Payette Documentation

Release 1.0

Tim Fuller, Scot Swan

April 02, 2012

CONTENTS

CHAPTER

ONE

INTRODUCTION

Payette is an object oriented material model driver written in Python designed for rapid development and testing of material models. The core of the *Payette* code base is written in Python, with the exception of many material models and optimization routines that are written in Fortran and wrapped by *f2py*.

Payette 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 *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 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

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

CHAPTER

TWO

DEVELOPER GUIDE

2.1 *Payette* Directory Structure

The `Payette` project has the following directory structure

```
PAYETTE_ROOT
  configure.py
  __init__.py
  README

  Aux/
    Inputs/

  MaterialsDatabase/

  Documents/
    Documentation/
    Presentations/

  Examples/

  Source/
    Fortran/
    Materials/
      Fortran/
      Library/

  Tests/
    Materials/
    Regression/

  Toolset/
```

2.2 Coding Style Guide

All new code in *Payette* should adhere sctrictly to the style described in Python's PEP-0008 and docstrings should follow NumPy's docstring guide. One exception to the Python style guide is the naming convention of files in *Payette*: all file names begin with `Payette_`.

It is highly recommended to use tools such as *pylint* and *pep8* to perform code analysis on all new and existing code.

Note: Many of the original files in *Payette* were not written to the Python style guide. As time allows, these files are being modified to adhere to the standards.

CHAPTER

THREE

OBTAINING *PAYETTE*

Payette is an open source project licensed under the MIT license. A copy of the source code may be obtained by contacting Tim Fuller.

For students in the School of Engineering at the University of Utah, a copy of *Payette* may by obtained by:

```
git clone cade_lab_user_name@lenny.eng.utah.edu:/csm/local/git/Payette.git Payette
```

CHAPTER

FOUR

BUILDING *PAYETTE*

Payette is an object oriented material driver. The majority of the source code is written in Python and requires no additional building. Many of the material models, however, are written in Fortran and require a seperate compile step.

4.1 System Requirements

Payette has been built and tested extensively on several versions of linux and the Apple Mac OSX 10.6 operating systems. It is unknown whether or not *Payette* will run on Windows.

4.2 Required Software

Payette requires the following software installed for your platform:

1. Python 2.6 or newer
2. NumPy 1.5 or newer
3. SciPy 0.1 or newer

The required software may be obtained in one of several ways:

- *Install from your systems package manager*

  `apt-get` on Ubuntu, `yum` on Fedora, `rpm` on Red Hat, etc. On the Mac, we have had great success using software installed by MacPorts.

  > **Warning:** We have had varying degrees of success using software installed by package managers on various linux distributions and have had to resort to building from source or using Sage, as described below.

- *Build each from source*

  An involved, but fairly straight forward process.

- *Use Python, NumPy, and SciPy installed with* Sage 4.7 *or newer*

  This option is perhaps the easiest, since Sage provides prebuilt binaries for the Mac and many linux platforms. Their build process has also proven to be very robust on many platforms.

  Note:

  – Some features are disabled when using Sage. In particular, callback functions from Fortran subroutines to Python functions.

- Using Sage is not as extensively tested as using a seperate Python installation.

4.3 Installation

1. Make sure that all *Payette* prerequisites are installed and working properly.
2. Add `PAYETTE_ROOT` to your `PYTHONPATH` environment variable
3. Add `PAYETTE_ROOT/Toolset` to your `PATH` environment variable
4. Change to `PAYETTE_ROOT` and run:

   ```
   % PYTHON configure.py
   ```

   where `PYTHON` is the python interpreter that has *NumPy* and *SciPy* installed. If using Sage, replace `PYTHON` with `sage -python`.

   `configure.py` will write the *Payette* configuration file and the following executable scripts:

   ```
   PAYETTE_ROOT
     Payette_config.py
     Toolset/
       buildPayette
       cleanPayette
       runPayette
       testPayette
   ```

5. execute:

   ```
   % buildPayette
   ```

   which will build the *Payette* material libraries and create a configuration file of all built and installed materials in `PAYETTE_ROOT/Source/Materials/Payette_installed_materials.py`

4.4 Testing the Installation

To test *Payette* after installation, execute:

```
% testPayette -k regression -k fast
```

which will run the "fast" "regression" tests. To run the full test suite execute:

```
% testPayette -j4
```

Please note that running all of the tests takes several minutes.

