`matmodlab`
a Material Model Laboratory

Tim Fuller

September 30, 2013

Chapter 1

Introduction to `matmodlab`

`matmodlab` (Material Model Laboratory) is a suite of tools whose purpose is to aid in the rapid development and testing of material models. `matmodlab` is made up of several components, the most notable being the Material Model Driver `mmd`. `mmd` can be thought to drive a single material point of a finite element simulation through very specific user designed paths. This permits exercising material models in ways not possible in finite element calculations, desgining verification and validation tests of the material response, among others. `matmodlab` is a small suite of tools at the developers disposal to aid in the design and implementation of material models in larger finite element host codes. It is also a useful tool to analysists for understanding and parameterizing a material's response to deformation.

The core of the `matmodlab` code base is written in Python and leverages Python's object oriented programming (OOP) design. OOP techniques are used throughout `matmodlab` to setup and manage simulation data. Computationally heavy portions of the code, and the material models themselves are written in Fortran for its speed and ubiquity in scientific computing. Calling Fortran procedures from Python is made possible by the `f2py` module, standard in Numpy, that compiles and creates Python shared object libraries from Fortran sources.

Output files from `matmodlab` simulations are in the `ExodusII` [?] database format, devloped at Sandia National Labs for storing finite element simulation data. Since `matmodlab` is designed to be used by material model developers, it is expected that the typical user will want access to *all* all available output from a material model, thus all simulation data is written to the output database. `ExodusII` database files can be post processed via

the `mmv` utility, in addition to other visualization packages such as `PARAVIEW` [?].
`matmodlab` is free software released under the MIT License.

1.1 Why a Single Element Driver?

Due to their complexity, it is often over kill to use a finite element code for constitutive model development. In addition, features such as artificial viscosity can mask the actual material response from constitutive model development. Single element drivers allow the constituive model developer to concentrate on model development and not the finite element response. Other advantages of the `matmodlab` (or, more generally, of any stand-alone constitutive model driver) are

- `matmodlab` is a very small, special purpose, code. Thus, maintaining and adding new features to `matmodlab` is very easy.
- Simulations are not affected by irrelevant artifacts such as artificial viscosity or uncertainty in the handling of boundary conditions.
- It is straightforward to produce supplemental output for deep analysis of the results that would otherwise constitute an unnecessary overhead in a finite element code.
- Specific material benchmarks may be developed and automatically run quickly any time the model is changed.
- Specific features of a material model may be exercised easily by the model developer by prescribing strains, strain rates, stresses, stress rates, and deformation gradients as functions of time.

1.2 Why Python?

Python is an interpreted, high level object oriented language. It allows for writing programs rapidly and, because it is an interpreted language, does not require a compiling step. While this might make programs written in python slower than those written in a compiled language, modern packages

and computers make the speed up difference between python and a compiled language for single element problems almost insignificant.
For numeric computations, the NumPy and SciPy modules allow programs written in Python to leverage a large set of numerical routines provided by LAPACK, BLASPACK, EIGPACK, etc. Python's APIs also allow for calling subroutines written in C or Fortran (in addition to a number of other languages), a prerequisite for model development as most legacy material models are written in Fortran. In fact, most modern material models are still written in Fortran to this day.
Python's object oriented nature allows for rapid installation of new material models.

1.3 Historical Background

When I was a graduate student at the University of Utah I had the good fortune to have as my advisor Dr. Rebecca Brannon. Prof. Brannon instilled in me the necessity to develop material models in small special purpose drivers, free from the complexities of larger finite element codes. To this end, I began developing material models in Prof. Brannon's `MED` driver (available upon request from Prof. Brannon). The `MED` driver was a special purpose driver for driving material models through predefined strain paths. After completing graduate school I began employment as a member of the Technical Staff at Sandia National Labs. Among the many projects I worked on was the development of material models for geologic applications. There, I found need to drive the material models through prescribed stress paths to match experimental records. This capability was not present in the `MED` and I sought a different solution. The solution came from the `MMD` driver, created years earlier at Sandia, by Tom Pucick. The `MMD` driver had the capability to drive material models through prescribed stress and strain paths, but also lacked many of the IO features of the `MED`. And so, for some time I used both the `MED` and `MMD` drivers in applications that suited their respective strengths. After some time using both drivers, I decided to combine the best features of each in to my own driver. Both the `MED` and `MMD` drivers were written in Fortran and I decided to write the new driver in Python so that I could leverage the large number of builtin libraries. The Numpy and Scipy Python libraries would be used for handling most number crunching. The new driver came to be known as `payette`. `payette` added many unique capabilities and became a capable piece of software used by other staff members at Sandia. But, `payette` suffered from the fact that it was my first foray in to programming with Python. After some time, the bloat and bad programming practices with `payette` caused me to spend a few weekends re-writing it in to what is now known as `matmodlab`.

1.4 Obtaining `matmodlab`

`matmodlab` is an open source project licensed under the MIT license. A copy of may be obtained from `https://github.com/tjfulle/matmodlab`

1.5 About This Guide

`matmodlab` is developed as a tool for developers and analysts who care to understand the responses of material models to specific deformation paths. The target audience is assumed to have a basic knowledge of continuum mechanics and familiarity with other finite element codes. Accordingly, concepts of continuum mechanics and finite element methods are not described in detail and programing techniques are also not described.

Chapter 2

`matmodlab` Quick Start Guide

This guide provides an outline for building and running `matmodlab`.
Build `matmodlab` See Chapter 3.

- Download `matmodlab` and setup environment
- $ `cd $MMLROOT/toolset && ./setup.py`
- $ `buildmtls`

Prepare Input Inputs are xml specification files. See Chapters 6 - 9.

- Set up the desired simulation path.
- Add material model.
- Add desired extraction requests.

Run

- $ `mmd` [options] *runid* [,*runid_1*, ..., *runid_n*]
  runid is prefix of ".xml" file.
- Complete list of options given by
  $ `mmd -h`

Postprocess

- $ `mmv` *runid* [,*runid_1*, ..., *runid_n*]
- PARAVIEW also reads exodus files.

Chapter 3

Building `matmodlab`

`matmodlab`'s code base is largely written in Python and requires no additional compiling. However, the `ExodusII` third party library and material models written in fortran must be built.

