fdfault Documentation

Release 1.0

Eric G. Daub

Apr 13, 2017

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

`fdfault` is a finite difference code for numerical simulation of elastodynamic fracture and friction problems in 2 and 3 dimensions, principally those arising in study of dynamic earthquake rupture. The code solves the elastic or elastic-plastic wave equation in the bulk material, coupled to frictional failure on the fault and external boundary conditions.

1.1 Governing Equations

1.1.1 Material Governing Equations

The code solves the elastodynamic wave equation in 2 or 3 dimensions, with either an elastic or viscoplastic bulk material. For a 3D continuum, momentum balance requires that

$$\begin{aligned}
\rho\frac{\partial v_x}{\partial t} &= \frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{xy}}{\partial y} + \frac{\partial \sigma_{xz}}{\partial z} \\
\rho\frac{\partial v_y}{\partial t} &= \frac{\partial \sigma_{xy}}{\partial x} + \frac{\partial \sigma_{yy}}{\partial y} + \frac{\partial \sigma_{yz}}{\partial z} \\
\rho\frac{\partial v_z}{\partial t} &= \frac{\partial \sigma_{xz}}{\partial x} + \frac{\partial \sigma_{yz}}{\partial y} + \frac{\partial \sigma_{zz}}{\partial z},
\end{aligned}$$

where ρ is density. Additionally, a constitutive law relates the stresses to the deformations. For a homogeneous, isotropic elastic-plastic material, these take the form

$$\frac{\partial \sigma_{ij}}{\partial t} = L_{ijkl}\left(\dot{\epsilon}_{kl} - \dot{\epsilon}_{kl}^{pl}\right)$$

where the elastic tensor is given by Hooke's Law for a homogeneous isotropic material

$$L_{ijkl}\dot{\epsilon}_{kl} = \lambda\delta_{ij}\frac{\partial v_k}{\partial x_k} + G\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right).$$

The elastic tensor includes two material parameters: λ is the first Lamé parameter and G is the shear modulus. Summation over repeated indices is implied. For an elastic material, $\dot{\epsilon}_{ij}^{pl} = 0$, while for a viscoplastic material the plastic strains are determined by the Drucker-Prager viscoplastic flow rule described below. For 2D problems, the code can handle either in-plane (v_x, v_y, σ_{xx}, σ_{xy}, σ_{yy} are nonzero) or anti-plane (v_z, σ_{xz}, σ_{yz} are nonzero) problems, where all variables are functions of x and y.

Initial conditions must be provided for the velocities and stresses. The code assumes that the velocities are initially zero, and provides numerous ways to set the initial stress depending on the complexity of the problem. More details are provided when describing the code specifics.

For Drucker-Prager Viscoplasticity, plastic flow occurs when stresses exceed the yield function $F(\sigma_{ij})$:

$$F(\sigma_{ij}) = \bar{\tau} - c + \mu\sigma_{kk}/3,$$

where $\bar{\tau} = \sqrt{s_{ij}s_{ij}/2}$ is the second invariant of the deviatoric stress tensor $s_{ij} = \sigma_{ij} - (\sigma_{kk}/3)\delta_{ij}$, c is related to the cohesion, and μ is related to the internal coefficient of friction. For viscoplasticity, flow is allowed to exceed the yield criterion according to

$$F(\sigma_{ij}) = \Lambda\eta,$$

where $\Lambda = \sqrt{2\dot{e}^{pl}_{ij}\dot{e}^{pl}_{ij}}$ is the equivalent plastic strain rate from the deviatoric plastic strain rate $\dot{e}^{pl}_{ij} = \dot{\epsilon}^{pl}_{ij} - (\dot{\epsilon}^{pl}_{kk}/3)\delta_{ij}$ and η is a viscoplastic "viscosity" defining the time scale over which stresses can exceed the yield stress. If stresses are accumulated at a rate faster than the relaxation time of the viscoplastic material, the material behaves elastically, and the stress then decays towards the yield surface if no further stresses are applied.

The components of plastic flow are determined by

$$\dot{\epsilon}^{pl}_{ij} = \Lambda P_{ij}\left(\sigma_{ij}\right),$$

with $P_{ij}(\sigma_{ij}) = s_{ij}/(2\bar{\sigma}) + (\beta/3)\delta_{ij}$, where the β parameter determines the ratio of volumetric to plastic strain. Thus, viscoplastic materials are determined by ρ, λ, G, c, μ, β, and η, while elastic materials require specification of ρ, λ, and G. Rate independent plasticity arises in the limit that $\eta \to 0$, though the equations become increasingly stiff in this limit and plastic strain can exhibit localization that is not resolved by the spatial grid in many problems.

The code can handle variable material properties in two different ways. The first type is "block-like" structures, where the material properties are piecewise constant in an arbitrary number of blocks. Alternatively, one can specify the elastic properties in a point-by-point fashion (the plasticity material properties must be block-like even in the case of continuously varying elastic properties in the present version of the code).

Regardless of the method used, the domain is composed of a regular grid of blocks with conforming edges, though the block boundaries can have complex shapes, so this does not limit the ability of the code to handle complex earth structures. Blocks are coupled together through interfaces, which can either be locked or frictional. Locked interfaces require continuity of velocity and forces across block edges, while frictional interfaces allow for relative slip across the interface according to several possible friction laws:

- **Frictionless interfaces** do not support shear stresses, so any stress applied to such interfaces result in shear slip.
- **Kinematic forcing** allows for a forced rupture following a prescribed set of rupture times.
- **Slip weakening interfaces** follow a slip-dependent friction law, where the friction coefficient has static and dynamic values and transitions between the two according to a function that decreases linearly with slip. Slip weakening laws also allow for cohesion. Slip weakening can also be combined with kinematic forcing to nucleate a rupture.
- **Shear Transformation Zone (STZ) Theory interfaces** have a frictional strength that depends on the slip rate and a dynamic state variable representing the configurational disorder in the fault gouge.

Specific details on each of the friction models is provided below. In addition to these base friction laws, a future version of the code will allow you to specify arbitrary slip- or rate- and state-dependent friction laws through the Python module, which automatically generates the required source code for new friction laws.

External boundaries (i.e. boundaries not between two blocks) can have absorbing, free surface (traction-free), or rigid (velocity-free) boundary conditions.

1.2 Frictional Descriptions

1.2.1 General Formulation

Friction laws are used to set the boundary conditions on the edges connecting block interfaces. At each edge, the velocities and stresses for each block are rotated into normal and tangential components. The material governing equations above provide one set of relationships between the velocities and tractions; an additional condition is required to fully specify the boundary values.

For the normal components, the code requires continuity of velocities and tractions across the interface. In the event that tensile normal stresses occur, the code sets the normal traction to zero but does not allow for the interface to open.

For the shear components, the tractions are required to be continuous, but the velocities can be discontinuous across the fault. The velocity discontinuity, called the slip velocity V with vector components V_1 and V_1, is the primary quantity of interest, as it describes the rate at which fault slip occurs. The wave equation provides one relationship between the shear traction and the slip velocity, and the other is given by another equation (along with the constraint that the traction and slip are parallel to one another). This relationship may simply specify the traction (so that the slip velocity can be solved for directly), or require a nonlinear solver if the relationship is nonlinear and cannot be solved in closed form.

The slip velocity is integrated in time, resulting in slip components U_1 and U_2, along with a scalar slip U that is computed as a line integral. The slip is used by some friction laws to control the evolution of strength as a function of time, but is computed for all frictional descriptions.

1.2.2 Frictionless Interface

The frictionless interface is the simplest friction description – the interface cannot support a shear traction, so the slip velocity is set to be whatever value is needed to ensure that no stress accumulates. This applies to both components of traction for 3D problems. This model requires no additional parameter specifications beyond the material properties.

1.2.3 Kinematic Forcing

Rupture propagation can be prescribed using kinematic forcing. Kinematic forcing requires specification of 4 parameters: a static friction coefficient μ_s, a dynamic friction coefficient μ_d, a time scale t_c which sets the time scale over which friction linearly weakens from static to dynamic, and a rupture time t_{rup} that determines when the frictional weakening initiates at a given point. The friction coefficient can be determined directly from the time, and the friction coefficient combined with the normal traction determines the shear traction on the fault. If the value of the shear traction set by the wave equation is less than the value set by the friction law, the fault is locked, the slip rate is zero, and the shear traction takes the value from the wave equation.

1.2.4 Linear Slip-Weakening

In order for rupture propagation to be truly spontaneous, a friction law that does not prescribe rupture time is required. The most common form used in dynamic rupture modeling is the linear slip-weakening law. As with kinematic forcing, a static and dynamic friction coefficient must be specified. However, instead of a rupture time and a weakening time, the weakening process is characterized by a slip weakening distance d_c. Friction weakens linearly with slip from the static to dynamic value:

$$\mu(U) = \begin{cases} (\mu_s - \mu_d)\left(1 - \frac{U}{d_c}\right) + \mu_d & (U < d_c) \\ \mu_d & (U \geq d_c). \end{cases}$$

Once the friction coefficient is known, the shear traction is set in a similar fashion to the Kinematic Forcing law described above. The code also allows for frictional cohesion c_0, in which case the shear traction τ is:

$$\tau = c_0 + \mu \max(0, -\sigma_n)$$

where σ_n is the normal traction (negative in compression).

For 3D problems with vector slip, each vector component of the velocity/traction is solved separately, but the total slip U is used to determine the weakening behavior.

Additionally, the code allows for a combination kinematic/slip-weakening law, where the code uses the minimum friction coefficient that is calculated for the kinematic and slip-weakening laws. This is used in cases where the rupture is initiated with a kinematic procedure, but then is allowed to propagate spontaneously.

1.2.5 Shear Transformation Zone Theory

Shear Transformation Zone (STZ) Theory is a rate- and state-dependent constitutive law, which ties the fault strength to the dynamic evolution of a state variable representing the effective disorder temperature χ. Frictional strength μ is determined by the slip rate and the effective temperature:

$$V = V_0 \exp\left(-f_0 + \frac{\mu}{a} - \frac{1}{\chi}\right)\left(1 - \frac{\mu_y}{\mu}\right).$$

Note that this cannot be solved in closed form for the friction coefficient, so the code solves this equation simultaneously with the elastic wave equation for μ and V. Additionally, the effective temperature evolves in time according to

$$\frac{d\chi}{dt} = \frac{V\tau}{c_0}\left(1 - \frac{\chi}{\hat{\chi}(V)}\right) - R\exp\left(\frac{\beta}{\chi}\right),$$

where the rate-dependent steady state effective temperature is $\hat{\chi}(V) = \chi_w/\log(V_1/V)$. The STZ model introduces several additional parameters: a reference slip rate V_0, an activation barrier for slip f_0, the frictional direct effect a, the friction coefficient at jamming μ_y, the effective temperature specific heat c_0, the normalized effective temperature activation barrier χ_w, the reference slip rate for STZ activation V_1, the STZ relaxation rate R, and the normalized effective relaxation barrier β.

More details on the STZ model and the parameters can be found in the papers listed below.

1.3 Numerical Details

The code solves the governing equations numerically using finite differences. The outer boundaries of each block is described by a series of 6 surfaces (4 curves in 2D), and each block is transformed from physical space to the unit cube (unit square in 2D). The governing equations are transformed as well, and the code solves the resulting problem on a structured grid using high order finite differences. The grid is generated using standard transfinite interpolation, and the required metric derivatives for solving the governing equations and applying boundary conditions are automatically calculated using finite differences. Grids between neighboring blocks must be conforming, though no other continuity condition is required across block interfaces. The grid must satisfy certain smoothness constraints (these are checked during the initialization steps in the code), though non-uniform grid spacing along interfaces is allowed, provided that the resulting grid meets the smoothness requirements. Boundary conditions at external boundaries and interfaces are applied in locally rotated normal/tangential coordinate systems using characteristic variables.

The specific finite difference operators used exhibit a summation-by-parts property that mimics the properties of integration by parts. This allows for estimates of the energy dissipation rate of the numerical scheme. Boundary conditions are imposed weakly using the Simultaneous Approximation Term approach, and this combined with the summation by parts difference operators allows for a provably stable numerical scheme that matches the energy dissipation rate of the continuous problem.

The code allows for central finite difference operators that are globally second, third, or fourth order accurate. Time integration is performed with a low memory Runge-Kutta method, with either first, second, third, or fourth order accuracy in time. Artificial dissipation can also be used for the finite difference operators, which will reduce the numerical artifact oscillations that can occur with large grid spacings.

The finite difference method is only applied to the elastic part of the problem. The plasticity equations are handled through an operator splitting procedure, where the elastic problem is solved first and then used as initial conditions for the plasticity problem. This is done using an implicit backward Euler method.

For more details on the numerical methods used, please consult the papers listed below.

1.4 References

CHAPTER

TWO

INSTALLATION

`fdfault` requires a C++ compiler and an MPI library, though you can run the code on a single processor if you want. The code has mostly been tested with the GNU compiler and Open MPI. Parallelization is achieved through domain decomposition, with interprocessor communication occuring after each Runge-Kutta stage to populate ghost cells with the appropriate values. Python with Numpy is required if you want to use the Python module for setting up problems and generating the input files, with support for either Python 2 or 3. The code also includes Python (requires Numpy) and MATLAB scripts for loading simulation data.

2.1 Building the Main Executable

If installing from a downloaded zip archive, enter

```
unzip fdfault-1.0.zip
```

Depending on which version you downloaded, the filename for the zipped archive might be different. Or clone the git repository using

```
git clone https://github.com/egdaub/fdfault.git
```

For most users, building the source code should simply require

```
cd fdfault/src
make
```

assuming you have Make and an appropriate C++ compiler with an MPI Library. You may need to change some of the compiler flags – I have mostly tested the code using the GNU Compilers and OpenMPI on both Linux and Mac OS X. This will create the fdfault executable in the main `fdfault` directory.

2.2 Installing the Python Module

You will also need to configure the python module. There are several ways to do this:

1. Install the Python module system-wide. To make the Python tools available system-wide, change to the python directory and run setup.py (you must have setuptools installed):

   ```
   cd fdfault/python
   python setup.py install
   ```

   You may also use Python 3 without any modifications. Depending on your setup, you might need administrative privileges to do the installation step. If you obtained the code by cloning the Git repository, installation in this manner will not update automatically if any of the source code files are updated. If you want to keep up to date without having to reinstall, install a development version:

   ```
   cd fdfault/python
   python setup.py develop
   ```

   This will simply place a link to the fdfault python directory in the system Python libraries, so any updates will automatically be available.

2. If you would like the Python module available in any directory without installing for other users, you can simply modify your PYTHONPATH environment variable to include the full path to `fdfault/python`. This will only effect the current user.

3. Some users prefer to only have the Python tools available in certain directories. The tools for setting up problems are most often used in the problems directory, and the analysis tools are most often used in the data directory.

   To make these tools available in these directories only, make a symbolic link to the python/fdfault directory in the problems directory:

   ```
   cd fdfault/problems
   ln -s ../python/fdfault/ fdfault
   ```

   This will allow you to simply type `import fdfault` in python from within the problems directory. Similarly, to make the analysis features available in the data directory:

   ```
   cd fdfault/data
   ln -s ../python/fdfault/analysis fdfault
   ```

   This allows you to type "import fdfault" from within the data directory and have the analysis tools at your disposal.

2.3 Building the Documentation

Finally you will need to build the User's Guide (requires Sphinx, with MathJax required for the HTML verions and a LaTeX distribution for the PDF version).

```
cd fdfault/docs/
make html && make latexpdf
```

This should build the notes in the `fdfault/docs/_build/html` or `fdfault/docs/_build/latex` directories. If you wish to build only the html or pdf version, use the appropriate command. If you do not have Sphinx or LaTeX on your machine, both versions of the documentation are available on the web:

http://www.ceri.memphis.edu/people/egdaub/fdfault/_build/html/index.html (html)

http://www.ceri.memphis.edu/people/egdaub/fdfault/_build/latex/fdfault_docs.pdf (pdf)

CHAPTER

THREE

SPECIFYING SIMULATION PARAMETERS

Parameters for simulations are set with input files. These are text files that are formatted to be understood by the C++ code. They mostly consist of section headers followed by raw numbers, strings, or file names. They are manageable for small, simple problems, but to really use the full power of the code it is suggested to take advantage of the Python module. Either way, parameters are set through an input file `problemname.in`, though the problem name is actually set in the input file itself so the input file name and problem name need not be the same.

Once the input file is written to disk, you can launch a simulation. From the main `fdfault` directory, simply enter:

> > mpirun -n 4 fdfault problems/problemname.in

This should run the problem on 4 processors, printing out information to the console as it progresses. If you wish to use a different number of processors, modify the 4 (some versions of MPI may require you to use the option flag `-np 4` to set the number of processors, and some versions of MPI may require that you use `mpiexec` to run a simulation). If you are running the code on a cluster, you should follow your normal procedure for submitting jobs.

The code assumes you will be running everything in the main code directory, and by default uses relative paths to that main directory to write the simulation files to disk. You are welcome to run the code from another directory, but you should either have a `data` directory already created or use the full path to the location where you wish to write data.

3.1 Text Input Files

Text input files contain multiple sections. With one exception, the order of the sections is not important – the file is read multiple times by various parts of the code, and each time the file is read the code starts from the beginning and scans through to find the appropriate section. Each section is designated by a text string `[fdfault.<name>]`, where `<name>` refers to which part of the code will be reading this section. If the code expects to find a certain section and it reaches the end of the text file without finding it, the code will abort. At minimum, the following sections are required to fully specify a problem:

```
[fdfault.problem]
[fdfault.domain]
[fdfault.fields]
[fdfault.blockXYZ]
[fdfault.outputlist]
[fdfault.frontlist]
```

In addition to these sections, optional sections describe additional blocks and interfaces, as well as other parameter settings. These include the following sections:

```
[fdfault.cartesian]
[fdfault.operator]
[fdfault.interfaceN]
[fdfault.friction]
[fdfault.slipweak]
[fdfault.stz]
```

If the problem has more than one block or more than one interface, the sections are designated with the numeric value in place of `XYZ` or `N` included in the section header.

3.1.1 Problem Input

A problem is specified under `[fdfault.problem]`. The entries under this section are as follows:

```
Problem Name
Data where simulation output will be saved
Number of time steps
Time step size
Total time
Courant ratio
Frequency at which status information is written to the terminal
Runge-Kutta Integration Order (integer 1-4)
```

Most of these are straightforward. The main tricky part in this section is that you typically will only specify two of the four options for determining the time step. You are free to specify any two of these, with the exception of the time step and Courant ratio (the ratio between the grid spacing and the distance a wave travels in one time step). If you specify both the time step and Courant ratio, the code defaults to the given time step. If you specify more than two parameters, the code defaults to the total time and either the time step or the Courant ratio.

3.1.2 Domain Input

Details of the spatial domain are determined by the `[fdfault.domain]` header. The following arguments are required:

```
Number of dimensions (2 or 3)
Rupture mode (only meaningful for 2D problems, but necessary for 3D␣
↪problems)
Number of grid points (3 integers, if 2D problem the third will be␣
↪reset to 1)
Number of blocks (3 integers, if 2D problem the third will be reset to␣
↪1)
Number of grid points for each block in x-direction
Number of grid points for each block in y-direction
Number of grid points for each block in z-direction
Number of interfaces
Type of each interface (list of strings)
Finite difference order (integer 2-4)
Material type (elastic or plastic)
```

For this section, the trickiest part is understanding how the blocks and sizes are set up. First, the number of grid points is specified (which must have a length of 3), and then the number of blocks in each dimension is specified (also must of of length 3). If the problem is 2D, then the third entry in each list will be reset to 1 if it is not already 1. Depending on these entries, the code expects the integers that follow to conform to a specific order. First comes the length of each block along the x-direction. The code expects the number of entries to match the number of blocks, and the sum of all entries must equal the total number of grid points along the x-direction. Similarly, the y and z directions are specified in the subsequent entries. While it is recommended that the entries for each direction are on separate lines, the spacing between entries, as well as the line spacing, are ignored when reading the input file.

After the block dimensions are set, the code reads the number of interfaces, followed by the interface types (it expects a number of strings corresponding to the number of interfaces). Again, line breaks are ignored. The type of each interface must be one of the following: `locked`, `frictionless`, `slipweak`, or `stz`.

The final two entries are fairly self explanatory, and determine the finite difference order (integer 2-4) and the material response (elastic or plastic).

3.1.3 Cartesian Input

The code automatically handles domain decomposition into a Cartesian grid based on the dimensionality of the problem and the number of processors specified when running the executable. However, you may also specify the number of processes manually by including `[fdfault.cartesian]` in the input file. This section must contain a list of three integers specifying the desired number of processes in each of the three spatial dimensions (if a 2D problem is run, the number of processes in the z direction is automatically set to one). It is up to the user to ensure that the numbers set here match the total number of processes set when launching the executable.

3.1.4 Fields Input

The initial stress fields are set with the `[fdfault.fields]` header. This section has three entries:

```
Uniform initial stress tensor
Filename for spatially heterogeneous initial stress tensor
Filename for spatially heterogeneous elastic properties
```

The uniform initial stress tensor is a list of 6 numbers, and the order is σ_{xx}, σ_{xy}, σ_{xz}, σ_{yy}, σ_{yz}, σ_{zz}). Components not involved in a 2D problem are in some cases used in the problem, particularly for anti-plane (mode 3) problems, where the in-plane normal stress components determine the compressive normal stresses acting on the fault. Line breaks are ignored.

If a heterogeneous stress tensor will be used, it is specified with a filename here. If no heterogeneous file is to be read, this entry should be `none`. The file should contain a sequence of double precision binary floating point numbers (endianness should match the processor where the code will be run). Components are entered one at a time, with the number of entries matching the grid size using row major order (C order). For 2D mode 3 problems, the order is σ_{xz}, σ_{yz}. For 2D mode 2 problems, the order is σ_{xx}, σ_{xy}, σ_{yy} (and for plasticity problems, σ_{zz}). For 3D problems, the order is the same as for the uniform stress tensor. Entering heterogeneous stresses is greatly simplified if you use the Python module.

Similarly, if a heterogeneous elastic properties will be used, it is specified with a filename here. If no heterogeneous file is to be read, this entry should be `none`. The format is the same as the stress tensor, but with three entries: density, first Lamé parameter, and shear modulus. Creation of these files is simplified with the Python module.

Note: for large 3D problems, the arrays for a heterogeneous stress field or elastic property may be too large to be handles by the Python module (Numpy seems to be limited to arrays that are 2 or 4 GB, depending on the version of Python that you use). In that case, you may need to generate these files manually.

3.1.5 Operator Input

Optionally, the code uses artificial dissipation to damp out spurious oscillations arising from the finite difference method. To use artificial dissipation, include a `[fdfault.operator]` section with a single floating point number to designate the the artificial dissipation coefficient. Correct selection of the dissipation coefficient is up to the user, and too large a value can result in numerical instabilities.

3.1.6 Block Input

Each block has its own section in the input file, designated by `[fdfault.blockXYZ]` for the block with indices (X, Y, Z) (the code assumes zero indexing). For each block, the input file entries set the material properties, boundary conditions, and geometry. The input file must contain a header for each block in the simulation, and other block headers that do not match a block in the simulation are ignored. Order is not important. Specific entries are as follows:

```
Material properties
Block lower left coordinate
Block side lengths
Block boundary conditions
Block boundary filenames
```

The number of entries expected in each item depends on the type of problem being solved, and are explained below.

The material properties are the density ρ, Lamé constant λ, and shear modulus G, and for plasticity problems the parameters defined in the yield function and flow rule. Order for the plasticity parameters is internal friction μ, cohesion c, dilatancy β, and viscosity η.

The lower left coordinate of the block determines its location in space, and requires 2 numbers for 2D problems and 3 numbers for 3D problems. Similarly, the block side lengths require the same number of entries for 2D and 3D problems. These coordinate values are used to create any boundary surfaces that are not set through a file by creating rectangular surfaces (in 3D) and straight lines (in 2D) for the appropriate block sides. If all sides are given as a file, these entries are ignored in creating the grid, though they are still used in adding surface tractions to frictional faults and modifying friction parameters.

The block boundary conditions is a list of 4 boundary conditions for 2D problems, and 6 boundary conditions for 3D problems. Order is left, right, front, back, top, bottom (where top and bottom are only for 3D problems). Each boundary condition must be one of the following strings: `absorbing` (no waves enter the domain), `free` (traction free surface), `rigid` (zero velocity), or `none` (do not apply a boundary condition, used if block is coupled to another through an interface).

Boundaries that are defined via a filename derive their data from files that contain binary data, rather than assuming a rectangular block edge. This method can be used to create non-planar block surfaces. The number of entries and order is the same as for the boundary conditions. Each file must contain double precision floating point binary data, with all x coordinates in row major (C) order, followed by all y coordinates, and if a 3D problem, all z coordinates. Endianness is set by the computer where the simulation will be run. When setting nonplanar boundaries, the surfaces must conform at their edges, and the code checks this during initialization. While you can easily create your own files for defining nonplanar boundaries, this is made much simpler with the Python module.

3.1.7 Interface Input

All interfaces are specified by a header with the form `[fdfault.interfaceN]`, where `N` determines the interface number (zero indexed). For problems with n interfaces, the input file must contain an interface header for all n interfaces, or you will get an error. Interface headers are followed by the following arguments:

```
Approximate normal direction
Minus block indices
Plus block indices
```

First, the list defines the approximate normal direction of the interface in the simulation based on the simulation geometry. For rectangular blocks, this is the true normal direction of the block, while if the block has boundaries specified through a file the normal direction may not be precisely in this direction, or the normal direction may not be uniform across the entire block. The direction is set by a string `x`, `y`, or `z`.

Following the direction specification, you must set the indices of the block on the "minus" side of the interface (a list of 3 integers). This can be any block in the simulation, but must be the block with the lowest set of indices that are joined by this interface. Order is not important for setting up the interfaces – interface 0 can join any pair of neighboring blocks in the simulation – and the lists can appear in the input file in any order.

Next, the block on the "plus" side of the interface is given by its indices. Because the blocks in the simulation must form a regular grid, the "plus" block must differ in only one index from the "minus" block, and the index that is different must be the same as the direction specified above (this is checked by the code when initializing). For instance, if the minus block is $(0, 0, 0)$, and the direction is `x`, then the plus block must have the index $(1, 0, 0)$. Line breaks are ignored when reading in the indices.

3.1.8 Friction Input

The information above is all that is required for locked interfaces. For frictional interfaces, additional information must be provided. All frictional interfaces must include a `[fdfault.friction]` header somewhere *after* the interface header. Note that the friction header does not include an interface number – the code scans through the input file to find the interface header for the interface in question, and then continues scanning until it finds the next friction header. This trick lets you easily set multiple frictional interfaces to have the same specifications.

Friction headers contain the following information:

```
Number of load perturbations
List of load perturbations
Load perturbation filename
```

First is an integer that determines the number of surface traction perturbations that will be applied to the interface during the simulation. Next is a list of perturbations, followed by a file containing additional perturbations to the load.

Load Perturbations

Load perturbations have the following format:

```
type t0 x0 dx y0 dy s1 s2 s3
```

`type` is a string that determines the spatial characteristics of the perturbation. Options are `constant` (spatially uniform), `boxcar` (spatially uniform inside a rectangular area, zero outside), `ellipse` (spatially uniform inside an elliptical area, zero outside), and `linear` (linear function in each direction).

`t0` determines the time scale over which the perturbation is added, with a linear ramp from zero over the given time scale. If the perturbation is to be added from the start of the simulation, give 0.

`x0 dx` are constants determining the shape of the perturbation along first coordinate direction of the interface (x for `y` and `z` interfaces, y for `x` interfaces). For constant perturbations, these parameters are ignored. For boxcar and ellipse perturbations, `x0` is the center of the perturbation and `dx` is the half width. If you want the width to span the entire interface width, enter 0 for `dx`. For linear perturbations, `x0` is the intercept and `dx` is the slope. If you want the linear gradient to only extend in one spatial direction, enter 0 for `dx` in the direction where you want the perturbation to be constant. Similarly, `y0 dy` set the same values for the other coordinate direction (y for `z` interfaces and z for `x` and `y` interfaces). For 2D problems, the second set of indices is ignored, but still must be present in the input file.

In the simulations, the values of `x0 dx y0 dy` are only interpreted literally for rectangular blocks. For non-rectangular blocks, these values are interpreted assuming the interface follows a rectangular block on the minus side, using the values given under the block header. This means that the values may not be interpreted exactly as you expect!

Finally, a trio of numbers set the vector surface traction applied to the interface. The first component is the normal traction, and the next two numbers are the two shear tractions. For 2D problems, the first shear component is the in-plane shear traction (only valid for mode 2 problems), and the second is the out of plane shear traction (always in the z-direction and only valid for mode 3 problems). The code sets the unused shear traction component to zero. For 3D problems, the exact meaning of the shear traction components are determined by the surface normal direction, described as follows.

The different interface components do not truly correspond to the corresponding coordinate directions. The code handles complex boundary conditions by rotating the fields into a coordinate system defined by three mutually orthogonal unit vectors. The normal direction is defined to always point into the "positive" block and is uniquely defined by the boundary geometry. The two tangential components are defined as follows for each different type of interface:

- Depending on the orientation of the interface in the computational space, a different convention is used to set the first tangent vector. For `'x'` or `'y'` oriented interfaces, the z component of the first tangent vector is set to zero. This is done to ensure that for 2D problems, the second tangent vector points in the z-direction. For `'z'` oriented interfaces, the y component of the first tangent vector is set to zero.
- With one component of the first tangent vector defined, the other two components can be uniquely determined to make the tangent vector orthogonal up to a sign. The sign is chosen such that the tangent vector points in the direction where the grid points are increasing.
- The second tangent vector is defined by taking the right-handed cross product of the normal and first tangent vectors, except for `'y'` interfaces, where the left-handed cross product is used. This is done to ensure that for 2D problems, the vertical component always points in the $+z$-direction.

File Perturbations

After all of the surface traction perturbations, the code takes a filename of a file that adds additional tractions to the surface. The file contains a series of double precision floating point binary numbers of length $3 \times n1 \times n2$, where $n1$ and $n2$ are the number of grid points along the interface. The first block of $n1 \times n2$ is for the normal traction (in row major order), then the in-plane shear traction component, and finally the out of plane shear traction component, with the same convention described above for setting the tangential directions. Endianness is assumed to match the computer where the simulation is being run.

Additional Friction Parameter Specifications

There are several specific types of frictional interfaces, two of which require additional parameters be specified:

1. **Frictionless** interfaces do not support shear tractions. No additional parameters are required when specifying frictionless interfaces.
2. **Slip-Weakening** interfaces require additional parameter specifications
3. **STZ** interfaces also require additional parameter specifications

For more information on how to set slip-weakening and STZ parameter values, consult the following pages.

3.1.9 Slip Weakening Input

The basic format for slip-weakening interaces is analogous to those for setting the surface tractions:

```
Number of parameter perturbations
List of parameter perturbations
Parameter perturbation filename
```

Each item in the list requires the same basic six parameters to describe the shape as for the load perturbations (type t0 x0 dx y0 dy), interpreted in the same way as described above. For slip weakening laws, there are six additional parameters that must be specified:

```
dc mus mud c0 tc trup
```

`dc` is the slip weakening distance, `mus` is the static friction coefficient, `mud` is the dynamic friction coefficient, `c0` is the frictional cohesion, `tc` is the characteristic weakening time, and `trup` is the forced rupture time. The first four parameters are fairly standard, while the final two parameters are used to impose kinematic forcing on the rupture front: `tc` is the time scale over which the friction weakens from static to dynamic, and `trup` is the time at which this process initiates. If `trup` is 0, then no kinematic forcing is used.

As with load perturbations, heterogeneous perturbations using an input file are also allowed. The format is the same as with load perturbations (C order double precision floats, with parameters given in the order listed above), and the ordering of the arrays is the same as for the perturbations listed above. The Python module can help with generating the appropriate files. If no file is used, the code must include `none` in place of the filename.

3.1.10 STZ Friction Input

The basic format for STZ interaces is analogous to those for setting the surface tractions, with the main difference being that additional parameters are needed to set the initial value of the state variable:

```
Initial effective temperature
Filename for heterogeneous initial effective temperature
Number of parameter perturbations
List of parameter perturbations
Parameter perturbation filename
```

First is the (uniform) initial value of the effective temperature, which is simply a floating point number. Next is a file holding double precision floating point numbers in C order for all points on the interface that set the initial value for the effective temperature (the sum of the grid-based values and the overall constant determines the initial value). Endianness should be the same as the computer where the simulations will be run. If no file is used, `none` should be entered in its place.

Next, the friction parameters are set, which uses both perturbations analogous to the load perturbations. Each item in the perturbation list requires the same basic six parameters to describe the shape as for the load perturbations (type t0 x0 dx y0 dy), interpreted in the same way as described in the load section. For STZ friction laws, there are nine additional parameters that must be specified:

```
V0 f0 a muy c0 R beta chiw V1
```

See the introduction for more details on the meaning of these parameters.

Fully heterogeneous parameter values can be set using a file holding grid-based data (again double precision in C order, with endinanness corresponding to the machine where the simulation will be run). The full arrays for all 9 parmeters must be given in the same order as the parameters in the perturbation. If no file is used, `none` must be used as a placeholder.