4.5 Known Issues

1. *callbacks*

   A callback function is a Python function accessible by Fortran subroutines through a callback mechanism provided by *f2py*. The *NumPy* and *SciPy* packages distributed by many of the different linux distributions package managers have broken dependencies. In particular, callback functions seem to be broken on many linux systems. The easy work around is to configure *Payette* with `--no-callback`. Another work around is passing different fortran compilers to configure.py (`--f77exec=` , `--f90exec=`) and seeing if that makes a difference.

2. *segfault*

   A segfault error is usually a result of a broken callback. See 0) above.

3. *Unable to build*

   Difficulty building *Payette* is usually the result of broken *NumPy* and *SciPy* installations and the workaround involves reinstalling all software packages from sourc. If you are uncomfortable installing these software packages from source, consider using Sage to build and run *Payette*.

4.6 Troubleshooting

If you experience problems when building/installing/testing *Payette*, you can ask help from Tim Fuller or Scot Swan. Please include the following information in your message:

1. Are you using Sage, or not
2. Platform information OS, its distribution name and version information etc.:

```
% PYTHON -c 'import os,sys;print os.name,sys.platform'
% uname -a
```

3. 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
```

4. Python version:

```
% PYTHON -c 'import sys;print sys.version'
```

5. *NumPy* version:

```
% PYTHON -c 'import numpy;print numpy.__version__'
```

6. *SciPy* version:

```
% PYTHON -c 'import scipy;print scipy.__version__'
```

7. The contents of the `PAYETTE_ROOT/Payette_config.py` file
8. Feel free to add any other relevant information.

CHAPTER

FIVE

RUNNING *PAYETTE*

5.1 Getting Started

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
```

5.2 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)
```

CHAPTER

SIX

PAYETTE INPUT FILE FORMATTING

6.1 Input File Blocks

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>
```

Note: 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.

6.2 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.

6.2.1 The `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.

Note: `runPayette` supports an arbitrary number of `simulation` blocks in a single input file.

6.2.2 The `material` Block

In the `material` block, the constitutive model and material parameters are defined. A `material` block takes the following form:

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

Note: 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
```

6.2.3 The `boundary` Block

In the `boundary` block, the boundary conditions for the simulation are defined. The `boundary` block is comprised of keyword instructions to *Payette* and a `legs` block. In the `boundary` block below, the default values for available keywords are 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
```

The various keywords and the `legs` block are described in the following sections.

6.2.4 The `boundary` Block Keywords

`[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.

`emit`

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

`screenout`

Print out all timestep information to the console.

`nprints`

Total number of writes to the output file during the simulation.

`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.

`ratfac`

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

`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 generalized strain definition

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

Where κ is the keyword `kappa`, $[\varepsilon]$ is the strain tensor, $[U]$ is the right Cauchy stretch tensor, and $[I]$ is the 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

6.2.5 The `legs` Block

The `legs` block defines the material states that will be applied to the single element during each "`leg`" of the simulation. Legs may be defined in one of two ways: 1) a general method in which all of the control parameters of each leg are explicitly defined or, 2) time, deformation type table. Each method of specifying `legs` is described below.

General Leg Specification

In the most general case, each leg will be defined as follows:

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

The `leg no` or leg number is 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. For symmetric second order tensors, the `c[ij]` user input components correlating to tensor components by

$$[C] = \begin{bmatrix} C_1 & C_4 & C_6 \\ & C_2 & C_5 \\ & & C_3 \end{bmatrix}$$

For general second order tensors, the `c[ij]` user input components correlating to tensor components by

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

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.

Time/Deformation Type Tables

In the event that the deformation control type is constant for all legs (e.g., all legs are strain controlled), a more convenient method of defining each leg is through specifying a time/deformation type table. In this specialized case, the legs block is defined as:

```
begin legs
  using <time,dt>, <deformation type>
  <time,dt> components of deformation...
                .
                .
                .
  <time,dt> components of deformation...
end legs
```

This method of input is convenient for reading in history files from finite element simulations, or data collection software.

6.2.6 `legs` Examples

Example 1: Deformation Gradient, Uniaxial Extension

This example extends the material in the x-direction:

```
begin legs
  0, 0.0, 0, 555555555, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0
  1, 1.0, 1, 555555555, 1.1, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0
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, 222222, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
  1, 1.0, 1, 222222, 0.1, 0.0, 0.0, 0.0, 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, 0.0, 0.0, 0.0
  1, 1.0, 0, 444444, 1.0e9, 0.0, 0.0, 0.0, 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, 244222, 0.00, 0.0, 0.0, 0.0, 0.0, 0.0
  1, 1.0, 0, 244222, 0.01, 0.0, 0.0, 0.0, 0.0, 0.0
end legs
```

Note: 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.

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.

