# LAME

SAND2007-0899P

- Material model implementation in LAME
  - Object Oriented Programming provides the interface with Adagio/Presto
    - Base class
    - Material model is derived from the base class
  - Implementation of material model requires the following
    - Creation of a class for the material model that is derived from the base class
    - Fortran / C / C++ routines that have the core model routines
    - Add model to a list of models that can be used (This is changing – for the better too!)
  - The derived class will have simple "look and feel"
    - Some freedom will be restricted for now
    - Interface will involve stress, kinematics, state variables, etc...

# LAME


- Capabilities must be the same for all models
  - Return a stress – mechanics
  - Update state variables – mechanics
  - Return effective incremental moduli – numerical
    - Moduli defined by model
    - Numerical moduli defined in base class
  - Return tangent moduli – numerical
    - Consistent tangent where possible
    - Numerical tangent defined in the base class
  - We need to clearly identify where, when and how we want use this information in the application codes
    - Preconditioner, critical time step, hourglass control, etc...

# Interface with Adagio/Presto


- The stress passed between Adagio/Presto and LAME is the un-rotated stress

$$T_{ij} = R_{ki}\sigma_{kl}R_{lj} \quad ; \quad F_{ij} = R_{ik}U_{kj} = V_{ik}R_{kj}$$

- The "strain-rate" passed between Adagio/Presto and LAME is the un-rotated rate of deformation

$$d_{ij} = R_{ki}D_{kl}R_{lj}$$

  - This is done for to preserve objectivity for hypoelastic models (Green-McInnis stress rate)
  - Hypothetically (hypoelastoetically?) this makes writing constitutive models easy... unless you want to write something different

# Interface with Adagio/Presto


- A state variable array is passed between Adagio/Presto and LAME
  - The size of the array is set by the constitutive model
  - Allocating space for the state variables is done by the application code (Adagio/Presto)
  - The updating of the state variables is done by the constitutive model
  - Output is done by the application code (Adagio/Presto); there is a method to associate a state variable name with the variable in the array

