PAYETTE:
An Object Oriented Material Model Driver
Developers and Users Guide

Tim Fuller and Scot Swan

March 17, 2012

1 Introduction

PAYETTE is an object oriented material model driver written in python designed for rapid development and testing of material models.

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

- PAYETTE is a very small, special purpose, code. Thus, maintaining and adding new features to PAYETTE 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 (`www.scipy.org`) modules allow programs written in Python to leverage a large set of numerical routines provided by the 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

Payette is an outgrowth of Tom Pucick's `MMD` and Rebecca Brannon's `MED` drivers. Both these other drivers are written in Fortran.

1.4 Simulation Approach

Payette exercises a material model directly by "driving" it through user specified mechanical and electrical inputs.

1.4.1 Supported Drivers

Mechanical

- Direct
  - Strain rate
  - Strain
  - Deformation gradient

  - Velocity
  - Displacement

- Inverse
  - Stress
  - Stress rate

Electrical

- Direct
  - Electric field

2 PAYETTE System Requirements

PAYETTE requires Python 2.6 or later with the scipy, numpy, and matplotlib modules installed.

2.1 Installing Required Python Modules

In the sections below, instructions are given for installing the required python modules via system level package management programs. Of course, all modules can be installed from source, but having a package manager handle all of the build dependencies is infinitely easier in most cases.

2.1.1 Installing on Macintosh

The Apple operating system does not provide a native package manager, but several third party package managers are available. The two most common are Fink (`www.finkproject.org`) and MacPorts (`www.macports.org`). The following instructions apply to MacPorts.

If you do not have MacPorts installed on your system, first install the optional Developer Tools from the Mac OSX installation DVD that came with your computer. If you do not have a copy, the developer tools can be downloaded directly from Apple (`https://developer.apple.com/technologies/tools`), but it is a very large download (~3 GB). After installing the Developer Tools, download the MacPorts binary installer (`http://www.macports.org/install.php`) and install.

Once MacPorts is installed, simply execute the following command from the terminal (`/Applications/Utilities/Terminal.app`):

```
% sudo port install pyXx-scipy pyXx-matplotlib
```

where `X` and `x` are the major and minor python release numbers, respectively.

2.1.2 Installing on Linux

Most Linux distributions come with a native package manager installed that will fetch and install the required python modules. On Ubuntu/Debian, simply execute in a terminal window:

```
% sudo apt-get python-numpy python-scipy python-matplotlib
```

♠*This will not work for everyone. Alternatives need to be described.*♠ ←—

3 Getting PAYETTE

The source code for PAYETTE is distributed via svn from `svn://lenny.eng.utah.edu/csm/local/svn/Payette/`. The repository is access limited. Contact Tim Fuller (`tjfulle@sandia.gov`) or Scot Swan (`mswan@sandia.gov`) to gain access to the repository. Once access is granted, execute

```
% svn co svn://lenny.eng.utah.edu/csm/local/svn/Payette/
```

at a command prompt. Once executed, the following directories will be checked out on to your local machine

```
Payette/
  branches/
  tags/
  trunk/
    Aux/
    Documents/
    Source/
    Tests/
    Toolset/
```

4 Building PAYETTE

4.1 Setting Up Your Environment

PAYETTE requires the following additions to your standard environment

- `PAYETTE_HOME` environment variable that points to `/path/to/Payette/trunk/`.
  bash: `export PAYETTE_HOME=/path/to/Payette/trunk`
  csh: `setenv PAYETTE_HOME /path/to/Payette/trunk`

- Append `$PAYETTE_HOME/Toolset` to your `$PATH`
  bash: `export PATH = $PATH:$PAYETTE_HOME/Toolset`
  csh: `setenv PATH $PATH:$PAYETTE_HOME/Toolset`

- If you are authorized to work on the Kayenta material model, set the environment variable `PAYETTE_KAYENTA` to point to the Kayenta `trunk/src` directory.