Example 5: Using Time/Deformation Type Table

In this example, the strain path of *Example 2: Strain Control, Uniaxial Extension* is specified with a time, strain table

```
begin legs
  using time, strain
  0. 0.0 0.0 0.0 0.0 0.0 0.0
```

```
  1. 0.1 0.0 0.0 0.0 0.0 0.0
end legs
```

Example 6: Using dt/Deformation Type Table

In this example, the strain path of *Example 2: Strain Control, Uniaxial Extension* is specified with a time step, strain table

```
begin legs
  using dt, strain
  0. 0.0 0.0 0.0 0.0 0.0 0.0
  .1 0.1 0.0 0.0 0.0 0.0 0.0
end legs
```

6.3 Optional Blocks

The following blocks are optional: `mathplot`.

6.3.1 `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 semi-colon delimited variables.

6.4 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_ROOT/Aux/MaterialsDatabase` directory, in that order.

6.5 Example: A Complete Input File

Below is an input file, any keywords not given take the default values shown in *The boundary Block Keywords*. In this input file, a material defined by the `elastic` constitutive model is cycled through the same deformation path by first prescribing the strain, then the stress, strain rate, stress rate, and finally the deformation gradient.

```
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
```

CHAPTER

SEVEN

RUNNING TESTS

Payette provides a mechanism for running material model regression tests with the `testPayette` script. `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 output against a "gold" result, allowing model developers to instantly see if and how changes to a material model effect its output.

7.1 Basic Usage of `testPayette`

From a command prompt execute:

```
% testPayette
```

`testPayette` will scan `PAYETTE_ROOT/Tests` for tests. After creating a lists of tests to run, `testPayette` will create a `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_ROOT/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.

7.1.1 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
```

7.2 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
```

Warning: When run on multiple processors, you cannot kill `testPayette` by `ctrl-c`, but must put the job in the background and then kill it. Any help from a python multiprocessor expert on getting around this limitation would be greatly appreciated!

7.3 Other `testPayette` Options

Execute:

```
% testPayette -h
```

for a complete list of options.

7.4 html Test Summary

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

7.5 Mathematca Post Processing

For the Kayenta material model, `testPayette` creates Rebecca 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.

7.6 Creating New Tests

Todo

need to add this section

CHAPTER

EIGHT

INSTALLING NEW MATERIALS

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 required elements of the constitutive model interface and instructions on compiling the material's Fortran source (if applicable) are provided.

8.1 Material File Naming Conventions

Each material model must provide its Python class definition in an "interface" file in the `PAYETTE_ROOT/Source/Materials`. The accepted naming convention for the material's interface file is:

```
PAYETTE_ROOT/Source/Materials/Payette_material_name.py
```

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

```
PAYETTE_ROOT/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_ROOT/Source/Materials/Payette_porous_rock.py
PAYETTE_ROOT/Source/Materials/Fortran/PorousRock
```

Note: 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 Interface File Required Attributes

The interface file must provide an `attributes` with the following keys

`attributes["payette material"]`
Boolean. Does the material represent an interface file, or not.

`attributes["name"]`
String. The material name.

`attributes["fortran source"]`
Boolean. Does the material have additional Fortran source code.

`attributes["build script"]`
String. Absolute path to Fortran build script, if applicable.

`attributes["aliases"]`
List. List of optional names.

`attributes["material type"]`
List. List of keyword descriptors of material type. Examples are "mechanical", "electro-mechanical".

Note: Using the `attributes` dictionary outside of the material's class definition allows *Payette* to scan `PAYETTE_ROOT/Source/Materials/` directory to find and import only the *Payette* material interface files during the build process.

8.3 Constitutive Model API: Required Elements

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 `ConstitutiveModelPrototype` base class.

8.3.1 Inheritance From Base Class

A new material model `MaterialModel` is only recognized as a material model by Payette if it inherits from the `ConstitutiveModelPrototype` base class:

```
class MaterialModel(ConstitutiveModelPrototype):
```

8.3.2 Required Data

`MaterialModel.`**`aliases`**
The aliases by which the constitutive model can be called (case insensitive).

`MaterialModel.`**`bulk_modulus`**
The bulk modulus. Used for determining the material's Jacobian matrix

`MaterialModel.`**`imported`**
Boolean indicating whether the material's extension library (if applicable) was imported.

`MaterialModel.`**`name`**
The name by which users can invoke the constituve model from the input file (case insensitive).

`MaterialModel.`**`nprop`**
The number of required parameters for the model.