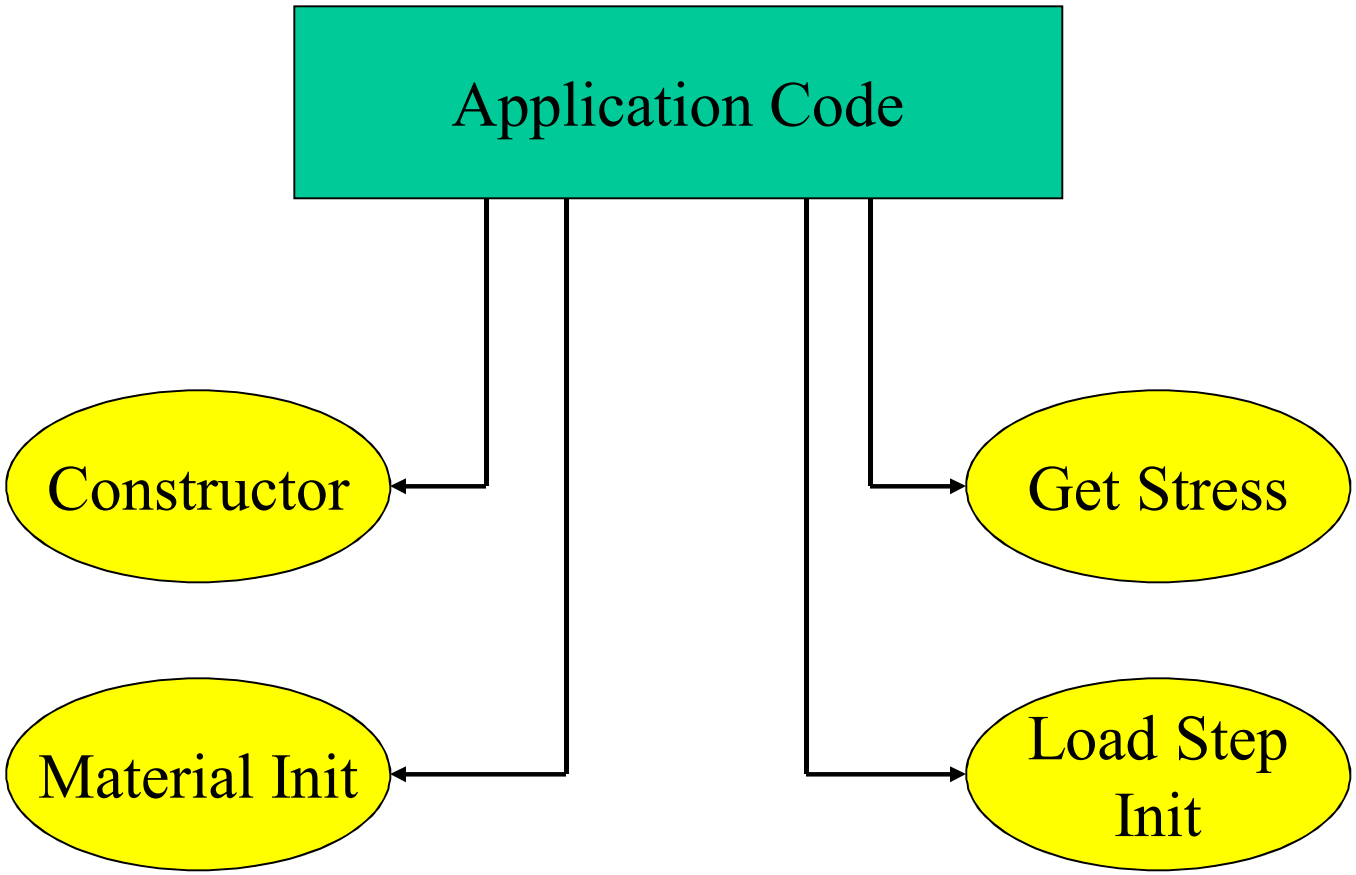
# Interface with Adagio/Presto


- Kinematic variables are passed from Adagio/Presto to LAME
  - The kinematic variables are calculated by the application code
  - Availability depends on what is / can be calculated
  - With Adagio and Presto we have a lot of control over this
  - Kinematic quantities that are available are:
    - Un-rotated rate of deformation tensor
    - Left stretch tensor
    - Rotation tensor
    - Inverse deformation gradient tensor
    - Deformation gradient tensor
  - For problems where the temperature changes we need to calculate the thermal strain
    - We currently do “legacy” thermal strain, but we might want to change this

