the Pieces Together

- Right now LAME does not depend on any other products or TPL's
  - We want to avoid depending on other products
  - Dependencies on TPL's could be supported
  - LAME is "like" a TPL
- Why is LAME in the Sierra system
  - Porting to supported platforms
  - Easier to compile and link with Presto and Adagio
  - ASC requires us to run on many platforms across the labs

Fitting the Pieces Together

- There are many ways to look at how the pieces fit together
- It is important that you have the “big” picture in mind
  - It is very likely that LAME will not meet your *exact* needs
  - Knowing the bigger picture allows us to decide where it is best to make changes
  - Current example: kinematics and objectivity

Fitting the Pieces Together

- Kinematics and Objectivity
  - Enforcing objectivity depends on kinematics
    - We enforce a Green-McInnis stress rate
  - In a finite element code, kinematics are done in the element
  - Does this restrict us?
    - Can we use a hyperelastic model?
    - Can we use a Jaumann stress rate?

$$\begin{tikzcd}
\boxed{\text{Application Code}} \arrow[r, shift left, "\varepsilon"] & \boxed{\text{LAME}} \arrow[l, shift left, "\sigma"]
\end{tikzcd}$$

Material Modeling Workshop
Sandia National Laboratories

Bill Scherzinger and Dan Hammerand

Solid Mechanics Department – 1524

Shane Schumacher

Thermal and Reactive Processes – 1516

Arne Gullerud

Computational Solid Mechanics and Structural Dynamics
1542

Numerical Issues

- Constitutive models are relatively simple

$$\sigma_{ij} = f_{ij}\left(\varepsilon_{kl}, \dot{\varepsilon}_{mn}, \xi_q\right)$$

- But they are not *that* simple
  - They can be difficult to integrate
  - They require many things from a code
  - They provide many things to a code

Numerical Issues

- One way to look at constitutive models...

- Two types:
  - Hyperelastic – strain
  - Hypoelastic – strain rate

  - Strain and strain rate are loaded terms...
  - Requires good continuum mechanics to handle safely!

Numerical Issues

- Strains
  - Different strain measures
    - small strain
    - Green-Lagrange strain
    - Logarithmic strain
    - Seth-Hill family of strains
- But all strains measure the same thing – what is it?

Numerical Issues

- Strain Rates
  - With different strains come different strain rates...
  - PLUS we seem to like to call the rate of deformation a strain rate, even though it isn't!

- Does this all matter?

Numerical Issues

- From a constitutive modeling point of view it does matter
  - This is a matter for constitutive modelers to consider
- From a code development point of view kinematics for constitutive modeling can pose a problem
  - What do we supply?
  - What should we supply?

Numerical Issues

- Most models we implement are hypoelastic
  - Our architecture is set up to handle hypoelastic models
  - Objectivity – Green-McInnis stress rate

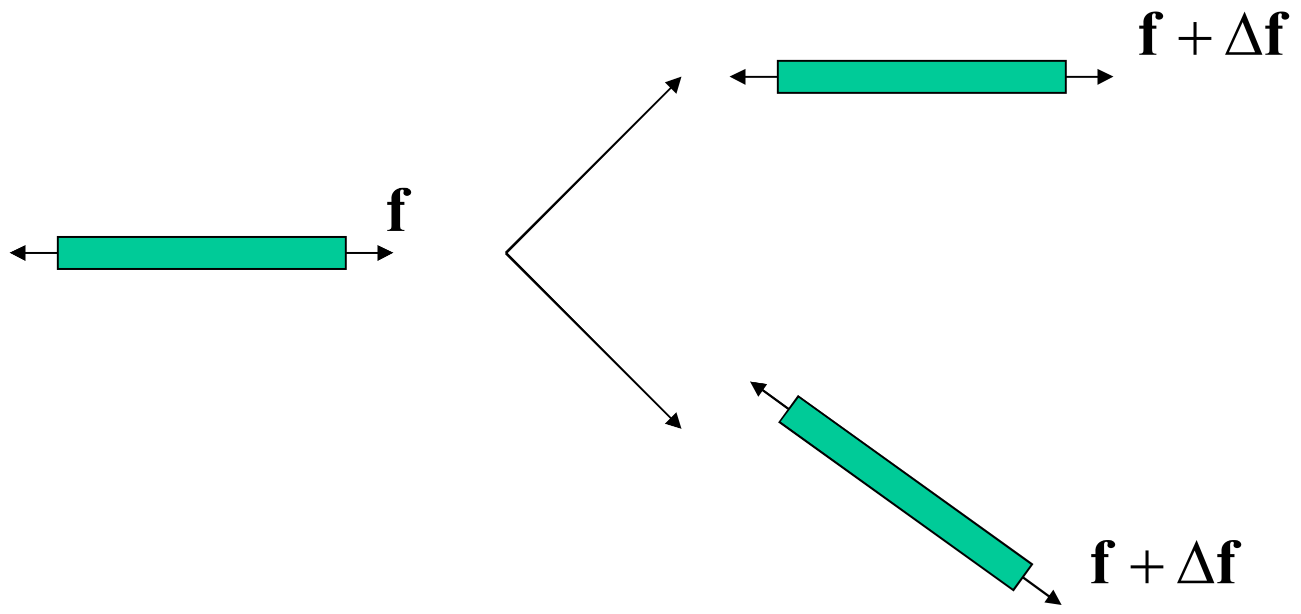
$$\hat{\sigma}_{ij} = \dot{\sigma}_{ij} - \Omega_{ik}\sigma_{kj} + \sigma_{ik}\Omega_{kj} = f_{ij}(D_{kl}, \xi_m)$$

$$\Omega_{ij} = \dot{R}_{ik}R_{jk} \quad ; \quad F_{ij} = R_{ik}U_{kj} = V_{ik}R_{kj}$$

$$\Omega_{ij} = -\Omega_{ji}$$

Numerical Issues

- Why do we care about objectivity?
  - Want constitutive model to be independent of rigid body rotations


f
$\mathbf{f} + \Delta\mathbf{f}$
$\mathbf{f} + \Delta\mathbf{f}$

Numerical Issues

- How can we integrate the Green-McInnis rate?
  - Un-rotated configuration

$$\boxed{\begin{gathered} T_{ij} = R_{ki}\sigma_{kl}R_{lj} \\ \dot{T}_{ij} = R_{ki}\left(\dot{\sigma}_{kl} - \Omega_{km}\sigma_{ml} + \sigma_{km}\Omega_{ml}\right)R_{lj} \end{gathered}}$$

continuum

$$\boxed{\begin{gathered} T^n_{ij} = R^n_{ki}\sigma^n_{kl}R^n_{lj} \\ T^{n+1}_{ij} = T^n_{ij} + \Delta t f_{ij}\left(d_{kl}, \xi_m\right) \\ d_{ij} = R_{ki}D_{kl}R_{lj} \end{gathered}}$$

discretized

Numerical Issues

- How do we implement the discretized algorithm?

$$\boxed{\begin{gathered} T_{ij}^{n} = R_{ki}^{n}\sigma_{kl}^{n}R_{lj}^{n} \\ T_{ij}^{n+1} = T_{ij}^{n} + \Delta t f_{ij}\left(d_{kl},\xi_{m}\right) \\ d_{ij} = R_{ki}D_{kl}R_{lj} \\ \sigma_{ij}^{n+1} = R_{ik}^{n+1}T_{kl}^{n+1}R_{jl}^{n+1} \end{gathered}}$$

- We need the rotation tensor
- The constitutive equation needs the un-rotated rate of deformation
- It may need other quantities un-rotated
- There is no indication of what rotation is used for the rate of deformation[1] (we use the current rotation)