`MaterialModel.`**`shear_modulus`**
The shear modulus. Used for determining the material's Jacobian matrix

8.3.3 Required Functions

`MaterialModel.`**`__init__`**`()`
Instantiate the material model. Register parameters with *Payette*.

`MaterialModel.`**`setUp`**`(`*simdat*, *matdat*, *user_params*, *f_params*`)`
Check user inputs and register extra variables with *Payette*. *simdat* and *matdat* are the simulation and material data containers, respectively, *user_params* are the parameters read in from the input file, and *f_params* are parameters from a parameters file.

`MaterialModel.`**`updateState`**`(`*simdat*, *matdat*`)`
Update the material state to the end of the current time step. *simdat* and *matdat* are the simulation and material data containers, respectively.

8.4 Example: Elastic Material Model Interface File

The required elements of the material's interface file described above are now demonstrated by an annotated version of the elastic material's interface.

View the source code: `Payette_elastic.py`

```
import sys
import os
import numpy as np

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

Note: The `Source.Payette_utils` module contains public methods for interfacing with *Payette*.

```
attributes = {
    "payette material":True,
    "name":"elastic",
    "fortran source":True,
    "build script":os.path.join(Payette_Materials_Fortran,"Elastic/build.py"),
    "aliases":["hooke","elasticity"],
    "material type":["mechanical"]
    }

try:
    import Source.Materials.Library.elastic as mtllib
    imported = True
except:
    imported = False
    pass
```

Note: We don't raise an exception just yet if the material's extension library is not importable. This allows users to run simulations even if all materials were not imported. Of course, if you try to run a simulation with a material that is not imported, an exception is raised.

```
class Elastic(ConstitutiveModelPrototype):
    """
    CLASS NAME
       Elastic

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

    METHODS
       Private:
         _check_props
         _set_field

       Public:
         setUp
         updateState

    FORTRAN
       The core code for the Elastic material is contained in
       Fortran/Elastic/elastic.f.  The module Library/elastic is created by f2py.
       elastic.f defines the following public subroutines

          hookechk: fortran data check routine called by _check_props
          hookerxv: fortran field initialization  routine called by _set_field
          hooke_incremental: fortran stress update called by updateState

       See the documentation in elastic.f for more information.

    AUTHORS
       Tim Fuller, Sandia National Laboratories, tjfulle@sandia.gov
    """
    def __init__(self):
        ConstitutiveModelPrototype.__init__(self)
```

Note: The base ConstitutiveModelPrototype class must be initialized.

```
self.name = attributes["name"]
self.aliases = attributes["aliases"]
self.imported = imported
```

Note: The required elastic material data `name`, `aliases`, and `imported` are assigned from the interface files `attributes` dictionary and the file scope variable `imported`.

Note: Below, the elastic material's parameters are registered with *Payette* through the `registerParameter` function:

`self.`**`registerParameter`**(*name*, *ui_loc*, *aliases*=[ ])

Register the parameter *name* with *Payette*. *ui_loc* is the integer location (starting at 0) of the parameter in the material's user input array. *aliases* are aliases by which the parameter can be specified in the input file.

```
# register parameters
self.registerParameter("LAM",0,aliases=[])
self.registerParameter("G",1,aliases=['SHMOD'])
self.registerParameter("E",2,aliases=['YMOD'])
self.registerParameter("NU",3,aliases=['POISSONS'])
self.registerParameter("K",4,aliases=['BKMOD'])
self.registerParameter("H",5,aliases=[])
self.registerParameter("KO",6,aliases=[])
self.registerParameter("CL",7,aliases=[])
self.registerParameter("CT",8,aliases=[])
self.registerParameter("CO",9,aliases=[])
self.registerParameter("CR",10,aliases=[])
self.registerParameter("RHO",11,aliases=[])
self.nprop = len(self.parameter_table.keys())
self.ndc = 0
pass
```

Note: `self.ndc` is the number of derived constants. This model has none.

```
# Public methods
def setUp(self,simdat,matdat,user_params,f_params):
    iam = self.name + ".setUp(self,material,props)"

    if not imported: return

    # parse parameters
    self.parseParameters(user_params,f_params)
```

Note: `parseParameters` passes the user input read from the input file to the initial user input array `self.UI0`. There is not return value.

```
# check parameters
self.dc = np.zeros(self.ndc)
self.ui = self._check_props()
self.nsv,namea,keya,sv,rdim,iadvct,itype = self._set_field()
```

Note: `_check_props` and `_set_field` are private functions that check the user input and assign initial values to extra variables.