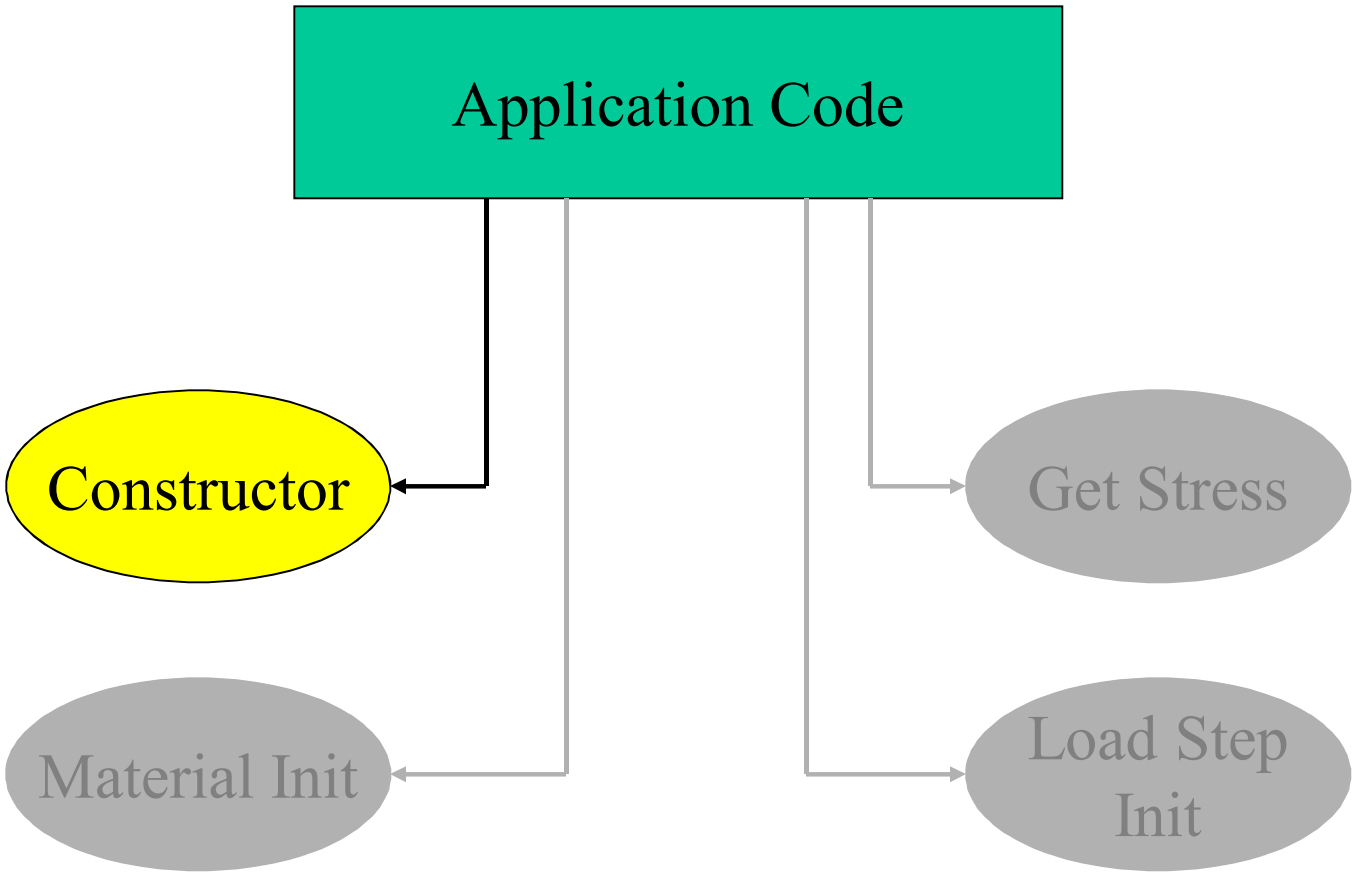
# Interface with Adagio/Presto


- Additional quantities can be passed to and from LAME
  - Temperature is an example
  - Other quantities will come up, especially with code coupling
  - This part of the interface is not fully developed since we are not sure what we will want
  - When the interface is developed it must be flexible and easy to use

