NEML Documentation

Release 1.5.2

Argonne National Laboratory

Aug 18, 2023

CONTENTS

NEML is a modular library for creating constitutive models for structural materials. While it was originally focused on materials slated for use in high temperature nuclear reactors, it covers a wide range of constitutive models used for all types of structural materials.

NEML can be tied into any finite element program that can use a C, C++, or Fortran interface. This distribution provides an Abaqus UMAT driver as an example of how to accomplish this. Additionally, NEML comes with a Python interface that can be used to develop, debug, and test material models. The neml python module includes helper routines to simulate various common experimental conditions using NEML material models.

Go *here* to started with NEML by compiling it, running examples, and learn how to link it to your finite element code or go *here* to learn more about the structure of the library.

CHAPTER

ONE

ABOUT NEML

1.1 What NEML is

NEML (the Nuclear Engineering Material model Library) is a tool for creating and running structural material models. While it was originally developed to model high temperature nuclear reactors, the tool is general enough to apply to most types of structural materials.

The focus of NEML is on modularity and extensibility. The library is structured so that adding a new feature to an existing material model should be as simple as possible and require as little code as possible.

NEML material models are modular – they are built up from smaller pieces into a complete model. For example, a model might piece together a temperature-dependent elasticity model, a yield surface, a flow rule, and several hardening rules. Each of these submodels is independent of the other objects so that, for example, switching from conventional J_2 plasticity to a non-J_2 theory requires only a one line change in an input file, if the model is already implemented, or a relatively small amount of coding to add the new yield surface if it has not been implemented. All of these objects are interchangeable. For example, the damage, viscoplastic, and rate-independent plasticity models all use the same yield (flow) surfaces, hardening rules, elasticity models, and so on.

As part of this philosophy, the library only requires new components provide a few partial derivatives and NEML uses this information to assemble the Jacobian needed to do a fully implement, backward Euler integration of the ordinary differential equations comprising the model form and to provide the algorithmic tangent needed to integrate the model into an implicit finite element framework.

There are two general ways to create and interface with NEML material models: the python bindings and the compiled library with XML input. The python bindings are generally used for creating, fitting, and debugging new material models. In python, a material model is built up object-by-object and assembled into a complete mathematical constitutive relation. NEML provides several python drivers for exercising these material models in simple loading configurations. These drivers include common test types, like uniaxial tension tests and strain-controlled cyclic fatigue tests along with more esoteric drivers supporting simplified models of high temperature pressure vessels, like n-bar models and generalized plane-strain axisymmetry. NEML provides a full Abaqus UMAT interface and examples of how to link the compiled library into C, C++, or Fortran codes. These interfaces can be used to call NEML models from finite element codes. When using the compiled library, NEML models can be created and archived using a hierarchical XML format.

NEML is developed under a strict quality assurance program. Because, as discussed below, the NEML distribution does not provide models for any actual materials, ensuring the quality of the library is a verification problem – testing to make sure that NEML is correctly implementing the mathematical models – rather than a validation problem of comparing the results of a model to an actual test. This verification is done with extensive unit testing through the python interface. This unit testing verifies every mathematical function and every derivative in the library is correctly implemented.

1.2 What NEML is not

NEML does not provide a database of models for any particular class of materials. There are many example materials contained in the library release, these models are included entirely for illustrative purposes and do not represent the response of any actual material.

NEML will not be the fastest constitutive model when call from an external FE program. The focus of the library is on extensibility, rather than computational efficiency.

1.3 Mathematical conventions

1.3.1 Mandel notation

NEML models work in three dimensions. This means second order tensors can be expressed by 3-by-3 arrays and fourth order tensors are 3-by-3-by-3-by-3 arrays. This three dimensional interface can naturally accommodate 3D and 2D plane strain stress updates. A python example demonstrates how to use the 3D interface to degenerate the models to the standard strain-controlled uniaxial material interface where the stress in the loading direction is strain controlled and all the remaining stress components are stress controlled to zero stress.

NEML uses the Mandel notation to convert symmetric second and fourth order tensors to vectors and matrices. The convention transforms the second order tensor

$$\begin{bmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} \\ \sigma_{12} & \sigma_{22} & \sigma_{23} \\ \sigma_{13} & \sigma_{23} & \sigma_{33} \end{bmatrix} \rightarrow \begin{bmatrix} \sigma_{11} & \sigma_{22} & \sigma_{33} & \sqrt{2}\sigma_{23} & \sqrt{2}\sigma_{13} & \sqrt{2}\sigma_{12} \end{bmatrix}$$

and, after transformation, a fourth order tensor $\mathfrak{C}$ becomes

$$\begin{bmatrix} C_{1111} & C_{1122} & C_{1133} & \sqrt{2}C_{1123} & \sqrt{2}C_{1113} & \sqrt{2}C_{1112} \\ C_{1122} & C_{2222} & C_{2233} & \sqrt{2}C_{2223} & \sqrt{2}C_{2213} & \sqrt{2}C_{2212} \\ C_{1133} & C_{2233} & C_{3333} & \sqrt{2}C_{3323} & \sqrt{2}C_{3313} & \sqrt{2}C_{3312} \\ \sqrt{2}C_{1123} & \sqrt{2}C_{2223} & \sqrt{2}C_{3323} & 2C_{2323} & 2C_{2313} & 2C_{2312} \\ \sqrt{2}C_{1113} & \sqrt{2}C_{2213} & \sqrt{2}C_{3313} & 2C_{2313} & 2C_{1313} & 2C_{1312} \\ \sqrt{2}C_{1112} & \sqrt{2}C_{2212} & \sqrt{2}C_{3312} & 2C_{2312} & 2C_{1312} & 2C_{1212} \end{bmatrix}.$$

For symmetric two second order tensors $\mathbf{A}$ and $\mathbf{B}$ and their Mandel vectors $\hat{\mathbf{a}}$ and $\hat{\mathbf{b}}$ the relation

$$\mathbf{A} : \mathbf{B} = \hat{\mathbf{a}} \cdot \hat{\mathbf{b}}$$

expresses the utility of this convention. Similarly, given the symmetric fourth order tensor $\mathfrak{C}$ and its equivalent Mandel matrix $\hat{\mathbf{C}}$ contraction over two adjacent indices

$$\mathbf{A} = \mathfrak{C} : \mathbf{B}$$

simply becomes matrix-vector multiplication

$$\hat{\mathbf{a}} = \hat{\mathbf{C}} \cdot \hat{\mathbf{b}}.$$

The Mandel convention is relatively uncommon in finite element software, and so the user must be careful to convert back and forth from the Mandel convention to whichever convention the calling software uses. The Abaqus UMAT interface provided with NEML demonstrates how to make this conversion before and after each call.

Typographically, the manual uses a lower case, standard font (x) to represent a scalar, a bold lower case to represent a vector ($\mathbf{x}$), a bold upper case to represent a second order tensor ($\mathbf{X}$), and a bold upper case Fraktur to represent a fourth

order tensor ($\mathfrak{X}$). There are certain exceptions to the upper case/lower case convention for commonly used notation. For example the stress and strain tensors are denoted σ and ε, respectively. Internally in NEML all tensor operations are implemented as Mandel dot products. However, the documentation writes the full tensor form of the equations.

Throughout the documentation the deviator of a second order tensor is denoted:

$$\mathrm{dev}\left(\mathbf{X}\right) = \mathbf{X} - \frac{1}{3}\mathrm{tr}\left(\mathbf{X}\right)\mathbf{I}$$

with tr the trace and $\mathbf{I}$ the identity tensor.

When describing collections of objects the manual uses square brackets. For example,

$$\left[\begin{array}{ccc} s & \mathbf{X} & \mathbf{v} \end{array}\right]$$

Indicates a collection of a scalar s, a vector representing a second order tensor $\mathbf{x}$ in Mandel notation, and a vector $\mathbf{v}$. These collections are ordered. This notation indicates the implementation is concatenating the quantities into a flat, 1D array (in this case with length $1 + 6 + 3 = 10$).

1.3.2 Interfaces

The documentation describes NEML as a collection of interfaces. An interface is a collection of functions with the same inputs but different outputs. These interfaces are implemented as C++ objects in NEML. The documentation describes an interface with the notation:

$$a, \mathbf{B}, \mathfrak{C} \leftarrow \mathcal{F}\left(d, \mathbf{e}, \mathbf{F}\right)$$

This is an interface that takes a scalar d, vector $\mathbf{e}$, and second order tensor $\mathbf{F}$ as input and returns a scalar a, second order tensor $\mathbf{B}$, and fourth order symmetric tensor $\mathfrak{C}$ as output. The interface might be implemented as three individual functions

$$\begin{aligned} a &= f\left(d, \mathbf{e}, \mathbf{F}\right) \\ \mathbf{B} &= \mathbf{F}\left(d, \mathbf{e}, \mathbf{F}\right) \\ \mathfrak{C} &= \mathfrak{F}\left(d, \mathbf{e}, \mathbf{F}\right). \end{aligned}$$

CHAPTER

TWO

GETTING STARTED

2.1 Installing NEML

Compiling NEML requires the NEML source, and a C++ compiler. Additionally, you will need BLAS and LAPACK.

Testing NEML and running the examples described here also requires compiling the Python bindings. This has the additional requirements of a Python 3.x installation, including the development headers, the pybind11 library (which is included in the NEML source), and the numpy, scipy, and networkx python packages. The nose python package can be useful in running the provided tests.

Linking NEML into your finite element package may additionally require a C or Fortran compiler, depending on what language your finite element software uses and can link to.

We have successfully installed NEML on Linux, Windows, and Mac OS. Directions for each operating system follow below.

2.1.1 Linux

Basic library

This instructions assume a clean installation of Ubuntu 18.04.1 LTS. Installation on other Linux systems will be similar. The package maintainers have compiled and run the library on CentOS and Debian without difficulty. The basic steps remain the same.

First install the prerequisites

```
apt-get install build-essential git cmake libblas-dev liblapack-dev
```

Clone the neml source code

```
git clone https://github.com/Argonne-National-Laboratory/neml.git
```

NEML builds in the source directory. Enter the NEML directory and configure NEML using CMake:

```
cmake .
```

Useful options might include `-D CMAKE_BUILD_TYPE="Release"` if you want to build an optimized version for production runs.

Then simply make the library:

```
make
```

Python bindings

Building the base library is sufficient to link NEML into external finite element software However, all the tests and provided examples use the Python bindings.

To build the bindings you will need a few more prerequisites, in addition to those mentioned above:

```
apt-get install python3-dev python3-networkx python3-numpy python3-scipy python3-
↪matplotlib python3-nose
```

Configure with CMake to setup the python bindings:

```
cmake -D WRAP_PYTHON=ON .
```

And build the library

```
make
```

You can now run the test suit with

```
nosetests
```

Running examples

Once you have the python bindings you can test your compilation of NEML using the python tests in the `tests/` directory. If you installed nose, all the tests can be run from the root `neml` directory by running **`nosetests`**.

Assuming the tests passed, you can begin to build material models with NEML. The manual has a *Tutorial* giving a brief tutorial on setting up a material model either with the python bindings or the XML input files and then running that model using the python drivers for some simple loadings. Additional examples can be found in the `examples/` directory.

Linking to external software

The main NEML library (in the `lib/`) directory is all that needs to be linked to your software to call NEML material models. You only need to include the `src/neml_interface.h` in order to load material models from XML datafile and use the resulting C++ object to call for the material response.

The `util/` directory contains example bindings of NEML into C++, C, and Fortran codes. The CMake variable `-D BUILD_UTILS=ON` option compiles these example interfaces. Turing this option on requires a Fortran and C compiler. Looking at these examples demonstrates how you can integrate NEML into your finite element code.

Abaqus UMAT interface

The `util/abaqus` directory contains a full UMAT interface that can be used to tie NEML into Abaqus. This first requires compiling the *main NEML library*. Say the full path to `libneml.so` is `${NEMLROOT}/lib/libneml.so`. You would need to alter your abaqus env file (for example `abaqus_v6.env`) to *add* the library to the `link_sl` command. For example, if the existing `link_sl` is:

```
link_sl = [fortCmd,
           '-cxxlib', '-fPIC', '-threads', '-shared','-Wl,--add-needed',
           '%E', '-Wl,-soname,%U', '-o', '%U', '%F', '%A', '%L', '%B', '-parallel',
           '-Wl,-Bdynamic', '-shared-intel']
```

then you would alter it to

```
link_sl = [fortCmd,
           '${NEMLROOT}/lib/libneml.so', '-V',
           '-cxxlib', '-fPIC', '-threads', '-shared','-Wl,--add-needed',
           '%E', '-Wl,-soname,%U', '-o', '%U', '%F', '%A', '%L', '%B', '-parallel',
           '-Wl,-Bdynamic', '-shared-intel']
```

You then need to determine the correct number of `*DEPVAR` and the correct `INITIAL CONDITIONS, TYPE=SOLUTION` to include in your input file in order to have Abaqus setup and maintain the correct number of history variables for the NEML model. The distribution provides a simple program in the `util/abaqus/` directory to report this information. The program, called `report` is compiled if the CMake `BUILD_UTILS` option is set. It requires two command line arguments:

report

file

Name of the XML input file

model

Material model to report on in the XML file

The program will print the correct lines to use in your Abaqus input file for that NEML material.

You should then copy the XML file containing the model you want to run to the directory containing the Abaqus input file. You must rename this XML input file to `neml.xml`. You should rename the model in that file you want to use in Abaqus to `abaqus`. The UMAT is hardcoded to load that material from that filename.

The remaining steps are standard for any UMAT. You need to request Abaqus call the UMAT in the input file:

```
*MATERIAL, NAME=CUSTOM

*USER MATERIAL, CONSTANTS=0, UNSYMM
```

Remembering to also include the output from `report` to initalize the required history variables.

Finally, run the UMAT

```
abaqus job=xxxx user=/path/to/neml/util/abaqus/nemlumat.f
```

ANSYS UMATERIAL interface

Directions are in preparation

2.1.2 macOS

Basic library

These instructions assume a clean installation of macOS Catalina 10.15 but may work on other version of the operating system. The instructions rely on homebrew to install the required packages. The procedure here uses the native LLVM install for compiling the library.

First install the prerequisites:

```
brew install cmake openblas superlu
```

Then clone the neml source code

```
git clone https://github.com/Argonne-National-Laboratory/neml.git
```

move into the neml directory, and configure the library using `cmake`

```
cmake -D CMAKE_BUILD_TYPE=Release -D USE_OPENMP=OFF .
```

The build type can be switched to `CMAKE_BUILD_TYPE=Debug` to compile a debug version of the library.

Finally build the library with

```
make
```

Python bindings

To compile the python bindings install Python3 using homebrew and then install the python package dependencies using pip:

```
brew install python
pip3 install --user networkx numpy scipy matplotlib nose
```

Configure the library in the `neml` directory

```
cmake -D CMAKE_BUILD_TYPE=Release -D WRAP_PYTHON=ON -D PYTHON_EXECUTABLE=$(python3-
↪config --prefix)/bin/python3.9 -D PYTHON_LIBRARY=$(python3-config --prefix)/lib/
↪libpython3.9.dylib -D PYTHON_INCLUDE_DIR=$(python3-config --prefix)/include/python3.9 -
↪D USE_OPENMP=OFF .
```

These instructions assume the current homebrew python version is `python3.9`. The python directories will need to be changed if you have a different version of python installed.

Compile the library as before

```
make
```

Finally, you can run the automated test suite with

```
~/Library/Python/3.9/bin/nosetests
```

The full path is required as `pip` does not install the `nosetests` script in a `PATH` location by default.

2.1.3 Windows

We compile and test NEML on Windows using the MSYS2 system. This provides a linux-like build environment for Windows. We build and test the library using the mingw-w64 compiler. MSYS2 supports other compilers and if needed you may be able to compile and run NEML using one of the other support compilers. These directions assume the use of mingw.

Basic library

Go to the MSYS2 website and follow the instructions there to install the framework. Follow the directions all the way through updating the package repository and installing mingw with the command:

```
pacman -Su
pacman -S --needed base-devel mingw-w64-x86_64-toolchain
```

As the directions state, at this point close the original MSYS2 window, go to the start menu, and run *"MSYS MinGW 64-bit"*. You are now ready to obtain the NEML source code and compile the library

Use the *"MSYS MinGW 64-bit"* terminal you opened to navigate to the location you want to download and install NEML to. Install the required dependencies with

```
pacman -S mingw-w64-x86_64-openblas mingw-w64-x86_64-cmake git
```

You can now clone the NEML source repository and enter the NEML directory with

```
git clone https://github.com/Argonne-National-Laboratory/neml.git
cd neml
```

Finally, configure the base library with CMake and build the library with

```
cmake . -G"MSYS Makefiles"
make
```

This will build the NEML dynamic library, located in *lib* subdirectory in the NEML install.

Python bindings

To go on and build the Python binding first follow the instructions in the previous section for the base library. Then install a few additional dependencies with

```
pacman -S mingw-w64-x86_64-python-pip mingw-w64-x86_64-python-numpy mingw-w64-x86_64-
↪python-scipy mingw-w64-x86_64-python-networkx mingw-w64-x86_64-python-matplotlib mingw-
↪w64-x86_64-python-nose
```

Configure and build the Python bindings

```
cmake . -G"MSYS Makefiles" -D"WRAP_PYTHON=ON"
make
```

This produces the NEML python package in the *neml* subdirectory.

You can run the automatic test suite using *nose* with the command

```
nosetests-3.8.exe
```

To use the *neml* Python package outside the source directory you need to add the *neml\neml* subdirectory to both the system *PATH* and *PYTHONPATH* environment variables.

2.2 Python package

An easier way to install NEML if you are only interested in the python bindings is to use the package uploaded to pypi. Currently only a source package is available, but the python install scripts simplify the process of compiling and linking the package. You still need a working compiler, the python development headers, and development versions of BLAS and LAPACK. These can be installed on Ubuntu, for example, with

```
sudo apt-get install python3-dev python3-pip cmake libboost-dev libblas-dev liblapack-dev
```

After that a python package manger, such as pip, can be used to install NEML

```
sudo pip3 install neml
```

CHAPTER

THREE

TUTORIAL

This tutorial assumes you have already *compiled* NEML, including the python bindings. The python and XML files built up in the tutorial are included in the `examples/` directory of NEML as `examples/tutorial.py` and `examples/tutorial.xml`.

3.1 Building a model in python

A NEML material model is assembled hierarchical from smaller constitutive parts. For example, a rate-dependent viscoplastic material model needs:

1. An elastic model.
2. A flow surface describing the initial inelastic domain.
3. A hardening rule describing how that flow surface evolves with time, temperature, and inelastic deformation.
4. A flow rule relating stress to inelastic strain.

With the python bindings you assemble models piece-by-piece.

First we import the required modules from neml, along with matplotlib to plot some results.

```
from neml import solvers, models, elasticity, drivers, surfaces, hardening, visco_flow,␣
↪general_flow, parse

import matplotlib.pyplot as plt
```

In this example we define the isotropic elastic model with the Young's modulus and Poisson's ratio:

```
E = 150000.0
nu = 0.3
elastic = elasticity.IsotropicLinearElasticModel(E, "youngs", nu, "poissons")
```

We will use a simple J_2 flow theory for the flow surface:

```
surface = surfaces.IsoJ2()
```

and a simple linear hardening rule

```
sY = 100.0
H = 2500.0
iso = hardening.LinearIsotropicHardeningRule(sY, H)
```

Notice that NEML associates the initial yield stress `s0` with the hardening model, rather than the flow surface.

We use a classical Perzyna flow rule and assume a power law relation between stress and inelastic strain rate.

```
gpower = visco_flow.GPowerLaw(n, eta)
vflow = visco_flow.PerzynaFlowRule(surface, iso, gpower)
```

We tell NEML how to integrate this flow rule

```
integrator = general_flow.TVPFlowRule(elastic, vflow)
```

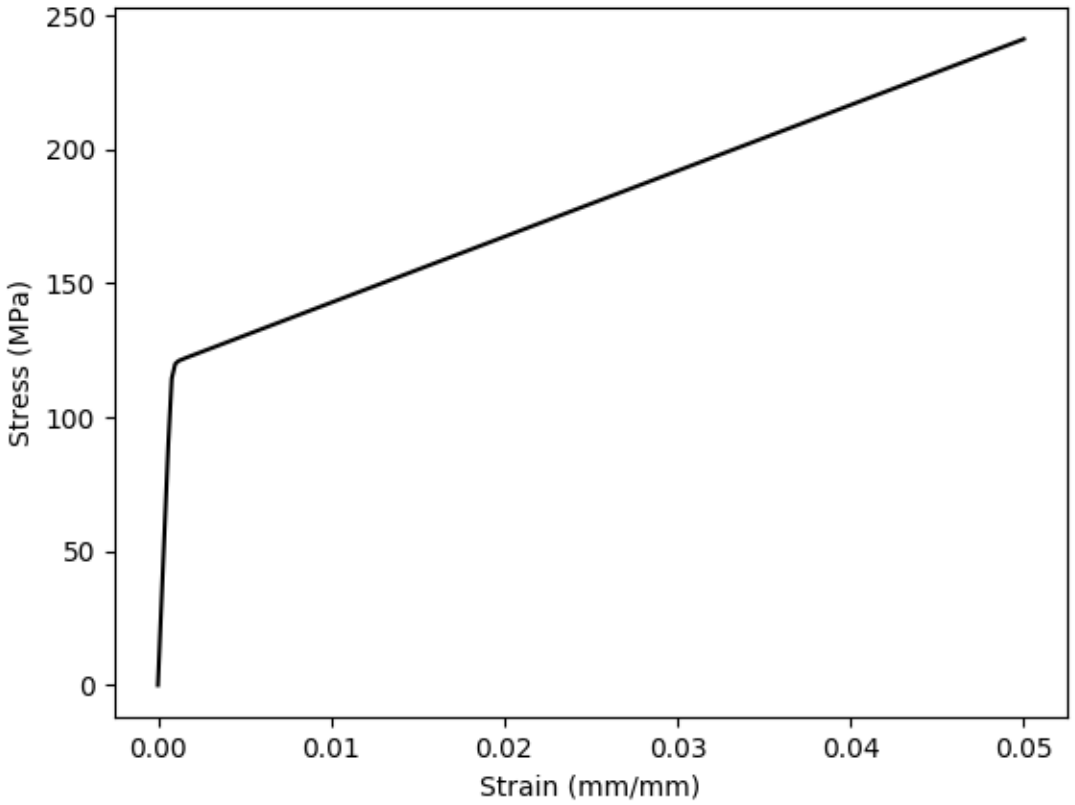
and assemble the whole thing into a material model

```
model = models.GeneralIntegrator(elastic, integrator)
```

We can use some simple python drivers included in NEML to test this model. For example, we can simulate a uniaxial tension test and plot the results

```
erate = 1.0e-4
res = drivers.uniaxial_test(model, erate)

plt.figure()
plt.plot(res['strain'], res['stress'], 'k-')
plt.xlabel("Strain (mm/mm)")
plt.ylabel("Stress (MPa)")
plt.show()
```



3.2 The same model in an XML file

NEML can also store definitions of material models in XML files. These files are useful for archiving material models and providing input to NEML when it is linked into finite element analysis software.

The XML file structure is generated *automatically* from the class structure of the NEML material model. The model developed in python above could be stored in an XML file as:

```
<materials>
  <tutorial_model type="GeneralIntegrator">
    <elastic type="IsotropicLinearElasticModel">
      <m1>150000.0</m1>
      <m1_type>youngs</m1_type>
      <m2>0.3</m2>
      <m2_type>poissons</m2_type>
    </elastic>

    <rule type="TVPFlowRule">
      <elastic type="IsotropicLinearElasticModel">
        <m1>150000.0</m1>
        <m1_type>youngs</m1_type>
        <m2>0.3</m2>
```

(continues on next page)

(continued from previous page)

```
        <m2_type>poissons</m2_type>
      </elastic>

      <flow type="PerzynaFlowRule">
        <surface type="IsoJ2"/>
        <hardening type="LinearIsotropicHardeningRule">
          <s0>100.0</s0>
          <K>2500.0</K>
        </hardening>
        <g type="GPowerLaw">
          <n>5.0</n>
          <eta>100.0</eta>
        </g>
      </flow>
    </rule>
  </tutorial_model>
</materials>
```

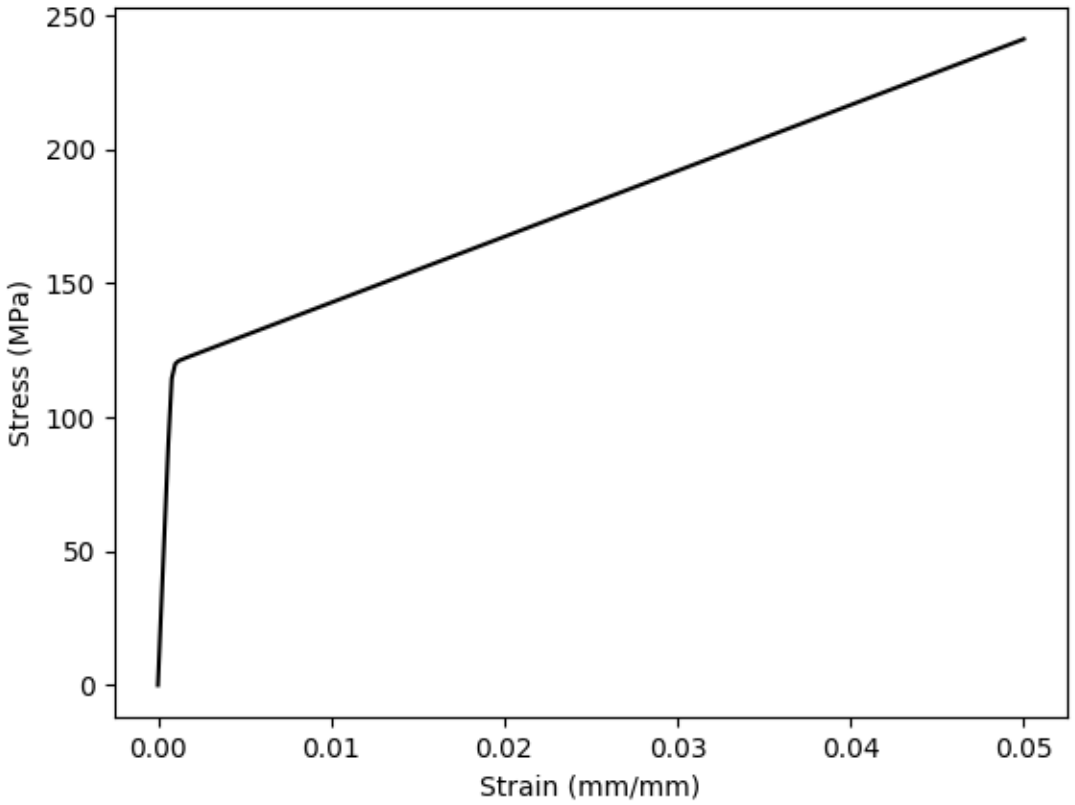
This model could be loaded into python

```
model2 = parse.parse_xml("tutorial.xml", "tutorial_model")

res2 = drivers.uniaxial_test(model2, erate)
```

and then used just as if it was assembled in python part by part

```
plt.figure()
plt.plot(res2['strain'], res2['stress'], 'k-')
plt.xlabel("Strain (mm/mm)")
plt.ylabel("Stress (MPa)")
plt.show()
```



A single XML file can hold more than one material. The top-level tag of each model gives a unique identifier used in the second argument of the `parse_xml` call to load the correct model. For example, this XML file (`example.xml`) has two material models:

```
<materials>
  <model_1 type="GeneralIntegrator">
    <elastic type="IsotropicLinearElasticModel">
      <m1>150000.0</m1>
      <m1_type>youngs</m1_type>
      <m2>0.3</m2>
      <m2_type>poissons</m2_type>
    </elastic>

    <rule type="TVPFlowRule">
      <elastic type="IsotropicLinearElasticModel">
        <m1>150000.0</m1>
        <m1_type>youngs</m1_type>
        <m2>0.3</m2>
        <m2_type>poissons</m2_type>
      </elastic>

      <flow type="PerzynaFlowRule">
        <surface type="IsoJ2"/>
        <hardening type="LinearIsotropicHardeningRule">
```

(continues on next page)

(continued from previous page)

```
        <s0>100.0</s0>
        <K>2500.0</K>
      </hardening>
      <g type="GPowerLaw">
        <n>5.0</n>
        <eta>100.0</eta>
      </g>
    </flow>
  </rule>
</model_1>

<model_2 type="SmallStrainRateIndependentPlasticity">
  <elastic type="IsotropicLinearElasticModel">
    <m1>84000.0</m1>
    <m1_type>bulk</m1_type>
    <m2>40000.0</m2>
    <m2_type>shear</m2_type>
  </elastic>

  <flow type="RateIndependentAssociativeFlow">
    <surface type="IsoKinJ2"/>
    <hardening type="CombinedHardeningRule">
      <iso type="VoceIsotropicHardeningRule">
        <s0>100.0</s0>
        <R>100.0</R>
        <d>1000.0</d>
      </iso>
      <kin type="LinearKinematicHardeningRule">
        <H>1000.0</H>
      </kin>
    </hardening>
  </flow>
</model_2>
</materials>
```

The two models would be loaded into python with:

```
model1 = parse.parse_xml("tutorial.xml", "model_1")
model2 = parse.parse_xml("tutorial.xml", "model_2")
```

For a description of how to use NEML and the XML input in an external finite element analysis program see the getting started *guide*.

CHAPTER

FOUR

INTERPOLATION FUNCTIONS

Throughout the code NEML uses interpolation functions to represent model parameters that depend on temperature or, occasionally, some other variable. A NEML interpolation function is a scalar function of a single variable $f(x)$.

The interface must also provide the first derivative of the function with respect that the variable.

These interpolation functions are used throughout NEML in place of scalar parameters. Except where noted, currently only for the InterpolatedIsotrpicHardeningRule, interpolation functions are used to give the parameter as a function of temperature. A constant parameter, e.g. one that does not depend on temperature can be expressed by using a *ConstantInterpolate* object.

4.1 Interpolate

The interface for all interpolate objects.

class **Interpolate** : public *neml*::*NEMLObject*

Base class for interpolation functions.

Subclassed by *neml::ConstantInterpolate*, neml::ExpInterpolate, *neml::GenericPiecewiseInterpolate*, neml::MTSInterpolate, *neml::MTSShearInterpolate*, *neml::PiecewiseLinearInterpolate*, *neml::PiecewiseLogLinearInterpolate*, neml::PiecewiseSemiLogXLinearInterpolate, *neml::PolynomialInterpolate*, neml::PowerLawInterpolate

Public Functions

Interpolate(*ParameterSet* &params)

virtual double **value**(double x) const = 0

Returns the value of the function.

virtual double **derivative**(double x) const = 0

Returns the derivative of the function.

double **operator()**(double x) const

Nice wrapper for function call syntax.

bool **valid**() const

Is the interpolate valid?

4.2 ConstantInterpolate

Expresses a constant parameter

$$f(x) = C.$$

class **ConstantInterpolate** : public *neml*::*Interpolate*

A constant value.

Public Functions

ConstantInterpolate(*ParameterSet* &params)

The parameter is the constant value!

virtual double **value**(double x) const

Returns the value of the function.

virtual double **derivative**(double x) const

Returns the derivative of the function.

Public Static Functions

static std::string **type**()

Type for the object system.

static *ParameterSet* **parameters**()

Create parameters for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Create object from a *ParameterSet*.

4.3 GenericPiecewiseInterpolate

A general piecewise interpolate. If $x \leq p_i$ then it calls interpolation function f_i. For $x > p_i$ it calls interpolation function f_{i+1}.

class **GenericPiecewiseInterpolate** : public *neml*::*Interpolate*

Generic piecewise interpolation.

Public Functions

GenericPiecewiseInterpolate(*ParameterSet* &params)

virtual double **value**(double x) const
Returns the value of the function.

virtual double **derivative**(double x) const
Returns the derivative of the function.

Public Static Functions

static std::string **type**()
Type for the object system.

static *ParameterSet* **parameters**()
Create parameters for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)
Create object from a *ParameterSet*.

4.4 PolynomialInterpolate

Polynomial interpolation of the type

$$f(x) = \sum_{i=1}^{n} c_i x^{i-1}.$$

The polynomial coefficients are given starting with the highest order, like in the numpy library.

class **PolynomialInterpolate** : public *neml*::*Interpolate*
Simple polynomial interpolation.

Public Functions

PolynomialInterpolate(*ParameterSet* &params)
Input is the coefficients of the polynomial, from highest to lowest order.

virtual double **value**(double x) const
Returns the value of the function.

virtual double **derivative**(double x) const
Returns the derivative of the function.

Public Static Functions

static std::string **type**()
Type for the object system.

static *ParameterSet* **parameters**()
Create parameters for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)
Create object from a *ParameterSet*.

4.5 PiecewiseLinearInterpolate

Piecewise linear interpolation where parameters give the x and y coordinates of the endpoints of each linear segment. For $x_i \leq x \leq x_{i+1}$

$$f(x) = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i) + y_i.$$

For $x < x_1$ the function returns y_1 and for $x > x_n$ the function returns y_n.

class **PiecewiseLinearInterpolate** : public *neml*::*Interpolate*
Piecewise linear interpolation.

Public Functions

PiecewiseLinearInterpolate(*ParameterSet* &params)
Parameters are a list of x coordinates and a corresponding list of y coordinates

virtual double **value**(double x) const
Returns the value of the function.

virtual double **derivative**(double x) const
Returns the derivative of the function.

Public Static Functions

static std::string **type**()
Type for the object system.

static *ParameterSet* **parameters**()
Create parameters for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)
Create object from a *ParameterSet*.

4.6 PiecewiseLogLinearInterpolate

Exactly like *PiecewiseLinearInterpolate* except the interpolation in y is done in log space. The input y values are given as $y_i = \ln v_i$ where v_i is the actual value of the function at x_i. For $x_i \leq x \leq x_{i+1}$

$$f(x) = \exp\left(\frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i) + y_i\right).$$

For $x < x_1$ the function returns $\exp(y_1)$ and for $x > x_n$ the function returns $\exp(y_n)$.

class **PiecewiseLogLinearInterpolate** : public *neml*::*Interpolate*

Piecewise loglinear interpolation.

Public Functions

PiecewiseLogLinearInterpolate(*ParameterSet* &params)

Similar to piecewise linear interpolation except the y coordinates are given as ln(y) and the interpolation is done in log space

virtual double **value**(double x) const

Returns the value of the function.

virtual double **derivative**(double x) const

Returns the derivative of the function.

Public Static Functions

static std::string **type**()

Type for the object system.

static *ParameterSet* **parameters**()

Create parameters for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Create object from a *ParameterSet*.

4.7 MTSShearInterpolate

The shear modulus interpolation used in the Mechanical Threshold Stress [MTS1999] flow stress model

$$f(x) = V_0 - \frac{D}{e^{T_0/x} - 1}.$$

class **MTSShearInterpolate** : public *neml*::*Interpolate*

The MTS shear modulus function proposed in the original paper.

Public Functions

MTSShearInterpolate(*ParameterSet* &params)

Interpolation using the MTS model form f(x) = V0 - D / (exp(T0 / x) - 1)

virtual double **value**(double x) const

Returns the value of the function.

virtual double **derivative**(double x) const

Returns the derivative of the function.

Public Static Functions

static std::string **type**()

Type for the object system.

static *ParameterSet* **parameters**()

Create parameters for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Create object from a *ParameterSet*.

4.8 Helper Functions

std::vector<std::shared_ptr<*Interpolate*>> *neml*::**make_vector**(const std::vector<double> &iv)

A helper to make a vector of constant interpolates from a vector.

std::vector<double> *neml*::**eval_vector**(const std::vector<std::shared_ptr<*Interpolate*>> &iv, double x)

A helper to evaluate a vector of interpolates.

std::vector<double> *neml*::**eval_deriv_vector**(const std::vector<std::shared_ptr<*Interpolate*>> &iv, double x)

A helper to evaluate the derivative of a vector of interpolates.

CHAPTER

FIVE

INTERFACES: TYPES OF MATERIAL MODELS

At the highest level a NEML model returns the stress-strain response of a material. This constitutive response depends on the input strain, temperature, and time along with a set of internal variables that the model maintains. When calling a model from another code the user must maintain the time series of strains, stresses, temperatures, times, history variables, and (optionally) total strain energy and inelastic dissipation. The user passes in the stress, history, and energy quantities at the previous time step, the strain, temperature, and time at both the previous and next time steps, and NEML provides the updated stresses, history, and energies as output.

NEML aims to provide a generic interface to external finite element software through the *NEMLModel* class. This class then provides routines that provide access to the constitutive response for different types of requested stress updates.

Subclasses of this class implement a certain type of stress update. For example, *NEMLModel_sd* implements small strain stress updates natively. However, these subclasses also provide wrapper interfaces around this native stress update to accommodate requests for other stress updates. For example, the *NEMLModel_sd* class provides an incremental large deformations stress update using an objective stress rate.

5.1 NEMLModel

5.1.1 Stress update interfaces

NEMLModel is the common interface to the constitutive models contained in NEML. It currently requires models to provide two types of stress updates. The first type of update is a small strain kinematics incremental update based on the interface

$$\sigma_{n+1}, \alpha_{n+1}, \mathfrak{A}_{n+1}, u_{n+1}, p_{n+1} \leftarrow \mathcal{M}\left(\varepsilon_{n+1}, \varepsilon_n, T_{n+1}, T_n, t_{n+1}, t_n, \sigma_n, \alpha_n, u_n, p_n\right).$$

Here n indicates values at the previous time step and $n+1$ values at the next time step. The quantities are stress (σ), strain (ε), the vector of history variables (α), strain energy (u) dissipated work (p), temperature (T), time (t), and the algorithmic tangent ($\mathfrak{A}$)

$$\mathfrak{A}_{n+1} = \frac{d\sigma_{n+1}}{d\varepsilon_{n+1}}.$$

The second update is a large strain kinematics update based on the interface

$$\sigma_{n+1}, \alpha_{n+1}, \mathfrak{A}_{n+1}, \mathfrak{B}_{n+1}, u_{n+1}, p_{n+1} \leftarrow \mathcal{M}\left(d_{n+1}, d_n, w_{n+1}, w_n, T_{n+1}, T_n, t_{n+1}, t_n, \sigma_n, \alpha_n, u_n, p_n\right).$$

Here d is the deformation rate tensor (the symmetric part of the spatial velocity gradient), w is the vorticity (the skew part of the spatial velocity gradient), $\mathfrak{A}$ is

$$\mathfrak{A}_{n+1} = \frac{d\sigma_{n+1}}{dd_{n+1}}$$

and $\mathfrak{B}$ is

$$\mathfrak{B}_{n+1} = \frac{d\sigma_{n+1}}{dw_{n+1}}$$

while the other quantities are defined identically to the small strain interface.

5.1.2 Implementations

NEMLModel_sd

Overview

The NEMLModel_sd object natively implements the *small strain* stress update interface.

It accommodates the *large strain incremental* stress update interface using the Treusdell objective stress rate of the form:

$$\dot{\sigma} = \hat{\dot{\sigma}} - \sigma \cdot L^T - L \cdot \sigma + \mathrm{tr}\,(L)\,\sigma$$

where σ is the Cauchy stress and $\hat{\dot{\sigma}}$ is the small strain stress rate implied by the small strain kinematics update interface. The update calculates the consistent tangents $\mathfrak{A}$ $\mathfrak{B}$ exactly and provides a helper routine to recombine these symmetric and skew parts into the full derivative with respect to the spatial velocity gradient.

Caution: The current Treusdell objective integration does not advect the material history variables. This means the integration of material models with vector or tensor history variables, such as backstresses, will be inaccurate for situations requiring large rotations. This limitation will be removed in future version of NEML.

The following sections describe the basic material model implemented from this generic interfaces. Another *section* of the manual details continuum damage models, which also use this same interface.

The descriptions here are given in the rate form. Details of the integration algorithm for most of the small strain models are described *here*.

Implementations

Linear elasticity

Overview

This object implements a simple linear elastic stress update of the type

$$\sigma_{n+1} = \mathfrak{C}_{n+1} : \varepsilon_{n+1}$$

where $\mathfrak{C}$ is a temperature-dependent elasticity tensor. The model does not maintain any history variables.

Parameters

Parameter	Object type	Description	Default
emodel	*neml::LinearElasticModel*	Temperature dependent elastic constants	No
alpha	*neml::Interpolate*	Temperature dependent instantaneous CTE	0.0

Class description

class **SmallStrainElasticity** : public *neml*::*NEMLModel_sd*

Small strain linear elasticity.

Public Functions

SmallStrainElasticity(*ParameterSet* &params)

Parameters are the minimum: an elastic model and a thermal expansion.

virtual void **update_sd_actual**(const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n, double *const A_np1, double &u_np1, double u_n, double &p_np1, double p_n)

Small strain stress update.

virtual void **populate_state**(*History* &h) const

Populate internal variables (none)

virtual void **init_state**(*History* &h) const

Initialize history (none to setup)

Public Static Functions

static std::string **type**()

Type for the object system.

static *ParameterSet* **parameters**()

Setup parameters for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

Perfect plasticity

Overview

This class implements rate independent perfect plasticity described by:

The elastic trial state:

$$\begin{aligned} \varepsilon_{tr}^p &= \varepsilon_n^p \\ \sigma_{tr} &= \mathfrak{C}_{n+1} : (\varepsilon_{n+1} - \varepsilon_{tr}^p) \end{aligned}$$

The plastic correction:

$$\begin{aligned} \sigma_{n+1} &= \mathfrak{C}_{n+1} : \left(\varepsilon_{n+1} - \varepsilon_{n+1}^p\right) \\ \varepsilon_{n+1}^p &= \begin{cases} \varepsilon_{tr}^p & f\left(\sigma_{tr}\right) \leq 0 \\ \varepsilon_{tr}^p + \frac{\partial f_{n+1}}{\partial \sigma_{n+1}} \Delta\gamma_{n+1} & f\left(\sigma_{tr}\right) > 0 \end{cases} \end{aligned}$$

Solving for $\Delta\gamma_{n+1}$ such that

$$f\left(\sigma_{n+1}\right) = 0$$

In these equations f is a yield function, parameterized by the yield stress σ_0.

If the step is plastic the stress update is solved through fully-implicit backward Euler integration. The algorithmic tangent is then computed using an implicit function scheme. The work and energy are integrated with a trapezoid rule from the final values of stress and plastic strain.

This model does not maintain any history variables.

Parameters

Parameter	Object type	Description	Default
`elastic`	*neml::LinearElasticModel*	Temperature dependent elastic constants	No
`surface`	*neml::YieldSurface*	The yield surface	No
`ys`	*neml::Interpolate*	The yield stress as a function of T	No
`alpha`	*neml::Interpolate*	Temperature dependent instantaneous CTE	`0.0`
`tol`	`double`	Integration tolerance	`1.0e-8`
`miter`	`int`	Maximum number of integration iters	`50`
`verbose`	`bool`	Print lots of convergence info	`false`
`max_divide`	`int`	Maximum number of adaptive subdivisions	`8`

Class description

class **SmallStrainPerfectPlasticity** : public *neml*::SubstepModel_sd

Small strain, associative, perfect plasticity.

Public Functions

SmallStrainPerfectPlasticity(*ParameterSet* &params)

Parameters: elastic model, yield surface, yield stress, CTE, integration tolerance, maximum number of iterations, verbosity flag, and the maximum number of adaptive subdivisions

virtual void **populate_state**(*History* &h) const

Populate the internal variables (nothing)

virtual void **init_state**(*History* &h) const

Initialize history (nothing to do)

virtual size_t **nparams**() const

Number of nonlinear equations to solve in the integration.

virtual void **init_x**(double *const x, *TrialState* *ts)

Setup an initial guess for the nonlinear solution.

virtual void **RJ**(const double *const x, *TrialState* *ts, double *const R, double *const J)

Integration residual and jacobian equations.

virtual *TrialState* ***setup**(const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, const double *const s_n, const double *const h_n)

Setup the trial state.

virtual bool **elastic_step**(const *TrialState* *ts, const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, const double *const s_n, const double *const h_n)

Take an elastic step.

virtual void **update_internal**(const double *const x, const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n)

Interpret the x vector.

virtual void **strain_partial**(const *TrialState* *ts, const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, const double *const s_np1, const double *const s_n, const double *const h_np1, const double *const h_n, double *de)

Minus the partial derivative of the residual with respect to the strain.

virtual void **work_and_energy**(const *TrialState* *ts, const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n, double &u_np1, double u_n, double &p_np1, double p_n)

Do the work calculation.

double **ys**(double T) const

Helper to return the yield stress.

void **make_trial_state**(const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, const double *const s_n, const double *const h_n, SSPPTrialState &ts)

Setup a trial state for the solver from the input information.

Public Static Functions

static std::string **type**()

Type for the object system.

static *ParameterSet* **parameters**()

Parameters for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Setup from a *ParameterSet*.

Rate independent plasticity

Overview

This class implements rate independent plasticity described by:

The elastic trial state:

$$\begin{aligned} \varepsilon_{tr}^p &= \varepsilon_n^p \\ \sigma_{tr} &= \mathfrak{C}_{n+1} : (\varepsilon_{n+1} - \varepsilon_{tr}^p) \\ \alpha_{tr} &= \alpha_n \end{aligned}$$

The plastic correction:

$$\sigma_{n+1} = \mathfrak{C}_{n+1} : \left(\varepsilon_{n+1} - \varepsilon_{n+1}^p\right)$$

$$\varepsilon_{n+1}^p = \begin{cases} \varepsilon_{tr}^p & f\left(\sigma_{tr}, \alpha_{tr}\right) \leq 0 \\ \varepsilon_{tr}^p + \mathbf{g}\left(\sigma_{n+1}, \alpha_{n+1}, T_{n+1}\right) \Delta\gamma_{n+1} & f\left(\sigma_{tr}, \alpha_{tr}\right) > 0 \end{cases}$$

$$\alpha_{n+1} = \begin{cases} \alpha_{tr} & f\left(\sigma_{tr}, \alpha_{tr}\right) \leq 0 \\ \alpha_{tr} + \mathbf{h}\left(\sigma_{n+1}, \alpha_{n+1}, T_{n+1}\right) \Delta\gamma_{n+1} & f\left(\sigma_{tr}, \alpha_{tr}\right) > 0 \end{cases}$$

Solving for $\Delta\gamma_{n+1}$ such that

$$f\left(\sigma_{n+1}, \alpha_{n+1}\right) = 0$$

In these equations f is a yield function, $\mathbf{g}$ is a flow function, evaluated at the next state, and $\mathbf{h}$ is the rate of evolution for the history variables, evaluated at the next state. NEML integrates all three of these functions into a *Rate independent flow rule* interface.

If the step is plastic the stress update is solved through fully-implicit backward Euler integration. The algorithmic tangent is then computed using an implicit function scheme. The work and energy are integrated with a trapezoid rule from the final values of stress and plastic strain.

This model maintains a vector of history variables defined by the model's *Rate independent flow rule* interface.

At the end of the step the model (optionally) checks to ensure the step met the Kuhn-Tucker conditions

$$\begin{aligned} \Delta\gamma_{n+1} &\geq 0 \\ f\left(\sigma_{n+1}, \alpha_{n+1}\right) &\leq 0 \\ \Delta\gamma_{n+1} f\left(\sigma_{n+1}, \alpha_{n+1}\right) &= 0. \end{aligned}$$

Parameters

Parameter	Object type	Description	Default
elastic	*neml::LinearElasticModel*	Temperature dependent elastic constants	No
surface	*neml::RateIndependentFlowRule*	Flow rule interface	No
alpha	*neml::Interpolate*	Temperature dependent instantaneous CTE	0.0
tol	double	Integration tolerance	1.0e-8
miter	int	Maximum number of integration iters	50
verbose	bool	Print lots of convergence info	false
kttol	double	Tolerance on the Kuhn-Tucker conditions	1.0e-2
check_kt	bool	Flag to actually check KT	false

Class description

class **SmallStrainRateIndependentPlasticity** : public *neml*::SubstepModel_sd

Small strain, rate-independent plasticity.

Public Functions

SmallStrainRateIndependentPlasticity(*ParameterSet* &params)

Parameters: elasticity model, flow rule, CTE, solver tolerance, maximum solver iterations, verbosity flag, tolerance on the Kuhn-Tucker conditions check, and a flag on whether the KT conditions should be evaluated

virtual void **populate_state**(*History* &h) const

Populate internal variables.

virtual void **init_state**(*History* &h) const

Initialize history at time zero.

virtual *TrialState* ***setup**(const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, const double *const s_n, const double *const h_n)

Setup the trial state.

virtual bool **elastic_step**(const *TrialState* *ts, const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, const double *const s_n, const double *const h_n)

Ignore update and take an elastic step.

virtual void **update_internal**(const double *const x, const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n)

Interpret the x vector.

virtual void **strain_partial**(const *TrialState* *ts, const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, const double *const s_np1, const double *const s_n, const double *const h_np1, const double *const h_n, double *const de)

Minus the partial derivative of the residual with respect to the strain.

virtual void **work_and_energy**(const *TrialState* *ts, const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n, double &u_np1, double u_n, double &p_np1, double p_n)

Do the work calculation.

virtual size_t **nparams**() const

Number of solver parameters.

virtual void **init_x**(double *const x, *TrialState* *ts)

Setup an iteration vector in the solver.

virtual void **RJ**(const double *const x, *TrialState* *ts, double *const R, double *const J)

Solver function returning the residual and jacobian of the nonlinear system of equations integrating the model

const std::shared_ptr<const *LinearElasticModel*> **elastic**() const

Return the elastic model for subobjects.

void **make_trial_state**(const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, const double *const s_n, const double *const h_n, SSRIPTrialState &ts)

Setup a trial state.

Public Static Functions

static std::string **type**()

Type for the object system.

static *ParameterSet* **parameters**()

Parameters for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Setup from a *ParameterSet*.

General rate dependent models

Overview

This class is a generic interface for integrating rate-dependent models. It solves the nonlinear equations described by

$$\sigma_{n+1} = \sigma_n + \dot{\sigma}\left(\sigma_{n+1}, \alpha_{n+1}, \dot{\varepsilon}_{n+1}, T_{n+1}, \dot{T}_{n+1}, t_{n+1}\right)\Delta t_{n+1}$$

$$\alpha_{n+1} = \alpha_n + \dot{\alpha}\left(\sigma_{n+1}, \alpha_{n+1}, \dot{\varepsilon}_{n+1}, T_{n+1}, \dot{T}_{n+1}, t_{n+1}\right)\Delta t_{n+1}$$

In these equations $\dot{\sigma}$ is some generic stress rate law and $\dot{\alpha}$ is some generic history evolution law. These equations are defined by a GeneralFlowRule interface. The only current purpose of this general integration routine is to integrate viscoplastic material models, but it could be used for other purposes in the future.

The integrator uses fully implicit backward Euler integration for both the stress and the history. It returns the algorithmic tangent, computed using the implicit function theorem. The work and energy are integrated with a trapezoid rule from the final values of stress and inelastic strain.

This model maintains a vector of history variables defined by the model's GeneralFlowRule interface.

Parameters

Parameter	Object type	Description	Default
elastic	*neml::LinearElasticModel*	Temperature dependent elastic constants	No
surface	*neml::GeneralFlowRule*	Flow rule interface	No
alpha	*neml::Interpolate*	Temperature dependent instantaneous CTE	0.0
tol	double	Integration tolerance	1.0e-8
miter	int	Maximum number of integration iters	50
verbose	bool	Print lots of convergence info	false
max_divide	int	Max adaptive integration divides	8

Class description

class **GeneralIntegrator** : public *neml*::SubstepModel_sd

Small strain general integrator.

Public Functions

GeneralIntegrator(*ParameterSet* &params)

Parameters are an elastic model, a general flow rule, the CTE, the integration tolerance, the maximum nonlinear iterations, a verbosity flag, and the maximum number of subdivisions for adaptive integration

virtual *TrialState* ***setup**(const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, const double *const s_n, const double *const h_n)

Setup the trial state.

virtual bool **elastic_step**(const *TrialState* *ts, const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, const double *const s_n, const double *const h_n)

Take an elastic step.

virtual void **update_internal**(const double *const x, const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n)

Interpret the x vector.

virtual void **strain_partial**(const *TrialState* *ts, const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, const double *const s_np1, const double *const s_n, const double *const h_np1, const double *const h_n, double *de)

Minus the partial derivative of the residual with respect to the strain.

virtual void **work_and_energy**(const *TrialState* *ts, const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n, double &u_np1, double u_n, double &p_np1, double p_n)

Do the work calculation.

virtual void **populate_state**(*History* &hist) const

Populate internal variables.

virtual void **init_state**(*History* &hist) const

Initialize the history at time zero.

virtual size_t **nparams**() const

Number of nonlinear equations.

virtual void **init_x**(double *const x, *TrialState* *ts)

Initialize a guess for the nonlinear iterations.

virtual void **RJ**(const double *const x, *TrialState* *ts, double *const R, double *const J)

The residual and jacobian for the nonlinear solve.

void **make_trial_state**(const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, const double *const s_n, const double *const h_n, GITrialState &ts)

Initialize a trial state.

virtual void **set_elastic_model**(std::shared_ptr<*LinearElasticModel*> emodel)

Set a new elastic model.

Public Static Functions

static std::string **type**()

Type for the object system.

static *ParameterSet* **parameters**()

Parameters for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Setup from a *ParameterSet*.

Creep + plasticity

Overview

A SmallStrainCreepPlasticity model combines any of the previous types of material models with a rate dependent creep model. The model solves the simultaneous nonlinear equations

$$\sigma_{n+1}^{base}\left(\varepsilon_{n+1}^{base}, \varepsilon_n^{base}, T_{n+1}, T_n, t_{n+1}, t_n, \sigma_n^{base}, \alpha_n^{base}\right) = \sigma_{n+1}^{cr}$$
$$\varepsilon_{n+1} = \varepsilon_{n+1}^{base} + \varepsilon_{n+1}^{cr}\left(\varepsilon_n^{cr}, \sigma_{n+1}^{cr}, \Delta t_{n+1}, T_{n+1}\right).$$

Here the model with superscript *base* is the base material model and the model with superscript *cr* is the creep model (a CreepModel object). The base model can be any *NEMLModel_sd* object. The creep model add no additional history variables, it is solely a function of stress, temperature, and strain.