3.1 System and Software Requirements

`matmodlab` has been built and tested extensively on several versions of linux and the Apple Mac OSX operating systems. It is unknown whether or not `matmodlab` will run on Windows.
`matmodlab` requires the following software installed for your platform:

- Python 2.7
- NumPy 1.6
- SciPy 0.10
- A fortran compiler

The required software may be obtained in several ways, though all development has been made using Enthought Canopy (`http://http://www.enthought.com`).
A note on the fortran compiler. It is recommended to use the same fortran compiler to build the `matmodlab` components that was used to build SciPy.

3.2 Installation

Ensure that all `matmodlab` prerequisites are installed and working properly before proceeding.

3.2.1 Set Environment and Path

`MMLROOT`	Optional, name of installation directory
`PATH`	`$MMLROOT/toolset:$PATH`
`MMLMTLS`	":" separated list of paths to directories containing user defined material models. See Section 10.2.
`MMLTESTS`	":" separated list of paths to directories containing user defined regression tests. See Section **??**.

3.2.2 Set Up

Set up and build the TPLs.

```
$ cd $MMLROOT/toolset
$ python setup.py
```

In addition to building the TPLs, `setup.py` generates the following executable scripts

`buildmtls`	Build material models
`mml`	Run `matmodlab` simulations
`exdump`	Read a `matmodlab` output and dumps requested variables to ascii columnar files
`mmv`	2D plots of `matmodlab` output
`runtests`	Run the regression tests

Each script is a wrapper to another `matmodlab` Python file. In the wrapper, relevant environment variables are set (e.g., `$PYTHONPATH`) and the correct Python executable (the one used to set up) is used to interpret the `matmodlab` source file. The full set of options for each script is obtained by

```
$ scriptname -h
```

where `scriptname` is the name of the script.
The TPLs will build the first time `matmodlab` is setup. Thereafter after, only the executable scripts are rewritten. Execute

```
$ python setup.py -h
```

for options to rebuild the TPLs.

3.2.3 Build

Build the material libraries

```
$ buildmtls
```

3.2.4 Test the Installation

To test `matmodlab` after installation, execute

```
$ runtests [-j N]
```

which will run the `matmodlab` regression tests.

3.2.5 Troubleshooting

If you experience problems when building/installing/testing `matmodlab`, you can ask help from the `matmodlab` developers. Please include the following information in your message:

- Platform information OS, its distribution name and version information etc.

  ```
  $ python -c "import os,sys;print os.name,sys.platform"
  $ uname -a
  ```

- Information about C,C++,Fortran compilers/linkers as reported by the compilers when requesting their version information, e.g., the output of

```
$ gcc -v
$ gfortran --version
```

- Python version

  ```
  $ python -c "import sys;print sys.version"
  ```

- NumPy version

  ```
  $ python -c "import numpy;print numpy.__version__"
  ```

- SciPy version

  ```
  $ python -c "import scipy;print scipy.__version__"
  ```

- Feel free to add any other relevant information.

Chapter 4

matmodlab Solution Method

matmodlab exercises a material model directly by “driving” it through user specified paths using a designated driver. Currently installed drivers are the solid and eos drivers. For each driver type, matmodlab computes an increment in deformation for a given step and requires that the material model update the stress in the material to the end of that step, given the current state and an increment in deformation. Because of the similarity of the material model interface in matmodlab with many commercial finite element codes, transitioning material models developed and tested in matmodlab to full finite element codes should be an easy process. In this chapter, the role and importance of the material model in a finite element procedure is reviewed. The solution method adopted by each driver in matmodlab is then described and compared with that of finite elements.

4.1 The Role of the Material Model in Continuum Mechanics

4.1.1 Conservation Laws

Conservation of mass, momentum, and energy are the central tenets of the analysis of the response of a continuous media to deformation and/or load. Each conservation law can be summarized by the following statement

$$\boxed{\begin{array}{c}\text{Time rate of}\\ \text{change of}\\ \text{quantity}\end{array}} = \boxed{\begin{array}{c}\text{Rate of}\\ \text{production}\\ \text{in the}\\ \text{interior}\end{array}} + \boxed{\begin{array}{c}\text{Flux}\\ \text{through the}\\ \text{boundary}\end{array}}$$

Mathematically, the conservation laws for a point in the continuum are

- Conservation of mass
$$\dot{\rho} + \rho \boldsymbol{\nabla} \cdot \dot{\boldsymbol{u}} = 0$$
- Conservtion of momentum per unit volume
$$\rho \frac{d}{dt} \dot{\boldsymbol{u}} = \underbrace{\boxed{\boldsymbol{\nabla} \cdot \boldsymbol{\sigma}}}_{\text{internal forces}} + \underbrace{\boxed{\boldsymbol{b}}}_{\text{body forces}}$$
- Conservation of energy per unit volume
$$\rho \frac{d}{dt} U = \underbrace{\boxed{\rho s}}_{\text{heat source}} + \underbrace{\boxed{\boldsymbol{\sigma} : \dot{\boldsymbol{\varepsilon}}}}_{\text{strain energy}} + \underbrace{\boxed{\boldsymbol{\nabla} \cdot \boldsymbol{q}}}_{\text{heat flux}}$$

where $\boldsymbol{u}$ is the displacement, ρ the mass density, $\boldsymbol{\sigma}$ the stress, $\dot{\boldsymbol{\varepsilon}}$ the rate of strain, $\boldsymbol{b}$ the body force per unit volume, $\boldsymbol{q}$ the heat flux, s the heat source, and U is the internal energy per unit mass.
In solid mechanics, mass is conserved trivially, and many problems are adiabatic or isotrhermal, so that only the momentum balance is explicitly solved

$$\rho \frac{d}{dt} \dot{\boldsymbol{u}} = \underbrace{\boxed{\boldsymbol{\nabla} \cdot \boldsymbol{\sigma}}}_{\text{internal forces}} + \underbrace{\boxed{\boldsymbol{b}}}_{\text{body forces}} \tag{4.1}$$

The balance of linear momentum is the continuum mechanics generalization of Newton's second law $F = ma$.
The first term on the RHS of (4.1) represents the internal forces, which arise in the medium to resist imposed deformation. This resistance is a fundamental response of matter and is given by the divergence of the stress field.
The balance of linear momentum represents an initial boundary value problem for applications of interest in solid dynamics:

$$\begin{aligned} \rho \frac{d}{dt} \dot{\boldsymbol{u}} = \boldsymbol{\nabla} \cdot \boldsymbol{\sigma} + \boldsymbol{b} & \qquad \text{in } \Omega \\ \boldsymbol{u} = \boldsymbol{u}_0 & \qquad \text{on } \Gamma_0 \\ \boldsymbol{\sigma} \cdot \boldsymbol{n} = \boldsymbol{t}^{(n)} & \qquad \text{on } \Gamma_t \\ \dot{\boldsymbol{u}}(\boldsymbol{x}, 0) = \dot{\boldsymbol{u}}_0(\boldsymbol{x}) & \qquad \text{on } \boldsymbol{x} \in \Omega \end{aligned} \tag{4.2}$$