1. Flanagan and Taylor, 1987

Numerical Issues

- Rate of deformation – two methods
  - Midpoint (strong objectivity)[1]

$$D_{ij} = \frac{1}{2}\left(\frac{\partial v_i^{n+1/2}}{\partial x_j^{n+1/2}} + \frac{\partial v_j^{n+1/2}}{\partial x_i^{n+1/2}} \right) \quad ; \quad x_i^{n+1/2} = x_i^n + \frac{1}{2}\Delta t v_i^{n+1/2}$$

  - Strong incremental objectivity[2]

$$dx_i^{n+1} = \hat{F}_{ij} dx_j^n \quad ; \quad \mathbf{D} = \frac{1}{\Delta t}\left(\ln \hat{\mathbf{U}}\right)$$

1. Hughes and Winget, 1980 ; 2. Rashid, 1994

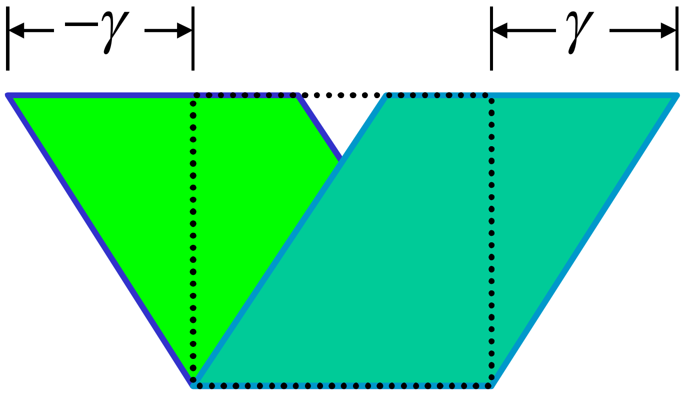
Numerical Issues

- With either method, we can assume that the code calculates a rate of deformation
- Why do we use two?
  - Strong incremental objectivity is closer to what we want
  - Midpoint rate of deformation is faster to calculate
  - For most applications, however, there is no appreciable difference
- Guideline: SIO has the most benefit in quasi-static analyses – large incremental deformations