Unlike base material models creep models in NEML are configured to return strain as a function of stress, rather than stress as a function of strain. Because of this these equations can be combined into a single nonlinear equation

$$\varepsilon_{n+1} = \varepsilon_{n+1}^{base} + \varepsilon_{n+1}^{cr}\left(\varepsilon_n - \varepsilon_n^{base}, \sigma_{n+1}\left(\varepsilon_{n+1}^{base}, \varepsilon_n^{ep}, \boldsymbol{h}_n, \Delta t_{n+1}, T_{n+1}\right), \Delta t_{n+1}, T_{n+1}\right)$$

The implementation solves this nonlinear equation and provides the appropriate Jacobian using a matrix decomposition formula.

Parameters

Parameter	Object type	Description	Default
elastic	*neml::LinearElasticModel*	Temperature dependent elastic constants	No
plastic	*neml::NEMLModel_sd*	Base material model	No
creep	*neml::CreepModel*	Rate dependent creep model	No
alpha	*neml::Interpolate*	Temperature dependent instantaneous CTE	0.0
tol	double	Integration tolerance	1.0e-8
miter	int	Maximum number of integration iters	50
verbose	bool	Print lots of convergence info	false
sf	double	Scale factor on strain equation	1.0e6

Note: The scale factor is multiplied by a strain residual equation that may involve very small values of strain. The default value works well for values of nominal strain (i.e. in/in or mm/mm).

Class description

class **SmallStrainCreepPlasticity** : public *neml*::*NEMLModel_sd*, public *neml*::*Solvable*

Small strain, rate-independent plasticity + creep.

Public Functions

SmallStrainCreepPlasticity(*ParameterSet* &params)

Parameters are an elastic model, a base *NEMLModel_sd*, a *CreepModel*, the CTE, a solution tolerance, the maximum number of nonlinear iterations, a verbosity flag, and a scale factor to regularize the nonlinear equations.

virtual void **update_sd_actual**(const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n, double *const A_np1, double &u_np1, double u_n, double &p_np1, double p_n)

Small strain stress update.

virtual void **populate_state**(*History* &hist) const

Populate list of internal variables.

virtual void **init_state**(*History* &hist) const

Passes call for initial history to base model.

virtual size_t **nparams**() const

The number of parameters in the nonlinear equation.

virtual void **init_x**(double *const x, *TrialState* *ts)

Initialize the nonlinear solver.

virtual void **RJ**(const double *const x, *TrialState* *ts, double *const R, double *const J)

Residual equation to solve and corresponding jacobian.

void **make_trial_state**(const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, const double *const s_n, const double *const h_n, SSCPTrialState &ts)

Setup a trial state from known information.

virtual void **set_elastic_model**(std::shared_ptr<*LinearElasticModel*> emodel)

Set a new elastic model.

Public Static Functions

static std::string **type**()

Type for the object system.

static *ParameterSet* **parameters**()

Setup parameters for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

Regime switching model

Overview

This model selects a stress and history update function from a input list of *NEMLModel_sd* objects based on a normalized activation energy. This activation energy is a function of strain rate and temperature and the form of the normalized energy comes from the work of Kocks and Mecking [KM2003].

The input to the metamodel is a list of material models and a corresponding list of activation energy cutoffs $[g_1, g_2, \ldots, g_n]$. When the caller requests a stress and history update this metamodel first calculates the normalized activation energy

$$g = \frac{kT}{\mu b^3} \ln \frac{\dot{\varepsilon}_0}{\dot{\varepsilon}}.$$

Here k is the Boltzmann constant, T the current value of temperature, μ the current, temperature-dependent shear modulus, b a Burgers vector, $\dot{\varepsilon}_0$ a reference strain rate, and $\dot{\varepsilon}$ the current equivalent strain rate, computed as

$$\dot{\varepsilon} = \frac{\varepsilon_{n+1} - \varepsilon_n}{t_{n+1} - t_n}$$

with

$$\varepsilon_{n+1} = \sqrt{\frac{2}{3}\varepsilon_{n+1} : \varepsilon_{n+1}}$$

and similarly for state n.

If $g < g_1$ the metamodel selects the first model in the input list, if $g \geq g_n$ the metamodel selects the last model in the input, and for values in between the model selects model i such that $g_{i-1} < g \geq g_i$.

The metamodel dispatches calls for the history evolution, algorithmic tangent, and energy likewise.

The Kocks-Mecking metamodel requires each model in the input list to use compatible history variables. That is, all the possible base model options must use the same history variables. This means the metamodel maintains only one

copy of the history variables, which are common for all the base models. The history evolution is therefore consistent no matter which base model is selected for a particular stress update.

Warning: The KMRegimeModel metamodel does not check for consitency of history for the base models, beyond checking to make sure that each model uses the same number of history variables.

Parameters

Parameter	Object type	Description	Default
`elastic`	*neml::LinearElasticModel*	Temperature dependent elastic constants	No
`models`	`std::vector<`*neml::NEMLModel_sd*`>`	Vector of base models	No
`gs`	`std::vector<double>`	Corresponding vector of energies	No
`kboltz`	`double`	Boltzmann's constant	No
`b`	`double`	Burger's vector length	No
`eps0`	`double`	Reference strain rate	No
`alpha`	*neml::Interpolate*	Temperature dependent instantaneous CTE	`0.0`

Class description

class **KMRegimeModel** : public *neml*::*NEMLModel_sd*

Combines multiple small strain integrators based on regimes of rate-dependent behavior.

Public Functions

KMRegimeModel(*ParameterSet* &params)

Parameters are an elastic model, a vector of valid *NEMLModel_sd* objects, the transition activation energies, the Boltzmann constant in appropriate units, a Burgers vector for normalization, a reference strain rate, and the CTE.

virtual void **update_sd_actual**(const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n, double *const A_np1, double &u_np1, double u_n, double &p_np1, double p_n)

The small strain stress update.

virtual void **populate_state**(*History* &hist) const

Populate internal variables.

virtual void **init_state**(*History* &hist) const

Initialize history at time zero.

virtual void **set_elastic_model**(std::shared_ptr<*LinearElasticModel*> emodel)

Set a new elastic model.

Public Static Functions

static std::string **type**()

Type for the object system.

static *ParameterSet* **parameters**()

Parameters for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Setup from a *ParameterSet*.

Class description

class **NEMLModel_sd** : public *neml*::*NEMLModel*

Small deformation stress update.

Subclassed by *neml::KMRegimeModel*, *neml::NEMLDamagedModel_sd*, *neml::SmallStrainCreepPlasticity*, *neml::SmallStrainElasticity*, neml::SubstepModel_sd

Public Functions

NEMLModel_sd(*ParameterSet* &params)

All small strain models use small strain elasticity and CTE.

virtual void **update_sd**(const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n, double *const A_np1, double &u_np1, double u_n, double &p_np1, double p_n)

Vector interface can go here.

virtual void **update_sd_actual**(const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n, double *const A_np1, double &u_np1, double u_n, double &p_np1, double p_n) = 0

The small strain stress update interface.

virtual void **update_ld_inc**(const double *const d_np1, const double *const d_n, const double *const w_np1, const double *const w_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n, double *const A_np1, double *const B_np1, double &u_np1, double u_n, double &p_np1, double p_n)

Large strain incremental update.

virtual void **populate_static**(*History* &history) const

Setup any static state.

virtual void **init_static**(*History* &history) const

Initialize any static state.

virtual double **alpha**(double T) const

Provide the instantaneous CTE.

const std::shared_ptr<const *LinearElasticModel*> **elastic**() const

Returns the elasticity model, for sub-objects that want to use it.

virtual void **elastic_strains**(const double *const s_np1, double T_np1, const double *const h_np1, double *const e_np1) const

Return the elastic strains.

virtual void **set_elastic_model**(std::shared_ptr<*LinearElasticModel*> emodel)

Used to override the linear elastic model to match another object's.

NEMLModel_ldi

Overview

This object natively implements the *large strain incremental*. It accommodates the *small strain* by setting the deformation rate tensor equal to the small strain rate

$$\mathbf{d}_{n+1} = \varepsilon_{n+1}$$

$$\mathbf{d}_n = \varepsilon_n$$

and the vorticity to zero

$$\mathbf{w}_{n+1} = \mathbf{0}$$

$$\mathbf{w}_n = \mathbf{0}.$$

This means the skew part of the algorithmic tangent is zero

$$\mathfrak{B}_{n+1} = 0.$$

Currently, this interface is only natively used by the *crystal plasticity* material models.

Class description

class **NEMLModel_ldi** : public *neml*::*NEMLModel*

Large deformation incremental update model.

Subclassed by *neml::PolycrystalModel*, *neml::SingleCrystalModel*

Public Functions

NEMLModel_ldi(*ParameterSet* &params)

virtual void **update_sd**(const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n, double *const A_np1, double &u_np1, double u_n, double &p_np1, double p_n)

The small strain stress update interface.

virtual void **update_ld_inc**(const double *const d_np1, const double *const d_n, const double *const w_np1, const double *const w_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n, double *const A_np1, double *const B_np1, double &u_np1, double u_n, double &p_np1, double p_n) = 0

Large strain incremental update.

5.1.3 Parameters

None

5.1.4 Class description

class **NEMLModel** : public *neml*::HistoryNEMLObject

NEML material model interface definitions.

Subclassed by *neml::NEMLModel_ldi*, *neml::NEMLModel_sd*

Public Functions

NEMLModel(*ParameterSet* &params)

inline virtual **~NEMLModel**()

virtual void **save**(std::string file_name, std::string model_name)

Store model to an XML file.

virtual void **populate_hist**(*History* &history) const

Setup the history.

virtual void **init_hist**(*History* &history) const

Initialize the history.

virtual void **populate_state**(*History* &history) const = 0

Setup the actual evolving state.

virtual void **init_state**(*History* &history) const = 0

Initialize the actual evolving state.

virtual void **populate_static**(*History* &history) const

Setup any static state.

virtual void **init_static**(*History* &history) const

Initialize any static state.

virtual void **update_sd**(const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n, double *const A_np1, double &u_np1, double u_n, double &p_np1, double p_n) = 0

Raw data small strain update interface.

virtual void **update_ld_inc**(const double *const d_np1, const double *const d_n, const double *const w_np1, const double *const w_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n, double *const A_np1, double *const B_np1, double &u_np1, double u_n, double &p_np1, double p_n) = 0

Raw data large strain incremental update.

virtual double **alpha**(double T) const = 0

Instantaneous thermal expansion coefficient as a function of temperature.

virtual void **elastic_strains**(const double *const s_np1, double T_np1, const double *const h_np1, double *const e_np1) const = 0

Elastic strain for a given stress, temperature, and history state.

virtual double **get_damage**(const double *const h_np1)

Used to find the damage value from the history.

virtual bool **should_del_element**(const double *const h_np1)

Used to determine if element should be deleted.

virtual bool **is_damage_model**() const

Used to determine if this is a damage model.

size_t **nstate**() const

Number of actual internal variables.

size_t **nstatic**() const

Number of static variables.

virtual std::vector<std::string> **report_internal_variable_names**() const

Report nice names for the internal variables.

CHAPTER

SIX

ELASTICITY MODELS

6.1 Overview

Currently, NEML only has small strain linear elasticity models. Such models implement the *LinearElasticModel* interface, defined as:

$$\mathfrak{C}, \mathfrak{S}, E, \nu, \mu, K \leftarrow \mathcal{E}(T)$$

where $\mathfrak{C}$ is the stiffness tensor, $\mathfrak{S}$ is the compliance tensor, E is the Young's modulus, ν is the Poisson's ratio, μ is the shear modulus, and K is the bulk modulus. The material model integration algorithms use the stiffness and compliance directly. The interface returns the scalar elastic properties for use in calculating material properties, for example the shear modulus is used in calculating normalized activation energy for the *Regime switching model*.

6.2 Implementations

6.2.1 Isotropic linear elasticity

Overview

This interface represents an isotropic linear elastic model. In Mandel notation the stiffness tensor is

$$\begin{bmatrix} C_{1111} & C_{1122} & C_{1122} & 0 & 0 & 0 \\ C_{1122} & C_{1111} & C_{1122} & 0 & 0 & 0 \\ C_{1122} & C_{1122} & C_{1111} & 0 & 0 & 0 \\ 0 & 0 & 0 & 2C_{1212} & 0 & 0 \\ 0 & 0 & 0 & 0 & 2C_{1212} & 0 \\ 0 & 0 & 0 & 0 & 0 & 2C_{1212} \end{bmatrix}.$$

In NEML C_{1111}, C_{1122}, and C_{1212} are determined by two scalar elasticity constants, some combination of the Young's modulus E, the Poisson's ratio ν, the shear modulus μ, and the bulk modulus:math:*K*. The input to this interfaces is the temperature T. Internally the object first converts the provide constants to the bulk and shear modulus and then constructs the stiffness tensor as

$$C_{1111} = \frac{4}{3}G + K$$
$$C_{1122} = K - \frac{2}{3}G$$
$$C_{1212} = G.$$

The compliance tensor is the matrix inverse of the stiffness tensor in Mandel notation. However, the implementation uses an analytic formula based on the scalar elastic constants to improve performance. Simple formulas link the bulk and shear moduli to the other scalar elastic constants.

Parameters

The user provides two modulus values `m1` and `m2` and two strings defining which constants are being provided, `m1_type` and `m2_type`. Valid moduli types are `"shear"`, `"bulk"`, `"youngs"`, and `"poissons"`. The implementation checks to ensure the user provides valid moduli types and that they provided two unique moduli. Any combination of two scalar elastic constants fully defines the isotropic elasticity tensor

Class description

class **IsotropicLinearElasticModel** : public *neml*::*LinearElasticModel*

Isotropic shear modulus generating properties from shear and bulk models.

Public Functions

IsotropicLinearElasticModel(*ParameterSet* &params)

See detailed documentation for how to initialize with elastic constants.

virtual void **C**(double T, double *const Cv) const

Implement the stiffness tensor.

virtual void **S**(double T, double *const Sv) const

Implement the compliance tensor.

virtual double **E**(double T) const

The Young's modulus.

virtual double **nu**(double T) const

Poisson's ratio.

virtual double **K**(double T) const

The bulk modulus.

Public Static Functions

static std::string **type**()

The string type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Setup default parameters for the object system.

static *ParameterSet* **parameters**()

Initialize from a parameter set.

6.3 Class description

class **LinearElasticModel** : public *neml*::*NEMLObject*

Interface of all linear elastic models.

Subclassed by neml::CubicLinearElasticModel, *neml::IsotropicLinearElasticModel*, neml::TransverseIsotropicLinearElasticModel

Public Functions

LinearElasticModel(*ParameterSet* &params)

virtual void **C**(double T, double *const Cv) const = 0

The stiffness tensor, in Mandel notation.

virtual void **S**(double T, double *const Sv) const = 0

The compliance tensor, in Mandel notation.

SymSymR4 **C**(double T) const

The stiffness tensor in a tensor object.

SymSymR4 **S**(double T) const

The compliance tensor in a tensor object.

SymSymR4 **C**(double T, const *Orientation* &Q) const

The rotated stiffness tensor in a tensor object.

SymSymR4 **S**(double T, const *Orientation* &Q) const

The rotated compliance tensor in a tensor object.

virtual double **G**(double T) const

An effective shear modulus.

virtual double **G**(double T, const *Orientation* &Q, const *Vector* &b, const *Vector* &n) const

CHAPTER

SEVEN

RATE INDEPENDENT FLOW RULE

7.1 Overview

A rate independent flow rule provides the interface

$$f, \frac{\partial f}{\partial \sigma}, \mathbf{g}, \frac{\partial \mathbf{g}}{\partial \sigma}, \frac{\partial \mathbf{g}}{\partial \alpha}, \mathbf{h}, \frac{\partial \mathbf{h}}{\partial \sigma}, \frac{\partial \mathbf{h}}{\partial \alpha} \leftarrow \mathcal{F}\left(\sigma, \alpha, T\right).$$

f is a yield surface, $\mathbf{g}$ is a flow function defining the direction of plastic flow, and $\mathbf{h}$ is a history evolution function. This interface is used to define a *Rate independent plasticity* model.

The base interface is entirely abstract. It maintains a set of history variables set by the specific implementation.

7.2 Implementations

7.2.1 Rate independent associative flow

Overview

This interface implements an associative flow rule where both the flow function and the hardening rule are *associated* to the yield surface by the functional relations

$$\mathbf{g}\left(\sigma, \alpha, T\right) = \frac{\partial f}{\partial \sigma}\left(\sigma, \mathbf{q}\left(\alpha\right), T\right)$$
$$\mathbf{h}\left(\sigma, \alpha, T\right) = \frac{\partial f}{\partial \mathbf{q}}\left(\sigma, \mathbf{q}\left(\alpha\right), T\right)$$

These quantities have all been defined previous, except for the function $\mathbf{q}$. This function maps the "strain-like" set of history vectors to the "stress-like" set of internal variables that enter the yield surface [SH1997]. These "stress-like" internal variables are most commonly an isotropic expansion/contraction of the yield surface and a kinematic backstress shifting the yield surface in space.

A fully-associative flow rule of this type results in a model with favorable theoretical and numerical properties. For example, these models all obey Drucker's postulate [D1959] and will have symmetric algorithmic tangents.

The hardening rule and yield surface are both defined with separate interfaces.

Parameters

Parameter	Object type	Description	Default
surface	*neml::YieldSurface*	Yield surface interface	No
hardening	*neml::HardeningRule*	Hardening rule interface	No

Class description

class **RateIndependentAssociativeFlow** : public *neml*::*RateIndependentFlowRule*

Implementation of associative RI flow.

Public Functions

RateIndependentAssociativeFlow(*ParameterSet* &params)

Parameters: yield surface and hardening rule.

virtual void **populate_hist**(*History* &h) const

Setup internal state.

virtual void **init_hist**(*History* &h) const

Setup the history at time zero.

virtual void **f**(const double *const s, const double *const alpha, double T, double &fv) const

Yield surface.

virtual void **df_ds**(const double *const s, const double *const alpha, double T, double *const dfv) const

Partial derivative of the surface wrt stress.

virtual void **df_da**(const double *const s, const double *const alpha, double T, double *const dfv) const

Partial derivative of the surface wrt history.

virtual void **g**(const double *const s, const double *const alpha, double T, double *const gv) const

Flow function.

virtual void **dg_ds**(const double *const s, const double *const alpha, double T, double *const dgv) const

Partial derivative of the flow function wrt stress.

virtual void **dg_da**(const double *const s, const double *const alpha, double T, double *const dgv) const

Partial derivative of the flow function wrt history.

virtual void **h**(const double *const s, const double *const alpha, double T, double *const hv) const

Hardening rule.

virtual void **dh_ds**(const double *const s, const double *const alpha, double T, double *const dhv) const

Partial derivative of the hardening rule wrt. stress.

virtual void **dh_da**(const double *const s, const double *const alpha, double T, double *const dhv) const

Partial derivative of the hardening rule wrt. history.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Setup default parameters.

static *ParameterSet* **parameters**()

Initialize from a parameter set.

7.2.2 Rate independent nonassociative flow

Overview

This interface implements a nonassociative flow rule where only the flow function is associated to the yield surface through the relation

$$\mathbf{g}\left(\sigma,\alpha,T\right)=\frac{\partial f}{\partial\sigma}\left(\sigma,\mathbf{q}\left(\alpha\right),T\right)$$

The hardening rule is left as a generic interface providing the rate of history variable evolution proportional to the equivalent plastic strain rate, $\mathbf{h}_{\gamma}$. Note that rate independent models cannot use hardening proportional to time or temperature rate or else the model will not be rate independent!

This type of model is much more common than a fully nonassociative model where neither the flow rule or the hardening rule is associated to the yield surface. For example, classical Frederick-Armstrong hardening falls into this category [FA2007].

Parameters

Parameter	Object type	Description	Default
surface	*neml::YieldSurface*	Yield surface interface	No
hardening	*neml::NonAssociativeHardening*	Nonassociative hardening rule interface	No

Class description

class **RateIndependentNonAssociativeHardening** : public *neml*::*RateIndependentFlowRule*

Associative plastic flow but non-associative hardening.

Public Functions

RateIndependentNonAssociativeHardening(*ParameterSet* &params)

virtual void **populate_hist**(*History* &h) const

Setup internal state.

virtual void **init_hist**(*History* &h) const

Setup the history at time zero.

virtual void **f**(const double *const s, const double *const alpha, double T, double &fv) const
Yield surface.

virtual void **df_ds**(const double *const s, const double *const alpha, double T, double *const dfv) const
Partial derivative of the surface wrt stress.

virtual void **df_da**(const double *const s, const double *const alpha, double T, double *const dfv) const
Partial derivative of the surface wrt history.

virtual void **g**(const double *const s, const double *const alpha, double T, double *const gv) const
Flow function.

virtual void **dg_ds**(const double *const s, const double *const alpha, double T, double *const dgv) const
Partial derivative of the flow function wrt stress.

virtual void **dg_da**(const double *const s, const double *const alpha, double T, double *const dgv) const
Partial derivative of the flow function wrt history.

virtual void **h**(const double *const s, const double *const alpha, double T, double *const hv) const
Hardening rule.

virtual void **dh_ds**(const double *const s, const double *const alpha, double T, double *const dhv) const
Partial derivative of the hardening rule wrt. stress.

virtual void **dh_da**(const double *const s, const double *const alpha, double T, double *const dhv) const
Partial derivative of the hardening rule wrt. history.

Public Static Functions

static std::string **type**()
String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)
Setup default parameters.

static *ParameterSet* **parameters**()
Initialize from a parameter set.

7.3 Class description

class **RateIndependentFlowRule** : public *neml*::HistoryNEMLObject
ABC describing rate independent flow.

Subclassed by *neml::RateIndependentAssociativeFlow*, *neml::RateIndependentNonAssociativeHardening*

Public Functions

RateIndependentFlowRule(*ParameterSet* &params)

virtual void **f**(const double *const s, const double *const alpha, double T, double &fv) const = 0
Yield surface.

virtual void **df_ds**(const double *const s, const double *const alpha, double T, double *const dfv) const = 0
Partial derivative of the surface wrt stress.

virtual void **df_da**(const double *const s, const double *const alpha, double T, double *const dfv) const = 0
Partial derivative of the surface wrt history.

virtual void **g**(const double *const s, const double *const alpha, double T, double *const gv) const = 0
Flow function.

virtual void **dg_ds**(const double *const s, const double *const alpha, double T, double *const dgv) const = 0
Partial derivative of the flow function wrt stress.

virtual void **dg_da**(const double *const s, const double *const alpha, double T, double *const dgv) const = 0
Partial derivative of the flow function wrt history.

virtual void **h**(const double *const s, const double *const alpha, double T, double *const hv) const = 0
Hardening rule.

virtual void **dh_ds**(const double *const s, const double *const alpha, double T, double *const dhv) const = 0
Partial derivative of the hardening rule wrt. stress.

virtual void **dh_da**(const double *const s, const double *const alpha, double T, double *const dhv) const = 0
Partial derivative of the hardening rule wrt. history.

CHAPTER

EIGHT

GENERAL FLOW RULE

8.1 Overview

A general flow rule provides the interface

$$\dot{\sigma}, \frac{\partial \dot{\sigma}}{\partial \sigma}, \frac{\partial \dot{\sigma}}{\partial \alpha}, \frac{\partial \dot{\sigma}}{\partial \dot{\varepsilon}}, \dot{\alpha}, \frac{\partial \dot{\alpha}}{\partial \sigma}, \frac{\partial \dot{\alpha}}{\partial \alpha}, \frac{\partial \dot{\alpha}}{\partial \dot{\varepsilon}} \leftarrow \mathcal{G}\left(\sigma, \alpha, \dot{\varepsilon}, T, \dot{T}\right).$$

So it provides a generic stress rate and history evolution rate as a function of the current state.

The base interface is entirely abstract. It maintains a set of history variables set by the specific implementation.

8.2 Implementations

8.2.1 Viscoplastic general flow rule

Overview

This class specializes the *General flow rule* interface to match the standard definition of a generic viscoplastic flow rule. It defines the stress and hardening rates as

$$\dot{\sigma} = \mathfrak{C} : \left(\dot{\varepsilon} - \mathbf{g}_\gamma \dot{\gamma} - \mathbf{g}_T \dot{T} - \mathbf{g}_t\right)$$
$$\dot{\alpha} = \mathbf{h}_\gamma \dot{\gamma} + \mathbf{h}_T \dot{T} + \mathbf{h}_t.$$

A *Elasticity models* interface defines the stiffness tensor and a viscoplastic flow rule interface defines the scalar flow rate $\dot{\gamma}$, the flow functions $\mathbf{g}_\gamma$, $\mathbf{g}_T$, and $\mathbf{g}_t$ and the hardening functions $\mathbf{h}_\gamma$, $\mathbf{h}_T$, and $\mathbf{h}_t$.

Parameters

Parameter	Object type	Description	Default
`elastic`	*`neml::LinearElasticModel`*	Elasticity model	No
`flow`	*`neml::ViscoPlasticFlowRule`*	Viscoplastic flow rule interface	No

Class description

class **TVPFlowRule** : public *neml*::*GeneralFlowRule*

Thermo-visco-plasticity.

Public Functions

TVPFlowRule(*ParameterSet* &params)

Parameters: elastic model and a viscoplastic flow rule.

virtual void **populate_hist**(*History* &h) const

Setup the internal state.

virtual void **init_hist**(*History* &h) const

Initialize history.

virtual void **s**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const sdot)

Stress rate.

virtual void **ds_ds**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const d_sdot)

Partial of stress rate wrt stress.

virtual void **ds_da**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const d_sdot)

Partial of stress rate wrt history.

virtual void **ds_de**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const d_sdot)

Partial of stress rate wrt strain rate.

virtual void **a**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const adot)

History rate.

virtual void **da_ds**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const d_adot)

Partial of history rate wrt stress.

virtual void **da_da**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const d_adot)

Partial of history rate wrt history.

virtual void **da_de**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const d_adot)

Partial of history rate wrt strain rate.

virtual void **work_rate**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double &p_rate)

The implementation needs to define inelastic dissipation.

virtual void **elastic_strains**(const double *const s_np1, double T_np1, double *const e_np1) const

The implementation needs to define elastic strain.

virtual void **set_elastic_model**(std::shared_ptr<*LinearElasticModel*> emodel)

Set a new elastic model.

virtual void **override_guess**(double *const x)

Override the initial guess.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Return default parameters.

static *ParameterSet* **parameters**()

Initialize from parameter set.

8.2.2 Walker-Krempl rate sensitivity modification

Overview

This class modifies an existing *Viscoplastic general flow rule* to scale between a rate sensitive response (given by the base model) and a rate insensitive response based on a temperature-dependent parameter λ. The approach was developed by Walker and Krempl [WK1978] to modify an underlying viscoplastic model to return a rate insensitive response at lower temperatures.

Given the base model scalar inelastic rate

$$\dot{\varepsilon}_{vp} = \dot{p}\mathbf{g}$$

the model modifies the scalar part of the flow rate to

$$\dot{p}_{mod} = \kappa \dot{p}$$

keeping the flow direction the same. The scaling function is

$$\kappa = 1 - \lambda + \frac{\lambda\sqrt{\frac{2}{3}}\|\dot{\boldsymbol{e}}\|}{\dot{\varepsilon}_{ref}}$$

where $\dot{\varepsilon}_{ref}$ is a parameter and $\|\dot{\boldsymbol{e}}\|$ is the total (not inelastic) deviatoric strain rate.

In the limit $\lambda = 0$ the model returns the rate sensitive, viscoplastic flow response from the underlying model. In the limit $\lambda \to 1$ the model returns an equivalent rate insensitive flow rate, asymptotically approximating an equivalent rate independent model. An interpolation function can change the value of λ as a function of temperature, for example to transition from a rate insensitive response at lower temperatures to a rate sensitive response at higher temperatures.

The implementation applies the same scale factor to all time derivatives in the base model's history evolution equations, maintaining a consistent modified time integration scheme for all internal variables.

The user must determine an appropriate value of λ (which cannot be set exactly to $\lambda = 1$, as this results in numerical instability) and an appropriate reference rate $\dot{\varepsilon}_{ref}$. A value of $\lambda = 0.99$ seems to work well and the reference rate should be set to a value much slower than the actual strain rates used by the material model.

Parameters

Parameter	Object type	Description	Default
elastic	*neml::LinearElasticModel*	Elasticity model	No
flow	*neml::ViscoPlasticFlowRule*	Viscoplastic flow rule interface	No
lambda	*neml::Interpolate*	Scaling parameter	No
eps0	double	Reference strain rate	No

Class description

class **WalkerKremplSwitchRule** : public *neml*::*GeneralFlowRule*

WalkerKremplSwitchRule.

Public Functions

WalkerKremplSwitchRule(*ParameterSet* &params)

Parameters: elastic model and a viscoplastic flow rule.

virtual void **populate_hist**(*History* &hist) const

Setup internal state.

virtual void **init_hist**(*History* &hist) const

Initialize history.

virtual void **s**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const sdot)

Stress rate.

virtual void **ds_ds**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const d_sdot)

Partial of stress rate wrt stress.

virtual void **ds_da**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const d_sdot)

Partial of stress rate wrt history.

virtual void **ds_de**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const d_sdot)

Partial of stress rate wrt strain rate.

virtual void **a**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const adot)

History rate.

virtual void **da_ds**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const d_adot)

Partial of history rate wrt stress.

virtual void **da_da**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const d_adot)

Partial of history rate wrt history.

virtual void **da_de**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const d_adot)

Partial of history rate wrt strain rate.

virtual void **work_rate**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double &p_rate)

The implementation needs to define inelastic dissipation.

virtual void **elastic_strains**(const double *const s_np1, double T_np1, double *const e_np1) const

The implementation needs to define elastic strain.

virtual void **set_elastic_model**(std::shared_ptr<*LinearElasticModel*> emodel)

Set a new elastic model.

void **kappa**(const double *const edot, double T, double &kap)

The kappa function controlling rate sensitivity (public for testing)

void **dkappa**(const double *const edot, double T, double *const dkap)

Derivative of kappa wrt the strain rate (public for testing)

virtual void **override_guess**(double *const x)

Override initial guess.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Return default parameters.

static *ParameterSet* **parameters**()

Initialize from parameter set.

8.3 Class description

class **GeneralFlowRule** : public *neml*::HistoryNEMLObject

ABC for a completely general flow rule…

Subclassed by *neml::TVPFlowRule*, *neml::WalkerKremplSwitchRule*

Public Functions

GeneralFlowRule(*ParameterSet* &params)

virtual void **s**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const sdot) = 0

Stress rate.

virtual void **ds_ds**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const d_sdot) = 0

Partial of stress rate wrt stress.

virtual void **ds_da**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const d_sdot) = 0

Partial of stress rate wrt history.

virtual void **ds_de**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const d_sdot) = 0

Partial of stress rate wrt strain rate.

virtual void **a**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const adot) = 0

History rate.

virtual void **da_ds**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const d_adot) = 0

Partial of history rate wrt stress.

virtual void **da_da**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const d_adot) = 0

Partial of history rate wrt history.

virtual void **da_de**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double *const d_adot) = 0

Partial of history rate wrt strain rate.

virtual void **work_rate**(const double *const s, const double *const alpha, const double *const edot, double T, double Tdot, double &p_rate)

The implementation needs to define inelastic dissipation.

virtual void **elastic_strains**(const double *const s_np1, double T_np1, double *const e_np1) const = 0

The implementation needs to define elastic strain.

virtual void **set_elastic_model**(std::shared_ptr<*LinearElasticModel*> emodel)

Set a new elastic model.

virtual void **override_guess**(double *const x)

Optional method for modifying the initial guess.

CHAPTER

NINE

VISCOPLASTIC FLOW RULE

9.1 Overview

This class provides the interface

$$\dot{\gamma}, \mathbf{g}_\gamma, \mathbf{g}_T, \mathbf{g}_t, \mathbf{h}_\gamma, \mathbf{h}_T, \mathbf{h}_t \leftarrow \mathcal{V}(\sigma, \alpha, T) .$$

and associated partial derivatives. $\dot{\gamma}$ is the scalar inelastic strain rate, $\mathbf{g}$ is the flow rule, and $\mathbf{h}$ is the hardening law. The subscripts γ indicates the part proportional to the scalar inelastic strain rate, T the part proportional to the temperature rate, t the part directly contributing towards the total time derivative. See *Viscoplastic general flow rule* for the specific definition of each quantity.

The base class implementation by default provides zero for the T and the t quantities, giving a standard model that evolves only in proportion to the inelastic strain rate. Static recovery or thermo-viscoplasticity requires the definition of the time parts and temperature parts of the flow rule and/or hardening rule.

9.2 Implementations

9.2.1 Superimposed viscoplastic flow rule

Overview

This model sums the contribution of multiple individual viscoplastic hardening rules according to the relations:

$$\dot{\gamma} = \sum_{i=1}^{n_{models}} \dot{\gamma}_i$$

$$\mathbf{g}_{...} = \frac{1}{\dot{\gamma}} \sum_{i=1}^{n_{models}} \dot{\gamma}_i \mathbf{g}_{...,i}$$

$$\mathbf{h}_{...} = \bigoplus_{i=1}^{n_{models}} \mathbf{h}_{...,i}$$

where ... here is all of γ, t, and T and $\oplus$ represents concatenation. The internal variables is then the set of all variables maintained by each individual flow rule (and these variables cannot overlap).

Parameters

Parameter	Object type	Description	Default
`flow_rules`	`std::vector<neml::ViscoPlasticFlowRule>`	List of individual flow rules	No

Class description

class **SuperimposedViscoPlasticFlowRule** : public *neml*::*ViscoPlasticFlowRule*

Superimpose multiple flow rules into a composite model.

Public Functions

SuperimposedViscoPlasticFlowRule(*ParameterSet* &params)

size_t **nmodels**() const

Number of individual models being summed.

virtual void **populate_hist**(*History* &hist) const

Populate a blank history object.

virtual void **init_hist**(*History* &hist) const

Initialize history at time zero.

virtual void **y**(const double *const s, const double *const alpha, double T, double &yv) const

Scalar flow rate.

virtual void **dy_ds**(const double *const s, const double *const alpha, double T, double *const dyv) const

Derivative of scalar flow wrt stress.

virtual void **dy_da**(const double *const s, const double *const alpha, double T, double *const dyv) const

Derivative of scalar flow wrt history.

virtual void **g**(const double *const s, const double *const alpha, double T, double *const gv) const

Contribution towards the flow proportional to the scalar inelastic strain rate

virtual void **dg_ds**(const double *const s, const double *const alpha, double T, double *const dgv) const

Derivative of g wrt stress.

virtual void **dg_da**(const double *const s, const double *const alpha, double T, double *const dgv) const

Derivative of g wrt history.

virtual void **g_time**(const double *const s, const double *const alpha, double T, double *const gv) const

Contribution towards the flow proportional directly to time.

virtual void **dg_ds_time**(const double *const s, const double *const alpha, double T, double *const dgv) const

Derivative of g_time wrt stress.

virtual void **dg_da_time**(const double *const s, const double *const alpha, double T, double *const dgv) const

Derivative of g_time wrt history.

virtual void **g_temp**(const double *const s, const double *const alpha, double T, double *const gv) const

Contribution towards the flow proportional to the temperature rate.

virtual void **dg_ds_temp**(const double *const s, const double *const alpha, double T, double *const dgv) const
Derivative of g_temp wrt stress.

virtual void **dg_da_temp**(const double *const s, const double *const alpha, double T, double *const dgv) const
Derivative of g_temp wrt history.

virtual void **h**(const double *const s, const double *const alpha, double T, double *const hv) const
Hardening rate proportional to the scalar inelastic strain rate.

virtual void **dh_ds**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h wrt stress.

virtual void **dh_da**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h wrt history.

virtual void **h_time**(const double *const s, const double *const alpha, double T, double *const hv) const
Hardening rate proportional directly to time.

virtual void **dh_ds_time**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h_time wrt stress.

virtual void **dh_da_time**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h_time wrt history.

virtual void **h_temp**(const double *const s, const double *const alpha, double T, double *const hv) const
Hardening rate proportional to the temperature rate.

virtual void **dh_ds_temp**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h_temp wrt. stress.

virtual void **dh_da_temp**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h_temp wrt history.

Public Static Functions

static std::string **type**()
String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)
Default parameters.

static *ParameterSet* **parameters**()
Initialize from parameters.

9.2.2 Perzyna viscoplastic flow rule

Overview

The Perzyna model is an associative viscoplastic model [P1966]. It is defined by:

$$\dot{\gamma} = g\left(\langle f\left(\sigma, \mathbf{q}\left(\alpha\right), T\right)\rangle\right)$$
$$\mathbf{g}_\gamma = \frac{\partial f}{\partial \sigma}\left(\sigma, \mathbf{q}\left(\alpha\right), T\right)$$
$$\mathbf{h}_\gamma = \frac{\partial f}{\partial \mathbf{q}}\left(\sigma, \mathbf{q}\left(\alpha\right), T\right)$$

where f is a flow surface, a hardening interface provides the $\mathbf{q}$ function, and g is a *Rate function*. The notation $\langle\rangle$ indicates the Macaulay brackets. The implementation uses the default zero values of the time and temperature rate contributions and so the model evolves only as a function of the inelastic strain rate.

The model maintains the set of history variables defined by the hardening model.

Parameters

Parameter	Object type	Description	Default
surface	*neml::YieldSurface*	Flow surface interface	No
hardening	*neml::HardeningRule*	Hardening rule interface	No
g	*neml::GFlow*	Rate sensitivity function	No

Class description

class **PerzynaFlowRule** : public *neml*::*ViscoPlasticFlowRule*

Perzyna associative viscoplasticity.

Public Functions

PerzynaFlowRule(*ParameterSet* &params)

Parameters: a flow surface, a hardening rule, and the rate sensitivity function

virtual void **populate_hist**(*History* &hist) const

Populate a blank history object.

virtual void **init_hist**(*History* &hist) const

Initialize history at time zero.

virtual void **y**(const double *const s, const double *const alpha, double T, double &yv) const

Scalar strain rate.

virtual void **dy_ds**(const double *const s, const double *const alpha, double T, double *const dyv) const

Derivative of y wrt stress.

virtual void **dy_da**(const double *const s, const double *const alpha, double T, double *const dyv) const

Derivative of y wrt history.

virtual void **g**(const double *const s, const double *const alpha, double T, double *const gv) const

Flow rule proportional to the scalar strain rate.

virtual void **dg_ds**(const double *const s, const double *const alpha, double T, double *const dgv) const

Derivative of g wrt stress.

virtual void **dg_da**(const double *const s, const double *const alpha, double T, double *const dgv) const

Derivative of g wrt history.

virtual void **h**(const double *const s, const double *const alpha, double T, double *const hv) const

Hardening rule proportional to the scalar strain rate.

virtual void **dh_ds**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h wrt stress.

virtual void **dh_da**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h wrt history.

Public Static Functions

static std::string **type**()
String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)
Default parameters.

static *ParameterSet* **parameters**()
Initialize from parameters.

Rate function

These functions define the rate sensitivity of the Peryna flow model. They have the form $g\left(f,T\right)$ where f is the current value of the flow surface.

class **GFlow** : public *neml*::*NEMLObject*
The "g" function in the Perzyna model — often a power law.

Subclassed by *neml::GPowerLaw*

Public Functions

GFlow(*ParameterSet* &params)

virtual double **g**(double f, double T) const = 0
The value of g.

virtual double **dg**(double f, double T) const = 0
The derivative of g wrt to the flow surface.

Power law

This rate function implements simple power law

$$g\left(f,T\right)=\left(\frac{f}{\eta}\right)^{n}$$

for some temperature-dependent rate sensitivity exponent n and fluidity η.

Parameters

Parameter	Object type	Description	Default
n	*neml::Interpolate*	Rate sensitivity exponent	No
eta	*neml::Interpolate*	Fluidity	No

Class description

class **GPowerLaw** : public *neml*::*GFlow*

g is a power law

Public Functions

GPowerLaw(*ParameterSet* &params)

Parameter: the power law exponent.

virtual double **g**(double f, double T) const

g = (f/eta)^n

virtual double **dg**(double f, double T) const

Derivative of g wrt f.

double **n**(double T) const

Helper, just return the power law exponent.

double **eta**(double T) const

Helper, just returns the fluidity.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Default parameters.

static *ParameterSet* **parameters**()

Initialize from parameters.

9.2.3 Linear vicous flow rule

Overview

This simple model provides a linear viscous response of the type

$$\dot{\gamma} = \frac{3}{2}\frac{f\left(\sigma, \mathbf{0}, T\right)}{\eta\left(T\right)}$$

$$\mathbf{g}_{\gamma} = \frac{\partial f}{\partial \sigma}\left(\sigma, \mathbf{0}, T\right)$$

where f is a flow surface

The model does not maintain internal variables.

Parameters

Parameter	Object type	Description	Default
surface	*neml::YieldSurface*	Flow surface interface	No
eta	*neml::Interpolate*	Drag stress	No

Class description

class **LinearViscousFlow** : public *neml*::*ViscoPlasticFlowRule*

Linear viscous perfect plasticity.

Public Functions

LinearViscousFlow(*ParameterSet* &params)

Parameters: just a surface and a drag stress.

virtual void **populate_hist**(*History* &hist) const

Populate a blank history object.

virtual void **init_hist**(*History* &hist) const

Initialize history at time zero.

virtual void **y**(const double *const s, const double *const alpha, double T, double &yv) const

Scalar strain rate.

virtual void **dy_ds**(const double *const s, const double *const alpha, double T, double *const dyv) const

Derivative of y wrt stress.

virtual void **dy_da**(const double *const s, const double *const alpha, double T, double *const dyv) const

Derivative of y wrt history.

virtual void **g**(const double *const s, const double *const alpha, double T, double *const gv) const

Flow rule proportional to the scalar strain rate.

virtual void **dg_ds**(const double *const s, const double *const alpha, double T, double *const dgv) const

Derivative of g wrt stress.

virtual void **dg_da**(const double *const s, const double *const alpha, double T, double *const dgv) const

Derivative of g wrt history.

virtual void **h**(const double *const s, const double *const alpha, double T, double *const hv) const

Hardening rule proportional to the scalar strain rate.

virtual void **dh_ds**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h wrt stress.

virtual void **dh_da**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h wrt history.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Default parameters.

static *ParameterSet* **parameters**()

Initialize from parameters.

9.2.4 Chaboche viscoplastic flow rule

Overview

The Chaboche viscoplastic flow model uses an associative flow function but nonassociative hardening. Traditionally the nonassociative hardening model used is the ChabocheHardening model, but this implementation can use any nonassociative hardening rule. The model is defined by:

$$\dot{\gamma} = \dot{\gamma}_0 \sqrt{\frac{3}{2}} \left\langle \frac{f\left(\sigma, \mathbf{q}\left(\alpha\right), T\right)}{\sqrt{2/3}\eta\left(\bar{\varepsilon}_{vp}, T\right)} \right\rangle^n$$

$$\mathbf{g}_\gamma = \frac{\partial f}{\partial \sigma}\left(\sigma, \mathbf{q}\left(\alpha\right), T\right)$$

The time and temperature rate contributions to the flow function are zero. The rate sensitivity exponent is generally temperature dependent; the prefactor $\dot{\gamma}_0$ can be temperature dependent; the *Fluidity model* η can depend on both temperature and inelastic strain. The hardening model is defined by a NonassociativeHardening model. The time and temperature rate contributions can be non-zero.

The model maintains the history variables defined by the hardening model.

Parameters

Parameter	Object type	Description	Default
surface	*neml::YieldSurface*	Flow surface interface	No
hardening	neml::NonAssociativeHardeningRule	Hardening rule interface	No
fluidity	*neml::FluidityModel*	Fluidity definition	No
n	*neml::Interpolate*	Rate sensitivity exponent	No
prefactor	*neml::Interpolate*	Prefactor	1.

Class description

class **ChabocheFlowRule** : public *neml*::*ViscoPlasticFlowRule*

Non-associative flow based on *Chaboche*'s viscoplastic formulation.

Public Functions

ChabocheFlowRule(*ParameterSet* &params)

Parameters: a yield surface, a nonassociative hardening rule, the fluidity function, and a rate sensitivity exponent

virtual void **populate_hist**(*History* &hist) const

Number of history variables.

virtual void **init_hist**(*History* &hist) const

Initialize history at time zero.

virtual void **y**(const double *const s, const double *const alpha, double T, double &yv) const

Scalar flow rate.

virtual void **dy_ds**(const double *const s, const double *const alpha, double T, double *const dyv) const

Derivative of y wrt stress.

virtual void **dy_da**(const double *const s, const double *const alpha, double T, double *const dyv) const

Derivative of y wrt history.

virtual void **g**(const double *const s, const double *const alpha, double T, double *const gv) const

Flow rule proportional to the scalar strain rate.

virtual void **dg_ds**(const double *const s, const double *const alpha, double T, double *const dgv) const

Derivative of g wrt stress.

virtual void **dg_da**(const double *const s, const double *const alpha, double T, double *const dgv) const

Derivative of g wrt history.

virtual void **h**(const double *const s, const double *const alpha, double T, double *const hv) const

Hardening rule proportional to the scalar strain rate.

virtual void **dh_ds**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h wrt stress.

virtual void **dh_da**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h wrt history.

virtual void **h_time**(const double *const s, const double *const alpha, double T, double *const hv) const

Hardening rule proportional to time.

virtual void **dh_ds_time**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h_time wrt stress.

virtual void **dh_da_time**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h_time wrt history.

virtual void **h_temp**(const double *const s, const double *const alpha, double T, double *const hv) const

Hardening rule proportional to temperature rate.

virtual void **dh_ds_temp**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h_temp wrt stress.

virtual void **dh_da_temp**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h_temp wrt history.

Public Static Functions

static std::string **type**()
String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)
Initialize from a parameter set.

static *ParameterSet* **parameters**()
Return default parameters.

Fluidity model

The general Chaboche model allows the fluidity to vary with the integrated effective inelastic strain

$$\bar{\varepsilon}_{vp} = \int_0^t \sqrt{\frac{2}{3}\dot{\varepsilon}_{vp} : \dot{\varepsilon}_{vp}} dt.$$

These models then define the fluidity as

$$\eta \leftarrow \mathcal{N}\left(\bar{\varepsilon}_{vp}, T\right).$$

class **FluidityModel** : public *neml*::*NEMLObject*
Various *Chaboche* type fluidity models.

Subclassed by *neml::ConstantFluidity*, *neml::SaturatingFluidity*

Public Functions

FluidityModel(*ParameterSet* &params)

virtual double **eta**(double a, double T) const = 0
Value of viscosity as a function of temperature and inelastic strain.

virtual double **deta**(double a, double T) const = 0
Derivative of viscosity wrt inelastic strain.

Constant fluidity

This option keeps the fluidity constant with the effective inelastic strain.

$$\eta = c$$

It can still vary with temperature.

Parameters

Parameter	Object type	Description	Default
`eta`	*neml::Interpolate*	Value of the fluidity	No

Class description

class **ConstantFluidity** : public *neml*::*FluidityModel*

The fluidity is constant with respect to plastic strain.

Public Functions

ConstantFluidity(*ParameterSet* &params)

Parameter: constant value of viscosity.

virtual double **eta**(double a, double T) const

Value of eta.

virtual double **deta**(double a, double T) const

Derivative of eta wrt inelastic strain (zero for this implementation)

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize with a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

Saturating fluidity

This option evolves the fluidity from some initial value through some increment as an exponential function of inelastic strain. The fluidity eventually saturates to a final, fixed value.

$$\eta = K_0 + A\left(1 - e^{-b\bar{\varepsilon}_{vp}}\right)$$

Parameters

Parameter	Object type	Description	Default
K0	*neml::Interpolate*	Initial fluidity	No
A	*neml::Interpolate*	Saturated fluidity is K0 + A	No
b	*neml::Interpolate*	Saturation speed exponent	No

Class description

class **SaturatingFluidity** : public *neml*::*FluidityModel*

Voce-like saturating fluidity.

Public Functions

SaturatingFluidity(*ParameterSet* &params)

Parameters: K0, initial viscosity, A, saturated increase in viscosity, b, sets saturation rate

virtual double **eta**(double a, double T) const

Value of eta.

virtual double **deta**(double a, double T) const

Derivative of eta wrt inelastic strain.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize with a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

9.2.5 Yaguchi & Takahashi viscoplastic model for Grade 91 steel

Warning: This model has been depreciated. It produces inconistent results because of the time-dependent parts of the formulation. Use at your own risk!

Overview

This flow rule implements the complete Yaguchi & Takahashi model for Grade 91 steel defined in [YT2000] and [YT2005]. The model has a modified Chaboche form. This object provides a complete implementation of the flow and hardening functions. Furthermore, the model hard codes the complicated interpolation formula the original authors provide for the model coefficients. So this implementation takes no parameters, but is nevertheless valid over the range 473 K to 873 K. This is the only model in NEML where units are hard-coded into the formulation, rather than being provided by the user.

The following equations define the model:

$$\dot{\gamma} = \left\langle \frac{J_2\left(\mathrm{dev}\left(\sigma\right) - \mathrm{dev}\left(\mathbf{X}\right)\right) - \sigma_a}{D} \right\rangle^n$$

$$J_2\left(\mathbf{Y}\right) = \sqrt{\frac{3}{2}\mathbf{Y} : \mathbf{Y}}$$

$$\mathbf{g}_\gamma = \frac{3}{2}\frac{\mathrm{dev}\left(\sigma\right) - \mathrm{dev}\left(\mathbf{X}\right)}{J_2\left(\mathrm{dev}\left(\sigma\right) - \mathrm{dev}\left(\mathbf{X}\right)\right)}$$

$$\alpha = \left[\begin{array}{cccc} \mathbf{X}_1 & \mathbf{X}_2 & Q & \sigma_a \end{array} \right]$$

$$\mathbf{h}_\gamma = \left[\begin{array}{cccc} \boldsymbol{X}_{1,\gamma} & \boldsymbol{X}_{2,\gamma} & Q_\gamma & \sigma_{a,\gamma} \end{array} \right]$$

$$\mathbf{X} = \mathbf{X}_1 + \mathbf{X}_2$$

$$\mathbf{X}_{1,\gamma} = C_1\left(\frac{2}{3}\left(a_{10} - Q\right)\mathbf{n} - \mathbf{X}_1\right)\dot{\gamma} - \gamma_1 J_2\left(\mathbf{X}_1\right)^{m-1}\mathbf{X}_1$$

$$\mathbf{X}_{2,\gamma} = C_2\left(\frac{2}{3}a_2\mathbf{n} - \mathbf{X}_2\right)\dot{\gamma} - \gamma_2 J_2\left(\mathbf{X}_2\right)^{m-1}\mathbf{X}_2$$

$$Q_\gamma = d\left(q - Q\right)\sigma_{a,\gamma} = b\left(\sigma_{as} - \sigma_a\right)$$

$$b = \begin{cases} b_h & \sigma_{as} - \sigma_a \geq 0 \\ b_r & \sigma_{as} - \sigma_a < 0 \end{cases}$$

$$\sigma_{as} = \langle A + B\log_{10}\dot{p}\rangle$$

Parameters

None, all parameters are hard coded into the object.

Class description

class **YaguchiGr91FlowRule** : public *neml*::*ViscoPlasticFlowRule*

Non-associative flow for Gr. 91 from Yaguchi & Takahashi (2000) + (2005)

Public Functions

YaguchiGr91FlowRule(*ParameterSet* &params)

All parameters are hard coded to those given in the paper.

virtual void **populate_hist**(*History* &hist) const

Number of history variables.

virtual void **init_hist**(*History* &hist) const

Initialize history at time zero.

virtual void **y**(const double *const s, const double *const alpha, double T, double &yv) const

Scalar inelastic strain rate.

virtual void **dy_ds**(const double *const s, const double *const alpha, double T, double *const dyv) const

Derivative of y wrt stress.

virtual void **dy_da**(const double *const s, const double *const alpha, double T, double *const dyv) const

Derivative of y wrt history.

virtual void **g**(const double *const s, const double *const alpha, double T, double *const gv) const

Flow rule proportional to the scalar strain rate.

virtual void **dg_ds**(const double *const s, const double *const alpha, double T, double *const dgv) const

Derivative of g wrt stress.

virtual void **dg_da**(const double *const s, const double *const alpha, double T, double *const dgv) const

Derivative of g wrt history.

virtual void **h**(const double *const s, const double *const alpha, double T, double *const hv) const

Hardening rule proportional to scalar inelastic strain rate.

virtual void **dh_ds**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h wrt stress.

virtual void **dh_da**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h wrt history.

virtual void **h_time**(const double *const s, const double *const alpha, double T, double *const hv) const

Hardening rule proportional to time.

virtual void **dh_ds_time**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h_time wrt stress.

virtual void **dh_da_time**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h_time wrt history.

double **D**(double T) const
Value of parameter D.

double **n**(double T) const
Value of parameter n.

double **a10**(double T) const
Value of parameter a_10.

double **C2**(double T) const
Value of parameter C2.

double **a2**(double T) const
Value of parameter a2.

double **g1**(double T) const
Value of parameter g1.

double **g2**(double T) const
Value of parameter g2.

double **m**(double T) const
Value of parameter m.

double **br**(double T) const
Value of parameter b_r.

double **bh**(double T) const
Value of parameter b_h.

double **A**(double T) const
Value of parameter A.

double **B**(double T) const
Value of parameter B.

double **d**(double T) const
Value of parameter d.

double **q**(double T) const
Value of parameter q.

double **C1**(double T) const
Value of parameter C1.

Public Static Functions

static std::string **type**()
String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)
Initialize from a parameter set.

static *ParameterSet* **parameters**()
Default parameter set.

9.3 Class description

class **ViscoPlasticFlowRule** : public *neml*::HistoryNEMLObject

ABC describing viscoplastic flow.