3.1.11 Saving Simulation Data (Output)

The code allows for flexible specification of output. Any field values can be saved, and the code allows for flexibility in specifying slices of the output fields in time and space. Writing files to disk is done in parallel using MPI, with binary output being the only supported format, though several scripts are included for converting the binary output to other formats (see the analysis section for more details).

Internally, the code handles output by defining a list of "output units" that contain information on what information is written to disk and how often to write that information to disk. The output units also automatically output time and spatial grid information for the particular data that is written to file. The code supports an arbitary number of output units, with the only limitations being memory and disk usage considerations.

Specifying an output unit requires that you provide a name, a field, and then time and coordinate information. For coordinate (3 space and 1 time), you must provide a minimum index, a maximum index, and a stride. The minimum value is the first index that is saved, the maximum value is the last index that is saved, and the stride tells how frequently in space or time to save this particular field. Note if the stride is such that the maximum index is skipped over, the last index falling within the desired range becomes the new maximum (this information is accounted for when writing output unit metadata to disk). This applied to all three spatial dimensions as well as time, so a total of 12 index values must be supplied. The exact details of the index information depends on the field that is chosen, as described below.

Field Types

The code supports output of two types of fields: grid-based fields, and interface-based fields. Grid-based fields are defined over arbitrary ranges of grid points in the entire simulaiton domain, and can be single points, 1D, 2D, or 3D slices of the simulation grid. Interface-based fields only exist along interfaces between blocks, and thus are single points, 1D, or 2D slices of the co-located grid points at the interface.

Grid-Based Fields

Grid-based fields are fields that are defined over the entire simulation domain. This includes the particle velocities and stress tensor, as well as fields associated with plastic strain. To define a grid-based output unit, a name, field, and 4 sets of index values (minimum, maximum, and stride) for time and 3 spatial dimensions are required. The name is simply a text string used to define the file names, so you cannot use the same name twice. The field can be any of the following strings:

Field	Corresponding String
Particle velocity, x-component	`vx`
Particle velocity, y-component	`vy`
Particle velocity, z-component	`vz`
Stress tensor, xx-component	`sxx`
Stress tensor, xy-component	`sxy`
Stress tensor, xz-component	`sxz`
Stress tensor, yy-component	`syy`
Stress tensor, yz-component	`syz`
Stress tensor, zz-component	`szz`
Scalar measure of Plastic Strain	`gammap`
Scalar measure of Plastic Strain Rate	`lambda`

The field name is case-sensitive, as we will see shortly when describing interface-based fields. Depending on the type of simulation that is being done, only certain fields are nonzero, so you cannot specify a field that is not accounted for by the simulation: mode 2 2D simulations can only use `vx`, `vy`, `sxx`, `sxy`, and `syy` (and `szz` for problems allowing plastic deformation); similarly mode 3 2D simulations can only use `vz`, `sxz`, and `syz`. Finally, elastic simulations cannot use either of the plasticity fields. If you select a field that is not defined for a given simulation, you will get an error and the code will abort.

Indices need to be given for all three simulation dimensions even if the simulation is in 2D. The code requires that the minimum index fall within the bounds for the simulation (i.e. it must be at least zero and smaller than the total number of grid points in the particular direction), and that the maximum index be greater than or equal to the minimum index. This means that if you make a mistake and make the maximum index larger than the number of grid points, the code will simply use the total number of grid points as the maximum value.

Interface-Based Fields

Interface-based fields are fields that only exist on a particular interface and are not defined over the entire simulation domain. They include the slip, slip velocity, interface tractions, and state variables, and most of these fields have a signed component version as well as a positive scalar value version. As with grid-based fields, a name, field, and 4 sets of index triplets are required to specify an output unit. The allowable fields for an interface-based field are as follows:

Field	Corresponding String
Slip, x-component	`Ux`
Slip, y-component	`Uy`
Slip, z-component	`Uz`
Slip, scalar magnitude (line integral of scalar slip velocity)	`U`
Slip velocity, x-component	`Vx`
Slip velocity, y-component	`Vy`
Slip velocity, z-component	`Vz`
Slip velocity, scalar magnitude	`V`
Normal traction	`Sn`
Shear traction, x-component	`Sx`
Shear traction, y-component	`Sy`
Shear traction, z-component	`Sz`
Shear traction, scalar magnitude	`S`
State variable	`state`

Note that these are distinguished from grid-based fields in that most fields start with a captial letter (field names are case sensitive).

The different interface components do not truly correspond to the corresponding coordinate directions. The code handles complex boundary conditions by rotating the fields into a coordinate system defined by three mutually orthogonal unit vectors. The normal direction is defined to always point into the "positive" block and is uniquely defined by the boundary geometry. The two tangential components are defined as follows for each different type of interface:

- Depending on the orientation of the interface in the computational space, a different convention is used to set the first tangent vector. For `'x'` or `'y'` oriented interfaces, the z component of the first tangent vector is set to zero. This is done to ensure that for 2D problems, the second tangent vector points in the z-direction. For `'z'` oriented interfaces, the y component of the first tangent vector is set to zero.
- With one component of the first tangent vector defined, the other two components can be uniquely determined to make the tangent vector orthogonal up to a sign. The sign is chosen such that the tangent vector points in the direction where the grid points are increasing.
- The second tangent vector is defined by taking the right-handed cross product of the normal and first tangent vectors, except for `'y'` interfaces, where the left-handed cross product is used. This is done to ensure that for 2D problems, the vertical component always points in the $+z$-direction.

The consequence of this is that the letter used to designate the desired component is only valid for rectangular geometries. For non-rectangular geometries, the components will be rotated into the coordinate system described above. For interfaces in the "x" direction (i.e. connecting blocks whose indices only differ in the x-direction), the y component of output units will be along the first tangent vector, and the z component will be along the second tangent vector. Similarly, for "y" interfaces the x component is set by the first tangent vector and the z component is determined by the second tangent vector, and for "z" interfaces the first tangent vector is in the x-direction and the second tangent vector corresponds to the y-direction. If you desire the components in a different coordinate system, you can convert them from the output data. Note that this also means that you can only specify certain components for interface output, depending on the direction of the interface.

Additionally, the state variable is only valid for friction laws for which a state variable is defined.

Because interfaces are not defined over the entire domain, you cannot specify arbitrary values for the grid indices. When the code sees that you have chosen a field appropriate for interface output, it takes the three minimum spatial indices used to define the output unit and searches over all interfaces until it finds one where the given indices are part of that interface. If none is found, an error is raised and the code aborts. Index values on either the "minus" or "plus" side are equally valid. Note also that since interfaces are 1D or 2D slices in the domain, that at least one set of index values must have the same minimum and maximum indices, and that this index must lie on some interface in the simulation.

One final note on interface output: because of how the code handles output in parallel, each output unit can only handle data from a single interface. Whichever interface is found for the three minimum spatial indices is used for output, even if the maximum spatial index extends to another interface. If output over multiple interfaces is desired, you must save each interface separately.

Output List

Each output unit is part of a longer "output list" that is set in the `[fdfault.outputlist]` block of the input file. The `outputlist` block consists of a series of individual output items, each of which is specified as follows:

```
name
field
tmin tmax tstride
xmin xmax xstride
ymin ymax ystride
zmin zmax zstride
```

Line breaks are optional within a single output unit, but required between consecutive output units. The code reads output units until it encounters a blank line, so you must terminate the list with a blank line.

3.1.12 Rupture Front Times Input

In addition to the extensive set of options for saving field data from the simulation, the code can also save rupture times (useful for determining rupture front contours). The rupture time is defined as the earliest time when a particular interface field first exceeds a threshold. If rupture front output is selected, the code will save a file for each interface that indicates the earliest time at which the chosen field exceeds the threshold value. If the particular point never ruptures, a value of `-1.`

is saved. Additionally, the spatial grid information for each interface is saved. Front output only applies to frictional interfaces, and the code will automatically set up output for any frictional interface in the simulation while ignoring others. For more details on interpreting the results of rupture front output, see the analysis section.

The front output is set using the `[fdfault.frontlist]` section of the input file. This section of the input file has the following format:

```
Boolean indicating if front output is desired
Field used to determine rupture time (required only if front output is␣
↪turned on)
Field value used to determine rupture time (required only if front␣
↪output is turned on)
```

The `frontlist` section requires one argument indicating whether or not rupture time output is desired (`0` means no output, `1` indicates that rupture times for all frictional interfaces should be saved). If output is turned on, two additional arguments are needed: first, a field must be indicated to be used to determine the rupture time, and a value for that field.

The field can be one of two strings: `U` if a slip threshold will be used to determine the rupture time, or `V` if a slip rate threshold is desired for determining the rupture time. The field value must be a numeric value, and the rupture time will be the earliest time at which the chosen field exceeds the threshold value.

Rupture front output is optional, and can be disabled by simply giving `0` as the only argument in the list (the `0` indicates "false" for front output). If this option is chosen, the remaining two arguments can be omitted.

3.2 Input Using the Python Module

`fdfault` is a python module for setting up dynamic rupture problems for use with the C++ code.

3.2.1 Overview

The Python module closely resembles the structure of the text input files and hence the structure of the C++ code, and has an interface for specifying all simulation parameters through the `problem` class. The module is particularly useful for handling situations where inputs must be written to file in binary format. The module also includes functions that facilitate finding coordinate values nearest certain spatial locations for choosing indices for creating output units.

One benefit in using the Python module is that the code performs an extensive series of checks prior to writing the simulation data to file. This, plus using the interfaces that are part of the `problem` class, grealy improves the likelihood that the simulation will be set up correctly, and is highly recommended if you will be using the code to simulate complex problems.

While the module contains all classes (and you can set up simulations yourself using them), mostly you will be using the wrappers provided through the `problem` class, plus the constructors for `surface`, `curve`, `material`, `load`, `loadfile`, `swparam`, `swparamfile`, `stzparam`, `stzparamfile`, `statefile`, and `output`.

Details on the methods are provided in the documentation for the individual classes.

3.2.2 Requirements

The only external package required to use the Python module is NumPy, which uses numerical arrays to hold parameter values and writes them to file in binary format. The code has been tested starting with Numpy 1.9 and subsequent releases, though it will probably also work with some older versions. The code supports both Python 2 and Python 3 and has been tested on both versions of Python.

3.2.3 Main Classes

The Python module is set up as a collection of classes. Most of them operate under the hood and correspond to an equivalent class in the C++ code and are not directly modified by the user. Instead, the `problem` class is used, which contains a series of interfaces for modifying the underlying classes. Additional classes that are directly instantiated by the user include load and friction perturbations and output units.

The `problem` Class

The main class used for creating problems is the `problem` class. `problem` holds all relevant variables and classes needed to specify a simulation, and provides interfaces to automatically create the necessary classes when modifying the simulation. The class also contains methods for searching the grids that will be generated in a simulation in order to find specific points where output is desired. After the simulation is set up, it also has a method to write all information to file when complete.

class `fdfault.`**`problem`**(*name*)

Class describing a dynamic rupture problem.

This is the main class used in the python module. The problem class holds all relevant information for setting up the simulation, and any modifications to a problem should be done with the included interfaces.

To create a problem, a problem name (string) is required to initialize an instance of `problem`

```
>>> import fdfault
>>> p = fdfault.problem('myproblem')
```

This will initialize a problem with the default attributes. This includes the following:

Variables

- **name** (*str*) – Problem name (string, must be provided when initializing)
- **datadir** (*str*) – Data directory where output will be saved (default is 'data/')
- **nt** (*int*) – Number of time steps (default is 0)
- **dt** (*float*) – Time step size (default is 0.)
- **ttot** (*float*) – Total simulation time (default is 0.)
- **cfl** (*float*) – Courant ratio (dt * wave speed / dx, must be less than 1., default 0.)
- **ninfo** (*int*) – Frequency at which information is printed to the terminal during a simulation (default 10)
- **rkorder** (*int*) – Order of accuracy of time integration (default is 1)
- **d** (domain) – Initializing a problem also creates a new domain, which can be modified using the methods below.
- **outputlist** (*list*) – Initializing a problem creates an empty output list. To create output items, add them to the list using the appropriate method.
- **frt** (front) – A new problem contains a front with output turned off. To turn on front output, use the appropriate method.

The four variables related to the time step provide several ways to set the time step. You can set the time step using any pair of the variables *except* the time step and the Courant ratio. If you provide more than two, the code defaults to the total time and either the time step or the Courant ratio if the time step is not provided.

Use the methods described below to modify the simulation. When the problem is fully set-up, you can write the result to file

```
>>> p.write_output()
```

This will create the file `myproblem.in` in the current directory as well as any necessary binary files.

__init__(*name*)

Creates a new instance of the `problem` class

Initializes a new instance of the `problem` class. Requires a problem name (string), other parameters are set to the following defaults:

- `datadir = 'data/'` (path is relative to the main code directory)
- `nt`, `dt`, `ttot`, and `cfl` are set to zero. You must specify two of these to set the time step, *except* the time step and the Courant ratio
- `ninfo = 10`
- `rkorder = 1`
- An empty output list is initialized
- front output is `False`
- A `domain` is created with a single block with 1 grid point in each direction, default material properties, and a 2nd order finite difference method. All boundary conditions are set to `'none'`

All properties can be modified using the provided interfaces through the problem class.

Parameters **`name`** (*`str`*) – Name for new problem

Returns New problem instance

Return type *problem*

`add_load`(*newload*, *index=None*)

Adds load to interface

Add a load perturbation to the interface with the given index. If no index is provided, the load will be added to all interfaces. If the index is an integer, the load will be added to the interface with that index. Finally, the index can be an interable (list or tuple) of integers, and the load will be added to all interfaces in the iterable. Indices that are out of bounds or indicate an interface that is not frictional will raise an error. Default value is `None` (all interfaces).

`newload` must be a load perturbation (i.e. have type ~fdfault.load), or the code will raise an error. `newload` will be appended to the load list

Parameters

- **`newload`** (`load`) – Load to be added
- **`index`** (*`int or tuple or list or None`*) – Interface to which the load should be added. Can be a single integer, iterable of integers, or `None` to add to all interfaces (default is `None`)

Returns None

`add_output`(*item*)

Adds output item to output list

Add new output item `item` to output list. `item` must have type `output` or the code will raise an error. The item will be added to the end of the list (order is not important for output lists)

Parameters **`item`** (`output`) – New output item

Returns None

`add_pert`(*newpert*, *index=None*)

Add new friction parameter perturbation to an interface

Method adds a frictional parameter perturbation to an interface. `newpert` must be a parameter perturbation of the correct kind for the given interface type (i.e. if the interface is of type `slipweak`, then `newpert` must have type `swparam`).

`index` indicates the index of the interface to which the perturbation will be added. `index` can be a single integer index, an iterable containing multiple indices, or `None` to add to all interfaces (default behavior is `None`). Out of bounds values will raise an error.

Parameters

- **`newpert`** (*`pert (more precisely, one of the derived classes of friction parameter perturbations)`*) – New perturbation to be added. Must have a type that matches the interface(s) in question.
- **`index`** (*`int or list or tuple or None`*) – Index of interface to which the perturbation will be added (single index or iterable of indices, or `None` for all interfaces, optional)

Returns None

`check`()

Checks problem for errors

No inputs, no return value, and the problem will not be modified.

This is run automatically when calling `write_input`. You may also run it manually to see if the problem contains self-consistent input values.

In addition to checking values relevant to the problem class, `check` is run for all relevant classes contained in a rupture problem. This includes `domain` (which will also run `check` on itself, as well as any included output units (which are individually checked against the total number of spatial grid points and time steps in the simulation). However, it will only print a warning and the input file will still be written if any of these situations fail (this is mostly to alert the user, as the C++ code will simply adjust the relevant indices to fall within the values in the simulation).

Returns None

`delete_block_surf`(*coords*, *loc*)

Removes boundary surface for a particular block edge

Removes the bounding surface of a particular block edge. The block is selected by using `coords`, which is a tuple or list of 3 integers indicated block coordinates. Within that block, location is determined by `loc` which is an integer that indexes into a list. Locations correspond to the following: 0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top. Note that the location must be 0 <= loc < 2*ndim

If `coords` or `loc` is out of bounds, the code will also signal an error.

Parameters

- **`coords`** (*`tuple or list`*) – Coordaintes of desired block (tuple or list of 3 integers)
- **`loc`** (*`int`*) – Location of desired boundary to be removed (0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top). For 2D problems, `loc` must be between 0 and 3.

Returns None

`delete_load`(*niface*, *index=-1*)

Deletes load from index niface at position index from the list of loads

Deletes loads from a frictional interface. `niface` is an index refering to the desired interface. Out of bounds values or interfaces that are not frictional will result in an error.

`index` indicates the position in the load list that should be deleted. Default for `index` is `-1` (most recently added).

Parameters

- **`niface`** (*`int`*) – Interface from which the load should be removed. `niface` must refer to a frictional interface
- **`index`** (*`int`*) – Index within the load perturbation that should be removed (default is last)

Returns None

`delete_loadfile`(*niface*)

Deletes loadfile for given interface

Deletes the loadfile for the specified interface. `niface` is the index of the interface from which to delete the loadfile, and values that are not valid indices, or indices that refer to non-frictional interfaces will result in an error.

Parameters **`niface`** (*`int`*) – Index of interface for loadfile removal

Returns None

`delete_output`(*index=-1*)

Delete output item

Deletes the output item at the given location `index` within the output list. If no index is provided, it pops the most recently added item.

Parameters **`index`** (*`int`*) – Index of output item to remove

Returns None

`delete_paramfile`(*niface*)

Deletes friction parameter file for given interface

Removes the friction parameter file for the interface with index `niface`. The interface in question must be a frictional interface that can accept parameter files.

Parameters **`niface`** (*`int`*) – Index of interface that will have its paramfile removed

Returns None

`delete_pert`(*niface*, *index=-1*)

Deletes frictional parameter perturbation from interface

`niface` is an integer indicating the index of the desired interface. If out of bounds, will give an error.

`index` is an integer that indicates the position within the list of perturbations. Default is most recently added (-1).

Parameters

- **`niface`** (*`int`*) – Index of interface from which to remove the parameter perturbation
- **`index`** (*`int`*) – Index within perturbation list of the given interface to remove. Default is last item (-1, or most recently added)

Returns None

`delete_statefile`(*niface*)

Deletes statefile for given interface

Delete the statefile for a given interface. `niface` must be a valid index that refers to an interface with a state variable. Will set the statefile attribute for the interface to None.

Parameters **`niface`** (*`int`*) – Index of interface that will have its statefile removed

Returns None

`find_nearest_point`(*point*, *known=None*, *knownloc=None*)

Finds the coordinate indices closest to a desired set of grid values

Method takes a set of grid values (tuple or list of 2 or 3 floats) and finds the indices of the grid point closest to that location (in terms of Euclidean distance). The method returns a set of coordinates (tuple of length 3 of integers) of point that is closest to the input point.

The method also allows you to search along a given interface. To do this, you must pass `known = 'x'` (or `'y'` or `'z'` depending on the normal direction of the interface) and the known index in `knownloc` (integer value, which does not necessarily need to be on an interface – it just fixes that coordinate when performing the search)

The location is found using an iterative binary search algorithm. The search begins along the x direction using binary search until the distance to the desired point's x coordinate is minimized. The search then proceeds in the y and z directions. The algorithm then searches again in the x direction, y direction, and z direction, until the coordinates do not change over an entire iteration. This iteration procedure needs to take place because the coordinate directions are not independent. The algorithm is usually fairly efficient and finds coordinates fairly quickly.

Parameters

- **point** (*tuple or list*) – Desired spatial location (tuple or list of floats)
- **known** (*str or None*) – Spatial direction to fix during search (optional, string)
- **knownloc** (*int or None*) – Fixed coordinate value along `known` direction (optional, integer)

Returns Closest spatial coordinate (tuple of 3 integers)

Return type tuple

get_block_lx(*coords*)

Returns physical size of a block with a given set of coordinates. Note that this assumes the block is rectangular. It can be overridden by setting the edge of a block to be a curve (2D) or surface (3D), so this is not always the definitive size of a block.

Parameters coords (*tuple or list*) – Coordinates of desired block (tuple or list of three integers)

Returns Dimensions of desired block (tuple of three floats)

Return type tuple

get_block_surf(*coords*, *loc*)

Returns block boundary surface for a block edge

Returns the surface assigned to a specific block along a specific edge. The block is chosen using `coords` which is a tuple or list of 3 positive integers that corresponds to the coordinates of the block. Within that block, `loc` determines the edge that is returned (integer, corresponding to an index). Location indices correspond to the following: 0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top Note that the location must be 0 <= loc < 2*ndim (for 2D problems, `loc` cannot be 5 or 6).

Returns either a curve (2D problems) or surface (3D problems) or None

If `coords` or `loc` indices are out of bounds, the code will raise an error.

Parameters

- **`coords`** (*`tuple or list`*) – Coordaintes of desired block (tuple or list of 3 integers)
- **`loc`** (*`int`*) – Location of desired boundary (0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top). For 2D problems, `loc` must be between 0 and 3.

Returns curve or surface corresponding to the selected block and location. If the desired edge does not have a bounding surface, returns None.

Return type curve or surface or None

`get_block_xm`(*coords*)

Returns starting index (zero-indexed) of each block (list of three lists of integers)

Parameters **`coords`** (*`tuple or list`*) – Coordinates of desired block (tuple or list of three integers)

Returns list of three lists (each list is a list of integers)

Return type list

`get_bm`(*index*)

Returns block in minus direction of interface index. Returns a tuple of 3 integers indicating block coordinates of target block

Parameters **`index`** (*`int`*) – index of desired interface (zero-indexed)

Returns tuple

`get_bounds`(*coords*, *loc=None*)

Returns boundary types of a particular block

If `loc` (int) is provided, the method returns a specific location (str). Otherwise it returns a list of all boundaries, which will have length 4 for 2D problems and length 6 for 3D problems. `loc` serves effectively as an index into the list, and the indices correspond to the following: 0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top. Note that the location must be 0 <= loc < 2*ndim

Parameters

- **`coords`** (*`tuple`*) – Block coordinate location (list or tuple of three integers)
- **`loc`** (*`int or None`*) – Location of boundary that is desired (optional). If `loc` is not provided, returns a list

Returns Boundary type (if `loc` provided, returns a string of the boundary type for the desired location, of not returns a list of strings indicating all boundary types)

Return type str or list

`get_bp`(*index*)
Returns block in plus direction of interface index. Returns a tuple of 3 integers indicating block coordinates of target block

Parameters **`index`** (*`int`*) – index of desired interface (zero-indexed)

Returns tuple

`get_cdiss`()
Returns artificial dissipation coefficient

Returns Artificial dissipation coefficient

Return type float

`get_cfl`()
Returns Courant ratio (dt * wave speed / grid spacing)

Returns Courant ratio

Return type float

`get_datadir`()
Returns data directory (data directory can be a relative or absolute path)

Returns Data directory

Return type str

`get_direction`(*index*)
Returns direction (formally, normal direction in computational space) of interface with given index Returns a string 'x', 'y', or 'z', which is the normal direction for a simulation with rectangular blocks

Parameters **`index`** (*`int`*) – index of desired interface (zero-indexed)

Returns str

`get_dt`()
Returns time step size

Returns Time step size

Return type float

`get_front_field`()
Returns front field

Returns Rupture front field (string, "U" denotes slip and "V" denotes slip velocity)

Return type str

`get_front_output``()`

Returns status of front output (boolean)

Returns Status of front output

Return type bool

`get_front_value``()`

Returns front threshold value. Front output is the earliest time at which the given field exceeds this value

Returns Threshold value for rupture front output

Return type float

`get_het_material``()`

Returns heterogeneous material properties for simulation

Returns a numpy array with shape `(3,nx,ny,nz)`. First index indicates parameter value (0 = density, 1 = Lame parameter, 2 = Shear modulus). The other three indicate grid coordinates. If no heterogeneous material parameters are specified, returns `None`

Returns ndarray

`get_het_stress``()`

Returns heterogeneous stress initial values.

Returns a numpy array with shape `(ns,nx,ny,nz)`. First index indicates stress component. The following three indices indicate grid coordinates.If no array is currently specified, returns `None`.

For 2D mode 3 problems, indices for `ns` are (0 = sxz, 1 = syz)

For elastic 2D mode 2 problems, indices for `ns` are (0 = sxx, 1 = sxy, 2 = syy). For plastic 2D mode 2 problems, indices for `ns` are (0 = sxx, 1 = sxy, 2 = syy, 3 = szz).

For 3D problems, indices for `ns` are (0 = sxx, 1 = sxy, 2 = sxz, 3 = syy, 4 = syz, 5 = szz)

Returns ndarray

`get_iftype``(`*index=None*`)`

Returns interface type of given index, if none provided returns full list

Parameters **`index`** (*`int`*) – (optional) index of desired interface (zero-indexed). If not given or if `None` is given the entire list of interface types is returned

Returns str or list

`get_load``(`*niface*, *index=None*`)`

Returns load for index niface at position index. If no index provided, returns entire list of perturbations

Parameters

- **`niface`** (*`int`*) – index of desire interface (zero-indexed)
- **`index`** (*`int`*) – (optional) index of perturbation. If not provided or None, then returns entire list

Returns load or list

`get_loadfile`(*niface*)

Returns loadfile for interface with index niface

Loadfile sets any surface tractions set for the particular interface in question. Note that these tractions are added to any any set by the constant initial stress tensor, initial heterogeneous stress, or interface traction perturbations

Parameters **`niface`** – index of desired interface (zero-indexed)

Returns Current loadfile for the interface (if the interface does not have a loadfile, returns None)

Return type loadfile or None

`get_material`(*coords*)

Returns material properties for a given block

Returns the material class associated with block with coordinates `coords`. `coords` must be a tuple or list of valid block indices

Parameters **`coords`** (*`tuple or list`*) – Coordinates of the target block (tuple or list of 3 nonnegative integers)

Returns Material class with properties for this block

Return type *material*

`get_mattype`()

Returns material type ('elastic' or 'plasitc')

Returns Material type

Return type str

`get_mode`()

Returns rupture mode (2 or 3), only valid for 2D problems (stored at domain level)

Returns Rupture mode

Return type int

`get_name`()

Returns problem name

Returns Problem name

Return type str

get_nblocks()
Returns number of blocks points in (nx, ny, nz) format

Returns Number of blocks (tuple of three integers)

Return type tuple

get_nblocks_tot()
Returns total number of blocks

Returns Total number of blocks

Return type int

get_ndim()
Returns Number of spatial dimensions

Returns Number of spatial dimensions

Return type int

get_nifaces()
Returns number of interfaces

Returns Number of interfaces

Return type int

get_ninfo()
Returns frequency at which information is written to the terminal during a simulation

Returns Frequency of terminal output

Return type int

get_nloads(*index*)
Returns number of loads on interface with given index

Parameters **index** – index of desire interface (zero-indexed)

Returns int

get_nperts(*index*)
Returns number of frictional parameter perturbations (integer) on given interface with given index

Parameters **index** (*int*) – index of desired interface (zero-indexed)

Returns int

get_nproc()
Returns number of processes (in x, y, z directions).

0 means MPI will do the domain decomposition in that direction automatically

Returns Number of processes in each spatial dimension (x, y, z) (tuple of three integers)

Return type tuple

get_nt()
Returns number of time steps

Returns Number of time steps

Return type int

get_nx()
Returns number of grid points in (nx, ny, nz) format

Returns Number of grid points (tuple of three integers)

Return type tuple

get_nx_block()
Returns number of grid points in each block along each spatial dimension

Returns Number of grid points in each block (list of three lists)

Return type list

get_output(*index=None*)
Returns output item at given index (if none give, returns entire list)

Parameters **index** (`int`) – (optional) index of desired output unit. If not given or if `None` is given the entire list of output units is returned

Returns output item or list of output items

Return type output or list

get_paramfile(*niface*)
Returns paramfile (holds arrays of heterogeneous friction parameters) for interface with index niface. Can return a subtype of paramfile corresponding to any of the specific friction law types.

Parameters **niface** (`int`) – index of desired interface (zero-indexed)

Returns paramfile

get_pert(*niface*, *index=None*)
Returns perturbation for index niface at position index

Method returns a perturbation from a particular interface. `niface` must be a valid integer index referring to an interface. `index` is the index into the perturbation list for the particular index. If `index` is not provided or is `None`, the method returns the entire list.

Parameters

- **`niface`** (*`int`*) – Index referring to an interface. (Must be a valid integer index.)
- **`index`** (*`int or None`*) – Index into the perturbation list for the index in question (optional, if not provided or `None`, then returns entire list)

Returns pert or list

`get_rkorder`()
Returns order of accuracy of time integration

Returns Order of accuracy of time integration

Return type int

`get_sbporder`()
Returns order of accuracy of finite difference method (stored at domain level)

Returns Order of accuracy of finite difference method

Return type int

`get_state`(*niface*)
Returns initial state variable value for interface with index niface

Parameters **`niface`** – index of desired interface (zero-indexed)

Returns Initial state variable

Return type float

`get_statefile`(*niface*)
Returns state file of given interface

If interface does not have a statefile returns None

Parameters **`niface`** – index of desired interface (zero-indexed)

Returns statefile or None

`get_stress`()
Returns uniform intial stress values

Note that 2D simulations do not use all stress components. Mode 2 elastic simulations only use `sxx`, `sxy`, and `syy`, and mode 3 elastic simulations use `sxz`, and `syz` (though the normal stresses `sxx` and `syy` can be set to constant values that are applied to any frictional failure criteria). Mode 2 plastic simulations use `szz`, and mode 3 plastic simulations use all three normal stress components in evaluating the yield criterion.

Returns Initial stress tensor (list of floats). Format is `[sxx,sxy,sxz,syy,syz,szz]`

Return type list

get_ttot()

Returns total simulation time `ttot`

Returns Total simulation time

Return type float

get_x(*coord*)

Returns grid value for given spatial index

For a given problem set up, returns the location of a particular set of coordinate indices. Note that since blocks are set up by setting values only on the edges, coordinates on the interior are not specified *a priori* and instead determined using transfinite interpolation to generate a regular grid on the block interiors. Calling `get_x` generates the interior grid to find the coordinates of the desired point.

Within each call to `get_x`, the grid is generated on the fly only for the relevant block where the desired point is located. It is not stored. This helps reduce memory requirements for large 3D problems (since the Python module does not run in parallel), but is slower. Because the computational grid is regular, though, it can be done in a single step in closed form.

Returns a numpy array of length 3 holding the spatial location (x, y, z).

Parameters **coord** (*tuple or list*) – Spatial coordinate where grid values are desired (tuple or list of 3 integers)

Returns (x, y, z) coordinates of spatial location

Return type ndarray

set_block_lx(*coords*, *lx*)

Sets block with coordinates `coords` to have dimension `lx`

`coords` is a tuple of nonnegative integers that indicates the coordinates of the desired block (0-indexed, must be less than the number of blocks in that particular direction or the code will raise an error). `lx` is a tuple of two (2D) or three (3D) positive floats indicating the block length in each spatial dimension. Note that this assumes each block is rectangular. When a single block is modified, the code automatically adjusts the lower left corner of all simulation blocks to be consistent with this change.

This can be overridden by setting a block edge to be a curve (2D) or surface (3D). However, traction and friction parameter perturbations still make use of these block lengths when altering interface tractions or friction parameters. More information on how this works is provided in the `pert` documentation.

Finally, note that neighboring blocks must have conforming grids. When writing simulation data to file, the code checks that all interfacial grids match, and raises an error if it disagrees. So while the `set_block_lx` method may not complain about an error like this, you will not be able to save the simulation to a file with such an error.

Parameters

- **coords** (*tuple or list*) – Coordinates (tuple or list of 3 nonnegative integers)
- **lx** (*tuple or list*) – New dimensions of desired block (tuple or list of 2 or 3 positive floats)

Returns None

set_block_surf (*coords*, *loc*, *surf*)

Sets boundary surface for a particular block edge

Changes the bounding surface of a particular block edge. The block is selected by using `coords`, which is a tuple or list of 3 integers indicated block coordinates. Within that block, location is determined by `loc` which is an integer that indexes into a list. Locations correspond to the following: 0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top. Note that the location must be 0 <= loc < 2*ndim

For 2D problems, `surf` must be a curve. For 3D problems, `surf` must be a surface. Other choices will raise an error. If `coords` or `loc` is out of bounds, the code will also signal an error.

Parameters

- **coords** (*tuple or list*) – Coordaintes of desired block (tuple or list of 3 integers)
- **loc** (*int*) – Location of desired boundary (0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top). For 2D problems, `loc` must be between 0 and 3.
- **surf** (*curve or surface*) – curve or surface corresponding to the selected block and location

Returns None

set_bounds (*coords*, *bounds*, *loc=None*)

Sets boundary types of a particular block.

Changes the type of boundary conditions on a block. Acceptable values are 'absorbing' (incoming wave amplitude set to zero), 'free' (no traction on boundary), 'rigid' (no displacement of boundary), or 'none' (boundary conditions set by imposing interface conditions).

The block to be modified is determined by `coords`, which is a tuple or list of 3 integers that match the coordinates of a block.

There are two ways to use `set_bounds`:

1. Set `loc` to be `None` (default) and provide a list of strings specifying boundary type for `bounds`. The length of `bounds` is 4 for a 2D simulation and 6 for 3D.
2. Set `loc` to be an integer denoting location and give `bounds` as a single string. The possible locations correspond to the following: 0 = left, 1 = right, 2 = front, 3

= back, 4 = bottom, 5 = top. 4 and 5 are only applicable to 3D simulations (0 <= loc < 2*ndim).

