`gmd`
a General Material Model Driver

Tim Fuller

September 23, 2013

Chapter 1

Introduction to gmd

gmd is a **G**eneral **M**aterial model **D**river designed for rapid development and testing of material models. gmd 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. gmd is a small tool at the developers disposal to aid in the design and implementation of material models in larger finite element host codes. gmd is the successor the payette material model [**?**] which was itself based in part on Tom Pucick's MMD [**?**] and Rebecca Brannon's MED [**?**] drivers.

The core of the gmd code base is written in Python and leverages Python's object oriented programming (OOP) design. OOP techniques are used throughout gmd 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 gmd simulations are in the ExodusII [**?**] database format, devloped at Sandia National Labs for storing finite element simulation data. Since gmd 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 gmdviz utility, in addition to other visualization packages such as PARAVIEW [**?**].

gmd 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 `gmd` (or, more generally, of any stand-alone constitutive model driver) are

- `gmd` is a very small, special purpose, code. Thus, maintaining and adding new features to `gmd` 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 Obtaining `gmd`

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

Chapter 2

`gmd` Quick Start Guide

This guide provides an outline for building and running `gmd`.

Build `gmd` See Chapter 3.

- Download `gmd` and setup environment
- `$ cd $GMDROOT/toolset && ./setup.py`
- `$ buildmtls`

Prepare Input Inputs are xml specification files. See Chapter **??**.

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

Run

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

Postprocess

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

Chapter 3

Building gmd

gmd'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

gmd has been built and tested extensively on several versions of linux and the Apple Mac OSX operating systems. It is unknown whether or not gmd will run on Windows.

gmd 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 gmd components that was used to build SciPy.

3.2 Installation

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

3.3 Set Environment and Path

`GMDROOT` Optional, name of installation directory

`PATH` `$GMDROOT/toolset:$PATH`

`GMDMTLS` ":" separated list of paths to directories containing user defined material models. See Section **??**.

`GMDTESTS` ":" separated list of paths to directories containing user defined regression tests. See Section **??**.

3.3.1 Set Up

Set up and build the TPLs.

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

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

`buildmtls` Build material models

`gmd` Run `gmd` simulations

`gmddump` Read a `gmd` output and dumps requested variables to ascii columnar files

`gmdviz` 2D plots of `gmd` output

`runtests` Run the regression tests

Each script is a wrapper to another `gmd` 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 `gmd` 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 `gmd` is setup. Thereafter after, only the executable scripts are rewritten. Execute `$ python setup.py -h` for options to rebuild the TPLs.

3.3.2 Build

Build the material libraries

```
$ buildmtls
```

3.3.3 Test the Installation

To test `gmd` after installation, execute

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

which will run the `gmd` regression tests.

3.3.4 Troubleshooting

If you experience problems when building/installing/testing `gmd`, you can ask help from the `gmd` 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

`gmd` Solution Method

`gmd` exercises a material model directly by "driving" it through user specified paths using a specified driver. Currently installed drivers are the `solid` and `eos` drivers. The details of the solution method depend on the driver and are described in the sections to follow.

In solid mechanics inevitably run in to the momentum equation. In equation is stress, need a constitutive model for stress.

4.1 Supported Drivers

4.1.1 Solid

The `solid` driver is designed to exercises material models designed to predict an increment in the material state given the current state and an increment in strain.

$$\boldsymbol{\sigma} = f\left[\boldsymbol{\sigma}, \boldsymbol{\eta}^k, \dot{\boldsymbol{\varepsilon}}\right] \tag{4.1}$$

where $\boldsymbol{\sigma}$ is the stress state, $\boldsymbol{\eta}^k$ are a set of path dependent internal state variables, and $\dot{\boldsymbol{\varepsilon}}$ is the strain rate. The definitions of $\boldsymbol{\sigma}$ and $\dot{\boldsymbol{\varepsilon}}$ are left intentionally vague, except that the pair is work conjugate. Further explanation of $\boldsymbol{\sigma}$ and $\dot{\boldsymbol{\varepsilon}}$ are deferred until a later section. Users drive the material through specified deformation paths. The path can also be a specified stress, in which case we solve for $\dot{\boldsymbol{\varepsilon}}$ to be

$$\dot{\boldsymbol{\varepsilon}} = \dot{\boldsymbol{\varepsilon}}_0 + f^{-1}\left[\boldsymbol{\sigma}, \boldsymbol{\eta}^k, \dot{\boldsymbol{\varepsilon}}\right]\left(\boldsymbol{\sigma} - \boldsymbol{\sigma}_0\right) \tag{4.2}$$

Mixed modes are also allowed. Paths can be prescribed by specifying the components of strain and their rates, components of deformation gradient, displacements of boundary of unit cube, components of stress and their rates. Mixed modes involving strains and stresses allowed.

