SANDIA REPORT
SAND2007-5515
Unlimited Release
Printed August 2007

Library of Advanced Materials for Engineering – LAME

William M. Scherzinger
Daniel C. Hammerand

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

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States 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@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd.
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2007-5515
Unlimited Release
Printed August 2007

Library of Advanced Materials for Engineering – LAME

William M. Scherzinger and Daniel C. Hammerand
Solid Mechanics – 1524
Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-MS0372

Abstract

Constitutive modeling is an important aspect of computational solid mechanics. Sandia National Laboratories has always had a considerable effort in the development of constitutive models for complex material behavior. However, for this development to be of use the models need to be implemented in our solid mechanics application codes. In support of this important role, the Library of Advanced Materials for Engineering (LAME) has been developed in Engineering Sciences. The library allows for simple implementation of constitutive models by model developers and access to these models by application codes. The library is written in C++ and has a very simple object oriented programming structure. This report summarizes the current status of LAME.

ACKNOWLEDGMENTS

The authors would like to acknowledge the help of a number of people at Sandia National Laboratories. The Adagio and Presto code development teams, including Arne Gullerud, Kendall Pierson, Jason Hales and Nathan Crane have been especially helpful in the development of LAME and the interface between Strumento and LAME. A large portion of the original implementation of LAME was written by William Gilmartin. The SNTools team, especially Mark Hamilton and Kevin Brown, have helped with code management issues. A number of constitutive model developers, including Bob Chambers, Mike Neilsen and Shane Schumacher, have given very useful feedback on the design of LAME. Finally, analysts that have been willing to use LAME, Jeff Gruda, Matthew Neidigk and Frank Dempsey, have also helped guide its design.

CONTENTS

FIGURES

TABLES

1. INTRODUCTION

Engineering Sciences at Sandia National Laboratories has a long history of solid mechanics code development and analysis in support of the Laboratory's mission. A large part of the code development effort in Engineering Sciences has been on developing and implementing constitutive models for large deformation, solid mechanics codes. Two codes – transient dynamic and quasi-static – have generally been used for analysis, depending on the physics that is important to the problem. For relatively fast events, those that occur in the range of ms, a transient dynamic code is needed. For this purpose a long line of codes, from HONDO [1] in the 1970's to PRONTO2D [2] and PRONTO3D [3] in the 1990's to the current ASC code Presto [4], has been developed and used at Sandia. For engineering problems on a longer time scale, those that occur in the range of seconds up to centuries, quasi-static codes have been developed. A few quasi-static codes, with different solution algorithms including nonlinear conjugate gradients and dynamic relaxation, were developed in the 1980's and early 1990's. These incuded JAC [5], JAC3D[6], and SANTOS [7]. These different solution algorithms were eventually combined into one code, JAS3D [8], and they are now found in the ASC code Adagio [9].

For either transient dynamic or quasi-static analyses, much of the physics in a problem is done in the model for the material behavior. These constitutive models can be fairly simple or very complex depending on the material that needs to be modeled and the loading regime involved. The need for new constitutive model development to support Sandia's mission led to the development of a relatively simple interface in our solid mechanics codes for the implementation of constitutive models. Following well documented step by step procedures in the source code, a model developer could implement a model in a relatively short time. In fact, one reason for the existence of our solid mechanics codes was as a platform for constitutive model development. As noted by Taylor and Flanagan [3], "The development of PRONTO was motivated by the need for a code which could serve as a testbed for research into numerical algorithms and new constitutive models for nonlinear materials."

As successful a system as this was, constitutive model development, and more importantly use, still had a number of drawbacks. One drawback was that models needed to be implemented in each code separately. If an analyst wanted to use a model for a transient dynamic analysis, but the model only existed in the quasi-static code, then the model would have to be re-implemented in the transient dynamic code or the analyst would be simply out of luck. Furthermore, if a model was in each code, implementation differences might cause the two versions of the model to behave differently. Bug fixes to a model were another source of code drift between a model in the transient dynamic code and the quasi-static code. Another problem with model implementation in our legacy codes was the reliance on a specific hypoelastic formulation. While this does not, in general, limit what types of constitutive models can be implemented, it does limit the efficiency of other types of models. As our material modeling capability has grown, these alternative formulations are increasingly more useful. Finally, another important consideration in constitutive model implementation and development was the use and development of other Sandia codes that use solid mechanics constitutive models. These codes (e.g. ALEGRA, CTH) could not use models implemented in our codes, which required them to

be re-implemented again. This problem led to the development of the Model Interface Guidelines (MIG) by Brannon and Wong [10]. These guidelines have been used by ALEGRA and CTH, but they have not been adopted in the ASC solid mechanics codes Presto and Adagio.

With the advent of our ASC codes, Presto and Adagio, we have had the chance to re-examine how we do constitutive modeling in our codes. This re-evaluation was driven by a number of factors, not least of which was the difficulty for many model developers to program in an object oriented environment. While the advantages of object oriented programming (OOP) are many, for most constitutive model routines a procedural method of coding is often far simpler. This procedural way of programming, usually in FORTRAN, agrees quite well with how problems are formulated and solved. However, even though the desire to simplify the actual constitutive model implementation with core routines written in FORTRAN, we still did not wish to loose the benefits of an object oriented design. A simple class definition along with some inheritance would go a long way toward a flexible interface between the constitutive model and the finite elements. Thinking of these needs, an interface between the constitutive models and the host codes begins to take shape. In some cases this interface is sharp and well defined; in other cases the interface is less well defined. In either case, the interface can be used to divide the roles of the host code and the constitutive models. If the roles of the constitutive models can be defined, then a library can be created that multiple codes can use. This is a useful development for Engineering Sciences since it helps isolate constitutive model development. Constitutive model implementation is easier for model developers; application code developers have consistent model implementations across their codes; analysts have access to the same models in different codes and the latest improvements/developments from constitutive model developers. Recognizing the usefulness of this we have developed the Library of Advanced Materials for Engineering (LAME). Currently LAME includes only three-dimensional solid material models. Future work will extend LAME to incorporate structural models needed for the structural elements (e.g., beams, plates, and shells) in our ASC codes.