Parameters

- **`coords`** (*`tuple or list`*) – Location of block to be modifies (tuple or list of 3 integers)
- **`bounds`** (*`str or list`*) – New boundary condition type (string or list of strings)
- **`loc`** (*`int or None`*) – If provided, only change one type of boundary condition rather than all (optional, loc serves as an index into the list if used)

Returns None

`set_cdiss`(*cdiss*)

Sets artificial dissipation coefficient

New artificial dissipation coefficient must be nonnegative. If it is set to zero, the code will not use artificial dissipation in the simulation.

There is not a hard and fast rule for setting the coefficient, so some degree of trial and error may be necessary. Values around 0.1 have worked well in the past, but that may not be true for all meshes.

Parameters **`cdiss`** (*`float`*) – New artificial dissipation coefficient

Returns None

`set_cfl`(*cfl*)

Sets CFL ratio The CFL ratio must be between 0. and 1. If the provided value is not a float, it will be converted into a float

The four variables related to the time step provide several ways to set the time step. You can set the time step using any pair of the variables *except* the time step and the Courant ratio. If you provide more than two, the code defaults to the total time and either the time step or the Courant ratio if the time step is not provided.

Parameters **`cfl`** (*`float`*) – New value for the CFL ratio

Returns None

`set_datadir`(*datadir*)

Sets problem data directory to new value Method checks if datadir is a string and ends in '/', but does not check that it is a valid path

Parameters **`name`** (*`str`*) – New problem data directory (must be a string)

Returns None

set_domain_xm(*xm*)

Sets lower left corner of domain to spatial coordinate xm

Moves the lower left corner of the simulation. This does not affect block lengths, only the minimum spatial location of the entire comain in each cartesian direction. Individual block locations are calculated automatically from this and the length information for each block. Thus, you cannot set the location of each block directly, just the overall value of the domain and then all other blocks are positioned based on the length of other blocks.

If the simulation is 2D and a nonzero value for the z-coordinate is provided, the z position of all blocks will be automatically set to zero.

Note that the location of any block can be overridden by setting the edges to be surfaces. The corners must still match one another (this is checked when writing the simulation data to file), and neighboring blocks must have conforming grids at the edges.

Parameters **xm** (`tuple or list`) – New lower left coordinate of simulation domain (tuple of 2 or 3 floats)

Returns None

set_dt(*dt*)

Sets time step New time step cannot be negative (will trigger an error) If time step is not a float, it is converted to a float

The four variables related to the time step provide several ways to set the time step. You can set the time step using any pair of the variables *except* the time step and the Courant ratio. If you provide more than two, the code defaults to the total time and either the time step or the Courant ratio if the time step is not provided.

Parameters **dt** (`float`) – New time step

Returns None

set_front_field(*field*)

Sets rupture front field

Sets new value of rupture front field `field`. `field` must be a string (slip (`'U'`) or slip velocity (`'V'`)). Other choices will raise an error.

Parameters **field** (`str`) – New rupture front field

Returns None

set_front_output(*output*)

Sets front output to be on or off

Sets rupture front output to be the specified value (boolean). Will raise an error if the provided value cannot be converted into a boolean.

Parameters **output** (`bool`) – New value of output

Returns None

`set_front_value`(*value*)

Sets front threshold value

Changes value of rupture front threshold. The rupture time is the earliest time at which the chosen field exceeds this value. `value` is the new value (must be a positive number).

Parameters **`value`** (*`float`*) – New values of the threshold for rupture front times.

Returns None

`set_het_material`(*mat*)

Sets heterogeneous material properties for simulation

New heterogeneous material properties must be a numpy array with shape `(3,nx,ny,nz)`. First index indicates parameter value (0 = density, 1 = Lame parameter, 2 = Shear modulus). The other three indicate grid coordinates

Parameters **`mat`** (*`ndarray`*) – New material properties array (numpy array with shape `(3,nx,ny,nz)`)

Returns None

`set_het_stress`(*s*)

Sets heterogeneous stress initial values

Sets initial heterogeneous stress. New stress must be a numpy array with shape `(ns,nx,ny,nz)`. First index indicates stress component. The following three indices indicate grid coordinates.

For 2D mode 3 problems, indices for `ns` are (0 = sxz, 1 = syz)

For elastic 2D mode 2 problems, indices for `ns` are (0 = sxx, 1 = sxy, 2 = syy). For plastic 2D mode 2 problems, indices for `ns` are (0 = sxx, 1 = sxy, 2 = syy, 3 = szz).

For 3D problems, indices for `ns` are (0 = sxx, 1 = sxy, 2 = sxz, 3 = syy, 4 = syz, 5 = szz)

Parameters **`s`** (*`ndarray`*) – New heterogeneous stress array (numpy array with shape `(3,nx,ny,nz)`

Returns None

`set_iftype`(*index*, *iftype*)

Sets type of interface with a given index

Changes type of a particular interface. `index` is the index of the interface to be modified and `iftype` is a string denoting the interface type. Valid values for `iftype` are `'locked'`, `'frictionless'`, `'slipweak'`, and `'stz'`. Any other values will result in an error, as will an interface index that is out of bounds.

Parameters

- **index** (*int*) – Index (nonnegative integer) of interface to be modified
- **iftype** (*str*) – New interface type (see valid values above)

Returns None

set_loadfile(*niface*, *newloadfile*)

Sets loadfile for interface with index niface

`niface` indicates the index of the interface that will be modified, and must be a frictional interface. `newloadfile` is the new loadfile (must have type `loadfile`). If the index is bad or the loadfile type is not correct, the code will raise an error. Errors can also result if the shape of the loadfile does not match with the interface.

Parameters

- **niface** – index of desired interface (zero-indexed)
- **newloadfile** (`loadfile`) – New loadfile to be used for the given interface

Returns None

set_material(*newmaterial*, *coords=None*)

Sets block material properties for the block with indices given by `coords`

If `coords` is not provided, all blocks are changed to have the given material properties. `newmaterial` must have a type `material` and `coords` must be a tuple or list of three integers that match the coordinates of a block.

If `set_material` changes all blocks in the simulation, it also changes the material type for the whole simulation (equivalent to calling `set_mattype`). If `set_material` acts only on a single block, the new material type of that block must match the one set in the `fields` type (i.e. the return value of `get_mattype`).

Parameters

- **newmaterial** (`material`) – New material properties
- **coords** (*tuple or list*) – Coordinates of block to be changed (optional, omitting changes all blocks). `coords` must be a tuple or list of three integers that match the coordinates of a block.

Returns None

set_mattype(*mattype*)

Sets field and block material type ('elastic' or 'plastic')

Sets the material type for the simulation. Options are 'elastic' for an elastic simulation and 'plastic' for a plastic simulation. Anything else besides these options will cause the code to raise an error.

Once the simulation type is altered, all blocks material types are changed as well. This is necessary to ensure that the right set of parameters are written to file. Note that all blocks must therefore have the same material type, though you can ensure that a given block always behaves elastically by setting an appropriate value for the yield criterion.

Parameters **`mattype`** (*`str`*) – New material type ('elastic' or 'plastic')

Returns None

`set_mode`(*mode*)
Sets rupture mode

Rupture mode is only valid for 2D problems, and is either 2 or 3 (other values will cause an error, and non-integer values will be converted to integers). For 3D problems, entering a different value of the rupture mode will alter the rupture mode cosmetically but will have no effect on the simulation.

Parameters **`mode`** (*`int`*) – New value of rupture mode

Returns None

`set_name`(*name*)
Sets problem name

Parameters **`name`** (*`str`*) – New problem name (must be a string)

Returns None

`set_nblocks`(*nblocks*)
Sets number of blocks

`set_nblocks` alters the number of blocks in the simulation. The method adds or deletes blocks from the list of blocks as needed. Depending on how the number of blocks is changed, new blocks may only have a single grid point, or if added in a direction where the number of blocks is already established the number of grid points may be copied from the existing simulation. If in doubt, use `get_nx_block` to check the number of grid points and use `set_nx_block` to modify if necessary.

Parameters **`nblocks`** (*`tuple`*) – New number of blocks (tuple of 3 positive integers)

Returns None

`set_ndim`(*ndim*)
Sets number of dimensions

The new number of spatial dimensions must be an integer, either 2 or 3. If a different value is given, the code will raise an error. If a non-integer value is given that is acceptable, the code will convert it to an integer.

Note: Converting a 3D problem into a 2D problem will automatically collapse the number of grid points and the number of blocks in the z direction to be 1. Any modifications to these quantities that were done previously will be lost.

Parameters **ndim** (*int*) – New value for ndim (must be 2 or 3)

Returns None

set_ninfo(*ninfo*)

Sets frequency of information output

The simulation will write out information to the terminal after each `ninfo` time steps. `ninfo` must be a positive integer (if less than zero, will trigger an error). `ninfo` is converted into an integer.

Parameters **ninfo** (*int*) – New value of ninfo

Returns None

set_nproc(*nproc*)

Sets number of processes in domain decomposition manually

New number of processes `nproc` must be a tuple/list of nonnegative integers. If the problem is 2D, the number of processes in the z direction will automatically be set to 1. Any number can be set to zero, in which case MPI will set the number of processes in that direction automatically. If all three numbers are nonzero, then it is up to the user to ensure that the total number of processors ($nx imes ny imes nz$) is the same as the total number when running the executable.

Parameters **nproc** (*tuple*) – New number of processes (must be a tuple of positive integers)

Returns None

set_nt(*nt*)

Sets number of time steps New number of time steps cannot be negative (will trigger an error) If number of time steps is not an integer, it is converted to an integer

The four variables related to the time step provide several ways to set the time step. You can set the time step using any pair of the variables *except* the time step and the Courant ratio. If you provide more than two, the code defaults to the total time and either the time step or the Courant ratio if the time step is not provided.

Parameters **nt** (*int*) – New number of time steps

Returns None

set_nx_block(*nx_block*)

Set number of grid points in each block as a list of lists.

Input must be a list or tuple of length 3, with each item a list of integers representing the number of grid points for each block along the respective dimension. If the list lengths do not match the number of blocks, the code will raise an error. The blocks must form a regular cartesian grid with conforming edges, so all blocks along a single spatial dimension must have the same number of grid points along that spatial dimension.

For example, if nblocks = (3,2,1), then nblock[0] has length 3, nblock[1] has length 2, and nblock[2] has length 1. All blocks that are at position 0 in the x-direction will have nblock[0][0] grid points in the x-direction, all blocks at position 1 in the x-direction will have nblock[0][1] grid points in the x-direction, etc.

Parameters **`nx_block`** (*`list`*) – New number of grid points (list of 3 lists of positive integers)

Returns None

`set_paramfile` (*niface*, *newparamfile*)

Sets paramfile for given interface

Method sets the file holding frictional parameters for an interface. Interface to be modified is set by `niface`, which must be a valid index for an interface.

`newparamfile` must be a parameter perturbation file of the correct type for the given interface type (i.e. if the interface is of type `slipweak`, then `newpert` must have type `swparamfile`). Errors can also result if the shape of the paramfile does not match with the interface.

Parameters

- **`niface`** (*`int`*) – index of desired interface (zero-indexed)
- **`newparamfile`** (*`paramfile (actual type must be the appropriate subclass for the friction law of the particular interface and have the right shape)`*) – New frictional parameter file (type depends on interface in question)

Returns None

`set_rkorder` (*rkorder*)

Sets order of low storage RK method (integer 1-4).

Value is converted to an integer, and error will be signaled if a different value is given outside of this range.

Parameters **`rkorder`** (*`int`*) – New value of order of accuracy of integration

Returns None

`set_sbporder` (*sbporder*)

Sets finite difference order

Finite difference method order must be an integer 2-4. A value outside of this range will result in an error. If a non-integer value is given that is acceptable, it will be converted to an integer and there will be no error message.

Parameters **`sbporder`** (*`int`*) – New value of finite difference method order (integer 2-4)

Returns None

set_state (*niface*, *state*)
Sets initial state variable for interface

Set the initial value for the state variable for a given interface. `niface` is the index of the interface to be set (must be a valid integer index). The interface must have a state variable associated with it, or an error will occur. `state` is the new state variable (must be a float or some other valid number).

Parameters

- **niface** (*int*) – Index of interface to modify. Must be an interface with a state variable
- **state** (*float*) – New value of state variable

Returns None

set_statefile (*niface*, *newstatefile*)
Sets state file for interface

Set the statefile for the indicated interface. `niface` must be a valid index to an interface, out of bounds values will lead to an error. `newstatefile``must have type ``statefile` and the interface must support a state variable. Errors can also result if the shape of the statefile does not match with the interface.

Parameters

- **niface** (*int*) – Index of interface to be modified
- **newstatefile** (`statefile`) – New statefile for the interface in question.

Returns None

set_stress (*s*)
Sets uniform intial stress

Changes initial uniform stress tensor. New stress tensor must be a list of six floats.

Note that 2D simulations do not use all stress components. Mode 2 elastic simulations only use `sxx`, `sxy`, and `syy`, and mode 3 elastic simulations use `sxz`, and `syz` (though the normal stresses `sxx` and `syy` can be set to constant values that are applied to any frictional failure criteria). Mode 2 plastic simulations use `szz`, and mode 3 plastic simulations use all three normal stress components in evaluating the yield criterion.

Params s New stress tensor (list of 6 floats). Format is `[sxx,sxy,sxz,syy,syz,szz]`

Returns None

set_ttot(*ttot*)

Sets total simulation time Total time cannot be negative (will trigger an error) If total time is not a float, it is converted to a float

The four variables related to the time step provide several ways to set the time step. You can set the time step using any pair of the variables *except* the time step and the Courant ratio. If you provide more than two, the code defaults to the total time and either the time step or the Courant ratio if the time step is not provided.

Parameters **ttot** (`float`) – New value for total time

Returns None

write_input(*filename=None*, *directory=None*, *endian='='*)

Writes problem to input file

Method writes the current state of a problem to an input file, also writing any binary data to file (i.e. block boundary curves, heterogeneous stress tensors, heterogeneous material properties, heterogeneous interface tractions, heterogeneous state variables, or heterogeneous friction parameters).

All input arguments are optional. Possible arguments include `filename` (string, default is problem name) which will set the input file name (the code adds on `.in` to the provided filename), `directory` (string holding the path to the location where the file will be written, default is current directory), and `endian` to set byte-ordering for writing binary files (default is = (native), other options inlcude < (little) and > (big)).

When `write_input` is called, the code calls `check`, which verifies the validity of the simulation and alerts the user to any problems. `check` examines if block surface edges match, if neighboring blocks have matching grids, and other things that cannot be checked when modifying the simulation. The same checks are run in the C++ code when initializing a problem, so a problem that runs into trouble when calling `check` is likely to have similar difficulties when running the simulation.

Parameters

- **filename** (`str`) – name of file (default is problem name); code will add `.in` to this
- **directory** (`str`) – Location where input file should be written (default current directory)
- **endian** – Byte-ordering for files. Should match byte ordering of the system where the simulation will be run (it helps to run the Python script directly with native byte ordering enabled). Default is native (=), other options include < for little endian and > for big endian.

Returns None

The `material` Class

The `material` class contains information regarding block material properties. This includes whether the block is linear elastic or elastic-plastic in its deformation style, density, elastic modulii, and plastic failure criteria such as the internal friction coefficient, cohesion, dilatancy, and a viscoplastic "viscosity."

Each block in the domain is assigned a material with default properties when it is initialized. This can be changed by assigning a new material to a particular block using the interface provided in the `problem` class. The density and elastic modulii can also be overridden by creating a heterogeneous array of material properties that varies in a point-by-point fashion rather than having block material properties.

class `fdfault.`**`material`**(*mattype*, *rho=2.67*, *lam=32.04*, *g=32.04*, *mu=0.5735*, *c=0.0*, *beta=0.2867*, *eta=0.2775*)

Class describing block material properties

When a new block is initialized, one is created with the following default properties. When creating a new `material` yourself, you must select whether the block is elastic or plastic by specifying the `mattype` attribute, but the other values will be given default values (see below) if they are not specified.

Variables

- **`mattype`** (*`str`*) – Specifies if a block is elastic or plastic (value must be `'elastic'` or `'plastic'`)
- **`rho`** (*`float`*) – Density (default 2.67 MPa s^2 / km / m, see note below about funny units)
- **`lam`** (*`float`*) – First Lame parameter (default is 32.04 GPa)
- **`g`** (*`float`*) – Shear modulus (default is 32.04 GPa)
- **`mu`** (*`float`*) – Internal friction coefficient (only relevant for plastic materials, default is 0.5735)
- **`c`** (*`float`*) – Cohesion (default is 0.)
- **`beta`** (*`float`*) – Plastic dilatancy, determines ratio of dilational to shear strain (default 0.2867)
- **`eta`** (*`float`*) – Plastic viscosity, determines time scale over which stresses in excees of the yield surface decay back to the yield surface (default 0.2775 GPa s)

You are free to choose any self-consistent unit system that you like. For practical purposes, it is best to have all parameters be of order unity to reduce round-off errors when dealing with quantities of vastly different magnitudes. To facilitate this, the default parameters measure distance in km but slip in m, and elastic modulii in GPa but stresses in MPa, which result in quantities of order unity in all fields calculated in the solution for typical values found in simulating earthquake rupture on seismogenic faults. The extra factor of 10^3 cancels correctly in Hooke's law, but does not in the momentum conservation eqaution, meaning that the density must have funny units of MPa s^2 / km / m to correct for this.

`__init__`(*mattype*, *rho=2.67*, *lam=32.04*, *g=32.04*, *mu=0.5735*, *c=0.0*, *beta=0.2867*, *eta=0.2775*)

Create a new instance of the `material` class

Initialize a new material. The user must specify whether the material type is elastic or plastic, but all other parameters are optional (any not specified will take default values). All parameters must be positive (rho, lam, g, mu) or nonnegative (beta, eta, c).

Parameters

- **`mattype`** (*`str`*) – Material type, must be `'elastic'` or `'plastic'`
- **`rho`** (*`float`*) – Density
- **`lam`** (*`float`*) – First Lame parameter
- **`g`** (*`float`*) – Shear modulus
- **`mu`** (*`float`*) – Internal friction coefficient
- **`c`** (*`float`*) – Cohesion
- **`beta`** (*`float`*) – Plastic dilatancy
- **`eta`** (*`float`*) – Plastic viscosity

Returns New material instance

Return type *material*

`get_beta`()

Returns plastic dilatancy (ratio of dilataional to shear strain)

Returns Plastic dilatancy

Return type float

`get_c`()

Returns cohesion

Returns Cohesion

Return type float

`get_cp`()

Returns compressional wave speed

Returns Compressional wave speed

Return type float

get_cs()

Returns shear wave speed

Returns Shear wave speed

Return type float

get_eta()

Returns plastic "viscosity"

Viscosity determines the time scale over which stresses can exceed the yield stress

:returns:Plastic "viscosity" :rtype: float

get_g()

returns shear modulus

get_lam()

Returns first Lame parameter

Returns First Lame parameter

Return type float

get_mu()

Returns internal friction coefficient

Returns Internal friction coefficient

Return type float

get_rho()

Returns Density

Returns Density

Return type float

get_type()

Returns material type

Returns Material type (`'elastic'` or `'plastic'`)

Return type str

get_zp()

Returns compressional impedance

Returns Compressional impedance

Return type float

get_zs()

Returns shear impedance

Returns Shear impedance

Return type float

`set_beta`(*beta*)
Sets plastic dilatancy to a new value

Parameters **`beta`** (*`float`*) – New value of plastic dilatancy

Returns None

`set_c`(*c*)
Sets cohesion to a new value

Parameters **`c`** (*`float`*) – New value of cohesion

Returns None

`set_eta`(*eta*)
Sets plastic "viscosity" to a new value

Parameters **`eta`** (*`float`*) – New value of plastic viscosity

Returns None

`set_g`(*g*)
Sets shear modulus to a new value

Parameters **`g`** (*`float`*) – New value of shear modulus

Returns None

`set_lam`(*lam*)
Sets first Lame parameter to a new value

Parameters **`lam`** (*`float`*) – New value of first Lame parameter

Returns None

`set_mu`(*mu*)
Sets internal friction to a new value

Parameters **`mu`** (*`float`*) – New value of internal friction coefficient

Returns None

`set_rho`(*rho*)
Sets density to a new value

Parameters **`rho`** (*`float`*) – New value of density

Returns None

`set_type`(*mattype*)
Sets material type (must be either `'elastic'` or `'plastic'`)

Parameters **`mattype`** (*`str`*) – New material type (must be either `'elastic'` or `'plastic'`)

Returns None

write_input (*f*)

Writes material properties to input file

This method is called when writing each block to the input file. It is called automatically, and writes the material properties in the correct location within the inputs for each block. It also automatically handles whether or not the simulation is elastic or plastic, writing out the plastic parameters only if needed.

Parameters **f** (*file*) – Input file handle

Returns None

The `surface` and `curve` Classes

The main classes used for handling complex geometries are the `surface` and `curve` classes. `surface` is used in 3D problems, while `curve` is used in 2D problems. The only differences in the classes are the number of dimensions; otherwise they are identical.

class `fdfault.`**surface** (*n1*, *n2*, *direction*, *x*, *y*, *z*)

The surface class represents a surface for defining interfaces and block boundaries

Each surface contains the following attributes:

Variables

- **n1** – Number of grid points in the first spatial direction (x unless the interface is an `'x'` interface)
- **n2** – Number of grid points in the second spatial direction (z unless the interface is a `'z'` interface)
- **direction** – Normal direction in computational space
- **x** – Numpy array holding x coordinates, must have shape (n1,n2)
- **y** – Numpy array holding y coordinates, must have shape (n1,n2)
- **z** – Numpy array holding z coordinates, must have shape (n1,n2)

__init__ (*n1*, *n2*, *direction*, *x*, *y*, *z*)

Initialize a `surface` instance

Required arguments are n1 and n2, which are number of grid points in each direction, a direction which indicates the surface orientation in computational space (`'x'`, `'y'`, or `'z'`), plus three arrays x, y, and z (must have shape `(n1,n2)` that hold the coordinates for the new surface. Initializing with a negative number of grid points, with arrays that do not have the correct shape, or with a bad string for the surface orientation will result in an error.

Parameters

- **n1** (*int*) – Number of grid points along first dimension
- **n2** (*int*) – Number of grid points along second dimension
- **direction** (*str*) – String indicating surface normal direction in computational space (must be `'x'`, `'y'`, or `'z'`)
- **x** (*ndarray*) – Array holding surface x coordinates (must have shape `(n1,n2)`)
- **y** (*ndarray*) – Array holding surface y coordinates (must have shape `(n1,n2)`)
- **z** (*ndarray*) – Array holding surface z coordinates (must have shape `(n1,n2)`)

Returns New surface with specified properties

Return type *surface*

get_direction()

Returns approximate normal direction

Returns Normal direction in computational space

Return type str

get_n1()

Returns number of grid points along first dimension

Returns Number of grid points along first dimension (x unless interface direction is `'x'`)

Return type int

get_n2()

Returns number of grid points along second dimension

Returns Number of grid points along second dimension (z unless interface direction is `'z'`)

Return type int

get_x(*i=None*)

Returns x coordinate array

if no argument is provided, the method returns the entire array. Otherwise, `i` must be a valid index tuple for the array.

Parameters **i** (*tuple or None*) – Index tuple (must be a valid index into the array). Optional, if not provided or if `None` is given, this returns the entire array.

Returns Value of x coordinate

Return type ndarray or float

`get_y`(*i=None*)

Returns y coordinate array

if no argument is provided, the method returns the entire array. Otherwise, `i` must be a valid index tuple for the array.

Parameters **`i`** (*`tuple or None`*) – Index tuple (must be a valid index into the array). Optional, if not provided or if `None` is given, this returns the entire array.

Returns Value of y coordinate

Return type ndarray or float

`get_z`(*i=None*)

Returns z coordinate array

if no argument is provided, the method returns the entire array. Otherwise, `i` must be a valid index tuple for the array.

Parameters **`i`** (*`tuple or None`*) – Index tuple (must be a valid index into the array). Optional, if not provided or if `None` is given, this returns the entire array.

Returns Value of z coordinate

Return type ndarray or float

`has_same_edge`(*edge1*, *edge2*, *othersurf*)

Compares the edges of two surfaces

The method compares the edges of two surfaces, using the indices 0-3 to indicate the edges (one argument must be provided for each surface)

- 0 means edge where second index is zero
- 1 means edge where first index is zero
- 2 means edge where second index is n2-1
- 3 means edge where first index is n1-1

Returns a boolean.

Parameters

- **`edge1`** (*`int`*) – Edge of first surface to be used. Must be integer 0-3
- **`edge2`** (*`int`*) – Edge of second surface to be used. Must be integer 0-3
- **`othersurf`** (*`surface`*) – The second surface, must be a surface

Returns Whether or not the selected edges match

Return type bool

`write` (*filename*, *endian='='*)

Write surface to binary file

Method writes the surface to a binary file. Input arguments include the desired filename (required) and the byte ordering of the file (`'='` native, `'>'` big endian, `'<'` little endian; default is native)

Parameters

- **`filename`** (*`str`*) – Filename for output
- **`endian`** (*`str`*) – Byte ordering of output (optional, default is native)

Returns None

class `fdfault.`**`curve`** (*n*, *direction*, *x*, *y*)

The curve class represents a curve for defining interfaces and block boundaries in 2D problems

A curve is simply a surface class with the z spatial dimension removed. However, you cannot use curves and surfaces interchangeably as the C++ code reads the files for 2D and 3D problems differently, thus appropriate typing is enforced.

Each curve contains the following attributes:

Variables

- **`n1`** – Number of grid points (x for `'y'` interfaces, y for `'x'` interfaces)
- **`direction`** – Normal direction in computational space
- **`x`** – Numpy array holding x coordinates, must have shape (n1,)
- **`y`** – Numpy array holding y coordinates, must have shape (n1,)

`__init__` (*n*, *direction*, *x*, *y*)

Initialize a `curve` instance

Required arguments are n, which are number of grid points in each direction, a direction which indicates the surface orientation in computational space (`'x'` or `'y'`), plus three arrays x and y (must have shape `(n,)` that hold the coordinates for the new surface. Initializing with a negative number of grid points, with arrays that do not have the correct shape, or with a bad string for the surface orientation will result in an error.

Parameters

- **`n`** (*`int`*) – Number of grid points
- **`direction`** (*`str`*) – String indicating curve normal direction in computational space (must be `'x'` or `'y'`)
- **`x`** (*`ndarray`*) – Array holding surface x coordinates (must have shape `(n1,)`)

- **y** (*ndarray*) – Array holding surface y coordinates (must have shape `(n1,)`)

Returns New curve with specified properties

Return type *curve*

get_direction()

Returns approximate normal direction

Returns Normal direction in computational space

Return type str

get_n1()

Returns number of grid points along first dimension

Returns Number of grid points along first dimension (x unless interface direction is `'x'`)

Return type int

get_n2()

Returns number of grid points along second dimension

Returns Number of grid points along second dimension (z unless interface direction is `'z'`)

Return type int

get_x(*i=None*)

Returns x coordinate array

if no argument is provided, the method returns the entire array. Otherwise, `i` must be a valid index tuple for the array.

Parameters **i** (*tuple or None*) – Index tuple (must be a valid index into the array). Optional, if not provided or if `None` is given, this returns the entire array.

Returns Value of x coordinate

Return type ndarray or float

get_y(*i=None*)

Returns y coordinate array

if no argument is provided, the method returns the entire array. Otherwise, `i` must be a valid index tuple for the array.

Parameters **i** (*tuple or None*) – Index tuple (must be a valid index into the array). Optional, if not provided or if `None` is given, this returns the entire array.

Returns Value of y coordinate

Return type ndarray or float

`get_z`(*i=None*)
Returns z coordinate array

if no argument is provided, the method returns the entire array. Otherwise, `i` must be a valid index tuple for the array.

Parameters **`i`** (*`tuple or None`*) – Index tuple (must be a valid index into the array). Optional, if not provided or if `None` is given, this returns the entire array.

Returns Value of z coordinate

Return type ndarray or float

`has_same_edge`(*edge1*, *edge2*, *othersurf*)
Compares the edges of two curves

The method compares the edges of two curves, using the indices 1 or 3 to indicate the edges (one argument must be provided for each curve). Note that this definition is made to be consistent with the surface class.

- 1 means edge where first index is zero
- 3 means edge where first index is n1-1

Returns a boolean.

Parameters

- **`edge1`** (*`int`*) – Edge of first surface to be used. Must be integer 1 or 3
- **`edge2`** (*`int`*) – Edge of second surface to be used. Must be integer 1 or 3
- **`othersurf`** (`curve`) – The second curve, must be a curve

Returns Whether or not the selected edges match

Return type bool

`write`(*filename*, *endian='='*)
Write curve to binary file

Method writes the curve to a binary file. Input arguments include the desired filename (required) and the byte ordering of the file (`'='` native, `'>'` big endian, `'<'` little endian; default is native)

Parameters

- **`filename`** (*`str`*) – Filename for output
- **`endian`** (*`str`*) – Byte ordering of output (optional, default is native)

Returns None

The `friction` Class

The `friction` class is the parent class of all frictional interfaces. Information on setting load perturbations is provided here, as this is common to all friction laws.

class `fdfault.`**`friction`**(*ndim*, *index*, *direction*, *bm*, *bp*)

Class representing a frictionless interface between blocks

This is the parent class of all other frictional interfaces. The `friction` class describes frictionless interfaces. While this interface type does not require any parameter specifications, it does calculate slip from traction and thus the interface tractions are relevant. Therefore, it allows for the user to specify interface tractions that are added to the stress changes calculated by the code. These tractions can be set either as "perturbations" (tractions following some pre-specified mathematical form), or "load files" where the tractions are set point-by-point and thus can be arbitrarily complex.

Frictionless interfaces have the following attributes:

Variables

- **`ndim`** – Number of dimensions in problem (2 or 3)
- **`iftype`** – Type of interface ('locked' for all standard interfaces)
- **`index`** – index of interface (used for identification purposes only, order is irrelevant in simulation)
- **`bm`** – Indices of block in the "minus" direction (tuple of 3 integers)
- **`bp`** – Indices of block in the "plus" direction (tuple of 3 integers)
- **`direction`** – Normal direction in computational space ("x", "y", or "z")
- **`nloads`** – Number of load perturbations (length of `loads` list)
- **`loads`** – List of load perturbations
- **`lf`** – Loadfile holding traction at each point

`__init__`(*ndim*, *index*, *direction*, *bm*, *bp*)

Initializes an instance of the `friction` class

Create a new `friction` given an index, direction, and block coordinates.

Parameters

- **`ndim`** (*`int`*) – Number of spatial dimensions (must be 2 or 3)
- **`index`** (*`int`*) – Interface index, used for bookkeeping purposes, must be nonnegative

- **`direction`** (*`str`*) – String indicating normal direction of interface in computational space, must be `'x'`, `'y'`, or `'z'`, with `'z'` only allowed for 3D problems)
- **`bm`** (*`tuple`*) – Coordinates of block in minus direction (tuple of length 3 of integers)
- **`bp`** (*`tuple`*) – Coordinates of block in plus direction (tuple of length 3 or integers, must differ from `bm` by 1 only along the given direction to ensure blocks are neighboring one another)

Returns New instance of friction class

Return type *friction*

`add_load`(*newload*)

Adds a load to list of load perturbations

Method adds the load provided to the list of load perturbations. If the `newload` parameter is not a load perturbation, this will result in an error.

Parameters **`newload`** (`fdfault.load`) – New load to be added to the interface (must have type `load`)

Returns None

`add_pert`(*newpert*)

Add new friction parameter perturbation to an interface

Method adds a frictional parameter perturbation to an interface. `newpert` must be a parameter perturbation of the correct kind for the given interface type (i.e. if the interface is of type `slipweak`, then `newpert` must have type `swparam`).

Parameters **`newpert`** (*`pert (more precisely, one of the derived classes of friction parameter perturbations)`*) – New perturbation to be added. Must have a type that matches the interface(s) in question.

Returns None

`delete_load`(*index=-1*)

Deletes load at position index from the list of loads

Method deletes the load from the list of loads at position `index`. Default is most recently added load if an index is not provided. `index` must be a valid index into the list of loads.

Parameters **`index`** (*`int`*) – Position within load list to remove (optional, default is -1)

Returns None

delete_loadfile()

Deletes the loadfile for the interface.

Returns None

delete_paramfile()

Deletes friction parameter file for the interface

Removes the friction parameter file for the interface. The interface must be a frictional interface that can accept parameter files.

Returns None

delete_pert(*index=-1*)

Deletes frictional parameter perturbation from interface

`index` is an integer that indicates the position within the list of perturbations. Default is most recently added (-1).

Parameters **index** (*int*) – Index within perturbation list of the given interface to remove. Default is last item (-1, or most recently added)

Returns None

get_bm()

Returns block on negative side

Returns tuple of block indices on negative size

Returns Block indices on negative side (tuple of integers)

Return type tuple

get_bp()

Returns block on positive side

Returns tuple of block indices on positive size

Returns Block indices on positive side (tuple of integers)

Return type tuple

get_direction()

Returns interface orientation

Returns orientation (string indicating normal direction in computational space).