4.1.2 Electrical

Electric field can be prescribed for testing piezoelectric models.

Chapter 5

Running

```
$ gmd runid[.xml]
```

The following files will be produced

```
$ 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.

Chapter 6

User Input

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. Additionally, the following

6.1 GMDSpec

`<GMDSpec>`

All input files must have as their root element `<GMDSpec>`. Recognized subelements of `<GMDSpec>` are

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

Additionally, the following elements are read from any scope in the input file

- `<Include>`
- `<Function>`

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.

6.2.1 Random Numbers

The `random()` expression generates a random number.

Random State Seed

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.

6.2.2 Example

Specify the `<Material>` parameter `K` and `<Path>` parameter `estar` as variables

```
<GMDSpec>
  <!-- {K = 23e9}
       {estar = -.05}
  -->
  <Physics>
    <Material model="elastic">
      <K> {K} </K>
      <G> 54e9 </G>
    </Material>
    <Path type="prdef" estar="{estar}">
      ...
    </Path>
  </Physics>
</GMDSpec>
```

6.3 Include

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

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

6.3.1 Example

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

6.4 Function

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

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.

6.4.1 Examples

Analytic expression

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

Piecewise linear table

```
<Function id="2" 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="2" 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

Physics

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

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>. Recognized subelements of <Physics> are

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

7.0.2 Path

```
<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">
```

Define deformation paths or equaiton of state surface boundaries, depending on `type`

prdef

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.

surface

Input is similar to the `type=''prdef''` specification, but leg termination time is not specified. Control parameters also differ, as shown in Table 7.2.

A note on `cfmt` and `Cij`

`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

kappa

The attribute `kappa` is only used/defined for the purposes of strain or strain rate control. It refers to the coefficient used in the Seth-Hill generalized strain definition

$$\boldsymbol{\varepsilon} = \frac{1}{\kappa}\left(\boldsymbol{U}^{\kappa} - \boldsymbol{\delta}\right) \tag{7.1}$$

Where κ is the keyword `kappa`, $\boldsymbol{\varepsilon}$ is the strain tensor, $\boldsymbol{U}$ is the right Cauchy stretch tensor, and $\boldsymbol{\delta}$ is the second order identity tensor. Common values of κ and the associated common names for each (there is some ambiguity in the names) are:

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

Examples

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

format: default 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>
```

format: default 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>
```

format: default 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>
```

format: table Read entries from table. Control type is uniform for all legs. Specify control type as `cfmt` attribute of `<Path>`. Optionally, specify the time format as `tfmt` and number of steps for each leg as `nfac`.

```
<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>
```

format: table Read the table from a file, first by the `<Include>` element and then the `href` attribute.

```
<Path type="prdef" format="table" cols="1 3:8" cfmt="222222" tfmt="time">
  <Include href="exmpls.tbl"/>
</Path>
```

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

format: fcnspec Create legs from functions. Functions are specified as `function id[:scale]`. Syntax is otherwise similar to table format. Only a single leg can be specified.

```
<Path type="prdef" kappa="0" tstar="1" amplitude="1" format="fcnspec"
      cfmt="222" nfac="200">
  {2 * pi} 2:1.e-1 1:0 1:0
</Path>
```

type: surface The following examples demonstrate the `type=''surface''`

format: default

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

format: table

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

7.0.3 Material

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

Specify the material model and parameters. Subelements of `<Material>` are

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

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

Matlabel

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

Insert model parameters from a database file. The default file `F_MTL_PARAM_DB` is in `/path/to/gmd/materials/material_properties.db`.

ParameterArray

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

InitialState

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

Key

`<Key> float </Key>`

Examples

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