# Interface with Adagio/Presto



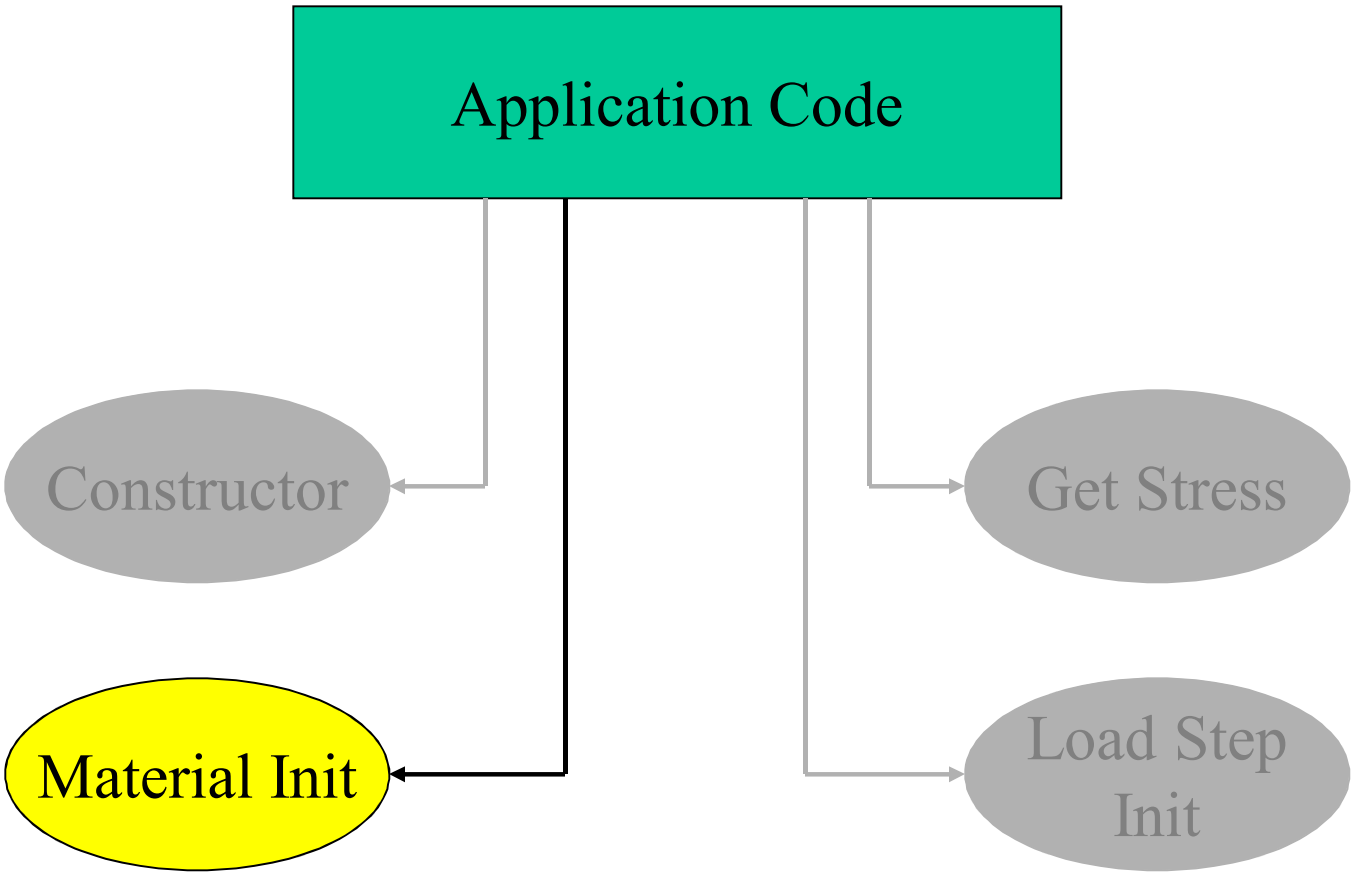
# Interface with Adagio/Presto



# Interface with Adagio/Presto


- Constructor
  - The material properties are stored in an array of double precision variables
  - The number of state variables is set
    - Adagio/Presto will have Sierra set aside memory for the state variables
  - The names of state variables are identified with terms in state variable array
    - These names are used so that Adagio/Presto can output state variables
  - Well defined instructions ***in the code*** will exist for how to write the constructor