4.2 Building the PAYETTE Executable Scripts

If all of the previous steps were successfully completed, building PAYETTE is completed through the `buildPayette` script in the `$PAYETTE_HOME/Toolset` directory. To build, execute

```
% cd $PAYETTE_HOME/Toolset
% python buildPayette
```

at a command prompt, making sure that the python interpreter used to build Payette is the same interpreter that you previously installed numPy and sciPy support. The `buildPayette` script will build two additional scripts in `$PAYETTE_HOME/Toolset`: `runPayette` and `testPayette`, both of which are described later in this document.

`buildPayette` has additional options/capabilities that can be seen by executing:

```
% python buildPayette -h
```

5 Running PAYETTE

Interacting with PAYETTE is done through the `runPayette` script and properly formatted input files. The basic usage of `runPayette` is

```
% runPayette input_file
```

For a complete list of options for `runPayette` execute

```
% runPayette -h
```

5.1 Simulation Output

For a simulation titled "`simnam`", the following output is created by `runPayette`:

`simnam.log` ascii log file

`simnam.out` ascii space delimited output file

`simnam.math1` ascii Mathematica auxiliary postprocessing file

`simnam.math2` ascii Mathematica auxiliary postprocessing file

`simnam.prf` binary restart file

`simnam.props` ascii list of checked material parameters

6 PAYETTE Input File Formatting

Input files are comprised of several "blocks" of instruction for `runPayette`. A block is a group of instructions contained in a `begin <block> [block name] ... end <block>` pair:

```
begin <block> [block name]
        .
        .
        .
end <block>
```

A Note on Case, Spacing, and Comment Characters For the most part, the case of the text in the input file does not matter, nor does the indentation on each line. Indentation of input blocks is used only for clarity in this document. `runPayette` supports # and $ as native comment characters and the user can pass the optional `--cchar=userchar` to specify any character "`userchar`" to be used as a comment character.

6.1 Required Blocks

The blocks required by `runPayette` are

```
begin simulation <title>
   begin material
      [material options]
   end material
   begin boundary
      [boundary options]
   end boundary
end simulation
```

The ordering of the blocks within the `simulation` block does not matter. However, all blocks for a given simulation must be nested in the `simulation` block. Details of the required content of each block follows.

6.1.1 `simulation` Block

Each input file must have a `simulation` block with title for that simulation. The title of the simulation will serve as the basename for all simulation output, with spaces replaced with underscores.

Number of Simulations per Input File `runPayette` supports an arbitrary number of `simulation` blocks in a single input file.

6.1.2 `material` Block

In the `material` block, the constitutive model and material parameters are defined. A `material` block would look like

```
begin material
   constitutive model <model>
   parameter 1 <value>
   parameter 2 <value>
   parameter n <value>
```

```
end material
```

An important philosophical point, parameters are associated with the material, and not the constitutive model. This allows different materials to be exercised by different constitutive models without changing parameters.

An example `material` input block for an elastic material would look like

```
begin material
   constitutive model elastic
   bkmod 130.e9
   shmod 57.e9
end material
```

6.1.3 `boundary` Block

In the `boundary` block, the boundary conditions for the simulation are defined. The `boundary` block has the following format, with the default values being shown.

```
begin boundary
   kappa = 0.
   ampl = 1.
   ratfac = 1.
   tstar = 1.
   sstar = 1.
   estar = 1.
   fstar = 1.
   dstar = 1.
   efstar = 1.
   stepstar = 1.
   emit = all {all,sparce}
   screenout = false {true,false}
   nprints = 0 {0-nsteps}
   begin legs
      # leg no, time, nsteps, ltyp, c[ij]
   end legs
end boundary
```

Keyword "[t,s,e,f,d,ef,step]star" Multiplier on all components of time, stress, strain, deformation gradient, strain rate, displacement, electric field,

and number of steps, respectively. All values of the previously listed quantities are defined in each leg will be multiplied by this factor. As an example, if the simulation times are given in microseconds, "`tstar`" could be set to "1.0e-6" and the times given in integer microsecond values.

Keyword "`emit`" Write all data (`emit` = `all`) or data from only 10 timesteps (`emit` = `sparse`) to the output file.

Keyword "`screenout`" Print out all timestep information to the console.

Keyword "`nprints`" Total number of writes to the output file during the simulation.

Keyword "`ampl`" Multiplier on all leg inputs. `ampl` may be used to increase or decrease the peak values of the given inputs without changing the rates of those inputs.