Numerical Issues

Strong incremental objectivity for cyclic simple shear

$\gamma_{\max} = 0.01$



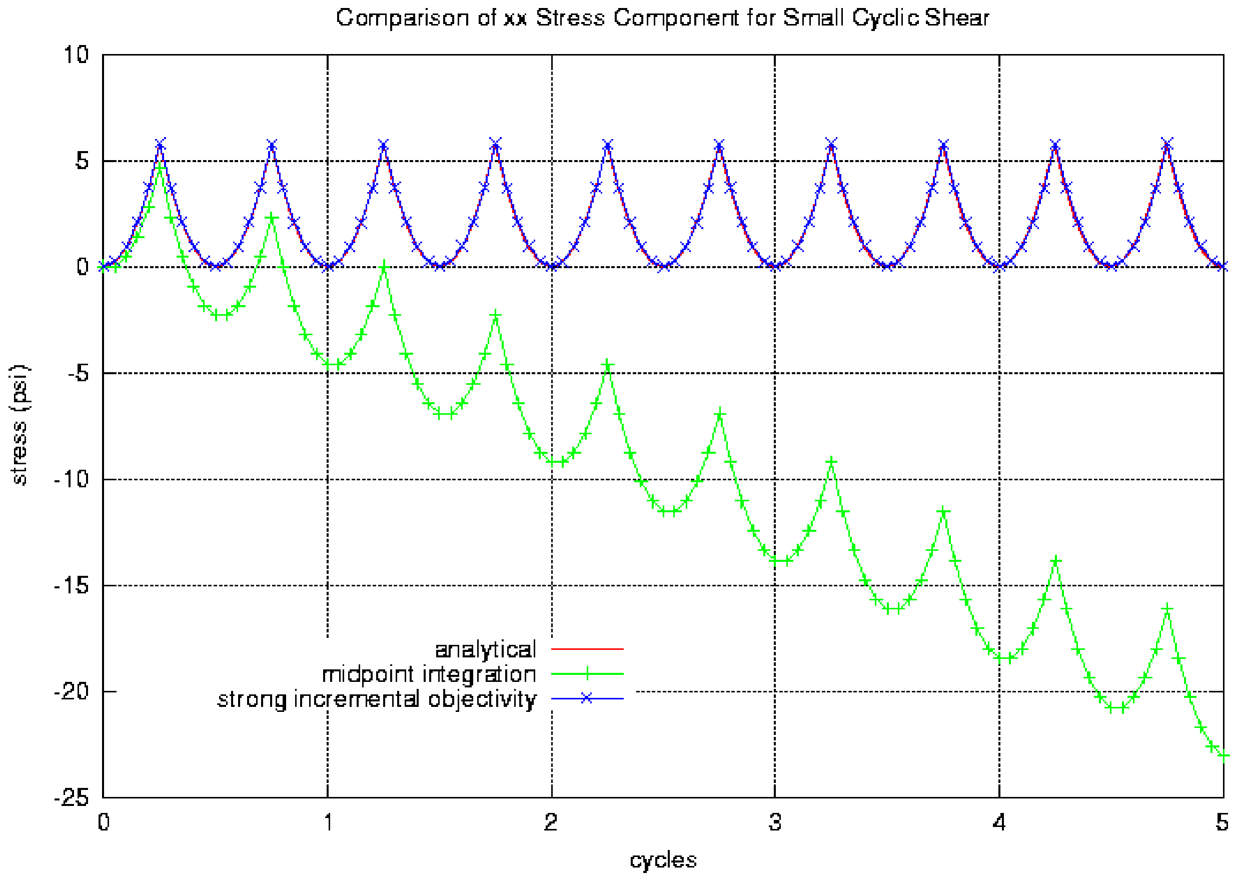
1 cycle goes between $\pm\gamma_{\max}$

$$\gamma = \gamma_{\max} \sin\left(\frac{2\pi t}{T}\right)$$

$$0 \leq t \leq 10T$$

$E = 31{,}870$ psi

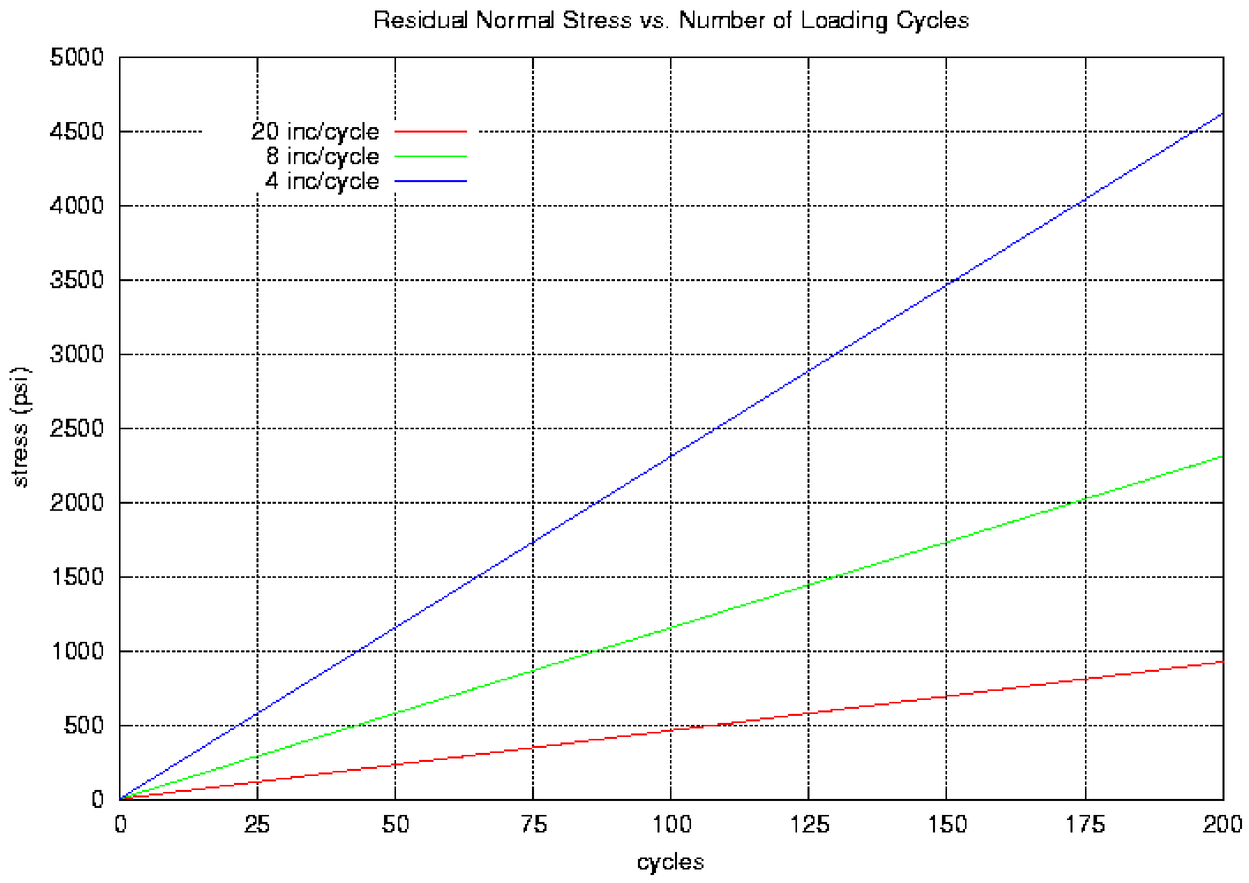
Numerical Issues



- Normal stress grows with Incremental Objectivity

- Non-zero stress due to implementation of finite deformation constitutive model

Numerical Issues



- Growth rate depends on size of increments
- Need for Strong Incremental Objectivity may depend on problem

Numerical Issues

- We also need the rotation – two methods
  - Midpoint – integrate a rate equation
  - Strong incremental objectivity – polar decomposition
- What is the difference between the two?

Numerical Issues

- Midpoint[1,2]

start with: V_{ij}^{n}, R_{ij}^{n}

compute: D_{ij}, W_{ij}

compute: $\Omega_{ij} = \varepsilon_{ikj}\omega_k$

$$\boxed{\begin{aligned} &W_{ij} = \varepsilon_{ikj} w_k \\ &z_i = \varepsilon_{ikj} D_{jm} V_{mk}^{n} \\ &\boldsymbol{\omega} = \mathbf{w} + \left[\mathbf{I}\mathrm{tr}\left(\mathbf{V}\right) - \mathbf{V} \right]^{-1} \cdot \mathbf{z} \end{aligned}}$$

solve:

$$\left(\delta_{ik} - \frac{1}{2}\Delta t \Omega_{ik} \right) R_{kj}^{n+1} = \left(\delta_{ik} + \frac{1}{2}\Delta t \Omega_{ik} \right) R_{kj}^{n}$$

1. Flanagan and Taylor, 1987 ; 2. Hughes and Winget, 1980

Numerical Issues

- Midpoint

$$\dot{V}_{ij} = \left(D_{ik} + W_{ik}\right)V_{kj}^{n} - V_{ik}^{n}\Omega_{kj}$$

$$V_{ij}^{n+1} = V_{ij}^{n} + \Delta t\dot{V}_{ij}$$

This is acceptable if the deformation over a time step is small e.g. Presto

There may be significant errors if the deformation is large e.g. Adagio

Numerical Issues

- Strong Incremental Objectivity

start with: V_{ij}^{n}, R_{ij}^{n}

compute:

$$F_{ij}^{-1} = \delta_{ij} - \frac{\partial u_i}{\partial x_j^{n+1}}$$

$$\hat{F}_{ij}^{-1} = \delta_{ij} - \frac{\partial \Delta u_i}{\partial x_j^{n+1}}$$

We do this because it is easy to calculate the gradient in the current configuration

Numerical Issues

- Strong Incremental Objectivity

$$\hat{F}_{ij}^{-1} = \delta_{ij} - \frac{\partial \Delta u_i}{\partial x_j^{n+1}} \quad \rightarrow \quad \hat{B}_{ij}^{-1} = \hat{F}_{ki}^{-1} \hat{F}_{kj}^{-1}$$

$$\mathbf{D} = -\frac{1}{2\Delta t} \ln \hat{\mathbf{B}}^{-1} \left(= \frac{1}{\Delta t} \ln \hat{\mathbf{V}} \right)$$

Perform a spectral decomposition on $\hat{\mathbf{B}}^{-1}$

i.e. find the eigenvalues and eigenvectors

Numerical Issues

- Strong Incremental Objectivity

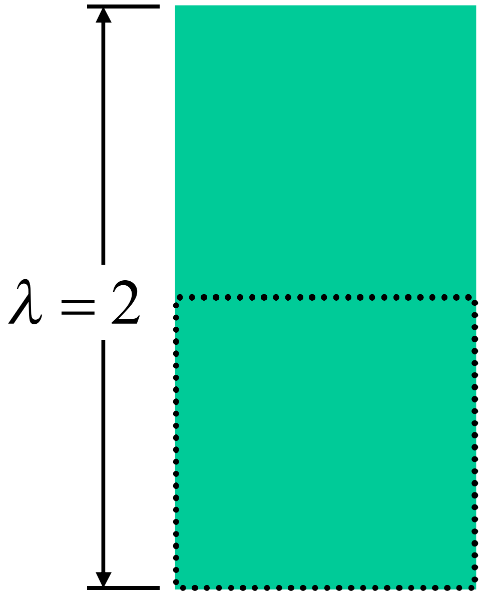
$$F_{ij}^{-1} = \delta_{ij} - \frac{\partial u_i}{\partial x_j^{n+1}} \quad \rightarrow \quad B_{ij}^{-1} = F_{ki}^{-1} F_{kj}^{-1}$$

$$\mathbf{V} = \left(\mathbf{B}^{-1}\right)^{-1/2} \quad ; \quad \mathbf{R} = \mathbf{V} \cdot \mathbf{F}^{-T}$$