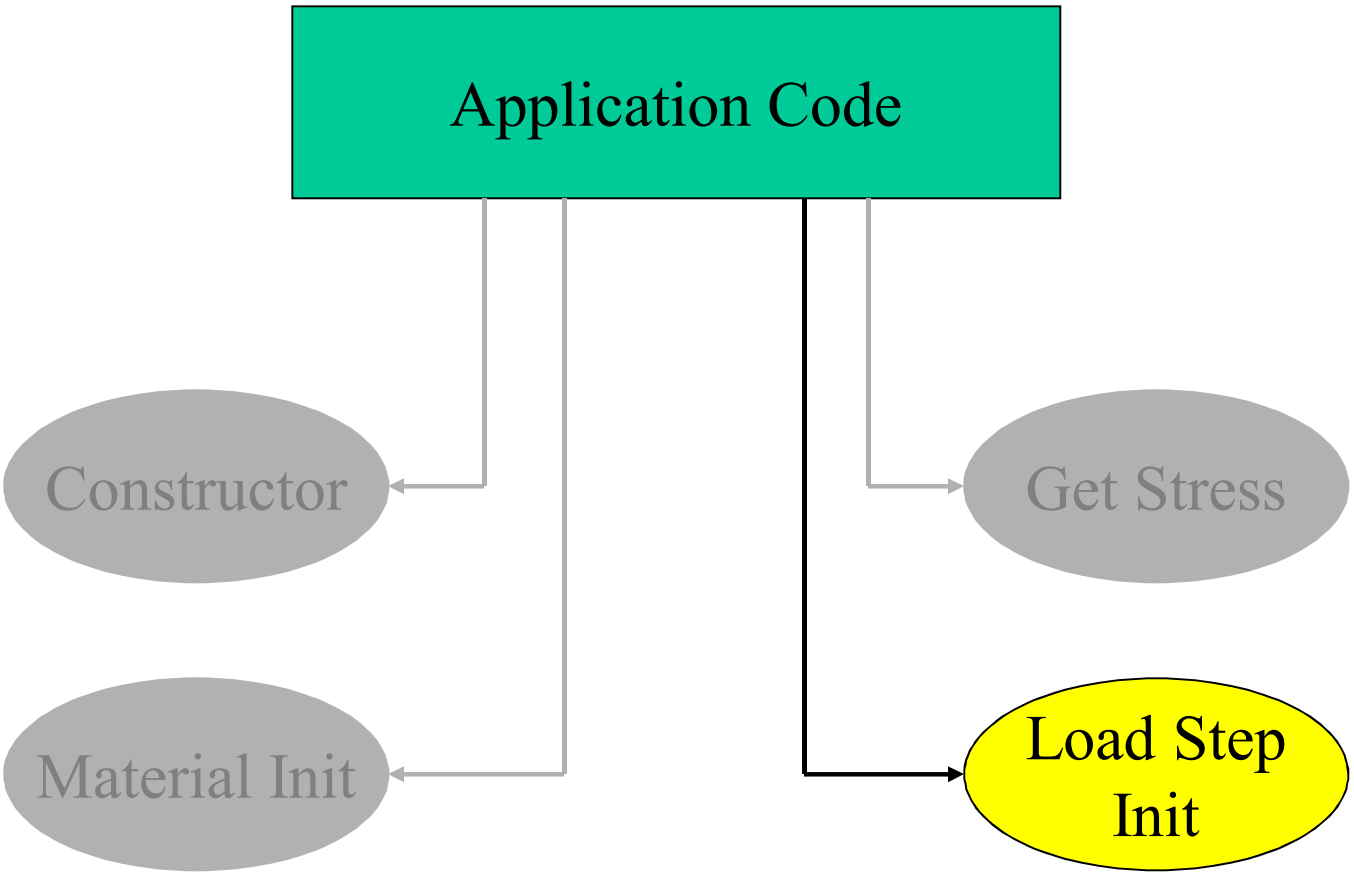
# Interface with Adagio/Presto



# Interface with Adagio/Presto


- Material Initialization
  - A number of things may have to be initialized for a material model
  - Initialization is done in a material initialization method
    - Example: initial values of state variables (radius of the yield surface)

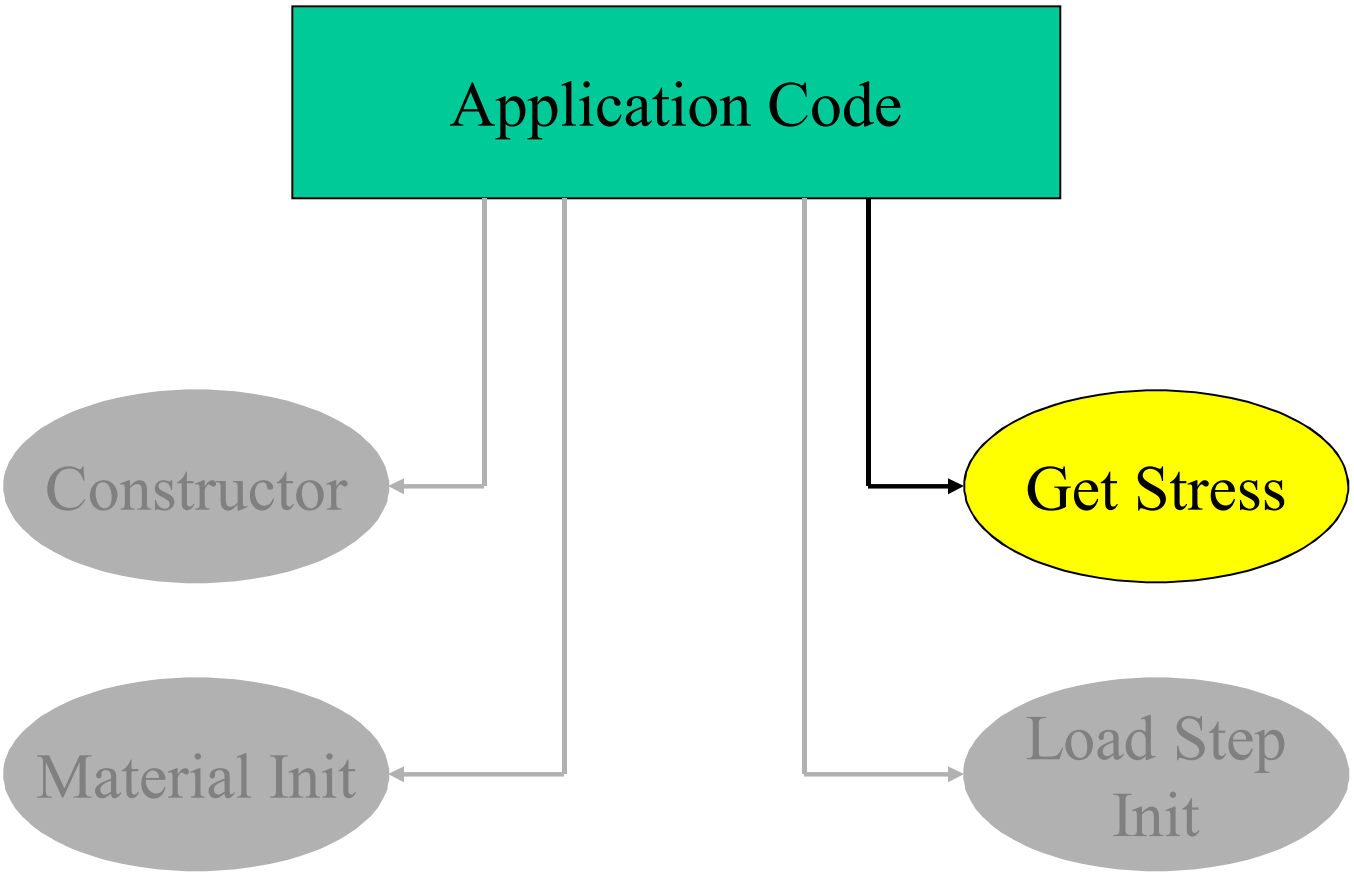
# Interface with Adagio/Presto



# Interface with Adagio/Presto


- Load step initialization
  - Material properties may need to be initialized at the start of a load step
    - example: temperature dependent quantities