Returns Interface orientation in computational space ('x', 'y', or 'z')

Return type str

get_index()

Returns index

Returns the numerical index corresponding to the interface in question. Note that this is just for bookkeeping purposes, the interfaces may be arranged in any order as long as no index is repeated. The code will automatically handle the indices, so this is typically not modified in any way.

Returns Interface index

Return type int

get_load(*index=None*)

Returns load at position index

Returns a load from the list of load perturbations at position `index`. If no index is provided (or `None` is given), the method returns entire list. `index` must be a valid list index given the number of loads.

Parameters **index** (`int or None`) – Index within load list (optional, default is `None` to return full list)

Returns load or list

get_loadfile()

Returns loadfile for interface

Loadfile sets any surface tractions set for the interface. Note that these tractions are added to any any set by the constant initial stress tensor, initial heterogeneous stress, or interface traction perturbations

Returns Current loadfile for the interface (if the interface does not have a loadfile, returns None)

Return type loadfile or None

get_nloads()

Returns number of load perturbations on the interface

Method returns the number of load perturbations presently in the list of loads.

Returns Number of load perturbations

Return type int

get_nperts()

Returns number of friction parameter perturbations on interface

Method returns the number of parameter perturbations for the list

Returns Number of parameter perturbations

Return type int

get_paramfile()

Returns paramfile (holds arrays of heterogeneous friction parameters) for interface. Can return a subtype of paramfile corresponding to any of the specific friction law types.

Returns paramfile

`get_pert`(*index=None*)

Returns perturbation at position index

Method returns a perturbation from the interface. `index` is the index into the perturbation list for the particular index. If `index` is not provided or is `None`, the method returns the entire list.

Parameters **`index`** (*`int or None`*) – Index into the perturbation list for the index in question (optional, if not provided or `None`, then returns entire list)

Returns pert or list

`get_type`()

Returns string of interface type

Returns the type of the given interface ("locked", "frictionless", "slipweak", or "stz")

Returns Interface type

Return type str

`set_index`(*index*)

Sets interface index

Changes value of interface index. New index must be a nonnegative integer

Parameters **`index`** (*`int`*) – New value of index (nonnegative integer)

Returns None

`set_loadfile`(*newloadfile*)

Sets loadfile for interface

`newloadfile` is the new loadfile (must have type `loadfile`). If the index is bad or the loadfile type is not correct, the code will raise an error. Errors can also result if the shape of the loadfile does not match with the interface.

Parameters **`newloadfile`** (`loadfile`) – New loadfile to be used for the given interface

Returns None

`set_paramfile`(*newparamfile*)

Sets paramfile for the interface

Method sets the file holding frictional parameters for the interface.

`newparamfile` must be a parameter perturbation file of the correct type for the given interface type (i.e. if the interface is of type `slipweak`, then `newpert` must have type `swparamfile`). Errors can also result if the shape of the paramfile does not match with the interface.

Parameters **`newparamfile`** (*`paramfile (actual type must be the appropriate subclass for the friction law of the particular interface and have the right shape)`*) – New frictional parameter file (type depends on interface in question)

Returns None

`write_input` (*f*, *probname*, *directory*, *endian='='*)

Writes interface details to input file

This routine is called for every interface when writing problem data to file. It writes the appropriate section for the interface in the input file. It also writes any necessary binary files holding interface loads, parameters, or state variables.

Parameters

- **`f`** (*`file`*) – File handle for input file
- **`probname`** (*`str`*) – problem name (used for naming binary files)
- **`directory`** (*`str`*) – Directory for output
- **`endian`** (*`str`*) – Byte ordering for binary files (`'<'` little endian, `'>'` big endian, `'='` native, default is native)

Returns None

class `fdfault.`**`load`**(*perttype='constant'*, *t0=0.0*, *x0=0.0*, *dx=0.0*, *y0=0.0*, *dy=0.0*, *sn=0.0*, *s2=0.0*, *s3=0.0*)

Class representing load perturbations to frictional interfaces

The `load` class represents interface traction perturbations that can be expressed in a simple functional form. The `load` class holds information on the shape of the perturbation and the three traction components to be applied to the interface.

Perturbations have the following attributes:

Variables

- **`perttype`** – String describing perturbation shape. See available types below.
- **`t0`** – Perturbation onset time (linear ramp function that attains its maximum at t0; `t0 = 0.` means perturbation is on at all times)
- **`x0`** – Perturbation location along first spatial dimension (see below for details)
- **`dx`** – Perturbation scale along first spatial dimension (see below for details)
- **`y0`** – Perturbation location along second spatial dimension (see below for details)

- **dy** – Perturbation scale along second spatial dimension (see below for details)
- **sn** – Normal traction perturbation
- **s2** – In-plane shear traction perturbation
- **s3** – Out of plane shear traction perturbation

By default, all time, shape, and load parameters are set to zero.

There are several available types of perturbations:

- `'constant'` – A spatially uniform perturbation. All spatial information is ignored
- `'boxcar'` – Perturbation is constant within a rectangle centered at `(x0,y0)` with a half width of `(dx,dy)` in each spatial dimension
- `'ellipse'` – Perturbation is constant within an ellipse centered at `(x0,y0)` with half axis lengths of `(x0,y0)`
- `'gaussian'` – Perturbation follows a Gaussian function centered at `(x0,y0)` with standard deviations `(dx,dy)` in each spatial dimension
- `'linear'` – Perturbation is a linear function with intercept `x0` and slope `1/dx` in the first spatial dimension and intercept `y0` and slope `1/dy` in the second spatial dimension. If either `dx` or `dy` is zero, the linear function is constant in that particular spatial dimension (i.e. set `dy = 0.` if you want to have a function that is only linear in the first spatial dimension)

The shape variables are only interpreted literally for rectangular blocks. If the block is not rectangular, then the shape variables are interpreted as if the block on the negative side were rectangular with the dimensions that are provided when setting up the problem. For example, if you run a problem with a dipping fault that has a trapezoidally shaped block on the minus side of the fault, then `x0` and `dx` would be measured in terms of depth rather than distance along the interface, since the “rectangular” version of the block would have depth along the fault dimension.

If you are in doubt regarding how a perturbation will be interpreted for a particular geometry, it is usually less ambiguous to use a file to set values, as they explicitly set the value at each grid point. However, for some simple forms, perturbations can be more convenient as they use less memory and do not require loading information in parallel from external files.

The different traction components may not correspond to unique coordinate directions for the interface. The code handles complex boundary conditions by rotating the fields into a coordinate system defined by three mutually orthogonal unit vectors. The normal direction is defined to always point into the “positive” block and is uniquely defined by the boundary geometry. The two tangential components are defined as follows for each different type of interface:

- Depending on the orientation of the interface in the computational space, a different convention is used to set the first tangent vector. For `'x'` or `'y'` oriented interfaces, the z component of the first tangent vector is set to zero. This is done to ensure that for 2D problems, the second tangent vector points in the z-direction. For `'z'` oriented interfaces, the y component of the first tangent vector is set to zero.

- With one component of the first tangent vector defined, the other two components can be uniquely determined to make the tangent vector orthogonal up to a sign. The sign is chosen such that the tangent vector points in the direction where the grid points are increasing.
- The second tangent vector is defined by taking the right-handed cross product of the normal and first tangent vectors, except for `'y'` interfaces, where the left-handed cross product is used. This is done to ensure that for 2D problems, the vertical component always points in the $+z$-direction.

The consequence of this is that the letter used to designate the desired component is only valid for rectangular geometries. For non-rectangular geometries, the components will be rotated into the coordinate system described above. For interfaces in the "x" direction (i.e. connecting blocks whose indices only differ in the x-direction), the y component of output units will be along the first tangent vector, and the z component will be along the second tangent vector. Similarly, for "y" interfaces the x component is set by the first tangent vector and the z component is determined by the second tangent vector, and for "z" interfaces the first tangent vector is in the x-direction and the second tangent vector corresponds to the y-direction.

`__init__`(*perttype='constant'*, *t0=0.0*, *x0=0.0*, *dx=0.0*, *y0=0.0*, *dy=0.0*, *sn=0.0*, *s2=0.0*, *s3=0.0*)

Initialize a new instance of an interface load perturbation

Method creates a new instance of a load perturbation. It calls the superclass routine to initialize the spatial and temporal details of the perturbation, and creates the variables holding the interface traction details. Default values are provided for all arguments (all zeros, with a perttype of `'constant'`).

Parameters

- **`perttype`** (*`str`*) – Perturbation type (string, default is `'constant'`)
- **`t0`** (*`float`*) – Linear ramp time scale (default 0.)
- **`x0`** (*`float`*) – Perturbation location along first interface dimension (default 0.)
- **`dx`** (*`float`*) – Perturbation scale along first interface dimension (default 0.)
- **`y0`** (*`float`*) – Perturbation location along second interface dimension (default 0.)
- **`dy`** (*`float`*) – Perturbation scale along second interface dimension (default 0.)

- **sn** (*float*) – Interface normal traction perturbation (negative in compression, default 0.)
- **s2** (*float*) – Interface horizontal shear traction perturbation (default 0.)
- **s3** (*float*) – Interface vertical shear traction perturbation (default 0.)

Returns New instance of perturbation

Return type *fdfault.load*

get_dx()

Returns perturbation scale along first interface coordinate

Returns Scale of perturbation along first interface coordinate

Return type float

get_dy()

Returns perturbation scale along second interface coordinate

Returns Scale of perturbation along second interface coordinate

Return type float

get_s2()

Returns in plane shear stress perturbation

Returns In plane stress perturbation

Return type float

get_s3()

Returns out of plane shear stress perturbation

Returns Out of plane shear stress perturbation

Return type float

get_sn()

Returns normal stress perturbation

Returns Normal stress perturbation

Return type float

get_t0()

Returns onset time

Returns Perturbation onset time

Return type float

get_type()

Returns perturbation type

Returns Perturbation type

Return type str

get_x0()

Returns perturbation location in first interface coordinate

Returns Location of perturbation along first interface coordinate

Return type float

get_y0()

Returns perturbation location in second interface coordinate

Returns Location of perturbation along second interface coordinate

Return type float

set_dx(*dx*)

Sets first coordinate of perturbation scale

Changes value of perturbation scale for first coordinate direction. New value must be nonnegative.

Parameters **dx** (*float*) – New value of perturbation scale along second coordinate

Returns None

set_dy(*dy*)

Sets second coordinate of perturbation scale

Changes value of perturbation scale for second coordinate direction. New value must be nonnegative.

Parameters **dy** (*float*) – New value of perturbation scale along second coordinate

Returns None

set_s2(*s2*)

Sets in-plane shear stress perturbation

Parameters **s2** (*float*) – New value of in plane shear stress perturbation

Returns None

set_s3(*s3*)

Sets out of plane shear stress perturbation

Parameters **s3** (*float*) – New value of out of plane shear stress perturbation

Returns None

set_sn(*sn*)
Sets normal stress perturbation

Parameters **sn** (*float*) – New value of normal stress perturbation

Returns None

set_t0(*t0*)
Sets onset time

Changes value of onset time. New value must be nonnegative.

Parameters **t0** (*float*) – New value of onset time

Returns None

set_type(*perttype*)
Sets perturbation type

Resets the perturbation type to `perttype`. Note that the new type must be among the valid perturbation types.

Parameters **perttype** (*str*) – New value for perttype, must be a valid perturbation type

Returns None

set_x0(*x0*)
Sets first coordinate of perturbation location

Parameters **x0** (*float*) – New value of perturbation location along first coordinate

Returns None

set_y0(*y0*)
Sets second coordinate of perturbation location

Parameters **x0** (*float*) – New value of perturbation location along second coordinate

Returns None

write_input(*f*)
Writes perturbation to input file

Method writes perturbation to input file (input file provided as input)

Parameters **f** (*file*) – Output file to which the perturbation will be written

Returns none

class `fdfault.`**loadfile**(*n1*, *n2*, *sn*, *s2*, *s3*)
The `loadfile` class is a class for loading interface tractions to simulation from file. It

includes arrays for normal and two components of shear tractions to be applied at the specific boundary.

All `loadfile` members contain the following internal parameters:

Variables

- **`n1`** – Number of grid points along first coordinate direction
- **`n2`** – Number of grid points along the second coordinate direction
- **`sn`** – Normal stress perturbation (must be an `(n1,n2)` shaped numpy array)
- **`s2`** – In plane shear stress perturbation (must be an `(n1,n2)` shaped numpy array)
- **`s3`** – Out of plane shear stress perturbation (must be an `(n1,n2)` shaped numpy array)

`loadfile` instances hold three numpy arrays with shape `(n1,n2)` for the normal and two shear tractions actin on the interface. Load files do not include any information about the shape of the boundary, and it is up to the user to ensure that that the parameter values correspond to the coordinates of the interface. However, because parameter files explicitly assign a value to each grid point, there is less ambiguity regarding the final values when compared to perturbations. Depending on the orientation of the interface, the two coordinate directions will have different orientations in space. The first coordinate direction is the x direction for y and z interfaces (for x interfaces, the first index is in the y direction), and the second coordinate is in the z direction except for z interfaces, where y is the second index}.

The different traction components may not correspond to unique coordinate directions for the interface. The code handles complex boundary conditions by rotating the fields into a coordinate system defined by three mutually orthogonal unit vectors. The normal direction is defined to always point into the “positive” block and is uniquely defined by the boundary geometry. The two tangential components are defined as follows for each different type of interface:

- Depending on the orientation of the interface in the computational space, a different convention is used to set the first tangent vector. For `'x'` or `'y'` oriented interfaces, the z component of the first tangent vector is set to zero. This is done to ensure that for 2D problems, the second tangent vector points in the z-direction. For `'z'` oriented interfaces, the y component of the first tangent vector is set to zero.
- With one component of the first tangent vector defined, the other two components can be uniquely determined to make the tangent vector orthogonal up to a sign. The sign is chosen such that the tangent vector points in the direction where the grid points are increasing.
- The second tangent vector is defined by taking the right-handed cross product of the normal and first tangent vectors, except for `'y'` interfaces, where the left-handed cross

product is used. This is done to ensure that for 2D problems, the vertical component always points in the $+z$-direction.

The consequence of this is that the letter used to designate the desired component is only valid for rectangular geometries. For non-rectangular geometries, the components will be rotated into the coordinate system described above. For interfaces in the "x" direction (i.e. connecting blocks whose indices only differ in the x-direction), the y component of output units will be along the first tangent vector, and the z component will be along the second tangent vector. Similarly, for "y" interfaces the x component is set by the first tangent vector and the z component is determined by the second tangent vector, and for "z" interfaces the first tangent vector is in the x-direction and the second tangent vector corresponds to the y-direction.

When writing `loadfile` instances to disk, the code uses numpy to write information to disk in binary format. Byte-ordering can be specified, and should correspond to the byte-ordering on the system where the simulation will be run (default is native).

`__init__`(*n1*, *n2*, *sn*, *s2*, *s3*)

Initialize a new instance of a loadfile object

Create a new instance of a loadfile, which is a class describing interface boundary traction perturbations in a file. Required information is the number of grid points for the interface and one array for each of the three interface traction component perturbations. All the array shapes must be `(n1,n2)` or the code will raise an error.

Parameters

- **n1** (*int*) – Number of grid points along first coordinate direction
- **n2** (*int*) – Number of grid points along the second coordinate direction
- **sn** (*ndarray*) – Interface normal traction perturbation array (negative in compression)
- **s2** (*ndarray*) – Interface horizontal shear traction perturbation array
- **s3** (*ndarray*) – Interface vertical shear traction perturbation array

Returns New loadfile instance

Return type *loadfile*

`get_n1`()

Returns number of grid points in 1st coordinate direction

Returns Number of grid points in 1st coordinate direction (x, except for x interfaces, where y is the first coordinate direction)

Return type int

`get_n2`()

Returns number of grid points in 2nd coordinate direction

Returns Number of grid points in 2nd coordinate direction (z, except for z interfaces, where y is the second coordinate direction)

Return type int

get_s2 (*index=None*)
Returns in plane shear stress at given indices

Returns in plane shear stress perturbation at the indices given by `index`. If no indices are provided, the method returns the entire array.

Parameters **index** (*float, tuple, or None*) – Index into `s2` array (optional, if not provided returns entire array)

Returns In plane shear stress perturbation (either ndarray or float, depending on value of `index`)

Return type ndarray or float

get_s3 (*index=None*)
Returns out of plane shear stress at given indices

Returns out of plane shear stress perturbation at the indices given by `index`. If no indices are provided, the method returns the entire array.

Parameters **index** (*float, tuple, or None*) – Index into `s3` array (optional, if not provided returns entire array)

Returns Out of plane shear stress perturbation (either ndarray or float, depending on value of `index`)

Return type ndarray or float

get_sn (*index=None*)
Returns normal stress at given indices

Returns normal stress perturbation at the indices given by `index`. If no indices are provided, the method returns the entire array.

Parameters **index** (*float, tuple, or None*) – Index into `sn` array (optional, if not provided returns entire array)

Returns Normal stress perturbation (either ndarray or float, depending on value of `index`)

Return type ndarray or float

write (*filename, endian='='*)
Write perturbation data to file

Parameters

- **filename** (*str*) – Name of binary file to be written

- **endian** (*str*) – Byte-ordering for output. Options inclue `'='` for native, `'<'` for little endian, and `'>'` for big endian. Optional, default is native

Returns None

The `slipweak` Class

The main class used for creating slip weakening interfaces is the `slipweak` class.

class `fdfault.`**`slipweak`**(*ndim*, *index*, *direction*, *bm*, *bp*)

Class representing a slip weakening frictional interface

This class describes slip weakening friction laws. This is a frictional interface with parameter values. Tractions on the interface are set using load perturbations and load files. Parameter values are set using parameter perturbations (the `swparam` class) and parameter files (the `swparamfile` class). Parameters that can be specified include:

- The slip weakening distance d_c, `dc`
- The static friction value μ_s, `mus`
- The dynamic friction value μ_d, `mud`
- The frictional cohesion c_0, `c0`
- The forced rupture time t_{rup}, `trup`
- The characteristic weakening time t_c, `tc`

Slip weakening Frictional interfaces have the following attributes:

Variables

- **ndim** – Number of dimensions in problem (2 or 3)
- **iftype** – Type of interface (‘locked’ for all standard interfaces)
- **index** – index of interface (used for identification purposes only, order is irrelevant in simulation)
- **bm** – Indices of block in the “minus” direction (tuple of 3 integers)
- **bp** – Indices of block in the “plus” direction (tuple of 3 integers)
- **direction** – Normal direction in computational space (“x”, “y”, or “z”)
- **nloads** – Number of load perturbations (length of `loads` list)
- **loads** – List of load perturbations
- **lf** – Loadfile holding traction at each point

- **`nperts`** – Number of parameter perturbations (length of `perts` list)
- **`perts`** – List of parameter perturbations (perturbations must be `swparam` type)
- **`pf`** – Paramfile holding traction at each point

`__init__`(*ndim*, *index*, *direction*, *bm*, *bp*)
Initializes an instance of the `slipweak` class

Create a new `slipweak` given an index, direction, and block coordinates.

Parameters

- **`ndim`** (*`int`*) – Number of spatial dimensions (must be 2 or 3)
- **`index`** (*`int`*) – Interface index, used for bookkeeping purposes, must be nonnegative
- **`direction`** (*`str`*) – String indicating normal direction of interface in computational space, must be `'x'`, `'y'`, or `'z'`, with `'z'` only allowed for 3D problems)
- **`bm`** (*`tuple`*) – Coordinates of block in minus direction (tuple of length 3 of integers)
- **`bp`** (*`tuple`*) – Coordinates of block in plus direction (tuple of length 3 or integers, must differ from `bm` by 1 only along the given direction to ensure blocks are neighboring one another)

Returns New instance of slipweak class

Return type *slipweak*

`add_load`(*newload*)
Adds a load to list of load perturbations

Method adds the load provided to the list of load perturbations. If the `newload` parameter is not a load perturbation, this will result in an error.

Parameters **`newload`** (`fdfault.load`) – New load to be added to the interface (must have type `load`)

Returns None

`add_pert`(*newpert*)
Add new friction parameter perturbation to an interface

Method adds a frictional parameter perturbation to an interface. `newpert` must must have type `swparam`).

Parameters **`newpert`** (`swparam`) – New perturbation to be added

Returns None

delete_load(*index=-1*)

Deletes load at position index from the list of loads

Method deletes the load from the list of loads at position `index`. Default is most recently added load if an index is not provided. `index` must be a valid index into the list of loads.

Parameters **index** (*int*) – Position within load list to remove (optional, default is -1)

Returns None

delete_loadfile()

Deletes the loadfile for the interface.

Returns None

delete_paramfile()

Deletes friction parameter file for the interface

Removes the friction parameter file for the interface. The interface must be a frictional interface that can accept parameter files.

Returns None

delete_pert(*index=-1*)

Deletes frictional parameter perturbation from interface

`index` is an integer that indicates the position within the list of perturbations. Default is most recently added (-1).

Parameters **index** (*int*) – Index within perturbation list of the given interface to remove. Default is last item (-1, or most recently added)

Returns None

get_bm()

Returns block on negative side

Returns tuple of block indices on negative size

Returns Block indices on negative side (tuple of integers)

Return type tuple

get_bp()

Returns block on positive side

Returns tuple of block indices on positive size

Returns Block indices on positive side (tuple of integers)

Return type tuple

get_direction()

Returns interface orientation

Returns orientation (string indicating normal direction in computational space).

Returns Interface orientation in computational space ('x', 'y', or 'z')

Return type str

get_index()

Returns index

Returns the numerical index corresponding to the interface in question. Note that this is just for bookkeeping purposes, the interfaces may be arranged in any order as long as no index is repeated. The code will automatically handle the indices, so this is typically not modified in any way.

Returns Interface index

Return type int

get_load(*index=None*)

Returns load at position index

Returns a load from the list of load perturbations at position `index`. If no index is provided (or `None` is given), the method returns entire list. `index` must be a valid list index given the number of loads.

Parameters **index** (*int or None*) – Index within load list (optional, default is `None` to return full list)

Returns load or list

get_loadfile()

Returns loadfile for interface

Loadfile sets any surface tractions set for the interface. Note that these tractions are added to any any set by the constant initial stress tensor, initial heterogeneous stress, or interface traction perturbations

Returns Current loadfile for the interface (if the interface does not have a loadfile, returns None)

Return type loadfile or None

get_nloads()

Returns number of load perturbations on the interface

Method returns the number of load perturbations presently in the list of loads.

Returns Number of load perturbations

Return type int

get_nperts()

Returns number of friction parameter perturbations on interface

Method returns the number of parameter perturbations for the list

Returns Number of parameter perturbations

Return type int

get_paramfile()

Returns paramfile (holds arrays of heterogeneous friction parameters) for interface. Can return a subtype of paramfile corresponding to any of the specific friction law types.

Returns Paramfile for this interface

Return type paramfile

get_pert(*index=None*)

Returns perturbation at position index

Method returns a perturbation from the interface. `index` is the index into the perturbation list for the particular index. If `index` is not provided or is `None`, the method returns the entire list.

Parameters **index** (*int or None*) – Index into the perturbation list for the index in question (optional, if not provided or `None`, then returns entire list)

Returns pert or list

get_type()

Returns string of interface type

Returns the type of the given interface ("locked", "frictionless", "slipweak", or "stz")

Returns Interface type

Return type str

set_index(*index*)

Sets interface index

Changes value of interface index. New index must be a nonnegative integer

Parameters **index** (*int*) – New value of index (nonnegative integer)

Returns None

set_loadfile(*newloadfile*)

Sets loadfile for interface

`newloadfile` is the new loadfile (must have type `loadfile`). If the index is bad or the loadfile type is not correct, the code will raise an error. Errors can also result if the shape of the loadfile does not match with the interface.

Parameters **`newloadfile`** (`loadfile`) – New loadfile to be used for the given interface

Returns None

`set_paramfile`(*newparamfile*)

Sets paramfile for the interface

Method sets the file holding frictional parameters for the interface.

`newparamfile` must be a parameter perturbation file of type `swparam`. Errors can also result if the shape of the paramfile does not match with the interface.

Parameters **`newparamfile`** (`swparamfile`) – New frictional parameter file

Returns None

`write_input`(*f*, *probname*, *directory*, *endian='='*)

Writes interface details to input file

This routine is called for every interface when writing problem data to file. It writes the appropriate section for the interface in the input file. It also writes any necessary binary files holding interface loads, parameters, or state variables.

Parameters

- **`f`** (*`file`*) – File handle for input file
- **`probname`** (*`str`*) – problem name (used for naming binary files)
- **`directory`** (*`str`*) – Directory for output
- **`endian`** (*`str`*) – Byte ordering for binary files (`'<'` little endian, `'>'` big endian, `'='` native, default is native)

Returns None

class `fdfault.`**`swparam`**(*perttype='constant'*, *t0=0.0*, *x0=0.0*, *dx=0.0*, *y0=0.0*, *dy=0.0*, *dc=0.0*, *mus=0.0*, *mud=0.0*, *c0=0.0*, *trup=0.0*, *tc=0.0*)

Class representing slip weakening parameter perturbations to frictional interfaces

The `swparam` class represents slip weakening parameter perturbations that can be expressed in a simple functional form. The `swparam` class holds information on the shape of the perturbation and the six parameter values for the interface.

Perturbations have the following attributes:

Variables

- **`perttype`** – String describing perturbation shape. See available types below.
- **`t0`** – Perturbation onset time (linear ramp function that attains its maximum at t0; `t0 = 0.` means perturbation is on at all times)

- **x0** – Perturbation location along first spatial dimension (see below for details)
- **dx** – Perturbation scale along first spatial dimension (see below for details)
- **y0** – Perturbation location along second spatial dimension (see below for details)
- **dy** – Perturbation scale along second spatial dimension (see below for details)
- **dc** – Slip weakening distance perturbation
- **mus** – Static friction coefficient perturbation
- **mud** – Dynamic friction coefficient perturbation
- **c0** – Frictional Cohesion perturbation
- **trup** – Rupture time perturbation
- **tc** – Characteristic weakening time perturbation

By default, all time, shape, and friction parameters are set to zero.

There are several available types of perturbations:

- `'constant'` – A spatially uniform perturbation. All spatial information is ignored
- `'boxcar'` – Perturbation is constant within a rectangle centered at `(x0,y0)` with a half width of `(dx,dy)` in each spatial dimension
- `'ellipse'` – Perturbation is constant within an ellipse centered at `(x0,y0)` with half axis lengths of `(x0,y0)`
- `'gaussian'` – Perturbation follows a Gaussian function centered at `(x0,y0)` with standard deviations `(dx,dy)` in each spatial dimension
- `'linear'` – Perturbation is a linear function with intercept `x0` and slope `1/dx` in the first spatial dimension and intercept `y0` and slope `1/dy` in the second spatial dimension. If either `dx` or `dy` is zero, the linear function is constant in that particular spatial dimension (i.e. set `dy = 0.` if you want to have a function that is only linear in the first spatial dimension)

The shape variables are only interpreted literally for rectangular blocks. If the block is not rectangular, then the shape variables are interpreted as if the block on the negative side were rectangular with the dimensions that are provided when setting up the problem. For example, if you run a problem with a dipping fault that has a trapezoidally shaped block on the minus side of the fault, then `x0` and `dx` would be measured in terms of depth rather than distance along the interface, since the "rectangular" version of the block would have depth along the fault dimension.

If you are in doubt regarding how a perturbation will be interpreted for a particular geometry, it is usually less ambiguous to use a file to set values, as they explicitly set the value at each grid point. However, for some simple forms, perturbations can be more convenient as they use less memory and do not require loading information in parallel from external files.

`__init__`(*perttype='constant'*, *t0=0.0*, *x0=0.0*, *dx=0.0*, *y0=0.0*, *dy=0.0*, *dc=0.0*, *mus=0.0*, *mud=0.0*, *c0=0.0*, *trup=0.0*, *tc=0.0*)

Initialize a new instance of an slip weakening parameter perturbation

Method creates a new instance of a slip weakening parameter perturbation. It calls the superclass routine to initialize the spatial and temporal details of the perturbation, and creates the variables holding the parameter values specific to the slip weakening law. Default values are provided for all arguments (all zeros, with a perttype of `'constant'`).

Parameters

- **`perttype`** (*`str`*) – Perturbation type (string, default is `'constant'`)
- **`t0`** (*`float`*) – Linear ramp time scale (default 0.)
- **`x0`** (*`float`*) – Perturbation location along first interface dimension (default 0.)
- **`dx`** (*`float`*) – Perturbation scale along first interface dimension (default 0.)
- **`y0`** (*`float`*) – Perturbation location along second interface dimension (default 0.)
- **`dy`** (*`float`*) – Perturbation scale along second interface dimension (default 0.)
- **`dc`** (*`float`*) – Slip weakening distance perturbation (negative in compression, default 0.)
- **`mus`** (*`float`*) – Static friction coefficient perturbation (default 0.)
- **`mud`** (*`float`*) – Dynamic friction coefficient perturbation (default 0.)
- **`c0`** (*`float`*) – Frictional cohesion perturbation (negative in compression, default 0.)
- **`trup`** (*`float`*) – Forced rupture time perturbation (default 0.)
- **`tc`** (*`float`*) – Characteristic weakening time perturbation (default 0.)

Returns New instance of slip weakening parameter perturbation

Return type *swparam*

`get_c0`()

Returns frictional cohesion perturbation

Returns Frictional cohesion perturbation

Return type float

get_dc()
Returns slip weakening distance perturbation

Returns Slip weakening distance perturbation

Return type float

get_dx()
Returns perturbation scale along first interface coordinate

Returns Scale of perturbation along first interface coordinate

Return type float

get_dy()
Returns perturbation scale along second interface coordinate

Returns Scale of perturbation along second interface coordinate

Return type float

get_mud()
Returns dynamic friction coefficient perturbation

Returns Dynamic friction coefficient perturbation

Return type float

get_mus()
Returns static friction coefficient perturbation

Returns Static friction coefficient perturbation

Return type float

get_t0()
Returns onset time

Returns Perturbation onset time

Return type float

get_tc()
Returns characteristic weakening time perturbation

Returns Characteristic weakening time perturbation

Return type float

get_trup()
Returns forced rupture time perturbation

Returns Forced rupture time perturbation

Return type float

get_type()

Returns perturbation type

Returns Perturbation type

Return type str

get_x0()

Returns perturbation location in first interface coordinate

Returns Location of perturbation along first interface coordinate

Return type float

get_y0()

Returns perturbation location in second interface coordinate

Returns Location of perturbation along second interface coordinate

Return type float

set_c0(*c0*)

Sets frictional cohesion perturbation

Parameters **c0** (*float*) – New value of frictional cohesion perturbation

Returns None

set_dc(*dc*)

Sets slip weakening distance perturbation

Parameters **dc** (*float*) – New value of slip weakening distance perturbation

Returns None

set_dx(*dx*)

Sets first coordinate of perturbation scale

Changes value of perturbation scale for first coordinate direction. New value must be nonnegative.

Parameters **dx** (*float*) – New value of perturbation scale along second coordinate

Returns None

set_dy(*dy*)

Sets second coordinate of perturbation scale

Changes value of perturbation scale for second coordinate direction. New value must be nonnegative.

Parameters **`dy`** (*`float`*) – New value of perturbation scale along second coordinate

Returns None

`set_mud`(*mud*)
Sets dynamic friction coefficient perturbation

Parameters **`mud`** (*`float`*) – New value of dynamic friction coefficient perturbation

Returns None

`set_mus`(*mus*)
Sets static friction coefficient perturbation

Parameters **`mus`** (*`float`*) – New value of static friction coefficient perturbation

Returns None

`set_t0`(*t0*)
Sets onset time

Changes value of onset time. New value must be nonnegative.

Parameters **`t0`** (*`float`*) – New value of onset time

Returns None

`set_tc`(*tc*)
Sets characteristic weakening time perturbation

Parameters **`tc`** (*`float`*) – New value of characteristic weakening time perturbation

Returns None

`set_trup`(*trup*)
Sets forced rupture time perturbation

Parameters **`trup`** (*`float`*) – New value of forced rupture time perturbation

Returns None

`set_type`(*perttype*)
Sets perturbation type

Resets the perturbation type to `perttype`. Note that the new type must be among the valid perturbation types.

Parameters **`perttype`** (*`str`*) – New value for perttype, must be a valid perturbation type

Returns None

set_x0 (*x0*)

Sets first coordinate of perturbation location

Parameters **x0** (*float*) – New value of perturbation location along first coordinate

Returns None

set_y0 (*y0*)

Sets second coordinate of perturbation location

Parameters **x0** (*float*) – New value of perturbation location along second coordinate

Returns None

write_input (*f*)

Writes perturbation to input file

Method writes perturbation to input file (input file provided as input)

Parameters **f** (*file*) – Output file to which the perturbation will be written

Returns none

class `fdfault.`**swparamfile** (*n1*, *n2*, *dc*, *mus*, *mud*, *c0*, *trup*, *tc*)

The `swparamfile` class is a class for loading heterogeneous friction parameter values from file. It is only used for slip weakening interfaces.