4.1.2 The Finite Element Method

The form of the momentum equation in (4.2) is termed the **strong** form. The strong form of the initial BVP problem can also be expressed in the weak form by introducing a test function $\boldsymbol{w}$ and integrating over space

$$
\begin{aligned}
\int_\Omega \boldsymbol{w}\cdot\left(\boldsymbol{\nabla}\cdot\boldsymbol{\sigma} + \boldsymbol{b} - \rho\frac{d}{dt}\dot{\boldsymbol{u}}\right) d\Omega & \qquad \forall \boldsymbol{w} \\
\boldsymbol{u} = \boldsymbol{u}_0 & \qquad \text{on } \Gamma_0 \\
\boldsymbol{\sigma}\cdot\boldsymbol{n} = \boldsymbol{t}^{(n)} & \qquad \text{on } \Gamma_t \\
\dot{\boldsymbol{u}}\left(\boldsymbol{x},0\right) = \dot{\boldsymbol{u}}_0\left(\boldsymbol{x}\right) & \qquad \text{on } \boldsymbol{x}\in\Omega
\end{aligned}
\tag{4.3}
$$

Integrating (4.3) by parts allows the traction boundary conditions to be incorporated in to the governing equations

$$
\begin{aligned}
\int_\Omega \rho\boldsymbol{w}\cdot\boldsymbol{a} + \boldsymbol{\sigma}:\boldsymbol{\nabla}\boldsymbol{w}\, d\Omega = \int_\Omega \boldsymbol{w}\cdot\boldsymbol{b}\, d\Omega + \int_\Gamma \boldsymbol{w}\cdot\boldsymbol{t}^{(n)}\, d\Gamma_t & \quad \forall \boldsymbol{w} \\
\boldsymbol{u} = \boldsymbol{u}_0 & \qquad \text{on } \Gamma_0 \\
\dot{\boldsymbol{u}}\left(\boldsymbol{x},0\right) = \dot{\boldsymbol{u}}_0\left(\boldsymbol{x}\right) & \qquad \text{on } \boldsymbol{x}\in\Omega
\end{aligned}
\tag{4.4}
$$

This form of the IBVP is called the **weak** form. The weak form poses the IBVP as a integro-differential equation and eliminates singularities that may arise in the strong form. Traction boundary conditions are incorporated in the governing equations. The weak form forms the basis for finite element methods.
In the finite element method, forms of $\boldsymbol{w}$ are assumed in subdomains (elements) in Ω and displacements are sought such that the force imbalance R is minimized:

$$
R = \int_\Omega \boldsymbol{w}\cdot\boldsymbol{b}\, d\Omega + \int_\Gamma \boldsymbol{w}\cdot\boldsymbol{t}^{(n)}\, d\Gamma_t - \int_\Omega \rho\boldsymbol{w}\cdot\boldsymbol{a} + \boldsymbol{\sigma}:\boldsymbol{\nabla}\boldsymbol{w}\, d\Omega \tag{4.5}
$$

The equations of motion as described in (4.5) are not closed, but require relationships relating $\boldsymbol{\sigma}$ to $\boldsymbol{u}$

$$\boxed{\text{Constitutive model} \longrightarrow \text{relationship between } \boldsymbol{\sigma} \text{ and } \boldsymbol{u}}$$

In the typical finite element procedure, the host finite element code passes to the constitutive routine the stress and material state at the beginning of a finite step (in time) and kinematic quantities at the end of the step.

The constitutive routine is responsible for updating the stress to the end of the step. At the completion of the step, the host code then uses the updated stress to compute kinematic quantities at the end of the next step. This process is continued until the simulation is completed. The host finite element handles the allocation and management of all memory, including memory required for material variables.

4.2 `matmodlab` Solution Procedure

In addition to providing a platform for material model developers to formulate and test constitutive routines, `matmodlab` aims to provide users of material models an independent platform to exercise, parameterize, and compare material responses against single element finite element simulations. To this end, the solution procedure in `matmodlab` is similar to that of the finite element method, in that the host code (`matmodlab`) provides to the constitutive routine a measure of deformation at the end of a finite step and expects the updated stress in return. However, rather than solve the momentum equation at the beginning of each step and advancing kinematic quantities to the step's end, `matmodlab` retrieves updated kinematic quantities from user defined tables and/or functions.

The path through which a material is exercised is defined by piecewise continuous "legs" in which components of the "control type" c_i are specified at discrete points in time, shown in Figure 4.1. The c_i are used to obtain a sequence of piecewise constant strain rates that are used to advance the kinematic state. Supported control types are strain, strain rate, stress, stress rate, deformation gradient, displacement, and velocity. "Mixed-modes" of strain and stress (and their rates) are supported. Components of displacement and velocity control are applied only to the "+" faces of a unit cube centered at the coordinate origin.

The components of strain are defined by

$$\boldsymbol{\varepsilon} = \frac{1}{\kappa}\left(\boldsymbol{U}^{\kappa} - \boldsymbol{\delta}\right) \tag{4.6}$$

where $\boldsymbol{U}$ is the right Cauchy stretch tensor, defined by the polar decomposition of the deformation gradient $\boldsymbol{F} = \boldsymbol{R}{\cdot}\boldsymbol{U}$, and κ is a user specified "Seth-Hill" parameter that controls the strain definition. Choosing $\kappa = 2$ gives the Lagrange strain, which might be useful when testing models cast in