# Interface with Adagio/Presto



# Interface with Adagio/Presto

- Stress calculations
  - The stress calculation is the “meat” of the constitutive model
  - Stress (un-rotated) is updated
  - State variables (in appropriate configuration *if necessary*) are updated
  - If they can be calculated, the effective moduli are calculated
  - If they can be calculated, the tangent moduli are calculated

# Example Model


```
#ifndef _ELASTIC_PLASTIC_H_
#define _ELASTIC_PLASTIC_H_

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

namespace materials {

  namespace lame {

    class ElasticPlastic : public Material{

    public:

      ElasticPlastic( MatProps props ) ;
      ElasticPlastic( MatPropsNew props ) ;
      ~ElasticPlastic() ;

      int initialize( int & npoints,
                      double dt,
                      double * state_old,
                      double * state_new );

      int getStress( matParams * p );

    } ;
```

```
//*****************************************************
//
// FORTRAN subroutine definitions
//
//*****************************************************

    extern "C" void
      LAME_FORTRAN(elastic_plastic_initialize)
      ( const int    & nelem,
        const double * props,
              double * state_old,
              double * state_new );

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

  } // lame

}  // materials

#endif
```

# Example Model


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

using namespace std;

namespace materials {

  namespace lame {

    //*************************************************************
    //
    // This is the Elastic Plastic Linear Hardening model
    //
    // The properties it reads into the properties array are:
    //
    //   YOUNGS_MODULUS
    //   POISSONS_RATIO
    //   YIELD_STRESS
    //   HARDENING_MODULUS
    //   BETA
    //
    // There are 8 state variables.  All of them are aliased for output
    //
    //   0      - equivelent plastic strain
    //   1 to 6 - components of the back stress tensor
    //   7      - radius of yield surface
    //
    //*************************************************************

    ...

}
```

# Example Model


```
ElasticPlastic::ElasticPlastic( MatProps props){

    //
    // Material Property Definitions
    //
    // 1) the number of material properties is given by
    //    the variable num_material_properties
    //
    // 2) memory is allocated for a property array
    //    of length equal to the number of material
    //    properties
    //
    // 3) the values are set by calling
    //      getMaterialProperty("PROPERTY_NAME", props);
    //

    num_material_properties = 5;

    properties = new double[num_material_properties];

    properties[0] = getMaterialProperty("YOUNGS_MODULUS", props);
    properties[1] = getMaterialProperty("POISSONS_RATIO", props);
    properties[2] = getMaterialProperty("YIELD_STRESS", props);
    properties[3] = getMaterialProperty("HARDENING_MODULUS", props);
    properties[4] = getMaterialProperty("BETA", props);

    ...

  }
```

# Example Model


```
ElasticPlastic::ElasticPlastic( MatProps props){

    ...

    //
    // State Variable Definitions
    //
    // 1) the number of state variables is given
    //    by the variable num_state_vars
    //
    // 2) names for state variables that are used for
    //    output can be defined through an alias
    //    the alias is set using set_state_variable_alias
    //

    num_state_vars = 8;

    set_state_variable_alias("EQPS",0);
    set_state_variable_alias("ALPHA_XX",1);
    set_state_variable_alias("ALPHA_YY",2);
    set_state_variable_alias("ALPHA_ZZ",3);
    set_state_variable_alias("ALPHA_XY",4);
    set_state_variable_alias("ALPHA_YZ",5);
    set_state_variable_alias("ALPHA_ZX",6);
    set_state_variable_alias("RADIUS",7);

  }
```

# Example Model


```
//*****************************************************************
//
// The destructor for the Elastic Plastic Linear Hardening
// model frees up the memory created for the material properties.
//
//*****************************************************************

ElasticPlastic::~ElasticPlastic(){

  //
  // these two lines should appear in order to free up memory
  //
  //   delete [] properties;
  //   properties = NULL;

  delete [] properties;
  properties = NULL;