Perform a spectral decomposition on $\mathbf{B}^{-1}$

Note that BOTH methods of computing the rotation also find the left stretch

Numerical Issues

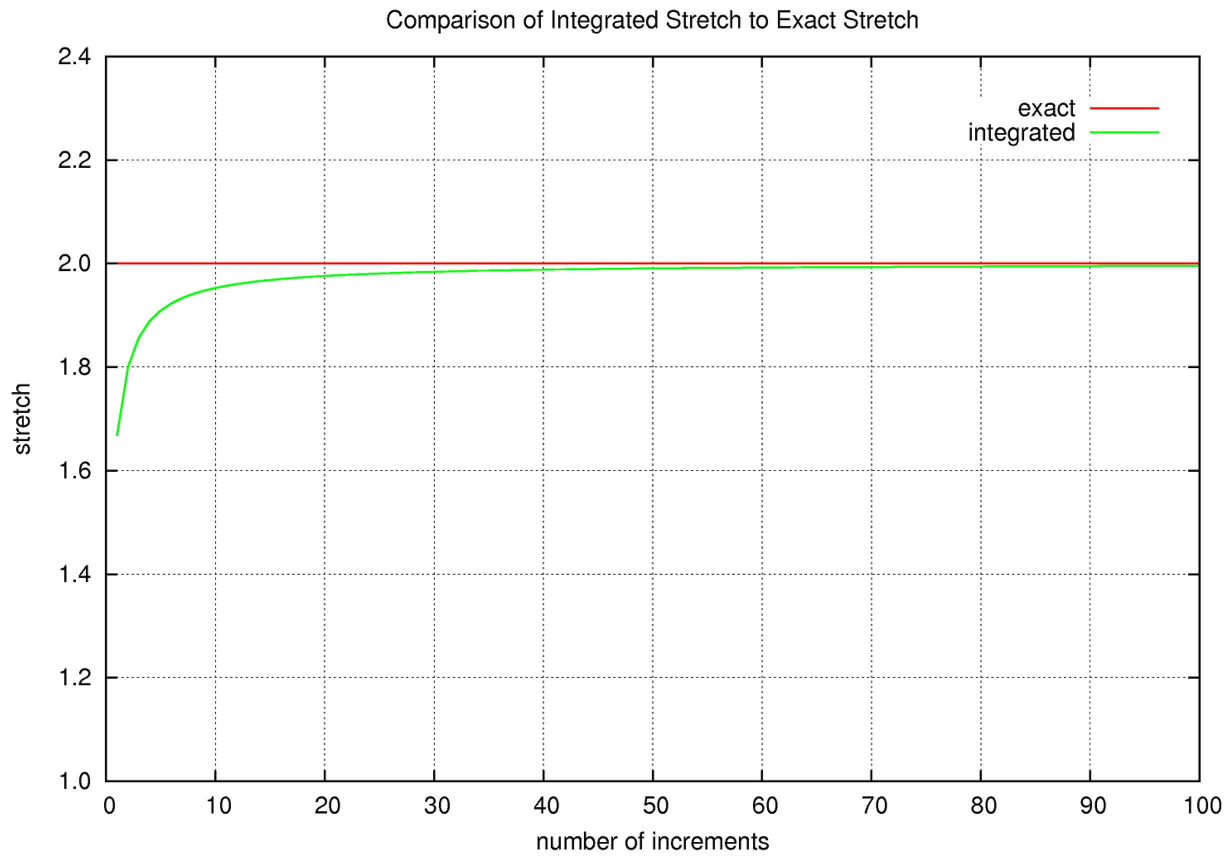


Finite Uniaxial Stretch
simple problem

$$\mathbf{V} = \begin{bmatrix} \lambda & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

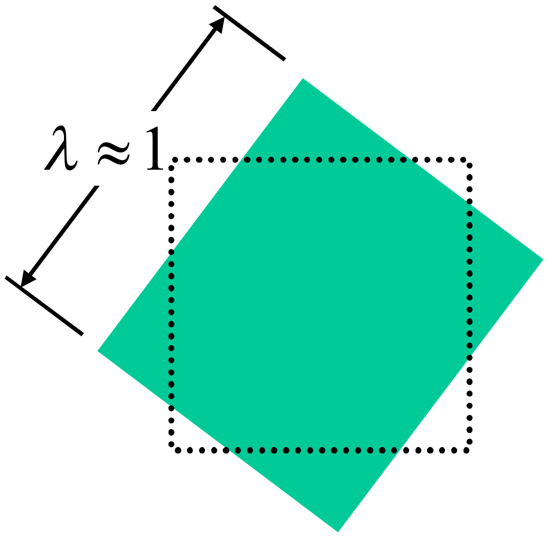
Solve this using 1 to 100 steps

Numerical Issues



Numerical Issues

Small Stretch / Finite Rotation

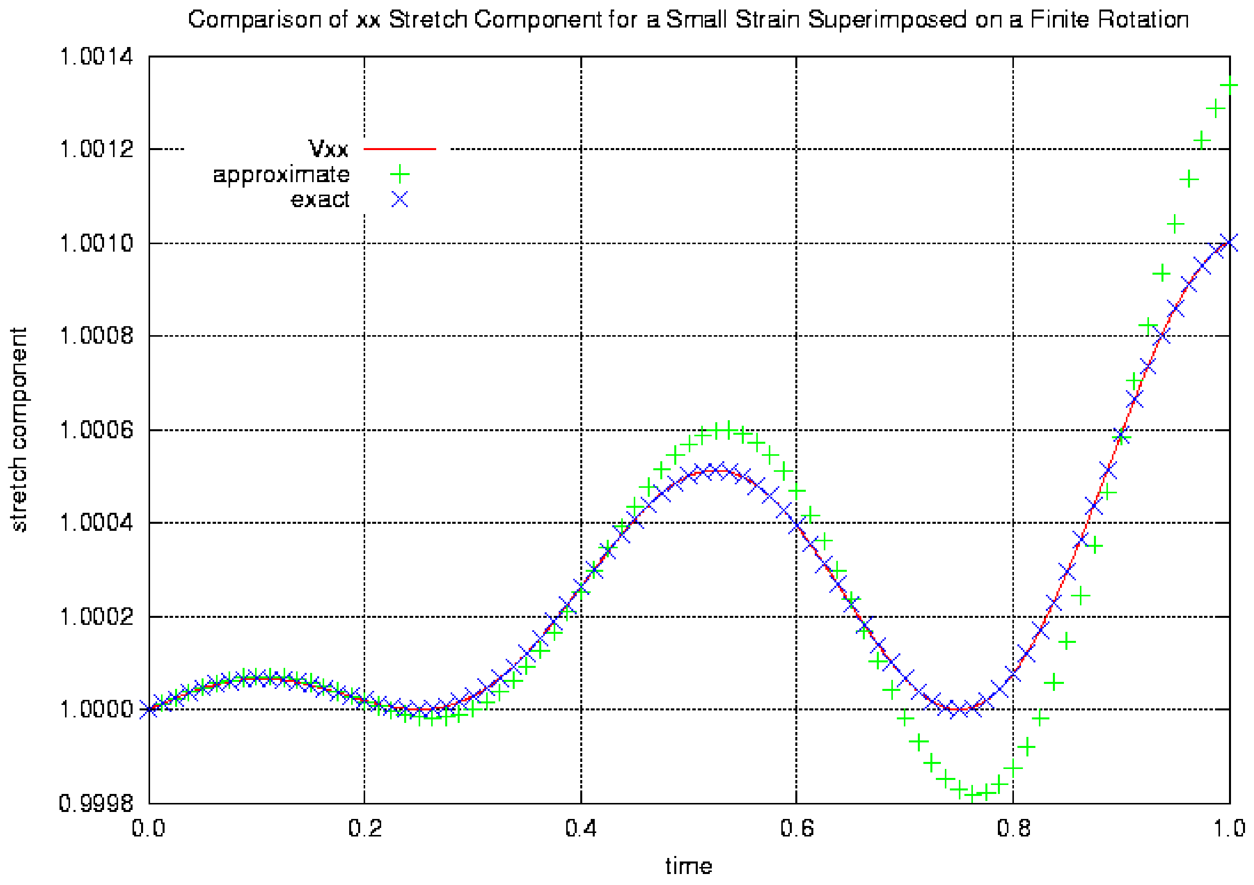


$$\mathbf{V} = \begin{bmatrix} \lambda\cos^2\theta + \sin^2\theta & (1-\lambda)\cos\theta\sin\theta & 0 \\ (1-\lambda)\cos\theta\sin\theta & \cos^2\theta + \lambda\sin^2\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Final $\lambda = 1.001$

Solve this for one full revolution using 80 time steps

Numerical Issues



Analytical $V_{xx} = 1.0010$

Integrated $V_{xx} \approx 1.0013$

Numerical Issues

- So, with the rate of deformation and the rotation, we have two ways to calculate them
  - Incremental objectivity and strong incremental objectivity
  - Integrated and polar decomposition

- We are also passing un-rotated stress and un-rotated rate of deformation
  - But what if we are using a hyperelastic model?
  - What if we want to use a Jaumann stress rate?

Numerical Issues

- Hyperelastic
  - We have the rotation and left stretch
  - We must pass back the un-rotated stress

- Jaumann stress rate
  - We do not have a well defined way to do this with our current code design – but there are ways to do it

Numerical Issues

- Effective moduli
  - Effective moduli for a constitutive model are used in many places
  - It is a way to make all constitutive model look the same – isotropic

$$\hat{L}_{ijkl} = \hat{\lambda}\delta_{ij}\delta_{kl} + \hat{\mu}\left(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}\right)$$

Numerical Issues

- Effective moduli – calculation
  - Look at increments in stress and strain

$$\Delta\sigma_{ij} = \sigma_{ij}^{n+1} - \sigma_{ij}^{n}$$

$$\Delta\varepsilon_{ij} = \Delta t D_{ij}$$

Actually – we use the un-rotated versions of these.
It is easy to show that the results are not dependent on rotation.