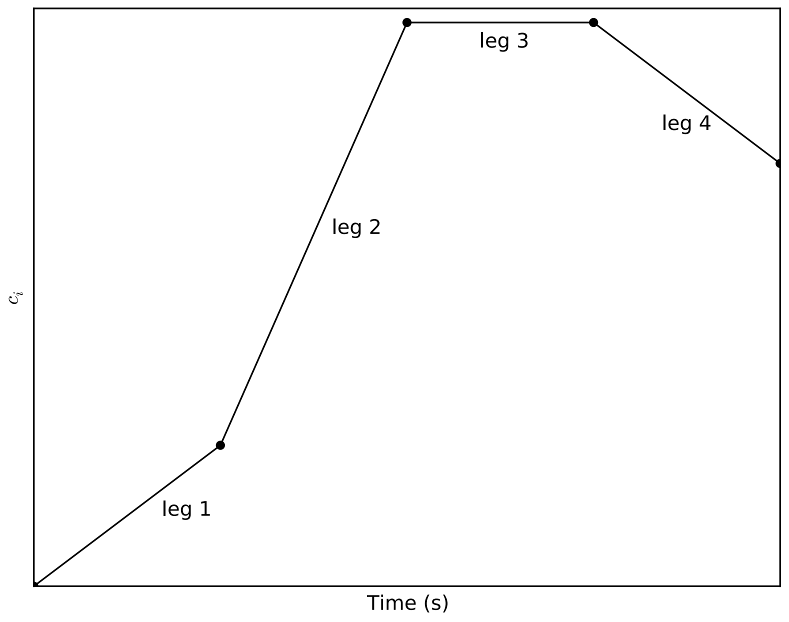

Figure 4.1: User defined path for the i^{th} component of "c". c may represent strain, strain rate, stress, stress rate, deformation gradient, displacement, or velocity.

a reference coordinate system. The choice $\kappa = 1$, which gives the engineering strain, is convenient when driving a problem over the same strain path as was used in an experiment. The choice $\kappa = 0$ corresponds to the logarithmic (Hencky) strain. Common values of κ and the associated names for each (there is some ambiguity in the names) are listed in Table 4.2.

κ	Name(s)
-2	Green
-1	True, Cauchy
0	Logarithmic, Hencky, True
1	Engineering, Swainger
2	Lagrange, Almansi

The volumetric strain ε_v is defined

$$\varepsilon_v = \begin{cases} \frac{1}{\kappa}\left(J^\kappa - 1\right) & \text{if } \kappa \neq 0 \\ \ln J & \text{if } \kappa = 0 \end{cases} \tag{4.7}$$

where the Jacobian J is the determinant of the deformation gradient.
Each leg in the control table, from time $t = 0$ to $t = t_f$ is subdivided into a user-specified number of steps and the material model evaluated at each step. When volumetric strain, deformation gradient, displacement, or velocity are specified for a leg, `matmodlab` internally determines the corresponding strain components. If a component of stress is specified, `matmodlab` determines the strain increment that minimizes the distance between the prescribed stress component and model response.

4.2.1 Strain Rate from Prescribed Stress

The approach to determining unknown components of the strain rate from the prescribed stress is an iterative scheme employing a multidimensional Newton's method that satisfies

$$\boldsymbol{\sigma}\left(\dot{\boldsymbol{\varepsilon}}\left[\mathrm{v}\right]\right) = \boldsymbol{\sigma}^{p}$$

where, v is a vector subscript array containing the components for which stresses (or stress rates) are prescribed, and $\boldsymbol{\sigma}^{p}$ are the components of prescribed stress.
Each iteration begins by determining the submatrix of the material stiffness

$$\mathbb{C}_{\mathrm{v}} = \mathbb{C}\left[\mathrm{v}, \mathrm{v}\right]$$

where $\mathbb{C}$ is the full stiffness matrix $\mathbb{C} = d\boldsymbol{\sigma}/d\boldsymbol{\varepsilon}$. The value of $\dot{\boldsymbol{\varepsilon}}\left[\mathrm{v}\right]$ is then updated according to

$$\dot{\boldsymbol{\varepsilon}}\left[\mathrm{v}\right] = \dot{\boldsymbol{\varepsilon}}\left[\mathrm{v}\right] - \mathbb{C}_{\mathrm{v}}\boldsymbol{:}\boldsymbol{\sigma}^{*}(\dot{\boldsymbol{\varepsilon}}\left[\mathrm{v}\right])/dt$$

where

$$\boldsymbol{\sigma}^{*}(\dot{\boldsymbol{\varepsilon}}\left[\mathrm{v}\right]) = \boldsymbol{\sigma}(\dot{\boldsymbol{\varepsilon}}\left[\mathrm{v}\right]) - \boldsymbol{\sigma}^{p}$$

The Newton procedure will converge for valid stress states. However, it is possible to prescribe invalid stress state, e.g. a stress state beyond the material's elastic limit. In these cases, the Newton procedure may not converge to within the acceptable tolerance and a Nelder-Mead simplex method is used as a back up procedure. A warning is logged in these cases.

4.2.2 Solid Driver

As the name implies, the `solid` driver is designed to exercise the type of material models encountered in solid mechanics. The solution method is similar to that of many finite element codes, so that material models developed and tested in `matmodlab` can be easily transitioned to them.

4.2.3 Electrical

Electric field can be prescribed for testing piezoelectric models.

Chapter 5

Running Simulations with `matmodlab`

`matmodlab` is a command line driven programming through the `mmd` executable. To run a simulation with `matmodlab`, be sure that `$MMLROOT` is on your path and execute

```
$ mmd runid[.xml]
```

where `runid` is the basename of the input file. Input file formatting is covered in Chapters 6 - 9.
The following files will be produced by `mmd` in the current working directory

```
$ ls runid.*
runid.exo       runid.log       runid.xml
```

`runid.exo` is the ExodusII output database, `runid.log` the log file, and `runid.xml` the input file.
For a complete list of options, execute

```
$ mmd -h
```

Chapter 6

User Input: Overview

User input is via xml control files. In general, tags use CamelCase and attributes lower case. Attributes are described in this document as

```
attr="type[default]{choices}"
```

where `default` is the default value (if any) and {`choices`} are valid choices (if any). Any attribute not having a default value is required. Types are `str`, `int`, `real`, `list`. Lists are given as space separated lists (e.g., ”1 2 3”). In the following, elements shown in red are required input.

6.1 MMLSpec

All input files must have as their root element `<MMLSpec>`.

```
<MMLSpec>
```

Recognized subelements of `<MMLSpec>` are

- `<Physics>`
- `<Permutation>`
- `<Optimization>`

The following elements are read from any scope in the input file

- `<Include>`

- `<Function>`

The `<Physics>`, `<Permutation>`, `<Optimization>` and input blocks are described separately in their own chapters.