```
namea = parseToken(self.nsv,namea)
keya = parseToken(self.nsv,keya)

# register the extra variables with the payette object
matdat.registerExtraVariables(self.nsv,namea,keya,sv)
```

Note: Above, the elastic material model registers its extra variables with the material data container.

`DataContainer.`**`registerExtraVariables`**(*nxv*, *namea*, *keya*, *exinit*)

Register extra varaibles with the data container. *nxv* is the number of extra variables, *namea* and *keya* are ordered lists of extra variable names and plot keys, respectively, and *exinit* is a list of initial values.

```
self.bulk_modulus,self.shear_modulus = self.ui[4],self.ui[1]
pass
```

Note: By default, *Payette* computes the material's Jacobian matrix numerically through a central difference algorithm. For some materials, like this elastic model, the Jacobian is constant. Here, we redefine the Jacobian to return the intial value.

```
# redefine Jacobian to return initial jacobian
def jacobian(self,simdat,matdat):
    if not imported: return
    v = simdat.getData("prescribed stress components")
    return self.J0[[[x] for x in v],v]

def updateState(self,simdat,matdat):
    """
       update the material state based on current state and strain increment
    """
    if not imported: return
```

Note: The *simdat* and *matdat* data containers contain all current data. Data is accessed by the `DataContainer.getData(name)` method.

```
dt = simdat.getData("time step")
d = simdat.getData("rate of deformation")
sigold = matdat.getData("stress")
svold = matdat.getData("extra variables")

a = [dt,self.ui,sigold,d,svold,migError,migMessage]
if not Payette_F2Py_Callback: a = a[:-2]
sig, sv, usm = mtllib.hooke_incremental(*a)
```

Note: `hooke_incremental(*a)` is a Fortran subroutine that performs the actual physics. Below, we store the update values of the extra variables and the stress.

```
matdat.storeData("extra variables",sv)
matdat.storeData("stress",sig)

return
```

8.5 Building Material Fortran Extension Modules in *Payette*

Note: This is not an exhaustive tutorial for how to link Python programs with compiled source code. Instead, it demonstrates through an annotated example the strategy that *Payette* uses to build and link with material models written in Fortran.

The strategy used in *Payette* to build and link to material models written in Fortran is to use *f2py* to compile the Fortran source in to a shared object library recognized by Python. The same task can be accomplished through Python's built in ctypes, weave, or other methods. We have found that *f2py* offers the most robust and easy to use solution. For more detailed examples of how to use compiled libraries with Python see Using Python as glue at the SciPy website or Using Compiled Code Interactively on Sage's website.

Rather than provide an exhaustive tutorial on linking Python programs to compiled libraries, we demonstrate how the `elastic` material model accomplishes this task through annotated examples.

8.5.1 Creating the Elastic Material Signature File

First, a Python signature file for the `elatic` material's Fortran source must be created. A signature file is a Fortran 90 file that contains all of the information that is needed to construct Python bindings to Fortran (or C) functions.

For the elastic model, change to `PAYETTE_ROOT/Source/Materials/Fortran/Elastic` and execute

```
% f2py -m elastic -h elastic.signature.pyf elastic.F
```

which will create the `elastic.signature.pyf` signature file.

f2py will create a signature for every function in `elastic.F`. However, only three public functions need to be bound to our Python program. So, after creating the signature file, all of the signatures for the private functions can safely be removed.

The signature file can be modified even further. See the above links on how to specialize your signature file for maximum speed and efficiency.

View the elastic.signature.pyf file: `elastic.signauture.pyf`

8.5.2 Elastic Material Build Script

Materials are built by *f2py* through the `MaterialBuilder` class from which each material derives its `Build` class. The `Build` class must provide a `build_extension_module` function, as shown below in the elastic material's build script.

View the elastic material build script: `build.py`

```
import os,sys

from Payette_config import *
from Source.Payette_utils import BuildError
from Source.Materials.Payette_build_material import MaterialBuilder

class Build(MaterialBuilder):

    def __init__(self,name,libname,compiler_info):

        fdir,fnam = os.path.split(os.path.realpath(__file__))
        self.fdir, self.fnam = fdir, fnam

        # initialize base class
        MaterialBuilder.__init__(self,name,libname,fdir,compiler_info)

        pass

    def build_extension_module(self):

        # fortran files
        self.source_files = [os.path.join(self.fdir,x) for x in os.listdir(self.fdir)
                             if x.endswith(".F")]

        self.build_extension_module_with_f2py()

        return 0
```

Note: For the elastic material, the `build_extension_module` function defines the Fortran source files and the calls the base class's `build_extension_module_with_f2py` function.

CHAPTER

NINE

INDICES AND TABLES

- *genindex*
- *search*

INDEX