```
<Material model="elastic">
  <Matlabel href="./materials.xml"> aluminum </Matlabel>
  <K> 124E+09 </K>
</Material>
```

7.0.4 Extract

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

Extract variables and paths from ExodusII output and (optionally) write to different formats. Recognized subelements of `<Extract>` are

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

* `eos` driver only

Variables

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

Path

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

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

Examples

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>
```

Extract 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

Permutation

```
<Permutation method="str[zip]{zip, combine, shotgun}" seed="real[12]"
                     correlation="list[none]{plot, table, none}">
```

Permutate model input parameters, running jobs with different realization of parameters. Good for investigating model sensitivities. Recognized subelements of `<Permutation>` are

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

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, finally, the `shotgun` method zips a uniform distribution for each parameter.

The `correlation` attribute is only meaningful if a `<ResponseFunction>` is specified. Also, note that issues relating to reading the `ExodusII` database prevent gmd from running simultaneous permutation jobs that define a `<ResponseFunction>`.

8.0.5 Permutate

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

Specify the paramaters to permutate. Variable names should occur elsewhere in the input file in preprocessing braces.

8.0.6 Example

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>
```

8.0.7 ResponseFunction

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

Name of response function that returns the response from permutation or optimization jobs. `href` must either be the path to an executable file script containing the response function, or the name of a builtin `gmd` response function.

Built in response functions are

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

Built in response functions operate only on variabes in the simulation output file.

If `href` is a user defined script, the script is called from the command line as

```
% ./scriptname simulation_output.exo [auxiliary_file_1 [... auxiliary_file_n]]
```

Examples

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

```
<ResponseFunction href="gmd.max(PRESSURE)" descriptor="PRES"/>
```

Chapter 9

Optimization

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

Optimize specified parameters against user specified objective function. Recognized subelements of `<Optimization>`

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

9.0.8 Optimize

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

Specify the variable to be optimized, giving initial value and, optionally, bounds. Only the `cobyla` method accepts bounds. Variable names should occur elsewhere in the input file in preprocessing braces.

9.0.9 ResponseFunction

Same as for `<Permutation>`. The value returned from the response function is interpreted as the error to be minimized.

9.0.10 AuxiliaryFile

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

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

9.0.11 Example

Optimize the K and G parameters

```
<Optimization method="simplex" maxiter="25" tolerance="1e-4" disp="0">
  <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

User Materials

10.1 Interface

`gmd` interacts with materials through a material interface file. The material interface file defines the material class which must be a subclass of `$GMDROOT/materials._material.Material`. To follow are data and methods that each material model must define.

class `_material`.**Material**

10.1.1 Class Data

`Material`.**name**
: The material name. Name by which material is invoked from the input file.

`Material`.**param_names**
: Ordered list of material parameter names as they appear in the input file.

The following is an example of a `Material` declaration.

```
class Elastic(Material)
    name = 'elastic'
    param_names = ['K', 'G']
```

10.1.2 Instantiation

Called by Driver. Instantiate the material model. A material's `__init__` method must call the `__init__` method of the parent class.

Material.__init__(params)

```
material = Material()
```

The following is an example of a `__init__` method

```
def __init__(self):
    super(Elastic, self).__init__()
```

10.1.3 Base Class Methods

10.1.4 Methods to be Defined by Each Material Model

Setup the Material

Called by Driver. The method `setup` sets up the material model by checking user input and requesting allocation of storage of material variables.

Material.setup(params)

```
material.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
```

Adjusting the Initial State

Called by Driver. 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(xtra)

```
material.adjust_initial_state(xtra)
```

Update the Material State

Called by Driver. 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(dt, d, sig, xtra, *args, **kwargs)**

```
stress, xtra = material.update_state(dt, d, sig, xtra, *args,
**kwargs)
```

`dt real`
: timestep size

`ndarray d`
: rate of deformation

`ndarray sig`
: stress at beginning of step

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

`tuple args`
: (`defgrad`, `efield`, `time`, `rho`, `tmpr`) extra positional arguments.

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

`ndarray stress`
: stress at end of step

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

10.1.5 Example

A complete example

```
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

`buildmtls` builds the `gmd` materials by searching the subdirectories of `$GMDROOT/materials/library` and the directories specified by the `$GMDMTLS` 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`.

`makemf`.**makemf**

```
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

Chapter 11

Regression Testing