Numerical Issues

- Effective moduli – calculation

volumetric $\Delta\sigma_{kk} = \left(3\hat{\lambda} + 2\hat{\mu}\right)\Delta\varepsilon_{kk}$

deviatoric

$$\Delta s_{ij} = \Delta\sigma_{ij} - \frac{1}{3}\delta_{ij}\Delta\sigma_{kk}$$

$$\dot{e}_{ij} = D_{ij} - \frac{1}{3}\delta_{ij}D_{kk}$$

Numerical Issues

- Effective moduli – calculation

$$3\hat{K} = 3\hat{\lambda} + 2\hat{\mu} = \frac{\Delta\sigma_{kk}}{\Delta\varepsilon_{ll}}$$

effective bulk modulus

$$2\hat{\mu} = \frac{\Delta s_{ij}\dot{e}_{ij}}{\Delta t\dot{e}_{kl}\dot{e}_{kl}}$$

effective shear modulus

$$\hat{\lambda} + 2\hat{\mu} = \frac{1}{3}\left(\hat{K} + 2\left(2\hat{\mu}\right)\right)$$

effective dilatational modulus

Numerical Issues

- Effective moduli – problems
  - Two issues
    - Softening

$$\hat{K} \le 0 \quad \hat{\mu} \le 0 \quad \hat{\lambda} + 2\hat{\mu} \le 0$$

    - Negligible strain rate

$$\left|\dot{\varepsilon}_{kk}\right| < \eta \quad \left|\dot{e}_{ij}\right| < \eta$$

  - A lot of logic is in place to handle these cases
    - Is the logic robust?

Numerical Issues

- Critical time step
  - Courant stability limit
  - Element by element
  - Uses effective dilatational modulus to calculate sounds speed

$$c = \sqrt{\frac{\hat{\lambda} + 2\hat{\mu}}{\rho}}$$

$$\Delta\hat{t} = \frac{d}{c} \longleftarrow \text{characteristic element dimension}$$

Numerical Issues

- Critical time step
  - Value is modified for artificial bulk viscosity

$$\Delta t = \Delta\hat{t}\left(\sqrt{1+\eta^2}-\eta\right) \longleftarrow \text{Presto}$$

$$= \frac{\Delta\hat{t}}{\sqrt{1+\eta^2}+\eta} \longleftarrow \text{EPIC}$$

η □ 0 unless there is a shock

Numerical Issues

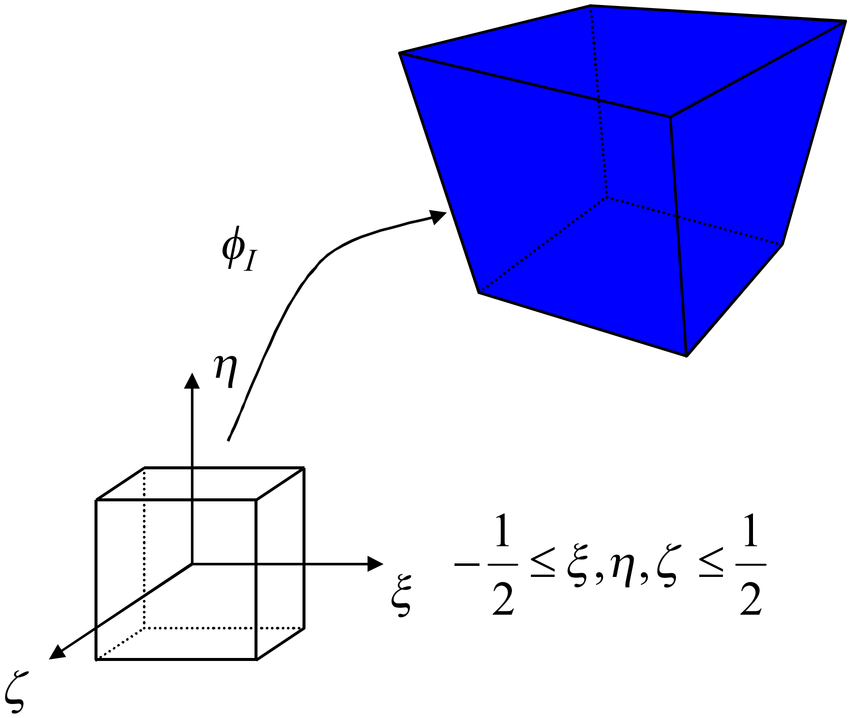
- Hourglass control
  - Underintegrated elements have zero-energy modes
  - These are resisted by fictitious hourglass forces
  - Stiffness used to compute hourglass forces is proportional to the effective moduli

Numerical Issues

- Shape functions for hexahedron

$$x_i = \sum_{I=1}^{8} \phi_I x_{iI} \qquad u_i = \sum_{I=1}^{8} \phi_I u_{iI}$$

$$\phi_I(\xi,\eta,\zeta) = \left(\frac{1}{2} \pm \xi\right)\left(\frac{1}{2} \pm \eta\right)\left(\frac{1}{2} \pm \zeta\right)$$



$$-\frac{1}{2} \le \xi,\eta,\zeta \le \frac{1}{2}$$

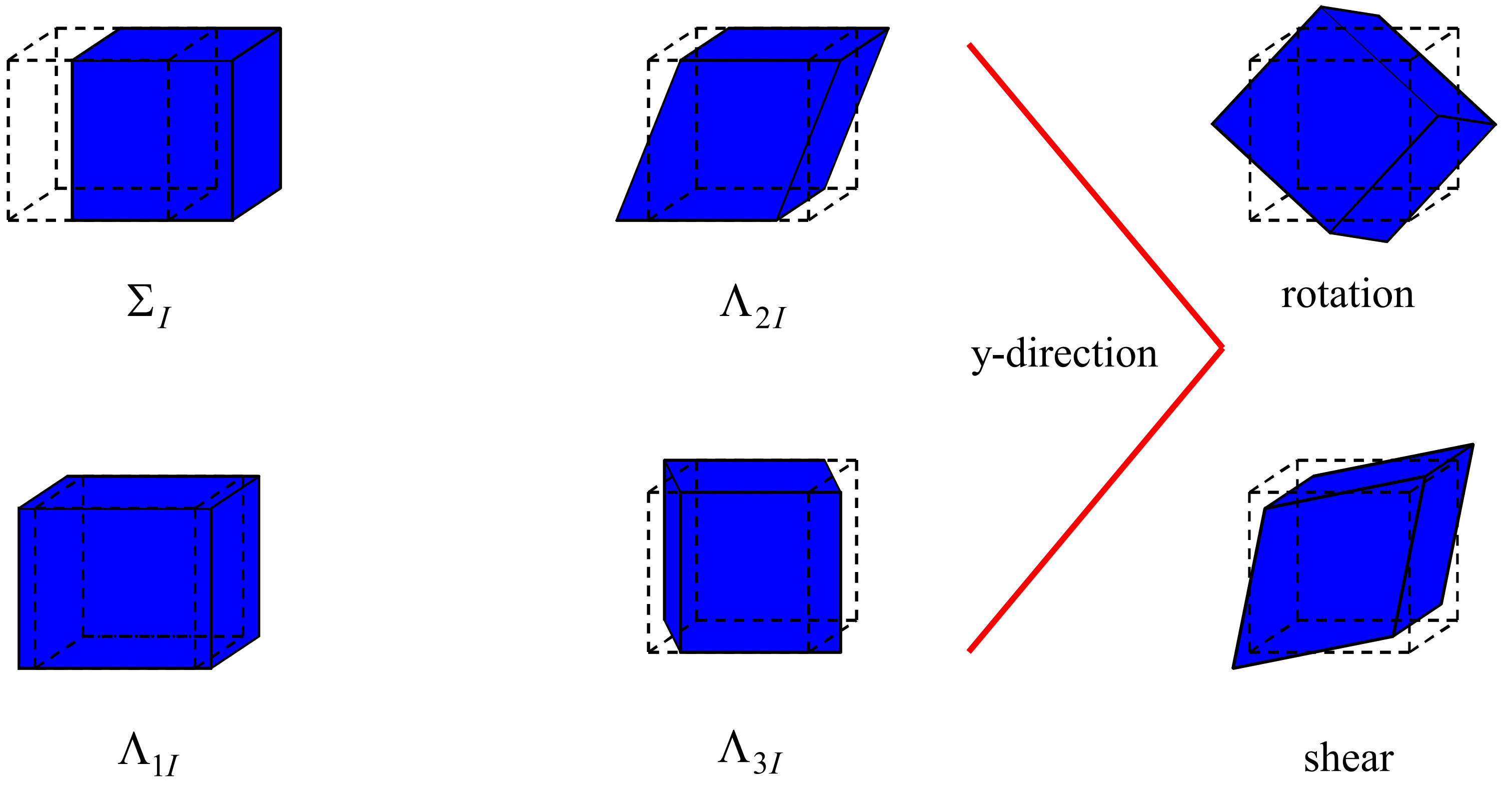
$$\phi_I = \frac{1}{8}\Sigma_I + \frac{1}{4}\xi\Lambda_{1I} + \frac{1}{4}\eta\Lambda_{2I} + \frac{1}{4}\zeta\Lambda_{3I}$$