2. INTERFACE TO LAME

LAME is a constitutive model library. With any code library, there needs to be a well defined interface between the application code (host code) and the library. The interface for LAME is used by a host code to access constitutive models. It is also used by LAME to access some features of the host code. This host code can be any solid mechanics code, e.g. a finite element code, a finite difference code or a material model driver. In order to develop a useful interface, an object oriented design using C++ has been developed for LAME. The description that follows covers the standard interface to models in the LAME library. Although not documented here, work is currently underway to be able to implement ITAR and CRADA protected material models in LAME.

2.1. Overview of the Object Oriented Interface

The interface for LAME is principally defined through a base class in C++. This base class is declared in the header file **`include/models/Material.h`**. A few other classes are also used in the interface: the **`MaterialCreator`** class, the **`app_interface`** class and the **`function`** class. The interface provides a simple, well defined way for the host code to use LAME. The details of a specific model implementation are left to the material model itself, which is derived from the **`Material`** base class.

The interface has two principal aspects that need to be understood. The first are the various methods on the base class. These methods define how the host code will use the constitutive models and provide access to the host code for the constitutive models. These methods can be found in the header file. There are four principal methods that the rest of the interface is designed around

```
int initialize();
int loadStepInit();
int getStress();
int pcElasticModuli();
```

These four methods perform constitutive model tasks for the host code. If a constitutive model needs to initialize parameters, e.g. initializing state variables, then this task is done with the **`initialize()`** method. If a model does not need to initialize anything, then this method will not be implemented on the derived class and the host code will call the **`initialize()`** method – a virtual method – on the base class – a method that does nothing. The only method that every model will implement is the **`getStress()`** method. This method returns the current stress and state variables for the constitutive model. The base class also has method that access the host code. This is done through the **`app_interface`** class and the **`function`** class. A more detailed look at the methods available in LAME, along with their implementation, will be presented below.

A second aspect of the interface involves the actual variables that are passed between the host code and the constitutive models. Each constitutive model will require different variables for its

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

Figure 1: The `matParams` structure in LAME

calculations. So, in addition to having base class methods that define what the material models can do for the host code, a struct is defined that is used for passing arguments between the host code and LAME. This struct – called **`matParams`** – is also defined in the header file, **`Material.h`**. The **`matParams`** structure is shown in Figure 1. There are other variables that can be passed between the host code and LAME that are not shown in Figure 1; the use of many of these other variables is still under development. A detailed description of the variables in the **`matParams`** structure that are shown in Figure 1 will be presented later.

The material properties are passed to the material model as a reference to a **`MatProps`** object. **`MatProps`** is defined in the header file, include**`/models/Material.h`**, as a **`typedef`**

```
typedef map< string, vector<double> > MatProps;
```

So **`MatProps`** is really just a map that associates a string – the material property name – with a vector of doubles – the material properties. The material properties are stored in a vector since a material property might have a number of material properties associated with a given name.

It is important to understand these two aspects of the design of LAME. There are *methods* that are common to all constitutive models and *data* that is common to all constitutive models. A particular constitutive model may not need to use a particular method or some of the data that is passed – but the methods and data defined in the base class will provide all of the methods and data that a mechanics code or a constitutive model could need. If there is some method or data that is not currently in the interface – it will be easy to add. A new method – with a default implementation – can be added in the base class and new data can be added to the structure. Of course any modifications to the interface in the library, if they are used by a mechanics code, will require some modifications in the host code.

2.1.1 The `matParams` *Structure*

Before presenting the methods that the interface uses it is useful to understand what data will be passed between the application code and the constitutive models. The **`matParams`** structure provides this data. A brief description of the more relevant terms in the **`matParams`** structure follows:

`int nelements` : the number of elements that need a material evaluation

`int nintg` : the number of integration points per element

`double dt` : the time step

`double time` : the total solution time

`double * strain_rate` : a pointer to the un-rotated rate of deformation for the time step.

`double * stress_old` : a pointer to the un-rotated Cauchy stress at time t_n.

`double * stress_new` : a pointer to the un-rotated Cauchy stress at time t_{n+1}.

`double * state_old` : a pointer to the state variables for the model at time t_n.

`double * state_new` : a pointer to the state variables for the model at time t_{n+1}.

`double * temp_old` : a pointer to the temperature at time t_n.

`double * temp_new` : a pointer to the temperature at time t_{n+1}.

`double * left_stretch` : a pointer to the left stretch tensor at time t_{n+1}.

`double * rotation` : a pointer to the rotation tensor at time t_{n+1}.

There are other quantities that can be passed between a host code and LAME. These other quantities are generally specific to certain solution algorithms in Adagio or they are quantities that are currently being tested for a specific model implementation. Once we are convinced that their use and implementation are robust they will be documented.