Keyword "`ratfac`" Multiplier on strain and stress rates - effectively achieved by dividing each time by `ratfac`.

Keyword "`kappa`" The keyword "`kappa`" is only used/defined for the purposes of strain or strain rate control. It refers to the coefficient used in the Seth-Hill general strain definition. For your convenience, the formula used is

$$[\varepsilon] = \frac{1}{\kappa}\left([U]^{\kappa} - [I]\right) \tag{1}$$

Where κ is the keyword "`kappa`", $[\varepsilon]$ is the strain tensor, $[U]$ is the right Cauchy stretch tensor, and $[I]$ is the identity tensor. Below in Table 1 is a table of common values of `kappa` and the associated common names for each (there is some ambiguity in the names).

6.1.4 `legs` Block

The `legs` block defines the material states that will be applied to the single element throughout the simulation. In the most general case, each leg will be defined as follows:

```
begin legs
  leg no, time, nsteps, ltyp, c[ij]
end legs
```

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

Table 1: Sampling of names encountered in the literature for common strain measures in the Seth-Hill family of strains for integer values of κ.

The "`legno`" or leg number is always a strictly monotonically increasing integer, starting from zero for the first leg. There can be an arbitrary number of legs defined.

The value "`time`" defines at what time in the simulation the material state will be as it is defined in that leg. Generally the first leg (leg zero) will have a time equal to 0 (seconds, microseconds, etc.). The values of time must increase strictly monotonically with leg number.

The value of "`nsteps`" is an integer that defines the number of steps the simulation will take to get from the previous leg to the current leg. Currently, it is not possible to explicitly define a timestep for the single element tests. However, by setting the time increment and value of "`nsteps`" you can set the timestep size for that leg.

The value of "`ltyp`" or leg type is a little more involved. This keyword frequently has more features added to it, most of which are experimental. However, in this document only the most basic and stable options will be addressed.

The basic form of the leg type is a string that specifies the material state for specific components by setting each character to one of the following:

1 strain rate control (mech,6)
2 strain control (mech,6)
3 stress rate control (mech,6)
4 stress control (mech,6)
5 deformation gradient control (mech,9)
6 electric field (elec,3)
8 displacement (mech,3)

There are two types of control here: mechanics control (mech) and electric field control (elec). Because these two types are seperated and handled

individually after parsing, the mechanics and electric control characters can be mixed together without change in behavior. The integer given in the parentheses in the above list represent the maximum number of components that may be defined. The user must define at least three mechanics options for any given simulation. If no electric field options are given, they default to zero.

Once the mechanics control characters are gathered, they are checked for compatibility. Specifically, the rules are as follows:

- Deformation gradient control (5) cannot be mixed with other mechanics control options and all 9 components must be defined.
- Displacement control (8) cannot be mixed with other mechanics control options.

Then, the component values "c[ij]" are read in and are assigned values based on the leg type string. The general case for "c[ij]" user input components (which is actually a 1D array) correlating to tensor components is

$$[C] = \begin{bmatrix} C_1 & C_4 & C_6 \\ C_7 & C_2 & C_5 \\ C_9 & C_8 & C_3 \end{bmatrix} \tag{2}$$

Payette simply follows this pattern for assigning variables. However, at least the first three must be defined (the x, y, and z components). If any variables are given beyond this, it fills in the matrix in that order up to the maximum number of components.

Example 1: Deformation Gradient, Uniaxial Extension This example extends the material in the x-direction.

```
begin legs
  0, 0.0, 0, 555555555, 1.0, 1., 1., 0., 0., 0., 0., 0., 0.
  1, 1.0, 0, 555555555, 1.1, 1., 1., 0., 0., 0., 0., 0., 0.
end legs
```

Example 2: Strain Control, Uniaxial Extension All of the following produce equivalent behavior (uniaxial strain extension). Remember to set "`kappa`" to the desired value.

```
begin legs
  0, 0.0, 0, 222, 0.0, 0., 0.
```

```
  1, 1.0, 0, 222, 0.1, 0., 0.
end legs
```

or

```
begin legs
  0, 0.0, 0, 2222, 0.0, 0., 0., 0.
  1, 1.0, 0, 2222, 0.1, 0., 0., 0.
end legs
```

or

```
begin legs
  0, 0.0, 0, 222222, 0.0, 0., 0., 0., 0., 0.
  1, 1.0, 0, 222222, 0.1, 0., 0., 0., 0., 0.
end legs
```