Subclassed by *neml::ChabocheFlowRule*, *neml::LinearViscousFlow*, *neml::PerzynaFlowRule*, *neml::SuperimposedViscoPlasticFlowRule*, *neml::WrappedViscoPlasticFlowRule*, *neml::YaguchiGr91FlowRule*

Public Functions

ViscoPlasticFlowRule(*ParameterSet* &params)

virtual void **y**(const double *const s, const double *const alpha, double T, double &yv) const = 0
Scalar flow rate.

virtual void **dy_ds**(const double *const s, const double *const alpha, double T, double *const dyv) const = 0
Derivative of scalar flow wrt stress.

virtual void **dy_da**(const double *const s, const double *const alpha, double T, double *const dyv) const = 0
Derivative of scalar flow wrt history.

virtual void **g**(const double *const s, const double *const alpha, double T, double *const gv) const = 0
Contribution towards the flow proportional to the scalar inelastic strain rate

virtual void **dg_ds**(const double *const s, const double *const alpha, double T, double *const dgv) const = 0
Derivative of g wrt stress.

virtual void **dg_da**(const double *const s, const double *const alpha, double T, double *const dgv) const = 0
Derivative of g wrt history.

virtual void **g_time**(const double *const s, const double *const alpha, double T, double *const gv) const
Contribution towards the flow proportional directly to time.

virtual void **dg_ds_time**(const double *const s, const double *const alpha, double T, double *const dgv) const
Derivative of g_time wrt stress.

virtual void **dg_da_time**(const double *const s, const double *const alpha, double T, double *const dgv) const
Derivative of g_time wrt history.

virtual void **g_temp**(const double *const s, const double *const alpha, double T, double *const gv) const
Contribution towards the flow proportional to the temperature rate.

virtual void **dg_ds_temp**(const double *const s, const double *const alpha, double T, double *const dgv) const
Derivative of g_temp wrt stress.

virtual void **dg_da_temp**(const double *const s, const double *const alpha, double T, double *const dgv) const
Derivative of g_temp wrt history.

virtual void **h**(const double *const s, const double *const alpha, double T, double *const hv) const = 0
Hardening rate proportional to the scalar inelastic strain rate.

virtual void **dh_ds**(const double *const s, const double *const alpha, double T, double *const dhv) const = 0
Derivative of h wrt stress.

virtual void **dh_da**(const double *const s, const double *const alpha, double T, double *const dhv) const = 0
Derivative of h wrt history.

virtual void **h_time**(const double *const s, const double *const alpha, double T, double *const hv) const
Hardening rate proportional directly to time.

virtual void **dh_ds_time**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h_time wrt stress.

virtual void **dh_da_time**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h_time wrt history.

virtual void **h_temp**(const double *const s, const double *const alpha, double T, double *const hv) const
Hardening rate proportional to the temperature rate.

virtual void **dh_ds_temp**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h_temp wrt. stress.

virtual void **dh_da_temp**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h_temp wrt history.

virtual void **override_guess**(double *const guess)
Optional method to give a better initial guess.

CHAPTER

TEN

YIELD SURFACES

10.1 Overview

NEML YieldSurfaces provide the interface

$$f, \frac{\partial f}{\partial \sigma}, \frac{\partial f}{\partial \mathbf{q}}, \frac{\partial^2 f}{\partial \sigma^2}, \frac{\partial^2 f}{\partial \mathbf{q}^2}, \frac{\partial^2 f}{\partial \sigma \partial \mathbf{q}}, \frac{\partial^2 f}{\partial \mathbf{q} \partial \sigma}, \leftarrow \mathcal{F}\left(\sigma, \mathbf{q}\left(\alpha\right), T\right).$$

A hardening interface provides the map between the "strain-like" history variables α and the "stress-like" history variables $\mathbf{q}$. NEML uses this interface both as a yield surface in rate independent models and as a flow surface for rate dependent plasticity.

While not required, the existing yield surfaces including kinematic hardening expect to receive a $\mathbf{q}$ vector unrolling the isotropic hardening stress followed by a Mandel vector describing the total backstress (i.e. $\mathbf{q}$ has length $1 + 6 = 7$). For pure isotropic hardening $\mathbf{q}$ consists of only the isotropic hardening stress (i.e. $\mathbf{q}$ has length 1).

Warning: YieldSurfaces have no knowledge of the hardening model. They expect to recieve an unrolled vector of the appropriate length. The only check the implementation can do is to make sure that HardeningRule provides the correct length of the $\mathbf{q}$ vector. It currently cannot check on the order of the provided "stress-like" history variables.

For convenience NEML provides a template class that automatically converts a yield surface with isotropic and kinematic hardening to a corresponding yield surface that only has isotropic hardening. Supposing `ExampleSurfaceIsoKin` correctly implements combined isotropic and kinematic hardening then defining a class `ExampleSurfaceIso` that inherits from `IsoFunction` with template arguments `<ExampleSurfaceIsoKin, Arg A, Arg B, ...>` where the arguments are the arguments required to initialize `ExampleSurfaceIsoKin` will automatically provide the isotropic-only version of the surface.

10.2 Implementations

10.2.1 IsoKinJ2

Overview

This object implements standard J_2 plasticity. The yield function is

$$f\left(\sigma, \mathbf{q}, T\right) = J_2\left(\sigma + \mathbf{X}\right) + \sqrt{\frac{2}{3}} Q$$

$$J_2\left(\mathbf{Y}\right) = \sqrt{\frac{3}{2}\,\mathrm{dev}\left(\mathbf{Y}\right) : \mathrm{dev}\left(\mathbf{Y}\right)}.$$

It assumes a "stress-like" history vector of

$$\mathbf{q} = \begin{bmatrix} Q & \mathbf{X} \end{bmatrix}$$

where Q is the isotropic hardening stress and $\mathbf{X}$ is the backstress.

Warning: All of the NEML yield surfaces assume the opposite of the standard sign convention for isotropic and kinematic hardening. The hardening model is expected to return a negative value of the isotropic hardening stress and a negative value of the backstress.

Parameters

None.

Class description

class **IsoKinJ2** : public *neml*::*YieldSurface*

Combined isotropic/kinematic hardening with a von Mises surface.

Public Functions

IsoKinJ2(*ParameterSet* &params)

No actual parameters.

virtual size_t **nhist**() const

Expects 7 history variables [isotropic 6-Mandel-vector-backstress].

virtual void **f**(const double *const s, const double *const q, double T, double &fv) const

J2(stress + backstress) + sqrt(2/3) * isotropic.

virtual void **df_ds**(const double *const s, const double *const q, double T, double *const df) const

Gradient wrt stress.

virtual void **df_dq**(const double *const s, const double *const q, double T, double *const df) const

Gradient wrt history.

virtual void **df_dsds**(const double *const s, const double *const q, double T, double *const ddf) const

Hessian dsds.

virtual void **df_dqdq**(const double *const s, const double *const q, double T, double *const ddf) const

Hessian dqdq.

virtual void **df_dsdq**(const double *const s, const double *const q, double T, double *const ddf) const

Hessian dsdq.

virtual void **df_dqds**(const double *const s, const double *const q, double T, double *const ddf) const

Hessian dqds.

Public Static Functions

static std::string **type**()
: String type for object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)
: Initialize from a parameter set.

static *ParameterSet* **parameters**()
: Default parameters.

10.2.2 IsoJ2

Overview

This is the isotropic-only version of *IsoKinJ2*. It implements standard J_2 plasticity with the yield function of

$$f(\sigma, \mathbf{q}, T) = J_2(\sigma) + \sqrt{\frac{2}{3}}Q$$

$$J_2(\mathbf{Y}) = \sqrt{\frac{3}{2}\operatorname{dev}(\mathbf{Y}) : \operatorname{dev}(\mathbf{Y})}.$$

It assumes a "stress-like" history vector of

$$\mathbf{q} = \begin{bmatrix} Q \end{bmatrix}$$

where Q is the isotropic hardening stress.

Warning: All of the NEML yield surfaces assume the opposite of the standard sign convention for isotropic and kinematic hardening. The hardening model is expected to return a negative value of the isotropic hardening stress and a negative value of the backstress.

Parameters

None.

Class description

class **IsoJ2** : public *neml*::IsoFunction<*IsoKinJ2*>
: Just isotropic hardening with a von Mises surface.

Public Functions

inline **IsoJ2**(*ParameterSet* &params)

No actual parameters.

Public Static Functions

static std::string **type**()

String type for object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Return default parameters.

10.2.3 IsoKinJ2I1

Overview

This object implements the yield function

$$f(\sigma, \mathbf{q}, T) = J_2(\sigma + \mathbf{X}) + \sqrt{\frac{2}{3}}Q + \text{sign}(I_1(\sigma))\, h\, (I_1(\sigma))^l$$

$$J_2(\mathbf{Y}) = \sqrt{\frac{3}{2}\,\text{dev}(\mathbf{Y}) : \text{dev}(\mathbf{Y})}$$

$$I_1(\mathbf{Y}) = \text{tr}(\mathbf{Y}).$$

It assumes a "stress-like" history vector of

$$\mathbf{q} = \begin{bmatrix} Q & \mathbf{X} \end{bmatrix}$$

where Q is the isotropic hardening stress and $\mathbf{X}$ is the backstress.

Warning: All of the NEML yield surfaces assume the opposite of the standard sign convention for isotropic and kinematic hardening. The hardening model is expected to return a negative value of the isotropic hardening stress and a negative value of the backstress.

Parameters

Parameter	Object type	Description	Default
h	*neml::Interpolate*	Power law prefactor	No
l	*neml::Interpolate*	Power law exponent	No

Class description

class **IsoKinJ2I1** : public *neml*::*YieldSurface*

Combined isotropic/kinematic hardening with some mean stress contribution.

Public Functions

IsoKinJ2I1(*ParameterSet* &params)

Parameters: h prefactor, l exponent.

virtual size_t **nhist**() const

Expects 7 history variables [isotropic 6-Mandel-vector-backstress].

virtual void **f**(const double *const s, const double *const q, double T, double &fv) const

J2(stress + backstress) + isotropic + sign(mean_stress) * h * |mean_stress|^l

virtual void **df_ds**(const double *const s, const double *const q, double T, double *const df) const

Gradient wrt stress.

virtual void **df_dq**(const double *const s, const double *const q, double T, double *const df) const

Gradient wrt q.

virtual void **df_dsds**(const double *const s, const double *const q, double T, double *const ddf) const

Hessian dsds.

virtual void **df_dqdq**(const double *const s, const double *const q, double T, double *const ddf) const

Hessian dqdq.

virtual void **df_dsdq**(const double *const s, const double *const q, double T, double *const ddf) const

Hessian dsdq.

virtual void **df_dqds**(const double *const s, const double *const q, double T, double *const ddf) const

Hessian dqds.

Public Static Functions

static std::string **type**()

String type for object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from parameters.

static *ParameterSet* **parameters**()

Default parameters.

10.2.4 IsoJ2I1

Overview

This object implements the yield function

$$f(\sigma, \mathbf{q}, T) = J_2(\sigma) + \sqrt{\frac{2}{3}}Q + \text{sign}(I_1(\sigma))\, h(I_1(\sigma))^l$$

$$J_2(\mathbf{Y}) = \sqrt{\frac{3}{2}\,\text{dev}(\mathbf{Y}) : \text{dev}(\mathbf{Y})}$$

$$I_1(\mathbf{Y}) = \text{tr}(\mathbf{Y}).$$

It assumes a "stress-like" history vector of

$$\mathbf{q} = \begin{bmatrix} Q \end{bmatrix}$$

where Q is the isotropic hardening stress.

Warning: All of the NEML yield surfaces assume the opposite of the standard sign convention for isotropic and kinematic hardening. The hardening model is expected to return a negative value of the isotropic hardening stress and a negative value of the backstress.

Parameters

Parameter	Object type	Description	Default
h	*neml::Interpolate*	Power law prefactor	No
l	*neml::Interpolate*	Power law exponent	No

Class description

class **IsoJ2I1** : public *neml*::IsoFunction<*IsoKinJ2I1*>

Isotropic only version of J2I1 surface.

Public Functions

inline **IsoJ2I1**(*ParameterSet* &params)

Public Static Functions

static std::string **type**()

String type for object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from parameters.

static *ParameterSet* **parameters**()

Default parameters.

10.3 Class description

class **YieldSurface** : public *neml*::*NEMLObject*

Base class for generic yield surfaces.

Subclassed by neml::IsoFunction< IsoKinJ2 >, neml::IsoFunction< IsoKinJ2I1 >, neml::IsoFunction< BT >, *neml::IsoKinJ2*, *neml::IsoKinJ2I1*

Public Functions

YieldSurface(*ParameterSet* &params)

virtual size_t **nhist**() const = 0

Indicates how many history variables the model expects to get.

virtual void **f**(const double *const s, const double *const q, double T, double &fv) const = 0

Yield function.

virtual void **df_ds**(const double *const s, const double *const q, double T, double *const df) const = 0

Gradient wrt stress.

virtual void **df_dq**(const double *const s, const double *const q, double T, double *const df) const = 0

Gradient wrt history.

virtual void **df_dsds**(const double *const s, const double *const q, double T, double *const ddf) const = 0

Hessian dsds.

virtual void **df_dqdq**(const double *const s, const double *const q, double T, double *const ddf) const = 0

Hessian dqdq.

virtual void **df_dsdq**(const double *const s, const double *const q, double T, double *const ddf) const = 0

Hessian dsdq.

virtual void **df_dqds**(const double *const s, const double *const q, double T, double *const ddf) const = 0

Hessian dqds.

CHAPTER

ELEVEN

HARDENING MODELS

NEML uses two types of hardening models: simple models intended for use with associative plasticity and nonassociative hardening models. The simple models only need to define the map between the "strain-like" history variables and the "stress-like" internal variables that feed into the *Yield surfaces*. Nonassociative models need to define both this map and the hardening evolution rate for each "strain-like" internal variable.

The same set of classes are used to implement isotropic and kinematic hardening.

11.1 Simple hardening rules

11.1.1 Overview

These rules are intended for use with associative plasticity, where the evolution rate of each hardening variable is determined by the map between the "strain-like" history variable and the "stress-like" internal variable that feeds into the yield surface. These models implement the interface

$$\mathbf{q}, \frac{\partial \mathbf{q}}{\partial \alpha} \leftarrow \mathcal{H}\left(\alpha, T\right)$$

where α are the actual "strain-like" history variables tracked and maintained by the model and $\mathbf{q}$ are the stress-like internal variables provided to the yield surface.

11.1.2 Implementations

Isotropic hardening

Overview

This object provides the interface for all simple isotropic hardening models. These models all provide a $\mathbf{q}$ vector consisting of a single entry, Q, the isotropic hardening variable. The function maps between a single "strain-like" variable α and this scalar isotropic hardening variable. The interface is then

$$Q, \frac{\partial Q}{\partial \alpha} \leftarrow \mathcal{I}\left(\alpha, T\right).$$

Implementations

Linear isotropic hardening

Overview

This object provides simple linear isotropic hardening. The isotropic hardening variable is defined as

$$Q = -\sigma_0 - k\alpha.$$

The model requires a single history variable (α) and maps to a single hardening variable (Q).

Warning: All of the NEML yield surfaces assume the opposite of the standard sign convention for isotropic and kinematic hardening. The hardening model is expected to return a negative value of the isotropic hardening stress and a negative value of the backstress.

Parameters

Parameter	Object type	Description	Default
s0	*neml::Interpolate*	Initial yield stress	No
k	*neml::Interpolate*	Linear hardening constant	No

Class description

class **LinearIsotropicHardeningRule** : public *neml*::*IsotropicHardeningRule*

Linear, isotropic hardening.

Public Functions

LinearIsotropicHardeningRule(*ParameterSet* &params)

Parameters: initial surface size and linear coefficient.

virtual void **q**(const double *const alpha, double T, double *const qv) const

q = -s0 - K * alpha[0]

virtual void **dq_da**(const double *const alpha, double T, double *const dqv) const

Derivative of map.

double **s0**(double T) const

Getter for the yield stress.

double **K**(double T) const

Getter for the linear coefficient.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

Interpolated isotropic hardening

Overview

This object provides interpolated isotropic hardening as a function of the equivalent plastic strain. The isotropic hardening variable is defined as

$$Q = -w(\alpha)$$

for some interpolation function w.

The model requires a single history variable (α) and maps to a single hardening variable (Q).

Warning: All of the NEML yield surfaces assume the opposite of the standard sign convention for isotropic and kinematic hardening. The hardening model is expected to return a negative value of the isotropic hardening stress and a negative value of the backstress.

Parameters

Parameter	Object type	Description	Default
`flow`	*neml::Interpolate*	Interpolated flow stress	No

Class description

class **InterpolatedIsotropicHardeningRule** : public *neml*::*IsotropicHardeningRule*

Isotropic hardening with flow stress from some interpolation function.

Public Functions

InterpolatedIsotropicHardeningRule(*ParameterSet* &params)

Parameter is the interpolate to use.

virtual void **q**(const double *const alpha, double T, double *const qv) const

q = -interpolate(T)

virtual void **dq_da**(const double *const alpha, double T, double *const dqv) const

Derivative of the map.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

Voce isotropic hardening

Overview

This object provides the Voce isotropic hardening. The isotropic hardening variable is defined as

$$Q = -\sigma_0 - R\left(1 - e^{-\delta\alpha}\right).$$

The model requires a single history variable (α) and maps to a single hardening variable (Q).

Warning: All of the NEML yield surfaces assume the opposite of the standard sign convention for isotropic and kinematic hardening. The hardening model is expected to return a negative value of the isotropic hardening stress and a negative value of the backstress.

Parameters

Parameter	Object type	Description	Default
s0	*neml::Interpolate*	Initial yield stress	No
R	*neml::Interpolate*	Saturated hardening increase	No
d	*neml::Interpolate*	Saturation rate constant	No

Class description

class **VoceIsotropicHardeningRule** : public *neml*::*IsotropicHardeningRule*

Voce isotropic hardening.

Public Functions

VoceIsotropicHardeningRule(*ParameterSet* &params)

Parameters: initial yield stress, total increase amount, saturation speed constant

virtual void **q**(const double *const alpha, double T, double *const qv) const

q = -s0 - R * (1 - exp(-d * alpha[0]))

virtual void **dq_da**(const double *const alpha, double T, double *const dqv) const

Derivative of map.

double **s0**(double T) const

Getter for initial yield stress.

double **R**(double T) const

Getter for R.

double **d**(double T) const

Getter for d.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

Power law isotropic hardening

Overview

This object provides power law isotropic hardening. The isotropic hardening variable is defined as

$$Q = -\sigma_0 - A\alpha^n.$$

The model requires a single history variable (α), which is the equivalent plastic strain, and maps to a single hardening variable (Q).

Warning: All of the NEML yield surfaces assume the opposite of the standard sign convention for isotropic and kinematic hardening. The hardening model is expected to return a negative value of the isotropic hardening stress and a negative value of the backstress.

Parameters

Parameter	Object type	Description	Default
s0	*neml::Interpolate*	Initial yield stress	No
A	*neml::Interpolate*	Prefactor	No
n	*neml::Interpolate*	Work hardening exponent	No

Class description

class **PowerLawIsotropicHardeningRule** : public *neml*::*IsotropicHardeningRule*

Power law hardening.

Public Functions

PowerLawIsotropicHardeningRule(*ParameterSet* &params)

Parameters: initial yield stress, prefactor, exponent.

virtual void **q**(const double *const alpha, double T, double *const qv) const

q = -s0 - A * alpha[0]**n

virtual void **dq_da**(const double *const alpha, double T, double *const dqv) const

Derivative of map.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

Combined isotropic hardening

Overview

This object combines several isotropic hardening models into a single effective value of the isotropic hardening variable. The isotropic hardening variable is defined as

$$Q = \sum_{i=1}^{n} Q_i(\alpha).$$

That is, it calls each individual isotropic hardening function in a list and sums the results.

The model requires a single history variable (α) and maps to a single hardening variable (Q).

Warning: All of the NEML yield surfaces assume the opposite of the standard sign convention for isotropic and kinematic hardening. The hardening model is expected to return a negative value of the isotropic hardening stress and a negative value of the backstress.

Parameters

Parameter	Object type	Description	Default
`rules`	`std::vector<neml::IsotropicHardeningRule>`	List of hardening models	No

Class description

class **CombinedIsotropicHardeningRule** : public *neml*::*IsotropicHardeningRule*

Combined hardening rule superimposing a bunch of separate ones.

Public Functions

CombinedIsotropicHardeningRule(*ParameterSet* &params)

Parameter is a vector of isotropic hardening rules.

virtual void **q**(const double *const alpha, double T, double *const qv) const

q = Sum(q_i(alpha))

virtual void **dq_da**(const double *const alpha, double T, double *const dqv) const

Derivative of map.

size_t **nrules**() const

Getter on the number of combined rules.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

Class description

class **IsotropicHardeningRule** : public *neml*::*HardeningRule*

Isotropic hardening rules.

Subclassed by *neml::CombinedIsotropicHardeningRule*, *neml::InterpolatedIsotropicHardeningRule*, *neml::LinearIsotropicHardeningRule*, *neml::PowerLawIsotropicHardeningRule*, *neml::VoceIsotropicHardeningRule*

Public Functions

IsotropicHardeningRule(*ParameterSet* &params)

virtual void **populate_hist**(*History* &h) const

Setup the internal state.

virtual void **init_hist**(*History* &h) const

Initialize the history.

virtual void **q**(const double *const alpha, double T, double *const qv) const = 0

Map between the strain variables and the single isotropic hardening parameter

virtual void **dq_da**(const double *const alpha, double T, double *const dqv) const = 0

Derivative of the map.

Kinematic hardening

Overview

This object provides the interface for all simple kinematic hardening models. These models provide a $\mathbf{q}$ consisting of a single backstress $\mathbf{X}$, implemented as a length 6 Mandel vector representing a full 2nd order tensor. The function maps between this backstress and similar backstrain χ, likewise implemented with a length 6 Mandel vector representing a full 2nd order tensor.

The interface is

$$\mathbf{X}, \frac{\partial \mathbf{X}}{\partial \chi} \leftarrow \mathcal{K}\left(\chi, T\right).$$

Implementations

Linear kinematic hardening

Overview

This object implements simple linear kinematic hardening. The backstress is defined as

$$\mathbf{X} = -H\chi.$$

The model requires a length 6 history variable (χ) and maps to a length 6 backstress ($\mathbf{X}$).

Warning: All of the NEML yield surfaces assume the opposite of the standard sign convention for isotropic and kinematic hardening. The hardening model is expected to return a negative value of the isotropic hardening stress and a negative value of the backstress.

Parameters

Parameter	Object type	Description	Default
H	*neml::Interpolate*	Linear hardening constant	No

Class description

class **LinearKinematicHardeningRule** : public *neml*::*KinematicHardeningRule*

Simple linear kinematic hardening.

Public Functions

LinearKinematicHardeningRule(*ParameterSet* &params)

Parameter is the linear hardening coefficient.

virtual void **q**(const double *const alpha, double T, double *const qv) const

q = -H * alpha[:6]

virtual void **dq_da**(const double *const alpha, double T, double *const dqv) const

Derivative of the map.

double **H**(double T) const

Getter for the hardening coefficieint.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

Class description

class **KinematicHardeningRule** : public *neml*::*HardeningRule*

Base class for pure kinematic hardening.

Subclassed by *neml::LinearKinematicHardeningRule*

Public Functions

KinematicHardeningRule(*ParameterSet* &params)

virtual void **populate_hist**(*History* &h) const

Setup the internal state.

virtual void **init_hist**(*History* &h) const

Initialize the history.

virtual void **q**(const double *const alpha, double T, double *const qv) const = 0

Map between the backstrain and the backstress.

virtual void **dq_da**(const double *const alpha, double T, double *const dqv) const = 0

Derivative of the map.

Combined isotropic/kinematic hardening

Overview

This object simply combines an isotropic hardening model and a kinematic hardening model to produce a combined hardening model suitable for yield surfaces that use combined isotropic and kinematic hardening.

All it does in concatenate the isotropic hardening variable Q and the kinematic backstress $\mathbf{X}$ into the "stress-like" hardening vector

$$\mathbf{q} = \begin{bmatrix} Q & \mathbf{X} \end{bmatrix}$$

and likewise concatenates the "strain-like" history variables into a vector α.

Parameters

Parameter	Object type	Description	Default
`iso`	*neml::IsotropicHardeningRule*	Isotropic hardening rule	No
`kin`	*neml::KinematicHardeningRule*	Kinematic hardening rule	No

Class description

class **CombinedHardeningRule** : public *neml*::*HardeningRule*

Class to combine isotropic and kinematic hardening rules.

Public Functions

CombinedHardeningRule(*ParameterSet* &params)

Parameters: a isotropic hardening rule and a kinematic hardening rule.

virtual void **populate_hist**(*History* &h) const

Setup the internal state.

virtual void **init_hist**(*History* &h) const

Initialize the history.

virtual void **q**(const double *const alpha, double T, double *const qv) const

q[0] = isotropic model, q[1:7] = kinematic model

virtual void **dq_da**(const double *const alpha, double T, double *const dqv) const

Derivative of the map.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

11.1.3 Class description

class **HardeningRule** : public *neml*::HistoryNEMLObject

Interface for a generic hardening rule.

Subclassed by *neml::CombinedHardeningRule*, *neml::IsotropicHardeningRule*, *neml::KinematicHardeningRule*

Public Functions

HardeningRule(*ParameterSet* &params)

virtual void **q**(const double *const alpha, double T, double *const qv) const = 0

The map between strain-like and stress-like variables.

virtual void **dq_da**(const double *const alpha, double T, double *const dqv) const = 0

The derivative of the map.

11.2 Nonassociative hardening rules

11.2.1 Overview

This object provides the interface for a generic nonassociative hardening rule. The interface is:

$$\mathbf{q}, \mathbf{h}_\gamma, \mathbf{h}_t, \mathbf{h}_T \leftarrow \mathcal{H}(\alpha, T).$$

Here $\mathbf{q}$ is a map between the history variables α and the "stress-like" hardening variables needed for the yield stress. Usually these will be an isotropic hardening variable Q and a kinematic backstress $\mathbf{X}$. The functions $\mathbf{h}_\gamma$, $\mathbf{h}_t$, and $\mathbf{h}_T$ are the parts of the history evolution rate that are proportional to the scalar inelastic strain rate, the temperature rate, and time, respectively. For more information see *Chaboche viscoplastic flow rule* and *Rate independent nonassociative flow*.

11.2.2 Implementations

Chaboche nonassociative hardening

Overview

The Chaboche model has been developed over a long period by Chaboche and coworkers [C2008] [C1989a] [C1989b]. It is one of a few canonical high temperature constitutive models for metals that accounts for the interaction of creep and kinematic hardening.

The Chaboche model includes an extension of the combined isotropic/kinematic hardening model originally proposed by Frederick and Armstrong [FA2007]. The hardening variables are a standard isotropic hardening variable defined by a strain-like equivalent inelastic strain α mapped to a stress-like isotropic hardening variable describing the expansion of the flow surface (Q). Supplementing this associative isotropic hardening is nonassociative kinematic hardening describing the shift in the flow surface in stress space. This kinematic hardening is often called *the* Chaboche model. It consists of a total backstress summed from several individual backstress contributions.

The total history vector α is

$$\alpha = \begin{bmatrix} \alpha & \boldsymbol{X}_1 & \boldsymbol{X}_2 & \dots & \boldsymbol{X}_n \end{bmatrix}$$

where each X_i is one of n individual backstresses. The model converts this vector of history variables into the stress-like quantities needed by the yield surface with the map

$$\mathbf{q} = \begin{bmatrix} Q(\alpha) & \sum_{i=1}^{n} \mathbf{X}_i \end{bmatrix}$$

The map for the isotropic hardening is provided by another object implementing the *isotropic hardening interface*.

The history evolution equation for the isotropic part of the model is associative and proportional only to the scalar inelastic strain rate

$$\mathbf{h}_\gamma^\alpha = \sqrt{\frac{2}{3}}.$$

The evolution equations for each individual backstress are:

$$\mathbf{h}_\gamma^{X_i} = -\frac{2}{3} C_i \frac{\mathrm{dev}(\sigma - \mathbf{X})}{\|\mathrm{dev}(\sigma - \mathbf{X})\|} - \sqrt{\frac{2}{3}} \gamma_i(\alpha, T) \mathbf{X}_i$$

$$\mathbf{h}_t^{X_i} = -\sqrt{\frac{3}{2}} A_i \|\mathbf{X}_i\|^{a_i - 1} \mathbf{x}_i$$

$$\mathbf{h}_t^{x_i} = -\sqrt{\frac{2}{3}} \frac{\mathrm{d}C_i/\mathrm{d}t}{C_i} \mathbf{X}_i$$

The model parameters for each backstress are then C_i, A_i, a_i, and γ_i as a function of the equivalent inelastic strain. Various options for gamma are *described below*. The model maintains $1 + 6n$ history variables, where n is the number of backstresses.

The implementation has an option to turn on or off the part of the backstress evolution proportional to the temperature rate. Note this contribution is in any event zero for isothermal loading.

For one backstress ($n = 1$), $A_1 = 0$, $a_1 = 1$, and the non-isothermal term turned off the model degenerates to the classical Frederick-Armstrong model [FA2007].

Warning: All of the NEML yield surfaces assume the opposite of the standard sign convention for isotropic and kinematic hardening. The hardening model is expected to return a negative value of the isotropic hardening stress and a negative value of the backstress.

Parameters

Parameter	Object type	Description	Default
`iso`	*neml::IsotropicHardeningRule*	Isotropic hardening	No
`c`	`std::vector<`*neml::InterpolateValue*`>`	Values of constant C for each backstress	No
`gmodels`	`std::vector<`*neml::GammaModel*`>`	The gamma functions for each backstress	No
`A`	`std::vector<`*neml::InterpolateValue*`>`	The value of A for each backstress	No
`a`	`std::vector<`*neml::InterpolateValue*`>`	The value of a for each backstress	No
`noniso`	`bool`	Include the nonisothermal term?	`true`

The number of backstresses is set implicitly from the lengths of these vectors. The model will return an error if they have different lengths.

Class description

class **Chaboche** : public *neml*::*NonAssociativeHardening*

Chaboche model: generalized Frederick-Armstrong.

Public Functions

Chaboche(*ParameterSet* &params)

Parameters: isotropic hardening model, vector of backstress constants C, vector of gamma functions, vector of static recovery constants A, vector static recovery constants a, flag trigger the nonisothermal terms

virtual size_t **ninter**() const

1 (isotropic) + 6 (backstress) = 7

virtual void **populate_hist**(*History* &h) const

Setup the internal state.

virtual void **init_hist**(*History* &h) const

Initialize the history.

virtual void **q**(const double *const alpha, double T, double *const qv) const

Map the isotropic variable, map the backstresses.

virtual void **dq_da**(const double *const alpha, double T, double *const qv) const

Derivative of map.

virtual void **h**(const double *const s, const double *const alpha, double T, double *const hv) const

Hardening proportional to inelastic strain rate Assume associated isotropic hardening, each backstress is -2/3 * C * X/norm(X) - sqrt(2/3) gamma(ep) * X

virtual void **dh_ds**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h wrt stress.

virtual void **dh_da**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h wrt history.

virtual void **h_time**(const double *const s, const double *const alpha, double T, double *const hv) const

Hardening proportional to time Zero for the isotropic part, -A sqrt(3/2) pow(norm(X), a-1) * X for each backstress

virtual void **dh_ds_time**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h_time wrt stress.

virtual void **dh_da_time**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h_time wrt history.

virtual void **h_temp**(const double *const s, const double *const alpha, double T, double *const hv) const

Hardening proportional to the temperature rate Zero for the isotropic part Zero for each backstress if noniso = False -sqrt(2/3) * dC/dT / C * X for each backstress if noniso = True

virtual void **dh_ds_temp**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h_temp wrt stress.

virtual void **dh_da_temp**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h_temp wrt history.

int **n**() const

Getter for the number of backstresses.

std::vector<double> **c**(double T) const

Getter for the C constants.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

Gamma models

The γ parameter describes dynamic backstress recovery in the Chaboche model. This tends to send the backstress to some saturated shift of the yield surface with increasing inelastic strain. The Chaboche model allows this dynamic recovery coefficient to vary with the accumulated inelastic strain. These objects then define the dynamic recovery parameter with the interface

$$\gamma, \frac{\partial \gamma}{\partial \alpha} \leftarrow \mathcal{G}(\alpha, T)$$

Class description

class **GammaModel** : public *neml*::*NEMLObject*

Model for the gamma constant used in *Chaboche* hardening.

Subclassed by *neml::ConstantGamma*, *neml::SatGamma*

Public Functions

GammaModel(*ParameterSet* &params)

virtual double **gamma**(double ep, double T) const = 0

Gamma as a function of equivalent inelastic strain.

virtual double **dgamma**(double ep, double T) const = 0

Derivative of the gamma function wrt inelastic strain.

Constant gamma

This function returns a value of γ that is independent of inelastic strain. It still might depend on temperature. The implementation is

$$\gamma = C.$$

Parameters

Parameter	Object type	Description	Default
g	*neml::Interpolate*	Value of gamma as a function of T	No

Class description

class **ConstantGamma** : public *neml*::*GammaModel*

Gamma is constant with respect to strain.

Public Functions

ConstantGamma(*ParameterSet* &params)

Parameter is just the constant value.

virtual double **gamma**(double ep, double T) const

gamma = C

virtual double **dgamma**(double ep, double T) const

derivative of the gamma function

double **g**(double T) const

Getter for the constant value.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

Saturating gamma

This gamma function begins a given value and transitions towards a second, saturated value as a function of accumulated inelastic strain. It implements the function

$$\gamma = \gamma_s + (\gamma_0 - \gamma_s)\, e^{-\beta\alpha}.$$

Parameters

Parameter	Object type	Description	Default
g0	*neml::Interpolate*	Initial value of gamma	No
gs	*neml::Interpolate*	Final value of gamma	No
beta	*neml::Interpolate*	Controls the saturation rate	No

Class description

class **SatGamma** : public *neml*::*GammaModel*

Gamma evolves with a saturating Voce form.

Public Functions

SatGamma(*ParameterSet* &params)

Parameters are the initial value of gamma, the saturated value of gamma and the saturation speed constant

virtual double **gamma**(double ep, double T) const

gamma = gs + (g0 - gs) * exp(-beta * ep)

virtual double **dgamma**(double ep, double T) const

Derivative of the gamma function.

double **gs**(double T) const

Parameter getter.

double **g0**(double T) const

Parameter getter.

double **beta**(double T) const

Parameter getter.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

ChabocheVoceRecovery: a special variant of the Chaboche model

Overview

This model describes a variant of the *Chaboche kinematic hardening* with *Voce* isotropic hardening. The only differences compared to using the standard *Chaboche* model with Voce isotropic hardening are:

1. The Voce model is reparameterized as described below
2. The Voce model includes static recovery
3. There is a subtle difference in the definition of the static recovery on the Chaboche backstress terms.

Voce reparameterization and recovery

The Voce model for this hardening option is reparameterized so that the strength is given as

$$Q = -\sigma_0 - R$$

Warning: All of the NEML yield surfaces assume the opposite of the standard sign convention for isotropic and kinematic hardening. The hardening model is expected to return a negative value of the isotropic hardening stress and a negative value of the backstress.

$$\dot{R} = \sqrt{\frac{2}{3}}\theta_0 \left(1 - \frac{R}{R_{max}}\right) \dot{p} + r_1 \left|R_{min} - R\right|^{r_2} \text{sign}\left(R_{min} - R\right)$$

where $\dot{p}$ is the scalar plastic strain rate and the remaining undefined terms are parameters.

The first term, proportional to the scalar plastic strain rate, is identical to the standard *Voce* model in NEML, just reparameterized. The second term provides power law static recovery, which can reduce the value of the isotropic strength down to the threshold strength R_{min}.

Slight change to Chaboche static recovery

The *standard* *Chaboche kinematic hardening* in NEML uses a static recovery term of

$$\dot{\mathbf{X}}_i = -\sqrt{\frac{3}{2}} A_i \left\|\mathbf{X}_i\right\|^{a_i - 1} \mathbf{X}_i$$

This model instead uses

$$\dot{\mathbf{X}}_i = -A_i \left(\sqrt{\frac{3}{2}} \left\|\mathbf{X}_i\right\|\right)^{a_i - 1} \mathbf{X}_i$$

The only change is the location of the $\sqrt{\frac{3}{2}}$ term. This change makes the current model directly compatible with a 1D formulation.

Parameters

Parameter	Object type	Description	Default
`s0`	*neml::Interpolate*	Threshold stress	No
`theta0`	*neml::Interpolate*	Initial isotropic hardening slope	No
`Rmax`	*neml::Interpolate*	Maximum (saturated) isotropic hardening strength	No
`Rmin`	*neml::Interpolate*	Minimum (threshold) isotropic strength for static recovery	No
`r1`	*neml::Interpolate*	Prefactor for isotropic recovery	No
`r2`	*neml::Interpolate*	Exponent for isotropic recovery	No
`c`	`std::vector<`*neml::Interpolate*`>`	Values of constant C for each backstress	No
`gmodels`	`std::vector<`*neml::GammaModel*`>`	The gamma functions for each backstress	No
`A`	`std::vector<`*neml::Interpolate*`>`	The value of A for each backstress	No
`a`	`std::vector<`*neml::Interpolate*`>`	The value of a for each backstress	No
`noniso`	`bool`	Include the nonisothermal term?	`true`

The number of backstresses is set implicitly from the lengths of these vectors. The model will return an error if they have different lengths.

Class description

class **ChabocheVoceRecovery** : public *neml*::*NonAssociativeHardening*

Chaboche model with hard-coded, recoverable Voce isotropic hardening.

Public Functions

ChabocheVoceRecovery(*ParameterSet* &params)

Parameters: isotropic hardening model, vector of backstress constants C, vector of gamma functions, vector of static recovery constants A, vector static recovery constants a, flag trigger the nonisothermal terms

virtual size_t **ninter**() const

1 (isotropic) + 6 (backstress) = 7

virtual void **populate_hist**(*History* &h) const

Setup the internal state.

virtual void **init_hist**(*History* &h) const

Initialize the history.

virtual void **q**(const double *const alpha, double T, double *const qv) const

Map the isotropic variable, map the backstresses.

virtual void **dq_da**(const double *const alpha, double T, double *const qv) const

Derivative of map.

virtual void **h**(const double *const s, const double *const alpha, double T, double *const hv) const

Hardening proportional to inelastic strain rate Assume associated isotropic hardening, each backstress is -2/3 * C * X/norm(X) - sqrt(2/3) gamma(ep) * X

virtual void **dh_ds**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h wrt stress.

virtual void **dh_da**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h wrt history.

virtual void **h_time**(const double *const s, const double *const alpha, double T, double *const hv) const

Hardening proportional to time.

virtual void **dh_ds_time**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h_time wrt stress.

virtual void **dh_da_time**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h_time wrt history.

virtual void **h_temp**(const double *const s, const double *const alpha, double T, double *const hv) const

Hardening proportional to the temperature rate.

virtual void **dh_ds_temp**(const double *const s, const double *const alpha, double T, double *const dhv) const

Derivative of h_temp wrt stress.

virtual void **dh_da_temp**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h_temp wrt history.

int **n**() const
Getter for the number of backstresses.

Public Static Functions

static std::string **type**()
String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)
Initialize from a parameter set.

static *ParameterSet* **parameters**()
Default parameters.

11.2.3 Class description

class **NonAssociativeHardening** : public *neml*::HistoryNEMLObject
ABC of a non-associative hardening rule.

Subclassed by *neml::Chaboche*, *neml::ChabocheVoceRecovery*

Public Functions

NonAssociativeHardening(*ParameterSet* &params)

virtual size_t **ninter**() const = 0
How many stress-like variables.

virtual void **q**(const double *const alpha, double T, double *const qv) const = 0
Map from strain to stress.

virtual void **dq_da**(const double *const alpha, double T, double *const qv) const = 0
Derivative of the map.

virtual void **h**(const double *const s, const double *const alpha, double T, double *const hv) const = 0
Hardening proportional to the equivalent inelastic strain.

virtual void **dh_ds**(const double *const s, const double *const alpha, double T, double *const dhv) const = 0
Derivative of h wrt stress.

virtual void **dh_da**(const double *const s, const double *const alpha, double T, double *const dhv) const = 0
Derivative of h wrt history.

virtual void **h_time**(const double *const s, const double *const alpha, double T, double *const hv) const
Hardening proportional to time.

virtual void **dh_ds_time**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h_time wrt stress.

virtual void **dh_da_time**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h_time wrt history.

virtual void **h_temp**(const double *const s, const double *const alpha, double T, double *const hv) const
Hardening proportional to the temperature rate.

virtual void **dh_ds_temp**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h_temp wrt stress.

virtual void **dh_da_temp**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h_temp wrt history.

CHAPTER

TWELVE

CREEP MODELS

12.1 Overview

Creep models in NEML relate stress and strain to a creep strain rate. This means they do not maintain a set of internal history variables. Currently, they are only used in conduction with a standard NEML material model through the *creep+plasticity* metamodel.

NEML creep models fulfill the interface

$$\dot{\varepsilon}^{cr} \leftarrow \mathcal{C}\left(\sigma, \varepsilon, t, T\right).$$

So the creep strain rate can depend on the stress, the total strain, time, and temperature.

12.2 Implementations

12.2.1 J2 creep

Overview

This interface specializes a generic creep rate model to J_2 flow. The model creeps in the direction of the deviatoric stress and the scalar creep rate is defined by an *additional interface*. The creep rate for these types of models is then

$$\dot{\varepsilon}^{cr} = \dot{\varepsilon}^{cr}\left(\sigma_{eff}, \varepsilon_{eff}, t, T\right) \frac{\mathrm{dev}\left(\sigma\right)}{\|\mathrm{dev}\left(\sigma\right)\|}$$

with the scalar creep rate a function of effective stress

$$\sigma_{eff} = \sqrt{\frac{3}{2}\sigma : \sigma},$$

effective strain

$$\varepsilon_{eff} = \sqrt{\frac{2}{3}\varepsilon : \varepsilon},$$

time, and temperature.

Scalar creep models

Scalar creep models

Overview

Scalar creep models provide the interface

$$\dot{\varepsilon}^{cr} \leftarrow \mathcal{C}\left(\sigma_{eq}, \varepsilon_{eq}, t, T\right).$$

They return a scalar creep rate as a function of effective stress, effective strain, time, and temperature.

Implementations

Power law creep

Overview

This object implements power law creep

$$\dot{\varepsilon}^{cr} = A\sigma_{eq}^{n}$$

for temperature dependent parameters A and n.

Parameters

Parameter	Object type	Description	Default
A	*neml::Interpolate*	Prefactor	No
n	*neml::Interpolate*	Exponent	No

Class description

class **PowerLawCreep** : public *neml*::*ScalarCreepRule*

Simple power law creep.

Public Functions

PowerLawCreep(*ParameterSet* &params)

Parameters: prefector A and exponent n.

virtual void **g**(double seq, double eeq, double t, double T, double &g) const

rate = A * seq**n

virtual void **dg_ds**(double seq, double eeq, double t, double T, double &dg) const

Derivative of rate wrt effective stress.

virtual void **dg_de**(double seq, double eeq, double t, double T, double &dg) const

Derivative of rate wrt effective strain = 0.

double **A**(double T) const

Getter for the prefactor.

double **n**(double T) const

Getter for the exponent.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Setup from a parameter set.

static *ParameterSet* **parameters**()

Return default parameters.

Power law creep

Overview

This object implements a better-normalized version of power law creep

$$\dot{\varepsilon}^{cr} = \left(\frac{\sigma_{eq}}{\sigma_0} \right)^n$$

for temperature dependent parameters σ_0 and n.

Parameters

Parameter	Object type	Description	Default
s0	*neml::Interpolate*	Normalization stress	No
n	*neml::Interpolate*	Exponent	No

Class description

class **NormalizedPowerLawCreep** : public *neml*::*ScalarCreepRule*

Power law creep normalized in a slightly nicer way.

Public Functions

NormalizedPowerLawCreep(*ParameterSet* &params)

Parameters: stress divisor s0 and exponent n.

virtual void **g**(double seq, double eeq, double t, double T, double &g) const

rate = (seq/s0)**n

virtual void **dg_ds**(double seq, double eeq, double t, double T, double &dg) const

Derivative of rate wrt effective stress.

virtual void **dg_de**(double seq, double eeq, double t, double T, double &dg) const

Derivative of rate wrt effective strain = 0.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Setup from a parameter set.

static *ParameterSet* **parameters**()

Return default parameters.

Regime switching Kocks-Mecking creep

Overview

This model implements a creep law based on the Kocks-Mecking normalized activation energy [KM2003]. The basic creep rate law is

$$F = -\frac{\mu b^3}{kT}$$

$$\dot{\varepsilon}^{cr} = \dot{\varepsilon}_0 \exp\left(BF\right) \left(\frac{\sigma_{eq}}{\mu}\right)^{AF}.$$

Here $\dot{\varepsilon}_0$, A, B are parameters, μ is the temperature dependent shear modulus, k is the Boltzmann constant, and b is a Burgers vector length. The model constants can be fit using a Kocks-Mecking diagram.

The formulation can switch between different Kocks-Mecking models as a function of normalized stress $s = \frac{\sigma_{eq}}{\mu}$. The model first computes the normalized stress. It then consults a table of n normalized stress cutoffs, c_i. If $s \leq c_1$ then the model applies the strain rate equation above using constants A_1 and B_1. If $c_i < s \leq c_{i+1}$ then the model uses constants A_{i+1} and B_{i+1}. If $s > c_n$ then the model uses A_n and B_n.

Parameters

Parameter	Object type	Description	Default
cuts	std::vector<double>	Normalized stress cutoffs	No
A	std::vector<*neml::InterpolaCon*	Corresponding A constants	No
B	std::vector<*neml::InterpolaCon*	Corresponding B constants	No
kboltz	double	Boltzmann constant	No
b	double	Burgers vector	No
eps0	double	Reference strain rate	No
emodel	*neml::LinearElasticModel*	Elastic model (for shear modulus)	No

Class description

class **RegionKMCreep** : public *neml*::*ScalarCreepRule*

A power law type model that uses KM concepts to switch between mechanisms.

Public Functions

RegionKMCreep(*ParameterSet* &params)

Inputs: cuts in normalized activation energy, prefactors for each region exponents for each region, boltzmann constant, burgers vector, reference strain rate, elastic model, to compute mu

virtual void **g**(double seq, double eeq, double t, double T, double &g) const

See documentation for details of the creep rate.

virtual void **dg_ds**(double seq, double eeq, double t, double T, double &dg) const

Derivative of creep rate wrt effective stress.

virtual void **dg_de**(double seq, double eeq, double t, double T, double &dg) const

Derivative of creep rate wrt effective strain.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Setup from a parameter set.

static *ParameterSet* **parameters**()

Return the default parameters.

Norton-Bailey creep

Overview

This implements the Norton-Bailey creep model

$$\varepsilon^{cr} = A\sigma_{eff}^{n} t^{m}$$

implemented in the strain-hardening formulation

$$\dot{\varepsilon}^{cr} = mA^{\frac{1}{m}} \sigma^{\frac{n}{m}} \varepsilon_{eff}^{\frac{m-1}{m}} .$$

Parameters

Parameter	Object type	Description	Default
A	*neml::Interpolate*	Prefactor	No
m	*neml::Interpolate*	Stress exponent	No
n	*neml::Interpolate*	Time exponent	No

Class description

class **NortonBaileyCreep** : public *neml*::*ScalarCreepRule*

Classical Norton-Bailey creep.

Public Functions

NortonBaileyCreep(*ParameterSet* &params)

Parameters: prefactor A, stress exponent n, time exponent m.

virtual void **g**(double seq, double eeq, double t, double T, double &g) const

creep rate = m * A**(1/m) * seq**(n/m) * eeq ** ((m-1)/m)

virtual void **dg_ds**(double seq, double eeq, double t, double T, double &dg) const

Derivative of creep rate wrt effective stress.

virtual void **dg_de**(double seq, double eeq, double t, double T, double &dg) const

Derivative of creep rate wrt effective strain.

double **A**(double T) const

Getter for the prefactor.

double **m**(double T) const

Getter for the stress exponent.

double **n**(double T) const

Getter for the time exponent.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Return default parameters.

Mukherjee creep

Overview

This implements the Mukherjee creep model [BMD1969]

$$\dot{\varepsilon}^{cr} = AD_0 e^{\frac{Q}{RT}} \frac{\mu b}{kT} \left(\frac{\sigma_{eq}}{\mu} \right)^n$$

with A, D_0, Q, and n parameters and R the gas constant, T absolute temperature, μ the temperature dependent shear modulus, b a Burgers vector length, and k the Boltzmann constant.

Parameters

Parameter	Object type	Description	Default
`emodel`	*`neml::LinearElasticModel`*	Elasticity model (for shear modulus)	No
`A`	`double`	Prefactor	No
`n`	`double`	Stress exponent	No
`D0`	`double`	Zero temperature lattice diffusivity	No
`Q`	`double`	Activation energy for diffusivity	No
`b`	`double`	Burgers vector	No
`k`	`double`	Boltzmann constant	No
`R`	`double`	Gas constant	No

Class description

class **MukherjeeCreep** : public *neml*::*ScalarCreepRule*

Classical Mukherjee creep.

Public Functions

MukherjeeCreep(*ParameterSet* &params)

Parameters: elastic model (for shear modulus), prefactor, stress exponent, reference lattice diffusivity, activation energy for lattice diffusion, burgers vector, boltzmann constant, gas constant

virtual void **g**(double seq, double eeq, double t, double T, double &g) const

scalar creep rate = A * D0 * exp(Q / (RT)) * mu * b / (k * T) * (seq / mu)**n

virtual void **dg_ds**(double seq, double eeq, double t, double T, double &dg) const

Derivative of creep rate wrt effective stress.

virtual void **dg_de**(double seq, double eeq, double t, double T, double &dg) const

Derivative of creep rate wrt effective strain.

double **A**() const

Getter for A.

double **n**() const

Getter for parameter n.

double **D0**() const

Getter for parameter D0.

double **Q**() const

Getter for parameter Q.

double **b**() const

Getter for parameter b.

double **k**() const

Getter for parameter k.

double **R**() const

Getter for parameter R.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Setup from a parameter set.

static *ParameterSet* **parameters**()

Return default parameters.

Generic creep law

Overview

This object implements the generic creep law

$$\dot{\varepsilon}^{cr} = \exp\left(f\left(\log \sigma_{eq}\right)\right)$$

where f is a generic function implemented as an *interpolate* object.

Parameters

Parameter	Object type	Description	Default
`cfn`	*`neml::Interpolate`*	Creep rate function	No

Class Description

class **`GenericCreep`** : public *neml*::*ScalarCreepRule*

A generic creep rate model where log(creep_rate) = Interpolate(log(stress))

Public Functions

`GenericCreep`(*ParameterSet* &params)

Parameters: interpolate giving the log creep rate.

virtual void **`g`**(double seq, double eeq, double t, double T, double &g) const

scalar creep rate = exp(f(log(sigma)))

virtual void **`dg_ds`**(double seq, double eeq, double t, double T, double &dg) const

Derivative of creep rate wrt effective stress.

virtual void **`dg_de`**(double seq, double eeq, double t, double T, double &dg) const

Derivative of creep rate wrt effective strain.

Public Static Functions

static std::string **`type`**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **`initialize`**(*ParameterSet* &params)

Setup from a parameter set.

static *ParameterSet* **`parameters`**()

Return default parameters.

Blackburn minimum creep rate model

Overview

This object implements the minimum creep rate model given in [B1972]:

$$\dot{\varepsilon}^{cr} = A \sinh \frac{\beta \sigma_{eq}}{n}^{n} \exp \frac{-Q}{RT}$$

for temperature dependent parameters A, β, and β and constants R and Q.

Parameters

Parameter	Object type	Description	Default
A	*neml::Interpolate*	Prefactor	No
n	*neml::Interpolate*	Exponent	No
beta	*neml::Interpolate*	Exponential factor	No
R	double	Gas constant	No
Q	double	Activation energy	No

Class description

class **BlackburnMinimumCreep** : public *neml*::*ScalarCreepRule*

Creep rate law from Blackburn 1972.

Public Functions

BlackburnMinimumCreep(*ParameterSet* &params)

virtual void **g**(double seq, double eeq, double t, double T, double &g) const

rate = A * (sinh(beta*s/n)^n * exp(-Q/(R*T))

virtual void **dg_ds**(double seq, double eeq, double t, double T, double &dg) const

Derivative of rate wrt effective stress.

virtual void **dg_de**(double seq, double eeq, double t, double T, double &dg) const

Derivative of rate wrt effective strain = 0.

virtual void **dg_dT**(double seq, double eeq, double t, double T, double &dg) const

Derivative of rate wrt temperature.

Public Static Functions

static std::string **type**()
String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)
Setup from a parameter set.

static *ParameterSet* **parameters**()
Return default parameters.

Swindeman minimum creep rate model

Overview

This object implements the minimum creep rate model given in [S1999]:

$$\dot{\varepsilon}^{cr} = C\sigma_{eq}^{n} \exp V\sigma_{eq} \exp \frac{-Q}{T}$$

for constants C, n, V, and Q.

Parameters

Parameter	Object type	Description	Default
C	double	Prefactor	No
n	double	Exponent	No
V	double	Exponential factor	No
Q	double	Activation energy	No

Class description

class **SwindemanMinimumCreep** : public *neml*::*ScalarCreepRule*
Creep rate law from Swindeman 1999.

Public Functions

SwindemanMinimumCreep(*ParameterSet* &params)

virtual void **g**(double seq, double eeq, double t, double T, double &g) const
rate = C * S^n * exp(V*S) * exp(-Q/T)

virtual void **dg_ds**(double seq, double eeq, double t, double T, double &dg) const
Derivative of rate wrt effective stress.

virtual void **dg_de**(double seq, double eeq, double t, double T, double &dg) const
Derivative of rate wrt effective strain = 0.

virtual void **dg_dT**(double seq, double eeq, double t, double T, double &dg) const
Derivative of rate wrt temperature.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Setup from a parameter set.

static *ParameterSet* **parameters**()

Return default parameters.