6.2 Preprocessing

Preprocessing allows specifying variables in the input inside of comment tags for use in other parts of the input. Syntax mirrors that of `aprepro`. Preprocessor also evaluates (nearly) any Python expression.
The `random()` expression generates a random number. The `random_seed` variable sets the random state seed. Note, expressions are evaluated in order, therefore, if setting the `random_seed` it should occur early.
The following input stub demonstrates specifying the `<Material>` parameter `K` and `G`, and `<Path>` parameter `estar` as variables

```
<MMLSpec>
  <!-- {random_seed = 7}
       {G = 54e9 * random()}
       {K = 23e9}
       {estar = -.05}
  -->
  <Physics>
    <Material model="elastic">
      <K> {K} </K>
      <G> {G} </G>
    </Material>
    <Path type="prdef" estar="{estar}">
      ...
    </Path>
  </Physics>
</MMLSpec>
```

6.3 Include

Path to file to be included as if its contents were inplace in the input file

```
<Include href="str"/>
```

The following stub input demonstrates how to include a file in place

```
<Include href="/path/to/some/file.ext"/>
```

6.4 Function

Define functions to be used elsewhere in input. `id=0` and `id=1` are reserved for the constant 0 and 1 functions, respectively. `href` is the path to a file containing the function definition (useful when the function is a large piecewise linear table). `cols` specifies the columns in which data is located in a piecewise linear table.

```
<Function id="int"
          type="str{analytic_expression, piecewise_linear}"
          var="str[x]" href="str[]" cols="list[1 2]">
```

The following input stub demonstrates how to define an analytic expression and piecewise linear table as functions

```
<Function id="2" type="analytic_expression" var="t">
  sin(t)
</Function>
<Function id="3" type="piecewise_linear">
  1 2
  2 3
  3 5
</Function>

<!-- Read a piecewise linear table from an external file using
     columns 1 and 3
-->
<Function id="4" type="piecewise_linear" href="./file.dat"
          cols="1 3"/>
```

```
$ cat file.dat
# Column1 Column2 Column3
```

```
1 1 4
2 3 7
.
.
.
100 4.2 1.43
```

Chapter 7

User Input: Physics

Define the physics of the simulation. If specified, `termination_time` defines the termination time for the simulation. If not specified, termination time is taken as final time in `<Path>`.

```
<Physics driver="str[solid]{solid, eos}"
         termination_time="real[]">
```

Recognized subelements of `<Physics>` are

- `<Path>`
- `<Material>`
- `<Extract>`

7.1 Path

Define deformation paths through with the material will be exercised.

```
<Path type="str{prdef, surface}"
      format="str[default]{default, table, fcnspec}"
      cols="list[1, ..., n]" cfmt="str"
      tfmt="str[time]{time,dt}"
      nfac="int[1]" kappa="real[0]" rstar="real[1]"
      tstar="real[1]" estar="real[1]" sstar="real[1]"
      amplitude="real[1]" ratfac="real[1]" href="str">
```

7.1.1 Path Attributes

type

The type of path specified. Valid types are `prdef` and `surface`.
The `prdef` type defines a prescribed deformation. The jth leg of `<Path>` is sent to the driver in form `[tf, n, cfmt, Cij]`, where `tf`, `n`, `cfmt`, and `Cij` are the termination time, number of steps, control format, and control values. Methods of inputing legs depends on the attributes of `<Path>` and will be shown in examples to follow.
The `surface` input is similar to the `prdef` specification, but leg termination time is not specified. Control parameters also differ, as shown in Table 7.2.

Format

The format by which the legs of the deformation path are specified. Valid formats are `default`, `table`, and `fcnspec`. In the following subsections, the different formats are described.

Format: default The `default` format offers the most control. In this format, the termination time, number of steps, control format, and components of deformation are specified for each leg as in the following stub input

```
<Path type="default">
  <!-- tterm nsteps cfmt c1 c2 c3 ... -->
  0   0 222222 0 0 0 0 0 0
  1  10 222222 1 0 0 0 0 0
</Path>
```

See Section 7.1.1 for a full description of the control format `cfmt` and its relationship with the `c1, c2, c3, ...`.

Format: table The `table` format allows reading in deformation paths from a columnar table of data. Control format is uniform for all legs. Specify control format as `cfmt` attribute of `<Path>`. Specify which columns to read data with the `cols` attribute. The first column is assumed to be the time specifier. See Section 7.1.1 for a description of the `cols` attribute. The `tfmt` attribute specifies if the time column represents the actual time (`tfmt="time"`) or time step (`tmft="dt"`). The number of steps for each leg

can be set by `nfac`. The `href` attribute specifies an external file to read the table.
The following input stubs demonstrate reading a table from the input file and from an external file.

```
<!-- Read entries from table. -->
<Path type="prdef" format="table" cols="1:4" cfmt="222"
      tfmt="time">
  0 0 0 0
  1 1 0 0
    ...
  n 2 0 0
</Path>
```

```
<!-- Read table from external file -->
<Path type="prdef" format="table" cols="1 3:8" cfmt="222222"
      tfmt="time" href="exmpls.tbl"/>
```

Format: fcnspec The `fcnspec` format allows defining a deformation path by a function. A deformation path defined by `fcnspec` must have only 1 leg defining the termination time and the function specifier defining the values of the components of deformation. The function specifier is of the form

```
function_id[:scale]
```

where `function_id` is the ID of the function as specified in its `<Function>` element. The optional scale is multiplied by the function.
The following input stub demonstrates uniaxial strain deformation, using a user defined function to specify the 11 component of strain through time.

```
<Path type="prdef" format="fcnspec" cfmt="222" nfac="200">
  <!-- termination time, fcn spec -->
  {2 * pi} 2:1.e-1 0 0
</Path>
```

Control Format

The control format `cfmt` is concatenated integer list specifying in its i^{th} component the i^{th} component of deformation, i.e., `cfmt[i]` instructs the driver

as to the type of deformation represented by `Cij[i]`. Types of deformation represented by `cfmt` are shown in Table 7.1.
For example, the following `cfmt` instructs the driver that the components of `Cij` represent [stress, strain, stress rate, strain rate, strain, strain], respectively:

```
cfmt="423122"
```

Mixed modes are allowed only for components of strain rate, strain, stress rate, and stress. Electric field components can be included with any deformation type.
The components `Cij` take the following order

Vectors: [X, Y, Z]

Symmetric tensors: [XX, YY, ZZ, XY, YZ, XZ]

Tensors: [XX, XY, XZ, YX, YY, YZ ZX, ZY, ZZ]
If `len(Cij)` $\neq$ `6` (or 9 for deformation gradient), the missing components are assumed to be zero strain.