All `swparamfile` instances contain the following internal parameters:

Variables

- **n1** – Number of grid points along first coordinate direction
- **n2** – Number of grid points along the second coordinate direction
- **dc** – Array holding slip weakening distance perturbation (numpy array with shape `(n1,n2)`)
- **mus** – Array holding static friction coefficient perturbation (numpy array with shape `(n1,n2)`)
- **mud** – Array holding dynamic friction coefficient perturbation (numpy array with shape `(n1,n2)`)
- **c0** – Array holding frictional cohesion perturbation (numpy array with shape `(n1,n2)`)
- **trup** – Array holding forced rupture time perturbation (numpy array with shape `(n1,n2)`)
- **tc** – Array holding characteristic failure time perturbation (numpy array with shape `(n1,n2)`)

`swparamfile` will also define six numpy array with shape `(n1,n2)` holding the various friction parameters (slip weakening distance, static friction coefficient, dynamic friction coefficient, frictional cohesion, forced rupture time, and characteristic failure time. Slip weakening parameter files do not include any information about the shape of the boundary, and it is up to the user to ensure that that the parameter values correspond to the coordinates of the interface. However, because parameter files explicitly assign a value to each grid point, there is less ambiguity regarding the final values when compared to perturbations. Depending on the orientation of the interface, the two coordinate directions will have different orientations in space. The first coordinate direction is the x direction for y and z interfaces (for x interfaces, the first index is in the y direction), and the second coordinate is in the z direction except for z interfaces, where y is the second index}.

When writing `swparamfile` instances to disk, the code uses numpy to write information to disk in binary format. Byte-ordering can be specified, and should correspond to the byte-ordering on the system where the simulation will be run (default is native).

`__init__`(*n1*, *n2*, *dc*, *mus*, *mud*, *c0*, *trup*, *tc*)
Initialize a new instance of a swparamfile object

Create a new instance of a swparamfile, which is a class describing slip weakening parameter perturbations in a file. Required information is the number of grid points for the interface and one array for each of the six parameter perturbations. All the array shapes must be `(n1,n2)` or the code will raise an error.

Parameters

- **`n1`** (*int*) – Number of grid points along first coordinate direction
- **`n2`** (*int*) – Number of grid points along the second coordinate direction
- **`dc`** (*ndarray*) – Slip weakening distance perturbation array
- **`mus`** (*ndarray*) – Static friction coefficient perturbation array
- **`mud`** (*ndarray*) – Dynamic friction coefficient perturbation array
- **`c0`** (*ndarray*) – Frictional cohesion perturbation array
- **`trup`** (*ndarray*) – Forced rupture time perturbation array
- **`tc`** (*ndarray*) – Characteristic weakening time perturbation array

Returns New swparamfile instance

Return type *swparamfile*

`get_c0`(*index=None*)
Returns frictional cohesion at given indices

Returns frictional cohesion perturbation at the indices given by `index`. If no indices are provided, the method returns the entire array.

Parameters **`index`** (*`float, tuple, or None`*) – Index into frictional cohesion array (optional, if not provided returns entire array)

Returns Frictional cohesion perturbation (either ndarray or float, depending on value of `index`)

Return type ndarray or float

`get_dc`(*index=None*)

Returns slip weakening distance at given indices

Returns slip weakening distance perturbation at the indices given by `index`. If no indices are provided, the method returns the entire array.

Parameters **`index`** (*`float, tuple, or None`*) – Index into slip weakening distance array (optional, if not provided returns entire array)

Returns Slip weakening distance perturbation (either ndarray or float, depending on value of `index`)

Return type ndarray or float

`get_mud`(*index=None*)

Returns dynamic friction coefficient at given indices

Returns dynamic friction coefficient perturbation at the indices given by `index`. If no indices are provided, the method returns the entire array.

Parameters **`index`** (*`float, tuple, or None`*) – Index into dynamic friction coefficient array (optional, if not provided returns entire array)

Returns Dynamic friction coefficient perturbation (either ndarray or float, depending on value of `index`)

Return type ndarray or float

`get_mus`(*index=None*)

Returns static friction coefficient at given indices

Returns static friction coefficient perturbation at the indices given by `index`. If no indices are provided, the method returns the entire array.

Parameters **`index`** (*`float, tuple, or None`*) – Index into static friction coefficient array (optional, if not provided returns entire array)

Returns Static friction coefficient perturbation (either ndarray or float, depending on value of `index`)

Return type ndarray or float

`get_n1`()

Returns number of grid points in 1st coordinate direction

Returns Number of grid points in 1st coordinate direction (x, except for x interfaces, where y is the first coordinate direction)

Return type int

get_n2()

Returns number of grid points in 2nd coordinate direction

Returns Number of grid points in 2nd coordinate direction (z, except for z interfaces, where y is the second coordinate direction)

Return type int

get_tc(*index=None*)

Returns characteristic failure time at given indices

Returns characteristic failure time perturbation at the indices given by `index`. If no indices are provided, the method returns the entire array.

Parameters **index** (*float, tuple, or None*) – Index into characteristic failure time array (optional, if not provided returns entire array)

Returns Characteristic failure time perturbation (either ndarray or float, depending on value of `index`)

Return type ndarray or float

get_trup(*index=None*)

Returns force rupture time at given indices

Returns forced rupture time perturbation at the indices given by `index`. If no indices are provided, the method returns the entire array.

Parameters **index** (*float, tuple, or None*) – Index into forced rupture time array (optional, if not provided returns entire array)

Returns Static forced rupture time perturbation (either ndarray or float, depending on value of `index`)

Return type ndarray or float

write(*filename*, *endian='='*)

Write perturbation data to file

Parameters

- **filename** (*str*) – Name of binary file to be written
- **endian** (*str*) – Byte-ordering for output. Options inclue `'='` for native, `'<'` for little endian, and `'>'` for big endian. Optional, default is native

Returns None

The `stz` Class

The main class used for creating STZ interfaces is the `stz` class. This also includes setting the STZ parameters through perturbations and files, as well as setting the initial state variable.

class `fdfault.`**`stz`**(*ndim*, *index*, *direction*, *bm*, *bp*)

Class representing a Shear Transformation Zone (STZ) Theory Frictional Interface

STZ Frictional Interfaces are an interface with a state variable, in this case an effective temperature. The interface also requires setting the interface tractions and parameter values in addition to the initial value of the state variable. All of these can be set using some combination of perturbations and files. Parameters include:

- Reference velocity V_0 , `v0`
- Reference activation barrier f_0, `f0`
- Frictional direct effect a, `a`
- Frictional yield coefficient μ_y, `muy`
- Effective temperature specific heat c_0, `c0`
- Effective temperature relaxation rate R, `R`
- Effective temperature relaxation barrier β, `beta`
- Effective temperature activation barrier χ_w, `chiw`
- Reference velocity for effective temperature activation V_1, `v1`

STZ Frictional interfaces have the following attributes:

Variables

- **`ndim`** – Number of dimensions in problem (2 or 3)
- **`iftype`** – Type of interface (‘locked’ for all standard interfaces)
- **`index`** – index of interface (used for identification purposes only, order is irrelevant in simulation)
- **`bm`** – Indices of block in the “minus” direction (tuple of 3 integers)
- **`bp`** – Indices of block in the “plus” direction (tuple of 3 integers)
- **`direction`** – Normal direction in computational space (“x”, “y”, or “z”)
- **`nloads`** – Number of load perturbations (length of `loads` list)
- **`loads`** – List of load perturbations
- **`lf`** – Loadfile holding traction at each point
- **`nperts`** – Number of parameter perturbations (length of `perts` list)

- **perts** – List of parameter perturbations (each must be `stzparam`)
- **pf** – Paramfile holding traction at each point
- **state** – Initial value of state variable
- **sf** – Statefile holding heterogeneous initial state variable values

`__init__`(*ndim*, *index*, *direction*, *bm*, *bp*)
Initializes an instance of the `stz` class

Create a new `stz` given an index, direction, and block coordinates.

Parameters

- **ndim** (*int*) – Number of spatial dimensions (must be 2 or 3)
- **index** (*int*) – Interface index, used for bookkeeping purposes, must be nonnegative
- **direction** (*str*) – String indicating normal direction of interface in computational space, must be `'x'`, `'y'`, or `'z'`, with `'z'` only allowed for 3D problems)
- **bm** (*tuple*) – Coordinates of block in minus direction (tuple of length 3 of integers)
- **bp** (*tuple*) – Coordinates of block in plus direction (tuple of length 3 or integers, must differ from `bm` by 1 only along the given direction to ensure blocks are neighboring one another)

Returns New instance of stz class

Return type *stz*

`add_load`(*newload*)
Adds a load to list of load perturbations

Method adds the load provided to the list of load perturbations. If the `newload` parameter is not a load perturbation, this will result in an error.

Parameters **newload** (`fdfault.load`) – New load to be added to the interface (must have type `load`)

Returns None

`add_pert`(*newpert*)
Add new friction parameter perturbation to an interface

Method adds a frictional parameter perturbation to an interface. `newpert` must must have type `stzparam`).

Parameters **newpert** (`stzparam`) – New perturbation to be added

Returns None

`delete_load``(`*index=-1*`)`
Deletes load at position index from the list of loads

Method deletes the load from the list of loads at position `index`. Default is most recently added load if an index is not provided. `index` must be a valid index into the list of loads.

Parameters **`index`** (*`int`*) – Position within load list to remove (optional, default is -1)

Returns None

`delete_loadfile``()`
Deletes the loadfile for the interface.

Returns None

`delete_paramfile``()`
Deletes friction parameter file for the interface

Removes the friction parameter file for the interface. The interface must be a frictional interface that can accept parameter files.

Returns None

`delete_pert``(`*index=-1*`)`
Deletes frictional parameter perturbation from interface

`index` is an integer that indicates the position within the list of perturbations. Default is most recently added (-1).

Parameters **`index`** (*`int`*) – Index within perturbation list of the given interface to remove. Default is last item (-1, or most recently added)

Returns None

`delete_statefile``()`
Deletes statefile for the interface

Delete the statefile for the interface. Will set the statefile attribute for the interface to None.

Returns None

`get_bm``()`
Returns block on negative side

Returns tuple of block indices on negative size

Returns Block indices on negative side (tuple of integers)

Return type tuple

`get_bp``()`
Returns block on positive side

Returns tuple of block indices on positive size

Returns Block indices on positive side (tuple of integers)

Return type tuple

`get_direction``()`

Returns interface orientation

Returns orientation (string indicating normal direction in computational space).

Returns Interface orientation in computational space ('x', 'y', or 'z')

Return type str

`get_index``()`

Returns index

Returns the numerical index corresponding to the interface in question. Note that this is just for bookkeeping purposes, the interfaces may be arranged in any order as long as no index is repeated. The code will automatically handle the indices, so this is typically not modified in any way.

Returns Interface index

Return type int

`get_load``(`*index=None*`)`

Returns load at position index

Returns a load from the list of load perturbations at position `index`. If no index is provided (or `None` is given), the method returns entire list. `index` must be a valid list index given the number of loads.

Parameters **`index`** (*`int or None`*) – Index within load list (optional, default is `None` to return full list)

Returns load or list

`get_loadfile``()`

Returns loadfile for interface

Loadfile sets any surface tractions set for the interface. Note that these tractions are added to any any set by the constant initial stress tensor, initial heterogeneous stress, or interface traction perturbations

Returns Current loadfile for the interface (if the interface does not have a loadfile, returns None)

Return type loadfile or None

`get_nloads``()`

Returns number of load perturbations on the interface

Method returns the number of load perturbations presently in the list of loads.

Returns Number of load perturbations

Return type int

get_nperts()

Returns number of friction parameter perturbations on interface

Method returns the number of parameter perturbations for the list

Returns Number of parameter perturbations

Return type int

get_paramfile()

Returns paramfile (holds arrays of heterogeneous friction parameters) for interface. Can return a subtype of paramfile corresponding to any of the specific friction law types.

Returns Paramfile for this interface

Return type paramfile

get_pert(*index=None*)

Returns perturbation at position index

Method returns a perturbation from the interface. `index` is the index into the perturbation list for the particular index. If `index` is not provided or is `None`, the method returns the entire list.

Parameters **index** (*int or None*) – Index into the perturbation list for the index in question (optional, if not provided or `None`, then returns entire list)

Returns pert or list

get_state()

Returns initial state variable value for interface

Returns Initial state variable

Return type float

get_statefile()

Returns state file of interface

If interface does not have a statefile returns None

Parameters **niface** – index of desired interface (zero-indexed)

Returns statefile or None

get_type()

Returns string of interface type

Returns the type of the given interface (“locked”, “frictionless”, “slipweak”, or “stz”)

Returns Interface type

Return type str

set_index(*index*)

Sets interface index

Changes value of interface index. New index must be a nonnegative integer

Parameters **index** (*int*) – New value of index (nonnegative integer)

Returns None

set_loadfile(*newloadfile*)

Sets loadfile for interface

`newloadfile` is the new loadfile (must have type `loadfile`). If the index is bad or the loadfile type is not correct, the code will raise an error. Errors can also result if the shape of the loadfile does not match with the interface.

Parameters **newloadfile** (`loadfile`) – New loadfile to be used for the given interface

Returns None

set_paramfile(*newparamfile*)

Sets paramfile for the interface

Method sets the file holding frictional parameters for the interface.

`newparamfile` must be a parameter perturbation file of type `stzparam`. Errors can also result if the shape of the paramfile does not match with the interface.

Parameters **newparamfile** (`stzparamfile`) – New frictional parameter file

Returns None

set_state(*newstate*)

Sets initial state variable for interface

Set the initial value for the state variable. `state` is the new state variable (must be a float or some other valid number).

Parameters **state** (*float*) – New value of state variable

Returns None

set_statefile(*newstatefile*)

Sets state file for interface

Set the statefile for the interface.‘‘newstatefile‘‘must have type `statefile`. Errors can also result if the shape of the statefile does not match with the interface.

Parameters **newstatefile** (`statefile`) – New statefile for the interface in question.

Returns None

write_input (*f*, *probname*, *directory*, *endian='='*)

Writes interface details to input file

This routine is called for every interface when writing problem data to file. It writes the appropriate section for the interface in the input file. It also writes any necessary binary files holding interface loads, parameters, or state variables.

Parameters

- **f** (*file*) – File handle for input file
- **probname** (*str*) – problem name (used for naming binary files)
- **directory** (*str*) – Directory for output
- **endian** (*str*) – Byte ordering for binary files (`'<'` little endian, `'>'` big endian, `'='` native, default is native)

Returns None

class `fdfault.`**stzparam**(*perttype='constant'*, *t0=0.0*, *x0=0.0*, *dx=0.0*, *y0=0.0*, *dy=0.0*, *v0=0.0*, *f0=0.0*, *a=0.0*, *muy=0.0*, *c0=0.0*, *R=0.0*, *beta=0.0*, *chiw=0.0*, *v1=0.0*)

Class representing STZ Theory parameter perturbations to frictional interfaces

The `stzparam` class represents STZ Theory parameter perturbations that can be expressed in a simple functional form. The `stzparam` class holds information on the shape of the perturbation and the nine parameter values for the interface.

Perturbations have the following attributes:

Variables

- **perttype** – String describing perturbation shape. See available types below.
- **t0** – Perturbation onset time (linear ramp function that attains its maximum at t0; `t0 = 0.` means perturbation is on at all times)
- **x0** – Perturbation location along first spatial dimension (see below for details)
- **dx** – Perturbation scale along first spatial dimension (see below for details)
- **y0** – Perturbation location along second spatial dimension (see below for details)

- **dy** – Perturbation scale along second spatial dimension (see below for details)
- **v0** – Reference slip rate perturbation
- **f0** – Friction activation barrier perturbation
- **a** – Frictional direct effect perturbation
- **muy** – Yielding friction coefficient perturbation
- **c0** – Effective temperature specific heat perturbation
- **R** – Effective temperature relaxation rate perturbation
- **beta** – Effective temperature relaxation barrier perturbation
- **chiw** – Effective temperature activation barrier perturbation
- **v1** – Effective temperature reference slip rate perturbation

By default, all time, shape, and friction parameters are set to zero.

There are several available types of perturbations:

- `'constant'` – A spatially uniform perturbation. All spatial information is ignored
- `'boxcar'` – Perturbation is constant within a rectangle centered at `(x0,y0)` with a half width of `(dx,dy)` in each spatial dimension
- `'ellipse'` – Perturbation is constant within an ellipse centered at `(x0,y0)` with half axis lengths of `(x0,y0)`
- `'gaussian'` – Perturbation follows a Gaussian function centered at `(x0,y0)` with standard deviations `(dx,dy)` in each spatial dimension
- `'linear'` – Perturbation is a linear function with intercept `x0` and slope `1/dx` in the first spatial dimension and intercept `y0` and slope `1/dy` in the second spatial dimension. If either `dx` or `dy` is zero, the linear function is constant in that particular spatial dimension (i.e. set `dy = 0.` if you want to have a function that is only linear in the first spatial dimension)

The shape variables are only interpreted literally for rectangular blocks. If the block is not rectangular, then the shape variables are interpreted as if the block on the negative side were rectangular with the dimensions that are provided when setting up the problem. For example, if you run a problem with a dipping fault that has a trapezoidally shaped block on the minus side of the fault, then `x0` and `dx` would be measured in terms of depth rather than distance along the interface, since the "rectangular" version of the block would have depth along the fault dimension.

If you are in doubt regarding how a perturbation will be interpreted for a particular geometry, it is usually less ambiguous to use a file to set values, as they explicitly set the value at each grid point. However, for some simple forms, perturbations can be more convenient as they use less memory and do not require loading information in parallel from external files.

`__init__`(*perttype='constant'*, *t0=0.0*, *x0=0.0*, *dx=0.0*, *y0=0.0*, *dy=0.0*, *v0=0.0*, *f0=0.0*, *a=0.0*, *muy=0.0*, *c0=0.0*, *R=0.0*, *beta=0.0*, *chiw=0.0*, *v1=0.0*)

Initialize a new instance of an STZ parameter perturbation

Method creates a new instance of a STZ parameter perturbation. It calls the superclass routine to initialize the spatial and temporal details of the perturbation, and creates the variables holding the parameter values specific to the STZ law. Default values are provided for all arguments (all zeros, with a perttype of `'constant'`).

Parameters

- **`perttype`** (*`str`*) – Perturbation type (string, default is `'constant'`)
- **`t0`** (*`float`*) – Linear ramp time scale (default 0.)
- **`x0`** (*`float`*) – Perturbation location along first interface dimension (default 0.)
- **`dx`** (*`float`*) – Perturbation scale along first interface dimension (default 0.)
- **`y0`** (*`float`*) – Perturbation location along second interface dimension (default 0.)
- **`dy`** (*`float`*) – Perturbation scale along second interface dimension (default 0.)
- **`v0`** (*`float`*) – Reference slip rate perturbation (default 0.)
- **`f0`** (*`float`*) – Friction activation barrier perturbation (default 0.)
- **`a`** (*`float`*) – Frictional direct effect perturbation (default 0.)
- **`muy`** (*`float`*) – Yielding friction coefficient perturbation (default 0.)
- **`c0`** (*`float`*) – Effective temperature specific heat perturbation (default 0.)
- **`R`** (*`float`*) – Effective temperature relaxation rate perturbation (default 0.)
- **`beta`** (*`float`*) – Effective temperature relaxation barrier perturbation (default 0.)
- **`chiw`** (*`float`*) – Effective temperature activation barrier perturbation (default 0.)
- **`v1`** (*`float`*) – Effective temperature reference slip rate perturbation (default 0.)

Returns New instance of slip weakening parameter perturbation

Return type *stzparam*

get_R()

Returns effective temperature relaxation rate perturbation

Returns Effective temperature relaxation rate perturbation

Return type float

get_a()

Returns frictional direct effect perturbation

Returns frictional direct effect perturbation

Return type float

get_beta()

Returns effective temperature relaxation activation barrier perturbation

Returns Effective temperature relaxation activation barrier perturbation

Return type float

get_c0()

Returns effective temperature specific heat perturbation

Returns Effective temperature specific heat perturbation

Return type float

get_chiw()

Returns effective temperature activation barrier perturbation

Returns Effective temperature activation barrier perturbation

Return type float

get_dx()

Returns perturbation scale along first interface coordinate

Returns Scale of perturbation along first interface coordinate

Return type float

get_dy()

Returns perturbation scale along second interface coordinate

Returns Scale of perturbation along second interface coordinate

Return type float

get_f0()

Returns activation barrier perturbation

Returns activation barrier perturbation

Return type float

get_muy()
Returns yielding friction coefficient perturbation

Returns yielding friction coefficient perturbation

Return type float

get_t0()
Returns onset time

Returns Perturbation onset time

Return type float

get_type()
Returns perturbation type

Returns Perturbation type

Return type str

get_v0()
Returns reference slip rate perturbation

Returns reference slip rate perturbation

Return type float

get_v1()
Returns effective temperature reference slip rate perturbation

Returns Effective temperature reference slip rate perturbation

Return type float

get_x0()
Returns perturbation location in first interface coordinate

Returns Location of perturbation along first interface coordinate

Return type float

get_y0()
Returns perturbation location in second interface coordinate

Returns Location of perturbation along second interface coordinate

Return type float

set_R(*R*)
Sets effective temperature relaxation rate perturbation

Parameters **R** (*float*) – Value of effective temperature relaxation perturbation

Returns None

set_a(*a*)
Sets frictional direct effect perturbation

Parameters **a** (*float*) – Value of frictional direct effect perturbation

Returns None

set_beta(*beta*)
Sets effective temperature relaxation activation barrier perturbation

Parameters **beta** (*float*) – Value effective temperature relaxation activation barrier perturbation

Returns None

set_c0(*c0*)
Sets effective temperature specific heat perturbation

Parameters **c0** (*float*) – Value of effective temperature specific heat perturbation

Returns None

set_chiw(*chiw*)
Sets effective temperature activation barrier perturbation

Parameters **chiw** (*float*) – Value of effective temperature activation barrier perturbation

Returns None

set_dx(*dx*)
Sets first coordinate of perturbation scale

Changes value of perturbation scale for first coordinate direction. New value must be nonnegative.

Parameters **dx** (*float*) – New value of perturbation scale along second coordinate

Returns None

set_dy(*dy*)
Sets second coordinate of perturbation scale

Changes value of perturbation scale for second coordinate direction. New value must be nonnegative.

Parameters **dy** (*float*) – New value of perturbation scale along second coordinate

Returns None

set_f0(*f0*)
Sets activation barrier perturbation

Parameters **f0** (*float*) – Value of activation barrier perturbation

Returns None

set_muy (*muy*)
Sets yielding friction coefficient perturbation

Parameters **muy** (*float*) – Value yielding friction coefficient perturbation

Returns None

set_t0 (*t0*)
Sets onset time

Changes value of onset time. New value must be nonnegative.

Parameters **t0** (*float*) – New value of onset time

Returns None

set_type (*perttype*)
Sets perturbation type

Resets the perturbation type to `perttype`. Note that the new type must be among the valid perturbation types.

Parameters **perttype** (*str*) – New value for perttype, must be a valid perturbation type

Returns None

set_v0 (*v0*)
Sets reference slip rate perturbation

Parameters **v0** (*float*) – Value of reference slip rate perturbation

Returns None

set_v1 (*v1*)
Sets effective temperature reference slip rate perturbation

Parameters **v1** (*float*) – Value of effective temperature reference slip rate perturbation

Returns None

set_x0 (*x0*)
Sets first coordinate of perturbation location

Parameters **x0** (*float*) – New value of perturbation location along first coordinate

Returns None

set_y0 (*y0*)
Sets second coordinate of perturbation location

Parameters **`x0`** (*`float`*) – New value of perturbation location along second coordinate

Returns None

`write_input` (*f*)

Writes perturbation to input file

Method writes perturbation to input file (input file provided as input)

Parameters **`f`** (*`file`*) – Output file to which the perturbation will be written

Returns none

class `fdfault.`**`stzparamfile`**(*n1*, *n2*, *v0*, *f0*, *a*, *muy*, *c0*, *R*, *beta*, *chiw*, *v1*)

The `stzparamfile` class is a class for loading heterogeneous friction parameter values from file. It is only used for STZ interfaces.

All `stzparamfile` instances contain the following internal parameters:

Variables

- **`n1`** – Number of grid points along first coordinate direction
- **`n2`** – Number of grid points along the second coordinate direction
- **`v0`** – Array holding reference slip rate perturbation (numpy array with shape `(n1,n2)`)
- **`f0`** – Array holding friction activation barrier perturbation (numpy array with shape `(n1,n2)`)
- **`a`** – Array holding frictional direct effect perturbation (numpy array with shape `(n1,n2)`)
- **`muy`** – Array holding yielding friction coefficient perturbation (numpy array with shape `(n1,n2)`)
- **`c0`** – Array holding effective temperature specific heat perturbation (numpy array with shape `(n1,n2)`)
- **`R`** – Array holding effective temperature relaxation rate perturbation (numpy array with shape `(n1,n2)`)
- **`beta`** – Array holding effective temperature relaxation activation barrier perturbation (numpy array with shape `(n1,n2)`)
- **`chiw`** – Array holding effective temperature activation barrier perturbation (numpy array with shape `(n1,n2)`)
- **`v1`** – Array holding effective temperature reference slip rate perturbation (numpy array with shape `(n1,n2)`)

`stzparamfile` will also define nine numpy array with shape `(n1,n2)` holding the various friction parameters (reference slip rate, friction activation barrier, frictional direct effect, yielding friction coefficient, effective temperature specific heat, effective temperature relaxation rate, effective temperature relaxation activation barrier, effective temperature activation barrier, and effective temperature reference slip rate. STZ parameter files do not include any information about the shape of the boundary, and it is up to the user to ensure that that the parameter values correspond to the coordinates of the interface. However, because parameter files explicitly assign a value to each grid point, there is less ambiguity regarding the final values when compared to perturbations. Depending on the orientation of the interface, the two coordinate directions will have different orientations in space. The first coordinate direction is the x direction for y and z interfaces (for x interfaces, the first index is in the y direction), and the second coordinate is in the z direction except for z interfaces, where y is the second index}.

When writing `stzparamfile` instances to disk, the code uses numpy to write information to disk in binary format. Byte-ordering can be specified, and should correspond to the byte-ordering on the system where the simulation will be run (default is native).

`__init__`(*n1*, *n2*, *v0*, *f0*, *a*, *muy*, *c0*, *R*, *beta*, *chiw*, *v1*)

Initialize a new instance of a stzparamfile object

Create a new instance of a stzparamfile, which is a class describing STZ parameter perturbations in a file. Required information is the number of grid points for the interface and one array for each of the nine parameter perturbations. All the array shapes must be `(n1,n2)` or the code will raise an error.

Parameters

- **n1** (*int*) – Number of grid points along first coordinate direction
- **n2** (*int*) – Number of grid points along the second coordinate direction
- **v0** (*ndarray*) – Reference slip rate perturbation array
- **f0** (*ndarray*) – Friction activation barrier perturbation array
- **a** (*ndarray*) – Frictional direct effect perturbation array
- **muy** (*ndarray*) – Yielding friction coefficient perturbation array
- **c0** (*ndarray*) – Effective temperature specific heat perturbation array
- **R** (*ndarray*) – Effective temperature relaxation rate perturbation array
- **beta** (*ndarray*) – Effective temperature relaxation barrier perturbation array
- **chiw** (*ndarray*) – Effective temperature activation barrier perturbation array
- **v1** (*ndarray*) – Effective temperature reference slip rate perturbation array

Returns New swparamfile instance

Return type *swparamfile*

get_R(*index=None*)

Returns effective temperature relaxation rate at given indices

Returns effective temperature relaxation rate perturbation at the indices given by `index`. If no indices are provided, the method returns the entire array.

Parameters **index** (*float, tuple, or None*) – Index into effective temperature relaxation rate array (optional, if not provided returns entire array)

Returns Effective temperature relaxation rate perturbation (either ndarray or float, depending on value of `index`)

Return type ndarray or float

get_a(*index=None*)

Returns frictional direct effect at given indices

Returns frictional direct effect perturbation at the indices given by `index`. If no indices are provided, the method returns the entire array.

Parameters **index** (*float, tuple, or None*) – Index into frictional direct effect array (optional, if not provided returns entire array)

Returns Frictional direct effect perturbation (either ndarray or float, depending on value of `index`)

Return type ndarray or float

get_beta(*index=None*)

Returns effective temperature relaxation activation barrier at given indices

Returns effective temperature relaxation activation barrier perturbation at the indices given by `index`. If no indices are provided, the method returns the entire array.

Parameters **index** (*float, tuple, or None*) – Index into effective temperature relaxation activation barrier array (optional, if not provided returns entire array)

Returns Effective temperature relaxation activation barrier perturbation (either ndarray or float, depending on value of `index`)

Return type ndarray or float

get_c0(*index=None*)

Returns effective temperature specific heat at given indices

Returns effective temperature specific heat perturbation at the indices given by `index`. If no indices are provided, the method returns the entire array.

Parameters **`index`** (*`float, tuple, or None`*) – Index into effective temperature specific heat array (optional, if not provided returns entire array)

Returns Effective temperature specific heat perturbation (either ndarray or float, depending on value of `index`)

Return type ndarray or float

`get_chiw`(*index=None*)

Returns effective temperature activation barrier at given indices

Returns effective temperature activation barrier perturbation at the indices given by `index`. If no indices are provided, the method returns the entire array.

Parameters **`index`** (*`float, tuple, or None`*) – Index into effective temperature activation barrier array (optional, if not provided returns entire array)

Returns Effective temperature activation barrier perturbation (either ndarray or float, depending on value of `index`)

Return type ndarray or float

`get_f0`(*index=None*)

Returns friction activation barrier at given indices

Returns friction activation barrier perturbation at the indices given by `index`. If no indices are provided, the method returns the entire array.

Parameters **`index`** (*`float, tuple, or None`*) – Index into friction activation barrier array (optional, if not provided returns entire array)

Returns Friction activation barrier perturbation (either ndarray or float, depending on value of `index`)

Return type ndarray or float

`get_muy`(*index=None*)

Returns yielding friction coefficient at given indices

Returns yielding friction coefficient perturbation at the indices given by `index`. If no indices are provided, the method returns the entire array.

Parameters **`index`** (*`float, tuple, or None`*) – Index into yielding friction coefficient array (optional, if not provided returns entire array)

Returns Yielding friction coefficient perturbation (either ndarray or float, depending on value of `index`)

Return type ndarray or float

`get_n1`()

Returns number of grid points in 1st coordinate direction

Returns Number of grid points in 1st coordinate direction (x, except for x interfaces, where y is the first coordinate direction)

Return type int

get_n2()

Returns number of grid points in 2nd coordinate direction

Returns Number of grid points in 2nd coordinate direction (z, except for z interfaces, where y is the second coordinate direction)

Return type int

get_v0(*index=None*)

Returns reference slip rate at given indices

Returns reference slip rate perturbation at the indices given by `index`. If no indices are provided, the method returns the entire array.

Parameters **index** (*float, tuple, or None*) – Index into reference slip rate array (optional, if not provided returns entire array)

Returns Reference slip rate perturbation (either ndarray or float, depending on value of `index`)

Return type ndarray or float

get_v1(*index=None*)

Returns effective temperature reference slip rate at given indices

Returns effective temperature reference slip rate perturbation at the indices given by `index`. If no indices are provided, the method returns the entire array.

Parameters **index** (*float, tuple, or None*) – Index into effective temperature reference slip rate array (optional, if not provided returns entire array)

Returns Effective temperature reference slip rate perturbation (either ndarray or float, depending on value of `index`)

Return type ndarray or float

write(*filename*, *endian='='*)

Write perturbation data to file

Parameters

- **filename** (*str*) – Name of binary file to be written
- **endian** (*str*) – Byte-ordering for output. Options inclue `'='` for native, `'<'` for little endian, and `'>'` for big endian. Optional, default is native

Returns None

class `fdfault.`**`statefile`**(*n1*, *n2*, *state*)

The `statefile` class is a class for loading heterogeneous initial state variable values from file. It is only used for friction laws that require an additional state variable in addition to the slip/slip rate to specify frictional strength

All `statefile` instances contain the following internal parameters:

Variables

- **`n1`** – Number of grid points along first coordinate direction
- **`n2`** – Number of grid points along the second coordinate direction
- **`state`** – Array holding state variable perturbation (numpy array with shape `(n1,n2)`)

`statefile` will also define a numpy array with shape `(n1,n2)` holding the state variable. State files do not include any information about the shape of the boundary, and it is up to the user to ensure that that the parameter values correspond to the coordinates of the interface. However, because parameter files explicitly assign a value to each grid point, there is less ambiguity regarding the final values when compared to perturbations. Depending on the orientation of the interface, the two coordinate directions will have different orientations in space. The first coordinate direction is the x direction for y and z interfaces (for x interfaces, the first index is in the y direction), and the second coordinate is in the z direction except for z interfaces, where y is the second index}.

When writing `statefile` instances to disk, the code uses numpy to write information to disk in binary format. Byte-ordering can be specified, and should correspond to the byte-ordering on the system where the simulation will be run (default is native).

`__init__`(*n1*, *n2*, *state*)

Initialize a new instance of a statefile object

Create a new instance of a statefile, which is a class describing interface state variable value perturbations in a file. Required information is the number of grid points for the interface and one array for the state variable. The array shape must be `(n1,n2)` or the code will raise an error.

Parameters

- **`n1`** (*`int`*) – Number of grid points along first coordinate direction
- **`n2`** (*`int`*) – Number of grid points along the second coordinate direction
- **`sn`** (*`ndarray`*) – State variable perturbation array

Returns New statefile instance

Return type *statefile*

`get_state`(*index=None*)

Returns state variable at given indices

Returns state variable perturbation at the indices given by `index`. If no indices are provided, the method returns the entire array.