It is worth noting here that the interface for the constitutive models has been developed from the constitutive model implementation in Sandia National Laboratories' legacy solid mechanics codes. The objective stress rate assumed by this implementation is a Green-McInnis stress rate. Therefore, as in our legacy codes, the stress that is passed to LAME – **`stress_old`** – is the un-

rotated Cauchy stress at time t_n. From continuum mechanics the un-rotated stress at time t_n is $\mathbf{T}_n$

$$\mathbf{T}_n = \mathbf{R}_n^T \cdot \boldsymbol{\sigma}_n \cdot \mathbf{R}_n \tag{1}$$

where $\boldsymbol{\sigma}_n$ is the Cauchy stress at time t_n and $\mathbf{R}_n$ is the rotation, derived from the polar decomposition of the deformation gradient, at time t_n. Similarly, the rate of deformation and the stress at time t_{n+1} – **`strain_rate`** and **`stress_new`** – are also un-rotated. They have the same form as the stress in (1) except that the rotation is evaluated at time t_{n+1}

$$\begin{aligned} \mathbf{d} &= \mathbf{R}_{n+1}^T \cdot \mathbf{D} \cdot \mathbf{R}_{n+1} \\ \mathbf{T}_{n+1} &= \mathbf{R}_{n+1}^T \cdot \boldsymbol{\sigma}_{n+1} \cdot \mathbf{R}_{n+1} \end{aligned} \tag{2}$$

where $\mathbf{D}$ is the rate of deformation and $\mathbf{d}$ is the un-rotated rate of deformation. Note that it is not specified when the rate of deformation is evaluated. This is because, in general, it varies depending on what the finite element code calculates as the rate of deformation. Sometimes the velocity gradient is calculated with respect to the mid-step configuration (incremental objectivity) and sometimes it is calculated as the time average of the logarithmic strain over a time step (strong incremental objectivity). So, in general we will not specify what is used for the rate of deformation. In any case it should be understood that the rate of deformation and the rotation that is used to un-rotate it may not be consistent. This is a minor detail that has been largely overlooked in the literature (see [11]).

Finally, it must also be realized that any internal state variables that depend on the configuration will also need to be stored in the un-rotated configuration – e.g. vectors and tensors. This could have a significant effect on output among other things.

It is necessary for a constitutive model developer to understand the details above since our current implementation locks us into a specific type of constitutive model formulation – i.e. a hypoelastic Green-McInnis rate. If a constitutive model developer wants to implement a hyperelastic model it is important that they realize that the stress that the application code uses is assumed to be the un-rotated Cauchy stress. Furthermore, if a comparison is made against another code that uses a different stress rate – e.g. a Jaumann stress rate – then slight differences should be expected in the two results.

In using **`matParams`**, the struct is actually created in the host code. This is done by including the header file **`include/models/Material.h`** in the host code where the constitutive model will be called. The code that creates **`matParams`** would look something like:

```
lame::matParams parameters;

. . .
```

```
int nelem, nintg;
double dt, time;
double * strain_rate;
double * stress_old, stress_new;
double * state_old, state_new;
double * rotation, left_stretch;

. . .

parameters.nelem = nelem;
parameters.nintg = nintg;
parameters.dt = dt;
parameters.time = time;
parameters.strain_rate = strain_rate;
parameters.stress_old = stress_old;
parameters.stress_new = stress_new;
parameters.state_old = state_old;
parameters.state_new = state_new;
parameters.rotation = rotation;
parameters.left_stretch = left_stretch;
```

Of course in this example the values for the number of elements, number of integration points, time step, solution time, the un-rotated rate of deformation, the rotation and the left stretch tensor need to be calculated by the host code before they are passed to LAME. Other variables, like **`stress_new`** and **`state_new`**, are calculated by the constitutive model and returned to the host code; these pointers need to point to memory that has been allocated for these values. It should also be noted that the terms above (except for the names of the variables in the structure, e.g. **`stress_old`**) are free to be called whatever makes sense in host code. All that matters is that the variables correspond to what the variables physically need to be for LAME – e.g. **`rotation`** in the **`matParams`** struct *must* point to the rotation tensor at time t_{n+1}.

2.1.2 The Application Interface

The application interface used by LAME is defined through the **`app_interface`** class. This class can be found in **`Material.h`**. The class provides a hook for LAME to access the host code. The host code will create a class that is derived from **`app_interface`** and then pass a pointer to the class back to LAME. The pointer that is passed back will be stored as a static variable – so that there will be only one. LAME will then have a pointer to the object that was created in the host code and every model will have access to that pointer.

The main purpose for the application interface is to pass messages for output, warning messages and error messages back to the host code. This is especially important for error messages since the host code, which may be running on a number of processors, will have a preferred way to terminate the code. LAME should not, under any circumstances, terminate the code.

The **`app_interface`** class in LAME only has one method, reportError

```
virtual void reportError( int error_code,
                          const char * message,
                          const into message_length );
```

The method is a public method that is also virtual. The method will be re-implemented in the host code in a derived class. The integer value is an error code that will have one of three values: 0 is an informational message, 1 is a warning message, and 2 is an error that will print a message and terminate the code.

2.1.3 Function Evaluations

The function interface used by LAME is defined through the **`function`** class. This class can be found in **`Material.h`**. This class provides a hook for LAME to access functions that are defined in the host code. The host code will create a class that is derived from **`function`** and then pass a pointer to the class back to LAME. The pointer that is passed back will be stored as a static variable – so that there will be only one. LAME will then have a pointer to the object that was created in the host code and every model will have access to that pointer.

The purpose for the function interface is to allow LAME to use the function evaluation capabilities of the host code. This is useful for any model that has user input functions to describe the material response. It is assumed that a host code will read and store any functions related to the constitutive model and have a capability to evaluate these functions.

The **`function`** class in LAME has the following methods

```
virtual void registerFunction( int fnum,
                               char * name,
                               char * mat_name,
                               int length,
                               int mat_length );

virtual void evaluateFunction( int fnum,
                               double & x,
                               double & y );
```

The first method registers a function. The function name is read in the material input and this method assigns an integer to that function. This vastly speeds up function evaluations in the constitutive models.

The second method is used to evaluate a function. Using the integer assigned to the function in the **`registerFunction`** method to access that function, this method evaluates the function for given abscissa values, **`x`**, it returns the ordinate values, **`y`**.

Since both methods are virtual, they need to be re-implemented in the host code in a derived class.

2.2 Data on the `Material` Base Class

All data on the **`Material`** base class is protected data. There is no public data – all data that needs to be accessed by an external application will be accessed through a public method. There is no private data either. If there were this data would only be accessible by the base class, and not any derived class. We didn't see a need for this in our design.