`cfmt`	Deformation type
1	Strain rate
2	Strain
3	Stress rate
4	Stress
5	Deformation gradient
6	Electric field

Table 7.1: Supported deformation types and `cfmt` code for `solid prdef` paths

`cfmt`	Variable type
1	Density
2	Temperature

Table 7.2: Supported surface variable types and `cfmt` code for `eos surface` paths

Time Format

The `tfmt=["time"]{"time","dt"}` flag specifies the time format. If `tfmt="time"` (default) the first value of each leg is interpreted as the termination time for the leg. For `tfmt="dt"` the first value of each leg is interpreted as a time increment.

The Column Specifier

The columns to read from a table or the `fcnspec` leg are specified by the `cols` attribute. `cols` are specified as a space separated list of columns. Numbering is 1 based. Ranges can be specified using Python slice syntax.
The following input stubs demonstrate two equivalent ways to to read columns 1, 3, 4, 5, 8, 9, and 13 from a table.

```
cols="1 3 4 5 8 9 13"
```

```
cols="1 3:5 8:9 13"
```

kappa

`kappa` is the Seth-Hill strain definition parameter κ described in section 4.2.

Step Multiplier

`nfac` is a multiplier on the number of steps for each leg.

Amplitude

`amplitude` is a factor multiplied to all components of deformation.

The “star” Multipliers

`[rtes(ef)]star` are multipliers on the components of density, time (temperature for `type="surface"`), strain, stress, and electric field, respectively. The `[rtes(ef)]star` are first multiplied by `amplitude`.

Rate Factor

`ratfac` is a divisor to the termination time of each leg, thereby effectively increasing the rate of deformation.

7.1.2 More Examples

The following examples will help clarify the `<Path>` input syntax

```
<!-- uniaxial strain, all six components of strain
     prescribed -->
<Path type="prdef" kappa="0" tstar="1" estar="-.5"
      amplitude="1" ratfac="1">
  <!-- termination time, number of steps, cfmt, Cij -->
  0   0 222222 0 0 0 0 0 0
  1 100 222222 1 0 0 0 0 0
  2 100 222222 2 0 0 0 0 0
  3 100 222222 1 0 0 0 0 0
  4 100 222222 0 0 0 0 0 0
</Path>
```

```
<!-- uniaxial strain, stress controlled -->
<Path type="prdef" nfac="100">
  0 0 444 0 0 0
  1 1 444 -7490645504 -3739707392 -3739707392
  2 1 444 -14981291008 -7479414784 -7479414784
  3 1 444 -7490645504 -3739707392 -3739707392
  4 1 444 0 0 0
</Path>
```

```
<!-- uniaxial stress, mixed mode -->
<Path type="prdef" nfac="100">
  0 0 222 0 0 0
  1 1 244 {epsmax} 0 0
  4 1 244 0 0 0
</Path>
```

Example of `type="surface"`

The following examples demonstrate the `type="surface"`

```
<Path type="surface">
  <!-- nsteps, control, Cij -->
  <!-- control goes as 1 -> density
                      2 -> temperature -->
    0 12 1 100
  100 12 5 300
</Path>
```

```
<Path type="surface" format="table" cfmt="12" nfac="100">
  <!-- Cij -->
  1 100
  5 300
</Path>
```

7.2 Material

Define the material model.

```
<Material model="str">
```

Subelements of `<Material>` are

- `<Matlabel>`
- `<ParameterArray>`
- `<InitialState>`
- `<Key>`*

*`<Key>` is a valid material parameter name.

7.2.1 Material Attributes

model

The name of the material model.

7.2.2 Matlabel

Insert model parameters from a database file.

```
<Matlabel href="str[F_MTL_PARAM_DB]" material="str"/>
```

Matlabel Attributes

href The path to the database file. Defaults to `$MMLROOT/materials/material_properties.db` if no file is given.

material Name of material as given in the database file.
The following input stub demonstrates the use of `<Matlabel>`

```
<Material model="elastic">
  <Matlabel href="./materials.xml" material="aluminum"/>
</Material>
```

7.2.3 ParameterArray

Specify the parameter array for the material as whitespace separated list of floats. The list of values must be the same length as the parameter array for the material or an error will occur.

```
<ParameterArray>
  VAL1 VAL2 ... VALN
</ParameterArray>
```

7.2.4 InitialState

Specify the initial state of the material as a whitespace separated list of floats. Six stress values must be followed by material variables (if any). The length of the material variables must be the same as the length of the `xtra` variable array for the material or an error will occur. Note, implementation is material model specific.

```
<InitialState>
  STRESS_XX STRESS_YY ... STRESS_XZ XTRA1 XTRA2 ... XTRAN
</InitialState>
```

7.2.5 Specify Individual Parameters

Specify individual parameters as xml text nodes

```
<Key> float </Key>
```

`<Key>` is replaced by specific material model parameters. The following stub inputs demonstrate the `<Material>` input

```
<Material model="elastic">
  <G>  54E+09 </G>
  <K> 124E+09 </K>
</Material>
```

7.3 Extract

Extract variables and paths from ExodusII output and (optionally) write to different formats.

```
<Extract format="str[ascii]{ascii, mathematica, ndarray}"
         step="int[1]" ffmt="str[.18f]">
```

Recognized subelements of `<Extract>` are

- `<Path>`*
- `<Variables>`

* `eos` driver only

7.3.1 Extract Attributes

format

The format to write the output. `ascii` format writes out columnar data as an ascii text file, `mathematica` writes an ascii text file that can be read by Mathematica, and `ndarray` writes the data to a file in the numpy .npy binary format.

step

Extract every `step`th timestep.

ffmt

The string format used write out variables.

7.3.2 Variables

Variables to extract from the ExodusII output database. Variables are specified children of the `<Variables>` element. All components of vector and tensor variables will be extracted if only the basename is specified. Time is always extracted as the first entry of the output file. Extracted variables are in `runid.out` or `runid.math` depending if the format is ascii or mathematica.

```
<Variables>
  VAR_1, ..., VAR_N
</Variables>
```

The following example demonstrates how to extract all components of stress and strain

```
<Extract format="ascii">
  <variables>
    STRESS STRAIN
  </variables>
</Extract>
```

Extract only the XX, YY, and ZZ components of stress

```
<Extract format="ascii">
  <variables>
    STRESS_XX STRESS_YY STRESS_ZZ
  </variables>
</Extract>
```

Extract all variables

```
<Extract format="ascii">
  <variables>
    ALL
  </variables>
</Extract>
```

Path

Extract a specified path from the equation of state surface through the specified density range starting at the initial temperature.

```
<Path type="str{isotherm, hugoniot}" increments="int[100]"
      density_range="list" initial_temperature="real">
```