Minimum creep law for 2.25Cr-1Mo steel

Overview

This is a quite complicated minimum creep rate law for 2.25Cr-1Mo (Gr 22) steel, to be documented in a PVP paper at some point in the future.

The creep rate law it implements is:

if $\sigma_{eq} \leq 60$:

$$\dot{\varepsilon}^{cr} = \dot{\varepsilon}_2$$

else if $T \leq 13.571\sigma_{eq}^{0.68127} - 1.8\sigma_{eq} + 437.63$

$$\dot{\varepsilon}^{cr} = \dot{\varepsilon}_1$$

else

$$\dot{\varepsilon}^{cr} = \dot{\varepsilon}_2$$

with

$$\dot{\varepsilon}_1 = \frac{10^{6.7475+0.011426\sigma_{eq}+\frac{987.72}{U}\log\sigma_{eq}-\frac{13494}{T}}}{100}$$

$$\dot{\varepsilon}_2 = \frac{10^{11.498-\frac{8.2226U}{T}-\frac{20448}{T}+\frac{5862.4}{T}\log\sigma_{eq}}}{100}$$

and U a parameter interpolated linearly as a function of temperature from the table:

Temperature	Value
644.15	471
673.15	468
723.15	452
773.15	418
823.15	634
873.15	284
894.15	300
922.15	270

Because the model is fully parameterized in the C++ implementation it must be used with units of MPa, hours, and Kelvin.

Parameters

None, all built into class.

Class description

class **MinCreep225Cr1MoCreep** : public *neml*::*ScalarCreepRule*

The hopelessly complicated 2.25Cr minimum creep rate model.

Public Functions

MinCreep225Cr1MoCreep(*ParameterSet* &params)

Parameters: prefector A and exponent n.

virtual void **g**(double seq, double eeq, double t, double T, double &g) const

See documentation for the hideous formula.

virtual void **dg_ds**(double seq, double eeq, double t, double T, double &dg) const

Derivative of rate wrt effective stress.

virtual void **dg_de**(double seq, double eeq, double t, double T, double &dg) const

Derivative of rate wrt effective strain = 0.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Setup from a parameter set.

static *ParameterSet* **parameters**()

Return default parameters.

Class description

class **ScalarCreepRule** : public *neml*::*NEMLObject*

Scalar creep functions in terms of effective stress and strain.

Subclassed by *neml::BlackburnMinimumCreep*, *neml::GenericCreep*, *neml::MinCreep225Cr1MoCreep*, *neml::MukherjeeCreep*, *neml::NormalizedPowerLawCreep*, *neml::NortonBaileyCreep*, *neml::PowerLawCreep*, *neml::RegionKMCreep*, *neml::SwindemanMinimumCreep*

Class description

class **J2CreepModel** : public *neml*::*CreepModel*

J2 creep based on a scalar creep rule.

Public Functions

J2CreepModel(*ParameterSet* &params)

Parameters: scalar creep rule, nonlinear tolerance, maximum solver iterations, and a verbosity flag

virtual void **f**(const double *const s, const double *const e, double t, double T, double *const f) const

creep_rate = dev(s) / ||dev(s)|| * scalar(effective_strain, effective_stress, time, temperature)

virtual void **df_ds**(const double *const s, const double *const e, double t, double T, double *const df) const

Derivative of creep rate wrt stress.

virtual void **df_de**(const double *const s, const double *const e, double t, double T, double *const df) const

Derivative of creep rate wrt strain.

virtual void **df_dt**(const double *const s, const double *const e, double t, double T, double *const df) const

Derivative of creep rate wrt time.

virtual void **df_dT**(const double *const s, const double *const e, double t, double T, double *const df) const

Derivative of creep rate wrt temperature.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize an object from a parameter set.

static *ParameterSet* **parameters**()

Return the default parameters.

12.3 Class description

class **CreepModel** : public *neml*::*NEMLObject*, public *neml*::*Solvable*

Interface to creep models.

Subclassed by *neml::J2CreepModel*

Public Functions

CreepModel(*ParameterSet* &params)

Parameters are a solver tolerance, the maximum allowable iterations, and a verbosity flag

void **update**(const double *const s_np1, double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, double *const A_np1)

Use the creep rate function to update the creep strain.

virtual void **f**(const double *const s, const double *const e, double t, double T, double *const f) const = 0

The creep rate as a function of stress, strain, time, and temperature.

virtual void **df_ds**(const double *const s, const double *const e, double t, double T, double *const df) const = 0

The derivative of the creep rate wrt stress.

virtual void **df_de**(const double *const s, const double *const e, double t, double T, double *const df) const = 0

The derivative of the creep rate wrt strain.

virtual void **df_dt**(const double *const s, const double *const e, double t, double T, double *const df) const

The derivative of the creep rate wrt time, defaults to zero.

virtual void **df_dT**(const double *const s, const double *const e, double t, double T, double *const df) const

The derivative of the creep rate wrt temperature, defaults to zero.

void **make_trial_state**(const double *const s_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, CreepModelTrialState &ts) const

Setup a trial state for the solver.

virtual size_t **nparams**() const

Number of solver parameters.

virtual void **init_x**(double *const x, *TrialState* *ts)

Setup the initial guess for the solver.

virtual void **RJ**(const double *const x, *TrialState* *ts, double *const R, double *const J)

The nonlinear residual and jacobian to solve.

CHAPTER

THIRTEEN

DAMAGE MECHANICS

13.1 Overview

NEML implements damage mechanics with metamodel place on top of an existing (small strain) base material model. The metamodel supplements the base material history variables with a vector of damage variables ω. In the most generic case, these damage variables somehow modify the base material's stress update. The damage model must also provide the history evolution equations describing the evolution of damage. The interface is then

$$\sigma_{n+1}, \alpha_{n+1}, \omega_{n+1}, \mathfrak{A}_{n+1}, u_{n+1}, p_{n+1} \leftarrow \mathcal{D}\left(\varepsilon_{n+1}, \varepsilon_n, T_{n+1}, T_n, t_{n+1}, t_n, \sigma_n, \alpha_n, \alpha_n, u_n, p_n\right)$$

which is identical to the interface for the *base small strain material models*, except now with the addition of some dependence on the damage variables ω.

13.2 Implementations

The generic interface described here could have any number of damage variables and could modify the stress with those variables in any way. The more standard damage model, depending on a single scalar damage variable is implemented as a `neml::NEMLScalarDamagedModel_sd`.

13.2.1 Model damaged by a scalar variable

Overview

This object implements a damage model that uses a single damage variable to degrade the stress of a base material model. It implements the stress update function

$$\sigma'_{n+1} = (1 - \omega_{n+1})\sigma\left(\varepsilon_{n+1}, \varepsilon_n, T_{n+1}, T_n, t_{n+1}, t_n, \sigma_n/(1-\omega_n), \alpha_{n+1}, \alpha_n, u_n, p_n\right)$$

where ω is the damage variable and σ is the base material stress update. A `neml::ScalarDamage` model provides the definition of ω as well as the associated derivatives to form the Jacobian.

The damage model maintains the set of history variables from the base material plus one additional history variable for the damage.

This class has the option for element extinction, useful in FEA simulations of damage. If the `ekill` option is set to true once the material point reaches a damage threshold of `dkill` the constitutive response will be replaced by a linear elastic response with an elastic stiffness of $\mathfrak{C}/f$ where the factor f is given by the parameter `sfact`.

Note: The scalar damage model passes in the modified stress $\sigma/(1-\omega)$ to the base stress update model in addition to modifying the stress update formula as shown in the above equation.

Warning: The model also passes the modified stress $\sigma/(1-\omega)$ to the damage update equation. That is, the stress passed into these functions is the modified effective stress, not the actual external stress. This means that the damage equations implemented in NEML vary slightly from the correpsonding literature equations working with the unmodified stress directly.

Parameters

Parameter	Object type	Description	Default
`elastic`	*neml::LinearElasticModel*	Elasticity model	No
`base`	*neml::NEMLModel_sd*	Base material model	No
`damage`	*neml::ScalarDamage*	Damage model	No
`alpha`	*neml::Interpolate*	Thermal expansion coefficient	`0.0`
`tol`	`double`	Solver tolerance	`1.0e-8`
`miter`	`int`	Maximum solver iterations	`50`
`verbose`	`bool`	Verbosity flag	`false`
`ekill`	`bool`	Trigger element death	`false`
`dkill`	`double`	Critical damage threshold	`0.5`
`sfact`	`double`	Stiffness factor for dead element	`100000`

Scalar damage models

Scalar damage models

Overview

This object defines a scalar damage model with the interface

$$\omega_{n+1}, \frac{\partial \omega_{n+1}}{\partial \omega}, \frac{\partial \omega_{n+1}}{\partial \sigma}, \frac{\partial \omega_{n+1}}{\partial \varepsilon} \leftarrow \mathcal{W}\left(\omega_{n+1}, \omega n, \sigma_{n+1}, \sigma_n \varepsilon_{n+1}, \varepsilon_n, T_{n+1}, T_n, t_{n+1}, t_n, \right)$$

where ω_{n+1} is the current value of the scalar damage parameter.

Implementations

Scalar damage, defined in rate form

Overview

This object further simplifies a :cpp:class`neml::ScalarDamage` model to provide the updated damage value by integrating the damage rate. The model then defines the updated damage as

$$\omega_{n+1} = \omega_n + \dot{\omega} \Delta t$$

where $\dot{\omega}$ is the damage rate, defined in a subclass with the *damage_rate* method. The associated derivatives of the updated damage with respect to the current damage, the stress, and the strain and defined analogously.

Implementations

Modular creep damage

Overview

This object implements the classical Hayhurst-Leckie-Rabotnov-Kachanov creep damage model [HL1977]. Specifically, it exactly replicates Eq. 2.6 in that paper, so that the implementation exactly replicates the analytic expressions in Eq. 2.5. The damage update is given by

$$\dot{\omega} = \left(\frac{\sigma_{eff}}{A}\right)^{\xi} (1-\omega)^{\xi-\phi}$$

where σ_{eff} is a modular effective stress, defined by a *Effective stress* object.

Effective stress

Overview

These objects provide a modular effective stress system for defining the stress correlating multiaxial stress states to uniaxial creep rupture data. Mathematically, this class provides the interface

$$\sigma_e, \frac{\partial \sigma_e}{\partial \sigma} \leftarrow \mathcal{S}(\sigma)$$

The effective stress, σ_e should be some scalar function of the stress σ such that for uniaxial load the effective stress maps the uniaxial stress tensor to the value of tensile stress.

Implementations

von Mises effective stress

Overview

This is the standard von Mises effective stress defined by

$$\sigma_e = \sqrt{\frac{3}{2} s : s}$$

with s the deviatoric stress.

Parameters

None.

Class description

class **VonMisesEffectiveStress** : public *neml*::*EffectiveStress*

von Mises stress

Public Functions

VonMisesEffectiveStress(*ParameterSet* &params)

virtual void **effective**(const double *const s, double &eff) const

virtual void **deffective**(const double *const s, double *const deff) const

Public Static Functions

static std::string **type**()

String type for the object system.

static *ParameterSet* **parameters**()

Return the default parameters.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

Max principal effective stress

Overview

The effective stress is the maximum principal stress

$$\sigma_e = \sigma_1$$

Parameters

None.

Class description

class **MaxPrincipalEffectiveStress** : public *neml*::*EffectiveStress*

Maximum principal stress.

Public Functions

MaxPrincipalEffectiveStress(*ParameterSet* &params)

virtual void **effective**(const double *const s, double &eff) const

virtual void **deffective**(const double *const s, double *const deff) const

Public Static Functions

static std::string **type**()

String type for the object system.

static *ParameterSet* **parameters**()

Return the default parameters.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

Huddleston effective stress

Overview

The effective stress is that of Huddleston [H1985], defined as

$$\sigma_e = \sigma_{vm} \exp\left(-b\left(\frac{I_1}{S_s} - 1\right)\right)$$

with

$$\sigma_{vm} = \sqrt{\frac{(\sigma_1 - \sigma_2)^2 + (\sigma_2 - \sigma_3)^2 + (\sigma_3 - \sigma_1)^2}{2}}$$

the von Mises stress,

$$I_1 = \sigma_1 + \sigma_2 + \sigma_3$$

the first stress invariant and

$$S_s = \sqrt{\sigma_1^2 + \sigma_2^2 + \sigma_3^2}$$

all in terms of the maximum principal stresses σ_1, σ_2, and σ_3.

Parameters

Parameter	Object type	Description	Default
b	double	Huddleston parameter	No

Class description

class **HuddlestonEffectiveStress** : public *neml*::*EffectiveStress*

Huddleston stress.

Public Functions

HuddlestonEffectiveStress(*ParameterSet* &params)

virtual void **effective**(const double *const s, double &eff) const

virtual void **deffective**(const double *const s, double *const deff) const

Public Static Functions

static std::string **type**()

String type for the object system.

static *ParameterSet* **parameters**()

Return the default parameters.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

Maximum of several effective stresses

Overview

The effective stress the maximum of several effective stresses:

$$\sigma_e = \max\left(\sigma_e^{(1)}, \sigma_e^{(2)}, ...\right)$$

where each individual effective stress is given by an *Effective stress* object.

Parameters

Parameter	Object type	Description	Default
`measures`	`std::vector<neml::EffectiveStress*>`	List of effective stress objects	No

Class description

class **`MaxSeveralEffectiveStress`** : public *neml*::*EffectiveStress*

Maximum of several effective stress measures.

Public Functions

`MaxSeveralEffectiveStress`(*ParameterSet* &params)

virtual void **`effective`**(const double *const s, double &eff) const

virtual void **`deffective`**(const double *const s, double *const deff) const

Public Static Functions

static std::string **`type`**()

String type for the object system.

static *ParameterSet* **`parameters`**()

Return the default parameters.

static std::unique_ptr<*NEMLObject*> **`initialize`**(*ParameterSet* &params)

Initialize from a parameter set.

Sum of several effective stresses

Overview

The effective stress is the weighted sum of several effective stresses:

$$\sigma_e = \sum_i w^{(i)} \sigma_e^{(i)}$$

where each individual effective stress is given by an *Effective stress* object.

Parameters

Parameter	Object type	Description	Default
`measures`	`std::vector<neml::EffectiveStress*>`	List of effective stress objects	No
`weights`	`std::vector<double>`	List of weights	No

Class description

class **SumSeveralEffectiveStress** : public *neml*::*EffectiveStress*

Weighted some of several effective stresses.

Public Functions

SumSeveralEffectiveStress(*ParameterSet* &params)

virtual void **effective**(const double *const s, double &eff) const

virtual void **deffective**(const double *const s, double *const deff) const

Public Static Functions

static std::string **type**()

String type for the object system.

static *ParameterSet* **parameters**()

Return the default parameters.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

Mean effective stress

Overview

This is the mean effective stress defined by

$$\sigma_e = \frac{\sigma_{11} + \sigma_{22} + \sigma_{33}}{3}$$

with σ being the full stress tensor.

Parameters

None.

Class description

class **MeanEffectiveStress** : public *neml*::*EffectiveStress*

Mean stress.

Public Functions

MeanEffectiveStress(*ParameterSet* &params)

virtual void **effective**(const double *const s, double &eff) const

virtual void **deffective**(const double *const s, double *const deff) const

Public Static Functions

static std::string **type**()

String type for the object system.

static *ParameterSet* **parameters**()

Return the default parameters.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

Class description

class **EffectiveStress** : public *neml*::*NEMLObject*

Base class of modular effective stresses used by *ModularCreepDamage*.

Subclassed by *neml::HuddlestonEffectiveStress*, *neml::MaxPrincipalEffectiveStress*, *neml::MaxSeveralEffectiveStress*, *neml::MeanEffectiveStress*, *neml::SumSeveralEffectiveStress*, *neml::VonMisesEffectiveStress*

Public Functions

EffectiveStress(*ParameterSet* &params)

virtual void **effective**(const double *const s, double &eff) const = 0

virtual void **deffective**(const double *const s, double *const deff) const = 0

Parameters

Parameter	Object type	Description	Default
elastic	*neml::LinearElasticModel*	Elasticity model	No
A	*neml::Interpolate*	Parameter	No
xi	*neml::Interpolate*	Stress exponent	No
phi	*neml::Interpolate*	Damage exponent	No
estress	*neml::EffectiveStress*	Effective stress	No

Class description

class **ModularCreepDamage** : public *neml*::*ScalarDamageRate*

Modular version of Hayhurst-Leckie-Rabotnov-Kachanov damage.

Public Functions

ModularCreepDamage(*ParameterSet* &params)

virtual void **damage_rate**(double d, const double *const e, const double *const s, double T, double t, double *const dd) const

The damage rate.

virtual void **ddamage_rate_dd**(double d, const double *const e, const double *const s, double T, double t, double *const dd) const

Derivative of damage rate wrt damage.

virtual void **ddamage_rate_de**(double d, const double *const e, const double *const s, double T, double t, double *const dd) const

Derivative of damage rate wrt strain.

virtual void **ddamage_rate_ds**(double d, const double *const e, const double *const s, double T, double t, double *const dd) const

Derivative of damage rate wrt stress.

Public Static Functions

static std::string **type**()

String type for the object system.

static *ParameterSet* **parameters**()

Return the default parameters.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

Classical creep damage

Overview

This object implements the classical Hayhurst-Leckie-Rabotnov-Kachanov creep damage model [HL1977]. The damage update is given by

$$\dot{\omega} = \left(\frac{\sigma_{eff}}{A}\right)^{\xi} (1-\omega)^{-\phi}$$
$$\sigma_{eff} = \sqrt{\frac{3}{2}\,\mathrm{dev}\,(\sigma) : \mathrm{dev}\,(\sigma)}.$$

Parameters

Parameter	Object type	Description	Default
`elastic`	*neml::LinearElasticModel*	Elasticity model	No
`A`	*neml::Interpolate*	Parameter	No
`xi`	*neml::Interpolate*	Stress exponent	No
`phi`	*neml::Interpolate*	Damage exponent	No

Class description

class **ClassicalCreepDamage** : public *neml*::*ScalarDamageRate*

Classical Hayhurst-Leckie-Rabotnov-Kachanov damage.

Public Functions

ClassicalCreepDamage(*ParameterSet* &params)

Parameters are the elastic model, the parameters A, xi, phi, the base model, the CTE, the solver tolerance, maximum iterations, and the verbosity flag.

virtual void **damage_rate**(double d, const double *const e, const double *const s, double T, double t, double *const dd) const

The damage rate.

virtual void **ddamage_rate_dd**(double d, const double *const e, const double *const s, double T, double t, double *const dd) const

Derivative of damage rate wrt damage.

virtual void **ddamage_rate_de**(double d, const double *const e, const double *const s, double T, double t, double *const dd) const

Derivative of damage rate wrt strain.

virtual void **ddamage_rate_ds**(double d, const double *const e, const double *const s, double T, double t, double *const dd) const

Derivative of damage rate wrt stress.

Public Static Functions

static std::string **type**()

String type for the object system.

static *ParameterSet* **parameters**()

Return the default parameters.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

Larson Miller Damage

Overview

This model implements the damage model

$$\dot{\omega} = \frac{1}{t_R \left(\sigma_{eff} \left(1 - \omega\right)\right)}$$

where σ_{eff} is a modular effective stress, defined by a *Effective stress* object, and t_R is a time-to-rupture Larson-Miller relation provided by a *Larson Miller correlations* object.

Parameters

Parameter	Object type	Description	Default
`elastic`	*neml::LinearElasticModel*	Elasticity model	No
`lmr`	`neml::LarsonMillerCorrelation`	Parameter	No
`estress`	*neml::EffectiveStress*	Effective stress	No

Class description

class **LarsonMillerCreepDamage** : public *neml*::*ScalarDamageRate*

Time-fraction ASME damage using a generic Larson-Miller relation and effective stress.

Public Functions

LarsonMillerCreepDamage(*ParameterSet* &params)

virtual void **damage_rate**(double d, const double *const e, const double *const s, double T, double t, double *const dd) const

The damage rate.

virtual void **ddamage_rate_dd**(double d, const double *const e, const double *const s, double T, double t, double *const dd) const

Derivative of damage rate wrt damage.

virtual void **ddamage_rate_de**(double d, const double *const e, const double *const s, double T, double t, double *const dd) const

Derivative of damage rate wrt strain.

virtual void **ddamage_rate_ds**(double d, const double *const e, const double *const s, double T, double t, double *const dd) const

Derivative of damage rate wrt stress.

Public Static Functions

static std::string **type**()

String type for the object system.

static *ParameterSet* **parameters**()

Return the default parameters.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

Class description

class **ScalarDamageRate** : public *neml*::*ScalarDamage*

Damage model where you provide the damage rate.

Subclassed by *neml::ClassicalCreepDamage*, *neml::LarsonMillerCreepDamage*, *neml::ModularCreepDamage*

Public Functions

ScalarDamageRate(*ParameterSet* &params)

virtual void **damage**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const

The combined damage variable.

virtual void **ddamage_dd**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const

Derivative with respect to damage.

virtual void **ddamage_de**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const

Derivative with respect to strain.

virtual void **ddamage_ds**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const

Derivative with respect to stress.

virtual void **damage_rate**(double d, const double *const e, const double *const s, double T, double t, double *const dd) const = 0

The damage rate.

virtual void **ddamage_rate_dd**(double d, const double *const e, const double *const s, double T, double t, double *const dd) const = 0

Derivative of damage rate wrt damage.

virtual void **ddamage_rate_de**(double d, const double *const e, const double *const s, double T, double t, double *const dd) const = 0

Derivative of damage rate wrt strain.

virtual void **ddamage_rate_ds**(double d, const double *const e, const double *const s, double T, double t, double *const dd) const = 0

Derivative of damage rate wrt stress.

Combined scalar damage models

Overview

This object combines several different scalar damage models by applying them all to the same base material model. It combines the damage models additively, so that the total damage at any time step is

$$\omega_{n+1} = \omega_n + \sum_{i=1}^{n} \Delta\omega_i$$

where $\Delta\omega_i$ is the increment in damage from the ith damage model.

Parameters

Parameter	Object type	Description	Default
`elastic`	*neml::LinearElasticModel*	Elasticity model	No
`models`	`std::vector<`*neml::NEMLScalarDamageModel*`>`	List of damage models to apply	No

Class description

class **CombinedDamage** : public *neml*::*ScalarDamage*

Stack multiple scalar damage models together.

Public Functions

CombinedDamage(*ParameterSet* &params)

Parameters: elastic model, vector of damage models, the base model CTE, solver tolerance, solver max iterations, and a verbosity flag

virtual void **damage**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const

The combined damage variable.

virtual void **ddamage_dd**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const

Derivative with respect to damage.

virtual void **ddamage_de**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const

Derivative with respect to strain.

virtual void **ddamage_ds**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const

Derivative with respect to stress.

Public Static Functions

static std::string **type**()

String type for the object system.

static *ParameterSet* **parameters**()

Return the default parameters.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

Standard damage

Overview

This object implements a "standard" damage model where the single damage variable varies only with the scalar effective inelastic strain rate. This simplifies the damage update to

$$\omega_{n+1} = \omega_n + w\left(\sigma_{n+1}, \omega_{n+1}\right) \Delta\varepsilon_{eff}^{in}.$$

A separate interface defines the damage update function w.

Implementations

Power law damage

Overview

This object implements a "standard" damage model proportional to a power law in stress and directly to the effective inelastic strain. The damage function is

$$w = A\sigma_{eff}^n$$

$$\sigma_{eff} = \sqrt{\frac{3}{2}\operatorname{dev}(\sigma) : \operatorname{dev}(\sigma)}.$$

The standard damage model multiplies this function by the inelastic strain rate in computing the damage update.

Parameters

Parameter	Object type	Description	Default
`elastic`	*neml::LinearElasticModel*	Elasticity model	No
`A`	*neml::Interpolate*	Prefactor	No
`a`	*neml::Interpolate*	Stress exponent	No

Class description

class **PowerLawDamage** : public *neml*::*StandardScalarDamage*

Simple power law damage.

Public Functions

PowerLawDamage(*ParameterSet* &params)

Parameters are an elastic model, the constants A and a, the base material model, the CTE, a solver tolerance, solver maximum number of iterations, and a verbosity flag

virtual void **f**(const double *const s_np1, double d_np1, double T_np1, double &f) const

Damage = A * s_eq**a (times the inelastic strain rate)

virtual void **df_ds**(const double *const s_np1, double d_np1, double T_np1, double *const df) const

Derivative of f wrt stress.

virtual void **df_dd**(const double *const s_np1, double d_np1, double T_np1, double &df) const

Derivative of f wrt damage.

Public Static Functions

static std::string **type**()

String type for the object system.

static *ParameterSet* **parameters**()

Return the default parameters.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

Exponential damage

Overview

This object implements a "standard" damage model proportional to the dissipated inelastic energy. The damage function is

$$w = \frac{(\omega + k_0)^{a_f}}{W_0}\sigma_{eff}$$

$$\sigma_{eff} = \sqrt{\frac{3}{2}\operatorname{dev}(\sigma) : \operatorname{dev}(\sigma)}.$$

The standard damage model multiplies this function by the inelastic strain rate in computing the damage update. Because the dissipation rate is equal to $\sigma_{eff}\dot{\varepsilon}^{in}_{eff}$ this model actually increases damage in proportion to the dissipation.

Parameters

Parameter	Object type	Description	Default
`elastic`	*neml::LinearElasticModel*	Elasticity model	No
`W0`	*neml::Interpolate*	Parameter	No
`k0`	*neml::Interpolate*	Parameter	No
`af`	*neml::Interpolate*	Parameter	No

Class description

class **ExponentialWorkDamage** : public *neml*::*StandardScalarDamage*

Simple exponential damage model.

Public Functions

ExponentialWorkDamage(*ParameterSet* &params)

Parameters are the elastic model, parameters W0, k0, and af, the base material model, the CTE, a solver tolerance, maximum number of iterations, and a verbosity flag.

virtual void **f**(const double *const s_np1, double d_np1, double T_np1, double &f) const

damage rate is (d + k0)**af / W0 * s_eq

virtual void **df_ds**(const double *const s_np1, double d_np1, double T_np1, double *const df) const

Derivative of damage wrt stress.

virtual void **df_dd**(const double *const s_np1, double d_np1, double T_np1, double &df) const

Derivative of damage wrt damage.

Public Static Functions

static std::string **type**()

String type for the object system.

static *ParameterSet* **parameters**()

Return the default parameters.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

Class description

class **StandardScalarDamage** : public *neml*::*ScalarDamage*

A standard damage model where the damage rate goes as the plastic strain.

Subclassed by *neml::ExponentialWorkDamage*, *neml::PowerLawDamage*

Public Functions

StandardScalarDamage(*ParameterSet* &params)

Parameters: elastic model, base model, CTE, solver tolerance, solver maximum number of iterations, verbosity flag

virtual void **damage**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const

Damage, now only proportional to the inelastic effective strain.

virtual void **ddamage_dd**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const

Derivative of damage wrt damage.

virtual void **ddamage_de**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const

Derivative of damage wrt strain.

virtual void **ddamage_ds**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const

Derivative of damage wrt stress.

virtual void **f**(const double *const s_np1, double d_np1, double T_np1, double &f) const = 0

The part of the damage rate proportional to the inelastic strain rate.

virtual void **df_ds**(const double *const s_np1, double d_np1, double T_np1, double *const df) const = 0

Derivative with respect to stress.

virtual void **df_dd**(const double *const s_np1, double d_np1, double T_np1, double &df) const = 0

Derivative with respect to damage.

Dissipated work damage

Overview

This object implements a damage model based on a critical value of dissipated work:

$$\dot{D} = nD^{\frac{n-1}{n}} \frac{\dot{W}}{W_{crit}\left(\dot{W}\right)}$$

$$\dot{W} = \sigma : \dot{\varepsilon}_{inelastic}.$$

The model has two parameters: W_{crit} the critical work to failure, as a function of the work rate, and n, a parameter controlling the onset of the appearance of damage in the material flow stress. In this implementation the critical work is provided as a NEML interpolate function, meaning it can have a wide variety of functional forms. In principle, the critical work might also depend on temperatures. However, at least for one material (Alloy 617) the temperature dependence is relatively negligible.

If requested the model will first take the log of the work rate before passing it to the W_{crit} function and uses 10^f of the returned value. This means the user is providing the function on a log-log scale.

Parameters

Parameter	Object type	Description	Default
`elastic`	*neml::LinearElasticModel*	Elasticity model	No
`Wcrit`	*neml::Interpolate*	Critical work	No
`n`	`double`	Damage exponent	No
`eps`	`double`	Numerical offset	`1.0e-30`
`log`	`bool`	Log transform the work to failure relation	`false`

Class description

class **WorkDamage** : public *neml*::*ScalarDamage*

The isothermal form of my pet work-based damage model.

Public Functions

WorkDamage(*ParameterSet* &params)

Parameters: elastic model, base model, CTE, solver tolerance, solver maximum number of iterations, verbosity flag

inline virtual double **d_guess**() const

Initial value of damage, overridable for models with singularities.

virtual void **damage**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const

damage rate = n * d**((n-1)/n) * W_dot / W_crit

virtual void **ddamage_dd**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const

Derivative of damage wrt damage.

virtual void **ddamage_de**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const

Derivative of damage wrt strain.

virtual void **ddamage_ds**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const

Derivative of damage wrt stress.

inline virtual double **d_init**() const

Initial value of the damage, overrideable for models with singularities.

Public Static Functions

static std::string **type**()

String type for the object system.

static *ParameterSet* **parameters**()

Return the default parameters.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

Class description

class **ScalarDamage** : public *neml*::*NEMLObject*

Scalar damage model.

Subclassed by *neml::CombinedDamage*, *neml::ScalarDamageRate*, *neml::StandardScalarDamage*, *neml::WorkDamage*

Public Functions

ScalarDamage(*ParameterSet* &params)

virtual void **damage**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const = 0

The combined damage variable.

virtual void **ddamage_dd**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const = 0

Derivative with respect to damage.

virtual void **ddamage_de**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const = 0

Derivative with respect to strain.

virtual void **ddamage_ds**(double d_np1, double d_n, const double *const e_np1, const double *const e_n, const double *const s_np1, const double *const s_n, double T_np1, double T_n, double t_np1, double t_n, double *const dd) const = 0

Derivative with respect to stress.

inline virtual double **d_init**() const

Initial value of the damage, overrideable for models with singularities.

Class description

class **NEMLScalarDamagedModel_sd** : public *neml*::*NEMLDamagedModel_sd*, public *neml*::*Solvable*

Special case where the damage variable is a scalar.

Public Functions

NEMLScalarDamagedModel_sd(*ParameterSet* &params)

Parameters are an elastic model, a base model, the CTE, a solver tolerance, the maximum number of solver iterations, and a verbosity flag

virtual void **update_sd_actual**(const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n, double *const A_np1, double &u_np1, double u_n, double &p_np1, double p_n)

Stress update using the scalar damage model.

virtual size_t **ndamage**() const

Equal to 1.

virtual void **populate_damage**(*History* &hist) const

Populate damage.

virtual void **init_damage**(*History* &hist) const

Initialize to zero.

virtual size_t **nparams**() const

Number of parameters for the solver.

virtual void **init_x**(double *const x, *TrialState* *ts)

Initialize the solver vector.

virtual void **RJ**(const double *const x, *TrialState* *ts, double *const R, double *const J)

The actual nonlinear residual and Jacobian to solve.

void **make_trial_state**(const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, const double *const s_n, const double *const h_n, double u_n, double p_n, SDTrialState &tss)

Setup a trial state from known information.

virtual double **get_damage**(const double *const h_np1)

Used to find the damage value from the history.

virtual bool **should_del_element**(const double *const h_np1)

Used to determine if element should be deleted.

virtual bool **is_damage_model**() const

Used to determine if this is a damage model.

Public Static Functions

static std::string **type**()

String type for the object system.

static *ParameterSet* **parameters**()

Return the default parameters.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

13.3 Class description

class **NEMLDamagedModel_sd** : public *neml*::*NEMLModel_sd*

Small strain damage model.

Subclassed by *neml::NEMLScalarDamagedModel_sd*

Public Functions

NEMLDamagedModel_sd(*ParameterSet* &params)

Input is an elastic model, an undamaged base material, and the CTE.

virtual void **populate_state**(*History* &hist) const

Populate the internal variables.

virtual void **init_state**(*History* &hist) const

Initialize base according to the base model and damage according to.

virtual void **update_sd_actual**(const double *const e_np1, const double *const e_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n, double *const A_np1, double &u_np1, double u_n, double &p_np1, double p_n) = 0

The damaged stress update.

virtual size_t **ndamage**() const = 0

Number of damage variables.

virtual void **populate_damage**(*History* &hist) const = 0

Populate the damage variables.

virtual void **init_damage**(*History* &hist) const = 0

Setup the damage variables.

virtual void **set_elastic_model**(std::shared_ptr<*LinearElasticModel*> emodel)

Override the elastic model.

CHAPTER

FOURTEEN

LARSON MILLER CORRELATIONS

14.1 Overview

This class implements a classical Larson-Miller [LM1952] correlation between stress, temperature, and rupture time. These correlations are developed using a large database of creep test results giving the applied stress, temperature, and the resulting time to rupture for a given material for many different experimental conditions.

The Larson-Miller relation takes the form:

$$\log_{10} \sigma_R = f\left(\mathrm{LMP}\right)$$

where σ_R is the time to rupture and f is some generic function (often a polynomial regression or a piecewise polynomial regression) of the Larson-Miller parameter LMP, defined as

$$\mathrm{LMP} = T\left(C + \log_{10} t_R\right)$$

where T is absolute temperature, C is a constant fit to data, and t_R is the time to rupture.

NEML implements Larson-Miller relations by using a generic *interpolate* object. That is, the interpolate object provides the function f in the above equation relating the Larson-Miller parameter to the *log* of the stress to rupture.

14.2 Parameters

Parameter	Object type	Description	Default
`function`	*`neml::Interpolate`*	Generic Larson-Miller relation	No
`lmr`	`double`	Constant C from the Larson-Miller parameter	No
`tol`	`double`	Solver tolerance	`1.0e-6`
`miter`	`int`	Maximum solver iterations	`20`
`verbose`	`bool`	Verbosity flag	`false`

14.3 Class description

class **LarsonMillerRelation** : public *neml*::*NEMLObject*, public *neml*::*Solvable*

Classical Larson-Miller relation between stress and rupture time.

Public Functions

LarsonMillerRelation(*ParameterSet* &params)

void **sR**(double t, double T, double &s) const

Stress as a function of rupture time (not really used)

void **tR**(double s, double T, double &t)

Rupture time as a function of stress.

void **dtR_ds**(double s, double T, double &dt)

Derivative of rupture time with respect to stress.

virtual size_t **nparams**() const

Number of solver parameters.

virtual void **init_x**(double *const x, *TrialState* *ts)

Setup an iteration vector in the solver.

virtual void **RJ**(const double *const x, *TrialState* *ts, double *const R, double *const J)

Solver function returning the residual and jacobian of the nonlinear system of equations integrating the model

Public Static Functions

static std::string **type**()

Type for the object system.

static *ParameterSet* **parameters**()

Parameters for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Setup from a *ParameterSet*.

CHAPTER

FIFTEEN

BLOCK EVALUATION FUNCTIONS

15.1 Overview

This module provides helper routines for evaluating batches of NEML models sequentially. These routines can take advantage of:

1. Shared memory parallelism to evaluate the stress updates on many cores.
2. Nicely-aligned memory.
3. Explicit block matrix forms for converting to and from Mandel notation

15.2 Module description

void *neml*::**block_evaluate**(std::shared_ptr<*NEMLModel*> model, size_t nblock, const double *const e_np1, const double *const e_n, const double *const T_np1, const double *const T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n, double *const A_np1, double *const u_np1, const double *const u_n, double *const p_np1, const double *p_n)

Block update in tensor notation.

This function takes input in full tensor notation, internally converts to Mandel notation using highly-efficient block matrix multiplications, runs the stress update, using OpenMP if enabled, and reconverts to full tensors using more block matrix multiplications.

CHAPTER

SIXTEEN

CRYSTAL PLASTICITY

This submodule of NEML provides models for the constitutive response of single crystals that deform though a combination of recoverable elastic deformation and unrecoverable inelastic slip or twinning. These constitutive models are implemented through the *large deformation incremental* interface, meaning the constitutive model is defined incrementally through the symmetric and skew parts of the spatial velocity gradient. The models can be used in small strain contexts as well, though this means that crystal orientations remain static throughout the calculation.

Fundamentally, crystal plasticity material models are treated no differently than any other macroscale plasticity model in NEML. They use the same *interface* as any other constitutive model and are at heart just a collection of ordinary differential equations that provide update equations for the stress, given the strain, and some set of internal variables. This means the user can use the crystal plasticity materials models in a finite element solver exactly as with any of the other models in NEML. For example, crystal models are defined using the XML-based input system just like macroscale plasticity models.

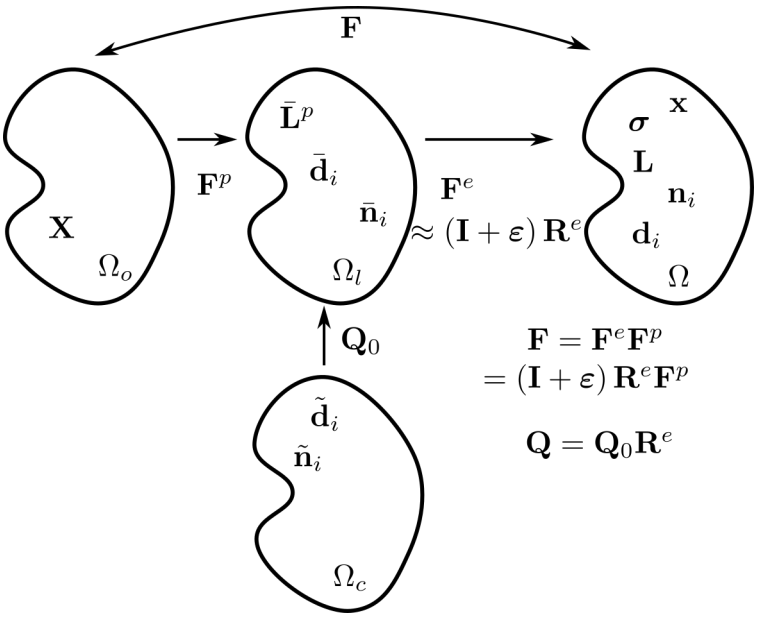
One difference between crystal plasticity and macroscale plasticity is that plastic (and elastic) deformation on the microscale is often anisotropic. The anisotropy is dictated by the orientation of the single crystal represented by the crystal plasticity model. In NEML this orientation is defined as the rotation from the original crystal frame of reference to the current frame of reference (see figure below). That is, NEML natively treats orientations as active rotations from the crystal to the lab frame. This orientation is called $\mathbf{Q}$ in the derivation below. The passive rotation from lab to crystal, often measured experimentally, is the transpose of this rotation $\mathbf{Q}^T$. The NEML interface provides helper methods for extracting orientations from the single crystal model in either convention.

Crystal plasticity was originally developed to track the evolution of this orientation as the single crystal undergoes mechanical loading (either by itself or in conjugation with neighboring grains either in a homogenized or full-field model). The models implemented in this module are sufficient for this task, though the particular implementation is focused more on fully-resolved calculations, like the Crystal Plasticity Finite Element Method (CPFEM), where the stress update is equally important.

16.1 Theoretical formulation

The basic kinematics of the single crystal plasticity model are described by Asaro [A1983]. The specific constitutive assumptions developed here were originally summarized in [M2015].

Four frames of reference define the model kinematics: the initial crystal frame Ω_c, the reference frame Ω_o, the unloaded intermediate frame Ω_l, and the current frame Ω. The kinematic update defines how the material deforms from the reference frame at time zero to the current frame at the given time t.



The kinematic framework begins with the deformation gradient:

$$\mathbf{F} = \mathbf{I} + \frac{\partial \mathbf{x}}{\partial \mathbf{X}}$$

where $\mathbf{X}$ is the original position of a material point in the reference frame and $\mathbf{x}$ is the location of that same point in the current coordinates. Adopt the multiplicative decomposition of the deformation gradient into elastic and plastic parts [L1969]:

$$\mathbf{F} = \mathbf{F}^e\mathbf{F}^p.$$

The spatial velocity gradient can then be defined as

$$\mathbf{L} = \dot{\mathbf{F}}\mathbf{F}^{-1} = \dot{\mathbf{F}}^e\mathbf{F}^{e-1} + \mathbf{F}^e\dot{\mathbf{F}}^p\mathbf{F}^{p-1}\mathbf{F}^{e-1}.$$

For the moment, consider a general constitutive model defined by the function

$$\bar{\mathbf{L}}^p = \dot{\mathbf{F}}^p\mathbf{F}^{p-1}$$

where $\bar{\mathbf{L}}^p$, called here the plastic deformation, is a function of the stress in the intermediate frame of reference and some set of internal variables.

Now assume that the elastic stretch is small

$$\mathbf{F}^e = (\mathbf{I}+\varepsilon)\,\mathbf{R}^e$$

with $\varepsilon \ll \mathbf{I}$ so that

$$(\mathbf{I}+\varepsilon)^{-1} \approx \mathbf{I} - \varepsilon.$$

The limited recoverable elastic deformation available in single metal crystals justifies this assumption. Furthermore, neglect the elastic stretch in kinematic push forward/pull back operations, so that the translation from the intermediate to the current coordinates is given by the rotation $\mathbf{R}^e$.

Then the spatial velocity gradient expands to

$$\mathbf{L} = \dot{\varepsilon} - \dot{\varepsilon}\varepsilon + \Omega^e - \Omega^e\varepsilon + \varepsilon\Omega^e - \varepsilon\Omega^e\varepsilon + \mathbf{L}^p - \mathbf{L}^p\varepsilon + \varepsilon\mathbf{L}^p - \varepsilon\mathbf{L}^p\varepsilon.$$

Label the elastic spin

$$\Omega^e = \dot{\mathbf{R}}^e \mathbf{R}^{eT}$$

and label the plastic deformation rotated to the current frame

$$\mathbf{L}^p = \mathbf{R}^e \bar{\mathbf{L}}^p \mathbf{R}^{eT}.$$

As described above, define the current lattice rotation as

$$\mathbf{Q} = \mathbf{R}^e \mathbf{Q}_0.$$

Note this relation exposes the fundamental kinematic assumption of crystal plasticity: that the plastic deformation $\mathbf{F}^p$ does not affect the lattice coordinates. This assumption reflects the fact that in a idealized single crystal the atomic lattice after some increment of inelastic deformation caused by planar slip is identical to the lattice before the increment of slip occurred. As indicated in the figure above, this relation ties the initial crystal frame to the intermediate configuration and *not* the reference frame. The elastic spin can be written solely in terms of the current lattice rotation

$$\dot{\mathbf{Q}}\mathbf{Q}^T = \dot{\mathbf{R}}^e \mathbf{Q}_0 \mathbf{Q}_0^T \mathbf{R}^{eT} = \dot{\mathbf{R}}^e \mathbf{R}^{eT} = \Omega^e$$

As ε, is small drop terms in the spatial velocity gradient that are quadratic with respect to the elastic stretch:

$$\mathbf{L} = \dot{\varepsilon} - \dot{\varepsilon}\varepsilon + \Omega^e - \Omega^e \varepsilon + \varepsilon \Omega^e + \mathbf{L}^p - \mathbf{L}^p \varepsilon + \varepsilon \mathbf{L}^p$$

Separate out the symmetric and skew-symmetric parts

$$\mathbf{D} = \frac{1}{2}\left(\mathbf{L} + \mathbf{L}^T\right) = \dot{\varepsilon} + \mathbf{D}^p + \varepsilon\left(\Omega^e + \mathbf{W}^p\right) - \left(\Omega^e + \mathbf{W}^p\right)\varepsilon - \frac{1}{2}\left(\dot{\varepsilon}\varepsilon + \varepsilon\dot{\varepsilon}\right)$$

$$\mathbf{W} = \frac{1}{2}\left(\mathbf{L} - \mathbf{L}^T\right) = \Omega^e + \mathbf{W}^p + \varepsilon \mathbf{D}^p - \mathbf{D}^p \varepsilon + \frac{1}{2}\left(\varepsilon\dot{\varepsilon} - \dot{\varepsilon}\varepsilon\right).$$

The derivation refers to the symmetric part of the spatial velocity gradient as the "deformation rate tensor" and the skew part as the "vorticity tensor."

The NEML single crystal model furthermore neglects the mixed term $\dot{\varepsilon}\varepsilon$. This assumption is reasonable only for slow loading rates, for example those typical of high temperature structural components. This approximation considerably simplifies the deformation rate and vorticity tensors

$$\mathbf{D} = \frac{1}{2}\left(\mathbf{L} + \mathbf{L}^T\right) = \dot{\varepsilon} + \mathbf{D}^p + \varepsilon\left(\Omega^e + \mathbf{W}^p\right) - \left(\Omega^e + \mathbf{W}^p\right)\varepsilon$$

$$\mathbf{W} = \frac{1}{2}\left(\mathbf{L} - \mathbf{L}^T\right) = \Omega^e + \mathbf{W}^p + \varepsilon \mathbf{D}^p - \mathbf{D}^p \varepsilon.$$

Label the total spin as

$$\Omega^\star = \Omega^e + \mathbf{W}^p$$

and make the hypoelastic assumption that

$$\dot{\varepsilon} = \mathfrak{S} : \dot{\sigma}$$

or

$$\dot{\sigma} = \mathfrak{C} : \dot{\varepsilon}$$

where $\mathfrak{S}$ is the elastic compliance tensor *in the current coordinates*, $\mathfrak{C}$ is the elastic stiffness tensor (so that $\mathfrak{C} = \mathfrak{S}^{-1}$), and $\dot{\sigma}$ is the rate of Cauchy stress (which is in the current frame of reference). Combining these definitions and assumptions with the definition of the deformation rate tensor produces the stress update equation

$$\dot{\sigma} = \mathbf{C} : \left[\mathbf{D} - \mathbf{D}^p - \mathbf{S} : \sigma \cdot \Omega^\star + \Omega^\star \cdot \mathbf{S} : \sigma\right].$$

This equation can be integrated to update the stress from time step to time step.

A similar rearrangement of the vorticity equation gives an update equation for the current crystal orientation:

$$\Omega^e = \dot{\mathbf{Q}}\mathbf{Q}^T = \mathbf{W} - \mathbf{W}^p - \varepsilon\mathbf{D}^p + \mathbf{D}^p\varepsilon.$$

The NEML single crystal model is defined by an implicit integration of the deformation rate equation, defining the stress update, and an explicit integration of the vorticity equation, defining the updated crystal orientation. Both equations require the definition of the plastic deformation rate in the current frame of reference, $\mathbf{L}^p$, to close the system of equations. Note the actual expressions decompose the plastic deformation rate into symmetric, $\mathbf{D}^p$, and skew, $\mathbf{W}^p$, parts. A NEML crystal model is defined by the constitutive model for the plastic deformation rate as a function of the Cauchy stress, temperature, and some set of internal variables $\mathbf{h}$. The stress and orientation update equations must be supplemented with the evolution equations for the internal variables, $\dot{\mathbf{h}}$, which are integrated implicitly.

In addition to this basic kinematic framework, NEML has the ability to provide crystal models with the Nye tensor

$$\alpha = -\nabla \times \mathbf{F}^{e-1}$$

Describing the areal density of geometrically necessary dislocations. For this definition of the Nye tensor, the curl is taken in the current configuration [DHT2019]. The Nye tensor can be used as an *ad hoc* contributor to slip system flow or hardening models. The NEML system cannot calculate the Nye tensor itself – that requires gradient information not available at the material point level – but can accept the calculated Nye tensor from a higher level framework, for example an FEA model, and process it accordingly. To help with this calculation, the model provides the inverse elastic tensor $\mathbf{F}^{e-1}$ as an available output.

16.2 Crystal plasticity continuum damage

NEML has the capability to supplement this general kinematic framework with a continuum damage model that degrades the elastic properties of the material in response to one or more damage variable, evolving as a function of internal history variables and the stress state. The details of the damage subsystem are described in their own section in the manual.

16.2.1 Crystal plasticity damage models

NEML views crystal plasticity continuum damage as an alternative kinematic model to the one described in the *main documentation*. The model takes a projection operator $\mathbf{P}$, defined below, which is a function of some set of internal variables describing damage evolution, $\mathbf{d}$, the current stress state σ, and uses the projection to project damage onto the elasticity tensor used in the crystal plasticity stress update formulation.

In the most general case, consider the base stress updated provided by a model as a function of time σ'. The damage projection will alter this stress history as:

$$\sigma = \boldsymbol{P} : \sigma'$$

so that the new, modified stress rate accounting for damage becomes

$$\dot{\sigma} = \dot{\boldsymbol{P}} : \sigma' + \boldsymbol{P} : \dot{\sigma}'$$
$$\dot{\sigma} = \dot{\boldsymbol{P}} : \boldsymbol{P}^{-1} : \sigma + \boldsymbol{P} : \dot{\sigma}'$$

in the case where the damage evolves slowly compared to the evolution of the stress we can approximate this as

$$\dot{\sigma} = \boldsymbol{P} : \dot{\sigma}'$$

which is the form currently implemented in NEML.

Starting then from the basic stress update kinematics defined in the *main documentation*, the modified stress rate equation becomes

$$\dot{\sigma}_{ij} = P_{ijkl} C_{klmn} \left(D_{mn} - D^p_{mn} \right) - \sigma_{ik} \Omega^\star_{kj} + \Omega^\star_{ik} \sigma_{kj}$$

NEML implements this modified stress update as a new *KinematicModel*. A *CrystalDamageModel* provides the definition of the projection operator, including the history evolution rate equations for any internal variables used in defining the projection.

CrystalDamageModel

Overview

This forms the base class for the implementation of a particular crystal plasticity continuum damage model. The class provides the definition of the projection operator $\mathbf{P}$ as well as the associated internal damage variables d_i. The class also must define all the partial derivatives required for an implicit integration of the projection in the context of the crystal plasticity kinematics and the set of internal variables.

Implementations

NilDamageModel

Overview

This model is mainly for testing the `neml::DamagedStandardKinematicModel` formulation. It returns the identify for the damage projection operator:

$$\mathbf{P} = \mathbf{I}$$

and does not maintain any internal variables.

Parameters

None

Class description

class **NilDamageModel** : public *neml*::*CrystalDamageModel*

Temp class to check interface (delete later)

Public Functions

NilDamageModel(*ParameterSet* &params)

virtual void **init_hist**(*History* &history) const

Initialize history.

virtual *SymSymR4* **projection**(const *Symmetric* &stress, const *History* &damage, const *Orientation* &Q, *Lattice* &lattice, const *SlipRule* &slip, double T)

Returns the current projection operator.

virtual SymSymSymR6 **d_projection_d_stress**(const *Symmetric* &stress, const *History* &damage, const *Orientation* &Q, *Lattice* &lattice, const *SlipRule* &slip, double T)

Return the derivative of the projection operator wrt to the stress.

virtual *History* **d_projection_d_history**(const *Symmetric* &stress, const *History* &damage, const *Orientation* &Q, *Lattice* &lattice, const *SlipRule* &slip, double T)

Return the derivative of the projection operator wrt to the damage vars.

virtual *History* **damage_rate**(const *Symmetric* &stress, const *History* &history, const *Orientation* &Q, *Lattice* &lattice, const *SlipRule* &slip, double T, const *History* &fixed) const

Damage along each slip plane.

virtual *History* **d_damage_d_stress**(const *Symmetric* &stress, const *History* &history, const *Orientation* &Q, *Lattice* &lattice, const *SlipRule* &slip, double T, const *History* &fixed) const

Derivative of each damage with respect to stress.

virtual *History* **d_damage_d_history**(const *Symmetric* &stress, const *History* &history, const *Orientation* &Q, *Lattice* &lattice, const *SlipRule* &slip, double T, const *History* &fixed) const

Derivative of damage with respect to history.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

PlanarDamageModel

Overview

This model forms the total damage projection operator by degrading the elasticity tensor along individual slip planes. For each slip plane the framework can degrade the elastic stiffness in the normal direction of the plane and in the shear (parallel) directions independently. The stiffness degradation in each of these directions is defined by a *TransformationFunction* which converts a damage internal variable, defined by a *SlipPlaneDamage* object, and the stress in the normal direction to the plane to a suitable damage index ranging from 0 (no damage) to 1 (complete loss of stiffness in that direction). The mathematical definition of the projection operator is then

$$P_{ijkl} = \prod_{i=1}^{n_{planes}} \delta_{ij}\delta_{kl} - N\left(d^{(i)}, \sigma_{\perp}^{(i)}\right) N_{ijkl}^{(i)} - S\left(d^{(i)}, \sigma_{\perp}^{(i)}\right) S_{ijkl}^{(i)}$$

where the product proceeds over each individual slip plane i defined by some normal vector *in the current coordinates* $n_i^{(i)}$. The projection for each plane then consists of the identity, a *TransformationFunction* for the normal direction N, the normal projection operator

$$N_{ijkl}^{(i)} = n_i^{(i)} n_j^{(i)} n_k^{(i)} n_l^{(i)}$$

a *TransformationFunction* for the shear direction S, and the shear projection operator

$$S_{ijkl}^{(i)} = \left(\delta_{ik} - n_i^{(i)} n_k^{(i)}\right) n_j^{(i)} n_l^{(i)}.$$

The stress normal to the plane can be calculated as

$$\sigma_{\perp}^{(i)} = \sigma_{ij} n_i^{(i)} n_j^{(i)}.$$

The available *TransformationFunction* options are described here:

TransformationFunction

These models are part of the *PlanarDamageModel* system. They map the stress normal to a slip plane $\sigma_{\perp}^{(i)}$ and an internal damage variable, defined with a *SlipPlaneDamage* object, to a suitable damage metric ranging from 0 (no damage) to 1 (no stiffness in the direction).

Superclass description

class **TransformationFunction** : public *neml*::*NEMLObject*

Transformation functions: map the damage variable + ancillary info into the range [0,1]

Subclassed by *neml::SigmoidTransformation*, *neml::SwitchTransformation*

Public Functions

TransformationFunction(*ParameterSet* &params)

virtual double **map**(double damage, double normal_stress) = 0

Map from damage and the normal stress to [0,1].

virtual double **d_map_d_damage**(double damage, double normal_stress) = 0

Derivative of the map with respect to the damage.

virtual double **d_map_d_normal**(double damage, double normal_stress) = 0

Derivative of the map with respect to the normal stress.

Individual models

SigmoidTransformation

Overview

This transformation function is independent of the normal stress. It is a sigmoid function that maps the damage variable to the interval $[0, 1]$:

$$T\left(d^{(i)}, \sigma_{\perp}^{(i)}\right) = \begin{cases} \frac{1}{1+\left(\frac{d^{(i)}}{c-d^{(i)}}\right)^{-\beta}} & d^{(i)} < c \\ 1 & d^{(i)} \geq c \end{cases}$$

The parameter $c^{(i)}$ is some critical value of the damage variable $d^{(i)}$ on plane i and the exponent $\beta^{(i)}$ is a smoothness parameter where $\beta = 1$ represents a linear transition from 0 to 1 and higher values of β represent a more abrupt onset of the effects of damage.

Parameters

Parameter	Object type	Description	Default
`c`	`double`	Critical damage value	No
`beta`	:c:type`double`	Abruptness of damage onset	No

Class description

class **SigmoidTransformation** : public *neml*::*TransformationFunction*

Sigmoid function. x=0 -> y=0, x=c -> y=1, beta controls smoothing.

Public Functions

SigmoidTransformation(*ParameterSet* &params)

virtual double **map**(double damage, double normal_stress)

Map from damage and the normal stress to [0,1].

virtual double **d_map_d_damage**(double damage, double normal_stress)

Derivative of the map with respect to damage.

virtual double **d_map_d_normal**(double damage, double normal_stress)

Derivative of the map with respect to the normal stress.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

SwitchTransformation

Overview

This is a "meta-transformation" that takes another *TransformationFunction* as a parameter and modifies it to return a different result. Specifically, this function returns 0 (no damage) if the loading on the plane is compressive and returns the base *TransformationFunction* value if the stress on the plane is tensile. Mathematically:

$$T\left(d^{(i)}, \sigma_{\perp}^{(i)}\right) = \begin{cases} \tilde{T}\left(d^{(i)}, \sigma_{\perp}^{(i)}\right) & \sigma_{\perp}^{(i)} \geq 0 \\ 0 & \sigma_{\perp}^{(i)} < 0 \end{cases}$$

where $\tilde{T}\left(d^{(i)}, \sigma_{\perp}^{(i)}\right)$ is the base *TransformationFunction*.

Parameters

Parameter	Object type	Description	Default
base	*neml::TransformationFunctio*	Base function	No

Class Description

class **SwitchTransformation** : public *neml*::*TransformationFunction*

Normal stress switch: don't damage compression.

Public Functions

SwitchTransformation(*ParameterSet* &params)

virtual double **map**(double damage, double normal_stress)

Map from damage and the normal stress to [0,1].

virtual double **d_map_d_damage**(double damage, double normal_stress)

Derivative of the map with respect to damage.

virtual double **d_map_d_normal**(double damage, double normal_stress)

Derivative of the map with respect to the normal stress.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

The available *SlipPlaneDamage* functions are described here:

SlipPlaneDamage

These objects define an internal variable describing some damage process or measure of damage along a given slip plane. Specifically, the object defines the evolution rate of an internal variable describing the damage process on that plane along with the partial derivatives needed for an implicit integration of the rate equations.

Superclass description

class **SlipPlaneDamage** : public *neml*::*NEMLObject*

Slip plane damage functions.

Subclassed by *neml::WorkPlaneDamage*

Public Functions

SlipPlaneDamage(*ParameterSet* &params)

virtual double **setup**() const = 0

Initial value.

virtual double **damage_rate**(const std::vector<double> &shears, const std::vector<double> &sliprates, double normal_stress, double damage) = 0

Damage rate.

virtual std::vector<double> **d_damage_rate_d_shear**(const std::vector<double> &shears, const std::vector<double> &sliprates, double normal_stress, double damage) = 0

Derivative wrt shears.

virtual std::vector<double> **d_damage_rate_d_slip**(const std::vector<double> &shears, const std::vector<double> &sliprates, double normal_stress, double damage) = 0

Derivative wrt slip rates.

virtual double **d_damage_rate_d_normal**(const std::vector<double> &shears, const std::vector<double> &sliprates, double normal_stress, double damage) = 0

Derivative wrt the normal stress.

virtual double **d_damage_rate_d_damage**(const std::vector<double> &shears, const std::vector<double> &sliprates, double normal_stress, double damage) = 0

Derivative wrt the damage variable.

Individual models

WorkPlaneDamage

Overview

This model represents damage as evolving with the total dissipated inelastic work on all slip systems on the given slip plane. The rate form of the damage variable is then

$$\dot{d}^{(i)} = \sum_{j \in S_i} \tau_j \dot{\gamma}_j$$

where the index i is the slip plane, the set S_i are all the slip directions on that slip plane, τ_j is the resolved shear on a given slip system, and $\dot{\gamma}_j$ is the slip rate on the slip system.

Parameters

None

Class description

class **WorkPlaneDamage** : public *neml*::*SlipPlaneDamage*

Accumulated work.

Public Functions

WorkPlaneDamage(*ParameterSet* &params)

virtual double **setup**() const

Initial value.

virtual double **damage_rate**(const std::vector<double> &shears, const std::vector<double> &sliprates, double normal_stress, double damage)

Damage rate.

virtual std::vector<double> **d_damage_rate_d_shear**(const std::vector<double> &shears, const std::vector<double> &sliprates, double normal_stress, double damage)

Derivative wrt shears.

virtual std::vector<double> **d_damage_rate_d_slip**(const std::vector<double> &shears, const std::vector<double> &sliprates, double normal_stress, double damage)

Derivative wrt slip rates.

virtual double **d_damage_rate_d_normal**(const std::vector<double> &shears, const std::vector<double> &sliprates, double normal_stress, double damage)

Derivative wrt the normal stress.

virtual double **d_damage_rate_d_damage**(const std::vector<double> &shears, const std::vector<double> &sliprates, double normal_stress, double damage)

Derivative wrt the damage variable.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

Parameters

Parameter	Object type	Description	Default
damage	*neml::SlipPlaneDamage*	The damage model	No
shear_transformation	*neml::TransformationFunction*	The shear transformation function	No
normal_transformation	*neml::TransformationFunction*	The normal transformation function	No
lattice	*neml::Lattice*	The lattice object describing the slip geometry	No

Class description

class **PlanarDamageModel** : public *neml*::*CrystalDamageModel*

Project damage on each plane proportional to some damage measure on the plane

Public Functions

PlanarDamageModel(*ParameterSet* &params)

virtual void **init_hist**(*History* &history) const

Initialize history.

virtual *SymSymR4* **projection**(const *Symmetric* &stress, const *History* &damage, const *Orientation* &Q, *Lattice* &lattice, const *SlipRule* &slip, double T)

Returns the current projection operator.

virtual SymSymSymR6 **d_projection_d_stress**(const *Symmetric* &stress, const *History* &damage, const *Orientation* &Q, *Lattice* &lattice, const *SlipRule* &slip, double T)

Return the derivative of the projection operator wrt to the stress.

virtual *History* **d_projection_d_history**(const *Symmetric* &stress, const *History* &damage, const *Orientation* &Q, *Lattice* &lattice, const *SlipRule* &slip, double T)

Return the derivative of the projection operator wrt to the damage vars.

virtual *History* **damage_rate**(const *Symmetric* &stress, const *History* &history, const *Orientation* &Q, *Lattice* &lattice, const *SlipRule* &slip, double T, const *History* &fixed) const

Damage along each slip plane.

virtual *History* **d_damage_d_stress**(const *Symmetric* &stress, const *History* &history, const *Orientation* &Q, *Lattice* &lattice, const *SlipRule* &slip, double T, const *History* &fixed) const

Derivative of each damage with respect to stress.

virtual *History* **d_damage_d_history**(const *Symmetric* &stress, const *History* &history, const *Orientation* &Q, *Lattice* &lattice, const *SlipRule* &slip, double T, const *History* &fixed) const

Derivative of damage with respect to history.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

Class description

class **CrystalDamageModel** : public *neml*::HistoryNEMLObject

Abstract base class for slip plane damage models.

Subclassed by *neml::NilDamageModel*, *neml::PlanarDamageModel*

Public Functions

CrystalDamageModel(*ParameterSet* &params, std::vector<std::string> vars)

virtual size_t **nvars**() const

Report the number of internal variables.

virtual std::vector<std::string> **varnames**() const

Report the names of the internal variables.

virtual void **set_varnames**(std::vector<std::string> names)

Set the internal variables to new names.

virtual void **populate_hist**(*History* &history) const

Setup whatever history variables the model requires.

virtual void **init_hist**(*History* &history) const = 0

Initialize history.

virtual *SymSymR4* **projection**(const *Symmetric* &stress, const *History* &damage, const *Orientation* &Q, *Lattice* &lattice, const *SlipRule* &slip, double T) = 0

Returns the current projection operator.

virtual SymSymSymR6 **d_projection_d_stress**(const *Symmetric* &stress, const *History* &damage, const *Orientation* &Q, *Lattice* &lattice, const *SlipRule* &slip, double T) = 0

Return the derivative of the projection operator wrt to the stress.

virtual *History* **d_projection_d_history**(const *Symmetric* &stress, const *History* &damage, const *Orientation* &Q, *Lattice* &lattice, const *SlipRule* &slip, double T) = 0

Return the derivative of the projection operator wrt to the damage vars.

virtual *History* **damage_rate**(const *Symmetric* &stress, const *History* &history, const *Orientation* &Q, *Lattice* &lattice, const *SlipRule* &slip, double T, const *History* &fixed) const = 0

Damage variable rate.

virtual *History* **d_damage_d_stress**(const *Symmetric* &stress, const *History* &history, const *Orientation* &Q, *Lattice* &lattice, const *SlipRule* &slip, double T, const *History* &fixed) const = 0

Derivative of each damage with respect to stress.

virtual *History* **d_damage_d_history**(const *Symmetric* &stress, const *History* &history, const *Orientation* &Q, *Lattice* &lattice, const *SlipRule* &slip, double T, const *History* &fixed) const = 0

Derivative of damage with respect to history.

DamagedStandardKinematicModel

Parameters

Parameter	Object type	Description	Default
emodel	*neml::LinearElasticModel*	Elasticity tensor	No
imodel	*neml::InelasticModel*	Definition of the plastic deformation rate	No
dmodel	*neml::CrystalDamageModel*	Definition of the damage projection and associated internal variables	No

Class description

class **DamagedStandardKinematicModel** : public *neml*::*StandardKinematicModel*

My standard kinematic assumptions with damage.

Public Functions

DamagedStandardKinematicModel(*ParameterSet* &params)

Initialize with elastic and inelastic models.

virtual void **populate_hist**(*History* &history) const

Populate a history object with the correct variables.

virtual void **init_hist**(*History* &history) const

Initialize the history object with the starting values.

virtual *Symmetric* **stress_rate**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Stress rate.

virtual *SymSymR4* **d_stress_rate_d_stress**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of stress rate with respect to stress.

virtual *SymSymR4* **d_stress_rate_d_d**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the stress rate with respect to the deformation rate.

virtual *History* **d_stress_rate_d_history**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the stress rate with respect to the history.

virtual *History* **history_rate**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

History rate.

virtual *History* **d_history_rate_d_stress**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the history rate with respect to the stress.

virtual *History* **d_history_rate_d_history**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the history rate with respect to the history.

virtual *SymSkewR4* **d_stress_rate_d_w_decouple**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed)

Derivative of the stress rate with respect to the vorticity keeping fixed variables fixed

virtual *Skew* **spin**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

The spin rate.

virtual *Symmetric* **elastic_strains**(const *Symmetric* &stress, *Lattice* &lattice, const *Orientation* &Q, const *History* &history, double T)

Helper to calculate elastic strains.

virtual *Symmetric* **stress_increment**(const *Symmetric* &stress, const *Symmetric* &D, const *Skew* &W, double dt, *Lattice* &lattice, const *Orientation* &Q, const *History* &history, double T)

Helper to predict an elastic stress increment.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from parameter set.

static *ParameterSet* **parameters**()

Default parameters.

16.3 Specialized models

The crystal plasticity module includes several material-specific combinations of slip system hardening and slip rate models. These are documented with special chapters in the documentation:

16.3.1 Hu and Cocks model for 316H SS

This subsystem implements a single crystal model for 316H stainless steel developed by Hu, Cocks, and coworkers [HC2020]. The model includes a specific flow rule as well as a complex hardening model designed to capture the key features of dislocation forest hardening, precipitation hardening caused by the carbide and Laves phases, and solid solution strengthening caused by Mo, C, and Cr in the alloy. For clarity, the implementation breaks the complex hardening model into several smaller objects.

ArrheniusSlipRule

Overview

This class implements a thermally-activated slip rule, where the slip rate is defined by

$$\dot{\gamma}_i = \dot{\gamma}_0 \exp\left[-\frac{\Delta F_0}{kT}\left(1 - \left|\frac{\tau_i}{\tau_{CRSS,i}}\right|^A\right)^B\right] \text{sign}\left(\tau_i\right)$$

where $\dot{\gamma}_i$ is the slip rate on each system, τ_i is the resolved shear, $\tau_{CRSS,i}$ is the slip system strength,

$$\Delta F_0 = \alpha_0 G_0 b^3$$

and the remaining terms are parameters, described below.

Parameters

Parameter	Object type	Description	Default
`resistance`	*neml::SlipHardening*	Slip resistance	N
`A`	`double`	Energy barrier shape A	N
`B`	`double`	Energy barrier shape B	N
`b`	`double`	Burgers vector b	N
`g0`	`double`	Reference shear rate $\dot{\gamma}_0$	N
`G0`	`double`	Activation energy G_0	N
`k`	`double`	Boltzmann constant k	`1.3806485e-23`

Class description

class **ArrheniusSlipRule** : public *neml*::*SlipStrengthSlipRule*

An Arrhenius slip rule ala Hu and Cocks.

Public Functions

ArrheniusSlipRule(*ParameterSet* &params)

Initialize with the strength object, the reference strain rate, and the rate sensitivity

virtual double **scalar_sslip**(size_t g, size_t i, double tau, double strength, double T) const

The slip rate definition.

virtual double **scalar_d_sslip_dtau**(size_t g, size_t i, double tau, double strength, double T) const

Derivative of slip rate with respect to the resolved shear.

virtual double **scalar_d_sslip_dstrength**(size_t g, size_t i, double tau, double strength, double T) const

Derivative of the slip rate with respect to the strength.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

Hardening model

The Hu and Cocks model combines dislocation forest hardening, precipitation hardening, and solid solution strengthening. The specific model described in the original work considers two precipitation reactions:

1. C and Cr form $Cr_{23}C_6$carbides
2. Mo forms Fe_2Mo

Precipitation strengthening and solid solution strengthening are coupled: as the chemical species contributing the to strengthen phase come out of solution that decreases the solid solution strengthening while increasing the precipitation strengthen. The precipitation reactions undergo two phases of evolution: diffusion-controlled growth drawing the required chemical species out of solution followed by Ostwald ripening where the larger precipitates cannibalize the smaller precipitates.

The implementation in NEML treats these reactions are general – the user can specify as many chemical species feeding into as many precipitation reactions as needed. The only current restriction is that two precipitation reactions cannot compete for the same chemical species in solution. This behavior could be added to the implementation if needed.

Each precipitation-solid solution strengthening group can then be treated separately. Similarly, dislocation hardening can be treated separately from the precipitation/solid solution strengthening. The implementation then uses three objects to form the slip system strengths:

1. A *DislocationSpacingHardening* object to manage dislocation hardening
2. An *HuCocksPrecipitationModel* object for each precipitation reactor
3. A *HuCocksHardening* object to sum the contributions of each mechanism on the slip system strength.

For full details of the implementation in NEML see [VM2021].

HuCocksHardening

Overview

This object sums the contributions of the individual precipitation reactions and the dislocation hardening into a single slip system strength. The equation it implements is

$$\tau_{CRSS,i} = \sqrt{\tau_{d,i}^2 + \tau_p^2} + \tau_s$$

where $\tau_{d,i}$ is the dislocation hardening strength, provided by the *DislocationSpacingHardening* model.

τ_p is the total precipitation hardening given by

$$\tau_p = \frac{\alpha_p G b}{L_p}$$

with α_p an interaction coefficient, G the shear modulus, b the Burgers vector, and L_p is

$$L_p = \sqrt{\frac{1}{\sum_i 2 r_i N_i}}.$$

with r_i and N_i internal variables defined by the individual *HuCocksPrecipitationModel* precipitation reaction models.

τ_s is then the total solid solution strengthening hardening given by

$$\tau_s = \frac{\alpha_s G b}{L_s}$$

with α_s an interaction coefficient and using

$$L_s = \sqrt{\frac{1}{b\sum_j \frac{c_j}{v_{m,i}}}}$$

with c_j the chemical concentrations contributing to each precipitation reaction and $v_{m,i}$ the corresponding molecular volumes. The chemical concentrations are again defined by the individual *HuCocksPrecipitationModel* models.

Parameters

Parameter	Object type	Description	Default
dmodel	*neml::SlipHardening*	Dislocation hardening model	N
pmodels	*std::vector*	Precipitation hardening models	N
ap	double	Precipitation hardening interaction coefficient α_p	N
ac	double	Solid solution interaction coefficient α_c	N
b	double	Burgers vector b	N
G	*neml::Interpolate*	Shear modulus	N

Class description

class **HuCocksHardening** : public *neml*::*SlipHardening*

Full Hu and Cocks hardening model.

Public Functions

HuCocksHardening(*ParameterSet* &params)

virtual std::vector<std::string> **varnames**() const

Report your variable names.

virtual void **set_varnames**(std::vector<std::string> vars)

Set new varnames.

virtual void **populate_hist**(*History* &history) const

Request whatever history you will need.

virtual void **init_hist**(*History* &history) const

Setup history.

virtual double **hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Map the set of history variables to the slip system hardening.

virtual *History* **d_hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Derivative of the map wrt to history.

virtual *History* **hist**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

The rate of the history.

virtual *History* **d_hist_d_s**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt stress.

virtual *History* **d_hist_d_h**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt the history.

virtual *History* **d_hist_d_h_ext**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed, std::vector<std::string> ext) const

Derivative of this history wrt the history, external variables.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

HuCocksPrecipitationModel

Overview

This class manages a *single* precipitation reaction. The model considers the evolution of a single precipitate through two phases:

1. Nucleation and growth by diffusion of the underlying chemical species out of the solid solution, followed by
2. Ostwald ripening after the solid solution concentrations reach their equilibrium values

During the first phase, the precipitates draw the chemical species out of the solution, affecting the solid solution strengthening provided by those elements.

The model tracks a precipitation reaction with three internal variables: *math:* $\mathcal{f}$ the volume fraction, N the number volume density, and r the average precipitate radius. One of these three variables is redundant, but the model evolves all three (consistency) to improve the numerical stability of the as a whole. The chemical concentrations in solution underlying the precipitation reaction can be determined given these three variables describing the precipitates.

The model applies a different ODE to evolve the internal variables in each of the regimes. The two regimes are split by the chemical concentration of the critical species in solution. The model is in the growth regime when

$$c_j < c_{eq,j}$$

for *all* species contributing to the reaction. Conversely, the model is in the ripening regime when

$$c_j = c_{eq,j}$$

for that critical species.

Growth regime

In the growth regime the chemical concentrations evolve as

$$c_j = \frac{c_{0,j} - f c_{p,j}}{1 - f}$$

where c_j is the concentration in solution for species j, $c_{0,j}$ is the initial solution concentration for that species, and $c_{p.j}$ is the chemical concentration of the species in the precipitate. The model stays in the growth regime until the first species contributing the precipitation reaction reaches the solution equilibrium concentration $c_{eq,j}$.

In this regime

$$\dot{f}_{growth} = \frac{4}{3}\pi\left(\dot{N}r^3 + 3Nr^2\dot{r}\right)$$

$$\dot{r}_{growth} = \frac{D}{r}\frac{c_j - c_{eq,j}}{c_{p,j} - c_{eq,j}} + \frac{\dot{N}_{growth}}{N}\left(r_c - r\right)$$

with G_v the Gibb's free energy driving the reaction

$$G_v = -\frac{kT}{v_m}\ln\frac{c_{eff}}{c_{eff,eq}}$$

using

$$c_{eff} = \prod_j c_j.nit$$

Diffusion of the *slowest species* controls the reaction rate, with

$$D = D_0 \exp\left(\frac{-Q_0}{RT}\right)$$

where D_0 is the diffusivity at absolute zero, Q_0 the activation energy, and R the universal gas constant.

Finally,

$$r_c = -2\frac{\chi}{G_v}$$

with χ the interface energy.

For the nucleation rate:

$$\dot{N}_{growth} = N_0 Z\beta \exp\left(-\frac{G^*}{kT}\right)$$

with

$$G^* = \frac{16\pi\chi^3}{3G_v^2}$$

and

$$Z\beta = \frac{2v_m D c_j}{a_m^4}\sqrt{\frac{\chi}{kT}}$$

where a_m is the relevant lattice parameter.

In the growth regime the nucleation rate is positive.

Ripening regime

In the ripening regime the solution chemical concentrations are frozen and do not change. The critical species, the element which first reaches the equilibrium concentration, is frozen at that solution equilibrium concentration $c_{j,eq}$ and the other species remained fixed at the final concentrations from the growth phase.

In this regime:

$$\dot{f}_{ripening} = 0$$

$$\dot{r}_{ripening} = \frac{M}{3r^2}$$

$$\dot{N}_{ripening} = -\frac{3N}{r}\dot{r}_{ripening}$$

with

$$M = C_f \frac{8\chi V_m D c_j}{9RT}$$

where C_f is a coarsening factor and $V_m = N_a v_m$ is the molar volume (N_a Avagadro's number).

In the ripening regime the nucleation rate is negative, the radius growth rate is positive, and there is no net growth in volume fraction.

Switching mechanisms

A hard switch between the growth and ripening regimes produces an unstable numerical model. Instead the NEML implementation mixes the two rates using a sigmoid function:

$$\dot{r} = f(c_j)\dot{r}_{growth} + (1 - f(c_j))\dot{r}_{ripening}$$

and

$$\dot{N} = f(c_j)\dot{N}_{growth} + (1 - f(c_j))\dot{N}_{ripening}$$

where

$$f\left(c_j\right) = \begin{cases} \frac{c_j - c_{0,j}}{c_{eq,j} - c_{0,j}} & c_j \leq c_{eq,j} \\ 1 & c_j > c_{eq,j} \end{cases}$$

With this setup the volume fraction evolution equation naturally trends towards zero in the ripening regime.

Additionally, the equations are scaled to equalize the magnitude of the internal variables, again to help with numerical performance

Parameters

Parameter	Object type	Description	Default
c0	*std::vector*	Initial solution concentration of each species	N
cp	*std::vector*	Precipitate concentration of each species	N
ceq	*std::vector*	Equilibrium solution concentration of each species	N
am	double	Lattice parameter	N
N0	double	Nucleation site density N_0	N
Vm	double	Molar volume	N
chi	double	Surface energy	N
D0	double	Reference diffusivity	N
Q0	double	Diffusion activation energy	N
Cf	*neml::Interpolate*	Coarsening factor	N
kboltz	double	Boltzmann constant	1.3806485e-23
R	double	Gas constant	8.31462
Na	double	Avagadro's number	6.02e23
rate	size_t	Index of rate-limiting chemical species	0
f_init	double	Initial volume fraction	4.18879e-16
r_init	double	Initial radius	1e-9
N_init	double	Initial number density	1e11
fs	double	Scaling factor on volume fraction	0.1
rs	double	Scaling factor on radius	1e-9
Ns	double	Scaling factor on number density	1e12

Class description

class **HuCocksPrecipitationModel** : public *neml*::HistoryNEMLObject

Implementation of a single chemistry <-> size model.

Public Functions

HuCocksPrecipitationModel(*ParameterSet* &params)

virtual std::vector<std::string> **varnames**() const

Report your variable names.

virtual void **set_varnames**(std::vector<std::string> vars)

Set new varnames.

virtual void **populate_hist**(*History* &history) const

Request whatever history you will need.

virtual void **init_hist**(*History* &history) const

Setup history.

double **f**(const *History* &history) const
Actual (unscaled) f.

double **r**(const *History* &history) const
Actual (unscaled) r.

double **N**(const *History* &history) const
Actual (unscaled) N.

std::vector<double> **rate**(const *History* &history, double T) const
Rate vector.

std::vector<std::vector<double>> **jac**(const *History* &history, double T) const
Jacobian matrix.

virtual double **f_rate**(double f, double r, double N, double T) const
The volume fraction rate.

virtual double **df_df**(double f, double r, double N, double T) const
df_df

virtual double **df_dr**(double f, double r, double N, double T) const
df_dr

virtual double **df_dN**(double f, double r, double N, double T) const
df_dN

virtual double **r_rate**(double f, double r, double N, double T) const
The radius rate.

virtual double **dr_df**(double f, double r, double N, double T) const
dr_df

virtual double **dr_dr**(double f, double r, double N, double T) const
dr_dr

virtual double **dr_dN**(double f, double r, double N, double T) const
dr_dN

virtual double **N_rate**(double f, double r, double N, double T) const
The number density rate.

virtual double **dN_df**(double f, double r, double N, double T) const
dN_df

virtual double **dN_dr**(double f, double r, double N, double T) const
dN_dr

virtual double **dN_dN**(double f, double r, double N, double T) const
dN_dN

size_t **nspecies**() const
Number of chemical species.

std::vector<double> **c**(double f, double T) const
Concentration vector.

std::vector<double> **dc_df**(double f, double T) const
Derivative of the concentration vector.

double **Gv**(double f, double T) const

Driving force for precipitation.

double **dG_df**(double f, double T) const

Derivative of the driving force wrt f.

double **vm**() const

Access the molecular volume.

inline double **fs**() const

Get the value of the volume fraction scaling term.

inline double **rs**() const

Get the value of the radius scaling term.

inline double **Ns**() const

Get the value of the number density scaling term.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

DislocationSpacingHardening

Overview

This model can be used as part of the *HuCocksHardening* model or as a stand-alone model for forest dislocation hardening. The model maintains a single, scalar dislocation density for each slip system, parameterized as a mean obstacle spacing. The slip system strength is given in terms of these obstacle spacings as

$$\tau_{d,i} = \frac{\alpha_d G b}{L_{d,i}}$$

with α_d an interaction coefficient, G the shear modulus, b the Burgers vector, and $L_{d,i}$ the dislocation obstacle spacing on system i. This obstacle spacing evolves as

$$\dot{L}_{d,i} = -L_{d,i}^3 \left(J_1 \left|\dot{\gamma}_i\right| + J_2 \sum_{j \neq i} \left|\dot{\gamma}_j\right| \right) + \frac{K}{L_{d,i}^3}$$

with J_1 the self hardening coefficient, J_2 the latent hardening coefficient, and K a temperature dependent parameter describing dislocation recovery.

Parameters

Parameter	Object type	Description	Default
J1	*neml::Interpolate*	Self hardening coefficient	N
J2	*neml::Interpolate*	Latent hardening coefficient	N
K	*neml::Interpolate*	Recovery coefficient	N
L0	double	Initial obstacle spacing	N
a	double	Interaction coefficient α_d	N
b	double	Burgers vector	N
G	*neml::Interpolate*	Shear modulus	N
L	neml::lattice	Lattice to extract number of systems	N
varprefix	std::string	Prefix of internal variables	"spacing"

Class description

class **DislocationSpacingHardening** : public *neml*::*SlipHardening*

Standard dislocation density model, here evolving the spacing.

Public Functions

DislocationSpacingHardening(*ParameterSet* &params)

virtual std::vector<std::string> **varnames**() const

Report your variable names.

virtual void **set_varnames**(std::vector<std::string> vars)

Set new varnames.

virtual void **populate_hist**(*History* &history) const

Request whatever history you will need.

virtual void **init_hist**(*History* &history) const

Setup history.

virtual double **hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Map the set of history variables to the slip system hardening.

virtual *History* **d_hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Derivative of the map wrt to history.

virtual *History* **hist**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

The rate of the history.

virtual *History* **d_hist_d_s**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt stress.

virtual *History* **d_hist_d_h**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt the history.

virtual *History* **d_hist_d_h_ext**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed, std::vector<std::string> ext) const

Derivative of this history wrt the history, external variables.

size_t **size**() const

Number of slip systems contributing.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

16.4 Postprocessors

A system of postprocessors provides a generic interface for altering the state of a crystal model in response to some internal (i.e. within the material system) or external (e.g. from a polycrystal or full-field solver) stimulus. As an example, this system can be used to reorient a crystal by some twinning operation when certain criteria are met.

16.4.1 Single crystal model postprocessors

This class provides means to alter the state of a crystal model when certain conditions are met, either based on the internal state of the crystal model or in response to a stimulus from an external object (like a full field solver). These actions are not integrated into the implicit time integration and so occur explicitly after successfully stress updates. This may lead to decreased numerical convergence for the time step right after the action triggers.

PTRTwinReorientation

Overview

This class implements the Predominate Twin Reorientation model [TLK1991], which reorients the crystal according to some twin geometry once the integrated slip on the twin system(s) reaches a critical value.

Specifically, the model reorients the crystal using the twin transformation for the twin system that reaches a given twinning fraction, defined as

$$f_i = \frac{\gamma_i}{s_i}$$

where γ_i is the integrated slip on twin system i and s_i is the characteristic shear for that twin system. The post process expects the base crystal model to provide the integrated slip rates on each twin system and the post processor itself maintains internal variables representing the twin fractions as well as a flag for if the crystal as a whole underwent the twin transformation.

Parameters

Parameter	Object type	Description	Default
threshold	*neml::Interpolate*	Twin fraction threshold	N
prefix	std::string	Integrated slip internal variable prefix	"slip"

Class description

class **PTRTwinReorientation** : public *neml*::CrystalPostprocessor

Reorients twins based on a PTR criteria.

Public Functions

PTRTwinReorientation(*ParameterSet* &params)

virtual void **populate_hist**(const *Lattice* &L, *History* &history) const

virtual void **init_hist**(const *Lattice* &L, *History* &history) const

virtual void **act**(*SingleCrystalModel* &model, const *Lattice*&, const double &T, const *Symmetric* &D, const *Skew* &W, *History* &state, const *History* &prev_state)

Public Static Functions

static std::string **type**()

Type for the object system.

static *ParameterSet* **parameters**()

Parameters for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Setup from a *ParameterSet*.

16.5 Mathematical operations

The preceding derivation shows that there are a lot of tensor operations used in the crystal plasticity stress update. Currently, the macroscale NEML material models use raw pointer arrays to represent vectors and tensors. This would be unwieldy for the crystal plasticity models. Instead, the crystal plasticity module uses C++ objects representing vectors, rank 2 tensors, rank 4 tensors, and rotations (internally represented as quaternions). This object system is described in the *math helpers* documentation.

The object system can either manage its own memory, in which case each individual vector, tensors, etc. is stored in separate, small blocks of memory. Alternatively, the system can work with externally-managed memory, for example if the data for numerous single crystal models are stored sequentially, as might occur in either a homogenized analysis of a polycrystal or in a CPFEM calculation. In this mode the object system provides a convenient, object-oriented interface to the user but does not copy data out of the large blocked arrays. This approach is used in the actual implementation of the crystal models to avoid excess memory use, copying, and to help the compiler perform vectorization optimizations.

Note: Future versions of NEML will replace the raw pointer math in the macroscale plasticity models with the object system.

16.6 Storing history variables

The macroscale NEML material models use raw pointer arrays to represent the model internal variables. This means the person implementing the model is responsible for remembering which variable aligns with which position in the flat array. Similarly, it means that the end-user must know the index of an internal variable of interest in order to include it in the output of a finite element simulation.

The history object stores the variables and corresponding meta-information – the name of the variable and the type of object (scalar, vector, rotation, etc.). The history object is described in a *seperate section*.

As with the tensor object system, the history set object can either manage its own memory or work with externally-managed memory. Again, this facilitate vectorization and avoids copying when the input program can provide the history of many material points in a large block array.

Note: Future versions of NEML will replace raw pointer arrays storing history vectors in the macroscale plasticity models with the object system.

16.7 Polycrystal homogenization and plotting

To help with developing and debugging models, NEML provides python implementations of simple polycrystal homogenization models and a suite of python functions for plotting pole figures, inverse pole figures, and other crystallographic information.

16.7.1 Polycrystal models

Overview

This python file provides polycrystal homogenization models. Here the goal is to homogenize the response of a large collection of single crystal orientations (i.e. grains) with individual single crystal plastic responses into a macroscale, effective plastic response.

Polycrystal homogenization models are simply *NEMLModel_ldi* objects with that standard interface. This means they can be used in any of the NEML python drivers or in finite element analysis exactly like any other NEML material model. They can even be defined (somewhat tediously) in the NEML XML file and saved for future use. The only difference is that they take as parameters a *Implementation* object (itself a *NEMLModel_ldi* object) and a list of orientations as input, instead of some set of material parameters.

Implementations

TaylorModel

Overview

This polycrystal homogenization model implements the standard Taylor approximation. Here the individual crystal receives the same deformation information and the resulting stresses are averaged.

Mathematically, each crystal receives the same, macroscopic $\mathbf{D}$ and $\mathbf{W}$ deformation rate objects

$$\mathbf{D}_i = \mathbf{D}$$
$$\mathbf{W}_i = \mathbf{W}$$

and the macroscale stress is:

$$\sigma = \frac{\mathbf{1}}{\mathbf{n}} \sum_{\mathbf{i=1}}^{\mathbf{n_{crystal}}} \sigma_{\mathbf{i}}$$

The stress updates can be completed in parallel using OpenMP threads.

Parameters

Parameter	Object type	Description	Default
`model`	*neml::SingleCrystalModel*	Single crystal update	N
`qs`	`std::vector<`*neml::Orientation*`>`	Vector of orientations	N
`nthreads`	`int`	Number of threads to use	1
`weights`	`std::vector<double>`	Weights	1.0

Class description

class **TaylorModel** : public *neml*::*PolycrystalModel*

Public Functions

TaylorModel(*ParameterSet* &params)

virtual void **update_ld_inc**(const double *const d_np1, const double *const d_n, const double *const w_np1, const double *const w_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n, double *const A_np1, double *const B_np1, double &u_np1, double u_n, double &p_np1, double p_n)

Large strain incremental update.

virtual double **alpha**(double T) const

Instantaneous thermal expansion coefficient as a function of temperature.

virtual void **elastic_strains**(const double *const s_np1, double T_np1, const double *const h_np1, double *const e_np1) const

Elastic strain for a given stress, temperature, and history state.

Public Static Functions

static std::string **type**()
Type for the object system.

static *ParameterSet* **parameters**()
Parameters for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)
Setup from a *ParameterSet*.

Class description

class **PolycrystalModel** : public *neml*::*NEMLModel_ldi*
Generic superclass.

Subclassed by *neml::TaylorModel*

Public Functions

PolycrystalModel(*ParameterSet* &params)

size_t **n**() const

virtual void **populate_hist**(*History* &hist) const
Setup the history.

virtual void **init_hist**(*History* &hist) const
Initialize the history.

inline virtual void **populate_state**(*History* &hist) const
Setup the actual evolving state.

inline virtual void **init_state**(*History* &hist) const
Initialize the actual evolving state.

double ***history**(double *const store, size_t i) const

double ***stress**(double *const store, size_t i) const

double ***d**(double *const store, size_t i) const

double ***w**(double *const store, size_t i) const

const double ***history**(const double *const store, size_t i) const

const double ***stress**(const double *const store, size_t i) const

const double ***d**(const double *const store, size_t i) const

const double ***w**(const double *const store, size_t i) const

virtual std::vector<*Orientation*> **orientations**(double *const store) const

virtual std::vector<*Orientation*> **orientations_active**(double *const store) const

16.7.2 Crystallography plotting routines

This python module provides functions for plotting pole figures, inverse pole figures, and other useful crystal orientation information.

16.8 Implementation

NEML crystal models are implemented as a subclass of the *large deformation incremental* interface. In addition to the basic stress, history, and orientation updates the class also provides methods for getting the current crystal orientation (in active or passive convention), useful for output, and similar methods for setting or resetting the crystal orientation, useful either for bulk input of crystal orientations from a separate file for CPFEM calculations or for interfacing with models which cause crystal to reorient, for example twinning or recrystallization models.

The implementation uses a coupled, Euler implicit integration of the stress and internal variable rate equations. After the model successfully updates these quantities it then uses a separate Euler explicit exponential integration of the elastic spin to update the crystal orientation. The exponential integrator ensures the orientation remains in the special orthogonal group.

The crystal model relies on two major subobjects: a *KinematicModel*, which defines the form of the stress, history, and orientation rates, and a *Lattice* object providing crystallographic information about the crystal system.

16.8.1 KinematicModel

Overview

A kinematics model provides the interfaces:

$$\dot{\sigma}, \frac{\partial \dot{\sigma}}{\partial \sigma}, \frac{\partial \dot{\sigma}}{\partial \mathbf{h}}, \frac{\partial \dot{\sigma}}{\partial \mathbf{D}}, \frac{\partial \dot{\sigma}}{\partial \mathbf{W}} \leftarrow \mathcal{S}\left(\sigma, \mathbf{h}, \mathbf{D}, \mathbf{W}, \alpha, T\right)$$

$$\dot{\mathbf{h}}, \frac{\partial \dot{\mathbf{h}}}{\partial \sigma}, \frac{\partial \dot{\mathbf{h}}}{\partial \mathbf{h}}, \frac{\partial \dot{\mathbf{h}}}{\partial \mathbf{D}}, \frac{\partial \dot{\mathbf{h}}}{\partial \mathbf{W}} \leftarrow \mathcal{H}\left(\sigma, \mathbf{h}, \mathbf{D}, \mathbf{W}, \alpha, T\right)$$

$$\Omega^e, \frac{\partial \Omega^e}{\partial \sigma}, \frac{\partial \Omega^e}{\partial \mathbf{h}}, \frac{\partial \Omega^e}{\partial \mathbf{D}}, \frac{\partial \Omega^e}{\partial \mathbf{W}} \leftarrow \mathcal{W}\left(\sigma, \mathbf{h}, \mathbf{D}, \mathbf{W}, \alpha, T\right)$$

This general interface defines the stress, history, and orientation rates used in integrating the single crystal model. The interface also allows the user to select which parts of the update are done in a coupled, implicit integration and which parts are left separate for an explicit, uncoupled integration.

Implementations

Currently there is only one implementation of this class, implementing the kinematic assumptions described in the *overview* of the crystal plasticity model. Other kinematics could be implemented in the future by deriving additional subclasses from this abstract base class.

StandardKinematicModel

Overview

This KinematicModel implements the kinematic assumptions described in the *overview* of the crystal plasticity model. The stress and rotation rates match those developed in that derivation. The history rate and variables are left to be generic, as are the exact form of the plastic deformation rate. These are defined by the

InelasticModel

Overview

Inelastic models provide the plastic deformation rate and internal variable history rate equations to the higher level objects, along with associated partial derivatives. The mathematical interface is:

$$\begin{aligned}
\mathbf{D}^p, \frac{\partial \mathbf{D}^p}{\partial \sigma}, \frac{\partial \mathbf{D}^p}{\partial \mathbf{h}} &\leftarrow \mathcal{D}\left(\sigma, \mathbf{h}, \alpha, T\right) \\
\mathbf{W}^p, \frac{\partial \mathbf{W}^p}{\partial \sigma}, \frac{\partial \mathbf{W}^p}{\partial \mathbf{h}} &\leftarrow \mathcal{W}\left(\sigma, \mathbf{h}, \alpha, T\right) \\
\dot{\mathbf{h}}, \frac{\partial \dot{\mathbf{h}}}{\partial \sigma}, \frac{\partial \dot{\mathbf{h}}}{\partial \mathbf{h}} &\leftarrow \mathcal{H}\left(\sigma, \mathbf{h}, \alpha, T\right)
\end{aligned}$$

where all these quantities are defined in the crystal plasticity *overview*. Note the implementations also have the crystallographic information available through the model's *Lattice* object.

Implementations

AsaroInelasticity

Overview

This class implements the standard crystal plasticity plastic deformation rate presented in [A1983]. The plastic deformation rate is the sum of simple shear deformations on each individual slip direction and plane:

$$\mathbf{L}^p = \sum_{g=1}^{n_{group}} \sum_{i=1}^{n_{slip}} \dot{\gamma}_{g,i} \left(\mathbf{d}_{g,i} \otimes \mathbf{n}_{g,i}\right)$$

where $\mathbf{d}_{g,i}$ are the slip directions and $\mathbf{n}_{g,i}$ are the slip normals *in the current frame* Ω. $\mathbf{D}^p$ and $\mathbf{W}^p$ are the symmetric and skew parts of this definition.

The slip rate, $\dot{\gamma}_{g,i}$, is a function of stress, temperature, and the internal variables. A separate class implements different models for the slip rate:

SlipRule

Overview

These objects provide a relation between the stress, history, and temperature and the slip rate on each individual slip system as well as the history evolution. The interface used is:

$$\dot{\gamma}_{g,i}, \frac{\partial \dot{\gamma}_{g,i}}{\partial \sigma}, \frac{\partial \dot{\gamma}_{g,i}}{\partial \mathbf{h}} \leftarrow \mathcal{G}\left(\sigma, \mathbf{h}, \alpha, T\right)$$
$$\dot{\mathbf{h}}, \frac{\partial \dot{\mathbf{h}}}{\partial \sigma}, \frac{\partial \dot{\mathbf{h}}}{\partial \mathbf{h}} \leftarrow \mathcal{H}\left(\sigma, \mathbf{h}, \alpha, T\right)$$

where g indicates the slip group and i indicates the system within the group.

Implementations

SlipStrengthSlipRule

Overview

These objects provide a relation between the stress, history, and temperature and the slip rate on each individual slip system where the slip rate is related to the resolved shear stress on the system

$$\tau_{g,i} = \sigma : \left(\mathbf{d}_{g,i} \otimes \mathbf{n}_{g,i}\right)$$

where $\mathbf{d}_{g,i}$ is the slip direction for group g, system *i in the current coordinates* and $\mathbf{n}_{g,i}$ is similarly the slip system normal. The interface used is:

$$\dot{\gamma}_{g,i}, \frac{\partial \dot{\gamma}_{g,i}}{\partial \tau_{g,i}}, \frac{\partial \dot{\gamma}_{g,i}}{\partial \bar{\tau}_{g,i}} \leftarrow \mathcal{G}\left(\tau_{g,i}, \bar{\tau}_{g,i}, \alpha, T\right)$$
$$\dot{\mathbf{h}}, \frac{\partial \dot{\mathbf{h}}}{\partial \sigma}, \frac{\partial \dot{\mathbf{h}}}{\partial \mathbf{h}} \leftarrow \mathcal{H}\left(\sigma, \mathbf{h}, \alpha, T\right)$$

where g indicates the slip group, i indicates the system within the group, and $\bar{\tau}_{g,i}$ is the slip system strength, defined by a SlipHardening model:

SlipHardening

Overview

These objects provide the slip system strength required to complete the constitutive description when using a SlipStrengthSlipRule. The implementation assumes that the slip system strengths are functions only of the set of history variables and temperature. Additionally, these objects provide the definition of the history evolution rate equations and, ultimately, the definition of the model internal variables.

The interface is then:

$$\bar{\tau}_{g,i}, \frac{\partial \bar{\tau}_{g,i}}{\partial \mathbf{h}} \leftarrow \mathcal{T}\left(\mathbf{h}, \alpha, T\right)$$
$$\dot{\mathbf{h}}, \frac{\partial \dot{\mathbf{h}}}{\partial \sigma}, \frac{\partial \dot{\mathbf{h}}}{\partial \mathbf{h}} \leftarrow \mathcal{H}\left(\sigma, \mathbf{h}, \alpha, T\right).$$

Implementations

SlipSingleHardening

Overview

This object degenerates the general SlipHardening model so that all slip groups and systems share the same scalar slip system strength. The actual internal variables remain generic.

This produces the interface

$$\bar{\tau}, \frac{\partial \bar{\tau}}{\partial \mathbf{h}} \leftarrow \mathcal{T}\left(\mathbf{h}, \alpha, T\right)$$
$$\dot{\mathbf{h}}, \frac{\partial \dot{\mathbf{h}}}{\partial \sigma}, \frac{\partial \dot{\mathbf{h}}}{\partial \mathbf{h}} \leftarrow \mathcal{H}\left(\sigma, \mathbf{h}, \alpha, T\right)$$

where $\bar{\tau}$ is now a single scalar.

Implementations

SlipSingleStrengthHardening

Overview

This object further degenerates the general SlipSingleHardening model to where the (single) slip system strength is equal to some static strength, a contribution from a single internal variable, defined by a scalar rate equation, and a contribution from the Nye tensor. If this scalar history variable is $\tilde{\tau}$ the history map is

$$\bar{\tau} = \tilde{\tau} + \tau_0 + \tau_{nye}$$

where τ_0 is some static strength that does not evolve with time and τ_{nye} is a function of the Nye tensor α. The Nye tensor contribution defaults to zero, which gives a classical slip hardening model.

The remaining interface must provide the evolution equation, and associated partial derivatives, of the scalar history variable:

$$\dot{\tilde{\tau}}, \frac{\partial \dot{\tilde{\tau}}}{\partial \sigma}, \frac{\partial \dot{\tilde{\tau}}}{\partial \tilde{\tau}}, \tau_0 \leftarrow \mathcal{H}\left(\sigma, \tilde{\tau}, \alpha, T\right).$$

Implementations

PlasticSlipHardening

Overview

This class performs yet another simplification of the SlipSingleStrengthHardening model so that the scalar history variable evolves only as a function of the variable itself, temperature, and the absolute sum of the slip rates on all the systems. That is

$$\dot{\tilde{\tau}} = f\left(\tilde{\tau}, T\right) \sum_{g=1}^{n_{groups}} \sum_{i=1}^{n_{slip}} \left|\dot{\gamma}_{g,i}\right|$$

The interface defines the function f, its partial derivative with respect to the history variable, and the static strength:

$$f, \frac{\partial f}{\partial \tilde{\tau}}, \tau_0 \leftarrow \mathcal{P}\left(\tilde{\tau}, T\right)$$

Implementations

VoceSlipHardening

Overview

This class implements a Voce hardening model with

$$\begin{aligned} \tau_0 &= \tau_0 \\ f &= b\left(\tau_{sat} - \tilde{\tau}\right) \\ \tau_{nye} &= k\sqrt{\|\alpha\|_F} \end{aligned}$$

where τ_s, b, τ_{sat}, and k all temperature-dependent material properties.

The value of k defaults to zero, which means by default the model is independent of the Nye tensor.

Parameters

Parameter	Object type	Description	Default
`tau_sat`	*neml::Interpolate*	Saturation strength	N
`b`	*neml::Interpolate*	Rate parameter	N
`tau_0`	*neml::Interpolate*	Static strength	N
`k`	*neml::Interpolate*	Nye hardening constant	0

Class description

class **VoceSlipHardening** : public *neml*::*PlasticSlipHardening*

Everyone's favorite Voce model.

Public Functions

VoceSlipHardening(*ParameterSet* &params)

Initialize with the saturated strength, the rate constant, and a constant strength.

virtual double **init_strength**() const

Setup the scalar.

virtual double **static_strength**(double T) const

Static strength.

virtual double **hist_factor**(double strength, *Lattice* &L, double T, const *History* &fixed) const

Prefactor.

virtual double **d_hist_factor**(double strength, *Lattice* &L, double T, const *History* &fixed) const

Derivative of the prefactor.

virtual bool **use_nye**() const

Dynamically determine if we're going to use the Nye tensor.

virtual double **nye_part**(const *RankTwo* &nye, double T) const

Actual nye contribution.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

LinearSlipHardening

Overview

This class implements linear hardening with

$$\begin{aligned} \tau_0 &= \tau_0 \\ f &= k_1 \\ \tau_{nye} &= k_2 \left\| \alpha \right\|_F \end{aligned}$$

where k_1 and k_2 are temperature-dependent material properties.

Parameters

Parameter	Object type	Description	Default
k_1	*neml::Interpolate*	Slip hardening coefficient	N
k_2	*neml::Interpolate*	Nye hardening coefficient	N

Class description

class **LinearSlipHardening** : public *neml*::*PlasticSlipHardening*

The simplest of all linear hardening models.

Public Functions

LinearSlipHardening(*ParameterSet* &params)

Initialize with the saturated strength, the rate constant, and a constant strength.

virtual double **init_strength**() const

Setup the scalar.

virtual double **static_strength**(double T) const

Static strength.

virtual double **hist_factor**(double strength, *Lattice* &L, double T, const *History* &fixed) const

Prefactor.

virtual double **d_hist_factor**(double strength, *Lattice* &L, double T, const *History* &fixed) const

Derivative of the prefactor.

virtual bool **use_nye**() const

Dynamically determine if we're going to use the Nye tensor.

virtual double **nye_part**(const *RankTwo* &nye, double T) const

Actual nye contribution.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

Class description

class **PlasticSlipHardening** : public *neml*::*SlipSingleStrengthHardening*

Slip strength rule where the single strength evolves with sum|dg|.

Subclassed by *neml::LinearSlipHardening*, *neml::VoceSlipHardening*

Public Functions

PlasticSlipHardening(*ParameterSet* &params)

virtual double **hist_rate**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Scalar evolution law.

virtual *Symmetric* **d_hist_rate_d_stress**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of scalar law wrt stress.

virtual *History* **d_hist_rate_d_hist**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of scalar law wrt the scalar.

virtual *History* **d_hist_rate_d_hist_ext**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed, std::vector<std::string> ext) const

Derivative of the scalar law wrt all other scalars.

virtual double **hist_factor**(double strength, *Lattice* &L, double T, const *History* &fixed) const = 0

Prefactor.

virtual double **d_hist_factor**(double strength, *Lattice* &L, double T, const *History* &fixed) const = 0

Derivative of the prefactor.

Class description

class **SlipSingleStrengthHardening** : public *neml*::*SlipSingleHardening*

Slip strength rule where all systems evolve on a single scalar strength.

Subclassed by *neml::PlasticSlipHardening*

Public Functions

SlipSingleStrengthHardening(*ParameterSet* &params)

virtual std::vector<std::string> **varnames**() const

Report varnames.

virtual void **set_varnames**(std::vector<std::string> vars)

Set new varnames.

virtual void **populate_hist**(*History* &history) const

Request whatever history you will need.

virtual void **init_hist**(*History* &history) const

Setup history.

virtual *History* **hist**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

The rate of the history.

virtual *History* **d_hist_d_s**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt stress.

virtual *History* **d_hist_d_h**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt the history.

virtual *History* **d_hist_d_h_ext**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed, std::vector<std::string> ext) const

Derivative of this history wrt the history, external variables.

virtual double **hist_map**(const *History* &history, double T, const *History* &fixed) const

The scalar map.

virtual *History* **d_hist_map**(const *History* &history, double T, const *History* &fixed) const

The derivative of the scalar map.

void **set_variable**(std::string name)

Set the variable's name.

virtual double **static_strength**(double T) const = 0

Static (not evolving) strength.

double **nye_contribution**(const *History* &fixed, double T) const

Nye contribution (defaults to zero)

virtual double **nye_part**(const *RankTwo* &nye, double T) const

Actual implementation of any Nye contribution (defaults to zero)

virtual double **init_strength**() const = 0

Setup the scalar.

virtual double **hist_rate**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const = 0

Scalar evolution law.

virtual *Symmetric* **d_hist_rate_d_stress**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const = 0

Derivative of scalar law wrt stress.

virtual *History* **d_hist_rate_d_hist**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const = 0

Derivative of scalar law wrt the scalar.

virtual *History* **d_hist_rate_d_hist_ext**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed, std::vector<std::string> ext) const = 0

Derivative of the scalar law wrt all others.

SumSlipSingleStrengthHardening

Overview

This model sums the contributions of multiple *SlipSingleStrengthHardening* models together, i.e.

$$\bar{\tau} = \sum_{i=1}^{n_{model}} \tilde{\tau}_i + \tau_{0,i} + \tau_{nye,i}$$

Note then that all parts of each model are summed, even the static strengths.

This class will rename the internal variables of the individual hardening models to avoid overlap in the History object.

Class description

class **SumSlipSingleStrengthHardening** : public *neml*::*SlipSingleHardening*

Sum of individual SlipSingleStrenghHardening models (static strengths also summed)

Public Functions

SumSlipSingleStrengthHardening(*ParameterSet* &params)

Initialize with a list of models.

virtual std::vector<std::string> **varnames**() const

Report varnames.

virtual void **set_varnames**(std::vector<std::string> vars)

Set new varnames.

virtual void **populate_hist**(*History* &history) const

Request whatever history you will need.

virtual void **init_hist**(*History* &history) const

Setup history.

virtual *History* **hist**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

The rate of the history.

virtual *History* **d_hist_d_s**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt stress.

virtual *History* **d_hist_d_h**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt the history.

virtual *History* **d_hist_d_h_ext**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed, std::vector<std::string> ext) const

Derivative of this history wrt the history, external variables.

virtual double **hist_map**(const *History* &history, double T, const *History* &fixed) const

The scalar map.

virtual *History* **d_hist_map**(const *History* &history, double T, const *History* &fixed) const

The derivative of the scalar map.

virtual bool **use_nye**() const

Whether this model uses the Nye tensor.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

Class description

class **SlipSingleHardening** : public *neml*::*SlipHardening*

Slip strength rules where all systems share the same strength.

Subclassed by *neml::SlipSingleStrengthHardening*, *neml::SumSlipSingleStrengthHardening*

Public Functions

SlipSingleHardening(*ParameterSet* &params)

virtual double **hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Map the set of history variables to the slip system hardening.

virtual *History* **d_hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Derivative of the map wrt to history.

virtual double **hist_map**(const *History* &history, double T, const *History* &fixed) const = 0

The scalar map.

virtual *History* **d_hist_map**(const *History* &history, double T, const *History* &fixed) const = 0

The derivative of the scalar map.

FixedStrengthHardening

Overview

This class implements the simplest hardening rule – all slip systems have a constant strength

$$\bar{\tau}_{g,i} = \tau_{g,i}$$

Parameters

Parameter	Object type	Description	Default
tau_sat	std::vector<*neml::Interpolat*	Unrolled slip system hardening vector	N

Class description

class **FixedStrengthHardening** : public *neml*::*SlipHardening*

Fixed strength.

Public Functions

FixedStrengthHardening(*ParameterSet* &params)

virtual std::vector<std::string> **varnames**() const

Report your variable names.

virtual void **set_varnames**(std::vector<std::string> vars)

Set new varnames.

virtual void **populate_hist**(*History* &history) const

Request whatever history you will need.

virtual void **init_hist**(*History* &history) const

Setup history.

virtual double **hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Map the set of history variables to the slip system hardening.

virtual *History* **d_hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Derivative of the map wrt to history.

virtual *History* **hist**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

The rate of the history.

virtual *History* **d_hist_d_s**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt stress.

virtual *History* **d_hist_d_h**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt the history.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

GeneralLinearHardening

Overview

General linear interaction hardening – each hardening variable evolves as a linear combination of the slip rates (or absolute value of the slip rates) of all other slip systems.

$$\dot{\tilde{\tau}}_k = \sum_{j=1}^{n_{total}} M_{k,j} \dot{\gamma}_j$$

or

$$\dot{\tilde{\tau}}_k = \sum_{j=1}^{n_{total}} M_{k,j} \left| \dot{\gamma}_j \right|$$

where the indices k and j are the unrolled indices corresponding to slip group and system numbers.

The initial model strengths are constants.

The *Matrix classes* classes provide the definition of the interaction matrix.

Parameters

Parameter	Object type	Description	Default
`M`	*`neml::SquareMatrix`*	Interaction matrix	N
`tau_0`	`std::vector<double>`	Initial strengths	N
`absvar`	`bool`	If true use absolute value slip rates	:c:type`true`

Class description

class **GeneralLinearHardening** : public *neml*::*SlipHardening*

Generic linear hardening of the form tau_i = tau_0_i + H.gamma.

Public Functions

GeneralLinearHardening(*ParameterSet* &params)

virtual std::vector<std::string> **varnames**() const

Report your variable names.

virtual void **set_varnames**(std::vector<std::string> vars)

Set new varnames.

virtual void **populate_hist**(*History* &history) const

Request whatever history you will need.

virtual void **init_hist**(*History* &history) const

Setup history.

virtual double **hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Map the set of history variables to the slip system hardening.

virtual *History* **d_hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Derivative of the map wrt to history.

virtual *History* **hist**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

The rate of the history.

virtual *History* **d_hist_d_s**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt stress.

virtual *History* **d_hist_d_h**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt the history.

virtual *History* **d_hist_d_h_ext**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed, std::vector<std::string> ext) const

Derivative of this history wrt the history, external variables.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

SimpleLinearHardening

Overview

A simple linear hardening model, commonly used to test features of more complex systems.

$$\tau_i = \tau_{0,i} + \sum_{j=1}^{n_{slip}} G_{ij}\gamma_j$$

where the indices i and j are the unrolled indices corresponding to slip group and system numbers.

The model maintains the integrated slips as the internal variables.

The *Matrix classes* classes provide the definition of the interaction matrix.

Parameters

Parameter	Object type	Description	Default
`G`	*neml::SquareMatrix*	Interaction matrix	N
`tau_0`	`std::vector<double>`	Initial strengths	N

Class description

class **SimpleLinearHardening** : public *neml*::*SlipHardening*

Simple linear hardening, using the accumulated slip.

Public Functions

SimpleLinearHardening(*ParameterSet* &params)

virtual std::vector<std::string> **varnames**() const

Report your variable names.

virtual void **set_varnames**(std::vector<std::string> vars)

Set new varnames.

virtual void **populate_hist**(*History* &history) const

Request whatever history you will need.

virtual void **init_hist**(*History* &history) const

Setup history.

virtual double **hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Map the set of history variables to the slip system hardening.

virtual *History* **d_hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Derivative of the map wrt to history.

virtual *History* **hist**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

The rate of the history.

virtual *History* **d_hist_d_s**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt stress.

virtual *History* **d_hist_d_h**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt the history.

virtual *History* **d_hist_d_h_ext**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed, std::vector<std::string> ext) const

Derivative of this history wrt the history, external variables.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

VocePerSystemHardening

Overview

This model provides a Voce hardening response on each individual slip system. The hardening evolution on each system is independent of all other systems and each slip system can have its own Voce parameters:

$$\dot{\bar{\tau}}_k = k_k \left(1 - \frac{\bar{\tau}_k - \tau_{0,k}}{\tau_{sat,k} - \tau_{0,k}}\right)^m$$

Parameters

Parameter	Object type	Description	Default
`k`	*std::vector*	Hardening prefactors	N
`saturation`	*std::vector*	Saturated hardening values	N
`m`	*std::vector*	Voce exponents	N
`initial`	`std::vector<double>`	Initial strengths	N

Class description

class **VocePerSystemHardening** : public *neml*::*SlipHardening*

Voce model with one hardening variable per system.

Public Functions

VocePerSystemHardening(*ParameterSet* &params)

virtual std::vector<std::string> **varnames**() const

Report your variable names.

virtual void **set_varnames**(std::vector<std::string> vars)

Set new varnames.

virtual void **populate_hist**(*History* &history) const

Request whatever history you will need.

virtual void **init_hist**(*History* &history) const

Setup history.

virtual double **hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Map the set of history variables to the slip system hardening.

virtual *History* **d_hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Derivative of the map wrt to history.

virtual *History* **hist**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

The rate of the history.

virtual *History* **d_hist_d_s**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt stress.

virtual *History* **d_hist_d_h**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt the history.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

FASlipHardening

Overview

This model provides an empirical Frederick-Armstrong type hardening response on each individual slip system. The hardening evolution on each system is independent of all other systems and each slip system can have its own parameters

$$\dot{\bar{\tau}}_k = k_k \left(\dot{\gamma}_k - \frac{\bar{\tau}_k}{\tau_{sat,k}} |\dot{\gamma}_k| \right)$$

Parameters

Parameter	Object type	Description	Default
k	*std::vector*	Hardening prefactors	N
sat	*std::vector*	Saturated hardening values	N

Class description

class **FASlipHardening** : public *neml*::*SlipHardening*

Frederick-Armstrong type hardening.

Public Functions

FASlipHardening(*ParameterSet* &params)

virtual std::vector<std::string> **varnames**() const

Report your variable names.

virtual void **set_varnames**(std::vector<std::string> vars)

Set new varnames.

virtual void **populate_hist**(*History* &history) const

Request whatever history you will need.

virtual void **init_hist**(*History* &history) const

Setup history.

virtual double **hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Map the set of history variables to the slip system hardening.

virtual *History* **d_hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Derivative of the map wrt to history.

virtual *History* **hist**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

The rate of the history.

virtual *History* **d_hist_d_s**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt stress.

virtual *History* **d_hist_d_h**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt the history.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

LANLTiModel

Overview

Forest dislocation density based hardening slip model, used to physically describe of dislocation dominant plastic deformation.

$$\tau_i = \tau^s_{0,i} + X^i b^i \mu^i \sqrt{\rho^i_{for}}$$

The forest dislocation density evloves with strain and varies with dislication trapping and annihilation rates of each slip system.

$$\dot{\rho^i_{for}} = k^i_1 \sqrt{\rho^i_{for}} - k^i_2 \rho^i_{for}$$

A pseudo-slip hardening model used to describe the resistance of the propagation of twinning.

$$\tau_i = \tau^t_{0,i} + \mu i \sum_{j=1}^{n_{slip}} C^{ij} b^i b^j \rho^j_{for}$$

where i is the unrolled indices corresponding to slip group.

Parameters

Parameter	Object type	Description	Default
`tau_0`	`std::vector<double>`	Initial strengths	N
`C_st`	*`neml::SquareMatrix`*	Twin-slip interaction matrix	N
`mu`	`std::vector<double>`	Elastic modulus on the system	N
`k1`	`std::vector<double>`	Coefficient for trapping of dislocation	N
`k2`	`std::vector<double>`	Coefficient for annihilation of dislocation	N
`X`	`double`	Dislocation interaction parameter	0.9
`inivalue`	`double`	Initial values of internal variables	1.0e-6

Class description

class **LANLTiModel** : public *neml*::*SlipHardening*

Public Functions

LANLTiModel(*ParameterSet* &params)

virtual std::vector<std::string> **varnames**() const

Report your variable names.

virtual void **set_varnames**(std::vector<std::string> vars)

Set new varnames.

virtual void **populate_hist**(*History* &history) const

Request whatever history you will need.

virtual void **init_hist**(*History* &history) const

Setup history.

virtual double **hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Map the set of history variables to the slip system hardening.

virtual *History* **d_hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Derivative of the map wrt to history.

virtual *History* **hist**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

The rate of the history.

virtual *History* **d_hist_d_s**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt stress.

virtual *History* **d_hist_d_h**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const

Derivative of the history wrt the history.

virtual *History* **d_hist_d_h_ext**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed, std::vector<std::string> ext) const

Derivative of this history wrt the history, external variables.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

Class description

class **SlipHardening** : public *neml*::HistoryNEMLObject

ABC for a slip hardening model.

Subclassed by *neml::DislocationSpacingHardening*, *neml::FASlipHardening*, *neml::FixedStrengthHardening*, neml::GeneralizedHuCocksHardening, *neml::GeneralLinearHardening*, *neml::HuCocksHardening*, *neml::LANLTiModel*, *neml::SimpleLinearHardening*, *neml::SlipSingleHardening*, *neml::VocePerSystemHardening*

Public Types

enum **CacheType**

Values:

enumerator **BLANK**

enumerator **DOUBLE**

Public Functions

SlipHardening(*ParameterSet* &params)

virtual std::vector<std::string> **varnames**() const = 0

Report your variable names.

virtual void **set_varnames**(std::vector<std::string> vars) = 0

Set new varnames.

virtual double **hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const = 0

Map the set of history variables to the slip system hardening.

virtual *History* **d_hist_to_tau**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const = 0

Derivative of the map wrt to history, model variables.

virtual *History* **d_hist_to_tau_ext**(size_t g, size_t i, const *History* &history, *Lattice* &L, double T, const *History* &fixed, std::vector<std::string> ext) const

Derivative of the map wrt to history, external variables.

virtual *History* **hist**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const = 0

The rate of the history.

virtual *History* **d_hist_d_s**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const = 0

Derivative of the history wrt stress.

virtual *History* **d_hist_d_h**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed) const = 0

Derivative of the history wrt the history, model variables.

virtual *History* **d_hist_d_h_ext**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *SlipRule* &R, const *History* &fixed, std::vector<std::string> ext) const

Derivative of this history wrt the history, external variables.

virtual bool **use_nye**() const

Whether this particular model uses the Nye tensor.

inline *History* **blank_hist**() const

Helper providing a blank (zero) history.

History **cache**(*CacheType* type) const

Helper accessing cached objects.

The definition of the history evolution is left to the SlipHardening model.

Implementations

PowerLawSlipRule

Overview

This implements the standard power law model for the slip rate:

$$\dot{\gamma}_{g,i} = \dot{\gamma}_0 \left| \frac{\tau_{g,i}}{\bar{\tau}_{g,i}} \right|^{(n-1)} \frac{\tau_{g,i}}{\bar{\tau}_{g,i}}$$

where $\dot{\gamma}_0$, the reference slip rate, and n, the rate sensitivity, are temperature-dependent parameters.

Parameters

Parameter	Object type	Description	Default
strength	*neml::SlipHardening*	Slip hardening definition	No
gamma0	*neml::Interpolate*	Reference slip rate	No
n	*neml::Interpolate*	Rate sensitivity	No

Class description

class **PowerLawSlipRule** : public *neml*::*SlipStrengthSlipRule*

The standard power law slip strength/rate relation.

Public Functions

PowerLawSlipRule(*ParameterSet* &params)

Initialize with the strength object, the reference strain rate, and the rate sensitivity

virtual double **scalar_sslip**(size_t g, size_t i, double tau, double strength, double T) const

The slip rate definition.

virtual double **scalar_d_sslip_dtau**(size_t g, size_t i, double tau, double strength, double T) const

Derivative of slip rate with respect to the resolved shear.

virtual double **scalar_d_sslip_dstrength**(size_t g, size_t i, double tau, double strength, double T) const

Derivative of the slip rate with respect to the strength.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

Parameters

Parameter	Object type	Description	Default
strength	*neml::SlipHardening*	Slip hardening definition	No

Class description

class **SlipStrengthSlipRule** : public *neml*::*SlipMultiStrengthSlipRule*

Class where all slip rules that give the system response proportional to some strength, which is in turn a function of the history

Subclassed by *neml::ArrheniusSlipRule*, *neml::PowerLawSlipRule*

Public Functions

SlipStrengthSlipRule(*ParameterSet* &params)

Initialize with the strength model.

virtual double **sslip**(size_t g, size_t i, double tau, std::vector<double> strengths, double T) const

The slip rate on group g, system i given the resolved shear, the strength, and temperature

virtual double **d_sslip_dtau**(size_t g, size_t i, double tau, std::vector<double> strengths, double T) const

Derivative of slip rate with respect to the resolved shear.

virtual std::vector<double> **d_sslip_dstrength**(size_t g, size_t i, double tau, std::vector<double> strengths, double T) const

Derivative of the slip rate with respect to the strengths.

virtual double **scalar_sslip**(size_t g, size_t i, double tau, double strength, double T) const = 0

The scalar equivalent of the slip rates.

virtual double **scalar_d_sslip_dtau**(size_t g, size_t i, double tau, double strength, double T) const = 0

Derivative of slip rate with respect to the resolved shear.

virtual double **scalar_d_sslip_dstrength**(size_t g, size_t i, double tau, double strength, double T) const = 0

Derivative of the slip rate with respect to the strength.

SlipMultiStrengthSlipRule

Overview

These objects provide a relation between the stress, history, and temperature and the slip rate on each individual slip system where the slip rate is related to the resolved shear stress on the system

$$\tau_{g,i} = \sigma : (\mathbf{d}_{g,i} \otimes \mathbf{n}_{g,i})$$

where $\mathbf{d}_{g,i}$ is the slip direction for group g, system *i in the current coordinates* and $\mathbf{n}_{g,i}$ is similarly the slip system normal. The interface used is:

$$\dot{\gamma}_{g,i}, \frac{\partial \dot{\gamma}_{g,i}}{\partial \tau_{g,i}}, \frac{\partial \dot{\gamma}_{g,i}}{\partial \bar{\tau}^j_{g,i}} \leftarrow \mathcal{G}\left(\tau_{g,i}, \bar{\tau}_{g,i}, \alpha, T\right)$$

$$\dot{\mathbf{h}}, \frac{\partial \dot{\mathbf{h}}}{\partial \sigma}, \frac{\partial \dot{\mathbf{h}}}{\partial \mathbf{h}} \leftarrow \mathcal{H}\left(\sigma, \mathbf{h}, \alpha, T\right)$$

where g indicates the slip group, i indicates the system within the group, and $\bar{\tau}^j_{g,i}$ is a collection of slip system strengths, defined by SlipHardening models:

The definition of the history evolution is left to the SlipHardening models.

The difference between this class and *SlipStrengthSlipRule* is that the slip system flow is proportional to multiple slip strength models, for example an isotropic and a kinematic strength, instead of a single flow strength model.

Implementations

KinematicPowerLawSlipRule

Overview

This implements multi strength flow rule model of the form:

$$\dot{\gamma}_{g,i} = \dot{\gamma}_0 \left\langle \frac{\left|\tau_{g,i} - \bar{\tau}_{g,i}^{back}\right| - \bar{\tau}_{g,i}^{iso}}{\bar{\tau}_{g,i}^{resistance}} \right\rangle^n \text{sign}\left(\tau_{g,i} - \bar{\tau}_{g,i}^{back}\right)$$

where $\dot{\gamma}_0$, the reference slip rate, and n, the rate sensitivity, are temperature-dependent parameters. *SlipHardening* models provide the back, isotropic, and flow resistance strengths.

Parameters

Parameter	Object type	Description	Default
backstrength	*neml::SlipHardening*	Back strength definition	No
isostrength	*neml::SlipHardening*	Isotropic strength definition	No
flowresistance	*neml::SlipHardening*	Flow resistance definition	No
gamma0	*neml::Interpolate*	Reference slip rate	No
n	*neml::Interpolate*	Rate sensitivity	No

Class description

class **KinematicPowerLawSlipRule** : public *neml*::*SlipMultiStrengthSlipRule*

Kinematic hardening type power law slip.

Public Functions

KinematicPowerLawSlipRule(*ParameterSet* &params)

A completely generic slip rule with a backstrength, a isostrength, and a flow resistance.

virtual double **sslip**(size_t g, size_t i, double tau, std::vector<double> strengths, double T) const

The slip rate on group g, system i given the resolved shear, the strength, and temperature

virtual double **d_sslip_dtau**(size_t g, size_t i, double tau, std::vector<double> strengths, double T) const

Derivative of slip rate with respect to the resolved shear.

virtual std::vector<double> **d_sslip_dstrength**(size_t g, size_t i, double tau, std::vector<double> strengths, double T) const

Derivative of the slip rate with respect to the strengths.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

Parameters

Parameter	Object type	Description	Default
`strength`	`std::vector<neml::SlipHardening>`	List of slip hardening rules	No

Class description

class **SlipMultiStrengthSlipRule** : public *neml*::*SlipRule*

Class relying on multiple strength models (each of which is a function of some history variables with corresponding evolution laws)

Subclassed by *neml::KinematicPowerLawSlipRule*, *neml::SlipStrengthSlipRule*

Public Functions

SlipMultiStrengthSlipRule(*ParameterSet* &params, std::vector<std::shared_ptr<*SlipHardening*>> strengths)

Initialize with the strength models.

size_t **nstrength**() const

Number of strengths.

virtual void **populate_hist**(*History* &history) const

Populate the history.

virtual void **init_hist**(*History* &history) const

Actually initialize the history.

virtual double **strength**(const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Helper for models that want an average strength.

virtual double **slip**(size_t g, size_t i, const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Slip rate on group g, system i.

virtual *Symmetric* **d_slip_d_s**(size_t g, size_t i, const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Derivative of slip rate with respect to stress.

virtual *History* **d_slip_d_h**(size_t g, size_t i, const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Derivative of slip rate with respect to history.

virtual *History* **hist_rate**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

History evolution equations.

virtual *History* **d_hist_rate_d_stress**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Derivative of the history rate with respect to stress.

virtual *History* **d_hist_rate_d_hist**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Derivative of the history rate with respect to the history.

virtual bool **use_nye**() const

Whether this model uses the Nye tensor.

virtual double **sslip**(size_t g, size_t i, double tau, std::vector<double> strengths, double T) const = 0

The slip rate on group g, system i given the resolved shear, the strength, and temperature

virtual double **d_sslip_dtau**(size_t g, size_t i, double tau, std::vector<double> strengths, double T) const = 0

Derivative of slip rate with respect to the resolved shear.

virtual std::vector<double> **d_sslip_dstrength**(size_t g, size_t i, double tau, std::vector<double> strengths, double T) const = 0

Derivative of the slip rate with respect to the strengths.

Class description

class **SlipRule** : public *neml*::HistoryNEMLObject

Abstract base class for a slip rule.

Subclassed by *neml::SlipMultiStrengthSlipRule*

Public Functions

SlipRule(*ParameterSet* &params)

virtual double **strength**(const *History* &history, *Lattice* &L, double T, const *History* &fixed) const = 0

Helper for models that want an average strength.

virtual double **slip**(size_t g, size_t i, const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const = 0

Slip rate on group g, system i.

virtual *Symmetric* **d_slip_d_s**(size_t g, size_t i, const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const = 0

Derivative of the slip rate with respect to stress.

virtual *History* **d_slip_d_h**(size_t g, size_t i, const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const = 0

Derivative of the slip rate with respect to history.

virtual *History* **hist_rate**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const = 0

History rate.

virtual *History* **d_hist_rate_d_stress**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const = 0

Derivative of the history rate with respect to the stress.

virtual *History* **d_hist_rate_d_hist**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const = 0

Derivative of the history rate with respect to the history.

double **sum_slip**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Calculate the sum of the absolute value of the slip rates.

Symmetric **d_sum_slip_d_stress**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Derivative of the sum of the absolute value of the slip rates wrt stress.

History **d_sum_slip_d_hist**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Derivative of the sum of the absolute value of the slip rate wrt history.

virtual bool **use_nye**() const

Whether this model uses the Nye tensor.

The implementation also defers the definition of the internal variables to the *SlipRule* object.

Parameters

Parameter	Object type	Description	Default
`rule`	*neml::SlipRule*	Slip rate and history definition	No

Class description

class **AsaroInelasticity** : public *neml*::*InelasticModel*

The classic crystal plasticity inelastic model, as described in the manual.

Public Functions

AsaroInelasticity(*ParameterSet* &params)

Provide a *SlipRule* defining the slip rate/strength relations.

virtual **~AsaroInelasticity**()

Destructor.

virtual void **populate_hist**(*History* &history) const

Populate the history, deferred to the *SlipRule*.

virtual void **init_hist**(*History* &history) const

Initialize the history with the starting values, deferred to the *SlipRule*.

virtual double **strength**(const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Helper for external models that want an average strength.

virtual *Symmetric* **d_p**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Symmetric part of the plastic deformation rate.

virtual *SymSymR4* **d_d_p_d_stress**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the symmetric part with respect to stress.

virtual *History* **d_d_p_d_history**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the symmetric part with respect to history.

virtual *History* **history_rate**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

History rate, deferred to the *SlipRule*.

virtual *History* **d_history_rate_d_stress**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the history rate with respect to stress.

virtual *History* **d_history_rate_d_history**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the history rate with respect to history.

virtual *Skew* **w_p**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Skew part of the plastic deformation rate.

virtual *SkewSymR4* **d_w_p_d_stress**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the skew part with respect to stress.

virtual *History* **d_w_p_d_history**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the skew part with respect to history.

virtual bool **use_nye**() const

Whether this model uses the Nye tensor.

inline const *SlipRule* &**slip_rule**() const

Access to the slip rule for other models to get detailed slip information.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from parameter set.

static *ParameterSet* **parameters**()

Default parameters.

NoInelasticity

Overview

This class, mostly for testing, implements a large-deformation linear-elastic model by setting

$$\mathbf{L}^p = \mathbf{0}.$$

The implementation does not require any internal variables.

Parameters

None

Class description

class **NoInelasticity** : public *neml*::*InelasticModel*

This model returns zero for the plastic deformation, resulting model would be linear-elastic

Public Functions

NoInelasticity(*ParameterSet* &params)

Don't need any parameters to return zero!

virtual **~NoInelasticity**()

Destructor.

virtual void **populate_hist**(*History* &history) const

Add history variables (none needed)

virtual void **init_hist**(*History* &history) const

Define initial history (none)

virtual double **strength**(const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Helper for external models that want an average strength.

virtual *Symmetric* **d_p**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Symmetric part of the plastic deformation = 0.

virtual *SymSymR4* **d_d_p_d_stress**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the symmetric part with respect to stress (=0)

virtual *History* **d_d_p_d_history**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the symmetric part with respect to history (null)

virtual *History* **history_rate**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

History rate (null, as there are no history variables)

virtual *History* **d_history_rate_d_stress**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the history rate with respect to stress (null)

virtual *History* **d_history_rate_d_history**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the history rate with respect to history (null)

virtual *Skew* **w_p**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Skew part of the plastic deformation rate.

virtual *SkewSymR4* **d_w_p_d_stress**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the skew part with respect to stress (=0)

virtual *History* **d_w_p_d_history**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the skew part with respect to history (null)

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from parameter set.

static *ParameterSet* **parameters**()

Default parameters.

PowerLawInelasticity

Overview

This class implements isotropic power-law perfect viscoplasticity with the form:

$$\mathbf{D}^p = A\sigma_{eq}^{n-1}\mathbf{s}$$
$$\mathbf{W}^p = \mathbf{0}$$

where

$$\sigma_{eq} = \sqrt{\frac{3}{2}\mathbf{s} : \mathbf{s}},$$
$$\mathbf{s} = \sigma - \frac{1}{3}\operatorname{tr}(\sigma)\mathbf{I},$$

and A and n are parameters. Note this is then the standard J_2 flow rule.

The implementation does not require any internal variables.

Parameters

Parameter	Object type	Description	Default
A	*neml::Interpolate*	Prefactor as a function of temperature	No
n	*neml::Interpolate*	Rate sensitivity as a function of temperature	No

Class description

class **PowerLawInelasticity** : public *neml*::*InelasticModel*

An isotropic power law creep model, typically combined with slip system models to represent diffusion

Public Functions

PowerLawInelasticity(*ParameterSet* &params)

virtual void **populate_hist**(*History* &history) const

Setup history variables (none used in this implementation)

virtual void **init_hist**(*History* &history) const

Initialize the history variables (n/a)

virtual double **strength**(const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Helper for external models that want an average strength.

virtual *Symmetric* **d_p**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Symmetric part of the deformation rate.

virtual *SymSymR4* **d_d_p_d_stress**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the symmetric part with respect to the stress.

virtual *History* **d_d_p_d_history**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the symmetric part with respect to the history (null here)

virtual *History* **history_rate**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

History rate (null)

virtual *History* **d_history_rate_d_stress**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the history with respect to the stress (null here)

virtual *History* **d_history_rate_d_history**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the history rate with respect to the history (null here)

virtual *Skew* **w_p**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Skew part of the deformation rate (zero here)

virtual *SkewSymR4* **d_w_p_d_stress**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the skew part with respect to stress (=0)

virtual *History* **d_w_p_d_history**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the skew part with respect to the history (null here)

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from parameter set.

static *ParameterSet* **parameters**()

Default parameters.

CombinedInelasticity

Overview

This class sums the plastic deformation rates from several InelasticModel objects into a single response. Mathematically then the plastic deformation becomes

$$\mathbf{D}^p = \sum_{i=1}^{n_{models}} \mathbf{D}_i^p$$

$$\mathbf{W}^p = \sum_{i=1}^{n_{models}} \mathbf{W}_i^p$$

where the i subscripts indicate the response of each individual model.

The history variables and associated history rates and derivatives from all the individual models are *concatenated* together. That is, the final set of history variables includes the history variables of all the individual models.

Parameters

Parameter	Object type	Description	Default
`models`	`std::vector<neml::InelasticModel>`	Individual models	No

Class description

class **`CombinedInelasticity`** : public *neml*::*InelasticModel*

Metamodel that combines the rates of several individual InelasticModels.

Public Functions

`CombinedInelasticity`(*ParameterSet* &params)

Initialize with the list of models.

virtual double **`strength`**(const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Helper for external models that want an average strength.

virtual void **`populate_hist`**(*History* &history) const

Setup all history variables.

virtual void **`init_hist`**(*History* &history) const

Initialize history with actual values.

virtual *Symmetric* **`d_p`**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Sum the symmetric parts of the plastic deformation rates.

virtual *SymSymR4* **`d_d_p_d_stress`**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Sum the derivatives of the symmetric part with respect to stress.

virtual *History* **d_d_p_d_history**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Concatenate the derivatives of the symmetric part with respect to history.

virtual *History* **history_rate**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Concatenate the history rates.

virtual *History* **d_history_rate_d_stress**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Concatenate the derivative of the history rates with respect to stress.

virtual *History* **d_history_rate_d_history**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Concatenate the derivative of the history rates with respect to history.

virtual *Skew* **w_p**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Sum the skew parts of the plastic deformation rates.

virtual *SkewSymR4* **d_w_p_d_stress**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Sum the derivatives of the skew parts with respect to the stress.

virtual *History* **d_w_p_d_history**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Concatenate the derivatives of the skew parts with respect to the history.

virtual bool **use_nye**() const

Whether this model uses the Nye tensor.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from parameter set.

static *ParameterSet* **parameters**()

Default parameters.

Class description

class **InelasticModel** : public *neml*::HistoryNEMLObject

A inelastic model supplying D_p and W_p.

Subclassed by *neml::AsaroInelasticity*, *neml::CombinedInelasticity*, *neml::NoInelasticity*, *neml::PowerLawInelasticity*

Public Functions

InelasticModel(*ParameterSet* &params)

virtual double **strength**(const *History* &history, *Lattice* &L, double T, const *History* &fixed) const = 0

Helper for external models that want an average strength.

virtual *Symmetric* **d_p**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

Symmetric part of the plastic deformation.

virtual *SymSymR4* **d_d_p_d_stress**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

Derivative of the symmetric part with respect to stress.

virtual *History* **d_d_p_d_history**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

Derivative of the symmetric part with respect to stress.

virtual *History* **history_rate**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

History rate.

virtual *History* **d_history_rate_d_stress**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

Derivative of the history rate with respect to stress.

virtual *History* **d_history_rate_d_history**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

Derivative of the history rate with respect to the history.

virtual *Skew* **w_p**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

Skew part of the plastic deformation rate.

virtual *SkewSymR4* **d_w_p_d_stress**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

Derivative of the skew part with respect to stress.

virtual *History* **d_w_p_d_history**(const *Symmetric* &stress, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

Derivative of the skew part with respect to history.

virtual bool **use_nye**() const

Whether this model uses the nye tensor.

class and associated objects.

Parameters

Parameter	Object type	Description	Default
emodel	*neml::LinearElasticModel*	Elasticity tensor	No
imodel	*neml::InelasticModel*	Definition of the plastic deformation rate	No

Class description

class **StandardKinematicModel** : public *neml*::*KinematicModel*

My standard kinematic assumptions, outlined in the manual.

Subclassed by *neml::DamagedStandardKinematicModel*

Public Functions

StandardKinematicModel(*ParameterSet* &params)

Initialize with elastic and inelastic models.

virtual void **populate_hist**(*History* &history) const

Populate a history object with the correct variables.

virtual void **init_hist**(*History* &history) const

Initialize the history object with the starting values.

virtual double **strength**(const *History* &history, *Lattice* &L, double T, const *History* &fixed) const

Helper for external models that want a strength.

virtual *History* **decouple**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed)

Anything added to this *History* instance will be kept fixed in the implicit integration

virtual *Symmetric* **stress_rate**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Stress rate.

virtual *SymSymR4* **d_stress_rate_d_stress**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of stress rate with respect to stress.

virtual *SymSymR4* **d_stress_rate_d_d**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the stress rate with respect to the deformation rate.

virtual *SymSkewR4* **d_stress_rate_d_w**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the stress rate with respect to the vorticity.

virtual *History* **d_stress_rate_d_history**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the stress rate with respect to the history.

virtual *History* **history_rate**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

History rate.

virtual *History* **d_history_rate_d_stress**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the history rate with respect to the stress.

virtual *History* **d_history_rate_d_d**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the history rate with respect to the deformation rate.

virtual *History* **d_history_rate_d_w**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the history rate with respect to the vorticity.

virtual *History* **d_history_rate_d_history**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

Derivative of the history rate with respect to the history.

virtual *SymSkewR4* **d_stress_rate_d_w_decouple**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed)

Derivative of the stress rate with respect to the vorticity keeping fixed variables fixed

virtual *Skew* **spin**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const

The spin rate.

virtual *Symmetric* **elastic_strains**(const *Symmetric* &stress, *Lattice* &lattice, const *Orientation* &Q, const *History* &history, double T)

Helper to calculate elastic strains.

virtual *Symmetric* **stress_increment**(const *Symmetric* &stress, const *Symmetric* &D, const *Skew* &W, double dt, *Lattice* &lattice, const *Orientation* &Q, const *History* &history, double T)

Helper to predict an elastic stress increment.

virtual bool **use_nye**() const

Whether this model uses the Nye tensor.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from parameter set.

static *ParameterSet* **parameters**()

Default parameters.

Class description

class **KinematicModel** : public *neml*::HistoryNEMLObject

Describes the stress, history, and rotation rates.

Subclassed by *neml::StandardKinematicModel*

Public Functions

KinematicModel(*ParameterSet* &params)

virtual double **strength**(const *History* &history, *Lattice* &L, double T, const *History* &fixed) const = 0

Helper for external models that want a strength.

virtual *History* **decouple**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) = 0

Hook to allow user to decouple parts of the update Any items added to the *History* object will be treated explicitly

virtual *Symmetric* **stress_rate**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

Stress rate.

virtual *SymSymR4* **d_stress_rate_d_stress**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

Derivative of stress rate with respect to stress.

virtual *SymSymR4* **d_stress_rate_d_d**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

Derivative of the stress rate with respect to the deformation rate.

virtual *SymSkewR4* **d_stress_rate_d_w**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

Derivative of the stress rate with respect to the vorticity.

virtual *History* **d_stress_rate_d_history**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

Derivative of the stress rate with respect to the history.

virtual *History* **history_rate**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

History rate.

virtual *History* **d_history_rate_d_stress**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

Derivative of the history rate with respect to the stress.

virtual *History* **d_history_rate_d_d**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

Derivative of the history rate with respect to the deformation rate.

virtual *History* **d_history_rate_d_w**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

Derivative of the history rate with respect to the vorticity.

virtual *History* **d_history_rate_d_history**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

Derivative of the history rate with respect to the history.

virtual *SymSymR4* **d_stress_rate_d_d_decouple**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed)

Derivative of the stress rate with respect to the deformation keeping fixed variables fixed.

virtual *SymSkewR4* **d_stress_rate_d_w_decouple**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed)

Derivative of the stress rate with respect to the vorticity keeping fixed variables fixed

virtual *History* **d_history_rate_d_d_decouple**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed)

Derivative of the history rate with respect to the deformation rate keeping fixed variables in fixed constant

virtual *History* **d_history_rate_d_w_decouple**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed)

Derivative of the history rate with respect to the vorticity keeping fixed variables in fixed constant

virtual *Skew* **spin**(const *Symmetric* &stress, const *Symmetric* &d, const *Skew* &w, const *Orientation* &Q, const *History* &history, *Lattice* &lattice, double T, const *History* &fixed) const = 0

The spin rate.

virtual *Symmetric* **elastic_strains**(const *Symmetric* &stress, *Lattice* &lattice, const *Orientation* &Q, const *History* &history, double T) = 0

Helper to calculate elastic strains.

virtual *Symmetric* **stress_increment**(const *Symmetric* &stress, const *Symmetric* &D, const *Skew* &W, double dt, *Lattice* &lattice, const *Orientation* &Q, const *History* &history, double T) = 0

Helper to predict an elastic stress increment.

virtual bool **use_nye**() const

Whether this model uses the Nye tensor.

16.8.2 Crystallography classes

Overview

These classes provide basic crystallographic information for the crystal plasticity system. There are two key classes:

Lattice

Overview

The Lattice class provides slip system information, given the lattice vectors and symmetry group describing the crystal system. The class is intelligent enough to automatically generate the complete set of slip group direction and normal vectors given the direction and planes in Miller indices.

NEML divides slip systems into crystallographically identical groups, i.e. all systems generated by the combination of a particular slip direction and normal. The documentation uses the index g to indicate the slip group and the index i to indicate the particular slip system.

Subclasses

Lattice subclasses specialize the general form to particular types of crystal systems, eliminating the need for the user to explicitly provide the lattice vectors and symmetry group.

GeneralLattice

Overview

A general *Lattice* class where the user can provide the lattice directions and the symmetry class.

Parameters

Parameter	Object type	Description	Default
a1	std::vector<double>	First lattice vector	No
a2	std::vector<double>	Second lattice vector	No
a3	std::vector<double>	Third lattice vector	No
symmetry_group	*neml::SymmetryGroup*	Symmetry group	No
slip_systems	list_systems	Initial list of slip systems	{}
twin_systems	list_systems	Initial list of twin systems	{}

Class description

class **GeneralLattice** : public *neml*::*NEMLObject*, public *neml*::*Lattice*

General lattice with no specialization.

CubicLattice

Overview

The *Lattice* class, specialized to cubic crystal systems.

Parameters

Parameter	Object type	Description	Default
a	double	Lattice parameter	No
slip_systems	list_systems	Initial list of slip systems	{}
twin_systems	list_systems	Initial list of twin systems	{}

Class description

class **CubicLattice** : public *neml*::*NEMLObject*, public *neml*::*Lattice*

HCPLattice

Overview

This class specializes the generic crystal lattice class for HCP systems. Additionally, it overrides the base class methods which take Miller indices as inputs to take Miller-Bravais indices instead.

Parameters

Parameter	Object type	Description	Default
a	double	Lattice parameter a	No
c	double	Lattice parameter c	No
slip_systems	list_systems	Initial list of slip systems	{}
twin_systems	list_systems	Initial list of twin systems	{}

Class description

class **HCPLattice** : public *neml*::*NEMLObject*, public *neml*::*Lattice*

Parameters

Parameter	Object type	Description	Default
a1	*neml::Vector*	First lattice vector	No
a2	*neml::Vector*	Second lattice vector	No
a3	*neml::Vector*	Third lattice vector	No
symmetry	*neml::SymmetryGroup*	Crystal symmetry group	No
isystems	list_systems	Initial list of slip systems	{}

Class description

class **Lattice**

Subclassed by *neml::CubicLattice*, *neml::GeneralLattice*, *neml::HCPLattice*

Public Types

enum **SlipType**

Type: slip or twin.

Values:

enumerator **Slip**

enumerator **Twin**

Public Functions

Lattice(*Vector* a1, *Vector* a2, *Vector* a3, std::shared_ptr<*SymmetryGroup*> symmetry, list_systems isystems = { }, twin_systems tsystems = { })

Initialize with the three lattice vectors, the symmetry group and (optionally) a initial list of slip systems

inline const *Vector* &**a1**()

First lattice vector.

inline const *Vector* &**a2**()

Second lattice vector.

inline const *Vector* &**a3**()

Third lattice vector.

inline const *Vector* &**b1**()

First reciprocal vector.

inline const *Vector* &**b2**()

Second reciprocal vector.

inline const *Vector* &**b3**()

Third reciprocal vector.

inline const std::vector<std::vector<*Vector*>> &**burgers_vectors**()

Return the list of burgers vectors.

inline const std::vector<std::vector<*Vector*>> &**slip_directions**()

Return the list of normalized slip directions.

inline const std::vector<std::vector<*Vector*>> &**slip_planes**()

Return the list of normalized slip normals.

inline const std::vector<double> **characteristic_shears**()

Return the list of characteristic shears.

inline const std::vector<*SlipType*> &**slip_types**()

Return the list of slip system types.

virtual *Vector* **miller2cart_direction**(std::vector<int> m)

Convert Miller directions to cartesian vectors.

virtual *Vector* **miller2cart_plane**(std::vector<int> m)

Convert Miller planes to cartesian normal vectors.

std::vector<*Vector*> **equivalent_vectors**(*Vector* v)

Find all sets of equivalent vectors (+/- different)

std::vector<*Vector*> **equivalent_vectors_bidirectional**(*Vector* v)

Find all all sets of equivalent vectors (+/- the same)

virtual void **add_slip_system**(std::vector<int> d, std::vector<int> p)

Add a slip system given the Miller direction and plane.

virtual void **add_twin_system**(std::vector<int> eta1, std::vector<int> K1, std::vector<int> eta2, std::vector<int> K2)

Add a twin system given the twin direction and plane.

size_t **ntotal**() const

Number of total slip systems.

size_t **ngroup**() const

Number of groups of slip systems.

size_t **nslip**(size_t g) const

Number of slip systems in group g.

size_t **flat**(size_t g, size_t i) const

Flat index of slip group g, system i.

SlipType **slip_type**(size_t g, size_t i) const

Type of system: slip or twin.

double **characteristic_shear**(size_t g, size_t i) const

Characteristic shear (0 for slip systems)

Orientation **reorientation**(size_t g, size_t i) const

Twin reorientation operator (identity for slip systems)

double **burgers**(size_t g, size_t i) const

Norm of the Burgers vector for a particular system.

const *Symmetric* &**M**(size_t g, size_t i, const *Orientation* &Q)

Return the sym(d x n) tensor for group g, system i, rotated with Q.

const *Skew* &**N**(size_t g, size_t i, const *Orientation* &Q)

Return the skew(d x n) tensor for group g, system i, rotated with Q.

double **shear**(size_t g, size_t i, const *Orientation* &Q, const *Symmetric* &stress)

Calculate the resolved shear stress on group g, system i, rotated with Q given the stress

Symmetric **d_shear**(size_t g, size_t i, const *Orientation* &Q, const *Symmetric* &stress)

Calculate the derivative of the resolved shear stress on group g, system i, rotated with Q, given the stress

const std::shared_ptr<*SymmetryGroup*> **symmetry**()

Access the symmetry operations.

const std::vector<*Vector*> **unique_planes**() const

Return a list of Cartesian vectors giving the unique slip planes.

size_t **nplanes**() const

The number of unique slip planes.

size_t **plane_index**(size_t g, size_t i) const

Given a slip system return the index into the unique slip planes.

std::vector<std::pair<size_t, size_t>> **plane_systems**(size_t i) const

Given the unique slip plane index return the vector of (g,i) tuples.

SymmetryGroup

Overview

This class represents the symmetry operations associated with a crystallographic point group. Essentially, it is an interface to a list of quaternion symmetry operations. These operations have been hardcoded and verified using an automated unit test.

Parameters

Parameter	Object type	Description	Default
`sclass`	`string`	Point group in Hermann-Mauguin notation	No

Class description

class **SymmetryGroup** : public *neml*::*NEMLObject*

Public Functions

SymmetryGroup(*ParameterSet* &params)

Initialize with the Hermann-Mauguin notation as a string.

const std::vector<*Orientation*> &**ops**() const

Quaternion symmetry operators.

size_t **nops**() const

Number of symmetry operators.

Orientation **misorientation**(const *Orientation* &a, const *Orientation* &b) const

Find the minimum misorientation transformation between a and b.

std::vector<*Orientation*> **misorientation_block**(const std::vector<*Orientation*> &A, const std::vector<*Orientation*> &B)

Find the disorientation in a blocked way that trades memory for cpu.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from parameter set.

static *ParameterSet* **parameters**()

Default parameters.

The *SymmetryGroup* class provides the symmetry operators for a particular point group. The *Lattice* class provides information about a particular crystal system, defined by a set of lattice vectors, a point group, and a set of slip systems. This information includes the cartesian slip system normal and direction vectors and the resolved shear stress calculation needed by other objects in the crystal plasticity module.

16.9 Parameters

Parameter	Object type	Description	Default
kinematics	*neml::KinematicModel*	Model kinematics	No
lattice	*neml::Lattice*	Crystallography information	No
initial_angle	*neml::Orientation*	Initial crystal orientation	Identity
update_rotation	bool	Evolve the crystal orientation	true
tol	double	Nonlinear solver relative tolerance	1.0e-8
miter	int	Maximum nonlinear solver iterations	30
verbose	bool	Print lots of debug messages	false
max_divide	int	Maximum number of adaptive integration subdivision	6

16.10 Class description

class **SingleCrystalModel** : public *neml*::*NEMLModel_ldi*, public *neml*::*Solvable*

Single crystal model integrator.

Public Functions

SingleCrystalModel(*ParameterSet* &params)

Raw constructor.

virtual void **populate_state**(*History* &history) const

Setup blank history.

virtual void **init_state**(*History* &history) const

Actually initialize history.

virtual void **populate_static**(*History* &history) const

Not changing state.

virtual void **init_static**(*History* &history) const

Static stat.

double **strength**(double *const hist, double T) const

Useful methods for external models that want an idea of an average strength

void **Fe**(double *const stress, double *const hist, double T, double *Fe) const

Used to calculate the Nye tensor.

virtual void **update_ld_inc**(const double *const d_np1, const double *const d_n, const double *const w_np1, const double *const w_n, double T_np1, double T_n, double t_np1, double t_n, double *const s_np1, const double *const s_n, double *const h_np1, const double *const h_n, double *const A_np1, double *const B_np1, double &u_np1, double u_n, double &p_np1, double p_n)

Large deformation incremental update.

virtual double **alpha**(double T) const

Instantaneous CTE.

virtual void **elastic_strains**(const double *const s_np1, double T_np1, const double *const h_np1, double *const e_np1) const

Helper to calculate the elastic strain.

virtual size_t **nparams**() const

Number of nonlinear equations to solve in the integration.

virtual void **init_x**(double *const x, *TrialState* *ts)

Setup an initial guess for the nonlinear solution.

virtual void **RJ**(const double *const x, *TrialState* *ts, double *const R, double *const J)

Integration residual and jacobian equations.

Orientation **get_active_orientation**(double *const hist) const

Get the current orientation in the active convention (raw ptr history)

Orientation **get_active_orientation**(const *History* &hist) const

Get the current orientation in the active convention.

Orientation **get_passive_orientation**(double *const hist) const

Get the current orientation in the passive convention (raw ptr history)

Orientation **get_passive_orientation**(const *History* &hist) const

Get the current orientation in the passive convention.

void **set_active_orientation**(double *const hist, const *Orientation* &q)

Set the current orientation given an active rotation (crystal to lab)

void **set_active_orientation**(*History* &hist, const *Orientation* &q)

Set the current orientation given an active rotation (crystal to lab)

void **set_passive_orientation**(double *const hist, const *Orientation* &q)

Set the current orientation given a passive rotation (lab to crystal)

void **set_passive_orientation**(*History* &hist, const *Orientation* &q)

Set the current orientation given a passive rotation (lab to crystal)

virtual bool **use_nye**() const

Whether this model uses the nye tensor.

void **update_nye**(double *const hist, const double *const nye) const

Actually update the Nye tensor.

Public Static Functions

static std::string **type**()

Type for the object system.

static *ParameterSet* **parameters**()

Parameters for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Setup from a *ParameterSet*.

CHAPTER

SEVENTEEN

WALKER ALLOY 617 MODEL SUBSYSTEM

This subsystem implements the constitutive model described by Sham and Walker [SW2008], aimed at capturing the long-term response of Alloy 617. The subsystem also prototypes merging the *History object system* system for maintaining named and tagged internal history variables with the base constitutive model system using flat vectors.

The subsystem contains *Viscoplastic flow rule*, *Hardening models*, and several dedicated submodels representing special functions in Walker's model. This document provides a description of the entire subsystem. Later, the history-wrapped models will replace the current flat vector system and this documentation can be merged into the main NEML module documentation.

17.1 Mathematical description

This description is extracted from an Argonne National Laboratory technical report describing the implementation of the model [MS2020].

17.1.1 Basic viscoplastic response

The model starts with the basic inelastic stress rate equation:

$$\dot{\sigma} = \boldsymbol{C} : (\dot{\varepsilon} - \dot{\varepsilon}_{vp} - \dot{\varepsilon}_{th})$$

with $\dot{\sigma}$ the stress rate, $\boldsymbol{C}$ an isotropic elasticity tensors described by Young's modulus E and Poisson's ratio ν, $\dot{\varepsilon}$ the total (applied) strain rate, $\dot{\varepsilon}_{vp}$ is the viscoplastic strain rate described below, and $\dot{\varepsilon}_{th}$ is the thermal strain rate given by

$$\dot{\varepsilon}_{th} = \alpha \dot{T} \boldsymbol{I}$$

with α the instantaneous coefficient of thermal expansion, $\dot{T}$ the temperature rate, and $\boldsymbol{I}$ the identity tensor.

The model definition reuses several common functions. These functions appear several places in the formulation, occasionally with the same parameters and occasionally with different parameters depending on the location. Where needed this exposition distinguishes the function parameters using parenthetical superscripts, e.g. $x^{(a)}$.

17.1.2 Temperature scaling function

The temperature scaling function:

$$\chi\left(T\right) = \frac{\exp\left(-\frac{Q}{R_{gas}T}\right)}{\exp\left(-\frac{Q}{R_{gas}T_{ref}}\right)}$$

with Q an activation energy, R_{gas} the gas constant, and T_{ref} a reference temperature, with temperature in Kelvin, is reused several places in the model. The thermal scaling constants Q and T_{ref} remain the same with each appearance and so no superscripts are required.

17.1.3 Strain softening/tertiary creep function

The tertiary creep function

$$\Phi = 1 + \phi_0 p^{\phi_1}$$

with p the equivalent plastic strain and ϕ_0 and ϕ_1 temperature-dependent constants likewise appears several times in the model. Different components of the model uses different parameters ϕ_0 and ϕ_1, differentiated by superscripts.

17.1.4 The viscoplastic strain rate

The viscoplastic strain rate is

$$\dot{\varepsilon}_{vp} = \dot{p}\boldsymbol{g}$$

where $\dot{p}$ is the scalar plastic strain rate and $\boldsymbol{g}$ the flow direction. The flow direction is

$$\boldsymbol{g} = \sqrt{\frac{3}{2}}\frac{\boldsymbol{s} - \boldsymbol{X}}{\|\boldsymbol{s} - \boldsymbol{X}\|}$$

with $\boldsymbol{s}$the deviatoric part of the stress, $s = \mathrm{dev}\left(\sigma\right)$ and $\boldsymbol{X}$the backstress defined below. In this expression and in the equations below the tensor norm is defined as

$$\|\boldsymbol{Y}\| = \sqrt{\boldsymbol{Y} : \boldsymbol{Y}}$$

with : indicating double contraction.

The scalar strain rate is

$$\dot{p} = \dot{\varepsilon}_0 \Phi^{(p)}\left(p\right) \chi\left(T\right) F$$

with $\dot{\varepsilon}_0$, $\Phi^{(p)}\left(p\right)$ a softening function defined by constants $\phi_0^{(p)}$ and $\phi_1^{(p)}$, $\chi\left(T\right)$ the temperature scaling function, and F the flow function

$$F = \left\langle \frac{\sqrt{3/2}\left\|\boldsymbol{s} - \boldsymbol{X}\right\| - Y}{D} \right\rangle^n$$

with D an internal variable defined later, n a parameter, $\langle\rangle$ the Macaulay brackets, and Y the threshold stress given as

$$Y = \left(k + R\right)\left(\frac{D - D_0}{D_\xi}\right)^m$$

with k, D_0, D_ξ, and m parameters and R an internal variable defined below.

17.1.5 Isotropic hardening

The isotropic hardening variable evolves as

$$\dot{R} = r_0 \left(R_\infty - R\right) \dot{p} + r_1 \left(R_0 - R\right) \left|R_0 - R\right|^{r_2 - 1}$$

where r_0, R_∞, r_1, R_0, and r_2 are all parameters. The initial value of the isotropic hardening parameter is zero.

17.1.6 Kinematic hardening

The net backstress is the sum of three individual backstress terms:

$$\boldsymbol{X} = \sum_{i=1}^{3} \boldsymbol{X}_i.$$

$$The evolution equation for each individual backstress is$$

$$\dot{\boldsymbol{X}} = \frac{2}{3} c\left(p, \dot{p}\right) \dot{\varepsilon}_{vp} - \frac{c\left(p, \dot{p}_0\right)}{L\left(p\right)} \dot{p} \boldsymbol{b} - \chi\left(T\right) x_0 \Phi^{(x)}\left(p\right) \left(\sqrt{\frac{3}{2}} \frac{\|\boldsymbol{X}\|}{D}\right)^{x_1} \frac{\boldsymbol{X}}{\|\boldsymbol{X}\|}$$

where x_0 and x_1 are parameters, $c\left(p, \dot{p}\right)$, is a function defined below of the the equivalent plastic strain, p and either the actual plastic strain $\dot{p}$ or a reference plastic strain rate $\dot{p}_0$, $\Phi^{(x)}\left(p\right)$ is a softening function with independent parameters,

$$L\left(p\right) = l\left(l_1 + \left(1 - l_1\right) \exp\left[-l_0 p\right]\right)$$

with l, l_0, and l_1 parameter, and

$$\boldsymbol{b} = \left(1 - b_0\right) \boldsymbol{X} + \frac{2}{3} b_0 \left(\boldsymbol{n} \otimes \boldsymbol{n}\right) : \boldsymbol{X}$$

with b_0 a parameter and

$$\boldsymbol{n} = \sqrt{\frac{3}{2}} \frac{\boldsymbol{s} - \boldsymbol{X}}{\|\boldsymbol{s} - \boldsymbol{X}\|}.$$

In these expressions the outer product symbol $\otimes$ between two rank two tensors denotes the product given in index notation as

$$\boldsymbol{a} \otimes \boldsymbol{b} = a_{ij} b_{kl}.$$

The backstresses all start at $\boldsymbol{X} = \boldsymbol{0}$.

Walker's original model defined

$$c\left(p, \dot{p}\right) = \left\{c_0 + c_1 \dot{p}^{1/c_2}\right\} \Omega\left(p\right)$$

with the Ω function of the equivalent plastic strain given as

$$\Omega\left(p\right) = 1 + \left(\frac{D - D_0}{D_\xi}\right)^{\omega_0} \omega\left(p\right) \left(\omega_1 - 1\right) \exp\left(-\omega_2 q\right)$$

with ω_0, ω_1, and ω_2 parameters. The function $\omega\left(p\right)$ is defined as

$$\omega\left(p\right) = \omega_3 + \left(1 - \omega_3\right) \exp\left[-\omega_4 p\right]$$

with ω_3 and ω_4 additional parameters and q is an additional internal variable with evolution equation

$$\dot{q} = \dot{p} - \chi\left(T\right) q_0 q$$

where q_0 is a parameter and $q\left(0\right) = 0$. The function $\Omega\left(p\right)$ and the associated internal variable describe the stress overshoot in the cyclic tests

This implementation omits the overshoot part of the model, leaving

$$c\left(\dot{p}\right) = c_0 + c_1 \dot{p}^{1/c_2}$$

with c_0, c_1, and c_2 parameters. Depending on the location of the c function (in the hardening or dynamic recovery terms), this function is either invoked with the actual plastic strain rate $\dot{p}$ or with some constant rate $\dot{p}_0$, which is a model parameter.

The subsequent tables differentiate the parameters for each backstress using superscripted indices.

17.1.7 Drag stress evolution

The drag stress evolves as

$$\dot{D} = d_0 \left(1 - \frac{D - D_0}{D_\xi}\right) \dot{p} - \chi\left(T\right) \Phi^{(D)}\left(p\right) d_1 \left(D - D_0\right)^{d_2}$$

where d_0, D_ξ, d_1, and d_2 are parameters, and $\Phi^{(D)}\left(p\right)$ is a softening function with independent coefficients.

CHAPTER

EIGHTEEN

NEML IMPLEMENTATION

The implementation includes a wrapper for the full history subsystem and the implementation within the wrapper of the walker flow rule and a simple test flow rule, for debugging the wrapper functions.

18.1 Viscoplastic model wrapper

18.1.1 Overview

This class wraps the standard *Viscoplastic general flow rule* class to work with the *History object system* variable system, which associates names and types with internal variables.

18.1.2 Parameters

Abstract base class

18.1.3 Class description

class **WrappedViscoPlasticFlowRule** : public *neml*::*ViscoPlasticFlowRule*

Wrapper between *ViscoPlasticFlowRule* and a version using the "fancy" objects.

Subclassed by *neml::TestFlowRule*, *neml::WalkerFlowRule*

Public Functions

WrappedViscoPlasticFlowRule(*ParameterSet* &params)

Default constructor sets up the structs for the conversion.

virtual void **y**(const double *const s, const double *const alpha, double T, double &yv) const

Scalar inelastic strain rate.

virtual void **y**(const State &state, double &res) const = 0

Wrapped scalar inelastic strain rate.

virtual void **dy_ds**(const double *const s, const double *const alpha, double T, double *const dyv) const

Derivative of y wrt stress.

virtual void **dy_ds**(const State &state, *Symmetric* &res) const = 0
Wrapped derivative of y wrt stress.

virtual void **dy_da**(const double *const s, const double *const alpha, double T, double *const dyv) const
Derivative of y wrt history.

virtual void **dy_da**(const State &state, *History* &res) const = 0
Wrapped derivative of y wrt history.

virtual void **g**(const double *const s, const double *const alpha, double T, double *const gv) const
Flow rule proportional to the scalar strain rate.

virtual void **g**(const State &state, *Symmetric* &res) const = 0
Wrapped flow rule proportional to the scalar strain rate.

virtual void **dg_ds**(const double *const s, const double *const alpha, double T, double *const dgv) const
Derivative of g wrt stress.

virtual void **dg_ds**(const State &state, *SymSymR4* &res) const = 0
Wrapped derivative of g wrt stress.

virtual void **dg_da**(const double *const s, const double *const alpha, double T, double *const dgv) const
Derivative of g wrt history.

virtual void **dg_da**(const State &state, *History* &res) const = 0
Wrapped derivative of g wrt history.

virtual void **h**(const double *const s, const double *const alpha, double T, double *const hv) const
Hardening rule proportional to the scalar strain rate.

virtual void **h**(const State &state, *History* &res) const = 0
Wrapped hardening rule proportional to the scalar strain rate.

virtual void **dh_ds**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h wrt stress.

virtual void **dh_ds**(const State &state, *History* &res) const = 0
Wrapped derivative of h wrt stress.

virtual void **dh_da**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h wrt history.

virtual void **dh_da**(const State &state, *History* &res) const = 0
Wrapped derivative of h wrt history.

virtual void **h_time**(const double *const s, const double *const alpha, double T, double *const hv) const
Hardening rule proportional to time.

virtual void **h_time**(const State &state, *History* &res) const
Wrapped hardening rule proportional to time.

virtual void **dh_ds_time**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h_time wrt stress.

virtual void **dh_ds_time**(const State &state, *History* &res) const
Wrapped derivative of h_time wrt stress.

virtual void **dh_da_time**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h_time wrt history.

virtual void **dh_da_time**(const State &state, *History* &res) const
Wrapped derivative of h_time wrt history.

virtual void **h_temp**(const double *const s, const double *const alpha, double T, double *const hv) const
Hardening rule proportional to temperature rate.

virtual void **h_temp**(const State &state, *History* &res) const
Wrapped hardening rule proportional to temperature rate.

virtual void **dh_ds_temp**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h_temp wrt stress.

virtual void **dh_ds_temp**(const State &state, *History* &res) const
Wrapped derivative of h_temp wrt stress.

virtual void **dh_da_temp**(const double *const s, const double *const alpha, double T, double *const dhv) const
Derivative of h_temp wrt history.

virtual void **dh_da_temp**(const State &state, *History* &res) const
Wrapped derivative of h_temp wrt history.

template<class **T**>
inline *History* **blank_derivative_**() const
Blank derivative of a history.

18.2 Test wrapped flow rule

18.2.1 Overview

As a test of the *Viscoplastic model wrapper* system, this class implements a power law flow rule with hard-coded isotropic hardening:

$$y = \dot{\varepsilon}_0 \left\langle \frac{\sqrt{3/2}\,\mathrm{dev}\,\sigma - h}{D} \right\rangle^n$$

with

$$\dot{h} = K$$

and $h(0) = \sigma_0$.

18.2.2 Parameters

Parameter	Object type	Description	Default
`eps0`	`double`	Reference strain rate	No
`D`	`double`	Drag stress	No
`n`	`double`	Rate sensitivity exponent	No
`s0`	`double`	Initial hardening strength	No
`K`	`double`	Hardening modulus	No

18.2.3 Class description

class **TestFlowRule** : public *neml*::*WrappedViscoPlasticFlowRule*

Test implementation of a simple flow rule.

Public Functions

TestFlowRule(*ParameterSet* &params)

virtual void **populate_hist**(*History* &h) const

Populate a history object.

virtual void **init_hist**(*History* &h) const

Initialize the history.

virtual void **y**(const State &state, double &res) const

Wrapped scalar inelastic strain rate.

virtual void **dy_ds**(const State &state, *Symmetric* &res) const

Derivative of y wrt stress.

virtual void **dy_da**(const State &state, *History* &res) const

Derivative of y wrt history.

virtual void **g**(const State &state, *Symmetric* &res) const

Flow rule proportional to the scalar strain rate.

virtual void **dg_ds**(const State &state, *SymSymR4* &res) const

Derivative of g wrt stress.

virtual void **dg_da**(const State &state, *History* &res) const

Derivative of g wrt history.

virtual void **h**(const State &state, *History* &res) const

Hardening rule proportional to the scalar strain rate.

virtual void **dh_ds**(const State &state, *History* &res) const

Derivative of h wrt stress.

virtual void **dh_da**(const State &state, *History* &res) const

Derivative of h wrt history.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Return default parameters.

static *ParameterSet* **parameters**()

Initialize from parameter set.

18.3 Walker flow rule

18.3.1 Overview

This class implements the full Walker viscoplastic flow rule described in *Walker Alloy 617 model subsystem*.

18.3.2 Parameters

Parameter	Object type	Description	Default
eps0	*neml::Interpolate*	Reference strain rate	No
softening	neml::SofteningModeling	Softening model	No
scaling	*neml::ThermalScaling*	Thermal scaling model	No
n	*neml::Interpolate*	Rate sensitivity	No
k	*neml::Interpolate*	Constant part of drag stress	No
m	*neml::Interpolate*	Drag stress exponent	No
R	*neml::IsotropicHardening*	Isotropic hardening model	No
D	*neml::DragStress*	Drag stress model	No
X	*neml::KinematicHardening*	Kinematic hardening model	No

18.3.3 Class description

class **WalkerFlowRule** : public *neml*::*WrappedViscoPlasticFlowRule*

Full Walker flow rule.

Public Functions

WalkerFlowRule(*ParameterSet* &params)

virtual void **populate_hist**(*History* &h) const

Populate a history object.

virtual void **init_hist**(*History* &h) const

Initialize history with time zero values.

virtual void **y**(const State &state, double &res) const

Wrapped scalar inelastic strain rate.

virtual void **dy_ds**(const State &state, *Symmetric* &res) const

Derivative of y wrt stress.

virtual void **dy_da**(const State &state, *History* &res) const

Derivative of y wrt history.

virtual void **g**(const State &state, *Symmetric* &res) const

Flow rule proportional to the scalar strain rate.

virtual void **dg_ds**(const State &state, *SymSymR4* &res) const

Derivative of g wrt stress.

virtual void **dg_da**(const State &state, *History* &res) const
Derivative of g wrt history.

virtual void **h**(const State &state, *History* &res) const
Hardening rule proportional to the scalar strain rate.

virtual void **dh_ds**(const State &state, *History* &res) const
Derivative of h wrt stress.

virtual void **dh_da**(const State &state, *History* &res) const
Derivative of h wrt history.

virtual void **h_time**(const State &state, *History* &res) const
Hardening rule proportional to time.

virtual void **dh_ds_time**(const State &state, *History* &res) const
Derivative of h_time wrt stress.

virtual void **dh_da_time**(const State &state, *History* &res) const
Derivative of h_time wrt history.

virtual void **override_guess**(double *const x)
Override initial guess.

Public Static Functions

static std::string **type**()
String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)
Return default parameters.

static *ParameterSet* **parameters**()
Initialize from parameter set.

The model implements the various subcomponents of the flow rule as individual classes

18.4 Walker softening models

18.4.1 Overview

Softening models providing the Φ term in the *Walker Alloy 617 model subsystem*.

18.4.2 Base class

class **SofteningModel** : public *neml*::*NEMLObject*
Softening/tertiary creep models.

Subclassed by *neml::WalkerSofteningModel*

Public Functions

SofteningModel(*ParameterSet* &params)

virtual double **phi**(double alpha, double T) const
Softening function.

virtual double **dphi**(double alpha, double T) const
Derivative of softening function wrt alpha.

Public Static Functions

static std::string **type**()
String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)
Initialize from a parameter set.

static *ParameterSet* **parameters**()
Return default parameters.

18.4.3 Walker's specific model

The specific softening model described in *Walker Alloy 617 model subsystem*.

Parameters

Parameter	Object type	Description	Default
phi0	*neml::Interpolate*	Softening prefactor	No
phi1	*neml::Interpolate*	Softening exponent	No

Class description

class **WalkerSofteningModel** : public *neml*::*SofteningModel*
The simple softening model Walker actually uses.

Public Functions

WalkerSofteningModel(*ParameterSet* &params)

virtual double **phi**(double alpha, double T) const
Softening function.

virtual double **dphi**(double alpha, double T) const
Derivative of softening function wrt alpha.

Public Static Functions

static std::string **type**()
: String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)
: Initialize from a parameter set.

static *ParameterSet* **parameters**()
: Return default parameters.

18.5 Walker thermal scaling models

18.5.1 Overview

Scaling models providing the χ term in the *Walker Alloy 617 model subsystem*.

18.5.2 Base class

class **ThermalScaling** : public *neml*::*NEMLObject*
: Thermal rate scaling models.

Subclassed by *neml::ArrheniusThermalScaling*

Public Functions

ThermalScaling(*ParameterSet* &params)

virtual double **value**(double T) const
: The thermal ratio.

Public Static Functions

static std::string **type**()
: String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)
: Initialize from a parameter set.

static *ParameterSet* **parameters**()
: Return default parameters.

18.5.3 Walker's specific model

The specific scaling model described in *Walker Alloy 617 model subsystem*.

Parameters

Parameter	Object type	Description	Default
`Q`	*`neml::Interpolate`*	Activation energy	No
`R`	`double`	Universal gas constant	No
`Tref`	`double`	Reference temperature	No

Class description

class **`ArrheniusThermalScaling`** : public *neml*::*ThermalScaling*

Walker's actual thermal scaling model.

Public Functions

`ArrheniusThermalScaling`(*ParameterSet* &params)

virtual double **`value`**(double T) const

The thermal ratio.

Public Static Functions

static std::string **`type`**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **`initialize`**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **`parameters`**()

Return default parameters.

18.6 Walker isotropic hardening models

18.6.1 Overview

Models providing the isotropic hardening contribution in the *Walker Alloy 617 model subsystem*.

18.6.2 Base class

class **IsotropicHardening** : public *neml*::InternalVariable<double>

Subclassed by *neml::ConstantIsotropicHardening*, *neml::WalkerIsotropicHardening*

Public Functions

IsotropicHardening(*ParameterSet* &params)

inline void **set_scaling**(std::shared_ptr<*ThermalScaling*> scale)

virtual double **ratet**(VariableState &state)

Return zero for time rate by default.

virtual double **d_ratet_d_h**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual double **d_ratet_d_a**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual double **d_ratet_d_adot**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual double **d_ratet_d_D**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual *Symmetric* **d_ratet_d_s**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual *Symmetric* **d_ratet_d_g**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual double **rateT**(VariableState &state)

Return zero for temperature rate by default.

virtual double **d_rateT_d_h**(VariableState &state)

Return zero for the temperature rate derivatives by default.

virtual double **d_rateT_d_a**(VariableState &state)

Return zero for the temperature rate derivatives by default.

virtual double **d_rateT_d_adot**(VariableState &state)

Return zero for the temperature rate derivatives by default.

virtual double **d_rateT_d_D**(VariableState &state)

Return zero for the temperature rate derivatives by default.

virtual *Symmetric* **d_rateT_d_s**(VariableState &state)

Return zero for the temperature rate derivatives by default.

virtual *Symmetric* **d_rateT_d_g**(VariableState &state)

Return zero for the temperature rate derivatives by default.

18.6.3 Constant isotropic hardening

Isotropic hardening fixed to zero.

Parameters

Parameter	Object type	Description	Default
`scale`	*`neml::ThermalScaling`*	Thermal scaling model	No scaling

Class description

class **`ConstantIsotropicHardening`** : public *neml*::*IsotropicHardening*

Public Functions

`ConstantIsotropicHardening`(*ParameterSet* &params)

virtual double **`initial_value`**()

virtual double **`ratep`**(VariableState &state)

virtual double **`d_ratep_d_h`**(VariableState &state)

virtual double **`d_ratep_d_a`**(VariableState &state)

virtual double **`d_ratep_d_adot`**(VariableState &state)

virtual double **`d_ratep_d_D`**(VariableState &state)

virtual *Symmetric* **`d_ratep_d_s`**(VariableState &state)

virtual *Symmetric* **`d_ratep_d_g`**(VariableState &state)

Public Static Functions

static std::string **`type`**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **`initialize`**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **`parameters`**()

Return default parameters.

18.6.4 Walker's specific model

The specific isotropic hardening model described in *Walker Alloy 617 model subsystem*.

Parameters

Parameter	Object type	Description	Default
`r0`	*neml::Interpolate*	Hardening rate prefactor	No
`Rinf`	*neml::Interpolate*	Final value of isotropic hardening	No
`r1`	*neml::Interpolate*	Static recovery prefactor	No
`r2`	*neml::Interpolate*	Static recovery exponent	No
`scale`	*neml::ThermalScaling*	Thermal scaling model	No scaling

Class description

class **WalkerIsotropicHardening** : public *neml*::*IsotropicHardening*

The actual hardening model used in Walker's A617 viscoplastic model.

Public Functions

WalkerIsotropicHardening(*ParameterSet* &params)

virtual double **initial_value**()

virtual double **ratep**(VariableState &state)

virtual double **d_ratep_d_h**(VariableState &state)

virtual double **d_ratep_d_a**(VariableState &state)

virtual double **d_ratep_d_adot**(VariableState &state)

virtual double **d_ratep_d_D**(VariableState &state)

virtual *Symmetric* **d_ratep_d_s**(VariableState &state)

virtual *Symmetric* **d_ratep_d_g**(VariableState &state)

virtual double **ratet**(VariableState &state)

Return zero for time rate by default.

virtual double **d_ratet_d_h**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual double **d_ratet_d_a**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual double **d_ratet_d_adot**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual double **d_ratet_d_D**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual *Symmetric* **d_ratet_d_s**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual *Symmetric* **d_ratet_d_g**(VariableState &state)

Return zero for the time rate derivatives by default.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Return default parameters.

18.7 Walker kinematic hardening models

18.7.1 Overview

Models providing the kinematic hardening contribution in the *Walker Alloy 617 model subsystem*.

18.7.2 Base class

class **KinematicHardening** : public *neml*::InternalVariable<V>

Subclassed by *neml::FAKinematicHardening*, *neml::WalkerKinematicHardening*

Public Functions

KinematicHardening(*ParameterSet* &params)

inline void **set_scaling**(std::shared_ptr<*ThermalScaling*> scale)

virtual *Symmetric* **ratet**(VariableState &state)

Return zero for time rate by default.

virtual *SymSymR4* **d_ratet_d_h**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual *Symmetric* **d_ratet_d_a**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual *Symmetric* **d_ratet_d_adot**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual *Symmetric* **d_ratet_d_D**(VariableState &state)
Return zero for the time rate derivatives by default.

virtual *SymSymR4* **d_ratet_d_s**(VariableState &state)
Return zero for the time rate derivatives by default.

virtual *SymSymR4* **d_ratet_d_g**(VariableState &state)
Return zero for the time rate derivatives by default.

virtual *Symmetric* **rateT**(VariableState &state)
Return zero for temperature rate by default.

virtual *SymSymR4* **d_rateT_d_h**(VariableState &state)
Return zero for the temperature rate derivatives by default.

virtual *Symmetric* **d_rateT_d_a**(VariableState &state)
Return zero for the temperature rate derivatives by default.

virtual *Symmetric* **d_rateT_d_adot**(VariableState &state)
Return zero for the temperature rate derivatives by default.

virtual *Symmetric* **d_rateT_d_D**(VariableState &state)
Return zero for the temperature rate derivatives by default.

virtual *SymSymR4* **d_rateT_d_s**(VariableState &state)
Return zero for the temperature rate derivatives by default.

virtual *SymSymR4* **d_rateT_d_g**(VariableState &state)
Return zero for the temperature rate derivatives by default.

18.7.3 Frederick-Armstrong hardening

The Frederick-Armstrong [FA2007] model, implemented in the Walker subsystem:

$$\dot{X} = c\dot{\varepsilon}_{vp} - gX$$

Parameters

Parameter	Object type	Description	Default
c	*neml::Interpolate*	Hardening constant	No
g	*neml::Interpolate*	Dynamic recover constant	No
scale	*neml::ThermalScaling*	Thermal scaling model	No scaling

Class description

class **FAKinematicHardening** : public *neml*::*KinematicHardening*

Standard Frederick-Armstrong hardening.

Public Functions

FAKinematicHardening(*ParameterSet* &params)

virtual *Symmetric* **initial_value**()

virtual *Symmetric* **ratep**(VariableState &state)

virtual *SymSymR4* **d_ratep_d_h**(VariableState &state)

virtual *Symmetric* **d_ratep_d_a**(VariableState &state)

virtual *Symmetric* **d_ratep_d_adot**(VariableState &state)

virtual *Symmetric* **d_ratep_d_D**(VariableState &state)

virtual *SymSymR4* **d_ratep_d_s**(VariableState &state)

virtual *SymSymR4* **d_ratep_d_g**(VariableState &state)

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Return default parameters.

18.7.4 Walker's specific model

The specific kinematic hardening model described in *Walker Alloy 617 model subsystem*.

Parameters

Parameter	Object type	Description	Default
c0	*neml::Interpolate*	Constant hardening parameter	No
c1	*neml::Interpolate*	Hardening evolution prefactor	No
c2	*neml::Interpolate*	Hardening evolution exponent	No
l0	*neml::Interpolate*	Dynamic recovery exponential rate	No
l1	*neml::Interpolate*	Dynamic recovery evolution prefactor	No
l	*neml::Interpolate*	Constant dynamic recovery coeficient	No
b0	*neml::Interpolate*	Recovery direction constant	No
x0	*neml::Interpolate*	Static recovery prefactor	No
x1	*neml::Interpolate*	Static recovery exponent	No
softening	*neml::SofteningModel*	Softening model	No
scale	*neml::ThermalScaling*	Thermal scaling model	No scaling

Class description

class **WalkerKinematicHardening** : public *neml*::*KinematicHardening*

Walker's kinematic hardening model.

Public Functions

WalkerKinematicHardening(*ParameterSet* &params)

virtual *Symmetric* **initial_value**()

virtual *Symmetric* **ratep**(VariableState &state)

virtual *SymSymR4* **d_ratep_d_h**(VariableState &state)

virtual *Symmetric* **d_ratep_d_a**(VariableState &state)

virtual *Symmetric* **d_ratep_d_adot**(VariableState &state)

virtual *Symmetric* **d_ratep_d_D**(VariableState &state)

virtual *SymSymR4* **d_ratep_d_s**(VariableState &state)

virtual *SymSymR4* **d_ratep_d_g**(VariableState &state)

virtual *Symmetric* **ratet**(VariableState &state)

Return zero for time rate by default.

virtual *SymSymR4* **d_ratet_d_h**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual *Symmetric* **d_ratet_d_a**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual *Symmetric* **d_ratet_d_adot**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual *Symmetric* **d_ratet_d_D**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual *SymSymR4* **d_ratet_d_s**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual *SymSymR4* **d_ratet_d_g**(VariableState &state)

Return zero for the time rate derivatives by default.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Return default parameters.

18.8 Walker drag stress models

18.8.1 Overview

Models providing the drag stress contribution in the *Walker Alloy 617 model subsystem*.

18.8.2 Base class

class **DragStress** : public *neml*::InternalVariable<V>

Subclassed by *neml::ConstantDragStress*, *neml::WalkerDragStress*

Public Functions

DragStress(*ParameterSet* &params)

inline void **set_scaling**(std::shared_ptr<*ThermalScaling*> scale)

virtual double **D_xi**(double T) = 0

Report the value of D_xi.

virtual double **D_0**(double T) = 0

Report the value of D_0.

virtual double **d_ratep_d_D**(VariableState &state)

Makes no sense in this context!

Makes no sense in this context.

virtual double **ratet**(VariableState &state)

Return zero for time rate by default.

virtual double **d_ratet_d_h**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual double **d_ratet_d_a**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual double **d_ratet_d_adot**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual double **d_ratet_d_D**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual *Symmetric* **d_ratet_d_s**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual *Symmetric* **d_ratet_d_g**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual double **rateT**(VariableState &state)

Return zero for temperature rate by default.

virtual double **d_rateT_d_h**(VariableState &state)

Return zero for the temperature rate derivatives by default.

virtual double **d_rateT_d_a**(VariableState &state)

Return zero for the temperature rate derivatives by default.

virtual double **d_rateT_d_adot**(VariableState &state)

Return zero for the temperature rate derivatives by default.

virtual double **d_rateT_d_D**(VariableState &state)

Return zero for the temperature rate derivatives by default.

virtual *Symmetric* **d_rateT_d_s**(VariableState &state)

Return zero for the temperature rate derivatives by default.

virtual *Symmetric* **d_rateT_d_g**(VariableState &state)

Return zero for the temperature rate derivatives by default.

18.8.3 Constant drag stress

Fixed, non-evolving drag stress.

Parameters

Parameter	Object type	Description	Default
`value`	`double`	Value of the drag stress	No
`scale`	*`neml::ThermalScaling`*	Thermal scaling model	No scaling

Class description

class **ConstantDragStress** : public *neml*::*DragStress*

Public Functions

ConstantDragStress(*ParameterSet* &params)

virtual double **initial_value**()

virtual double **D_xi**(double T)
Report the value of D_xi.

virtual double **D_0**(double T)
Report the value of D_0.

virtual double **ratep**(VariableState &state)

virtual double **d_ratep_d_h**(VariableState &state)

virtual double **d_ratep_d_a**(VariableState &state)

virtual double **d_ratep_d_adot**(VariableState &state)

virtual *Symmetric* **d_ratep_d_s**(VariableState &state)

virtual *Symmetric* **d_ratep_d_g**(VariableState &state)

Public Static Functions

static std::string **type**()
String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)
Initialize from a parameter set.

static *ParameterSet* **parameters**()
Return default parameters.

18.8.4 Walker's specific model

The specific drag stress evolution model described in *Walker Alloy 617 model subsystem*.

Parameters

Parameter	Object type	Description	Default
do	*neml::Interpolate*	Hardening prefactor	No
d1	*neml::Interpolate*	Recovery prefactor	No
d2	*neml::Interpolate*	Recovery exponent	No
D_xi	*neml::Interpolate*	Saturated drag stress	No
D_0	double	Initial drag stress	No
softening	*neml::SofteningModel*	Softening model	No
scale	*neml::ThermalScaling*	Thermal scaling model	No scaling

Class description

class **WalkerDragStress** : public *neml*::*DragStress*

Public Functions

WalkerDragStress(*ParameterSet* &params)

virtual double **initial_value**()
Initial value of drag stress.

virtual double **D_xi**(double T)
Report the value of D_xi.

virtual double **D_0**(double T)
Report the value of D_0.

virtual double **ratep**(VariableState &state)

virtual double **d_ratep_d_h**(VariableState &state)

virtual double **d_ratep_d_a**(VariableState &state)

virtual double **d_ratep_d_adot**(VariableState &state)

virtual *Symmetric* **d_ratep_d_s**(VariableState &state)

virtual *Symmetric* **d_ratep_d_g**(VariableState &state)

virtual double **ratet**(VariableState &state)
Return zero for time rate by default.

virtual double **d_ratet_d_h**(VariableState &state)
Return zero for the time rate derivatives by default.

virtual double **d_ratet_d_a**(VariableState &state)
Return zero for the time rate derivatives by default.

virtual double **d_ratet_d_adot**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual *Symmetric* **d_ratet_d_s**(VariableState &state)

Return zero for the time rate derivatives by default.

virtual *Symmetric* **d_ratet_d_g**(VariableState &state)

Return zero for the time rate derivatives by default.

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Return default parameters.

CHAPTER

NINETEEN

PYTHON BINDINGS AND HELPERS

If you compile NEML with the python bindings the build will produce compiled python modules in the `neml/` directory. The names of these modules match the underlying C++ files and also match the chapter titles in this manual. So for example, yield surfaces will be in the module surfaces and so on. All public methods of all classes are available in python. The modules are configured to take standard Python lists as input, rather than numpy arrays.

To help developing, testing, and debugging material models NEML provides several python drivers in the neml python module. These helpers run material models degenerate NEML's 3D formulation to 1D uniaxial stress, exercise NEML material models in common experimental loading situations (uniaxial tension, creep, stress relaxation, strain and strain controlled cyclic loading, with holds, and strain rate jump tests), run simulations of arbitrarily connected collections of uniaxial bars, where a NEML model gives the constitutive response of each bar, and runs simplified, 1D models of pressure vessels, again using a NEML model to define the vessel's constitutive response.

19.1 uniaxial

NEML material models are strain-controlled and 3D. Degenerating this response to plane strain is trivial – simply pass in zeros in the appropriate strain components. However, another very useful stress state is strain-controlled uniaxial stress. This is the stress state in many common experimental tests, for example standard tension tests. The stress state in these conditions is:

$$\sigma = \begin{bmatrix} \sigma & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

where σ is the unknown uniaxial stress. The strain state is

$$\varepsilon = \begin{bmatrix} \varepsilon & \varepsilon_{12} & \varepsilon_{13} \\ \varepsilon_{12} & \varepsilon_{22} & \varepsilon_{23} \\ \varepsilon_{13} & \varepsilon_{23} & \varepsilon_{33} \end{bmatrix}$$

where ε is the known, controlled input strain and the remaining strain components are unknowns.

The neml axisym module solves a system of nonlinear equations to impose this state of strain and stress on a standard, 3D NEML material model. The module returns the uniaxial stress, the history, the stored energy and dissipated work, along with the new, uniaxial algorithmic tangent

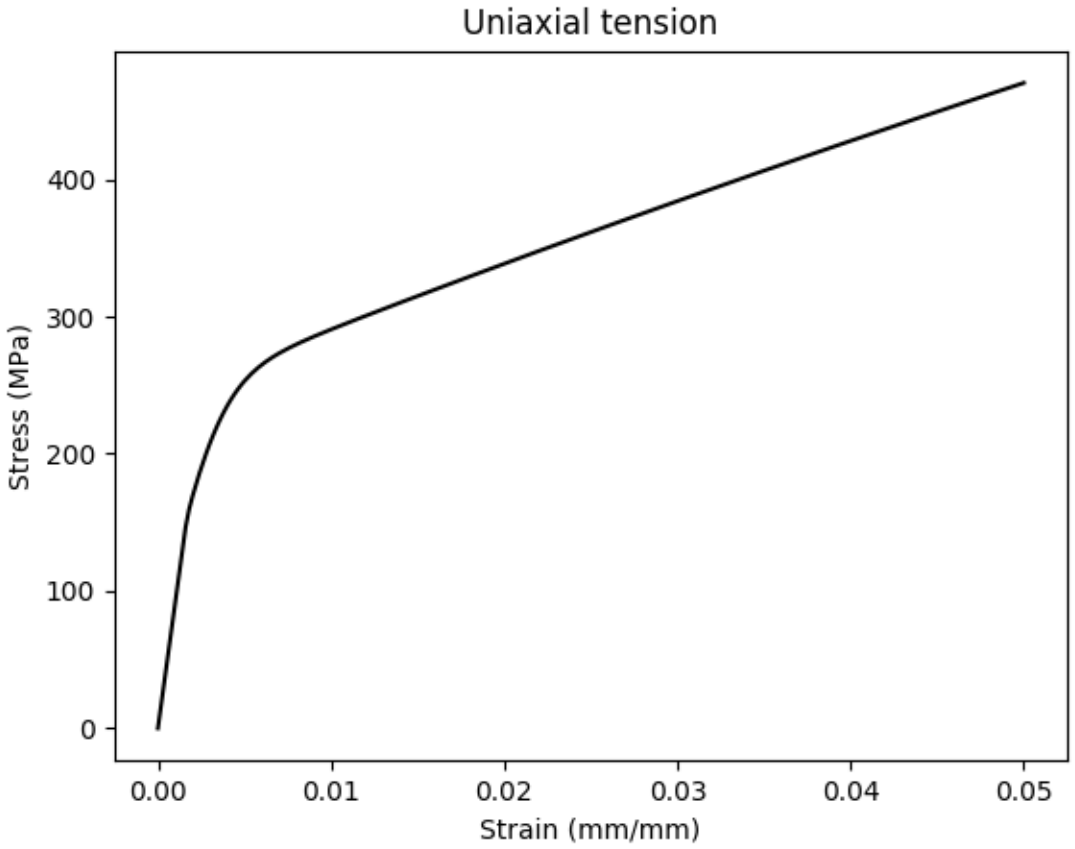
$$\mathbf{A} = \frac{\mathrm{d}\sigma}{\mathrm{d}\varepsilon}$$

19.2 drivers

This python module provides methods to drive NEML material models in loading conditions representative of common experimental tests.

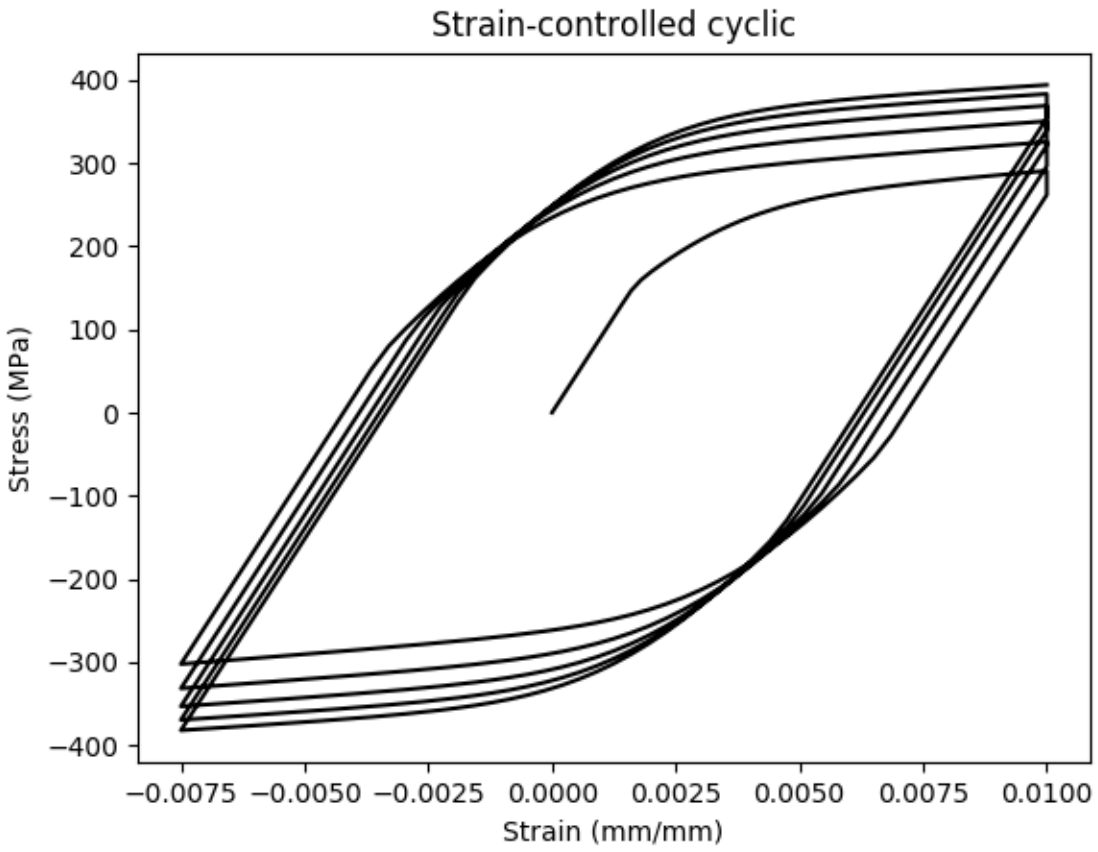
19.2.1 Uniaxial tension

This routine produces a uniaxial tension (or compression) curve for a NEML model.



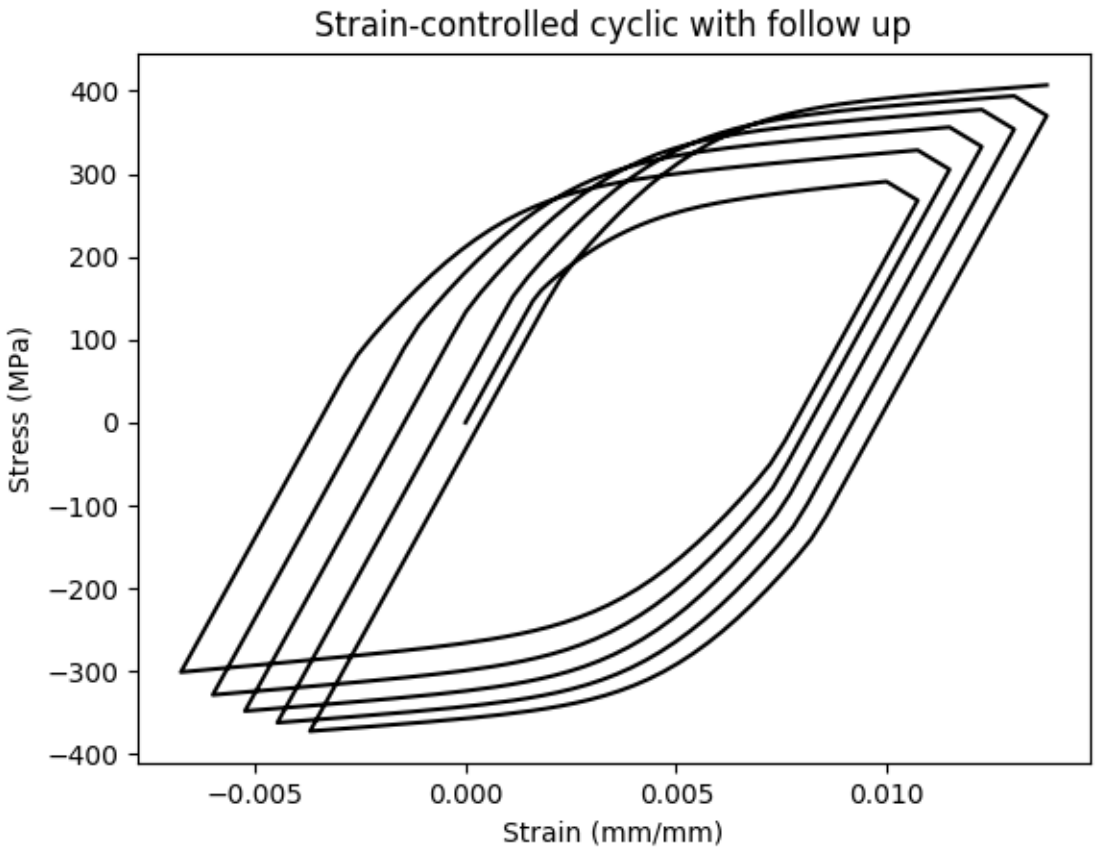
19.2.2 Strain controlled cyclic

This routine mimic a strain controlled cyclic test, with or without holds on the maximum tension and compressive strains.



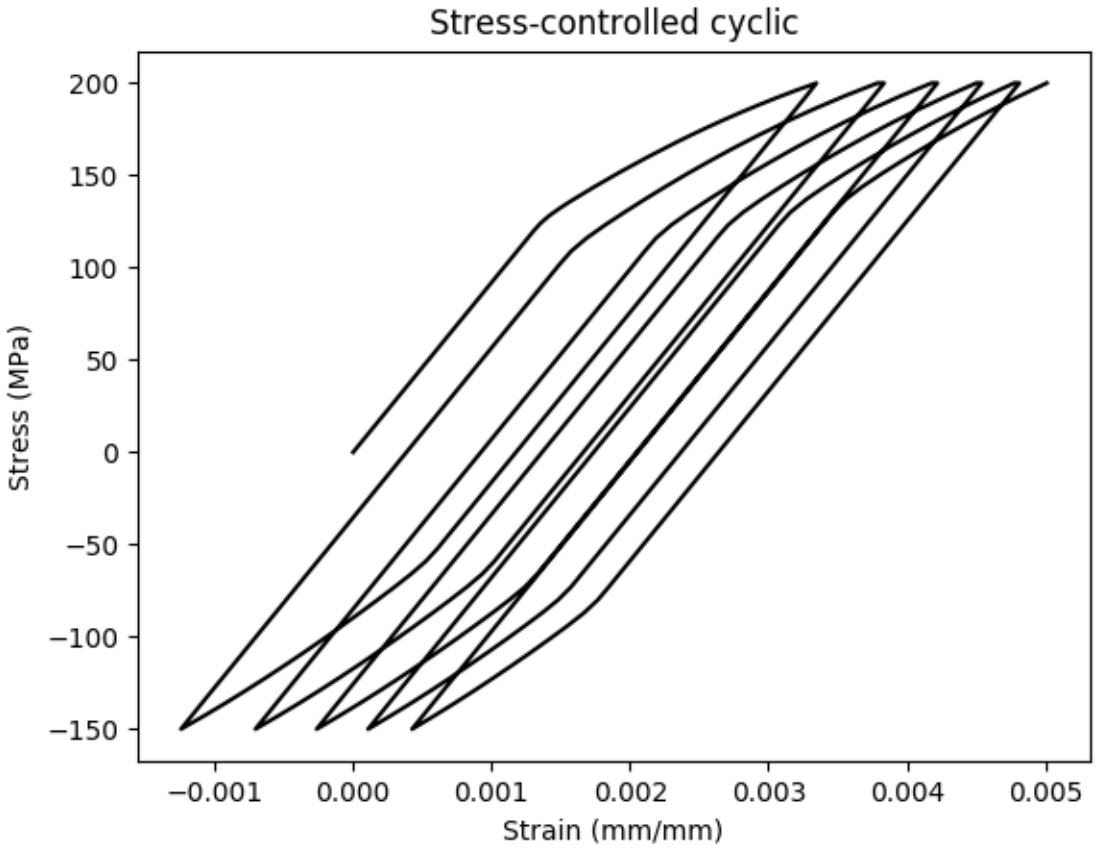
19.2.3 Strain controlled cyclic with follow up

This routine mimic a strain controlled cyclic test, with or without holds on the maximum tension and compressive strains. This version can impose some known elastic follow up on the stress relaxation parts of the cycle.



19.2.4 Stress controlled cyclic

This routine mimic a stress controlled cyclic test, with or without holds on the maximum tension and compressive strains.

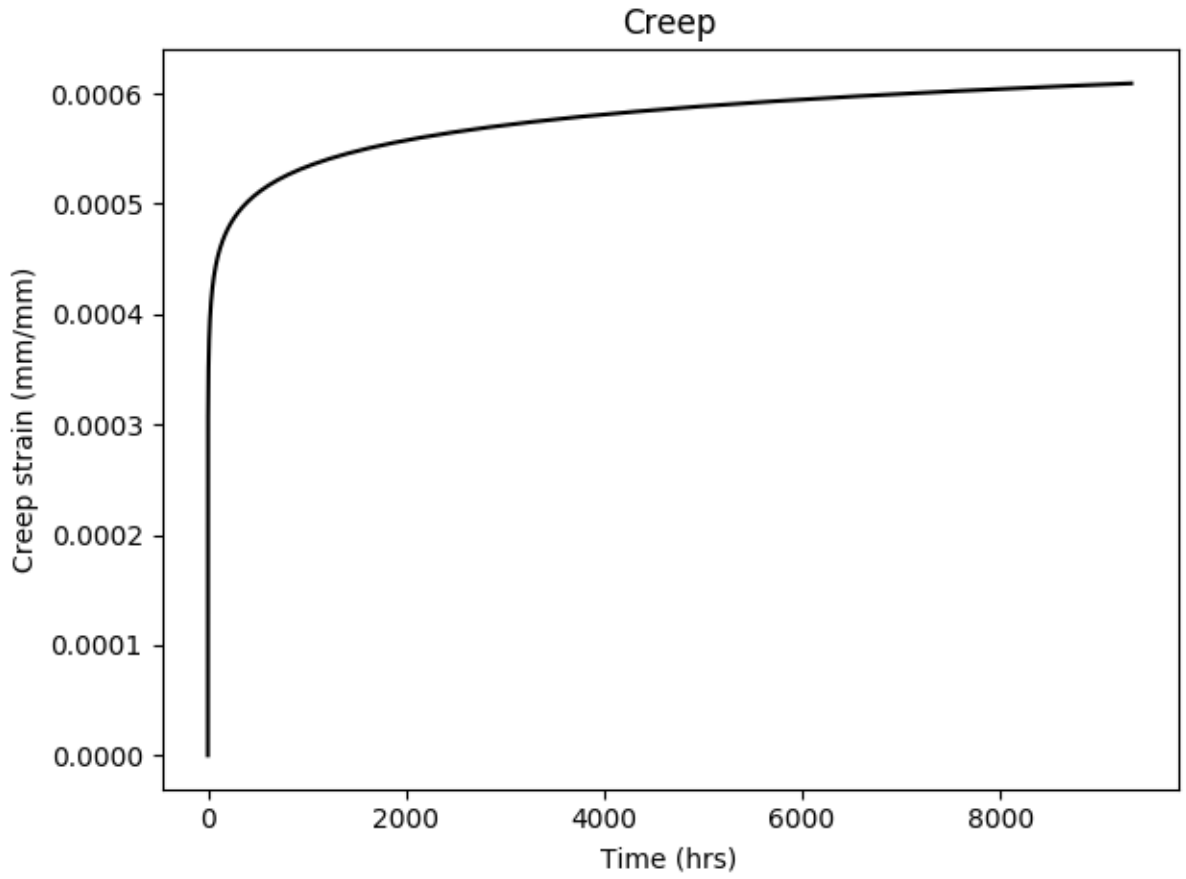


19.2.5 Stress relaxation

This routine simulates a stress relaxation tests where the sample is loaded under strain control to some strain level and then then strain is held constant and the stress allowed to relax.

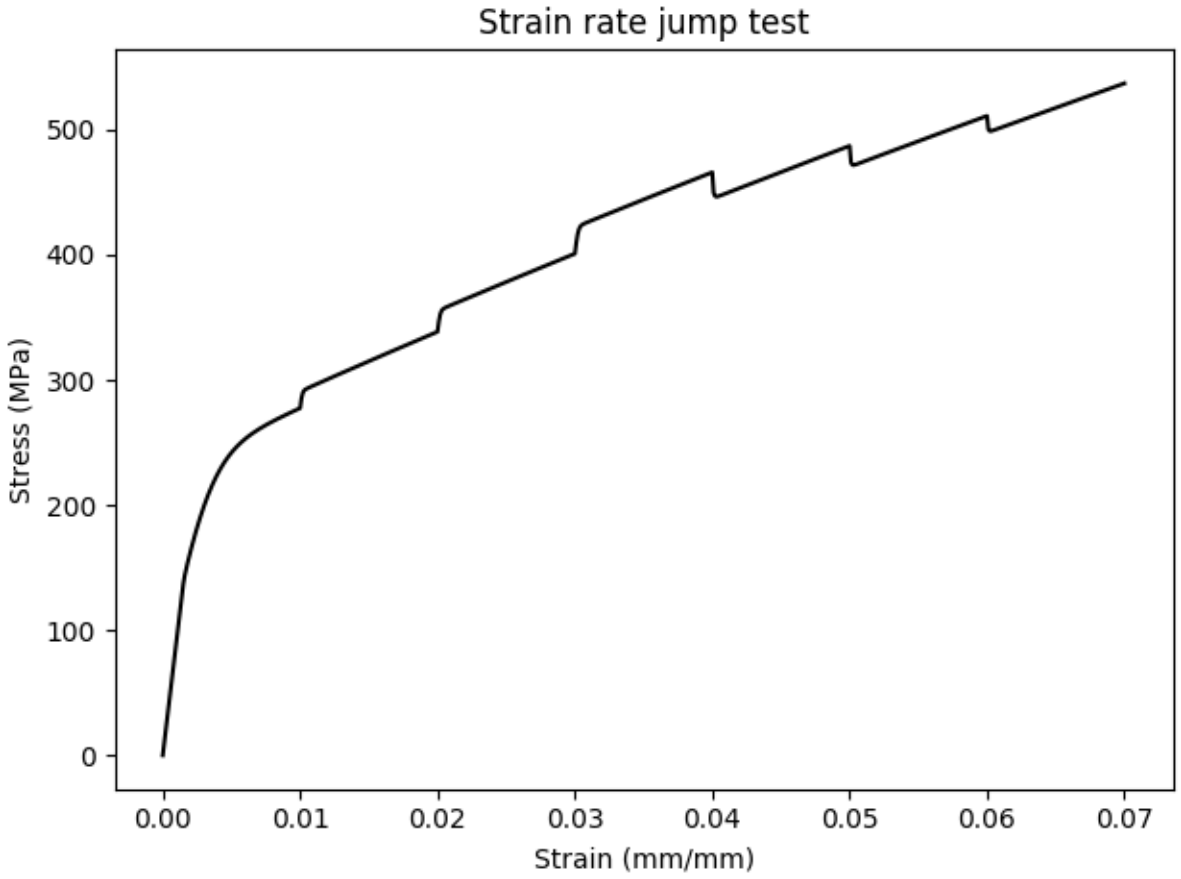
19.2.6 Creep

This routine simulates a creep test where the sample is loaded to some stress under stress control and then the stress is held fixed as the strain increases.



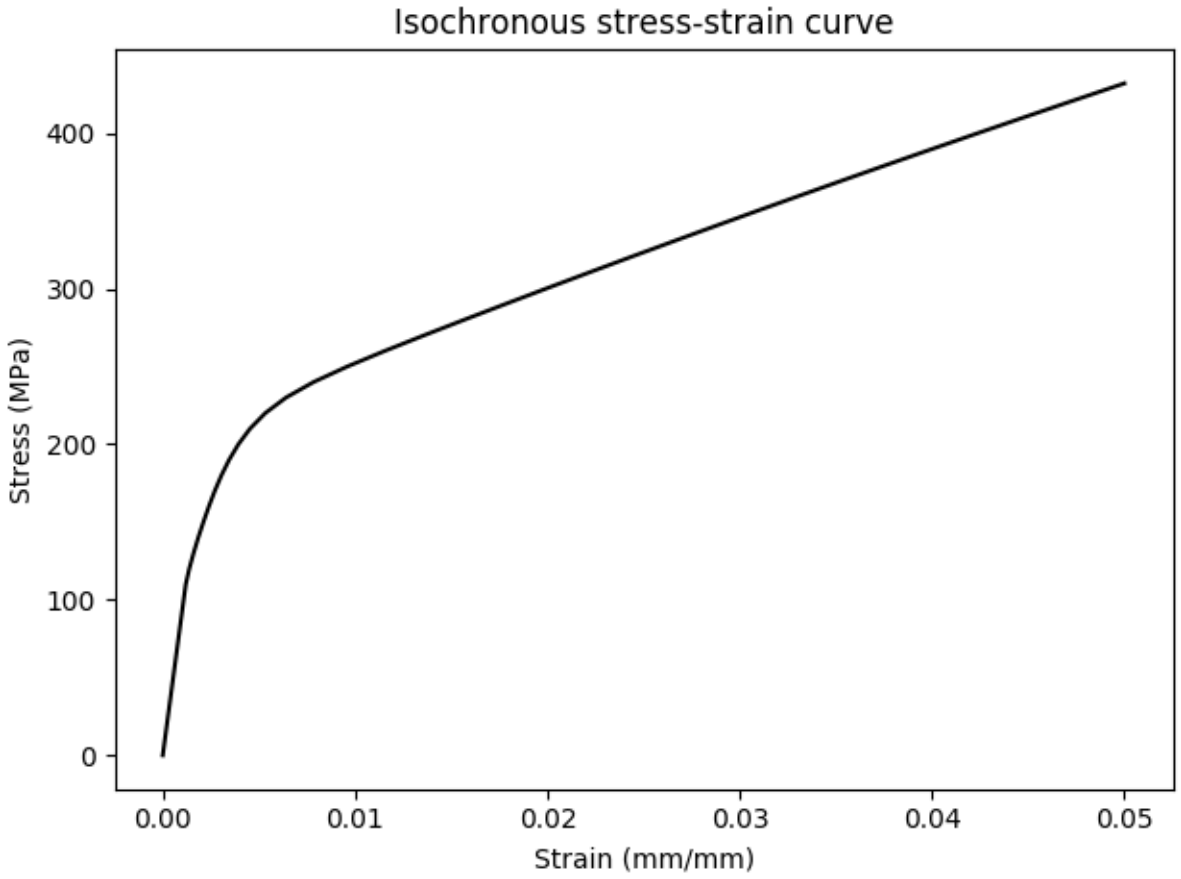
19.2.7 Strain rate jump test

This routine simulates a strain rate jump test. The specimen is under uniaxial stress and strain control, like a standard tension test. However, at certain times during the test the strain rate is either increased or decreased, causing a jump in the flow stress for rate sensitive materials.



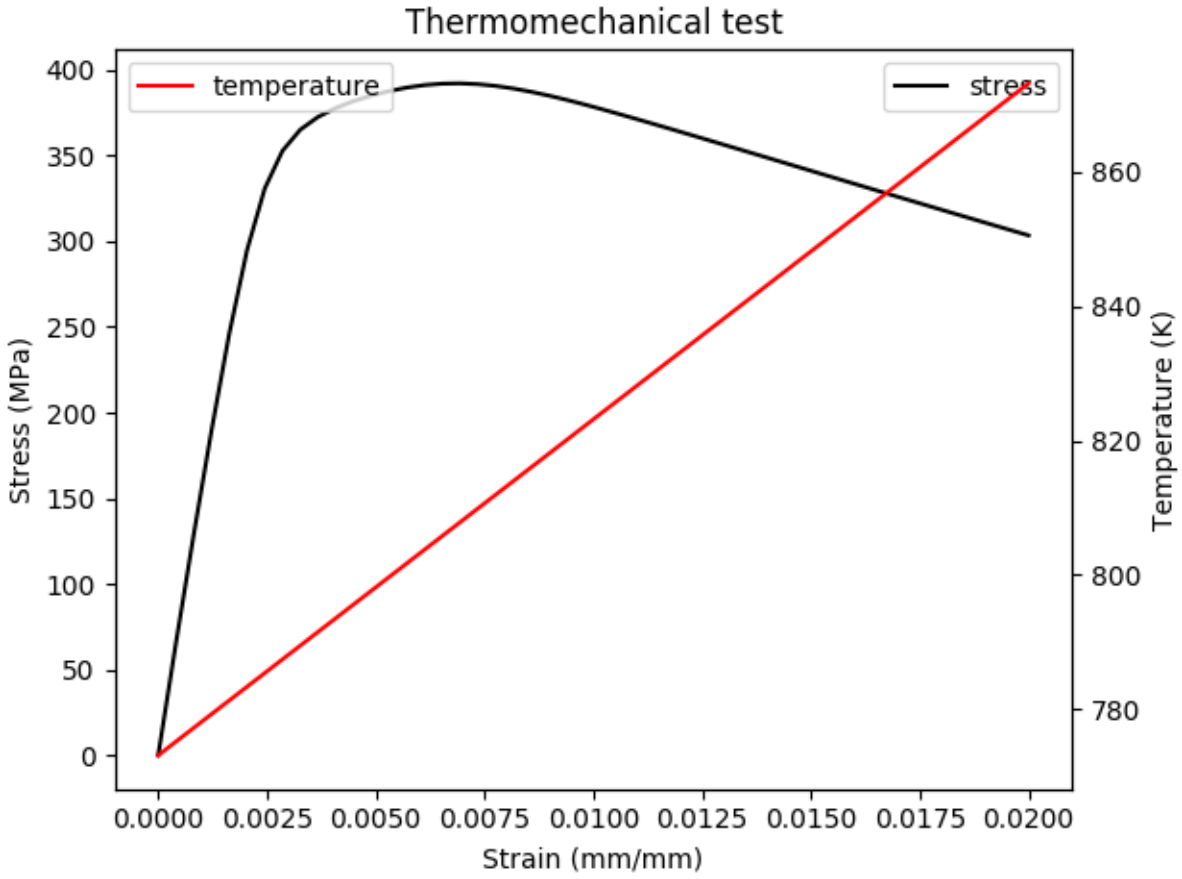
19.2.8 Isochronous stress-strain curve

This routine generates an isochronous stress-strain curve for a model and for a given time and temperature. Isochronous stress-strain curves are a method for summarizing a collection of creep tests at the same temperature for the same material. The isochronous stress-strain curve is the locus of (strain,stress) points the material passes through at time t during the collection of creep tests.



19.2.9 Thermomechanical test

A thermomechanical test is a strain-controlled experiment where both the strain and the temperature are simultaneously changed. This driver takes a collection of time, strain, and temperature points and drives a NEML model through that collection of data, reporting the stress as a function of time.



19.2.10 Driver classes

All the individual drivers use these classes to actually advance the NEML models in uniaxial stress states.

19.2.11 Helpers

These routines are used to help postprocess the results of some of the driver simulations.

19.3 arbbar

To be added later.

19.4 axisym

To be added later.

CHAPTER

TWENTY

ADVANCED TOPICS

These topics are only important if you are going to add new material model components to NEML. They describe the mathematical helpers used in NEML to do common operations on tensors in Mandel notation, the class used to solve nonlinear equations, and the object system used to manage all the individual components in NEML and generate XML input and the python bindings.

20.1 Integrating models

A generic material model has two jobs:

1. Integrate the stress and some set of internal variables from time step n to time step $n+1$, given the strain increment (and, for some models, the time increment and temperature increment) as input.
2. Provide the calling routine (often the stress update at a Gauss point in a finite element code) the algorithmic tangent – the derivative of the stress at time $n+1$ with respect to the strain increment.

In alternate formulations the model might be stress-based – the calling routine provides the stress and expects the strain – or in multiphysics simulations the calling routine might require a similar algorithmic tangent with respect to the temperature or some other variable. The following derivation can be adjusted to accommodate either condition.

NEML uses a single algorithm to accomplish these tasks for the:

- *Perfect plasticity*
- *Rate indepement plasticity*
- *General rate dependent*

models. The same algorithm could be used, pointlessly, for the *linear elasticity* update. These classes encompass all flavors of small strain constitutive models commonly used in structural simulations.

The remainder of the models, including the *damage* models, *creep + plasticity*, and *regime switching* models use specialized integration algorithms. The common feature for these model is that they take as input one of the base material models listed above and modify that basic stress update formulation.

The standard material model update formulation in NEML can be described as the implicit time integration of the stress rate and a set of generic internal varaibles:

$$\dot{x} = \begin{bmatrix} \dot{\sigma}\left(\sigma, h, t, \varepsilon, T\right) \\ \dot{h}\left(\sigma, h, t, \varepsilon, T\right) \end{bmatrix}$$

As suggested by this equation, the implementation divides the variables in the stress and update functions into two sets: the implicit variables x (i.e. the stress and generic set of history variables) and the explicit variables y (i.e. the strain if we are only interested in the classical algorithmic tangent). For a backward Euler implementation of the rate equations we can formulate the generic residual equation:

$$R = x_{n+1} - x_n - \dot{x}\left(x; y\right) \Delta t$$

i.e. the material point integration only updates the implicit variables and leaves the explicit variables fixed. This generic equation can be solved with Newton's method. That requires the Jacobian:

$$J = I - \frac{\partial \dot{x}}{\partial x}.$$

The stress update algorithms for the all of the standard models can be cast into this form. Each model must then implement the correct residual and jacobian. The remainder of the integration can use a common algorithm.

The complete, generalized algorithmic tangent is the total derivative of the implicit variables x with respect to the explicit variables y. The classical algorithmic tangent, required by FE solvers, is simply the $\frac{d\sigma}{d\varepsilon}$ block of this generic derivative. Note that multiphysics codes might be interested in other blocks of this "generalized" tangent matrix, for example the $\frac{d\sigma}{dT}$ block.

Calculating the generalized tangent matrix is a straightforward application of the implicit function theorem. After successfully solving the residual equation we have the condition:

$$dR = 0 = \frac{\partial R}{\partial x} : dx + \frac{\partial R}{\partial y} : dy$$

which implies that

$$\frac{dx}{dy} = -\left(\frac{\partial R}{\partial x}\right)^{-1} : \frac{\partial R}{\partial y}$$

The partial derivative $J = \frac{\partial R}{\partial x}$ is the jacobian of the local constitutive update. The other partial of the residual with respect to the explicit variables is not required during the local stress update and must be determined only for the tangent calculation. The final expression for the generalized algorithmic tangent is then:

$$T = -J^{-1} \cdot E$$

with T the generalized tangent and

$$E = \frac{\partial R}{\partial y}$$

Again we are only typically interested in certain blocks of T.

However, calculating the whole generalized tangent is useful when doing adaptive substepping. A typical scheme might subdivide the input values of the explicit variables y into several smaller steps called substeps. Here we are typically interested in the tangent matrix over the whole step and not the individual tangents for each substep. As demonstrated by [PRH2001] there is a recursive formula for calculating the complete adaptive tangent. Consider the substepping scheme defined by:

$$y^{i+1} = y^i + \alpha_{i+1} \left(y_{n+1} - y_n\right)$$

The recursive formula for the tangent T^{i+1} covering the complete step from y_n to the current substep is

$$T^{i+1} = J_{i+1}^{-1} \cdot \left(\alpha_{i+1} E_{i+1} + T^i\right)$$

where the partial derivatives J_{i+1} and E_{i+1} are for the current substep. Applying this recursion relation though each substep produces the consistent tangent for the whole step.

Note this algorithm depends on propagating the whole generalized consistent tangent, not just the derivative of the stress with respect to the strain. This is because the history variables also evolve throughout the substepping. However, as described again in [PRH2001] some optimizations are possible. Only minor *columns* of T pertaining to the strain ε are required for standard FE codes and so the recursive relation can be restricted to the approach subblocks of the generalized tangent. Additionally, some types of internal variables, notably the plastic multiplier for rate independent plasticity models, do not propagate from substep to substep but instead reset with each substep. The minor *rows* for these sorts of internal variables can be omitted from the recursive propagation. Currently NEML does not make either optimization.

20.2 Mathematical helpers

NEML contains many helper functions to do tensor operations on Mandel-notation representations of tensors stored as pointers as well as common linear algebra operations like solving systems of equations and inverting matrices. These helpers are fairly self documenting. The interfaces are shown below.

Additionally, the *Crystal plasticity* module uses a new system for tensor operations using objects, rather than raw pointers. The new system considerably simplifies many common mathematical operations. The long-term plan for NEML is to switch the macroscale plasticity modules over to this new approach.

20.2.1 Tensor and rotation objects

Rotations

Rotations in NEML are stored as quaternions. In turn, these are stored as length-4 flat arrays.

NEML provides classes implementing general quaternions and unit quaternions, which are representations of 3D rotations (i.e. elements of the special orthogonal group). Currently, the general quaternion class is not used in NEML, it just serves as the base class for the Orientation class, which implements rotations.

Quaternion

This class represents a general quaternion, i.e. a quaternion that may not have $\|\mathbf{Q}\| = 1$. It's not used in NEML natively, just as a base class to the unit quaternion *orientation* class.

class **Quaternion**

A generic quaternion, stored as [s v1 v2 v3].

Subclassed by *neml::Orientation*

Public Functions

Quaternion()

Default constructor (manage own memory)

Quaternion(const std::vector<double> v)

Construct from vector (manage own memory)

Quaternion(double *v)

Construct from raw pointer (don't manage memory)

Quaternion(const *Quaternion* &other)

Copy constructor.

Quaternion(const *Quaternion* &&other)

Move constructor.

virtual **~Quaternion**()

Quaternion &**operator=**(const *Quaternion* &rhs)

Copy.

Quaternion &**operator=**(const *Quaternion* &&rhs)
Move.

bool **store**() const
Do you store your own data?

const double ***quat**() const
Raw quaternion as a const pointer.

double ***data**()
Raw quaternion as a nonconst pointer.

virtual double **norm**() const
Quaternion norm.

Quaternion **opposite**() const
Opposite.

Quaternion **operator-**() const
C++ operator opposite.

Quaternion **conj**() const
Conjugation.

Quaternion **flip**() const
Opposite scalar.

Quaternion **inverse**() const
Inversion.

Quaternion **exp**() const
Exponential map.

Quaternion **log**() const
Inverse exponential map.

Quaternion &**operator***=(const *Quaternion* &rhs)
Quaternion multiplication.

Quaternion &**operator***=(double scalar)
Scalar multiplication.

Quaternion &**operator/=**(const *Quaternion* &rhs)
Quaternion division.

Quaternion &**operator/=**(double scalar)
Scalar division.

Quaternion **pow**(double w) const
Power.

double **dot**(const *Quaternion* &other) const
Dot product, useful for various distances.

size_t **hash**() const
Hash function for quick comparisons.

void **to_product_matrix**(double *M) const
Product matrix for quaternion composition.

Orientation

This class represents unit quaternions, i.e. 3D rotations, i.e. members of the special orthogonal group $SO(3)$. It provides the `apply` method capable of applying a rotation to:

- Vector
- RankTwo
- Symmetric
- Skew
- RankFour
- SymSymR4

classes.

The class has static methods that can be used to create rotations from

- Euler angles
  - Kocks convention
  - Bunge convention
- Axis/angle pairs
- Rodrigues vectors
- Rotation matrices
- Hopf coordinates
- Hyperspherical coordinates

Similar methods can be used to convert the quaternion to these representations for output.

class **Orientation** : public *neml*::*Quaternion*

Subclassed by neml::CrystalOrientation

Public Functions

void **setRodrigues**(const double *const r)

Set from an input Rodrigues vector.

void **setMatrix**(const double *const M)

Set from an input matrix.

void **setAxisAngle**(const double *const n, double a, std::string angles = "radians")

Set from an input axis/angle pair.

void **setEulerAngles**(double a, double b, double c, std::string angles = "radians", std::string convention = "kocks")

Set from inputEuler angles.

void **setHopf**(double psi, double theta, double phi, std::string angles = "radians")

Set from input Hopf coordinates.

void **setHyperspherical**(double a1, double a2, double a3, std::string angles = "radians")

Set from input hyperspherical coordinates.

void **setVectors**(const *Vector* &x, const *Vector* &y)

Set from input two vectors.

Orientation()

Default constructor (defaults to identity, manage own memory)

Orientation(double *v)

Raw pointer constructor (don't manage memory)

Orientation(const std::vector<double> v)

vector<double> constructor (manage own memory)

Orientation(const *Quaternion* &other)

Copy constructor.

Orientation **deepcopy**() const

Explicit deepcopy.

void **to_euler**(double &a, double &b, double &c, std::string angles = "radians", std::string convention = "kocks") const

Convert to Euler angles.

void **to_axis_angle**(double *const n, double &a, std::string angles = "radians") const

Convert to an axis/angle pair.

void **to_matrix**(double *const M) const

Convert to a rotation matrix.

RankTwo **to_tensor**() const

Convert to a rank 2 tensor.

void **to_rodrigues**(double *const v) const

Convert to a Rodrigues vector.

void **to_hopf**(double &alpha, double &beta, double &gamma, std::string angles = "radians") const

Convert to Hopf coordinates.

void **to_hyperspherical**(double &a1, double &a2, double &a3, std::string angles = "radians") const

Convert to hyperspherical coordinates.

Orientation **opposite**() const

Opposite.

Orientation **operator-**() const

C++ operator opposite.

Orientation **conj**() const

Conjugation.

Orientation **flip**() const

Opposite scalar.

Orientation **inverse**() const

Inversion.

Orientation &**operator*=**(const *Orientation* &rhs)

Orientation multiplication.

Orientation &**operator/=**(const *Orientation* &rhs)

Orientation division.

Orientation **pow**(double w) const

Power.

Vector **apply**(const *Vector* &a) const

Rotate a *Vector*.

RankTwo **apply**(const *RankTwo* &a) const

Rotate a Rank2.

Symmetric **apply**(const *Symmetric* &a) const

Rotate a *Symmetric* tensor.

Skew **apply**(const *Skew* &a) const

Rotate a *Skew* tensor.

RankFour **apply**(const *RankFour* &a) const

Rotate a *RankFour* tensor.

SymSymR4 **apply**(const *SymSymR4* &a) const

Rotate a *SymSymR4* rank four tensor.

double **distance**(const *Orientation* &other) const

Geodesic distance.

Public Static Functions

static *Orientation* **createRodrigues**(const double *const r)

Create from a Rodrigues vector.

static *Orientation* **createMatrix**(const double *const M)

Create from a rotation matrix.

static *Orientation* **createAxisAngle**(const double *const n, double a, std::string angles = "radians")

Create from an axis-angle representation.

static *Orientation* **createEulerAngles**(double a, double b, double c, std::string angles = "radians", std::string convention = "kocks")

Create from various Euler angles.

static *Orientation* **createHopf**(double psi, double theta, double phi, std::string angles = "radians")

Create from the Hopf coordinates.

static *Orientation* **createHyperspherical**(double a1, double a2, double a3, std::string angles = "radians")

Create from hyperspherical coordinates.

static *Orientation* **createVectors**(const *Vector* &x, const *Vector* &y)

Create from two vectors.

As with the *Tensors* classes, NEML can either manage the memory of a quaternion or use externally managed memory, to allow for efficient block storage of data.

The quaternion classes provide several helpful unary operators, for example methods for inverting and exponentiating quaternions. The implementation also provides binary operators for composing quaternions (equivalent to composing rotations for the Orientation unit quaternion class). It also provides several specialized mathematical operators:

Warning: doxygenfunction: Unable to resolve function "neml::random_orientations" with arguments None in doxygen xml output for project "neml" from directory: ../class-doc/xml. Potential matches:

```
- std::vector<CrystalOrientation> random_orientations(int n)
- std::vector<CrystalOrientation> random_orientations(int n, unsigned long seed)
```

Orientation neml::**wexp**(const *Skew* &w)

Exponential map of a skew tensor in my convention.

Skew neml::**wlog**(const *Orientation* &q)

Inverse exponential map of a quaternion to a skew tensor in my convention.

double neml::**distance**(const *Orientation* &q1, const *Orientation* &q2)

Geodesic distance.

Orientation neml::**rotate_to**(const *Vector* &a, const *Vector* &b)

Arbitrary rotation from a to b.

Orientation neml::**rotate_to_family**(const *Vector* &a, const *Vector* &b, double ang)

Family of rotations from a to b parameterized by an angle.

Tensors

The tensor objects provide a method for easily managing scalar, vector, and tensor mathematical operations. The classes maintain a representation of the tensor type and provide common mathematical operations, both single tensors operations and binary operations between various types of tensors.

The classes are structured to either maintain their own memory or to use externally managed memory. The latter mode is useful in finite element calculations where the calling program can present a large collection of material points in a blocked array. If this type of memory management can be used it reduces copying when NEML models are called from external program and allows optimizing compilers to attempt vectorization.

The objects are set up to be transparent as to the type of memory model used to store the data. Generally, initializing an object with a raw pointer sets the objects to external memory management. This means that when the object is deleted the memory is not deallocated. Initializing an object with either a default (no parameter) constructor or a STL container sets the classes to manage their own memory and hence the memory will be freed when the object is deleted.

All the tensor classes derive from a common base class (see below). NEML implements the following tensor types:

Vector

This class represents a 3D vector, stored as a length 3 flat array.

class **Vector** : public *neml*::*Tensor*

Public Functions

Vector()

Vector(const std::vector<double> v)

Vector(double *v)

Vector(const double *v)

Vector **opposite**() const

Vector **operator-**() const

Vector &**operator+=**(const *Vector* &other)

Vector &**operator-=**(const *Vector* &other)

double &**operator()**(size_t i)

const double &**operator()**(size_t i) const

double **dot**(const *Vector* &rhs) const

RankTwo **outer**(const *Vector* &o) const

double **norm**() const

Vector **cross**(const *Vector* &other) const

Vector &**normalize**()

RankTwo

This class represents a general 3D rank two tensor stored as a length 9 flat array.

class **RankTwo** : public *neml*::*Tensor*

Full Rank 2 tensor.

Public Functions

RankTwo()

RankTwo(const std::vector<double> v)

RankTwo(double *v)

RankTwo(const double *v)

RankTwo(const std::vector<std::vector<double>> A)

RankTwo(const *Symmetric* &other)

Helper.

RankTwo(const *Skew* &other)

RankTwo **opposite**() const

RankTwo **operator-**() const

RankTwo &**operator+=**(const *RankTwo* &other)

RankTwo &**operator-=**(const *RankTwo* &other)

RankTwo &**operator+=**(const *Symmetric* &other)

RankTwo &**operator-=**(const *Symmetric* &other)

RankTwo &**operator+=**(const *Skew* &other)

RankTwo &**operator-=**(const *Skew* &other)

double &**operator()**(size_t i, size_t j)

const double &**operator()**(size_t i, size_t j) const

RankTwo **dot**(const *RankTwo* &other) const

RankTwo **dot**(const *Symmetric* &other) const

RankTwo **dot**(const *Skew* &other) const

Vector **dot**(const *Vector* &other) const

RankTwo **inverse**() const

RankTwo **transpose**() const

double **norm**() const

double **contract**(const *RankTwo* &other) const

double **contract**(const *Symmetric* &other) const

double **contract**(const *Skew* &other) const

Public Static Functions

static inline *RankTwo* **id**()

Symmetric

This class represents a symmetric 3D rank two tensor ($T_{ij} = T_{ji}$). Internally, the tensor is stored as a length 6 Mandel vector.

class **Symmetric** : public *neml*::*Tensor*

Symmetric Mandel rank 2 tensor.

Public Functions

Symmetric()

Symmetric(const std::vector<double> v)

Symmetric(double *v)

Symmetric(const double *v)

Symmetric(const *RankTwo* &other)

I guess take them seriously and symmetrize it.

RankTwo **to_full**() const

Symmetric **opposite**() const

Symmetric **operator-**() const

Symmetric &**operator+=**(const *Symmetric* &other)

Symmetric &**operator-=**(const *Symmetric* &other)

Symmetric **inverse**() const

Symmetric **transpose**() const

double **trace**() const

Symmetric **dev**() const

double **norm**() const

Vector **dot**(const *Vector* &other) const

Symmetric **dot**(const *Symmetric* &other) const

RankTwo **dot**(const *RankTwo* &other) const

RankTwo **dot**(const *Skew* &other) const

double **contract**(const *RankTwo* &other) const

double **contract**(const *Symmetric* &other) const

double **contract**(const *Skew* &other) const

double &**operator()**(size_t i)

const double &**operator()**(size_t i) const

Public Static Functions

static inline *Symmetric* **id**()

static inline *Symmetric* **zero**()

Skew

This class represents a skew-symmetric 3D rank two tensor ($T_{ij} = -T_{ji}$). Internally, the tensor is stored as a length 3 vector with the convention listed in the *introduction*.

class **Skew** : public *neml*::*Tensor*

Public Functions

Skew()

Skew(const std::vector<double> v)

Skew(double *v)

Skew(const double *v)

Skew(const *RankTwo* &other)

Skew a general tensor.

RankTwo **to_full**() const

Skew **opposite**() const

Skew **operator-**() const

Skew &**operator+=**(const *Skew* &other)

Skew &**operator-=**(const *Skew* &other)

Skew **transpose**() const

Vector **dot**(const *Vector* &other) const

Skew **dot**(const *Skew* &other) const

RankTwo **dot**(const *RankTwo* &other) const

RankTwo **dot**(const *Symmetric* &other) const

double **contract**(const *RankTwo* &other) const

double **contract**(const *Symmetric* &other) const

double **contract**(const *Skew* &other) const

Public Static Functions

static inline *Skew* **zero**()

RankFour

This class represents a general 3D rank four tensor stored as a length 81 flat array.

class **RankFour** : public *neml*::*Tensor*

Public Functions

RankFour()

RankFour(const std::vector<double> v)

RankFour(const std::vector<std::vector<std::vector<std::vector<double>>>> A)

RankFour(double *v)

RankFour(const double *v)

RankFour **opposite**() const

RankFour **operator-**() const

RankFour &**operator+=**(const *RankFour* &other)

RankFour &**operator-=**(const *RankFour* &other)

double &**operator()**(size_t i, size_t j, size_t k, size_t l)

const double &**operator()**(size_t i, size_t j, size_t k, size_t l) const

SymSymR4 **to_sym**() const

SymSkewR4 **to_symskew**() const

SkewSymR4 **to_skewsym**() const

RankFour **dot**(const *RankFour* &other) const

RankFour **dot**(const *SymSymR4* &other) const

RankFour **dot**(const *SymSkewR4* &other) const

RankFour **dot**(const *SkewSymR4* &other) const

RankTwo **dot**(const *RankTwo* &other) const

RankTwo **dot**(const *Symmetric* &other) const

RankTwo **dot**(const *Skew* &other) const

SymSymR4

This class represents a rank four tensor with symmetries $C_{ijkl} = C_{jikl}$, $C_{ijkl} = C_{ijlk}$, and $C_{ijkl} = C_{klij}$. It is stored as a length 36 vector with the convention described *here*.

class **SymSymR4** : public *neml*::*Tensor*

Public Functions

SymSymR4()

SymSymR4(const std::vector<double> v)

SymSymR4(const std::vector<std::vector<double>> A)

SymSymR4(double *v)

SymSymR4(const double *v)

SymSymR4 **opposite**() const

SymSymR4 **operator-**() const

SymSymR4 &**operator+=**(const *SymSymR4* &other)

SymSymR4 &**operator-=**(const *SymSymR4* &other)

RankFour **to_full**() const

double &**operator()**(size_t i, size_t j)

const double &**operator()**(size_t i, size_t j) const

SymSymR4 **inverse**() const

SymSymR4 **transpose**() const

RankFour **dot**(const *RankFour* &other) const

SymSymR4 **dot**(const *SymSymR4* &other) const

RankFour **dot**(const *SymSkewR4* &other) const

RankFour **dot**(const *SkewSymR4* &other) const

RankTwo **dot**(const *RankTwo* &other) const

RankTwo **dot**(const *Skew* &other) const

Symmetric **dot**(const *Symmetric* &other) const

Public Static Functions

static inline *SymSymR4* **id**()

static inline *SymSymR4* **id_dev**()

static inline *SymSymR4* **zero**()

SymSkewR4

This class represents a rank four tensor with symmetries $C_{ijkl} = C_{jikl}$ and $C_{ijkl} = -C_{ijlk}$. It is stored as a length 18 flat array in a convention that makes it the natural way to store the derivative of a symmetric rank 2 tensor with respect to a skew symmetric rank 2 tensor.

class **SymSkewR4** : public *neml*::*Tensor*

Public Functions

SymSkewR4()

SymSkewR4(const std::vector<double> v)

SymSkewR4(const std::vector<std::vector<double>> A)

SymSkewR4(double *v)

SymSkewR4(const double *v)

SymSkewR4 **opposite**() const

SymSkewR4 **operator-**() const

SymSkewR4 &**operator+=**(const *SymSkewR4* &other)

SymSkewR4 &**operator-=**(const *SymSkewR4* &other)

RankFour **to_full**() const

double &**operator()**(size_t i, size_t j)

const double &**operator()**(size_t i, size_t j) const

RankFour **dot**(const *RankFour* &other) const

RankFour **dot**(const *SymSymR4* &other) const

RankFour **dot**(const *SymSkewR4* &other) const

RankFour **dot**(const *SkewSymR4* &other) const

RankTwo **dot**(const *RankTwo* &other) const

RankTwo **dot**(const *Skew* &other) const

RankTwo **dot**(const *Symmetric* &other) const

SkewSymR4

This class represents a rank four tensor with symmetries $C_{ijkl} = -C_{jikl}$ and $C_{ijkl} = C_{ijlk}$. It is stored as a length 18 flat array in a convention that makes it the natural way to store the derivative of a symmetric rank 2 tensor with respect to a skew symmetric rank 2 tensor.

class **SkewSymR4** : public *neml*::*Tensor*

Public Functions

SkewSymR4()

SkewSymR4(const std::vector<double> v)

SkewSymR4(const std::vector<std::vector<double>> A)

SkewSymR4(double *v)

SkewSymR4(const double *v)

SkewSymR4 **opposite**() const

SkewSymR4 **operator-**() const

SkewSymR4 &**operator+=**(const *SkewSymR4* &other)

SkewSymR4 &**operator-=**(const *SkewSymR4* &other)

RankFour **to_full**() const

double &**operator()**(size_t i, size_t j)

const double &**operator()**(size_t i, size_t j) const

RankFour **dot**(const *RankFour* &other) const

RankFour **dot**(const *SymSymR4* &other) const

RankFour **dot**(const *SkewSymR4* &other) const

RankFour **dot**(const *SymSkewR4* &other) const

RankTwo **dot**(const *RankTwo* &other) const

RankTwo **dot**(const *Skew* &other) const

RankTwo **dot**(const *Symmetric* &other) const

Binary operators

The module provides a large variety of binary operators. All scalar multiplication and division operations are covered as are all rational addition/subtraction operations between tensors. Implemented tensor products include the outer products between vectors ($C_{ij} = a_i b_j$) and rank two tensors ($C_{ijkl} = A_{ij} B_{kl}$), matrix-vector products ($c_j = A_{ij} b_j$), matrix-matrix products ($C_{ij} = A_{ik} B_{kj}$), rank four/rank two composition ($C_{ij} = A_{ijkl} B_{kl}$), rank four/rank four composition ($C_{ijkl} = A_{ijmn} B_{mnkl}$), and a few specialized operators that occur in the crystal plasticity integration formula.

These binary operators are implemented for all sensible combinations of specialized tensor types (for example, all RankTwo binary operations can be completed with any combination of RankTwo, Symmetric, and Skew tensors).

Base class description

class **Tensor**

Subclassed by *neml::RankFour*, *neml::RankTwo*, *neml::Skew*, *neml::SkewSymR4*, *neml::Symmetric*, *neml::SymSkewR4*, *neml::SymSymR4*, neml::SymSymSymR6, *neml::Vector*

Public Functions

Tensor(std::size_t n)

Tensor(const *Tensor* &other)

Tensor(*Tensor* &&other)

Tensor(const std::vector<double> flat)

Tensor(double *flat, size_t n)

Tensor(const double *flat, size_t n)

virtual **~Tensor**()

inline bool **istore**() const

Do I own my own data?

Tensor &**operator=**(const *Tensor* &rhs)

Tensor &**operator=**(*Tensor* &&rhs)

inline const double ***data**() const

inline double ***s**()

inline std::size_t **n**() const

void **copy_data**(const double *const indata)

Tensor &**operator*=**(double s)

Tensor &**operator/=**(double s)

20.2.2 Generic matrix system

Matrix classes

The purpose of this small class system is to provide an easy way for users to design slip interaction matrices. These are square matrices constructed with certain typical block patterns.

FlatVector

This is just an interface to a standard, dense vector use to provide matrix-vector products using the Matrix classes

class **FlatVector**

Dense vector class.

Public Functions

FlatVector(size_t n)

FlatVector(size_t n, double *data)

FlatVector(const std::vector<double> input)

FlatVector(const *FlatVector* &other)

virtual **~FlatVector**()

inline size_t **n**() const

inline bool **owns_data**() const

void **copy**(double *data)

inline double ***data**()

inline const double ***data**() const

Matrix

Generic superclass of all matrices, could be expanded to handle sparse matrices if required in the future.

class **Matrix**

Dense matrix class.

Subclassed by *neml::SquareMatrix*

Public Functions

Matrix(size_t m, size_t n)

virtual **~Matrix**()

inline size_t **m**() const

inline size_t **n**() const

inline size_t **size**() const

FlatVector **dot**(const *FlatVector* &other)

void **matvec**(const *FlatVector* &other, *FlatVector* &res)

inline double ***data**()

const double &**operator()**(size_t i, size_t j) const

double &**operator()**(size_t i, size_t j)

SquareMatrix

Dense, square matrix with various helpers for setting up block structures. Initialization options are:

Option	Description
zero	The zero matrix
identity	The identity matrix
diagonal	Diagonal matrix where the user gives the diagonal entries
diagonal_blocks	Diagonal matrix where the entries are blocked
block	General block matrix
dense	Dense matrix, user gives all entries in row-major form

class **SquareMatrix** : public *neml*::*NEMLObject*, public *neml*::*Matrix*

Specialized square dense matrix.

Public Functions

SquareMatrix(*ParameterSet* &params)

Public Static Functions

static std::string **type**()

String type for the object system.

static std::unique_ptr<*NEMLObject*> **initialize**(*ParameterSet* &params)

Initialize from a parameter set.

static *ParameterSet* **parameters**()

Default parameters.

20.2.3 Pointer array math functions

namespace **neml**

Functions

void **SymSymR4SkewmSkewSymR4SymR4**(const double *const M, const double *const W, double *const SS)
Specialty crystal plasticity operator: M_kmab*W_ml - W_km*M_mlab.

void **SymSkewR4SymmSkewSymR4SymR4**(const double *const D, const double *const M, double *const SS)
Specialty crystal plasticity operator: D_km * M_mlab - M_kmab D_ml.

void **SpecialSymSymR4Sym**(const double *const D, const double *const M, double *const SW)
Specialty operator for the skew part of the tangent: C_ijkb * e_ka - C_ijal * e_bl.

void **transform_fourth**(const double *const D, const double *const W, double *const M)
Convert the symmetric and skew parts voido a complete fourth order.

void **truesdell_tangent_outer**(const double *const S, double *const M)
The outer product used in constructing the truesdell tangent.

void **full2skew**(const double *const A, double *const M)
Convert a 9x9 to a skew derivative matrix.

void **skew2full**(const double *const M, double *const A)
Convert a skew derivative matrix to a 9x9.

void **full2wws**(const double *const A, double *const M)
Convert a 9x9 voido a ws matrix.

void **wws2full**(const double *const M, double *const A)
Convert a 3x6 ws matrix voido a 9x9.

void **full2mandel**(const double *const A, double *const M)
Convert a 9x9 to a mandel matrix.

void **mandel2full**(const double *const M, double *const A)
Convert a mandel matrix to a full 9x9.

void **truesdell_update_sym**(const double *const D, const double *const W, const double *const Sn, const double *const So, double *const Snp1)
Convect a symmetric tensor with a Truesdell rate.

void **truesdell_mat**(const double *const D, const double *const W, double *const M)
Form the 9x9 matrix used in the update.

void **truesdell_rhs**(const double *const D, const double *const W, const double *const Sn, const double *const So, double *const St)
Form the RHS of the update, as a symmetric vector.

void **sym**(const double *const A, double *const v)
Convert a full symmetric rank 2 to a Mandel vector.

void **usym**(const double *const v, double *const A)
Convert a symmetric vector to a full matrix.

void **skew**(const double *const A, double *const v)

Convert a full skew rank 2 to a skew vector.

void **uskew**(const double *const v, double *const A)

Convert a skew vector to a full matrix.

void **minus_vec**(double *const a, int n)

Negate a vector in place.

void **add_vec**(const double *const a, const double *const b, int n, double *const c)

Add vectors.

void **sub_vec**(const double *const a, const double *const b, int n, double *const c)

Subtract vectors.

double **dot_vec**(const double *const a, const double *const b, int n)

Compute a dot product.

double **norm2_vec**(const double *const a, int n)

Compute a two norm.

void **normalize_vec**(double *const a, int n)

Normalize a vector in place (2-norm)

void **dev_vec**(double *const a)

Return the deviatoric vector.

void **outer_vec**(const double *const a, int na, const double *const b, int nb, double *const C)

Outer product of two vectors.

void **outer_update**(const double *const a, int na, const double *const b, int nb, double *const C)

Rank 2 update.

void **outer_update_minus**(const double *const a, int na, const double *const b, int nb, double *const C)

Rank 2 update.

void **mat_vec**(const double *const A, int m, const double *const b, int n, double *const c)

Matrix-vector c = A . b.

void **mat_vec_trans**(const double *const A, int m, const double *const b, int n, double *const c)

Matrix-vector c = A.T . b.

void **invert_mat**(double *const A, int n)

Invert a matrix in place.

void **mat_mat**(int m, int n, int k, const double *const A, const double *const B, double *const C)

void **mat_mat_ABT**(int m, int n, int k, const double *const A, const double *const B, double *const C)

void **solve_mat**(const double *const A, int n, double *const x)

Solve unsymmetric system.

double **condition**(const double *const A, int n)

Get the condition number of a matrix.

double **polyval**(const std::vector<double> &poly, double x)

Evaluate a polynomial with Horner's method, highest order term first.

std::vector<double> **poly_from_roots**(const std::vector<double> &roots)

Construct a polynomial with the given roots.

std::vector<double> **differentiate_poly**(const std::vector<double> &poly, int n = 1)

Get the derivative of a polynomial.

int **gcd**(int a, int b)

The greatest common divisor between two numbers.

int **common_gcd**(std::vector<int> in)

The greatest common divisor between a lot of numbers.

std::vector<int> **reduce_gcd**(std::vector<int> in)

Divide a vector by the collective GCD.

double **convert_angle**(double a, std::string)

Convert to radians.

double **cast_angle**(double a, std::string angles)

Convert from radians.

void **qmult_vec**(const double *const As, const double *const B, size_t n, double *const C)

Vectorized quaternion multiplication.

bool **isclose**(double a, double b)

This is only to be used for testing.

void **rotate_matrix**(int m, int n, const double *const A, const double *const B, double *C)

Perform A * B * A.T.

int **fact**(int n)

Factorial.

double **factorial**(int n)

Factorial as a double + cacheing.

void **eigenvalues_sym**(const double *const s, double *values)

Get the eigenvalues of a symmetric 3x3 matrix in Mandel notation.

void **eigenvectors_sym**(const double *const s, double *vectors)

Get the eigenvectors of a symmetric 3x3 matrix (row major)

double **I1**(const double *const s)

First principal invariant.

double **I2**(const double *const s)

Second principal invariant.

20.3 History object system

20.3.1 Overview

The History class stores internal variables of material models. Originally, NEML stored internal variables in a flat pointer array. This meant that the programmer or end user had to manually track which variable was located at which index in the array and, for non-scalar variables, correctly flatten and interpret the data.

The History class helps manage model internal variables. Fundamentally, it is a dictionary that returns and stores an internal variable associated with a string key naming the actual variable in the model implementation. The class also manages types so that it returns the correct type of object. For example, the History class could manage a `"direction"` internal variable with type `Vector`. The user can request the object return `"direction"` and the History object would return the correct `Vector` type.

Currently, the class is configured to store and return objects of the following types:

- Scalars (i.e. `double`)
- *Vector*
- *RankTwo*
- *Symmetric*
- *Skew*
- *Orientation*
- *SymSymR4*

Internally, all of these classes are stored in a single flat array. The system can either manage its own memory (freeing it on deletion of the object) or accept a pointer to externally managed memory. For example, a finite element analysis program can pass in the material model history as a large array in memory and NEML can *wrap* that memory using the History class and give descriptive access to each variable *without* copying the underlying data. This allows for seamlessly converting a flat array containing the model internal variables into an instance of the History class without copying. It also allows the calling program to block the internal variables of several material points, potentially allowing the compiler to perform vector optimizations.

Oftentimes, implementations of material models will need to store the derivatives of a set of internal variables. The History class contains methods for duplicating a given History instance but changing the types (and corresponding storage) of the new instance to reflect the types appropriate for storing the derivative of the original History with respect to some other object type.

20.3.2 Class description

class **`History`**

Public Functions

History()
Default constructor (manage own memory)

History(bool store)
Default constructor (option to not manage memory)

History(const *History* &other)
Copy constructor.

History(const *History* &&other)
Move constructor.

History(double *data)
Dangerous constructor, only use if you know what you're doing.

History(const double *data)
Dangerous constructor, only use if you know what you're doing.

virtual **~History**()
Destructor.

History &**operator=**(const *History* &other)
Copy.

History &**operator=**(const *History* &&other)
Move.

History **deepcopy**() const
Explicit deepcopy.

inline bool **store**() const
Do I own my own data?

inline const double ***rawptr**() const
Raw data pointer (const)

inline double ***rawptr**()
Raw data pointer (nonconst)

void **set_data**(double *input)
Set storage to some external pointer.

void **copy_data**(const double *const input)
Copy data from some external pointer.

size_t **size**() const
Size of storage required.

void **make_store**()
Convert to store.

template<typename **T**>
inline void **add**(std::string name)
Add a generic object.

void **add**(std::string name, StorageType type, size_t size)

Add known object.

template<class **T**>
inline *item_return*<*T*>::type **get**(std::string name) const

Get an item (provide with correct class)

inline double ***get_data**(std::string name)

Get a pointer to the raw location of an item.

inline const std::unordered_map<std::string, size_t> &**get_loc**() const

Get the location map.

inline const std::unordered_map<std::string, StorageType> &**get_type**() const

Get the type map.

inline const std::vector<std::string> &**get_order**() const

Get the name order.

inline std::unordered_map<std::string, size_t> &**get_loc**()

Get the location map.

inline std::unordered_map<std::string, StorageType> &**get_type**()

Get the type map.

inline std::vector<std::string> &**get_order**()

Get the order.

inline const std::vector<std::string> &**items**() const

Return all the items in this object.

size_t **size_of_entry**(std::string name) const

Helper to get the size of a particular object.

void **resize**(size_t inc)

Resize method.

void **increase_store**(size_t newsize)

Actually increase internal storage.

void **scalar_multiply**(double scalar)

Multiply everything by a scalar.

History &**operator+=**(const *History* &other)

Add another history to this one.

History &**add_union**(const *History* &other)

Combine another history object through a union.

History **copy_blank**(std::vector<std::string> exclude = {}) const

Make a blank copy.

void **copy_maps**(const *History* &other)

Copy over the order maps.

History &**zero**()

Make zero.

template<class **T**>

inline *History* **derivative**() const

Make a *History* appropriate to hold the derivatives of the indicated items.

History **history_derivative**(const *History* &other) const

Derivative with respect to a different history.

History **split**(std::vector<std::string> sep, bool after = true) const

Split a history in two.

History **subset**(std::vector<std::string> vars) const

Extract a subset.

History &**reorder**(std::vector<std::string> names)

Reorder based on the provided list of names.

inline bool **contains**(std::string name) const

Quick function to check to see if something is in the vector.

History **postmultiply**(const *SymSymR4* &T)

Postmultiply by various objects.

void **unravel_hh**(const *History* &base, double *const array)

This unravels a history derivative into row major storage.

double ***start_loc**(std::string name)

Starting location of an entry.

std::vector<std::string> **formatted_names**() const

Nicely formatted names for the flat storage.

template<>
inline *History*::*item_return*<double>::type **get**(std::string name) const

Special case for a double.

template<>
inline *History* **derivative**() const

Special case for self derivative.

template<class **T**>

struct **item_return**

Helper for template magic.

template<>

struct **item_return**<double>

20.4 Solver interface

A common task in NEML is solving a nonlinear system of equations. For example, because NEML uses implicit time integration it must solve a nonlinear system each time it integrates a material model defined with a rate form. NEML provides an opaque mechanism for using different nonlinear equation solvers throughout the code. It does this by having objects that need to solve nonlinear equations solvable by inheriting from *neml::Solvable*.

class **Solvable**

Generic nonlinear solver interface.

Subclassed by *neml::CreepModel*, *neml::LarsonMillerRelation*, *neml::NEMLScalarDamagedModel_sd*, *neml::SingleCrystalModel*, *neml::SmallStrainCreepPlasticity*, neml::SubstepModel_sd, neml::TestPower

Public Functions

virtual size_t **nparams**() const = 0

Number of parameters in the nonlinear equation.

virtual void **init_x**(double *const x, *TrialState* *ts) = 0

Initialize a guess to start the solution iterations.

virtual void **RJ**(const double *const x, *TrialState* *ts, double *const R, double *const J) = 0

Nonlinear residual equations and corresponding jacobian.

This class requires an object implement three virtual methods:

1. `nparams`: Return the number of variables in the nonlinear system to be solved.
2. `init_x`: Given a vector of length `nparams` and a *neml::TrialState* object setup an initial guess to start the nonlinear solution iterations.
3. `RJ`: Given the current guess at the solution `x` (length `nparams`) and the *neml::TrialState* object return the residual equations (`R`, length `nparams`) and the Jacobian of the residual equations with respect to the variables (`J`, `nparams` $\times$ `nparams`).

A *neml::TrialState* is a completely generic object that contains any information beyond the current guess at the solution the class will need to construct an initial guess and to calculate the residual and the Jacobian. Essentially, it contains any variables that are held fixed during the nonlinear solution process. While *neml::TrialState* objects all inherit from a base class, this is currently entirely cosmetic.

class **TrialState**

Trial state Store data the solver needs and can be passed into solution interface

Subclassed by neml::CreepModelTrialState, neml::GITrialState, neml::LMTrialState, neml::SCTrialState, neml::SDTrialState, neml::SSCPTrialState, neml::SSPPTrialState, neml::SSRIPTrialState

Essentially they are C++ structs holding field data. Below is an example TrialState for the *neml::SmallStrainPerfectPlasticity* object.

```
/// Small strain perfect plasticity trial state
//  Store data the solver needs and can be passed into solution interface
class SSPPTrialState : public TrialState {
 public:
  double ys, T;    // Yield stress, temperature
  double ee_n[6];  // Previous value of elastic strain
  double s_n[6];   // Previous stress
  double s_tr[6];  // Elastic trial stress
  double e_np1[6]; // Next strain
  double e_n[6];   // Previous strain
  double S[36];    // Compliance
  double C[36];    // Stiffness
};
```

All this is a `struct` containing the information required to setup the nonlinear residual equations. Note this object does not contain the current value of stress or history. This information is contained (and updated) in the solution vector `x`.

Currently, NEML has two options for solving nonlinear equations. NEML contains a built-in implementation of the Newton-Raphson method or NEML can use the *NOX <https://trilinos.org/packages/nox-and-loca/>* solver contained in the *Trilinos <https://trilinos.org/>* package, developed by Sandia National Laboratories. The solver is configured at build time, using the CMake configuration.

20.5 Object and input management

20.5.1 Using the object system

NEML uses a factory system tied to a ParameterSet class to create objects and manage input from python or XML data files. The system is setup to incur minimal work when setting up a new object while ensuring that the input syntax for the XML files is generated automatically and will be consistent with the C++ library.

All a user needs to do to include a new object in the global factory and make it available from the XML input is:

1. Inherit from *neml::NEMLObject*.
2. **Implement three static class methods:**
    1. `static std::string type()`
    2. `static ParameterSet parameters()`
    3. `static std::unique_ptr<NEMLObject> initialize(ParamemterSet & params)`
3. In the header file describing the object include `static Register<ObjectName> regObjectName` where `ObjectName` is the name of the new class.

`type()` must return a string type of the object, used to refer to it in the factory. This must be unique across all the NEML objects and we require it to match the class name for classes to be merged into the main NEML project. This example shows the implementation of this method for *neml::SmallStrainPerfectPlasticity*

```
std::string SmallStrainPerfectPlasticity::type()
{
  return "SmallStrainPerfectPlasticity";
}
```

`parameters()` returns a ParameterSet object that informs the object system what parameters the object will require and sets values for default parameters, if the object has any. The implementation for *neml::SmallStrainPerfectPlasticity* is

```
ParameterSet SmallStrainPerfectPlasticity::parameters()
{
  ParameterSet pset(SmallStrainPerfectPlasticity::type());

  pset.add_parameter<NEMLObject>("elastic");
  pset.add_parameter<NEMLObject>("surface");
  pset.add_parameter<NEMLObject>("ys");

  pset.add_optional_parameter<NEMLObject>("alpha",
                                          std::make_shared<ConstantInterpolate>(0.0));
  pset.add_optional_parameter<double>("tol", 1.0e-8);
  pset.add_optional_parameter<int>("miter", 50);
```

(continues on next page)

(continued from previous page)

```
  pset.add_optional_parameter<bool>("verbose", false);
  pset.add_optional_parameter<int>("max_divide", 8);

  return pset;
}
```

Note that other NEML objects must be added as type `NEMLObject` and not their subclass type.

`initialize(ParameterSet &)` initializes an object from a parameter set, returning it as a `std::unique_ptr<NEMLObject>`. *neml::SmallStrainPerfectPlasticity* implements it as

```
std::unique_ptr<NEMLObject> SmallStrainPerfectPlasticity::initialize(ParameterSet &␣
↪params)
{
  return make_unique<SmallStrainPerfectPlasticity>(
      params.get_object_parameter<LinearElasticModel>("elastic"),
      params.get_object_parameter<YieldSurface>("surface"),
      params.get_object_parameter<Interpolate>("ys"),
      params.get_object_parameter<Interpolate>("alpha"),
      params.get_parameter<double>("tol"),
      params.get_parameter<int>("miter"),
      params.get_parameter<bool>("verbose"),
      params.get_parameter<int>("max_divide")
      );
}
```

Finally, including the static registration class in the object header registers it automatically with the factory.

20.5.2 XML input

The XML input system automatically picks up the correct parameters from the `parameters()` static method. For example, the following XML creates a *neml::SmallStrainPerfectPlasticity* object. Notice how the parameter names match those provided in the definition of the C++ object. The input system knows how to recursively ask for parameters to define the other types of NEML objects needed. For example, this *neml::SmallStrainPerfectPlasticity* requires a *neml::LinearElasticModel* and the parameters all require *neml::Interpolate* objects.

```
<test_perfect type="SmallStrainPerfectPlasticity">
 <elastic type="IsotropicLinearElasticModel">
   <m1 type="PolynomialInterpolate">
     <coefs>
       -100.0 100000.0
     </coefs>
   </m1>
   <m1_type>youngs</m1_type>
   <m2>0.3</m2>
   <m2_type>poissons</m2_type>
 </elastic>

 <surface type="IsoJ2"/>

 <ys type="PiecewiseLinearInterpolate">
   <points>100.0   300.0 500.0 700.0</points>
```

(continues on next page)

(continued from previous page)

```
  <values>1000.0  120.0 60.0  30.0 </values>
 </ys>

 <alpha type="ConstantInterpolate">
  <v>0.1</v>
 </alpha>
</test_perfect>
```

20.5.3 NEMLObject

class **NEMLObject**

NEMLObject base calls for serialization.

Subclassed by *neml::CreepModel*, neml::CrystalOrientation, neml::CrystalPostprocessor, *neml::CubicLattice*, *neml::EffectiveStress*, *neml::FluidityModel*, *neml::GammaModel*, neml::GeneralizedHuCocksSpecies, *neml::GeneralLattice*, *neml::GFlow*, *neml::HCPLattice*, neml::HistoryNEMLObject, neml::InternalVariable< V >, *neml::Interpolate*, *neml::LarsonMillerRelation*, *neml::LinearElasticModel*, *neml::ScalarCreepRule*, *neml::ScalarDamage*, *neml::SlipPlaneDamage*, *neml::SofteningModel*, *neml::SquareMatrix*, *neml::SymmetryGroup*, *neml::ThermalScaling*, *neml::TransformationFunction*, *neml::YieldSurface*

Public Functions

virtual *ParameterSet* &**current_parameters**()

Return the current parameter set, including any updates from construction.

std::string **serialize**(std::string object_name = "object", std::string top_node = "")

Serialize an object to ASCII XML.

20.5.4 ParameterSet

class **ParameterSet**

Parameters for objects created through the *NEMLObject* interface.

Public Functions

ParameterSet()

Default constructor, needed to push onto stack.

ParameterSet(std::string type)

Constructor giving object type.

const std::string &**type**() const

Return the type of object you're supposed to create.

template<typename **T**>
inline void **add_parameter**(std::string name)

Add a generic parameter with no default.

inline void **assign_parameter**(std::string name, param_type value)

Immediately assign an input of the right type to a parameter.

template<typename **T**>
inline void **add_optional_parameter**(std::string name, param_type value)

Add a generic parameter with a default.

template<typename **T**>
inline *T* **get_parameter**(std::string name)

Get a parameter of the given name and type.

void **assign_defered_parameter**(std::string name, *ParameterSet* value)

Assign a parameter set to be used to create an object later.

template<typename **T**>
inline std::shared_ptr<*T*> **get_object_parameter**(std::string name)

Helper method to get a *NEMLObject* and cast it to subtype in one go.

template<typename **T**>
inline std::vector<std::shared_ptr<*T*>> **get_object_parameter_vector**(std::string name)

Helper to get a vector of NEMLObjects and cast them to subtype in one go.

ParamType **get_object_type**(std::string name)

Get the type of parameter.

bool **is_parameter**(std::string name) const

Check if this is an actual parameter.

std::vector<std::string> **unassigned_parameters**()

Get a list of unassigned parameters.

bool **fully_assigned**()

Check to make sure this parameter set is ready to go.

inline const std::vector<std::string> &**param_names**() const

Name getter.

20.6 Python bindings

To be added later.

20.7 Adding an object to NEML

To be added later.

CHAPTER

TWENTYONE

REFERENCES

BIBLIOGRAPHY

[TLK1991] Tome, C. N. and Lebensohn, R. A. and U. F. Kocks. "A model for texture development dominated by deformation twinning: Application to zirconium alloys." *Acta Metallurgica et Materialia,* 39(11): pp. 2667-2680 (1991).

[VM2021] Venkataraman, A. and M. C. Messner. *An initial framework for the rapid qualification of long-term creep rupture strength via microstrucetural modeling*. Argonne National Laboratory technical report ANL-21/34 (2021).

[HC2020] Hu, Jianan and Green, Graham and Hogg, Simon and Higginson, Rebecca and Alan Cocks. "Effect of microstructure evoltuion on the creep properties of polycrystalline 316H austenitic stainless steel." *Materials Science and Engineering A,* 772: pp. 138787 (2020).

[MS2020] Messner, M. C. and T.-L. Sham. *Initial High Temperature Inelastic Constitutive Model for Alloy 617*. Argonne National Laboratory technical report ANL-ART-195 (2020).

[SW2008] Sham, T.-L. and Kevin P. Walker. "Preliminary Development of a Unified Viscoplastic Constitutive Model for Alloy 617 with Special Reference to Long Term Creep Behavior." In the proceedings of the *Fourth International Topical Meeting on High Temperature Reactor Technology* pp. 81-89 (2008).

[WK1978] Walker, K. P. and E. Krempl. "An implicit functional representation of stress-strain behavior." *Mechanics Research Communications*, 5(4): pp. 185-190 (1978).

[MTS1999] Goto, D. M., Garrett, R. K., Bingert, J., Chen, Shuh-Rong, and George T. Gray. *Mechanical Threshold Stress Constitutive Strength Model Description of HY-100 Steel,* Indian Head Division, Naval Surface Warefare Center Technical Report IHTR 2168 (1999).

[KM2003] Kocks, U. and H. Mecking. "Physics and phenomenology of strain hardening: The FCC case," *Progress in Materials Science,* 48(3): pp. 171-273 (2003).

[SH1997] Simo, J. C. and T. J. R. Hughes. *Computational Inelasticity,* Springer-Verlang: New York (1997).

[D1959] D. C. Drucker. "A definition of a stable inelastic material," *ASME Journal of Applied Mechanics,* 6: pp. 101-195 (1959).

[FA2007] Frederick, C. and P. Armstrong. "A mathematical representation of the multiaxial Bauschinger effect," *Materials at High Temperatures,* 24(1): pp. 1-26 (2007).

[P1966] P. Perzyna. "Fundamental problems in viscoplasticity," *Advances in Applied Mechanics,* 9(2): pp. 244-368 (1966).

[YT2000] Yaguchi, M. and Y. Takahashi. "A viscoplastic constitutive model incorporating dynamic strain aging effect during cyclic deformation conditions," *International Journal of Plasticity,* 16(3-4): pp. 241-262 (2000).

[YT2005] Yaguchi, M. and Y. Takahashi. "Ratchetting of viscoplastic material with cyclid softening, Part 2: application to constitutive models," *International Journal of Plasticity,* 21: pp. 835-860 (2005).

[C2008] J. L. Chaboche. "A review of some plasticity and viscoplasticity constiutive theories," *International Journal of Plasticity,* 24(10): pp. 1642-1693 (2008).

[C1989a] Chaboche, J. L. and D. Nouailhas. "A unified constitutive model for cyclic viscoplasticity and its application to various stainless steels," *Journal of Engineering Materials and Technology,* 111: pp. 424-430 (1989).

[C1989b] J. L. Chaboche. "Constitutive equations for cyclic plasticity and cyclic viscoplasticity," *International Journal of Plasticity,* 5: pp. 247-302 (1989).

[BMD1969] Bird, J. E., Mukherjee, A. K., and J. E. Dorn. "Correlations between high-temperature creep behavior and structure," In the proceedings of *The International Conference on Quantitive Relation Between Properties and Microstructure,* Haifa, Israel (1969).

[HL1977] Hayhurst, D. R. and F. A. Leckie. "Constutitive equations for creep rupture," *Acta Metallurgica,* 25(9): pp. 1059-1070 (1977).

[H1985] Huddleston, R. "An improved multiaxial creep-rupture strength criterion," *Journal of Pressure Vessel Technology,* 107(Nov): pp. 421-429 (1985).

[B1972] Blackburn, L. "Isochronous Stress-Strain Curves for Austenitic Stainless Steels," In *The Generation of Isochronous Stress-Strain Curves,* ed. A. Shaefer, American Society of Mechanical Engineers, New York, NY: pp. 15-48 (1972).

[S1999] Swindeman, R. "Construction of isochronous stress-strain curves for 9Cr-1Mo-V steel," *Advances in Life Prediction Methodology,* 391: pp. 95-100 (1999).

[A1983] Asaro, R. "Micromechanics of crystals and polycrystals," *Advances in Applied Mechanics,* 25(C): pp. 1-115 (1983).

[M2015] Messner, M. C., Beaudoin, A. and R. H. Dodds, Jr. "Consistent crystal plasticity kinematics and linearization for the implicit finite element method," *Engineering Computations,* 32(6): pp. 1526-1548 (2015).

[L1969] Lee, E. H. "Elastic-plastic deformation at finite strains," *ASME Journal of Applied Mechanics,* 36: pp. 1-6 (1969).

[LM1952] Larson, F. and J. Miller. "A time-temperature relationship for rupture and creep stresses," *Transactions of the American Society of Mechanical Engineers,* 74: pp. 765-771 (1952).

[DHT2019] Das, S., Hofmann, F., and E. Tarleton. "Consistent determination of geometrically necessary dislocation density from simulations and experiments," *International Journal of Plasticity*, 109 pp. 18–42 (2018).

[PRH2001] Perez-Foguet, A., Rodrigues-Ferran, A. and A. Huerta. "Consistent tangent matrices for substepping schemes," *Computer Methods in Applied Mechanics and Engineering*, 190(35-36): pp. 4627-4647 (2001).

INDEX

Symbols

E

F

M

N

R