}
```

# Example Model

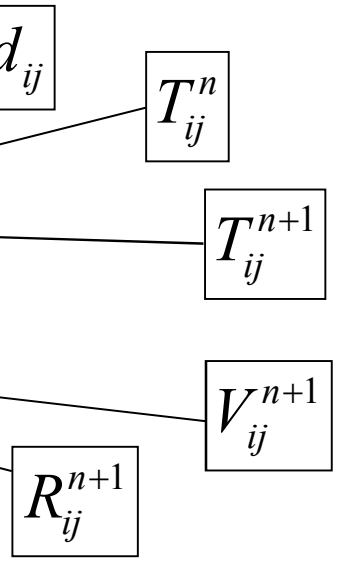

```
//*****************************************************************
//
// The initialize method for the Elastic Plastic Linear
// Hardening model initializes the state variables
//
//*****************************************************************

int ElasticPlastic::initialize( int & npoints,
                                double * state_old,
                                double * state_new ) {

  //
  // state variables are initialized at the start of an analysis
  // using this method
  //
  //   npoints   : number of material points
  //   state_old : state variables at time t_n
  //   state_new : state variables at time t_n+1
  //
  //   properties: material property array
  //

  LAME_FORTRAN(elastic_plastic_initialize)( npoints,
                                            properties,
                                            state_old,
                                            state_new );

  return 0;

}
```

# Example Model


```
//*****************************************************************
//
// The getStress method for the Elastic Plastic Linear
// Hardening model finds the new stress for the material
//
//*****************************************************************

int ElasticPlastic::getStress(matParams *p) {

  //
  // matParams * p : a pointer to memory of a structure that has
  //                 information we need to evaluate the new stress
  //                 state
  //
  // properties : material property array
  //

  LAME_FORTRAN(elastic_plastic_get_stress)(p->nelements,
                                           p->dt,
                                           properties,
                                           p->strain_rate,
                                           p->stress_old,
                                           p->stress_new,
                                           p->state_old,
                                           p->state_new) ;

  return 0;

}
```

# Example Model

```
struct matParams {

   int nelements;
   int n_intg;
   double dt;
   double * strain_rate;
   double * stress_old;
   double * stress_new;
   double * state_old;
   double * state_new;
   double * temp_old;
   double * temp_new;
   double * left_stretch;
   double * rotation;
   double * entropy;
   double * energy_balance_term;
   double lambda;
   double twomu;
   double * tangent_moduli;
   double * ym_old;
   double * ym_new;
   double * nU_new;
   double * nU_old;
   double * bulk_scaling;
   double * shear_scaling;

} ;
```



# LAME – Future Work


- Needs:
  - Implementation of effective moduli / tangent moduli interface
  - Improve function definition / evaluation
  - Allocation of scratch space
- Wants:
  - Integer / logical material property types
  - General kinematics
  - Thermal strains
- Goals for FY07:
  - Default location for constitutive models
  - Documentation for developers, application codes and users

# LAME and Application Codes


- What is needed in Adagio/Presto?
  - Input is handled by Adagio/Presto – this includes material properties
  - For Adagio/Presto to use a constitutive model – some work needs to be done in Adagio/Presto
- xml file
  - xml file defines the material model and the material properties for the model
- Smod_Material_Input_Structural.C
  - Location of input parameters for constitutive models – parser handler?

# LAME and Application Codes


```
<?xml version='1.0'?>
<!DOCTYPE sierraCommandSyntax SYSTEM "/home/sntools/Src/Sierra/Config/sierra.dtd" >
<!-- XML File for individual Sierra Commands -->
<?xml-stylesheet href="http://www.engsci.sandia.gov/Sierra/codes/scms/xml_html/sierra.xsl" type="text/xsl"?>
<?cocoon-process type="xslt"?>
<sierraCommandSyntax version="1.0">
<!--

  BEGIN PARAMETERS FOR MODEL ELASTIC_PLASTIC

-->
  <block_command usage="MATERIAL" physics="STRUCTURAL" instance="200">
    <keyword> PARAMETERS FOR MODEL ELASTIC_PLASTIC </keyword>
    <line_parameter> </line_parameter>
    <summary>
       Material parameters for elastic plastic linear hardening model.
    </summary>
<!--

    YIELD STRESS = sigma_y :: defined in another file

-->
```

# LAME and Application Codes


```
<!--

    HARDENING MODULUS = H

-->
    <line_command usage="MATERIAL" physics="STRUCTURAL" instance="201" >
      <keyword>HARDENING MODULUS</keyword>
      <parameter type="ENUMERATED">Delimiter</parameter>
      <parameter type="REAL">H<nonnegative/></parameter>
      <summary>
        Supplies a value for the material's hardening modulus.  This
        is a linear term in the relation between the material's equivalent
        stress and equivalent plastic strain
      </summary>
    </line_command>
```

# LAME and Application Codes


```
<!--

    BETA = beta

-->
    <line_command usage="MATERIAL" physics="STRUCTURAL" instance="202" >
      <keyword>BETA</keyword>
      <parameter type="ENUMERATED">Delimiter</parameter>
      <parameter type="REAL">Beta
        <minInclusive>0.0</minInclusive>
        <maxInclusive>1.0</maxInclusive>
      </parameter>
      <summary>
        Specifies the mix of isotropic and kinematic hardening for the
        model.  This is especially useful if the material in question
        has some Bauschinger effect.