Example 3: Stress Control, Uniaxial Tension Stress control can be defined in much the same way as strain control (see example 2). Assuming that the material parameters are defined in MKS, stress will be defined in terms of Pa.

```
begin legs
  0, 0.0, 0, 444444, 0.0e9, 0., 0., 0., 0., 0.
  1, 1.0, 0, 444444, 1.0e9, 0., 0., 0., 0., 0.
end legs
```

Example 4: Mixed Control This example could be used to reproduce the stress state found in the standard tension test when the strain is known. Specifically, we prescribe the strain value in the x-direction while prescribing the lateral stresses to be held at zero stress.

```
begin legs
  0, 0.0, 0, 244, 0.00, 0., 0.
  1, 1.0, 0, 244, 0.01, 0., 0.
end legs
```

6.1.5 Notes on Stress Control

As prescribed stress is essentially an inverse problem, prescribed stress simulations take much longer than prescribed strain simulations. This is due to the need of using optimization routines to determine the correct strain increment to input to the material model.

The general objective function used in the optimization routine is simply an L2 norm of the difference of the prescribed stress components with the output stress components. This is of particular importance when a stress is prescribed outside of the limit surface for a material and the material stress simply cannot reach the prescribed stress. For example, if the user desired uniaxial stress but prescribed the stress to some value beyond the limit surface then the stress state might develop non-zero values in components that could reasonably be expected to stay zero.

To alleviate this problem, Payette has a command line argument "–proportional" that enforces proportional loading for prescribed stress problems. This changes the objective function to optimize for a stress state that is proportional to the prescribed stress by some factor and that is as close as possible to the prescribed stress. Specifically, it decomposes the output stress tensor into "proportional" and "perpendicular" tensors with the proportional stress tensor begin a multiple of the prescribed stress tensor and the perpendicular tensor being the balance. The function takes the square of the L2 norm of the perpendicular tensor and the L2 norm of the difference of the proportional and prescribed tensors. In this way Payette weights the stress state such that being proportional is more important than being closer. For this objective function using the same simulation as described for the default objective function, we would maintain uniaxial stress even when the limit surface is attained.

As a warning, stress control (especially proportional loading) can introduce excessive noise when stress states are prescribed outside of the yield surface. However, it can also perform flawlessly. Be aware of the limits of the objective functions when choosing.

6.2 Optional Blocks

The following blocks are optional: `mathplot`.

6.2.1 `mathplot` Block

The `mathplot` block is used to specify optional plotting information written to the `simnam.math[1,2]` output files for use in Rebecca Brannon's Mathematica post processing files. The basic syntax is

```
begin mathplot
  var1 var2 var3
  var4, var5, var6
  var7; ...; varn
```

```
end mathplot
```

where var? are PAYETTE and material model variables. A complete list of plotable variables is listed in each simulation's log file. Each line in the `mathplot` block can contain an arbitrary number of space, comma, or semicolon delimited variables.

6.3 Inserting External Files

External files containing formatted PAYETTE input can be included anywhere in the input file through the `include` and `insert` directives. For example, material parameters can be kept in a separate parameter file and inserted in to an input file by

```
begin material
  constitutive model kayenta
  insert salem_limestone.dat
end material
```

When PAYETTE encounters an [`insert,include`] directive, it looks for the inserted file by it's absolute path, in the current directory, and in the `$PAYETTE_HOME/Aux/MaterialsDatabase`, in that order.