$$+\frac{1}{2}\eta\zeta\Gamma_{1I} + \frac{1}{2}\zeta\xi\Gamma_{2I} + \frac{1}{2}\xi\eta\Gamma_{3I} + \xi\eta\zeta\Gamma_{4I}$$

$\Sigma_I, \Lambda_{iI}, \Gamma_{\alpha I}$
are **basis vectors**

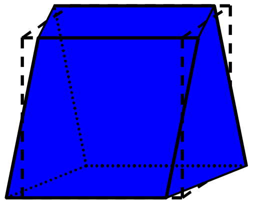
Numerical Issues

- Rigid body and uniform strain modes (x-direction)

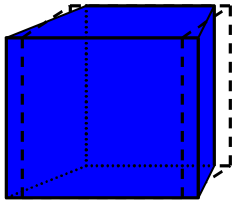


Numerical Issues

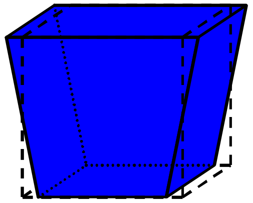
- Hourglass modes



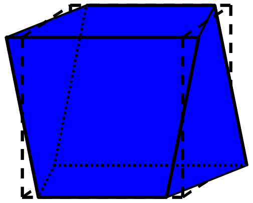
Γ_{1I}



Γ_{2I}



Γ_{3I}

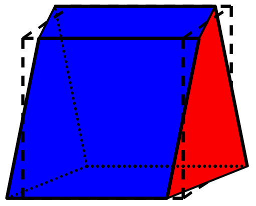


Γ_{4I}

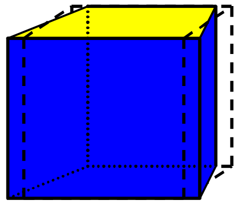
Numerical Issues

- Hourglass modes are not all the same

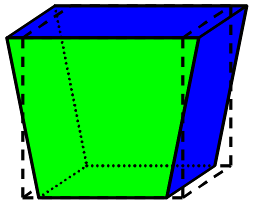
  - Hourglass modes that contribute to gradient operator



mode 1

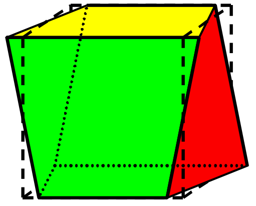


mode 2



mode 3

  - Hourglass mode that does not contribute to gradient operator



mode 4

Numerical Issues

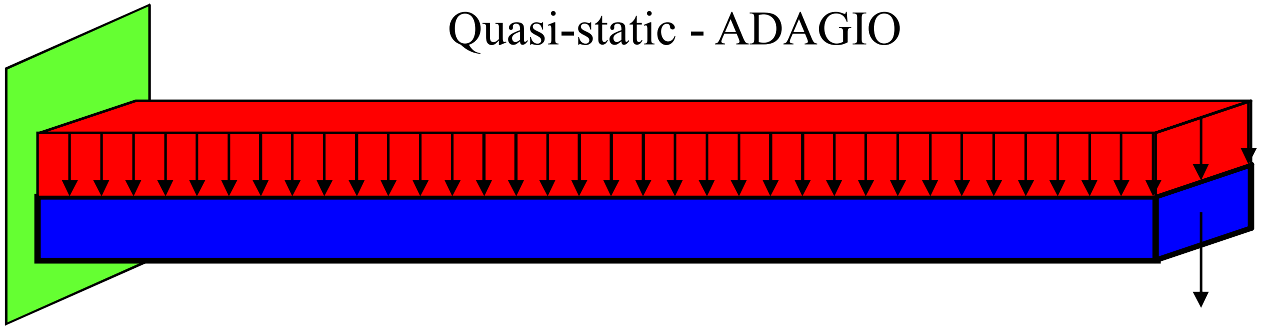
- Hourglass forces are proportional to the effective moduli

$$\underset{\text{hourglass force}}{\underset{\uparrow}{Q_{i\alpha}}} = \kappa\, \underset{\text{maximum element stiffness (effective moduli)}}{\underset{\uparrow}{K_{\max}}}\, \underset{\text{hourglass mode}}{\underset{\uparrow}{q_{i\alpha}}}$$

Numerical Issues

- Cantilever beam – pressure load

Quasi-static - ADAGIO



Beam dimensions are
20 x 1 x 4

Pressure chosen so that $\delta = 0.01$ with a beam theory solution

Material Properties

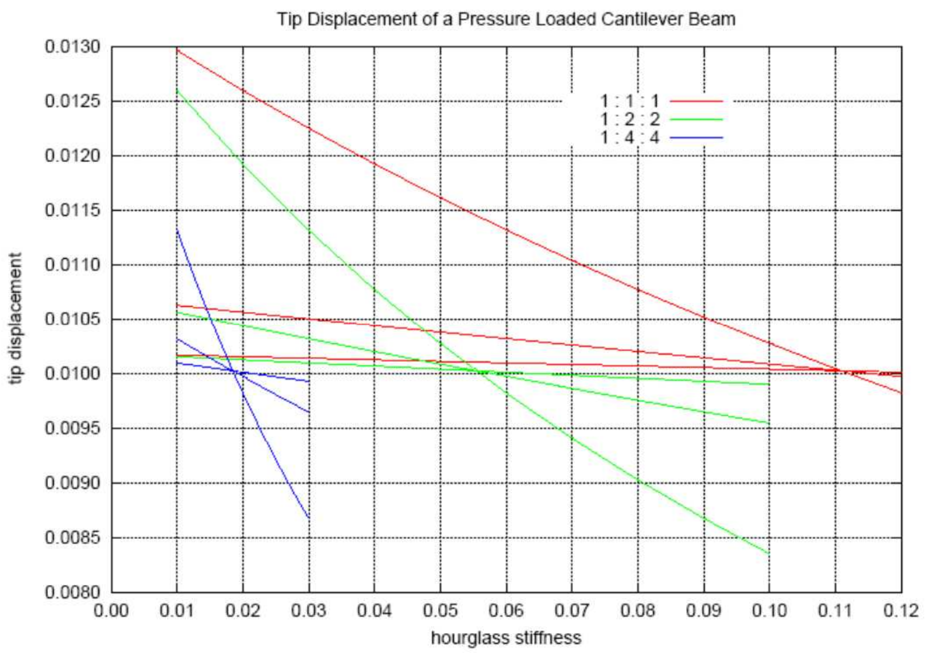
$E = 10\times10^6$

$\nu = 0.3$

Can we get coarse mesh accuracy?