The following input stub demonstrates extracting Hugoniot and Isotherm paths

```
<Extract>
  <Path type="isotherm" increments="200"
        density_range="1 3" initial_temperature="225"/>
  <Path type="hugoniot" increments="100"
        density_range="1 3" initial_temperature="100"/>
</Extract>
```

Chapter 8

User Input: Permutation

Permutate model input parameters, running jobs with different realization of parameters. Ideal for investigating model sensitivities.

```
<Permutation method="str[zip]{zip,combine,shotgun}"
                    seed="real[date]"
                    correlation="list[none]{plot,table,none}">
```

Recognized subelements of `<Permutation>` are

- `<Permutate>`
- `<ResponseFunction>`

Each `<Permutation>` job creates a directory `runid.eval`

```
$ ls runid.eval
eval_0/    eval_2/    mml-evaldb.xml
eval_1/    ...        runid.log
```

The `eval_i` directory holds the output of the `i`th job, including `params.in` with the values of each permutated parameter for that job. `mml-tabular.xml` contains a summary of each job run. `mmv` recognizes `mml-tabular.xml` files.

8.1 Permutation Attributes

8.1.1 method

The `method` attribute describes which method to use to determine parameter combinations to run.

The `zip` method runs one job for each set of parameters (and, thus, the number of realizations for each parameter must be identical), the `combine` method runs every combination of parameters.

8.1.2 correlation

Create correlation table and plots of relating permutated parameters and value of response function. `correlation` is only meaningful if a `<ResponseFunction>` is specified.

8.1.3 seed

The seed for the random number generator. `date` is todays date in seconds.

8.2 Permutate

Specify the paramaters to permutate.

```
<Permutate var="str"
           values="func{range,list,weibull,uniform,
                      normal,percentage}"
```

8.2.1 Permutate Attributes

var

`var` is the name of the variabe and should occur elsewhere in the input file in preprocessing braces.

values

`values` are the specific values. The `range,list,weibull,uniform,normal,percentage` are all specified as functions with the following form

```
values="func(start,stop,N)"
```

The following input stub demonstrates how to permutate the `K` and `G` parameters

```
<Permutation method="zip" seed="12">
  <Permutate var="K" values="weibull(125.e9, 14, 3)"/>
  <Permutate var="G" values="percentage(45.e9, 10, 3)"/>
</Permutation>
```

In the `<Material>` element, the `K` and `G` parameters are specified as

```
<Material model="elastic">
  <K> {K} </K>
  <G> {G} </G>
</Material>
```

8.3 ResponseFunction

The `<ResponseFunction>` returns the response from permutation or optimization jobs.

```
<ResponseFunction href="str" function="mmlfcn"
                  descriptor="str[]"/>
```

One of `href` or `function` must be specified.

8.3.1 ResponseFunction Attributes

descriptor

`descriptor` is the name given to the response function in the output.

href

`href` is the path to an executable file script containing the response function. The script is called from the command line as

```
% ./scriptname runid.exo
```

An example of a response function specifying `href` is

```
<ResponseFunction href="./scriptname" descriptor="PRES"/>
```

function

`function` is the name of a builtin `matmodlab` response function. Built in response functions are

- `mml.max` maximum value of a simulation variable output
- `mml.min` minimum value of a simulation variable output
- `mml.mean` mean value of a simulation variable output
- `mml.ave` average value of a simulation variable output
- `mml.absmax` maximum absolute value of a simulation variable output
- `mml.absmin` minimum absolute value of a simulation variable output

Built in response functions operate only on variabes in the simulation output file.
An example of a response function specifying `function` is

```
<ResponseFunction function="mml.max(PRESSURE)"
                  descriptor="PRES"/>
```

Chapter 9

User Input: Optimization

Optimize specified parameters against user specified objective function.

```
<Optimization method="str[simplex]{simplex, powell, cobyla}"
              maxiter="int[25]" tolerance="real[1e-6]">
```

Recognized subelements of `<Optimization>` are

- `<Optimize>`
- `<ResponseFunction>`
- `<AuxiliaryFile>`

Like `<Permutation>` jobs, each `<Optimization>` job creates a directory `runid.eval`

```
$ ls runid.eval
eval_0/    eval_2/    mml-evaldb.xml    runid.log
eval_1/    ...        params.opt
```

The `eval_i` directory holds the output of the i$^{\mathrm{th}}$ job, including `params.in` with the values of each parameter for that job. `mml-tabular.xml` contains a summary of each job run. `mmv` recognizes `mml-tabular.xml` files. `params.opt` has the final, optimized, parameters.

9.1 Optimization Attributes

9.1.1 method

`method` specifies the optimization method. All optimization routines utilize the scipy.optimize module.

9.1.2 maxiter

`maxiter` is the maximum number of iterations.

9.1.3 tolerance

`tolerance` is the optimization tolerance.

9.2 Optimize

Specify the variable to be optimized.

```
<Optimize var="str" initial_value="real" bounds="list[]"/>
```

9.2.1 Optimize Attributes

var

`var` is the name of the variabe and should occur elsewhere in the input file in preprocessing braces.

initial_value

`initial_value` is the initial value of `var`

bounds

`bounds` specifies lower and upper bounds on `var`. Only the `cobyla` method accepts bounds.

9.3 AuxiliaryFile

Path to any auxiliary file needed by the optimization objective function.

```
<AuxiliaryFile href="str"/>
```

9.4 ResponseFunction

Same as for `<Permutation>`, except that auxiliary files are also passed to the function. The value returned from the response function is interpreted as the error to be minimized.
If the `<ResponseFunction>` is given by `href`, it is called as

```
% ./scriptname runid.exo [AuxFile1[AuxFile2[...]]]
```

9.5 Example

Optimize the `K` and `G` parameters

```
<Optimization method="simplex" maxiter="25" tolerance="1e-4">
  <ResponseFunction href="opt-sig-v-time"
                    descriptor="SIG_V_TIME"/>
  <AuxiliaryFile href="opt-baseline.dat"/>
  <Optimize var="opt_k" initial_value="129.e9"/>
  <Optimize var="opt_g" initial_value="54.e9"/>
</Optimization>
```

In the `<Material>` element, the `K` and `G` parameters are specified as

```
<Material model="elastic">
  <K> {opt_k} </K>
  <G> {opt_g} </G>
</Material>
```

Chapter 10

matmodlab User Material Interface

matmodlab can be made to find, build, and execute user materials outside of $MMLROOT. User materials can be written in Python or Fortran and and matmodlab interacts with them through the application programming interface (API). In general, the following pattern is followed for exercising a material model with matmodlab:

1. create a material model interface (MMI)
2. build and link the material model to matmodlab
3. exercise the model

10.1 Material Model Interface

matmodlab interacts with materials through a material interface file. The material interface file defines the material class which must be a subclass of $MMLROOT/materials._material.Material. In this section, methods of the Material class are described.