6.4 Complete Input File

Below is an input file, any keywords not given take the default values shown in Section .

```
begin simulation elastic unistrain cycle
  begin material
    constitutive model elastic
    shmod 53.e9
    bkmod 135.e9
  end material
  begin boundary
    kappa = 0.
    tfac = 1.
    amplitude = 1
    begin legs
#     l  t   n     ltyp       c[ij]...
      0, 0., 0.,   222222,    0., 0., 0., 0., 0., 0.
      1, 1., 100,  222222,    .1, 0., 0., 0., 0., 0.
      2, 2., 100,  222222,    0., 0., 0., 0., 0., 0.
      3, 3., 100,  444444,    20.566e9, 9.966e9, 9.966e9, 0., 0., 0.
      4, 4., 100,  444444,    0. ,0. ,0. ,0. ,0. ,0.
      5, 5., 100,  111111,    0.1 ,0. ,0. ,0. ,0. ,0.
      6, 6., 100,  111111,   -0.1 ,0. ,0. ,0. ,0. ,0.
      7, 7., 100,  333333,    20.566e9, 9.966e9, 9.966e9, 0., 0., 0.
      8, 8., 100,  333333,   -20.566e9, -9.966e9, -9.966e9, 0., 0., 0.
      9, 9., 100,  555555555, 1.1052, 1., 1., 0., 0., 0., 0., 0., 0.
     10, 10.,100,  555555555, 1., 1., 1., 0., 0., 0., 0., 0., 0.
    end legs
  end boundary
  begin mathplot
    sig11 sig22 sig33
    eps11 eps22 eps33
  end mathplot
end simulation
```

7 testPayette

testPayette is what makes PAYETTE an invaluable tool to material model development. Simplistically, testPayette is a tool that exercises runPayette on previously setup simulations and compares the results against a "gold" result, allowing model developers to instantly see if changes to a material model affect its results.

7.1 Basic Usage of testPayette

From a command prompt execute

```
% testPayette
```

testPayette will then scan $PAYETTE_HOME/Tests for tests. After creating a lists of tests to run, runPayette will create $PWD/TestResults.$OSTYPE directory and subdirectories in which the tests will be run. The subdirectories created in $PWD/TestResults.$OSTYPE/ mirror the subdirectories of $PAYETTE_HOME/Tests. After creating the $PWD/TestResults.$OSTYPE/ directory tree, it will then move in to the appropriate subdirectory to run and analyze each individual test. For example, if the Tests/Materials/Elastic/elastic-unistrain.py test is run, testPayette will create a $PWD/TestResults.$OSTYPE/Materials/Elastic/elastic-unistrain/ directory, move to it, and run the elastic-unistrain.py test in it.

7.2 Filtering Tests to Run

The tests run can be filtered by keyword with the -k option. For example, to run only the "fast" "kayenta" material tests, execute

```
% testPayette -k kayenta -k fast
```

Or, you can run specific tests with the -t option. For example, to run only the elastic-unistrain.py test, execute

```
% testPayette -t elastic-unistrain
```

7.3 Multiprocessor Support

By default, `testPayette` will run all collected tests serially on a single processor, but supports running tests on multiple processors with the `-j nproc` option, where `nproc` is the number of processors. For example, to run all of the kayenta tests on 8 processors, execute

```
% testPayette -k kayenta -j8
```

Be warned, however, that when run on multiple processors, you cannot kill `testPayette` by `ctrl-c`, but must put the job in the background and kill it with `kill -9 pid`, where `pid` is the process id. Any help from a python multiprocessor expert on getting around this limitation would be greatly appreciated!

7.4 Other `testPayette` Options

Execute

```
% testPayette -h
```

for a complete list of options.

7.5 `$PWD/TestResults.$OSTYPE/summary.html`

`testPayette` gives a complete summary of test results in `$PWD/TestResults.$OSTYPE/summary.html` viewable in any web browser.

7.6 Mathematca Post Processing

For the Kayenta material model, `testPayette` creates Prof. Brannon's Mathematica post processing file `kayenta.nb` (formerly `030626.nb`) in `$PWD/TestResults.$OSTYPE/Materials/Kayenta/`, that aids greatly in analyzing test results for the Kayenta material model.

7.7 Creating New Tests

NOT DONE YET!

8 Installing New Materials in PAYETTE

PAYETTE was born from the need for an environment in which material constitutive models could be rapidly developed, tested, deployed, and maintained, independent of host finite element code implementation. Important prerequisites in the design of PAYETTE were ease of model installation and support for constitutive routines written in Fortran. This requirement is met by providing a simple API with which a model developer can install a material in PAYETTE as a new Python class and use `f2py` to compile Python extension modules from the material's Fortran source (if applicable). In this section, the PAYETTE API is described through an example elastic material model and instructions for extending a material model to include Fortran subroutines are given.