Parameters **`index`** (*`float, tuple, or None`*) – Index into state variable array (optional, if not provided returns entire array)

Returns State variable perturbation (either ndarray or float, depending on value of `index`)

Return type ndarray or float

`write` (*filename*, *endian='='*)

Write perturbation data to file

Parameters

- **`filename`** (*`str`*) – Name of binary file to be written
- **`endian`** (*`str`*) – Byte-ordering for output. Options inclue `'='` for native, `'<'` for little endian, and `'>'` for big endian. Optional, default is native

Returns None

The `output` Class

The `output` class contains information on saving simulation data to file.

Each problem contains a list of "output units," each of which saves a particular set of data points from the simulation to file. An output unit is given a name, selects a field from the simulation, and picks grid and time points to output. The corresponding spatial grid and time information is automatically output for each item.

Output items fall into two main groups: fields that are defined in the volume (particle velocities, stress components, and plastic strain/strain rate). These are defined for every grid point in the simulation, and so there are no restrictions on the range that the spatial points can take other than them being within the domain and the min/max values are self-consistent.

Other fields such as slip, slip rate, interface tractions, and state variables, are only defined on a particular interface between blocks. These output units require more careful specification of their limits, as the points chosen must lie on an interface. The Python and C++ code check the validity of these limits (the Python code only checks that the points make up a 2D slice, while the C++ code ensures that the 2D slice lies on an interface in the simulation).

The grid point limits can be used to specify a single point, 1D, 2D, or 3D selection of the domain for output by setting the min/max values to be identical or different.

Output can only be done for specific grid points within the computational domain. However, a helper function `find_nearest_point` can be used for a given problem to find the nearest grid point to a spatial location for complex domains.

Acceptable field names for volume output:

- `'vx'` - x particle velocity
- `'vy'` - y particle velocity
- `'vz'` - z particle velocity
- `'sxx'` - xx stress component
- `'sxy'` - xy stress component
- `'sxz'` - xz stress component
- `'syy'` - yy stress component
- `'syz'` - yz stress component
- `'szz'` - zz stress component
- `'gammap'` - scalar plastic strain
- `'lambda'` - scalar plastic strain rate

Acceptable field names for interface output (coordinates must form a 2D (1D for 2D problems) slice through the domain along a single interface between two blocks. Note that if your slice extends through multiple interfaces, only one will be saved due to how parallel I/O is handled in the code.

- `'U'` - scalar slip (line integral of slip velocity)
- `'Ux'` - x slip component (signed)
- `'Uy'` - y slip component (signed)
- `'Uz'` - z slip component (signed)
- `'V'` - scalar slip velocity (vector magnitude of components)
- `'Vx'` - x slip velocity component (signed)
- `'Vy'` - y slip velocity component (signed)
- `'Vz'` - z slip velocity component (signed)
- `'Sn'` - Interface normal traction (negative in compression)
- `'S'` - scalar interface shear traction (vector magnitude)
- `'Sx'` - x shear traction component (signed)
- `'Sy'` - y shear traction component (signed)
- `'Sz'` - z shear traction component (signed)

The different interface components do not truly correspond to the corresponding coordinate directions. The code handles complex boundary conditions by rotating the fields into a coordinate system defined by three mutually orthogonal unit vectors. The normal direction is defined to always point into the "positive" block and is uniquely defined by the boundary geometry. The two tangential components are defined as follows for each different type of interface:

- Depending on the orientation of the interface in the computational space, a different convention is used to set the first tangent vector. For `'x'` or `'y'` oriented interfaces, the z component of the first tangent vector is set to zero. This is done to ensure that for 2D problems, the second tangent vector points in the z-direction. For `'z'` oriented interfaces, the y component of the first tangent vector is set to zero.
- With one component of the first tangent vector defined, the other two components can be uniquely determined to make the tangent vector orthogonal up to a sign. The sign is chosen such that the tangent vector points in the direction where the grid points are increasing.
- The second tangent vector is defined by taking the right-handed cross product of the normal and first tangent vectors, except for `'y'` interfaces, where the left-handed cross product is used. This is done to ensure that for 2D problems, the vertical component always points in the $+z$-direction.

The consequence of this is that the letter used to designate the desired component is only valid for rectangular geometries. For non-rectangular geometries, the components will be rotated into the coordinate system described above. For interfaces in the "x" direction (i.e. connecting blocks whose indices only differ in the x-direction), the y component of output units will be along the first tangent vector, and the z component will be along the second tangent vector. Similarly, for "y" interfaces the x component is set by the first tangent vector and the z component is determined by the second tangent vector, and for "z" interfaces the first tangent vector is in the x-direction and the second tangent vector corresponds to the y-direction. If you desire the components in a different coordinate system, you can convert them from the output data. Note that this also means that you can only specify certain components for interface output, depending on the direction of the interface.

class `fdfault.`**`output`**(*name*, *field*, *tm=0*, *tp=0*, *ts=1*, *xm=0*, *xp=0*, *xs=1*, *ym=0*, *yp=0*, *ys=1*, *zm=0*, *zp=0*, *zs=1*)

Class representing a simulation dataset to be written to file.

Attributes include a name (used for setting the file name of the resulting files), output field, and grid point information. Initializing an output unit requires specifying a name and field, the grid point information is optional.

Variables

- **`name`** – Name used in files for saving data
- **`field`** – Field to be saved to file (see list of acceptable values above)
- **`tm`** – Minimum time index to be written to file (inclusive)
- **`tp`** – Maximum time index to be written to file (inclusive)
- **`ts`** – Stride for time output (will skip over appropriate number of time steps so that every `ts` time steps are saved between `tm` and `tp`)

- **`xm`** – Minimum x index to be written to file (inclusive)
- **`xp`** – Maximum x index to be written to file (inclusive)
- **`xs`** – Stride for x output (will skip over appropriate number of x grid points so that one per every `xs` points are saved between `xm` and `xp`)
- **`ym`** – Minimum y index to be written to file (inclusive)
- **`yp`** – Maximum y index to be written to file (inclusive)
- **`ys`** – Stride for y output (will skip over appropriate number of y grid points so that one per every `ys` points are saved between `ym` and `yp`)
- **`zm`** – Minimum z index to be written to file (inclusive)
- **`zp`** – Maximum z index to be written to file (inclusive)
- **`zs`** – Stride for z output (will skip over appropriate number of z grid points so that one per every `zs` points are saved between `zm` and `zp`)

`__init__`(*name*, *field*, *tm=0*, *tp=0*, *ts=1*, *xm=0*, *xp=0*, *xs=1*, *ym=0*, *yp=0*, *ys=1*, *zm=0*, *zp=0*, *zs=1*)

Initialize a new ouput unit

Creates a new output unit. Required arguments are the name and field to be saved. Specifying additional parameters gives the user control over what data is saved. Time and three spatial dimensions can be set with a triplet of integers representing minus, plius, and stride values. Minus sets the first index that is saved, plus sets the last, and stride controls how frequently the data is saves (stride of 1 means every value is saved, 2 means every other point is saved, etc.). Specifying values out of bounds such as minus > plus or any number less than zero will result in an error. If values that are outside the simulation range are given, the code will give a warning but not an error.

All triplets have default values of minus = 0, plus = 0, and stride = 1, and are optional.

Parameters

- **`name`** (*`str`*) – Name used in files for saving data
- **`field`** (*`str`*) – Field to be saved to file (see list of acceptable values above)
- **`tm`** (*`int`*) – Minimum time index to be written to file (inclusive)
- **`tp`** (*`int`*) – Maximum time index to be written to file (inclusive)
- **`ts`** (*`int`*) – Stride for time output (will skip over appropriate number of time steps so that every `ts` time steps are saved between `tm` and `tp`)
- **`xm`** (*`int`*) – Minimum x index to be written to file (inclusive)
- **`xp`** (*`int`*) – Maximum x index to be written to file (inclusive)

- **xs** (*int*) – Stride for x output (will skip over appropriate number of x grid points so that one per every `xs` points are saved between `xm` and `xp`)
- **ym** (*int*) – Minimum y index to be written to file (inclusive)
- **yp** (*int*) – Maximum y index to be written to file (inclusive)
- **ys** (*int*) – Stride for y output (will skip over appropriate number of y grid points so that one per every `ys` points are saved between `ym` and `yp`)
- **zm** (*int*) – Minimum z index to be written to file (inclusive)
- **zp** (*int*) – Maximum z index to be written to file (inclusive)
- **zs** (*int*) – Stride for z output (will skip over appropriate number of z grid points so that one per every `zs` points are saved between `zm` and `zp`)

Returns New instance of output unit

Return type *output*

get_field()

Returns output field

Returns Output field

Return type str

get_name()

Returns output unit name

Returns Output unit name

Return type str

get_time_indices()

Returns all index info for time output as (tm, tp, ts)

Returns Set of time step info indices (tm, tp, ts)

Return type tuple

get_tm()

Returns minimum time step for output

Returns minimum time step index

Return type int

get_tp()

Returns maximum time step for output

Returns maximum time step index

Return type int

get_ts()
Returns time stride for output

Returns time stride

Return type int

get_x_indices()
Returns all index info for x output as (xm, xp, xs)

Returns all x grid point information (xm, xp, xs)

Return type tuple

get_xm()
Returns minimum x grid point for output

Returns minimum x grid point

Return type int

get_xp()
Returns maximum x grid point for output

Returns maximum x grid point

Return type int

get_xs()
Returns x stride for output

Returns x stride

Return type int

get_y_indices()
Returns all index info for y output as (ym, yp, ys)

Returns Index information for the y direction (ym, yp, ys)

Return type tuple

get_ym()
Returns minimum y grid point for output

Returns minimum y grid point

Return type int

get_yp()
Returns maximum y grid point for output

Returns maximum y grid index

Return type int

get_ys()

Returns y stride for output

Returns y stride for data output

Return type int

get_z_indices()

Returns all index info for z output as (zm, zp, zs)

Returns z output information (zm, zp, zs)

Return type tuple

get_zm()

Returns minimum z grid point for output

Returns minimum z grid point

Return type int

get_zp()

Returns maximum z grid point for output

Returns maximum z grid point

Return type int

get_zs()

Returns z stride for output

Returns z stride

Return type int

set_field(*field*)

Sets output file (must be a valid block or interface field)

Parameters **field** (`str`) – New output field (must match option above)

Returns None

set_name(*name*)

Set output unit name (if not a string, the code will produce an error)

Parameters **name** (`str`) – New name for output unit

Returns None

set_time_indices(*tm*, *tp*, *ts*)

Sets all time indices

Method sets all three values of `tm`, `tp`, and `ts`

Parameters

- **tm** (`int`) – New value of minimum time index

- **tp** (*int*) – New value of maximum time index
- **ts** (*int*) – New value of time stride

Returns None

set_tm(*tm*)
Sets minimum time index for output

New minimum must be nonnegative and less than `tp`

Parameters **tm** (*int*) – New value of minimum time step

Returns None

set_tp(*tp*)
Sets maximum time index for output

New value of maximum time must be nonnegative and not less than `tm`

Parameters **tp** (*int*) – New value of minimum time index

Returns None

set_ts(*ts*)
Sets t stride for output

Stride must be a positive integer.

Parameters **ts** (*int*) – New value of time stride

Returns None

set_x_indices(*xm*, *xp*, *xs*)
Sets all x indices

Method sets all three values of `xm`, `xp`, and `xs`

Parameters

- **xm** (*int*) – New value of minimum x index
- **xp** (*int*) – New value of maximum x index
- **xs** (*int*) – New value of x stride

Returns None

set_xm(*xm*)
Sets minimum x index for output

New minimum must be nonnegative and less than `xp`

Parameters **xm** (*int*) – New value of minimum x grid point

Returns None

set_xp (*xp*)
Sets maximum x index for output

New value of maximum x index must be nonnegative and not less than `xm`

Parameters **xp** (*int*) – New value of maximum x index

Returns None

set_xs (*xs*)
Sets x stride for output

Stride must be a positive integer.

Parameters **xs** (*int*) – New value of x stride

Returns None

set_y_indices (*ym, yp, ys*)
Sets all y indices

Method sets all three values of `ym`, `yp`, and `ys`

Parameters

- **ym** (*int*) – New value of minimum y index
- **yp** (*int*) – New value of maximum y index
- **ys** (*int*) – New value of y stride

Returns None

set_ym (*ym*)
Sets minimum y index for output

New minimum must be nonnegative and less than `yp`

Parameters **ym** (*int*) – New value of minimum y grid point

Returns None

set_yp (*yp*)
Sets maximum y index for output

New value of maximum y index must be nonnegative and not less than `ym`

Parameters **yp** (*int*) – New value of maximum y index

Returns None

set_ys (*ys*)
Sets y stride for output

Stride must be a positive integer.

Parameters **ys** (*int*) – New value of y stride

Returns None

`set_z_indices`(*zm*, *zp*, *zs*)
Sets all z indices

Method sets all three values of `zm`, `zp`, and `zs`

Parameters

- **`zm`** (*`int`*) – New value of minimum z index
- **`zp`** (*`int`*) – New value of maximum z index
- **`zs`** (*`int`*) – New value of z stride

Returns None

`set_zm`(*zm*)
Sets minimum z index for output

New minimum must be nonnegative and less than `zp`

Parameters **`zm`** (*`int`*) – New value of minimum z grid point

Returns None

`set_zp`(*zp*)
Sets maximum z index for output

New value of maximum z index must be nonnegative and not less than `zm`

Parameters **`zp`** (*`int`*) – New value of maximum z index

Returns None

`set_zs`(*zs*)
Sets z stride for output

Stride must be a positive integer.

Parameters **`zs`** (*`int`*) – New value of z stride

Returns None

`write_input`(*f*)
Writes output unit to file

Method writes the information for the output unit to file. The information is inserted into a list of output units that are formatted for input into the C++ code.

Parameters **`f`** (*`file`*) – file handle for output file

Returns None

3.2.4 Additional Classes

The classes below are not normally accessed by the user. Instead, use the interfaces provided in the `problem` class, which modify the underlying classes in a more robust way and prevent you from setting up a problem incorrectly. However, full documentation for the additional classes are included here for completeness.

The `block` Class

The `block` class represents a block of material in a simulation. Blocks are not meant to be interacted with directly – when using the `problem` class to set up a simulation, blocks are automatically added or deleted as needed using the provided interfaces. This documentation is provided for completeness, but should not need to be used in regular use of the code.

The class contains information on the number of grid points in the block, the location of the block within the simulation, the material properties using the `material` class, the types of boundary conditions at the block edges, and any complex geometries specified through the `surface` and `curve` classes.

class `fdfault.`**`block`**(*ndim*, *mode*, *nx*, *mat*)

Class representing a block in a simulation

A block contains the following internal variables:

Variables

- **`ndim`** – Number of dimensions (2 or 3)
- **`mode`** – Rupture mode (2 or 3, relevant only for 2D problems)
- **`nx`** – Number of grid points (tuple of 3 positive integers)
- **`xm`** – Coordinates of lower left corner in simulation (tuple of 3 nonnegative integers)
- **`coords`** – Location of block within simulation domain (tuple of 3 nonnegative integers)
- **`lx`** – Block length in each spatial dimension (tuple of 3 positive floats, can be overridden by setting a curve or surface to one of the edges)

Parameters

- **`m`** (`material`) – Material properties (see `material` class)
- **`bounds`** (*`list`*) – List of boundary conditions. Position indicates boundary location (0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top). Possible strings for boundary condition include `'absorbing'`

(no incoming wave), `'free'` (traction free surface), `'rigid'` (no displacement), or `'none'` (boundary conditions determined by interface conditions)

- **surfs** (*list*) – List of bounding surfaces. Position indicates boundary location (0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top). For 2D problems, you can only populate the list with curves, and 3D problems require surfaces. If the default rectangular surface is to be used, use `None` for a particular surface.

`__init__`(*ndim*, *mode*, *nx*, *mat*)

Initialize a new `block` instance

Creates a new block instance with the given dimensionality, rupture mode, number of grid points, and material type. If the problem is 2d, the number of z grid points will be automatically set to one. By default, the block length is unity in each direction, the lower left coordinate is `(0.,0.,0.)`, all boundary conditions are set to `'none'`, material properties take on their default values, and there are no irregular edge shapes. All of these default properties can be modified using the provided interfaces.

Parameters

- **ndim** (*int*) – Number of spatial dimensions (must be 2 or 3)
- **mode** (*int*) – Slip mode (2 or 3, only relevant if the problem is in 2D)
- **nx** (*tuple or list*) – Tuple of length 3 with number of grid points `(nx,ny,nz)`
- **mat** (*str*) – Block material type (string, must be `'elastic'` or `'plastic'`). The method initializes a default set of material properties based on this type.

Returns New block instance

Return type *block*

`check`()

Checks for errors before writing input file

Checks that edges of bounding surfaces match. If the edges are not defined as surfaces, the code temporarily creates them to check that they match.

Returns None

`checksurfs`(*tmpsurfs*)

Checks that surface boundaries match

Input is a list of surfaces, with order corresponding to (left, right, front, back, top, bottom). In 2D problems, there is no top or bottom surface.

Parameters **tmpsurfs** (*list*) – List of surfaces to compare (lenth 4 or 6)

Returns None

delete_surf(*loc*)

Removes boundary surface for a particular block edge

Removes the bounding surface of a particular block edge. Location is determined by `loc` which is an integer that indexes into a list. Locations correspond to the following: 0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top. Note that the location must be 0 <= loc < 2*ndim

If `loc` is out of bounds, the code will also signal an error.

Parameters **loc** (*int*) – Location of desired boundary to be removed (0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top). For 2D problems, `loc` must be between 0 and 3.

Returns None

get_bounds(*loc=None*)

Returns boundary types

If `loc` (int) is provided, the method returns a specific location (str). Otherwise it returns a list of all boundaries, which will have length 4 for 2D problems and length 6 for 3D problems. `loc` serves effectively as an index into the list, and the indices correspond to the following: 0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top. Note that the location must be 0 <= loc < 2*ndim

Parameters **loc** (*int or None*) – Location of boundary that is desired (optional). If `loc` is not provided, returns a list

Returns Boundary type (if `loc` provided, returns a string of the boundary type for the desired location, of not returns a list of strings indicating all boundary types)

Return type str or list

get_coords()

Returns block coordinates (tuple of integer indices in each coordinate direction)

Returns Block coordinates (tuple of 3 integers)

Return type tuple

get_lx()

Returns block lengths as (lx, ly, lz) tuple

Returns Block dimensions (tuple of 3 floats) in x, y, and z dimensions

Return type tuple

get_material()

Returns material

Returns the material class associated with this block

Returns Material class with properties for this block

Return type *material*

get_mode()

Returns rupture mode (2 or 3), only valid for 2D problems (stored at domain level)

Returns Rupture mode

Return type int

get_ndim()

Returns Number of spatial dimensions

Returns Number of spatial dimensions

Return type int

get_nx()

Returns number of grid points in (nx, ny, nz) format for the given block

Returns Number of grid points (tuple of three integers)

Return type tuple

get_surf(*loc*)

Returns block boundary surface for a block edge

Returns the surface assigned to a specific edge. `loc` determines the edge that is returned (integer, corresponding to an index). Location indices correspond to the following: 0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top Note that the location must be 0 <= loc < 2*ndim (for 2D problems, `loc` cannot be 5 or 6).

Returns either a curve (2D problems) or surface (3D problems) or None

If `loc` indices are out of bounds, the code will raise an error.

Parameters **loc** (*int*) – Location of desired boundary (0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top). For 2D problems, `loc` must be between 0 and 3.

Returns curve or surface corresponding to the selected location. If the desired edge does not have a bounding surface, returns None.

Return type curve or surface or None

get_x(*coord*)

Returns grid value for given spatial index

For a given problem set up, returns the location of a particular set of coordinate indices. Note that since blocks are set up by setting values only on the edges, coordinates on the interior are not specified *a priori* and instead determined using transfinite interpolation to generate a regular grid on the block interiors. Calling `get_x` generates the interior grid to find the coordinates of the desired point.

Within each call to `get_x`, the grid is generated on the fly only for the relevant block where the desired point is located. It is not stored. This helps reduce memory requirements for large 3D problems (since the Python module does not run in parallel), but is slower. Because the computational grid is regular, though, it can be done in a single step in closed form.

Returns a numpy array of length 3 holding the spatial location (x, y, z).

Parameters **`coord`** (*`tuple or list`*) – Spatial coordinate where grid values are desired (tuple or list of 3 integers or 2 integers for 2D problems)

Returns (x, y, z) coordinates of spatial location

Return type ndarray

`get_xm``()`

Returns starting index (zero-indexed) of block (tuple of 3 integers)

Returns Coordinates of lower left corner (tuple of 3 integers)

Return type tuple

`make_tempsurfs``()`

Create temporary surface list

This method generates all six (four in 2D) bounding surfaces (curves in 2D). Note that these surfaces are not usually stored for rectangular block edges to save memory, as they are trivial to create. The temporary surfaces can be used to check that the edges of the surfaces/curves match or to use transfinite interpolation to generate the grid.

Returns List of all bounding surfaces (not stored beyond the time they are needed)

Return type list

`set_bounds``(`*bounds*, *loc=None*`)`

Sets boundary types

Changes the type of boundary conditions on a block. Acceptable values are ‘absorbing’ (incoming wave amplitude set to zero), ‘free’ (no traction on boundary), ‘rigid’ (no displacement of boundary), or ‘none’ (boundary conditions set by imposing interface conditions).

There are two ways to use `set_bounds`:

1. Set `loc` to be `None` (default) and provide a list of strings specifying boundary type for `bounds`. The length of `bounds` is 4 for a 2D simulation and 6 for 3D.
2. Set `loc` to be an integer denoting location and give `bounds` as a single string. The possible locations correspond to the following: 0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top. 4 and 5 are only applicable to 3D simulations ($0 <= loc < 2*ndim$).

Parameters

- **bounds** (*str or list*) – New boundary condition type (string or list of strings)
- **loc** (*int or None*) – If provided, only change one type of boundary condition rather than all (optional, loc serves as an index into the list if used)

Returns None

set_coords(*coords*)
Sets block coordinates to a new value

Set block coordinates to `coords`, and tuple of nonnegative integers denoting the location of the block in the domain.

Parameters **coords** (*tuple or list*) – New coordaintes (tuple or list of nonnegative integers)

Returns None

set_lx(*lx*)
Sets block lengths

Changes block length to `lx` (tuple of 2 (2D only) or 3 floats) where the block length in each dimension is given by `(lx,ly,lz)`

Parameters **lx** (*tuple or list*) – New value of block lengths (tuple of 2 (2D) or 3 floats)

Returns None

set_material(*mat*)
Sets block material properties

Sets new material properties stored in an instance of the `material` class.

Parameters

- **newmaterial** (`material`) – New material properties
- **coords** (*tuple or list*) – Coordinates of block to be changed (optional, omitting changes all blocks). `coords` must be a tuple or list of three integers that match the coordinates of a block.

Returns None

set_mattype(*mattype*)
Sets block material type ('elastic' or 'plastic')

Sets the material type for the block. Options are 'elastic' for an elastic simulation and 'plastic' for a plastic simulation. Anything else besides these options will cause the code to raise an error.

Parameters **mattype** (*str*) – New material type ('elastic' or 'plastic')

Returns None

set_mode(*mode*)

Sets rupture mode

Rupture mode is only valid for 2D problems, and is either 2 or 3 (other values will cause an error, and non-integer values will be converted to integers). For 3D problems, entering a different value of the rupture mode will alter the rupture mode cosmetically but will have no effect on the simulation.

Parameters **mode** (*int*) – New value of rupture mode

Returns None

set_ndim(*ndim*)

Sets number of dimensions

The new number of spatial dimensions must be an integer, either 2 or 3. If a different value is given, the code will raise an error. If a non-integer value is given that is acceptable, the code will convert it to an integer.

Note: Converting a 3D problem into a 2D problem will automatically collapse the number of grid points and the number of blocks in the z direction to be 1. Any modifications to these quantities that were done previously will be lost.

Parameters **ndim** (*int*) – New value for ndim (must be 2 or 3)

Returns None

set_nx(*nx*)

Sets number of grid points

Changes the number of grid points to the specified tuple/list of 3 nonnegative integers. Bad values of `nx` will raise an error.

Parameters **nx** (*tuple or list*) – New value of number of grid points (tuple of 3 positive integers)

Returns None

set_surf(*loc*, *surf*)

Sets boundary surface for a particular block edge

Changes the bounding surface of a particular block edge. Location is determined by `loc` which is an integer that indexes into a list. Locations correspond to the following: 0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top. Note that the location must be 0 <= loc < 2*ndim

For 2D problems, `surf` must be a curve. For 3D problems, `surf` must be a surface. Other choices will raise an error. If `loc` is out of bounds, the code will also signal an error.

Parameters

- **loc** (*int*) – Location of desired boundary (0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top). For 2D problems, `loc` must be between 0 and 3.
- **surf** (*curve or surface*) – curve or surface corresponding to the selected block and location

Returns None

set_xm (*xm*)

Sets block lower left coordinate

Changes lower left coordinate of a block to the provided tuple/list of integers.

Parameters **xm** (*tuple or list*) – New value of lower left coordinate (list/tuple of integers)

Returns None

write_input (*f*, *probname*, *directory*, *endian='='*)

Writes block information to input file

Method writes information for block to file. It also writes all relevant surface data to file to describe non-rectangular geometries. Inputs inlcude the file handle for the input file, the problem name (used for naming surface files), the destination directory for all files, and endianness (optional, default is native) for binary surface files.

Parameters

- **f** (*file*) – file handle for input file
- **probname** (*str*) – Problem name
- **directory** (*str*) – Directory where output should be written
- **endian** (*str*) – Byte-ordering for binary files for surface data. Possible values are `'<'` (little endian), `'>'` (big endian), or `'='` (native, default)

Returns None

The `domain` Class

The `domain` class holds information regarding the physical problem setup. This includes the problem dimension, rupture mode, if a problem is an elastic or plastic simulation, number of blocks and interfaces, grid spacing, finite difference method, and parallelization.

The user does not typically interact directly with `domain` objects, as all problems automatically contain one (and can only handle one) domain. All methods in `domain` contain interfaces through the `problem` object, and these wrapper functions should be the preferred method for altering a problem. Documentation is provided here for completeness.

class `fdfault.`**`domain`**

Class describing rupture problem domain

When initializing a domain, one is created with default attributes, including:

Variables

- **`ndim`** (*`int`*) – Number of spatial dimensions (2 or 3; default is 2)
- **`mode`** (*`int`*) – Rupture mode (2 or 3, default is 2; only relevant for 2D problems)
- **`mattype`** (*`str`*) – Simulation material type (`'elastic'` or `'plastic'`, default is `'elastic'`)
- **`nx`** (*`tuple`*) – Number of grid points (tuple of 3 integers, default (1,1,1)). This cannot be modified as it is set automatically when modifying `nx_block`.
- **`nblocks`** (*`tuple`*) – Number of blocks in each spatial dimension (tuple of 3 integers). Blocks must form a Cartesian grid.
- **`nx_block`** (*`tuple`*) – Number of grid points for each block along each spatial dimension. Represented as a tuple of lists. Default is ([1], [1], [1]). Note that modifying `nx_block` automatically changes `nx` to match.
- **`xm_block`** (*`tuple`*) – Grid location of minimum grid point in each block in each spatial dimension. This is also calculated automatically based on the values of `nx_block`, and cannot be modified directly. Represented as a tuple of lists, default is ([0], [0], [0]).
- **`nifaces`** (*`int`*) – Number of interfaces (integer, default is zero). This is automatically set when `nblocks` is changed, and is not set by the user. Order is not important here, but when generating interfaces the code orders them by first creating all `'x'` interfaces, then all `'y'` interfaces, then all `'z'` interfaces.
- **`iftype`** (*`list`*) – List holding type of all interfaces (list of strings). Can be modified by changing interface type for a single interface. When a new interface is created it is by default a `'locked'` interface.
- **`sbporder`** (*`int`*) – Finite difference method order (integer 2-4, default 2).
- **`nproc`** (*`tuple`*) – Number of processes in each dimension for parallelization. Represented as a tuple of integers (default (0,0,0)). A zero in a given dimension indicates that the number of processes in that direction will be set automatically. Thus, (0,0,0) indicates that the entire

decomposition process will be automated, while (0,2,1) fixes the number of processes in the y and z directions (x will still be determined automatically). Note that specifying all three numbers requires that the product of all three numbers match the total number of processes selected when running the simulation.

- **`cdiss`** (*`float`*) – Artificial dissipation coefficient (float, default 0.). A nonzero value will turn on artificial dissipation in the simulation. It is up to the user to select this value correctly.

`__init__`()

Create a new instance of a `domain` class

Initializes a new domain with a 2D mode 2 rupture containing a single elastic block. The block has the default material properties, one grid point in each direction, unit length in each direction, and the block is located at (0., 0., 0.) in space. The domain has a default finite difference order of 2 and no artificial dissipation. All defaults can be modified using the provided interfaces in the class.

Returns New `domain` instance with default properties

Return type *domain*

`add_load`(*newload*, *index=None*)

Adds load to interface

Add a load perturbation to the interface with the given index. If no index is provided, the load will be added to all interfaces. If the index is an integer, the load will be added to the interface with that index. Finally, the index can be an interable (list or tuple) of integers, and the load will be added to all interfaces in the iterable. Indices that are out of bounds or indicate an interface that is not frictional will raise an error. Default value is `None` (all interfaces).

`newload` must be a load perturbation (i.e. have type `load`), or the code will raise an error. `newload` will be appended to the load list

Parameters

- **`newload`** (`load`) – Load to be added
- **`index`** (*`int or tuple or list or None`*) – Interface to which the load should be added. Can be a single integer, iterable of integers, or `None` to add to all interfaces (default is `None`)

Returns None

`add_pert`(*newpert*, *index=None*)

Add new friction parameter perturbation to an interface

Method adds a frictional parameter perturbation to an interface. `newpert` must be a parameter perturbation of the correct kind for the given interface type (i.e. if the interface is of type `slipweak`, then `newpert` must have type `swparam`).

`index` indicates the index of the interface to which the perturbation will be added. `index` can be a single integer index, an iterable containing multiple indices, or `None` to add to all interfaces (default behavior is `None`). Out of bounds values will raise an error.

Parameters

- **newpert** (*pert (more precisely, one of the derived classes of friction parameter perturbations)*) – New perturbation to be added. Must have a type that matches the interface(s) in question.
- **index** (*int or list or tuple or None*) – Index of interface to which the perturbation will be added (single index or iterable of indices, or `None` for all interfaces, optional)

Returns None

check()

Checks domain for errors

No inputs, no return value, and the problem will not be modified.

This is run automatically when calling `write_input`. You may also run it manually to see if the problem contains self-consistent input values. Checks that all block corners match and all neighboring block edge grids conform.

Returns None

delete_block_surf(*coords*, *loc*)

Removes boundary surface for a particular block edge

Removes the bounding surface of a particular block edge. The block is selected by using `coords`, which is a tuple or list of 3 integers indicated block coordinates. Within that block, location is determined by `loc` which is an integer that indexes into a list. Locations correspond to the following: 0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top. Note that the location must be 0 <= loc < 2*ndim

If `coords` or `loc` is out of bounds, the code will also signal an error.

Parameters

- **coords** (*tuple or list*) – Coordaintes of desired block (tuple or list of 3 integers)
- **loc** (*int*) – Location of desired boundary to be removed (0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top). For 2D problems, `loc` must be between 0 and 3.

Returns None

delete_load(*niface*, *index=-1*)

Deletes load from index niface at position index from the list of loads

Deletes loads from a frictional interface. `niface` is an index refering to the desired interface. Out of bounds values or interfaces that are not frictional will result in an error.

`index` indicates the position in the load list that should be deleted. Default for `index` is `-1` (most recently added).

Parameters

- **`niface`** (*`int`*) – Interface from which the load should be removed. `niface` must refer to a frictional interface
- **`index`** (*`int`*) – Index within the load perturbation that should be removed (default is last)

Returns None

`delete_loadfile`(*niface*)

Deletes loadfile for given interface

Deletes the loadfile for the specified interface. `niface` is the index of the interface from which to delete the loadfile, and values that are not valid indices, or indices that refer to non-frictional interfaces will result in an error.

Parameters **`niface`** (*`int`*) – Index of interface for loadfile removal

Returns None

`delete_paramfile`(*niface*)

Deletes friction parameter file for given interface

Removes the friction parameter file for the interface with index `niface`. The interface in question must be a frictional interface that can accept parameter files.

Parameters **`niface`** (*`int`*) – Index of interface that will have its paramfile removed

Returns None

`delete_pert`(*niface*, *index=-1*)

Deletes frictional parameter perturbation from interface

`niface` is an integer indicating the index of the desired interface. If out of bounds, will give an error.

`index` is an integer that indicates the position within the list of loads. Default is most recently added (-1).

Parameters

- **`niface`** (*`int`*) – Index of interface from which to remove the parameter perturbation

- **`index`** (*`int`*) – Index within perturbation list of the given interface to remove. Default is last item (-1, or most recently added)

Returns None

`delete_statefile`(*niface*)

Deletes statefile for given interface

Delete the statefile for a given interface. `niface` must be a valid index that refers to an interface with a state variable. Will set the statefile attribute for the interface to None.