2.2.1 Protected Data

The following data is considered as necessary for the constitutive model. This data is all protected data so that classes (i.e. actual constitutive models) that are derived from the **`Material`** base class will have access to this data – they can set the data and use the data.

2.2.1.1 `properties`

The **`properties`** variable is a pointer to the material property array

```
double * properties;
```

Each constitutive model will populate its own material property array based on input received from the host code (which is obtained from the user input). That is, the parsing of the material properties from the input deck is performed by the host code and passed via a **`MatProps`** object which must be populated by the host code. This material data is then stored in the **`properties`** array in the LAME material model. Furthermore, based on the exact material property input, the size of this array can vary. For example, if a constitutive model uses a series expansion, the size of the material properties array will vary depending on the number of terms that are used on the input deck.

2.2.1.2 `num_material_properties`

The variable **`num_material_properties`** is an integer that holds the number of material properties in the property array

```
int num_material_properties;
```

Depending on the exact user input, the number of material properties, and thus the size of the material property array, can vary.

2.2.1.3 `num_state_vars`

The variable **`num_state_vars`** is an integer that holds the number of state variables for the model

```
int num_state_vars
```

Depending on the exact user input, the number of state variables can vary.

2.2.1.4 `state_variable_map`

The variable **`state_variable_map`** is a map that pairs state variable names with indices in the state variable array

```
map<string,int> state_variable_map;
```

This map can be used by the host code to output state variables through the public method **`getStateVariableIndex()`** which returns the state variable number corresponding to the string matching a particular state variable alias.

As an example, suppose a plasticity model has a state variable for the equivalent plastic strain. If this state variable is the first state variable in the state variable array we would have

```
state_variable_map["EQPS"] = 1;
```

The map is set through the protected method **`set_state_variable_alias()`**.

2.2.1.5 `p_function`

In order to evaluate functions in LAME, a pointer to a **`function`** object is stored in **`p_function`**

```
static lame::function * p_function;
```

The function class is in the **`lame`** namespace. The actual object is created in the host code and derived from the **`function`** class.

2.2.1.6 `p_host`

In order to output error messages from LAME, a pointer to the host code is needed. This is done through an **`app_interface`** object. A pointer to this object is stored in **`p_host`**

```
static lame::app_interface * p_host;
```

The application interface is in the **`lame`** namespace. The actual object is created in the host code and derived from the **`app_interface`** class.

2.3 Methods on the `Material` Base Class

The methods that are defined on the **`Material`** base class provide two things: an interface to the application code and utilities for the constitutive models that are derived from the base class. The interface methods are public methods. These methods are accessible by any code with access to an object from the library. The utility methods are protected methods. These methods are accessible by any object that is derived from the base class.

2.3.1 Public Methods

There are a number of public methods that define the interface in the **`Material`** base class. These methods define what the host code can expect from the constitutive model and what the constitutive model can expect from the host code. They also define the tasks that an arbitrary constitutive model needs to do. If a specific constitutive model does not need a specific task, for example an elasticity model will not need an initialization routine at the start of a load step. In this case the model will not implement the method and a default method will be used. These default methods are virtual methods defined on the base class. In all cases the virtual methods do nothing; what they provide is a common interface for the application code.

2.3.1.1 The `initialize()` Method

The **`initialize()`** method provides a way for a constitutive model to set internal state variables. The prototype for the **`initialize()`** method is

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

Since the number of state variables depends on the number of material points in the mechanics code – e.g. in a finite element code this would depend on the number and type of elements – the initialization can not be done in the constructor for the model. In the constructor, the number of state variables the each material point needs is specified, but the allocation of that memory is done by the application code. Once the memory has been allocated, the constitutive model can initialize that memory through the **`initialize()`** method. While it is possible to compute additional properties for a given material in the **`initialize()`** method, such property completion is typically performed in the constructor of each material model object. However, if those additional properties are needed in the host code, they typically are passed back as state variables via the **`matParams`** object. Further code developments for performing this type of functionality more elegantly are under consideration.

2.3.1.2 The `loadStepInit()` Method

The **`loadStepInit()`** method provides a way for a constitutive model to initialize state variables at the beginning of a load step. The prototype for the **`loadStepInit()`** method is

```
int loadStepInit( matParams * p );
```

Some constitutive models require the calculation of parameters for a specific load step. One example is a property that depends on the temperature. With the loose coupling between our thermal and solid mechanics codes, the temperature at the beginning and end of a time step is usually known before a solid mechanics time step is performed. In the loadStepInit() method we can calculate any temperature dependent properties that need to be calculated. The benefit is that these properties are calculated once per time step – not once per iteration. This can make a big difference in a quasi-static analysis. The temperature dependent properties are stored as state variables that can be used in the constitutive model computations.

2.3.1.3 The `getStress()` Method

The **`getStress()`** method does what a constitutive model is developed for – returning the stress given a strain input. The prototype for the **`getStress()`** method is

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

In this method a constitutive model is called. This model can be written in FORTRAN, C or C++. The code for the model will exist in a separate file. Only the call to the constitutive model will be in the **`getStress`** method, with the possible exception of some pre or post processing steps that put the data into the proper form for the constitutive model. An example of this would be if a constitutive model is written with only one stress and state variable array. Then a preprocessing step might be to copy the **`stress_old`** array into the **`stress_new`** array prior to sending the data to the constitutive model. In a similar way the **`state_old`** array would be copied into the **`state_new`** array.

2.3.1.4 The `pcElasticModuli()` Method

ADAGIO [9] uses a nonlinear preconditioned conjugate method in solving quasi-static solid mechanics problems and incorporates several choices for the preconditioner. One such choice is an isotropic linear elastic preconditioner. Typically, the moduli for such a preconditioner are set once at the beginning of an analysis, but sometimes this is not sufficient. For instance, material models with temperature-dependent moduli can have significant moduli changes that should be incorporated into the elastic preconditioner for rapid iterative convergence. Likewise, some models have time-dependent scaling of the bulk and/or shear responses of the materials in order to improve convergence in Adagio and the changes in these scalings need to be reflected in the elastic preconditioner when it is being used. The **`pcElasticModuli()`** method calculates and returns these updated moduli. The prototype for the **`pcElasticModuli()`** method is

```
int pcElasticModuli( matParams * p );
```

2.3.2 Protected Methods

There are a number of protected methods on the base class that define utilities that constitutive models can use. For example, constitutive models will need to get their material properties during construction. There is a protected method on the base class that does this task.

2.3.2.1 `getNumberProps()` Method

Some material properties are contained in arrays, for example the coefficients of a series expansion. For materials with these properties there may be a variable number of terms that are read in from the input deck. This creates the need to dynamically allocate memory for these materials. The method **`getNumberProps()`** returns the number of properties associated with a given property name

```
int getNumberProps( string name,
                    const MatProps & props );
```

This method takes the name of the material property as a string along with the **`MatProps`** map. This method is very useful for determining the number of properties that will go into the material property array. This method is used in the constructor for a constitutive model.

2.3.2.2 `getMaterialProperty()` Method

Since the material properties are held as an array in a map, LAME has a method that extracts the property from the array. Many properties only have one element in the array while some have a variable number of elements. For this reason the **`getMaterialProperty()`** is overloaded with an implementation for both cases

```
double getMaterialProperty( string name,
                            const MatProps & props );

double getMaterialProperty( string name,
                            const MatProps & props,
                            int n );
```

The first method extracts the value for the material property from **`props`** when there is only one material property. When the array in **`props`** has more than one value, the second method extracts the "**n**th" value in the array. This method is used in the constructor for a constitutive model.

2.3.2.3 `set_state_variable_alias()` Method

While the material properties are ultimately held in the specific object that is created for the material model, the state variables are created and held by the host code. This is done since the total number of state variables is dependent on things outside of LAME, e.g. the number of elements in the problem. The host code, therefore, will have access to all of the state variables but will not generally know what they represent. This can pose a problem for output. For this reason there is a state variable alias in LAME that allows the host code to determine which state variable is which. While this is used for output, the capability could be used for other things in the host code.

The **`set_state_variable_alias()`**

```
int set_state_variable_alias( string name,
                              int pos );
```

associates the indices for a state variable with a state variable name. The state variable name is passed in as a **`string`** while the location in the state variable is passed in as an **`int`**. This method is used in the constructor for a constitutive model.

2.4 Creating a Material

The principal concept behind the interface between LAME and a host code is that the material models all look the same to the host code. This is accomplished by passing from LAME, to the host code, a pointer to a **`Material`** object. The actual object that the pointer points to could be an **`Elastic`** object, a **`Viscoplastic`** object or a **`Hyperfoam`** object, etc; the host code will not know exactly what object the pointer points to. One issue associated with this is that all material models must have all of the possible methods that the host code might use. This is accomplished by defining virtual functions in the base class and redefining them in the derived classes. This was described in Section 2.3.

A second issue associated with this idea involves creating the object in the first place. Before we can pass a pointer back to the host code we need to actually create the correct object. In object oriented programming this can be done using a factory method. This is accomplished in LAME through what may only be loosely considered a factory method. A lot of the subtlety of a factory method is not seen in LAME since we wanted to design something that is easy to use for virtually anyone. But the basic concept is the same in that it provides a solution to the problem of creating a specific derived class.

2.4.1 `MaterialCreator` *Class*

The **`MaterialCreator`** class defines an object that creates a constitutive model and returns a pointer to that model. Most classes based on a factory method in C++ are more elegant than the MaterialCreator class, that is why we say that it is "loosely" a factory method.

The principal method on the **`MaterialCreator`** class is the **`create_material()`** method. The **`create_material()`** method has the following signature

```
Material * create_material( const string & material_name,
                            const MatProps & props) const;
```

The **`create_material()`** method is passed the name of the constitutive model and the material properties. The material name is expected as a reference to a string. The material properties are expected as a reference to a **`MatProps`** object. As previously noted, **`MatProps`** is defined in the header file, **`include/models/Material.h`**, as a **`typedef`**

```
typedef map< string, vector<double> > MatProps;
```

Also recall that the property values are stored in a vector since a material property name might actually be associated with an array of values. For example, in a viscoelastic model a Prony series is often used to model relaxation. In general, a model developer will not know how many terms you need in your Prony series. This capability allows a developer to call a property, for example **`Prony_Coefficient`**, and expect a list of the coefficients. All of the Prony coefficient are now stored in the map as a vector of **`doubles`** that can be accessed with the keyword **`Prony_Coefficient`**.

3. MODELS CURRENTLY IMPLEMENTED

A number of models are already implemented in LAME. Some of these models have a complete implementation – the model is in LAME, it has been tested and all of the expected features work. These models, which have a complete implementation, and their tests are being documented in separate SAND reports [12,13]. Many models only have a partial, or initial, implementation. These models will have a more complete implementation in the future as they are completed and tested.

The models that are implemented are able to be used with the Sierra based application codes Adagio and Presto. In fact, the models in LAME have close ties to Adagio and Presto in that many of the models were originally implemented in either the Sierra framework or Strumento. However, as the capabilities of LAME have increased, model implementation – and even development – has increasingly been done in LAME. The Cowper-Symonds model, the Low Density Foam model and the Thermoelastic-Plastic Power Law Hardening Weld model are examples of models that are only available in Adagio and Presto through LAME. Furthermore, the Universal Polymer Model is currently being developed in LAME, using Adagio.

Some capabilities, however, still need to be worked out for LAME. One such capability involves the use of arbitrary coordinate systems. While some orthotropic models are in LAME, these models have coordinate systems that are fixed for an element block. Orthotropic elastic models that were implemented in Strumento are able to have coordinate systems that are a function of position. Working out how to use these coordinate systems in LAME has not been done. It may be a simple as what has been done with function evaluations, where the host code is accessed from LAME to evaluate coordinate systems, but until the problem is examined in more detail we will not know.

Another capability that is needed in LAME is the ability to compile FORTRAN 90 code. Many constitutive models are written in FORTRAN 90 and we would like to implement these models in LAME. However, LAME is a part of the Sierra system in SNTools and FORTRAN 90 is not supported on all of their platforms. Therefore, we need to find some way of compiling the FORTRAN 90 code when we have access to a F90 compiler and not compiling the code when we do not have a F90 compiler. This can probably be done through pre-processor definitions, but we have not had the time to look into the problem. We have had success compiling FORTRAN 90 code on our desktop machines, so there is no big technical barrier to this problem – just a time barrier.

Table 1. Models that are currently implemented in LAME

Model	Initial Implementation	Complete Implementation
BCJ	yes	yes
BCJ_MEM	yes	yes
Bulk	yes	no
Cowper-Symonds	yes	no
Ductile Fracture	yes	yes
Elastic	yes	yes
Elastic Fracture	yes	yes
Elastic-Plastic	yes	yes
EP Power Law	yes	yes
Foam Plasticity	yes	yes
Incompressible Solid	yes	yes
Johnson Cook	yes	yes
Honeycomb	yes	yes
Hyperfoam	yes	yes
Isotropic Geomodel	yes	no
Low Density Foam	yes	no
Multilinear EP	yes	yes
Multilinear EP with Failure	yes	yes
Neo-Hookean	yes	yes
NLVE Polymer	yes	yes
NLVE Thermoset	yes	yes
Orthotropic Crush	yes	yes
Orthotropic Rate	yes	yes
Piezoelectric	yes	no
Power Law Creep	yes	yes
Shape Memory	yes	no
Soil and Foam	yes	yes
Solder	yes	yes
Solder with Damage	yes	yes
Stiff Elastic	yes	yes
Thermoelastic	yes	yes
Thermo EP Power Law	yes	yes
Thermo EP Power Law Weld	yes	yes
Mooney-Rivlin	yes	yes
Swanson	yes	yes
Universal Polymer	yes	yes
Viscoelastic Swanson	yes	yes
Viscoplastic	yes	yes
Viscoplastic Foam	yes	yes

4. CONCLUSIONS

The Library of Advanced Materials for Engineering (LAME) provides a library with a simple interface to a solid mechanics application code. Furthermore, the interface for implementing a model is also extremely simple and provides support for a wide range of constitutive models. The amount of object oriented programming knowledge required for implementing a material model in LAME is minimal and easy to pattern match. These aspects of LAME make the library very versatile and a useful platform for constitutive model development in support of Sandia National Laboratories' mission.

A few capabilities that are needed still need to be implemented, and certainly more capabilities will arise as LAME sees more use. The current library has a flexible interface and it is believed that future capabilities can be provided relatively easily. In the future, an extension to LAME to structural models is being planned. Finally, current work is already underway on incorporating ITAR and CRADA protected models into LAME.

6. REFERENCES

1. Key, S.W., Beisinger, Z.E., and Krieg, R.D., H*ONDO-2: A Finite Element Computer Program for the Large Deformation Dynamic Response of Axisymmetric Solids*, SAND78-0422, Sandia National Laboratories, Albuquerque, NM, October 1978.
2. Taylor, L.M. and Flanagan, D.P., *PRONTO 2D: A Two-Dimensional Transient Solid Dynamics Program*, SAND86-0594, Sandia National Laboratories, Albuquerque, NM, March 1987.
3. Taylor, L. M. and Flanagan, D.P., *PRONTO 3D: A Three-Dimensional Transient Solid Dynamics Program*, SAND87-1912, Sandia National Laboratories, Albuquerque, NM, March 1989.
4. Koteras, J.R., Gullerud, A.S., Crane, N.K., and Hales, J.D., *Presto User's Guide Version 2.6*, SAND2006-6093, Sandia National Laboratories, Albuquerque, NM, October 2006.
5. Biffle, J.H., *JAC – A Two-Dimensional Finite Element Computer Program for the Nonlinear Response of Solids with the Conjugate Gradient Method*, SAND81-0998, Sandia National Laboratories, Albuquerque, NM, April 1984.
6. Biffle, J.H., *JAC3D – A Three-Dimensional Finite Element Computer Program for the Nonlinear Quasi-Static Response of Solids with the Conjugate Gradient Method*, SAND87-1305, Sandia National Laboratories, Albuquerque, NM, February 1993.
7. Stone, C.M., *SANTOS – A Two-Dimensional Finite Element Program for the Quasistatic, Large Deformation, Inelastic Response of Solids*, SAND90-0543, Sandia National Laboratories, Albuquerque, NM, July 1997.
8. Blanford, M.L., Heinstein, M.W., Key, S.W., 'JAS3D – A Multi-Strategy Iterative Code for Solid Mechanics Analysis Users' Instructions, Release 2.0', Memo, Sandia National Laboratories, Albuquerque, NM, September 2001.
9. Pierson, K.H. and Hales, J.D., 'ADAGIO/ANDANTE User's Guide Version 2.0', Memo, Sandia National Laboratories, Albuquerque, NM, September 2004.
10. Brannon, R.M. and Wong, M.K., MIG Version 0.0: Model Interface Guidelines: Rules to Accelerate Installation of Numerical Models into any Compliant Parent Code, SAND96-2000, Sandia National Laboratories, Albuquerque, NM, August 1996.
11. Flanagan, D.P. and Taylor, L.M., "An Accurate Numerical Algorithm for Stress Integration with Finite Rotations," *Computer Methods in Applied Mechanics and Engineering*, 62, pp. 305-320, 1987.
12. Scherzinger, W.M. and Hammerand, D.C., *Constitutive Models in LAME*, SAND07-XXXX, Sandia National Laboratories, Albuquerque, NM, XXXX 2007.
13. Scherzinger, W.M. and Hammerand, D.C., *Testing of Constitutive Models in LAME*, SAND07-XXXX, Sandia National Laboratories, Albuquerque, NM, XXXX 2007.

APPENDIX A: EXAMPLE MODEL IMPLEMENTATION

An example model implementation of a linear-hardening elastic-plastic model is presented here. Implementation of the linear-hardening elastic-plastic model involves three files. The first is the header file **`ElasticPlastic.h`** which sets the function signatures for the methods from the **`Material`** base class which are replaced with material specific functions (**`initialize`** and **`getStress`**). The signatures for the FORTRAN routines that are called by these methods are also given there. The second file is the implementation file **`ElasticPlastic.C`** which implements the constructor, destructor, and **`initialize`** and **`getStress`** methods. In the constructor, the material properties are retrieved from the **`MatProps`** map and stored in the properties array and aliases are set for the state variables. The destructor eliminates the storage space that was used for the **`properties`** array. Note that the **`initialize`** and **`getStress`** C++ methods serve to extract the necessary data from the **`matParams`** structure in calling the FORTRAN routines which perform the necessary numerical calculations. Finally, the FORTRAN subroutines associated with the **`initialize`** method which initializes the state variables for this model and **`getStress`** method which does the actual stress calculation are given in **`elastic_plastic.F`**.

A.1 ElasticPlastic.h

```
#ifndef _ELASTIC_PLASTIC_H_
#define _ELASTIC_PLASTIC_H_

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

namespace lame {

  /**
   *
   * This is the class for the elastic-plastic linear hardening
   * constitutive model.  This mode is a hypoelastic model that
   * uses an elastic predictor with a radial return method to
   * integrate the flow rule.
   *
   * \f$
   *     D_{ij} = D_{ij}^{e} + D_{ij}^{p}
   * \f$
   *
   * \f$
   *     \sigma_{ij}^{n+1} = \sigma_{ij}^{n}
   *     + \Delta t \left( \lambda \delta_{ij} D_{kk}^{e}
   *                       + 2\mu D_{ij}^{e} \right)
   * \f$
   *
   * \f$
   *     D_{ij}^{p} = \gamma \partial\phi/\partial\sigma_{ij}
   * \f$
   *
   */

  class ElasticPlastic : public Material{

  public:

    ElasticPlastic( MatProps props ) ;
    ~ElasticPlastic() ;
```

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

  private:

    //
    // private and unimplemented to prevent use
    //

    ElasticPlastic( const ElasticPlastic & );
    ElasticPlastic & operator= ( const ElasticPlastic & );

  } ;

  // ********************************************************************
  //
  // 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

#endif
```

A.2 ElasticPlastic.C

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

namespace lame {

  //********************************************************************
  //
  // This is the constructor for 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
  //
  //********************************************************************
```

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

  //
  // Material Property Definitions
  //

  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);

  //
  // State Variable Definitions
  //

  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);

}

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

ElasticPlastic::~ElasticPlastic(){

  delete [] properties;
  properties = NULL;

}

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

int ElasticPlastic::initialize(  matParams * p ) {

  LAME_FORTRAN(elastic_plastic_initialize)( p->nelements,
                                            properties,
                                            p->state_old,
                                            p->state_new );

  return 0;

}

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

```
  //*************************************************************

  int ElasticPlastic::getStress( matParams * p ) {

    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;

  }

} // lame
```

A.3 elastic_plastic.F

```
      subroutine elastic_plastic_get_stress(nelem,dt,props,
     *     strain_rate,stress_old,stress_new,state_old,state_new)
c
c*********************************************************************
c*********************************************************************
c
c description:
c   elastic plastic material model.
c
c formal parameters - input:
c     nelem         int   number of elements in the workset
c     dt            real  time increment
c     strain_rate   real  strain rate
c     stress_old    real  stress
c
c formal parameters - output:
c     stress_new    real  stress
c
c*********************************************************************
c
#include <lame_precision.par>
#include <lame_numbers.par>
#include <lame_type_sizes.par>
c
c make some shorter aliases for tensor component parameters
c
      parameter( kxx = 1, kyy = 2, kzz = 3,
     *           kxy = 4, kyz = 5, kzx = 6 )
c
      dimension stress_old(6,nelem),stress_new(6,nelem)
      dimension state_old(8,nelem),state_new(8,nelem)
      dimension strain_rate(6,nelem)
      dimension props(5)
c
      youngs   = props(1)
      pr       = props(2)
      yield_stress = props(3)
      hard_mod     = props(4)
      beta         = props(5)
c
      twog     = youngs/(one+pr)
      alambda  = twog*pr/(one-two*pr)
c
      twogdt = twog * dt
      alamdt = alambda * dt
```

```
c
      threeg = twog / two3rds
      ratio = sqrt( two3rds ) * beta * hard_mod
      term = one / ( twog * ( one + hard_mod / threeg ) )
      ohard = ( one - beta ) * two3rds * hard_mod
      radius0 = sqrt( two3rds ) * yield_stress
c
      do k=1,nelem
c
c compute new stress
c
         traced = strain_rate(kxx,k)
     *          + strain_rate(kyy,k)
     *          + strain_rate(kzz,k)
c
         stress_new(kxx,k) = stress_old(kxx,k) + alamdt * traced
     *                     + twogdt * strain_rate(kxx,k)
         stress_new(kyy,k) = stress_old(kyy,k) + alamdt * traced
     *                     + twogdt * strain_rate(kyy,k)
         stress_new(kzz,k) = stress_old(kzz,k) + alamdt * traced
     *                     + twogdt * strain_rate(kzz,k)
         stress_new(kxy,k) = stress_old(kxy,k)
     *                     + twogdt * strain_rate(kxy,k)
         stress_new(kyz,k) = stress_old(kyz,k)
     *                     + twogdt * strain_rate(kyz,k)
         stress_new(kzx,k) = stress_old(kzx,k)
     *                     + twogdt * strain_rate(kzx,k)
c
c trial stress measured from the back stress
c
         s1 = stress_new(kxx,k) - state_old(2,k)
         s2 = stress_new(kyy,k) - state_old(3,k)
         s3 = stress_new(kzz,k) - state_old(4,k)
c
         smean = ( s1 + s2 + s3 )/3.0
c
         ds1 = s1 - smean
         ds2 = s2 - smean
         ds3 = s3 - smean
         ds4 = stress_new(kxy,k) - state_old(5,k)
         ds5 = stress_new(kyz,k) - state_old(6,k)
         ds6 = stress_new(kzx,k) - state_old(7,k)
c
         dsmag2 = ds1**2 + ds2**2 + ds3**2
     *          + two * ( ds4**2 + ds5**2 + ds6**2 )
c
         radius = radius0 + state_old(1,k) * ratio
         r2 = radius * radius
c
         if ( dsmag2.gt.r2 ) then
c
c trial stress outside yield surface
c
            dsmag = sqrt(dsmag2)
            diff = dsmag - radius
            xlam = term * diff
            state_new(1,k) = state_old(1,k) + xlam * sqrt(two3rds)
            state_new(8,k) = sqrt(two3rds)*yield_stress
     *           + ratio*state_new(1,k)
c
c update back stress
c
            factor = ohard * xlam / dsmag
            state_new(2,k) = state_old(2,k) + factor * ds1
            state_new(3,k) = state_old(3,k) + factor * ds2
            state_new(4,k) = state_old(4,k) + factor * ds3
            state_new(5,k) = state_old(5,k) + factor * ds4
            state_new(6,k) = state_old(6,k) + factor * ds5
            state_new(7,k) = state_old(7,k) + factor * ds6
c
c update stress
```

```
c
            factor = twog * xlam / dsmag
            stress_new(kxx,k) = stress_new(kxx,k) - factor * ds1
            stress_new(kyy,k) = stress_new(kyy,k) - factor * ds2
            stress_new(kzz,k) = stress_new(kzz,k) - factor * ds3
            stress_new(kxy,k) = stress_new(kxy,k) - factor * ds4
            stress_new(kyz,k) = stress_new(kyz,k) - factor * ds5
            stress_new(kzx,k) = stress_new(kzx,k) - factor * ds6
c
         else
c
            state_new(1,k) = state_old(1,k)
c
            state_new(2,k) = state_old(2,k)
            state_new(3,k) = state_old(3,k)
            state_new(4,k) = state_old(4,k)
            state_new(5,k) = state_old(5,k)
            state_new(6,k) = state_old(6,k)
            state_new(7,k) = state_old(7,k)
c
            state_new(8,k) = state_old(8,k)
c
         endif
c
      enddo
c
      return
      end
c
c*******************************************************************
c*******************************************************************
c
      subroutine elastic_plastic_initialize( nelem, props,
     *     state_old, state_new)
c
c*******************************************************************
c*******************************************************************
c
c description:
c   initialization routine for the elastic/plastic material model
c   with power law hardening.
c
c formal parameters - input:
c     nelem         int   number of elements in the workset
c     props         real  array of material properties
c     state_old     real  equivalent plastic strain
c
c formal parameters - output:
c     state_new     real  equivalent plastic strain
c
c*******************************************************************
c
#include <lame_precision.par>
#include <lame_numbers.par>
#include <lame_type_sizes.par>
c
      character*80 message
c
      dimension state_old(8,nelem),state_new(8,nelem)
      dimension props(5)
c
c Error checking
c
      hard_mod = props(4)
c
      if (hard_mod.lt.0.d0) then
         write(message,101)
         call lame_report_error(3,message)
      endif
c
c Set initial state variables
```

```
c
      yield_stress = props(3)
      radius = sqrt(two3rds)*yield_stress
c
      do k = 1,nelem
c
         do i = 1,7
            state_old(i,k) = zero
            state_new(i,k) = zero
         enddo
c
         state_old(8,k) = radius
         state_new(8,k) = radius
c
      enddo
c
 101  format('Hardening modulus is less than zero')
c
      return
      end
```

Distribution

1	MS0346	Tom Baca	1523
1	MS0372	Lupe Arguello	1525
1	MS0372	Joe Bishop	1525
1	MS0372	Robert Chambers	1524
3	MS0372	Daniel Hammerand	1524
1	MS0372	John Pott	1524
1	MS0372	Jim Redmond	1525
3	MS0372	William Scherzinger	1524
1	MS0372	Mike Stone	1525
1	MS0372	Leah Tuttle	1524
1	MS0372	Gerald Wellman	1525
1	MS0380	Manoj Bhardwaj	1542
1	MS0380	Nathan Crane	1542
1	MS0380	Arne Gullerud	1542
1	MS0380	Jason Hales	1542
1	MS0380	Martin Heinstein	1542
1	MS0380	Joe Jung	1542
1	MS0380	Richard Koteras	1542
1	MS0380	Hal Morgan	1540
1	MS0380	Kendall Pierson	1542
1	MS0380	Vicki Porter	1542
1	MS0380	Garth Reese	1542
1	MS0380	Ben Spencer	1542
1	MS0380	Tim Walsh	1542
1	MS0555	Rod May	1522
1	MS0557	Dave Clauss	1521
1	MS0660	Chris Lamb	9512
1	MS0836	Shane Schumacher	1516
1	MS0847	Mike Neilsen	1526
1	MS0847	Pete Wilson	1520
1	MS1070	Channy Wong	1526
2	MS9018	Central Technical Files	8945-1
2	MS0899	Technical Library	4536