        When Beta = 0 the hardening is entirely kinematic.
        When Beta = 1 the hardening is entirely isotropic.
        When Beta is between 0 and 1 the hardening is a combination.
      </summary>
    </line_command>
  </block_command><!-- END PARAMETERS FOR MODEL ELASTIC_PLASTIC -->
</sierraCommandSyntax>
```

# LAME and Application Codes


## Smod_MaterialInputStructural.C

```
//-----------------------------------------
//
// Structural material properties
//
// Material property identifiers (some are
// shared between models)
//
// Convention:
//
//   * Identifiers that are multiples of
//     100 are for model blocks
//   * Properties that are shared between
//     models have ids 101-199
//   * Properties that are specific to a
//     single model have ids of model block
//     id + [1..99]
//
// If a material property identifier is
// unique for a model, then it is referenced
// directly with the parser id number.
//
// Steps for a new model are as follows.
// (See below for examples.)
//
```

```
// (1) Create space for a block of code and
//     label the block
//
// "'Material Name' Material Model".
//
// (2) Define a const Prsr::Identifier object
//     with the material model name.
//     The argument list will be
//
// ( material, structural, #### )
//
//     Here the number corresponds to the
//     parser info for the material in the
//     xml file.
//
// (3) Create a reference to an
//     Apub::MaterialInput object
```

# LAME and Application Codes


## Smod_MaterialInputStructural.C

```
// (4) Call the Abub::MaterialInput object's
//     property_command method as many times
//     as needed.  The argument will be one
//     of the following "const Prsr::Identifier"
//     objects, or one that is specific to
//     the material.  If it is specific to the
//     material then it needs to have the
//     argument list
//
// Prsr::Identifier( material, structural,#### )
//
//     Here the number corresponds to the parser
//     info for the material in the xml file.
//
//  Here is a list of the material model
//  "centuries"
//
//   199  ELASTIC*
//   200  ELASTIC PLASTIC
//   300  EP POWER HARD
//   400  INCOMPRESSIBLE SOLID
//   500  ORTHOTROPIC CRUSH
//   600  POWER LAW CREEP
//   700  THERMOELASTIC
//   800  CURING EPOXY
```

```
//   900  USED
//  1000  FOAM PLASTICITY
//  1100  ELASTIC FRACTURE
//  1200  VISCOPLASTIC
//  1300  USED
//  1400  SOLDER

...

//  5300  NLVE POLYMER
//  5400
//  5500
//  5600  MOONEY RIVLIN
//  5700  USED
//
//    *(The ELASTIC model is not a century, but
//      it was put in a long time ago, before we
//      really understood how we wanted to
//      organize this.)
//
//-------------------------------------------
```

# LAME and Application Codes


## Smod_MaterialInputStructural.C

```
//-------------------------------------------------------------------
//
// Elastic-Plastic Linear Hardening Material Model
//
//-------------------------------------------------------------------

const Prsr::Identifier ELASTIC_PLASTIC( material, structural, 200 );

Apub::MaterialInput & ep = root.nested_input( ELASTIC_PLASTIC,
                                              property_completion );

Smod::General_Solid_Support::self().parser_blk_map["ELASTIC_PLASTIC"] = & ep;

ep.property_command( YOUNGS_MODULUS );
ep.property_command( POISSONS_RATIO );
ep.property_command( BULK_MODULUS );
ep.property_command( SHEAR_MODULUS );
ep.property_command( LAMBDA );
ep.property_command( TWO_MU );
ep.property_command( YIELD_STRESS );
ep.property_command( Prsr::Identifier( material, structural, 201 ) );
ep.property_command( Prsr::Identifier( material, structural, 202 ) );
```

# LAME and Adagio/Presto


- After a model is implemented – it must be tested
  - Model developer/implementer must test model
  - Model developer/implementer owns all of the mechanics
  - It should be easy!
- Impossible to foresee all needs
  - If interface does not have what you need...
    - Modify interface on LAME side
    - Modify interface on Adagio/Presto side
- We want a flexible interface that is ***easy*** to use