10.1.1 Material Class Instantiation

The base class Material in $MMLROOT/materials._material creates new matmodlab materials and provides the interface with which matmodlab interacts. Each class must define its name (Material.name) and an ordered list of material parameter names (`Material.param_names`) as they appear in the input file. The class should not define an `__init__` method and if it does, should call the `__init__` of the base class.

Material: Interface

```
    mtl = Material()
```

The following is an example of a `Material` declaration for the `Elastic` material model

```
from materials._material import Material
class Elastic(Material)
    name = 'elastic'
    param_names = ['K', 'G']
```

10.1.2 Setup the Material

The method `setup` sets up the material model by checking and setting the material parameter array, requesting allocation of storage of material variables, and computing and storing the `bulk_modulus` and `shear_modulus` of the material.

Material.setup: Interface

```
    mtl.setup(params)
```

`ndarray params`

    Material parameters parsed from the input file

The following is an example of a `setup` method

```
def setup(self, params):
    if elastic is None:
        raise Error1("elastic model not imported")
    elastic.elastic_check(params, log_error, log_message)
    K, G, = params
    self.set_param_vals(params)
    self.bulk_modulus = K
    self.shear_modulus = G
```

10.1.3 Store Parameter Array with `matmodlab`

The method `set_param_vals` stores the checked parameter values with `matmodlab`. It must be called by each instance of the material model.

Material.set_param_vals: Interface

```
mtl.set_param_vals(params)
```

`ndarray params`
    Checked material parameters

The following is an example of how `set_param_vals` is used within a mterials `setup` method.

```
params = self._check_params(params)
self.set_param_val(params)
```

10.1.4 Adjust the Initial State

The method `adjust_initial_state` adjusts the initial state after the material is setup. Method provided by base class should be adequate for most materials. A material should only overide the base method if absolutely necessary.

Material.adjust_initial_state: Interface

```
mtl.adjust_initial_state(xtra)
```

`ndarray xtra`
    Material variables

10.1.5 Update the Material State

The material state is updated to the end of the step via the `update_state` method. Each material model must provide its own `update_state` method.

Material.update_state: Interface

```
stress, xtra = mtl.update_state(dt, d, sig, xtra,
                                f, ef, t, rho, tmpr, *args)
```

`real dt`
    timestep size

`ndarray d`
: rate of deformation

`ndarray sig`
: stress at beginning of step

`ndarray xtra`
: extra state variables at beginning of step

`ndarray f`
: deformation gradient at end of step

`ndarray ef`
: electric field

`real t`
: time

`real rho`
: density at end of step

`real tmpr`
: temperature at end of step

`tuple args`
: extra args (not used)

`dict kwargs`
: extra keyword args (not used)

`ndarray stress`
: stress at end of step

`ndarray xtra`
: extra state variables at end of step

The following code segment is used by the driver to update the material state

```
args = []
sig, xtra = mtl.update_state(dt, d, sig, xtra,
                             f, ef, t, rho, tmpr, *args)
```

10.1.6 Example

The following example demonstrates the implementation of a simple elastic model.

```
import numpy as np
from materials._material import Material
from core.io import Error1, log_error, log_message
try:
    import lib.elastic as elastic
except ImportError:
    elastic = None

class Elastic(Material):
    name = "elastic"
    param_names = ["K", "G"]
    def __init__(self):
        super(Elastic, self).__init__()

    def setup(self, params):
        if elastic is None:
            raise Error1("elastic model not imported")
        elastic.elastic_check(params, log_error, log_message)
        K, G, = params
        self.set_param_vals(params)
        self.bulk_modulus = K
        self.shear_modulus = G

    def update_state(self, dt, d, stress, xtra, *args):
        elastic.elastic_update_state(dt, self._param_vals,
                                     d, stress,
                                     log_error, log_message)
        return stress, xtra

    def jacobian(self, dt, d, stress, xtra, v):
        return self.constant_jacobian(v)
```

10.2 Building and Linking Materials to `matmodlab`

`buildmtls` builds the `matmodlab` materials by searching the subdirectories of

`$MMLROOT/materials/library` and the directories specified by the `$MMLMTLS` environment variables and building materials it finds. It does so by searching for a single file `makemf.py` in each directory. `makemf.py` is responsible for building the materials and communicating back to `buildmtls`.

10.2.1 Building User Materials

User materials are built by the `makemf` function of the `makemf.py` scripts

makemf.makemf: Interface

```
    blt, fld, skp = makemf(destd, fc, fio, materials=None, *args)
```

`str dest`
: path to directory to copy built shared object libraries (if any)

`str fc`
: path to fortran compiler

`str fio`
: path to the fortran IO routines

`list materials`
: list of materials to build. if empty, build all

`tuple args`
: not used

`tuple blt`
: `(name, interface, mclass, parameters)`. `name` is the name of the material, `interface` the file path to the interface file, `mclass` the material model class name in `interface`, `parameters` an order list of parameter names

`list fld`
: list of names of materials that failed to build

`list skp`
: list of names of materials that were skipped

10.2.2 Building Materials with `f2py`

Material models written in Fortran must compiled in to a Python shared object library with `f2py`. The function `utils.if2py.f2yp` provides an interface to `f2py`.

utils.if2py.f2py: Interface

```
    stat = f2py(name, source_files, signature, fc, incd=None)
```

`str name`
: material model name. will be used to set name of shared object library.

`list source_files`
: list of absolute values of Fortran source files

`str signature`
: path to signature file

`str fc`
: path to Fortran compiler

`str ind`
: (optional) path to include directories

`int stat`
: returns 0 if successful, 1 otherwise

Below is an example of using `if2py.f2py`

```
from utils.if2py import f2py
def makemf(destd, fc, fio, materials=None, *args)
    ...
    stat = f2py(name, source_files, signature, fc, incd)
    if stat != 0:
        return [], [name], []

    shutil.move(name + ".so", os.path.join(destd, name + ".so"))
    return (name, filepath, mclass, parameters), [], []
```

Chapter 11

Regression Testing