Numerical Issues

- Cantilever beam – pressure load



Element aspect ratios:

(1) 1:1:1

(2) 1:2:2

(3) 1:4:4

Elements through thickness

(1) 2

(2) 4

(3) 8

Numerical Issues

- Hourglass results
  - Mesh refinement is the only real guarantee of accuracy
  - Choice of hourglass stiffness combined with element aspect ratio determines "coarse mesh accuracy"
  - Coarse mesh accuracy can only be attained if we have an accurate approximation to maximum element stiffness – i.e. accurate tangent (effective) moduli

Numerical Issues

- Tangent moduli
  - For Newton methods in quasi-static problems – Adagio
  - We primarily use iterative methods (Nonlinear Preconditioned Conjugate Gradient)
  - We have not used tangent moduli in our codes

Numerical Issues

- Uses for tangent moduli in addition to forming a stiffness matrix
  - Replacing/augmenting effective moduli routines
  - Stability calculations
  - Two moduli of interest
    - Instantaneous tangent – effective moduli, stability calculations
    - Consistent tangent – stiffness matrix

Numerical Issues

- Augmented Lagrange
  - Used in quasi-static solution procedure – Adagio
    - Incompressible materials
    - Multiple materials stiff/soft
  - Solves a series of (easier) model problems that converge to the true solution
  - Especially useful for iterative solvers
  - AL strategies depend on constitutive models

Numerical Issues

- There are many numerical issues associated with constitutive models
  - Constitutive model type
  - Kinematics
  - Transient dynamic / quasi-static
  - Element design
  - Solution strategy

- Handling these issue will define an interface

Material Modeling Workshop Sandia National Laboratories

Bill Scherzinger and Dan Hammerand
Solid Mechanics Department – 1524

Shane Schumacher
Thermal and Reactive Processes – 1516

Arne Gullerud
Computational Solid Mechanics and Structural Dynamics
1542

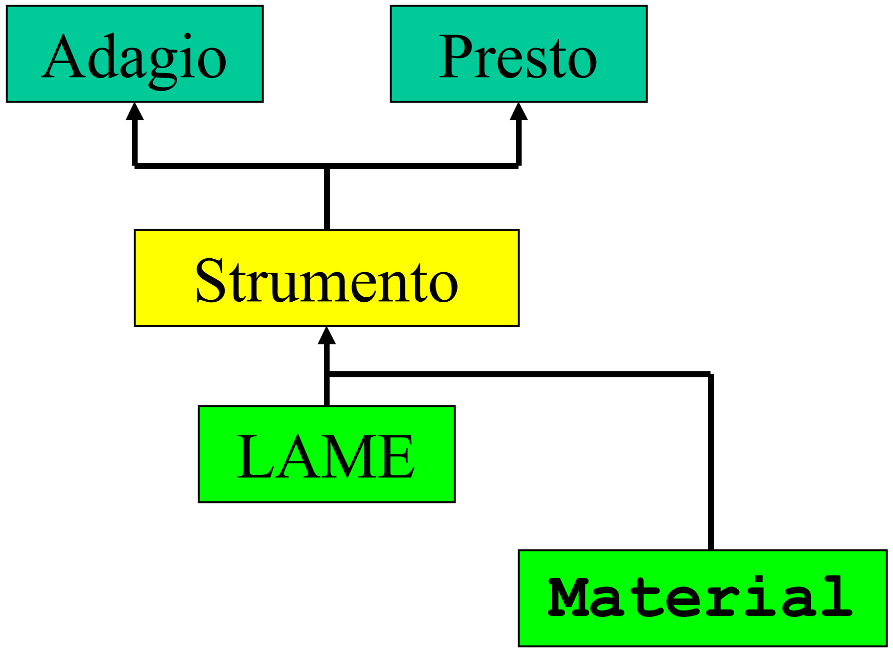
LAME Design

- C++/OOP interface
  - Not an expert in C++/OOP
  - Modelers are not expected to be experts either
  - Blind leading the blind?

- Problems with OOP (...as I see it)
  - Not the way we normally solve problems
    - Objects and relationships vs. sequential thinking
  - Way more rope than we need to hang ourselves!

LAME Design

- So why use C++/OOP? – Design an interface



- Base class provides the interface through the **Material** class

- Host code has a **Material** object, it could be an elastic material, and elastic-plastic material,...

LAME Design

- A material model calculates the stress given a strain
  - This design is great!
    - Host code: “Hey, Material, here’s a strain, what’s the new stress?”
    - Material: “Hold on... Here it is.”
- But there is more to it than that...
  - What strain (strain rate) does the material need?
  - If the material uses a stress rate, what stress rate is it?
  - How do we get output from the material other than stress?
  - How does the material affect the numerical method?
  - ....

LAME Design

- A well designed interface needs to handle a wide range of material models and capabilities
  - variable length inputs
  - update and output state variables

Material.h

- The **`Material`** base class – line by line

```
class Material{
public:
   ...
protected:
   ...
private:
   ...
};
```

public data and methods
- accessible by outside code
- interface to host code

protected data and methods
- accessible by derived classes
- not seen by host code

private data and methods
- accessible only by this class

Material.h

- **`public`** methods

```
public:
  Material();
  virtual ~Material();
```

<u>Constructor and destructor for the base class</u>

• acts as the interface for creating and destroying the derived class

• constructor sets a few initial variables

• destructor is virtual to ensure that the destructor for the derived class is called

Material.h

- `protected` data

```
protected:
   double * properties;
   double * p_data;

   int num_material_properties;
   int num_scratch_vars;
   int num_state_vars;
```

`properties` – a pointer to a material property array
`p_data` – **not used**
`num_material_properties` – number of material properties
`num_scratch_vars` – number of scratch variables – **not used**
`num_state_vars` – number of state variables

Material.h

- **`public`** methods (again)

```
public:
  int getNumStateVars(){
    return num_state_vars;
  };
  int getNumScratchVars(){
    return num_scratch_vars;
  };
  void setScratchPtr( double * p_vars){
    p_data = p_vars;
  };
```

These methods are used by the host code to allocate memory

`getNumStateVars()` – returns the number of state variables needed

`getNumScratchVars()` and **`setScratchPtr()`** – **not used**

Material.h

- **`public`** methods

```
public:
  virtual int initialize( matParams * p );
  virtual int getStress( matParams * p );
  virtual int loadStepInit( matParams * p );
  virtual int getConsistentTangent( matParams * p );
  virtual int pcElasticModuli( matParams * p );
```

These methods define most of the interface

- **`initialize()`** – initializes the model
- **`getStress()`** – returns the updated stress
- **`loadStepInit()`** – initializes variables at the beginning of a load step
- **`getConsistentTangent()`** – will be used with Adagio (unimplemented)
- **`pcElasticModuli()`** – used with Adagio

Material.h

- `public` methods

```
public:
  virtual int initialize( matParams * p );
  virtual int getStress( matParams * p );
  virtual int loadStepInit( matParams * p );
  virtual int getConsistentTangent( matParams * p );
  virtual int pcElasticModuli( matParams * p );
```

Each of these methods, if needed, will be implemented in a derived class – i.e. a specific material model.

The **`getStress()`** method must be implemented for each material model

Material.h

- **`public`** methods

```
public:
  virtual int initialize( matParams * p );
  virtual int getStress( matParams * p );
  virtual int loadStepInit( matParams * p );
  virtual int getConsistentTangent( matParams * p );
  virtual int pcElasticModuli( matParams * p );
```

This is the meat of the interface

Notice that everything is passed using a structure: **`matParams`**. This make the interface very easy to modify. If a model needs some information that is not in **`matParams`**, we can add it.

Material.h

- **`matParams`**

```
struct matParams{
   int nelements;
   int nintg;
   double dt;
   double time;
   ...
}
```

`nelements` – number of elements (material points) to be processed
`nintg` – number of integration points
`dt` – time increment
`time` – current solution time, t_{n+1}

Material.h

- `matParams`

```
struct matParams{
   ...
   double * strain_rate;
   double * stress_old;
   double * stress_new;
   ...
}
```

$$\boxed{d_{ij} = R^{n+1}_{ki} D_{kl} R^{n+1}_{lj}}$$

$$\boxed{T^{n}_{ij} = R^{n}_{ki} \sigma^{n}_{kl} R^{n}_{lj}}$$

$$\boxed{T^{n+1}_{ij} = R^{n+1}_{ki} \sigma^{n+1}_{kl} R^{n+1}_{lj}}$$

`strain_rate` – un-rotated strain rate
`stress_old` – un-rotated stress at time t_n
`stress_new` – un-rotated stress at time t_{n+1}

Material.h

- **`matParams`**

```
struct matParams{
   ...
   double * state_old;
   double * state_new;
   double * temp_old;
   double * temp_new;
   ...
}
```

$\boxed{\xi_n}$

$\boxed{\xi_{n+1}}$

$\boxed{\theta_n}$

`state_old` – state variables at time t_n

`state_new` – state variables at time t_{n+1}

`temp_old` – temperature at time t_n

$\boxed{\theta_{n+1}}$

`temp_new` – temperature at time t_{n+1}

Material.h

- **matParams**

```
struct matParams{
   ...
   double * left_stretch;
   double * rotation;
   ...
}
```

$$\boxed{F_{ij} = V_{ik} R_{kj}}$$

$$\boxed{V_{ij}^{n+1}}$$

left stretch – left stretch tensor at time t_{n+1}

rotation – rotation tensor at time t_{n+1}

$$\boxed{R_{ij}^{n+1}}$$

other variables in **matParams** are either not fully implemented or used for certain quasi-static solution procedures

Material.h

- `protected` methods

```
protected:
  int getNumberProps( string name,
                      const MatProps & props);
  double getMaterialProperty( string name,
                              const MatProps & props,
                              int n);
}
```

`getNumberProps()` – for a given property, it determines how many entries exist
`getMaterialProperty()` – retrieves the **`n`**th property

Material.h

- **`protected`** data

```
protected:
  static materials::function::function * p_function;
  static materials::lame::app_interface * p_host;
```

These pointers are used to access the host code:
`p_function` – evaluate functions
`p_host` – report information/errors.

Material.h

- **`protected`** data and methods

```
protected:
  map<string,int> state_variable_map;
  int set_state_variable_alias(string name,
                               int pos);
```

The host code allocates memory for state variables, but you must keep track of which variable is which inside of your model.

The state variable alias is a name associated with a specific variable in the state variable array – this is used for output

Material.h

- **`public`** methods

```
public:
  void setFunctionPtr(...);
  static materials::function::function * getFunctionPtr()
  static void reportError();
```

These pointers are used to access the host code:

`p_function` – evaluate functions

`p_host` – report information/errors.

Material.h

- **`private`** methods

```
private:
  Material( const Material & );
  Material & operator= ( const Material & );
```

The copy constructor and assignment operator are private and unimplemented to prevent use

Elastic Model

- Elastic model
  - This is the simplest constitutive model we have
  - Three files:
    - **`lame/include/models/Elastic.h`**
    - **`lame/src/models/Elastic.C`**
    - **`lame/src/models/elastic.F`**
  - Header file, implementation file, FORTRAN file

Elastic Model

- `lame/include/models/Elastic.h`

```
#ifndef _ELASTIC_H_
#define _ELASTIC_H_

#include <models/Material.h>
#include <Lame_Fortran.h>

namespace materials {
  namespace lame {
    ...
  }
}

#endif
```

Elastic Model

- **`lame/include/models/Elastic.h`**

```
namespace materials {
  namespace lame {
    class Elastic : public Material{
      ...
    };
    ...
  }
}
```

All models are derived from **`Material`**

Elastic Model

- **`lame/include/models/Elastic.h`**

```
class Elastic : public Material{
  public:
    Elastic( MatProps * props );
    ~Elastic();

    int initialize( matParams * p );
    int getStress( matParams * p );
};
```

Note that there is no **`loadStepInit`** method for the Elastic model

Elastic Model

- `lame/include/models/Elastic.h`

```
class Elastic : public Material{
  private:
    Elastic( const Elastic & );
    Elastic & operator=( const Elastic & );
};
```

This is a good coding practice – all models will have it

Elastic Model

- `lame/include/models/Elastic.h`

```
extern "C" void
  LAME_FORTRAN(elastic_get_stress)
  ( const int & npts,
    const double & dt,
    const double * props,
    double * strain_rate,
    double * stress_old,
    double * stress_new );

extern "C" void
  LAME_FORTRAN(elastic_initialize)
  ( const double * props );
```

These allow for calls to the FORTRAN

Elastic Model

- `lame/src/models/Elastic.C`

```
#include <models/Elastic.h>

namespace materials {
  namespace lame {
    ...
  }
}
```

Elastic Model

- **lame/src/models/Elastic.C**

```
Elastic::Elastic( MatProps props ){
  num_material_properties = 2;

  properties = new double[num_material_properties];

  properties[0] = getMaterialProperty("YOUNGS_MODULUS",
                                      props);
  properties[1] = getMaterialProperty("POISSONS_RATIO",
                                      props);
}
```

The Elastic model needs two material properties

Elastic Model

- `lame/src/models/Elastic.C`

```
Elastic::Elastic( MatProps props ){
  num_material_properties = 2;

  properties = new double[num_material_properties];

  properties[0] = getMaterialProperty("YOUNGS_MODULUS",
                                      props);
  properties[1] = getMaterialProperty("POISSONS_RATIO",
                                      props);
}
```

Allocate memory for the material properties
THIS MUST BE FREED LATER!

Elastic Model

- `lame/src/models/Elastic.C`

```
Elastic::Elastic( MatProps props ){
  num_material_properties = 2;

  properties = new double[num_material_properties];

  properties[0] = getMaterialProperty("YOUNGS_MODULUS",
                                      props);
  properties[1] = getMaterialProperty("POISSONS_RATIO",
                                      props);
}
```

The property array is filled
Note that C++ starts counting at 0

Elastic Model

- `lame/src/models/Elastic.C`

```
Elastic::~Elastic(){
  delete [] properties;
  properties = NULL;
}
```

The memory allocated for the material properties is freed when the destructor is called

Elastic Model

- `lame/src/models/Elastic.C`

```
int Elastic::initialize( matParams * p ){

  LAME_FORTRAN(elastic_initialize)(properties);

  return 0;
}
```

The initialization for the Elastic model is called

- the pointer to the **`struct matParams`** comes from the host code
- **`properties`** is a pointer that is owned by this model

Elastic Model

- **lame/src/models/elastic.F**

```
subroutine elastic_initialize( prop )

character *80 message

dimension prop(2)

youngs_modulus = prop(1)
poissons_ratio = prop(2)

...

return
end
```

This subroutine will only do error checking

Elastic Model

- **lame/src/models/elastic.F**

```
...
if (youngs_modulus.lt.zero) then
  write(message,101)
  call lame_report_error(3,message)
endif
...
101 format('Youngs modulus is less than zero')
...
```

Elastic Model

- `lame/src/models/Elastic.C`

```
int Elastic::getStress( matParams * p ){

  LAME_FORTRAN(elastic_get_stress)(
    p->nelements,
    p->dt,
    properties,
    p->strain_rate,
    p->stress_old,
    p->stress_new);

  return 0;
}
```

The updated stress is found