8.1 Material File Naming Conventions

Each material model must provide its Python class definition in its own file in the `$PAYETTE_HOME/Source/Materials`. The accepted naming convention for the material's Python class file is

```
$PAYETTE_HOME/Source/Materials/Payette_material_name.py
```

A material's Fortran source code (if applicable) is stored in its own directory in the `$PAYETTE_HOME/Source/Materials/Fortran` directory. The accepted naming convention for the material's Fortran directory is

```
$PAYETTE_HOME/Source/Materials/Fortran/MaterialName
```

There is no defined naming convention for Fortran source code within the materials `Fortran/MaterialName` directory.

As an example, suppose you were developing a new material model "Porous Rock". The source files and directories would be named:

```
$PAYETTE_HOME/Source/Materials/Payette_porous_rock.py
$PAYETTE_HOME/Source/Materials/Fortran/PorousRock
```

8.1.1 A Note On Including Fortran Source Code

The PAYETTE project is an open source project, without restrictions on how the source code is used and/or distributed. However, many material

models do have restrictive access controls and cannot, therefore, include their source files in PAYETTE. This is the case for many materials currently being developed with PAYETTE. For these materials, rather than include the source code in the material's Fortran directory, the Fortran build script used by PAYETTE to build the material's Python extension module is used to direct PAYETTE to the location of the material's source files, elsewhere in the developer's file system. The contents of the Fortran build script are discussed later in this document in the section on building Fortran source code in PAYETTE.

8.2 `ConstitutiveModel` – Constitutive Model Base Class

PAYETTE provides a simple interface for interacting with material models through the Python class structure. Material models are installed as separate Python classes, derived from the `ConstitutiveModel` base class. The `ConstitutiveModel` base class offers the following methods in its API:

8.2.1 Data

`ConstitutiveModel`.**name** [type:`str`, default:`None`]
Constitutive model name, used in parsing user input.

`ConstitutiveModel`.**aliases** [type:`list`, default:[]]
Constitutive model aliases, used in parsing user input.

`ConstitutiveModel`.**parameter_table** [type:`dict`, default:{}]
Table of parameter names, position in material user input array, and aliases. Used in parsing user input. Derived classes must return a parameter_table of the form
`param_table` = {
"`param_1_name`":{"ui pos":0,"aliases":["alias 1","alias 2",...]},
"`param_2_name`":{"ui pos":1,"aliases":["alias 1","alias 2",...],
...
"`param_n_name`":{"ui pos":n-1,"aliases":["alias 1","alias 2",...]}
}

`ConstitutiveModel`.**nprop**
type:int default:0

`ConstitutiveModel`.**ndc**
type:int default:0

`ConstitutiveModel.`**nsv**
type:int default:0

`ConstitutiveModel.`**ui0**
type:numpy.array
default:numpy.zeros(self.nprop)

`ConstitutiveModel.`**ui**
type:numpy.array
default:numpy.zeros(self.nprop)

`ConstitutiveModel.`**dc**
type:numpy.array
default:numpy.zeros(self.ndc)

`ConstitutiveModel.`**namea**
type:list
default: [None]*self.nsv

`ConstitutiveModel.`**keya**
type:list
default: [None]*self.nsv

`ConstitutiveModel.`**bulk_modulus**
type:real
default: 0.

`ConstitutiveModel.`**shear_modulus**
type:real
default: 0.