Parameters **`niface`** (*`int`*) – Index of interface that will have its statefile removed

Returns None

`find_nearest_point`(*point*, *known=None*, *knownloc=None*)

Finds the coordinate indices closest to a desired set of grid values

Method takes a set of grid values (tuple or list of 2 or 3 floats) and finds the indices of the grid point closest to that location (in terms of Euclidean distance). The method returns a set of coordinates (tuple of length 3 of integers) of point that is closest to the input point.

The method also allows you to search along a given interface. To do this, you must pass `known = 'x'` (or `'y'` or `'z'` depending on the normal direction of the interface) and the known index in `knownloc` (integer value, which does not necessarily need to be on an interface – it just fixes that coordinate when performing the search)

The location is found using an iterative binary search algorithm. The search begins along the x direction using binary search until the distance to the desired point's x coordinate is minimized. The search then proceeds in the y and z directions. The algorithm then searches again in the x direction, y direction, and z direction, until the coordinates do not change over an entire iteration. This iteration procedure needs to take place because the coordinate directions are not independent. The algorithm is usually fairly efficient and finds coordinates fairly quickly.

Parameters

- **`point`** (*`tuple or list`*) – Desired spatial location (tuple or list of floats)
- **`known`** (*`str or None`*) – Spatial direction to fix during search (optional, string)
- **`knownloc`** (*`int or None`*) – Fixed coordinate value along `known` direction (optional, integer)

Returns Closest spatial coordinate (tuple of 3 integers)

Return type tuple

get_block_lx(*coords*)

Returns physical size of a block with a given set of coordinates. Note that this assumes the block is rectangular. It can be overridden by setting the edge of a block to be a curve (2D) or surface (3D), so this is not always the definitive size of a block.

Parameters **coords** (*tuple or list*) – Coordinates of desired block (tuple or list of three integers)

Returns Dimensions of desired block (tuple of three floats)

Return type tuple

get_block_surf(*coords*, *loc*)

Returns block boundary surface for a block edge

Returns the surface assigned to a specific block along a specific edge. The block is chosen using `coords` which is a tuple or list of 3 positive integers that corresponds to the coordinates of the block. Within that block, `loc` determines the edge that is returned (integer, corresponding to an index). Location indices correspond to the following: 0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top Note that the location must be 0 <= loc < 2*ndim (for 2D problems, `loc` cannot be 5 or 6).

Returns either a curve (2D problems) or surface (3D problems) or None

If `coords` or `loc` indices are out of bounds, the code will raise an error.

Parameters

- **coords** (*tuple or list*) – Coordaintes of desired block (tuple or list of 3 integers)
- **loc** (*int*) – Location of desired boundary (0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top). For 2D problems, `loc` must be between 0 and 3.

Returns curve or surface corresponding to the selected block and location. If the desired edge does not have a bounding surface, returns None.

Return type curve or surface or None

get_block_xm(*coords*)

Returns starting index (zero-indexed) of each block (list of three lists of integers)

Parameters **coords** (*tuple or list*) – Coordinates of desired block (tuple or list of three integers)

Returns list of three lists (each list is a list of integers)

Return type list

get_bm(*index*)

Returns block in minus direction of interface index. Returns a tuple of 3 integers indicating block coordinates of target block

Parameters **`index`** (*`int`*) – index of desired interface (zero-indexed)

Returns tuple

`get_bounds`(*coords*, *loc=None*)

Returns boundary types of a particular block. If `loc` (int) is provided, the method returns a specific location (str). Otherwise it returns a list of all boundaries, which will have length 4 for 2D problems and length 6 for 3D problems. `loc` serves effectively as an index into the list, and the indices correspond to the following: 0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top. Note that the location must be 0 <= loc < 2*ndim

Parameters

- **`coords`** (*`tuple`*) – Block coordinate location (list or tuple of three integers)
- **`loc`** (*`int or None`*) – Location of boundary that is desired (optional). If `loc` is not provided, returns a list

Returns Boundary type (if `loc` provided, returns a string of the boundary type for the desired location, of not returns a list of strings indicating all boundary types)

Return type str or list

`get_bp`(*index*)

Returns block in plus direction of interface index. Returns a tuple of 3 integers indicating block coordinates of target block

Parameters **`index`** (*`int`*) – index of desired interface (zero-indexed)

Returns tuple

`get_cdiss`()

Returns artificial dissipation coefficient

Returns Artificial dissipation coefficient

Return type float

`get_direction`(*index*)

Returns direction (formally, normal direction in computational space) of interface with given index Returns a string 'x', 'y', or 'z', which is the normal direction for a simulation with rectangular blocks

Parameters **`index`** (*`int`*) – index of desired interface (zero-indexed)

Returns str

`get_het_material`()

Returns heterogeneous material properties for simulation

Returns a numpy array with shape `(3,nx,ny,nz)`. First index indicates parameter value (0 = density, 1 = Lame parameter, 2 = Shear modulus). The other three indicate grid coordinates. If no heterogeneous material parameters are specified, returns `None`

Returns ndarray

`get_het_stress``()`

Returns heterogeneous stress initial values.

Returns a numpy array with shape `(ns,nx,ny,nz)`. First index indicates stress component. The following three indices indicate grid coordinates.If no array is currently specified, returns `None`.

For 2D mode 3 problems, indices for `ns` are (0 = sxz, 1 = syz)

For elastic 2D mode 2 problems, indices for `ns` are (0 = sxx, 1 = sxy, 2 = syy). For plastic 2D mode 2 problems, indices for `ns` are (0 = sxx, 1 = sxy, 2 = syy, 3 = szz).

For 3D problems, indices for `ns` are (0 = sxx, 1 = sxy, 2 = sxz, 3 = syy, 4 = syz, 5 = szz)

Returns ndarray

`get_iftype``(`*index=None*`)`

Returns interface type of given index, if none provided returns full list

Parameters **`index`** (*`int`*) – (optional) index of desired interface (zero-indexed). If not given or if `None` is given the entire list of interface types is returned

Returns str or list

`get_load``(`*niface*, *index=None*`)`

Returns load for index niface at position index. If no index provided, returns entire list of perturbations

Parameters

- **`niface`** (*`int`*) – index of desire interface (zero-indexed)
- **`index`** (*`int`*) – (optional) index of perturbation. If not provided or None, then returns entire list

Returns load or list

`get_loadfile``(`*niface*`)`

Returns loadfile for interface with index niface

Loadfile sets any surface tractions set for the particular interface in question. Note that these tractions are added to any any set by the constant initial stress tensor, initial heterogeneous stress, or interface traction perturbations

Parameters **`niface`** – index of desired interface (zero-indexed)

Returns Current loadfile for the interface (if the interface does not have a loadfile, returns None)

Return type loadfile or None

get_material(*coords*)

Returns material properties for a given block

Returns the material class associated with block with coordinates `coords`. `coords` must be a tuple or list of valid block indices

Parameters **coords** (*tuple or list*) – Coordinates of the target block (tuple or list of 3 nonnegative integers)

Returns Material class with properties for this block

Return type *material*

get_mattype()

Returns material type ('elastic' or 'plasitc')

Returns Material type

Return type str

get_mode()

Returns rupture mode (2 or 3), only valid for 2D problems (stored at domain level)

Returns Rupture mode

Return type int

get_nblocks()

Returns number of blocks points in (nx, ny, nz) format

Returns Number of blocks (tuple of three integers)

Return type tuple

get_nblocks_tot()

Returns total number of blocks

Returns Total number of blocks

Return type int

get_ndim()

Returns Number of spatial dimensions

Returns Number of spatial dimensions

Return type int

get_nifaces()

Returns number of interfaces

Returns Number of interfaces

Return type int

get_nloads(*index*)

Returns number of loads on interface with given index

Parameters **index** – index of desire interface (zero-indexed)

Returns int

get_nperts(*index*)

Returns number of perturbations (integer) on given interface with given index

Parameters **index** (*int*) – index of desired interface (zero-indexed)

Returns int

get_nproc()

Returns number of processes (in x, y, z directions).

0 means MPI will do the domain decomposition in that direction automatically

Returns Number of processes in each spatial dimension (x, y, z) (tuple of three integers)

Return type tuple

get_nx()

Returns number of grid points in (nx, ny, nz) format

Returns Number of grid points (tuple of three integers)

Return type tuple

get_nx_block()

Returns number of grid points in each block along each spatial dimension

Returns Number of grid points in each block (list of three lists)

Return type list

get_paramfile(*niface*)

Returns paramfile (holds arrays of heterogeneous friction parameters) for interface with index niface. Can return a subtype of paramfile corresponding to any of the specific friction law types.

Parameters **niface** (*int*) – index of desired interface (zero-indexed)

Returns paramfile

get_pert(*niface*, *index=None*)

Returns perturbation for index niface at position index

Method returns a perturbation from a particular interface. `niface` must be a valid integer index referring to an interface. `index` is the index into the perturbation list for the particular index. If `index` is not provided or is `None`, the method returns the entire list.

Parameters

- **`niface`** (*`int`*) – Index referring to an interface. (Must be a valid integer index.)
- **`index`** (*`int or None`*) – Index into the perturbation list for the index in question (optional, if not provided or `None`, then returns entire list)

Returns pert or list

`get_sbporder``()`

Returns order of accuracy of finite difference method (stored at domain level)

Returns Order of accuracy of finite difference method

Return type int

`get_state``(`*niface*`)`

Returns initial state variable value for interface with index niface

Parameters **`niface`** – index of desired interface (zero-indexed)

Returns Initial state variable

Return type float

`get_statefile``(`*niface*`)`

Returns state file of given interface

If interface does not have a statefile returns None

Parameters **`niface`** – index of desired interface (zero-indexed)

Returns statefile or None

`get_stress``()`

Returns uniform intial stress values

Note that 2D simulations do not use all stress components. Mode 2 elastic simulations only use `sxx`, `sxy`, and `syy`, and mode 3 elastic simulations use `sxz`, and `syz` (though the normal stresses `sxx` and `syy` can be set to constant values that are applied to any frictional failure criteria). Mode 2 plastic simulations use `szz`, and mode 3 plastic simulations use all three normal stress components in evaluating the yield criterion.

Returns Initial stress tensor (list of floats). Format is `[sxx,sxy,sxz,syy,syz,szz]`

Return type list

get_x(*coord*)

Returns grid value for given spatial index

For a given problem set up, returns the location of a particular set of coordinate indices. Note that since blocks are set up by setting values only on the edges, coordinates on the interior are not specified *a priori* and instead determined using transfinite interpolation to generate a regular grid on the block interiors. Calling `get_x` generates the interior grid to find the coordinates of the desired point.

Within each call to `get_x`, the grid is generated on the fly only for the relevant block where the desired point is located. It is not stored. This helps reduce memory requirements for large 3D problems (since the Python module does not run in parallel), but is slower. Because the computational grid is regular, though, it can be done in a single step in closed form.

Returns a numpy array of length 3 holding the spatial location (x, y, z).

Parameters **coord** (*tuple or list*) – Spatial coordinate where grid values are desired (tuple or list of 3 integers)

Returns (x, y, z) coordinates of spatial location

Return type ndarray

set_block_lx(*coords*, *lx*)

Sets block with coordinates `coords` to have dimension `lx`

`coords` is a tuple of nonnegative integers that indicates the coordinates of the desired block (0-indexed, must be less than the number of blocks in that particular direction or the code will raise an error). `lx` is a tuple of two (2D) or three (3D) positive floats indicating the block length in each spatial dimension. Note that this assumes each block is rectangular. When a single block is modified, the code automatically adjusts the lower left corner of all simulation blocks to be consistent with this change.

This can be overridden by setting a block edge to be a curve (2D) or surface (3D). However, traction and friction parameter perturbations still make use of these block lengths when altering interface tractions or friction parameters. More information on how this works is provided in the `pert` documentation.

Finally, note that neighboring blocks must have conforming grids. When writing simulation data to file, the code checks that all interfacial grids match, and raises an error if it disagrees. So while the `set_block_lx` method may not complain about an error like this, you will not be able to save the simulation to a file with such an error.

Parameters

- **coords** (*tuple or list*) – Coordinates (tuple or list of 3 nonnegative integers)
- **lx** (*tuple or list*) – New dimensions of desired block (tuple or list of 2 or 3 positive floats)

Returns None

set_block_surf(*coords*, *loc*, *surf*)

Sets boundary surface for a particular block edge

Changes the bounding surface of a particular block edge. The block is selected by using `coords`, which is a tuple or list of 3 integers indicated block coordinates. Within that block, location is determined by `loc` which is an integer that indexes into a list. Locations correspond to the following: 0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top. Note that the location must be 0 <= loc < 2*ndim

For 2D problems, `surf` must be a curve. For 3D problems, `surf` must be a surface. Other choices will raise an error. If `coords` or `loc` is out of bounds, the code will also signal an error.

Parameters

- **coords** (*tuple or list*) – Coordaintes of desired block (tuple or list of 3 integers)
- **loc** (*int*) – Location of desired boundary (0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top). For 2D problems, `loc` must be between 0 and 3.
- **surf** (*curve or surface*) – curve or surface corresponding to the selected block and location

Returns None

set_bounds(*coords*, *bounds*, *loc=None*)

Sets boundary types of a particular block.

Changes the type of boundary conditions on a block. Acceptable values are ‘absorbing’ (incoming wave amplitude set to zero), ‘free’ (no traction on boundary), ‘rigid’ (no displacement of boundary), or ‘none’ (boundary conditions set by imposing interface conditions).

The block to be modified is determined by `coords`, which is a tuple or list of 3 integers that match the coordinates of a block.

There are two ways to use `set_bounds`:

1. Set `loc` to be `None` (default) and provide a list of strings specifying boundary type for `bounds`. The length of `bounds` is 4 for a 2D simulation and 6 for 3D.
2. Set `loc` to be an integer denoting location and give `bounds` as a single string. The possible locations correspond to the following: 0 = left, 1 = right, 2 = front, 3 = back, 4 = bottom, 5 = top. 4 and 5 are only applicable to 3D simulations (0 <= loc < 2*ndim).

Parameters

- **coords** (*tuple or list*) – Location of block to be modifies (tuple or list of 3 integers)
- **bounds** (*str or list*) – New boundary condition type (string or list of strings)
- **loc** (*int or None*) – If provided, only change one type of boundary condition rather than all (optional, loc serves as an index into the list if used)

Returns None

set_cdiss(*cdiss*)

Sets artificial dissipation coefficient

New artificial dissipation coefficient must be nonnegative. If it is set to zero, the code will not use artificial dissipation in the simulation.

There is not a hard and fast rule for setting the coefficient, so some degree of trial and error may be necessary. Values around 0.1 have worked well in the past, but that may not be true for all meshes.

Parameters **cdiss** (*float*) – New artificial dissipation coefficient

Returns None

set_domain_xm(*xm*)

Sets lower left corner of domain to spatial coordinate xm

Moves the lower left corner of the simulation. This does not affect block lengths, only the minimum spatial location of the entire comain in each cartesian direction. Individual block locations are calculated automatically from this and the length information for each block. Thus, you cannot set the location of each block directly, just the overall value of the domain and then all other blocks are positioned based on the length of other blocks.

If the simulation is 2D and a nonzero value for the z-coordinate is provided, the z position of all blocks will be automatically set to zero.

Note that the location of any block can be overridden by setting the edges to be surfaces. The corners must still match one another (this is checked when writing the simulation data to file), and neighboring blocks must have conforming grids at the edges.

Parameters **xm** (*tuple or list*) – New lower left coordinate of simulation domain (tuple of 2 or 3 floats)

Returns None

set_het_material(*mat*)

Sets heterogeneous material properties for simulation

New heterogeneous material properties must be a numpy array with shape `(3,nx,ny,nz)`. First index indicates parameter value (0 = density, 1 = Lame parameter, 2 = Shear modulus). The other three indicate grid coordinates

An array with the wrong shape will result in an error.

Parameters **`mat`** (*`ndarray`*) – New material properties array (numpy array with shape `(3,nx,ny,nz)`)

Returns None

`set_het_stress`(*s*)

Sets heterogeneous stress initial values

Sets initial heterogeneous stress. New stress must be a numpy array with shape `(ns,nx,ny,nz)`. First index indicates stress component. The following three indices indicate grid coordinates.

For 2D mode 3 problems, indices for `ns` are (0 = sxz, 1 = syz)

For elastic 2D mode 2 problems, indices for `ns` are (0 = sxx, 1 = sxy, 2 = syy). For plastic 2D mode 2 problems, indices for `ns` are (0 = sxx, 1 = sxy, 2 = syy, 3 = szz).

For 3D problems, indices for `ns` are (0 = sxx, 1 = sxy, 2 = sxz, 3 = syy, 4 = syz, 5 = szz)

Providing arrays of the incorrect size will result in an error.

Parameters **`s`** (*`ndarray`*) – New heterogeneous stress array (numpy array with shape `(3,nx,ny,nz)`

Returns None

`set_iftype`(*index*, *iftype*)

Sets type of interface with a given index

Changes type of a particular interface. `index` is the index of the interface to be modified and `iftype` is a string denoting the interface type. Valid values for `iftype` are `'locked'`, `'frictionless'`, `'slipweak'`, and `'stz'`. Any other values will result in an error, as will an interface index that is out of bounds.

Parameters

- **`index`** (*`int`*) – Index (nonnegative integer) of interface to be modified
- **`iftype`** (*`str`*) – New interface type (see valid values above)

Returns None

`set_loadfile`(*niface*, *newloadfile*)

Sets loadfile for interface with index niface

`niface` indicates the index of the interface that will be modified, and must be a frictional interface. `newloadfile` is the new loadfile (must have type `loadfile`). If the index is bad or the loadfile type is not correct, the code will raise an error. Errors can also result if the shape of the loadfile does not match with the interface.

Parameters

- **niface** – index of desired interface (zero-indexed)
- **newloadfile** (`loadfile`) – New loadfile to be used for the given interface

Returns None

set_material(*newmaterial*, *coords=None*)

Sets block material properties for the block with indices given by `coords`

If `coords` is not provided, all blocks are changed to have the given material properties. `newmaterial` must have a type `material` and `coords` must be a tuple or list of three integers that match the coordinates of a block.

If `set_material` changes all blocks in the simulation, it also changes the material type for the whole simulation (equivalent to calling `set_mattype`). If `set_material` acts only on a single block, the new material type of that block must match the one set in the `fields` type (i.e. the return value of `get_mattype`).

Parameters

- **newmaterial** (`material`) – New material properties
- **coords** (*tuple or list*) – Coordinates of block to be changed (optional, omitting changes all blocks). `coords` must be a tuple or list of three integers that match the coordinates of a block.

Returns None

set_mattype(*mattype*)

Sets field and block material type ('elastic' or 'plastic')

Sets the material type for the simulation. Options are 'elastic' for an elastic simulation and 'plastic' for a plastic simulation. Anything else besides these options will cause the code to raise an error.

Once the simulation type is altered, all blocks material types are changed as well. This is necessary to ensure that the right set of parameters are written to file. Note that all blocks must therefore have the same material type, though you can ensure that a given block always behaves elastically by setting an appropriate value for the yield criterion.

Parameters **mattype** (*str*) – New material type ('elastic' or 'plastic')

Returns None

set_mode(*mode*)

Sets rupture mode

Rupture mode is only valid for 2D problems, and is either 2 or 3 (other values will cause an error, and non-integer values will be converted to integers). For 3D problems, entering a different value of the rupture mode will alter the rupture mode cosmetically but will have no effect on the simulation.

Parameters **`mode`** (*`int`*) – New value of rupture mode

Returns None

`set_nblocks`(*nblocks*)

Sets number of blocks

`set_nblocks` alters the number of blocks in the simulation. The method adds or deletes blocks from the list of blocks as needed. Depending on how the number of blocks is changed, new blocks may only have a single grid point, or if added in a direction where the number of blocks is already established the number of grid points may be copied from the existing simulation. If in doubt, use `get_nx_block` to check the number of grid points and use `set_nx_block` to modify if necessary.

Parameters **`nblocks`** (*`tuple`*) – New number of blocks (tuple of 3 positive integers)

Returns None

`set_ndim`(*ndim*)

Sets number of dimensions

The new number of spatial dimensions must be an integer, either 2 or 3. If a different value is given, the code will raise an error. If a non-integer value is given that is acceptable, the code will convert it to an integer.

Note: Converting a 3D problem into a 2D problem will automatically collapse the number of grid points and the number of blocks in the z direction to be 1. Any modifications to these quantities that were done previously will be lost.

Parameters **`ndim`** (*`int`*) – New value for ndim (must be 2 or 3)

Returns None

`set_nproc`(*nproc*)

Sets number of processes in domain decomposition manually

New number of processes `nproc` must be a tuple/list of nonnegative integers. If the problem is 2D, the number of processes in the z direction will automatically be set to 1. Any number can be set to zero, in which case MPI will set the number of processes in that direction automatically. If all three numbers are nonzero, then it is up to the user to ensure that the total number of processors ($nx imes ny imes nz$) is the same as the total number when running the executable.

Parameters **`nproc`** (*`tuple`*) – New number of processes (must be a tuple of positive integers)

Returns None

set_nx_block(*nx_block*)

Set number of grid points in each block as a list of lists.

Input must be a list or tuple of length 3, with each item a list of integers representing the number of grid points for each block along the respective dimension. If the list lengths do not match the number of blocks, the code will raise an error. The blocks must form a regular cartesian grid with conforming edges, so all blocks along a single spatial dimension must have the same number of grid points along that spatial dimension.

For example, if nblocks = (3,2,1), then nblock[0] has length 3, nblock[1] has length 2, and nblock[2] has length 1. All blocks that are at position 0 in the x-direction will have nblock[0][0] grid points in the x-direction, all blocks at position 1 in the x-direction will have nblock[0][1] grid points in the x-direction, etc.

Parameters **nx_block** (*list*) – New number of grid points (list of 3 lists of positive integers)

Returns None

set_paramfile(*niface*, *newparamfile*)

Sets paramfile for given interface

Method sets the file holding frictional parameters for an interface. Interface to be modified is set by `niface`, which must be a valid index for an interface.

`newparamfile` must be a parameter perturbation file of the correct type for the given interface type (i.e. if the interface is of type `slipweak`, then `newpert` must have type `swparamfile`). Errors can also result if the shape of the paramfile does not match with the interface.

Parameters

- **niface** (*int*) – index of desired interface (zero-indexed)
- **newparamfile** (*paramfile (actual type must be the appropriate subclass for the friction law of the particular interface and have the right shape)*) – New frictional parameter file (type depends on interface in question)

Returns None

set_sbporder(*sbporder*)

Sets finite difference order

Finite difference method order must be an integer 2-4. A value outside of this range will result in an error. If a non-integer value is given that is acceptable, it will be converted to an integer and there will be no error message.

Parameters **sbporder** (*int*) – New value of finite difference method order (integer 2-4)

Returns None

set_state (*niface*, *state*)

Sets initial state variable for interface

Set the initial value for the state variable for a given interface. `niface` is the index of the interface to be set (must be a valid integer index). The interface must have a state variable associated with it, or an error will occur. `state` is the new state variable (must be a float or some other valid number).

Parameters

- **niface** (*int*) – Index of interface to modify. Must be an interface with a state variable
- **state** (*float*) – New value of state variable

Returns None

set_statefile (*niface*, *newstatefile*)

Sets state file for interface

Set the statefile for the indicated interface. `niface` must be a valid index to an interface, out of bounds values will lead to an error. `newstatefile``must have type ``statefile` and the interface must support a state variable. Errors can also result if the shape of the statefile does not match with the interface.

Parameters

- **niface** (*int*) – Index of interface to be modified
- **newstatefile** (`statefile`) – New statefile for the interface in question.

Returns None

set_stress (*s*)

Sets uniform intial stress

Changes initial uniform stress tensor. New stress tensor must be a list of six floats.

Note that 2D simulations do not use all stress components. Mode 2 elastic simulations only use `sxx`, `sxy`, and `syy`, and mode 3 elastic simulations use `sxz`, and `syz` (though the normal stresses `sxx` and `syy` can be set to constant values that are applied to any frictional failure criteria). Mode 2 plastic simulations use `szz`, and mode 3 plastic simulations use all three normal stress components in evaluating the yield criterion.

Params s New stress tensor (list of 6 floats). Format is `[sxx,sxy,sxz,syy,syz,szz]`

Returns None

write_input (*f*, *probname*, *directory*, *endian='='*)
Writes domain information to input file

Method writes the current state of a domain to an input file, also writing any binary data to file (i.e. block boundary curves, heterogeneous stress tensors, heterogeneous material properties, heterogeneous interface tractions, heterogeneous state variables, or heterogeneous friction parameters).

Arguments include the input file `f` (file handle), problem name `probname` (string), output directory `directory,and endianness of binary files ``endian`. `endian` has a default value of = (native), other options inlcude < (little) and > (big).

When `write_input` is called, the code calls `check`, which verifies the validity of the simulation and alerts the user to any problems. `check` examines if block surface edges match, if neighboring blocks have matching grids, and other things that cannot be checked when modifying the simulation. The same checks are run in the C++ code when initializing a problem, so a problem that runs into trouble when calling `check` is likely to have similar difficulties when running the simulation.

Parameters

- **f** (*file*) – file handle for text input file
- **probname** (*str*) – problem name (used for any binary files)
- **directory** (*str*) – Location where input file should be written
- **endian** – Byte-ordering for files. Should match byte ordering of the system where the simulation will be run (it helps to run the Python script directly with native byte ordering enabled). Default is native (=), other options include < for little endian and > for big endian.

Returns None

The `fields` Class

The `fields` class holds information regarding initial conditions and material properties. This includes the initial stresses (which can be spatially uniform or spatially heterogeneous) as well as heterogeneous elastic properties of the medium. While this information more accurately exists at the block level, because of how the C++ code is parallelized there are some performance benefits to placing all grid data in a single array to facilitate data sharing between processors.

The user does not typically interact with `fields` objects, as one is created automatically when initializing a domain. The `problem` class contains wrapper functions that perform any relevant modifications of the fields for a given simulation. Documentation is provided here for completeness.

class `fdfault.`**`fields`**(*ndim*, *mode*)

Class representing fields in a dynamic rupture problem

When initializing a fields object, one is created holding default attributes, including

Variables

- **`ndim`** (*`int`*) – Number of spatial dimensions (2 or 3; default is 2)
- **`mode`** (*`int`*) – Rupture mode (2 or 3, default is 2; only relevant for 2D problems)
- ***`material`*** (*`str`*) – Simulation material type (`'elastic'` or `'plastic'`, default is `'elastic'`)
- **`s0`** (*`list`*) – Initial stress tensor (list of floats, ordering is `[sxx,sxy,sxz,syy,syz,szz]`, default is zero for all components). All components must be specified for any problem type, however, the code will only refer to the relevant values. Note that for mode 3 problems, the in-plane normal stresses `sxx` and `syy` will be used to determine the normal stress on any faults in the simulation, even though the normal stresses do not change during the simulation.

`__init__`(*ndim*, *mode*)

Initializes an instance of the `fields` class

Creates a new instance of the `fields` class. Attributes required to create a new instance is the number of dimensions and rupture mode. By default, the new `fields` instance is an elastic simulation with a stress tensor initialized to zero. The simulation also does not have a heterogeneous stress or heterogeneous material properties.

Parameters

- **`ndim`** (*`int`*) – Number of dimensions in the simulation (must be 2 or 3)
- **`mode`** (*`int`*) – Rupture mode (2 or 3, only relevant for 2D problems)

Returns New instance of the fields class

Return type *fields*

`get_het_material`()

Returns heterogeneous material properties for simulation

Returns a numpy array with shape `(3,nx,ny,nz)`. First index indicates parameter value (0 = density, 1 = Lame parameter, 2 = Shear modulus). The other three indicate grid coordinates. If no heterogeneous material parameters are specified, returns `None`

Returns ndarray

`get_het_stress`()

Returns heterogeneous stress initial values.

Returns a numpy array with shape `(ns,nx,ny,nz)`. First index indicates stress component. The following three indices indicate grid coordinates.If no array is currently specified, returns `None`.

For 2D mode 3 problems, indices for `ns` are (0 = sxz, 1 = syz)

For elastic 2D mode 2 problems, indices for `ns` are (0 = sxx, 1 = sxy, 2 = syy). For plastic 2D mode 2 problems, indices for `ns` are (0 = sxx, 1 = sxy, 2 = syy, 3 = szz).

For 3D problems, indices for `ns` are (0 = sxx, 1 = sxy, 2 = sxz, 3 = syy, 4 = syz, 5 = szz)

Returns ndarray

`get_material``()`

Returns material type ('elastic' or 'plasitc')

Returns Material type

Return type str

`get_stress``()`

Returns uniform intial stress values

Note that 2D simulations do not use all stress components. Mode 2 elastic simulations only use `sxx`, `sxy`, and `syy`, and mode 3 elastic simulations use `sxz`, and `syz` (though the normal stresses `sxx` and `syy` can be set to constant values that are applied to any frictional failure criteria). Mode 2 plastic simulations use `szz`, and mode 3 plastic simulations use all three normal stress components in evaluating the yield criterion.

Returns Initial stress tensor (list of floats). Format is `[sxx,sxy,sxz,syy,syz,szz]`

Return type list

`set_het_material``(`*mat*`)`

Sets heterogeneous material properties for simulation

New heterogeneous material properties must be a numpy array with shape `(3,nx,ny,nz)`. First index indicates parameter value (0 = density, 1 = Lame parameter, 2 = Shear modulus). The other three indicate grid coordinates

An array with the wrong shape will result in an error.

Parameters **`mat`** (*`ndarray`*) – New material properties array (numpy array with shape `(3,nx,ny,nz)`)

Returns None

`set_het_stress``(`*s*`)`

Sets heterogeneous stress initial values

Sets initial heterogeneous stress. New stress must be a numpy array with shape `(ns,nx,ny,nz)`. First index indicates stress component. The following three indices indicate grid coordinates.

For 2D mode 3 problems, indices for `ns` are (0 = sxz, 1 = syz)

For elastic 2D mode 2 problems, indices for `ns` are (0 = sxx, 1 = sxy, 2 = syy). For plastic 2D mode 2 problems, indices for `ns` are (0 = sxx, 1 = sxy, 2 = syy, 3 = szz).

For 3D problems, indices for `ns` are (0 = sxx, 1 = sxy, 2 = sxz, 3 = syy, 4 = syz, 5 = szz)

Providing arrays of the incorrect size will result in an error.

Parameters **`s`** (*`ndarray`*) – New heterogeneous stress array (numpy array with shape `(3,nx,ny,nz)`

Returns None

`set_material`(*material*)
Sets field and block material type ('elastic' or 'plastic')

Sets the material type for the simulation. Options are 'elastic' for an elastic simulation and 'plastic' for a plastic simulation. Anything else besides these options will cause the code to raise an error.

Parameters **`mattype`** (*`str`*) – New material type ('elastic' or 'plastic')

Returns None

`set_stress`(*s*)
Sets uniform intial stress

Changes initial uniform stress tensor. New stress tensor must be a list of six floats.

Note that 2D simulations do not use all stress components. Mode 2 elastic simulations only use `sxx`, `sxy`, and `syy`, and mode 3 elastic simulations use `sxz`, and `syz` (though the normal stresses `sxx` and `syy` can be set to constant values that are applied to any frictional failure criteria). Mode 2 plastic simulations use `szz`, and mode 3 plastic simulations use all three normal stress components in evaluating the yield criterion.

Params **s** New stress tensor (list of 6 floats). Format is `[sxx,sxy,sxz,syy,syz,szz]`

Returns None

`write_input`(*f*, *probname*, *directory*, *endian='='*)
Writes field information to input file

Method writes the current state of a domain to an input file, also writing any binary data to file (i.e. block boundary curves, heterogeneous stress tensors, heterogeneous material properties, heterogeneous interface tractions, heterogeneous state variables, or heterogeneous friction parameters).

Arguments include the input file `f` (file handle), problem name `probname` (string), output directory `directory,and endianness of binary files ``endian`. `endian` has a default value of = (native), other options inlcude < (little) and > (big).

Parameters

- **`f`** (*`file`*) – file handle for text input file
- **`probname`** (*`str`*) – problem name (used for any binary files)
- **`directory`** (*`str`*) – Location where input file should be written
- **`endian`** – Byte-ordering for files. Should match byte ordering of the system where the simulation will be run (it helps to run the Python script directly with native byte ordering enabled). Default is native (=), other options include < for little endian and > for big endian.

Returns None

The `front` Class

The `front` class holds information about rupture time output. Rupture times are found by recording the earliest time at which a given field (slip or slip rate) exceeds a threshold value.

An instance of `front` is automatically generated for all problems, so the user should not use this directly, but rather modify the front for the problem using the provided interfaces in the `problem` class.

class `fdfault.`**`front`**

Class holding information regarding rupture front output.

Relevant internal variables include:

Variables

- **`out`** – Boolean indicating if front output is on/off (Default `False`)
- **`field`** – Field to use for determining rupture time (rupture time is earliest time this field exceeds the threshold). Default is `'V'` (scalar slip velocity), can also be `'U'` (slip).
- **`value`** – Threshold value (default is 0.001)

`__init__``()`

Initializes rupture front

Create a new instance of the `front` class with default properties (output is `False`, output field is `'V'`, and output threshold value is `0.001`).