`ConstitutiveModel.`**multi_level_fail_model**
type:bool
default:False

`ConstitutiveModel.`**electric_field_model**
type:bool
default:False

`ConstitutiveModel.`**cflg_idx** default: None

`ConstitutiveModel.`**fratio_idx**
None

8.2.2 Methods

```
def __init__(self, props):
    pass

def Initialize_State(self,*args,**kwargs):
    pass

def Set_Up(self,*args,**kwargs):
    msg = 'Consitutive model must provide Set_Up method'
    reportError(__file__,msg)
    return 1

def Update_State(self,*args,**kwargs):
    msg = 'Consitutive model must provide updateState method'
    reportError(__file__,msg)
    return 1

def initialParameters(self):
    return self.ui0

def checkedParameters(self):
    return self.ui

def modelParameters(self):
    return self.ui

def initialJacobian(self):
    return self.J0

def internalStateVariables(self):
    return self.sv

def derivedConstants(self):
    return self.dc

def isvKeys(self):
    return self.keya
```

```
def computeInitialJacobian(self):
```

- `def __init__(self,props)`
- `def jacobian(self,dt,d,Fold,Fnew,EF,sig,sv,v,*args,**kwargs)`
- `def checkProperties(self,*args,**kwargs)`
- `def setField(self,*args,**kwargs)`
- `def updateState(self,*args,**kwargs)`

Once a material object is instantiated by PAYETTE, PAYETTE interacts with that model by calling any of the above listed methods.

```
ConstitutiveModel.checkProperties(self,*args,**kwargs)

ConstitutiveModel.setField(self,*args,**kwargs)

ConstitutiveModel.updateState(self,*args,**kwargs)
```

8.3 Derived Material Class

8.3.1 `attributes` Dictionary

8.4 Fortran Code

8.4.1 Build Script

```
import sys
import os
import numpy as np

from Source.Payette_utils import *
from Source.Payette_constitutive_model import ConstitutiveModelPrototype

attributes = {"payette material":True,
              "name":"py elastic",
              "aliases":["python elastic","py hooke","python hook"],
              "material type":["mechanical"]
              }
class PyElastic(ConstitutiveModelPrototype):
    """
```

```
CLASS NAME
   PyElastic

PURPOSE

   Constitutive model for an elastic material in python. When instantiated,
   the Elastic material initializes itself by first checking the user input
   (checkProperties) and then initializing any internal state variables
   (setField). Then, at each timestep, the driver update the Material state
   by calling updateState.

METHODS
   checkProperties
   setField
   updateState

AUTHORS
   Tim Fuller, Sandia National Laboratories, tjfulle@sandia.gov
"""

imported = True
name = attributes["name"]
aliases = attributes["aliases"]
parameter_table = {
"K":{"ui pos":0,"aliases":["BKMOD"]},
"G":{"ui pos":1,"aliases":["SHMOD"]}}
nprop = len(parameter_table.keys())

def __init__(self,props):
    self.dc = np.zeros(0)
    self.ui0 = np.array(props)
    self.ui = self.checkProperties(props=props)
    (self.nsv,self.namea,self.keya,self.sv,
     self.rdim,self.iadvct,self.itype) = self.setField()
    self.J0 = self.computeInitialJacobian()

def jacobian(self,dt,d,Fold,Fnew,EF,sig,sv,v,*args,**kwargs):
    return self.J0[[[x] for x in v],v]

def checkProperties(self,*args,**kwargs):
```

```
        K,G = kwargs["props"]

        msg = ""
        if K <= 0.0: msg += "Bulk modulus K must be positive.  "
        if G <= 0.0: msg += "Shear modulus G must be positive"
        if msg: reportError(__file__,msg)

        if 3.*K < 2.*G: msg += "neg Poisson (to avoid warning, set 3*B0>2*G0)"
        if msg: reportWarning(__file__,msg)
        self.bulk_modulus,self.shear_modulus = K,G

        # internally, work with lame parameters
        lam = K - 0.66666666666666666667*G
        return np.array([G,lam])

    def setField(self,*args,**kwargs):
        nsv = 2
        sv = np.array([0.,1.])
        namea,keya = ["Free 01","Free 02"], ["F01","F02"]
        rdim,iadvct,itype = [None]*3
        return nsv,namea,keya,sv,rdim,iadvct,itype

    def updateState(self,*args,**kwargs):
        """
           update the material state based on current state and strain increment
        """
        dt,d,fold,fnew,efield,sigold,svold = args
        de, delta = d*dt, np.array([1.,1.,1.,0.,0.,0.])
        dev = (de[0] + de[1] + de[2])
        twog, lam = 2.*self.ui[0], self.ui[1]
        signew = sigold + twog*de + lam*dev*delta
        return signew, self.sv
```