Returns New rupture front class:

Return type *front*

get_field()
Returns front field

Returns Rupture front field (string, "U" denotes slip and "V" denotes slip velocity)

Return type str

get_output()
Returns status of front output (boolean)

Returns Status of front output

Return type bool

get_value()
Returns front threshold value. Front output is the earliest time at which the given field exceeds this value

Returns Threshold value for rupture front output

Return type float

set_field(*field*)
Sets rupture front field

Sets new value of rupture front field `field`. `field` must be a string (slip (`'U'`) or slip velocity (`'V'`)). Other choices will raise an error.

Parameters **field** (*str*) – New rupture front field

Returns None

set_output(*newoutput*)
Sets front output to be on or off

Sets rupture front output to be the specified value (boolean). Will raise an error if the provided value cannot be converted into a boolean.

Parameters **newoutput** (*bool*) – New value of output

Returns None

set_value(*value*)
Sets front threshold value

Changes value of rupture front threshold. The rupture time is the earliest time at which the chosen field exceeds this value. `value` is the new value (must be a positive number).

Parameters **`value`** (*`float`*) – New values of the threshold for rupture front times.

Returns None

`write_input` (*f*)

Writes front information to input file

Method writes the current state of a front to an input file.

Input argument the input file `f` (file handle).

Parameters **`f`** (*`file`*) – file handle for text input file

Returns None

The `interface` Class

The `interface` class and its derived classes describe interfaces that link neighboring blocks together. The code includes several types of interfaces: the standard `interface` class is for a locked interface where no relative slip is allowed between the neighboring blocks. Other interface types allow for frictional slip following several possible constitutive friction laws. The other types are derived from the main `interface` class and thus inherit much of their functionality.

The `interface` class will not usually be invoked directly. This is because interfaces are created automatically based on the number of blocks in the simulation. When the user changes the number of blocks in the simulation, locked interfaces are automatically created between all neighboring blocks. To modify the type of interface, it is preferred to use the `set_iftype` method of a problem to ensure that only the correct interfaces remain in the simulation.

Other interface types include: `friction`, which describes frictionless interfaces; `paramfric`, which is a generic class for interfaces with parameters describing their behavior; `statefric`, which is a generic class for friction laws with a state variable; `slipweak`, which describes slip weakening and kinematically forced rupture interfaces; and `stz`, which describes friction laws governed by Shear Transformation Zone Theory. As with basic interfaces, none of these will be invoked directly, and `paramfric` and `statefric` only create template methods for the generic behavior of the corresponding type of interfaces and thus are not used in setting up a problem.

class `fdfault.`**`interface`**(*ndim*, *index*, *direction*, *bm*, *bp*)

Class representing a locked interface between blocks

This is the parent class of all other interfaces. The `interface` class describes locked interfaces, while other interfaces require additional information to describe how relative slip can occur between the blocks.

Interfaces have the following attributes:

Variables

- **`ndim`** – Number of dimensions in problem (2 or 3)

- **`iftype`** – Type of interface ('locked' for all standard interfaces)
- **`index`** – index of interface (used for identification purposes only, order is irrelevant in simulation)
- **`bm`** – Indices of block in the "minus" direction (tuple of 3 integers)
- **`bp`** – Indices of block in the "plus" direction (tuple of 3 integers)
- **`direction`** – Normal direction in computational space ("x", "y", or "z")

`__init__`(*ndim*, *index*, *direction*, *bm*, *bp*)

Initializes an instance of the `interface` class

Create a new `interface` given an index, direction, and block coordinates.

Parameters

- **`ndim`** (*`int`*) – Number of spatial dimensions (must be 2 or 3)
- **`index`** (*`int`*) – Interface index, used for bookkeeping purposes, must be nonnegative
- **`direction`** (*`str`*) – String indicating normal direction of interface in computational space, must be `'x'`, `'y'`, or `'z'`, with `'z'` only allowed for 3D problems)
- **`bm`** (*`tuple`*) – Coordinates of block in minus direction (tuple of length 3 of integers)
- **`bp`** (*`tuple`*) – Coordinates of block in plus direction (tuple of length 3 or integers, must differ from `bm` by 1 only along the given direction to ensure blocks are neighboring one another)

Returns New instance of interface class

Return type *interface*

`add_load`(*newload*)

Adds a load to list of load perturbations

Method adds the load provided to the list of load perturbations. If the `newload` parameter is not a load perturbation, this will result in an error.

Parameters **`newload`** (`load`) – New load to be added to the interface (must have type `load`)

Returns None

`add_pert`(*newpert*)

Add new friction parameter perturbation to an interface

Method adds a frictional parameter perturbation to an interface. `newpert` must be a parameter perturbation of the correct kind for the given interface type (i.e. if the interface is of type `slipweak`, then `newpert` must have type `swparam`).

Parameters **`newpert`** (*`pert (more precisely, one of the derived classes of friction parameter perturbations)`*) – New perturbation to be added. Must have a type that matches the interface(s) in question.

Returns None

`delete_load`(*index=-1*)

Deletes load at position index from the list of loads

Method deletes the load from the list of loads at position `index`. Default is most recently added load if an index is not provided. `index` must be a valid index into the list of loads.

Parameters **`index`** (*`int`*) – Position within load list to remove (optional, default is -1)

Returns None

`delete_loadfile`()

Deletes the loadfile for the interface.

Returns None

`delete_paramfile`()

Deletes friction parameter file for the interface

Removes the friction parameter file for the interface. The interface must be a frictional interface that can accept parameter files.

Returns None

`delete_pert`(*index=-1*)

Deletes frictional parameter perturbation from interface

`index` is an integer that indicates the position within the list of perturbations. Default is most recently added (-1).

Parameters **`index`** (*`int`*) – Index within perturbation list of the given interface to remove. Default is last item (-1, or most recently added)

Returns None

`get_bm`()

Returns block on negative side

Returns tuple of block indices on negative size

Returns Block indices on negative side (tuple of integers)

Return type tuple

get_bp()
Returns block on positive side

Returns tuple of block indices on positive size

Returns Block indices on positive side (tuple of integers)

Return type tuple

get_direction()
Returns interface orientation

Returns orientation (string indicating normal direction in computational space).

Returns Interface orientation in computational space (‘x’, ‘y’, or ‘z’)

Return type str

get_index()
Returns index

Returns the numerical index corresponding to the interface in question. Note that this is just for bookkeeping purposes, the interfaces may be arranged in any order as long as no index is repeated. The code will automatically handle the indices, so this is typically not modified in any way.

Returns Interface index

Return type int

get_load(*index=None*)
Returns load at position index

Returns a load from the list of load perturbations at position `index`. If no index is provided (or `None` is given), the method returns entire list. `index` must be a valid list index given the number of loads.

Parameters **index** (*int or None*) – Index within load list (optional, default is `None` to return full list)

Returns load or list

get_loadfile()
Returns loadfile for interface

Loadfile sets any surface tractions set for the interface. Note that these tractions are added to any any set by the constant initial stress tensor, initial heterogeneous stress, or interface traction perturbations

Returns Current loadfile for the interface (if the interface does not have a loadfile, returns None)

Return type loadfile or None

get_nloads()
Returns number of load perturbations on the interface

Method returns the number of load perturbations presently in the list of loads.

Returns Number of load perturbations

Return type int

get_nperts()
Returns number of friction parameter perturbations on interface

Method returns the number of parameter perturbations for the list

Returns Number of parameter perturbations

Return type int

get_paramfile()
Returns paramfile (holds arrays of heterogeneous friction parameters) for interface. Can return a subtype of paramfile corresponding to any of the specific friction law types.

Returns paramfile

get_pert(*index=None*)
Returns perturbation at position index

Method returns a perturbation from the interface. `index` is the index into the perturbation list for the particular index. If `index` is not provided or is `None`, the method returns the entire list.

Parameters **index** (*int or None*) – Index into the perturbation list for the index in question (optional, if not provided or `None`, then returns entire list)

Returns pert or list

get_type()
Returns string of interface type

Returns the type of the given interface (“locked”, “frictionless”, “slipweak”, or “stz”)

Returns Interface type

Return type str

set_index(*index*)
Sets interface index

Changes value of interface index. New index must be a nonnegative integer

Parameters **index** (*int*) – New value of index (nonnegative integer)

Returns None

set_loadfile(*newloadfile*)
Sets loadfile for interface

`newloadfile` is the new loadfile (must have type `loadfile`). If the index is bad or the loadfile type is not correct, the code will raise an error. Errors can also result if the shape of the loadfile does not match with the interface.

Parameters **newloadfile** (`loadfile`) – New loadfile to be used for the given interface

Returns None

set_paramfile(*newparamfile*)
Sets paramfile for the interface

Method sets the file holding frictional parameters for the interface.

`newparamfile` must be a parameter perturbation file of the correct type for the given interface type (i.e. if the interface is of type `slipweak`, then `newpert` must have type `swparamfile`). Errors can also result if the shape of the paramfile does not match with the interface.

Parameters **newparamfile** (*paramfile (actual type must be the appropriate subclass for the friction law of the particular interface and have the right shape)*) – New frictional parameter file (type depends on interface in question)

Returns None

write_input(*f*, *probname*, *directory*, *endian='='*)
Writes interface details to input file

This routine is called for every interface when writing problem data to file. It writes the appropriate section for the interface in the input file. It also writes any necessary binary files holding interface loads, parameters, or state variables.

Parameters

- **f** (*file*) – File handle for input file
- **probname** (*str*) – problem name (used for naming binary files)
- **directory** (*str*) – Directory for output
- **endian** (*str*) – Byte ordering for binary files (`'<'` little endian, `'>'` big endian, `'='` native, default is native)

Returns None

CHAPTER

FOUR

INCLUDED EXAMPLE PROBLEMS

2D problems:

4.1 Example Problem in 2D

To illustrate how to parameters in a text file, here is an example problem `test2d.in` (included in the `problems` directory). This example illustrates a simple 2D rupture problem based on the SCEC Rupture Code Verification Group TPV3 (this is a horizontal slice of the 3D simulation at hypocentral depth). The initial stress and friction parameters are homogeneous, with the exception of a nucleation patch at the center of the fault and strong frictional barriers at the external boundaries of the fault. The simulation saves several fields, both on-fault and off-fault.

```
[fdfault.problem]
test2d
data/
1000
0
0
0.3
50
4

[fdfault.domain]
2
2
801 802 1
1 2 1
801
401 401
1
1
slipweak
4
```

```
elastic

[fdfault.fields]
0. 0. 0. 0. 0. 0.
none
none

[fdfault.block000]
2.67 32.04 32.04
0. 0.
40. 20.
absorbing
absorbing
absorbing
none
none
none
none
none

[fdfault.block010]
2.67 32.04 32.04
0. 20.
40. 20.
absorbing
absorbing
none
absorbing
none
none
none
none

[fdfault.operator]
0.

[fdfault.interface0]
y
0 0 0
0 1 0

[fdfault.friction]
2
constant 0. 0. 0. 0. 0. -120. 70. 0.
boxcar 0. 20. 1.5 0. 0. 0. 11.6 0.
none
```

```
[fdfault.slipweak]
3
constant 0. 0. 0. 0. 0. 0.4 0.677 0.525 0. 0. 0.
boxcar 0. 2.5 2.5 0. 0. 0. 0. 0. 0. 0. 0.
boxcar 0. 37.5 2.5 0. 0. 0. 0. 0. 0. 0. 0.
none

[fdfault.outputlist]
V
V
0 1000 100
0 800 1
401 401 1
0 0 1
S
S
0 1000 100
0 800 1
401 401 1
0 0 1
U
U
0 1000 100
0 800 1
401 401 1
0 0 1


[fdfault.frontlist]
0
```

This model is fairly simple, so use of a text input file rather than a python script is a reasonable choice. The following highlights

4.2 Example 2D Problem in Python

```
import fdfault
import numpy as np

# create problem

p = fdfault.problem('testprob')

# set rk and fd order
```

```
p.set_rkorder(4)
p.set_sbporder(4)

# set time step info

p.set_nt(1601)
p.set_cfl(0.3)
p.set_ninfo(50)

# set number of blocks and coordinate information

p.set_nblocks((2,1,1))
p.set_nx_block(([601, 601], [1601], [1]))

# set block dimensions

p.set_block_lx((0,0,0),(12.,32.))
p.set_block_lx((1,0,0),(12.,32.))

# set block boundary conditions

p.set_bounds((0,0,0),['absorbing', 'none', 'absorbing', 'absorbing'])
p.set_bounds((1,0,0),['none', 'absorbing', 'absorbing', 'absorbing'])

# set block surface

y = np.linspace(0., 32., 1601)
x = 12.*np.ones(1601)+0.5*np.sin(np.pi*y/32.)

surf = fdfault.curve(1601, 'x', x, y)

p.set_block_surf((0,0,0), 1, surf)
p.set_block_surf((1,0,0), 0, surf)

# set initial fields

p.set_stress((-120., 70., 0., -100., 0., 0.))

# set interface type

p.set_iftype(0,'slipweak')

# set slip weakening parameters

p.add_pert(fdfault.swparam('constant', dc = 0.4, mus = 0.676, mud = 0.
↪525),0)
```

```
p.add_pert(fdfault.swparam('boxcar', x0 = 2., dx = 2., mus = 10000.),0)
p.add_pert(fdfault.swparam('boxcar', x0 = 30., dx = 2., mus = 10000.),
 ↪0)

# add load perturbations

p.add_load(fdfault.load('boxcar',0., 16., 1.5, 0., 0., 0., 11.6, 0.))

# add output unit

p.add_output(fdfault.output('vxbody','vx',0, 1600, 50, 0, 1200, 2, 0,␣
 ↪1600, 2, 0, 0, 1))
p.add_output(fdfault.output('vybody','vy',0, 1600, 50, 0, 1200, 2, 0,␣
 ↪1600, 2, 0, 0, 1))
p.add_output(fdfault.output('vfault','V',0, 1600, 10, 601, 601, 1, 0,␣
 ↪1600, 2, 0, 0, 1))

p.write_input()
```

3D Problems:

4.3 The Problem, Version 4

```
[fdfault.problem]
tpv4
data/
701
0.0
0.0
0.3
100
4

[fdfault.domain]
3
2
201 202 101
1 2 1
201
101 101
101
1
slipweak
4
elastic
```

```
[fdfault.fields]
0.0 0.0 0.0 0.0 0.0 0.0
none

[fdfault.block000]
2.67 32.04 32.04
0.0 0.0 0.0
40.0 20.0 20.0
absorbing
absorbing
absorbing
none
absorbing
free
none
none
none
none
none
none

[fdfault.block010]
2.67 32.04 32.04
0.0 20.0 0.0
40.0 20.0 20.0
absorbing
absorbing
none
absorbing
absorbing
free
none
none
none
none
none
none

[fdfault.interface0]
y
0 0 0
0 1 0

[fdfault.friction]
2
constant 0.0 0.0 0.0 0.0 0.0 -120.0 70.0 0.0
```

```
boxcar 0.0 20.0 1.5 12.5 1.5 0.0 11.6 0.0
none

[fdfault.slipweak]
4
constant 0.0 0.0 0.0 0.0 0.0 0.4 0.677 0.525 0.0 0.0 0.0
boxcar 0.0 20.0 20.0 2.5 2.5 0.0 10000.0 0.0 0.0 0.0 0.0
boxcar 0.0 2.5 2.5 12.5 7.5 0.0 10000.0 0.0 0.0 0.0 0.0
boxcar 0.0 37.5 2.5 12.5 7.5 0.0 10000.0 0.0 0.0 0.0 0.0
none

[fdfault.outputlist]
vf
V
0 700 10
0 200 1
101 101 1
0 100 1


[fdfault.frontlist]
1
V
0.001
```

4.4 The Problem, Version 5

```
import fdfault
import numpy as np

# create problem

p = fdfault.problem('tpv5')

# set rk and fd order

p.set_rkorder(4)
p.set_sbporder(4)

# set time step info

refine =  1
nt = 700*refine
```

```
p.set_nt(nt)
p.set_cfl(0.3)
p.set_ninfo(50*refine)

p.set_ndim(3)

# set number of blocks and coordinate information

nx = 200*refine+1
ny = 100*refine+1
nz = 100*refine+1

p.set_nblocks((1,2,1))
p.set_nx_block(([nx], [ny, ny], [nz]))

# set block dimensions

p.set_block_lx((0,0,0),(40.,20., 20.))
p.set_block_lx((0,1,0),(40.,20., 20.))

p.set_domain_xm((-20., -20., -20.))

# set block boundary conditions

p.set_bounds((0,0,0),['absorbing', 'absorbing', 'absorbing', 'none',
↪'absorbing', 'free'])
p.set_bounds((0,1,0),['absorbing', 'absorbing', 'none', 'absorbing',
↪'absorbing', 'free'])

# turn on artificial dissipation

#p.set_cdiss(0.1)

# set material

cs = 3.464
cp = 6.
rho = 2.67

p.set_material(fdfault.material('elastic', rho, rho*(cp**2-2.*cs**2),␣
↪rho*cs**2))

# set interface type

p.set_iftype(0,'slipweak')
```

```
# set slip weakening parameters

p.add_pert(fdfault.swparam('constant',0., 0., 0., 0., 0., 0.4, 0.677,␣
↪0.525))
p.add_pert(fdfault.swparam('boxcar',0., 0., 20., -17.55, 2.5, 0.,␣
↪10000., 0., 10.))
p.add_pert(fdfault.swparam('boxcar',0., -17.55, 2.5, -7.5, 7.5, 0.,␣
↪10000., 0., 10.))
p.add_pert(fdfault.swparam('boxcar',0., 17.55, 2.5, -7.5, 7.5, 0.,␣
↪10000., 0., 10.))

# add load perturbations

p.add_load(fdfault.load('constant',0., 0., 0., 0., 0., -120., 70., 0.))
p.add_load(fdfault.load('boxcar',0., 0., 1.5, -7.5, 1.5, 0., 11.6, 0.))
p.add_load(fdfault.load('boxcar',0., -7.5, 1.5, -7.5, 1.5, 0., 8., 0.))
p.add_load(fdfault.load('boxcar',0., 7.5, 1.5, -7.5, 1.5, 0., -8., 0.))

# add output unit

#p.add_output(fdfault.output('vf','V',0, nt, 5*refine, 0, nx-1, refine,
↪ ny, ny, 1, 0, nz-1, refine))

# on fault stations

onfault = [('-120', '000'), ('-075', '000'), ('-045', '000'), ('000',
↪'000'), ('045', '000'), ('075', '000'), ('120', '000'),
           ('000', '030'), ('-120', '075'), ('-075', '075'), ('-045',
↪'075'), ('000', '075'), ('045', '075'), ('075', '075'), ('120', '075
↪'),
           ('000', '120')]
fields = ['h-slip', 'h-slip-rate', 'h-shear-stress', 'v-slip', 'v-slip-
↪rate', 'v-shear-stress']
fname = ['Ux', 'Vx', 'Sx', 'Uz', 'Vz', 'Sz']
for station in onfault:
    xcoord = float(station[0])/10.
    zcoord = -float(station[1])/10.
    xpt, ypt, zpt = p.find_nearest_point((xcoord, 0., zcoord), known='y
↪', knownloc=ny)
    for fld, fn in zip(fields, fname):
        p.add_output(fdfault.output('faultst'+station[0]+'dp
↪'+station[1]+'-'+fld, fn, 0, nt, 1, xpt, xpt, 1,
                                    ypt, ypt, 1, zpt, zpt, 1))



# off fault stations
```

```
offfault = [('-120', '030', '000'), ('120', '030', '000'), ('-120',
↪'030', '075'), ('120', '030', '075')]
fields = ['h-vel', 'v-vel', 'n-vel']
fname = ['vx', 'vz', 'vy']

for station in offfault:
    xcoord = float(station[0])/10.
    ycoord = float(station[1])/10.
    zcoord = -float(station[2])/10.
    xpt, ypt, zpt = p.find_nearest_point((xcoord, ycoord, zcoord))
    for fld, fn in zip(fields, fname):
        p.add_output(fdfault.output('body'+station[1]+'st'+station[0]+
↪'dp'+station[2]+'-'+fld, fn, 0, nt, 1, xpt, xpt, 1,
                                    ypt, ypt, 1, zpt, zpt, 1))

p.set_front_output(True)

p.write_input()
```

CHAPTER

FIVE

ANALYZING SIMULATION RESULTS

5.1 Analysis With Python

`fdfault.analysis` is a python module for analyzing dynamic rupture problems for use with the C++ code.

It contains classes corresponding to the two types of outputs from the code

The module contains the following classes:

- `fdfault.analysis.output` – output unit for grid or fault data
- `fdfault.analysis.front` – rupture front output for fault surfaces

The output units can hold a variety of different fields. See the documentation for the output units for more details.

The rupture front holds rupture time values for all points on the fault. This can be used to examine rupture speeds and determine the areas of the fault that slip.

Details on the data structures in each class can be found in the documentation.

The module also contains several functions for converting binary output files to text files for the SCEC Rupture Code Verification Group. It contains functions for on- and off-fault stations, and 2D and 3D problems, as well as rupture front times. See the details of each individual function. These are not imported by default when loading the analysis submodule

class `fdfault.analysis.`**`output`**(*problem*, *name*, *datadir=None*)

Class representing an output object

Output objects contain the following attributes:

Variables

- ***`problem`*** – Name of problem
- **`name`** – Name of output unit
- **`datadir`** – Directory where simulation data is held, if `None` (default) this is the current directory

- **field** – Field that was saved to file
- **nt** – Number of time steps
- **nx** – Number of x grid points
- **ny** – Number of y grid points
- **nz** – Number of z grid points
- **endian** – Byte-ordering of simulation data (`'='` for native, `'>'` for big endian, `'<'` for little endian)
- **fielddata** – Numpy array holding simulation data (also aliased using the field name itself)

load()

Load data from data file for output item

Method loads the data from file into the `fielddata` attribute, a Numpy array, and also creates an alias with the `field` attribute itself. If you have an existing instance of an output class whose simulation data has changed, `load` can be run more than once and will refresh the contents of the simulation output.

Method takes no inputs and has no outputs. Class is modified by running this method as the simulation data will be reloaded from file if it already exists.

Returns None

class `fdfault.analysis.`**front**(*problem*, *iface*, *datadir=None*)

Class for rupture front objects

Object describing a rupture front, with the following attributes:

Variables

- ***problem*** – Name of problem
- **iface** – Interface that was output
- **datadir** – Directory where simulation data is held, if `None` (default) this is the current directory
- **nx** – Number of x grid points (or number of y grid points if the target interface has an x normal)
- **ny** – Number of y grid points (or number of z grid points if the target interface has an x or y normal)
- **endian** – Byte-ordering of simulation data (`'='` for native, `'>'` for big endian, `'<'` for little endian)
- **t** – Numpy array holding rupture time data

load()

Load data from data file for rupture front

Method loads the data from file into the `t` attribute. If you have an existing instance of an front class whose simulation data has changed, `load` can be run more than once and will refresh the contents of the simulation output.

Method takes no inputs and has no outputs. Class is modified by running this method as the simulation data will be reloaded from file if it already exists.

Returns None

5.1.1 The `write_scec` submodule

The `write_scec` module contains several functions useful for converting binary output from a simulation to the text file format used by the SCEC Rupture Code Verification group. Functions are written for on fault, off fault, and front data types, and there are versions for both 2D and 3D problems. The functions take several common optional arguments, including `depthsign` (indicates whether depth is positive or negative), `author` (the person who is submitting the results), `version` (the code version used to run the simulation), and `grid_spacing` (the resolution of the simulation, in the event you are submitting data for multiple grid spacings).

Each function writes a text file in the current directory following the file naming convention used by the server for the Code Verification group. More information on the file outputs are given in the documentation for each function.

`fdfault.analysis.write_scec.`**`write_off_fault`**(*problem*, *station*, *depthsign=1.0*, *author=''*, *version=''*, *grid_spacing=''*)

Converts code output units for off-fault station from a 3D simulation into a formatted text file for SCEC website

This function converts off fault data from binary (written by the C++ code) to ASCII text for a 3D benchmark simulation. Required inputs are the problem name (string) and station (tuple of strings in the format `(strike,across,depth)`). Optional inputs include depthsign (1. by default, changes sign on depth coordinate if -1.), and author, verision, and grid spacing strings which will be inserted into the header of the output file.

The text file is written to `{problem}_body{across}st{strike}dp{depth}.txt` in the current directory.

Parameters

- **problem** (*str*) – Problem name to write to file
- **station** (*tuple*) – Coordinates in 3D of output station (tuple of 3 strings)

- **depthsign** (*float*) – Sign of depth output, must be `1.` or `-1.` (optional, default is `1.`)
- **author** (*str*) – Person who ran the simulation (optional, default is `""`)
- **version** (*str*) – Code version used in simulation (optional, default is `""`)
- **grid_spacing** (*str*) – Grid spacing used in simulation (optional, default is `""`)

Returns None

`fdfault.analysis.write_scec.`**`write_off_fault_2d`**(*problem*, *station*, *depthsign=1.0*, *author='‘*, *version='‘*, *grid_spacing='‘*)

Converts code output units for off-fault station into a formatted text file for SCEC website

This function converts off fault data from binary (written by the C++ code) to ASCII text for a 3D benchmark simulation. Required inputs are the problem name (string) and station (tuple of strings in the format `(strike,across,depth)`). Optional inputs include depthsign (1. by default, changes sign on depth coordinate if -1.), and author, verision, and grid spacing strings which will be inserted into the header of the output file.

The text file is written to `{problem}_body{across}st{strike}dp{depth}.txt` in the current directory.

Parameters

- **problem** (*str*) – Problem name to write to file
- **station** (*tuple*) – Coordinates in 2D of output station (tuple of 3 strings, but `strike` should be `'0'`)
- **depthsign** (*float*) – Sign of depth output, must be `1.` or `-1.` (optional, default is `1.`)
- **author** (*str*) – Person who ran the simulation (optional, default is `""`)
- **version** (*str*) – Code version used in simulation (optional, default is `""`)
- **grid_spacing** (*str*) – Grid spacing used in simulation (optional, default is `""`)

Returns None

`fdfault.analysis.write_scec.`**`write_on_fault`**(*problem*, *station*, *depthsign=1.0*, *normal=True*, *author='‘*, *version='‘*, *grid_spacing='‘*)

Converts code output units for on-fault station into a formatted text file for SCEC website

This function converts on fault data from binary (written by the C++ code) to ASCII text for a 3D benchmark simulation. Required inputs are the problem name (string) and station (tuple of strings in the format `(strike,depth)`). Optional inputs include depthsign (1. by default, changes sign on depth coordinate if -1.), and author, verision, and grid spacing strings which will be inserted into the header of the output file.

The text file is written to `{problem}_faultst{strike}dp{depth}.txt` in the current directory.

Parameters

- **`problem`** (*`str`*) – Problem name to write to file
- **`station`** (*`tuple`*) – Coordinates of output station (tuple of 2 strings for strike and depth coordinates)
- **`depthsign`** (*`float`*) – Sign of depth output, must be `1.` or `-1.` (optional, default is `1.`)
- **`author`** (*`str`*) – Person who ran the simulation (optional, default is `""`)
- **`version`** (*`str`*) – Code version used in simulation (optional, default is `""`)
- **`grid_spacing`** (*`str`*) – Grid spacing used in simulation (optional, default is `""`)

Returns None

`fdfault.analysis.write_scec.`**`write_on_fault_2d`**(*problem*, *station*, *depthsign=1.0*, *normal=True*, *author=''*, *version=''*, *grid_spacing=''*)

Converts code output units for on-fault station into a formatted text file for SCEC website

This function converts on fault data from binary (written by the C++ code) to ASCII text for a 2D benchmark simulation. Required inputs are the problem name (string) and station (tuple of strings in the format `(strike,depth)`, with values chosen appropriately for a 2D simulation). Optional inputs include depthsign (1. by default, changes sign on depth coordinate if -1.), and author, verision, and grid spacing strings which will be inserted into the header of the output file.

The text file is written to `{problem}_faultst{strike}dp{depth}.txt` in the current directory.

Parameters

- **`problem`** (*`str`*) – Problem name to write to file
- **`station`** (*`tuple`*) – Coordinates of output station (tuple of 2 strings for strike and depth coordinates)

- **depthsign** (*float*) – Sign of depth output, must be `1.` or `-1.` (optional, default is `1.`)
- **author** (*str*) – Person who ran the simulation (optional, default is `""`)
- **version** (*str*) – Code version used in simulation (optional, default is `""`)
- **grid_spacing** (*str*) – Grid spacing used in simulation (optional, default is `""`)

Returns None

`fdfault.analysis.write_scec.`**`write_front`**(*problem, iface=0, depthsign=1.0, author='', version='', grid_spacing=''*)

Converts code output units for rupture front times into a formatted text file for SCEC website

This function converts rupture time data from binary (written by the C++ code) to ASCII text for a 3D benchmark simulation. Required inputs are the problem name (string). Optional inputs include the interface to write to file (default is `0`), depthsign (1. by default, changes sign on depth coordinate if -1.), and author, verision, and grid spacing strings which will be inserted into the header of the output file.

The text file is written to `{problem}_cplot.txt` in the current directory.

Parameters

- **problem** (*str*) – Problem name to write to file
- **iface** – Interface to be written to file (default is `0`)
- **depthsign** (*float*) – Sign of depth output, must be `1.` or `-1.` (optional, default is `1.`)
- **author** (*str*) – Person who ran the simulation (optional, default is `""`)
- **version** (*str*) – Code version used in simulation (optional, default is `""`)
- **grid_spacing** (*str*) – Grid spacing used in simulation (optional, default is `""`)

Returns None

5.1.2 Example

An example of how to use the included classes for analysis is included in the file `python_example.py`, included in the `python` directory.

```
# example using output class in python

# required arguments are problem name and output unit name
# data directory is optional, if no argument provided assumes it is␣
↪the current working directory

from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
from fdfault.analysis import output

vybody = output('testprob', 'vybody')

# load data structure containing information

vybody.load()

# field arrays are indexed by (t,x,y,z), with any singleton dimensions␣
↪removed
# print statement prints basic information

print(vybody)

# can also access fields directly

print(vybody.vy)

# plot velocity

plt.figure()
plt.pcolor(vybody.x, vybody.y, vybody.vy[0,:,:])
plt.colorbar()
plt.show()
```

5.2 Analysis With MATLAB

The code contains two MATLAB functions for reading in simulation data, which are roughly equivalent to the Python classes.

5.2.1 `load_output` function

```
function output = load_output(probname,name,datadir)
```

Inputs: `probname` (string), problem name `name` (string), output unit name `datadir` (string, optional) location of data directory (default is current directory)

Returns: `output`, data structure holding the following simulation data: `field` (string), type of field that is saved `endian` (string), endianness of binary data `nt` (integer), number of time steps `nx` (integer), number of x grid points `ny` (integer), number of y grid points `nz` (integer), number of z grid points `x` (float array), x grid values `y` (float array), y grid values `z` (float array), z grid values actual field data (identifier is the string contained in `field`) (float array), possible values are:

> `vx`, x-component of particle velocity `vy`, y-component of particle velocity `vz`, z-component of particle velocity `sxx`, xx component of stress tensor `sxy`, xy component of stress tensor `sxz`, xz component of stress tensor `syy`, yy component of stress tensor `syz`, yz component of stress tensor `szz`, zz component of stress tensor `lambda`, scalar plastic strain rate `gammap`, scalar plastic strain `V`, slip velocity magnitude `Vx`, x component of slip velocity `Vy`, y component of slip velocity `Vz`, z component of slip velocity `U`, slip (calculated as line integral) `Ux`, x component of slip `Uy`, y component of slip `Uz`, z component of slip `Sn`, interface normal stress `S`, interface shear traction magnitude `Sx`, x-component of interface shear traction `Sy`, y-component of interface shear traction `Sz`, z-component of interface shear traction `state`, value of state variable

Because the data is written in row major order in the C++ code, but MATLAB stores data in column major order, index order is (z, y, x, t).

5.2.2 `load_front` function

```
function front = load_front(probname,iface,datadir)
```

Inputs: `probname` (string), problem name `iface` (integer), interface number `datadir` (string, optional) location of data directory (default is current directory)

Returns: `front`, data structure holding the following simulation data: `endian` (string), endianness of binary data `nx` (integer), number of x grid points `ny` (integer), number of y grid points `x` (float array), x grid values `y` (float array), y grid values `z` (float array), z grid values `t` (float array), rupture time values

Because the data is written in row major order in the C++ code, but MATLAB stores data in column major order, index order is (y, x). Note also that because the interface is a 2D slice, `nx` and `ny` are used generically to describe the number of grid points on the interface no matter what the orientation of the interface is. Thus, if the array has an approximate normal in the x direction, `nx` is the number of grid points in the y direction and `ny` is the number of grid points in the z direction.

5.2.3 Example

An example of how to use the MATLAB functions is provided in the file `matlab_example.m`, located in the `matlab` directory. The file is reproduced here.

```
% example using load_output function in MATLAB

% required arguments are problem name and output unit name
% data directory is optional, if no argument provided assumes it is␣
↪the current working directory

vybody = load_output('hpctest','vybody');

% loads data structure containing information

vybody

% because of differences in how MATLAB and C++ order arrays, field␣
↪arrays are indexed by (z,y,x,t)
% any singleton dimensions are removed

% plot velocity

pcolor(vybody.x, vybody.y, vybody.vy(:,:,4));
shading flat;
axis image;
colorbar;
```

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX