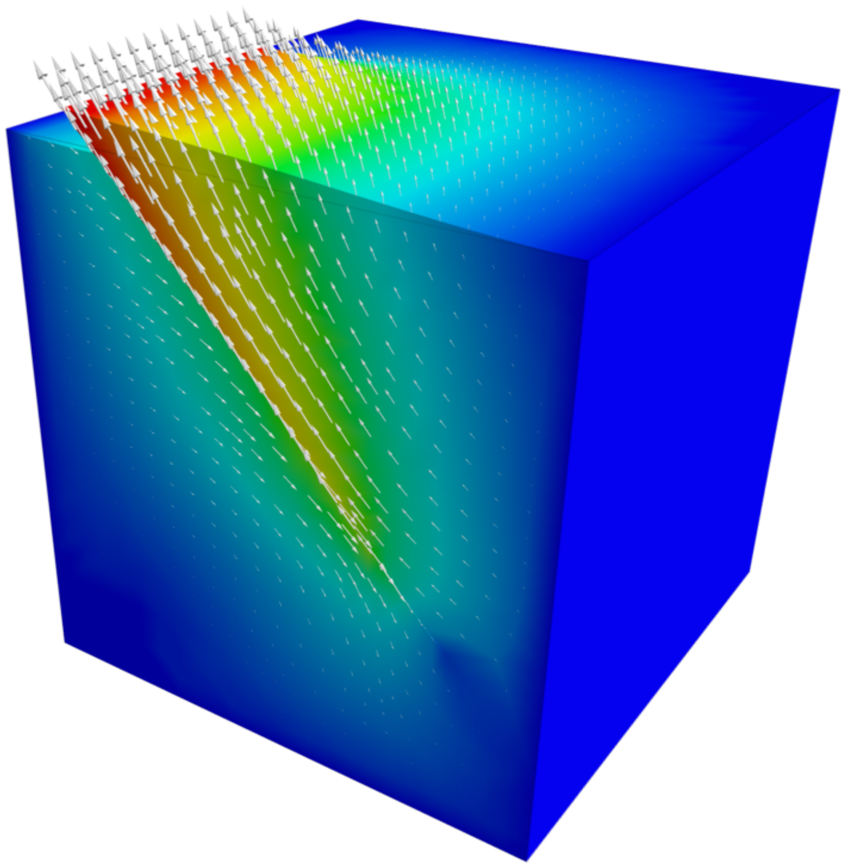
COMPUTATIONAL INFRASTRUCTURE FOR GEODYNAMICS (CIG)

# PyLith

User Manual

Version 2.2.1




Brad Aagaard

Matthew Knepley

Charles Williams


geodynamics.org

# PyLith User Manual


Version 2.2.1

July 2, 2019

# Contents

# List of Figures

# List of Tables

# Preface

## 0.1 About This Document

This document is organized into two parts. The first part begins with an introduction to PyLith and discusses the types of problems that PyLith can solve and how to run the software; the second part provides appendices and references.

## 0.2 Who Will Use This Documentation

This documentation is aimed at two categories of users: scientists who prefer to use prepackaged and specialized analysis tools, and experienced computational Earth scientists. Of the latter, there are likely to be two classes of users: those who just run models, and those who modify the source code. Users who modify the source are likely to have familiarity with scripting, software installation, and programming, but are not necessarily professional programmers.

## 0.3 Conventions

**Warning**

This is a warning.

**Important**

This is something important.

**Tip**

This is a tip, helpful hint, or suggestion.

For features recently added to PyLith, we show the version number when they were added. **New in v2.2.1**

### 0.3.1 Command Line Arguments

Exmaple of a command line argument: `--help`.

### 0.3.2 Filenames and Directories

Example of filenames and directories: `pylith`, `/usr/local`.

### 0.3.3 Unix Shell Commands

Commands entered into a Unix shell (i.e., terminal) are shown in a box. Comments are delimited by the # character. We use `$` to indicate the bash shell prompt.

```
# This is a comment.
$ ls -l
```

### 0.3.4 Excerpts of cfg Files

Example of an excerpt from a `.cfg` file:

```
# This is a comment.
[pylithapp.problem]
timestep = 2.0*s ; Time step comment.
bc = [x_pos, x_neg]
```

## 0.4 Citation

The Computational Infrastructure for Geodynamics (CIG) (`geodynamics.org`) is making this source code available to you at no cost in hopes that the software will enhance your research in geophysics. A number of individuals have contributed a significant portion of their careers toward the development of this software. It is essential that you recognize these individuals in the normal scientific practice by citing the appropriate peer-reviewed papers and making appropriate acknowledgments in talks and publications. The preferred way to generate the list of publications (in BibTEX format) to cite is to run your simulations with the `--include-citations` command line argument, or equivalently, the `--petsc.citations` command line argument. The `--help-citations` command line argument will generate the BibTEX entries for the references mentioned below.

The following peer-reviewed paper discussed the development of PyLith:

- Aagaard, B. T., M. G. Knepley, and C. A. Williams (2013). A domain decomposition approach to implementing fault slip in finite-element models of quasi-static and dynamic crustal deformation, *Journal of Geophysical Research: Solid Earth*, 118, doi: 10.1002/jgrb.50217.

To cite the software and manual, use:

- Aagaard, B., M. Knepley, C. Williams (2017), *PyLith v2.2.1.* Davis, CA: Computational Infrastructure of Geodynamics. DOI: 10.5281/zenodo.886600.
- Aagaard, B., M. Knepley, C. Williams (2017), *PyLith User Manual, Version 2.2.1.* Davis, CA: Computational Infrastructure of Geodynamics. URL: geodynamics.org/cig/software/github/pylith/v2.2.1/pylith-2.2.1_manual.pdf

## 0.5 Support

Current PyLith development is supported by the CIG, and internal GNS Science `www.gns.cri.nz` and U.S. Geological Survey `www.usgs.gov` funding. Pyre development was funded by the Department of Energy's `www.doe.gov/engine/`

content.do Advanced Simulation and Computing program and the National Science Foundation's Information Technology Research (ITR) program.

This material is based upon work supported by the National Science Foundation under Grants No. 0313238, 0745391, 1150901, and EAR-1550901. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## 0.6 Acknowledgments

Many members of the community contribute to PyLith through reporting bugs, suggesting new features and improvements, running benchmarks, and asking questions about the software. In particular, we thank Surendra Somala for contributing to the development of the fault friction implementation.

## 0.7 Request for Comments

Your suggestions and corrections can only improve this documentation. Please report any errors, inaccuracies, or typos to the CIG Short-Term Tectonics email list cig-short@geodynamics.org or create a GitHub pull request.

# Chapter 1

# Introduction

## 1.1 Overview

PyLith is is portable, scalable software for simulation of crustal deformation across spatial scales ranging from meters to hundreds of kilometers and temporal scales ranging from milliseconds to thousands of years. Its primary applications are quasi-static and dynamic modeling of earthquake faulting.

## 1.2 New in PyLith Version 2.2.1

- Added new examples
  - `examples/3d/subduction`: New suite of examples for a 3-D subduction zone. This intermediate level suite of examples illustrates a wide range of PyLith features for quasi-static simulations.
  - `examples/2d/subduction`: Added quasi-static spontaneous rupture earthquake cycle examples (Steps 5 and 6) for slip-weakening and rate- and state-friction.
  - These new examples make use of ParaView Python scripts to facilitate using ParaView with PyLith.
- Improved the PyLith manual
  - Added diagram to guide users on which installation method best meets their needs.
  - Added instructions for how to use the Windows Subsystem for Linux to install the PyLith Linux binary on systems running Windows 10.
- Fixed bug in generating Xdmf files for 2-D vector output. Converted Xdmf generator from C++ to Python for more robust generation of Xdmf files from Python scripts.
- Updated spatialdata to v1.9.10. Improved error messages when reading SimpleDB and SimpleGridDB files.
- Updated PyLith parameter viewer to v1.1.0. Application and documentation are now available online (`https://geodynamics.github.io/pylith_parameters/`). Small fix to insure hierarchy path listed matches the one for PyLith.
- Updated PETSc to v3.7.6. See the PETSc documentation for a summary of all of the changes.
- Switched to using CentOS 6.9 for Linux binary builds to insure compatibility with glibc 2.12 and later.

The `CHANGES` file in the top-level source directory contains a summary of features and bugfixes for each release.

## 1.3 History

PyLith 1.0 was the first version to allow the solution of both implicit (quasi-static) and explicit (dynamic) problems and was a complete rewrite of the original PyLith (version 0.8). PyLith 1.0 combines the functionality of EqSim [Aagaard et al., 2001a, Aagaard et al., 2001b] and PyLith 0.8. PyLith 0.8 was a direct descendant of LithoMop and was the first version that ran in parallel, as well as providing several other improvements over LithoMop. LithoMop was the product of major reengineering of Tecton, a finite-element code for simulating static and quasi-static crustal deformation. The major new features present in LithoMop included dynamic memory allocation and the use of the Pyre simulation framework and PETSc solvers. EqSim was written by Brad Aagaard to solve problems in earthquake dynamics, including rupture propagation and seismic wave propagation.

The release of PyLith 1.0 has been followed by additional releases that expand the number of features as well as improve performance. The PyLith 1.x series of releases allows the solution of both quasi-static and dynamic problems in one, two, or three dimensions. The code runs in either serial or parallel, and the design allows for relatively easy scripting using the Python programming language. Material properties and values for boundary and fault conditions are specified using spatial databases, which permit easy prescription of complex spatial variations of properties and parameters. Simulation parameters are generally specified through the use of simple ASCII files or the command line. At present, mesh information may be provided using a simple ASCII file (PyLith mesh ASCII format) or imported from CUBIT or LaGriT, two widely-used meshing packages. The elements currently available include a linear bar in 1D, linear triangles and quadrilaterals in 2D, and linear tetrahedra and hexahedra in 3D. Materials presently available include isotropic elastic, linear Maxwell viscoelastic, generalized Maxwell viscoelastic, power-law viscoelastic, and Drucker-Prager elastoplastic. Boundary conditions include Dirichlet (prescribed displacements and velocities), Neumann (traction), point forces, and absorbing boundaries. Cohesive elements are used to implement slip across interior surfaces (faults) with both kinematically-specified fault slip and slip governed by fault constitutive models. PyLith also includes an interface for computing static Green's functions for fault slip.

PyLith 2.0 replaces the finite-element data structures provided by the C++ Sieve implementation with those provided by the C DMPlex implementation. The newly developed DMPlex implementation by the PETSc developers conforms to the PETSc data manager (DM) interface, thereby providing tighter integration with other PETSc data structures, such as vectors and matrices. Other improvements include significantly reduced memory use and memory balancing.

PyLith is under active development and we expect a number of additions and improvements in the near future. Likely enhancements will include additional bulk and fault constitutive models, coupled quasi-static and dynamic simulations for earthquake cycle modeling, and coupling between elasticity, heat flow, and/or fluid flow.

## 1.4 PyLith Workflow

PyLith is one component in the process of investigating problems in tectonics (Figure 1.1 on the facing page). Given a geological problem of interest, a scientist must first provide a geometrical representation of the desired structure. Once the structure has been defined, a computational mesh must be created. PyLith presently provides three mesh importing options: CUBIT Exodus format, LaGriT GMV and Pset files, and PyLith mesh ASCII format. The modeling of the physical processes of interest is performed by a code such as PyLith. Present output consists of VTK or HDF5/Xdmf files which can be used by a number of visualization codes (e.g., ParaView, Visit, and Matlab).

## 1.5 PyLith Design

PyLith is separated into modules to encapsulate behavior and facilitate use across multiple applications. This allows expert users to replace functionality of a wide variety of components without recompiling or polluting the main code. PyLith employs external packages (see Figure 1.2 on the next page) to reduce development time and enhance computational efficiency; for example, PyLith 0.8 ran two times faster when the PETSc linear solver was used.

PyLith is written in two programming languages. High-level code is written in Python; this rich, expressive interpreted language with dynamic typing reduces development time and permits flexible addition of user-contributed modules. This high-level code makes use of Pyre, a science-neutral simulation framework developed at Caltech, to link the modules together at runtime and gather user-input. Low-level code is written in C++, providing fast execution while still allowing an object-oriented implementation. This low-level code relies on PETSc to perform operations on matrices and vectors in parallel. We also make extensive use of two Python packages. SWIG is a package that simplifies the task of adding C++ extensions to Python code, and FIAT provides tabulated basis functions and numerical quadrature points.

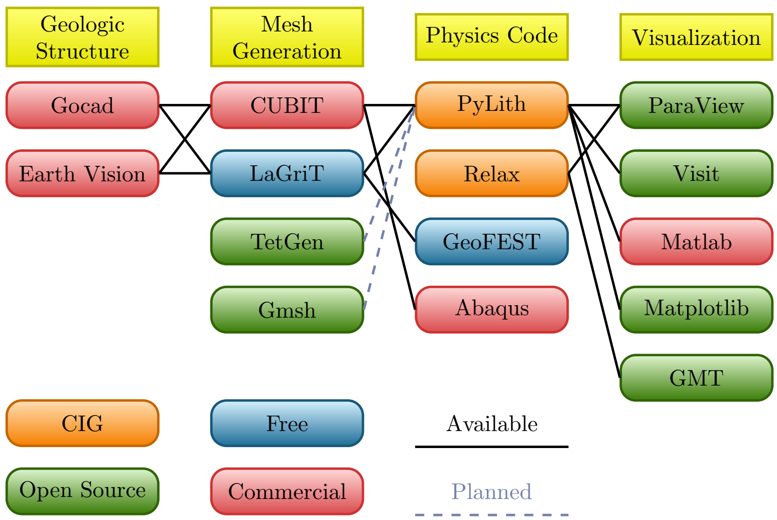



Figure 1.1: Workflow involved in going from geologic structure to problem analysis.

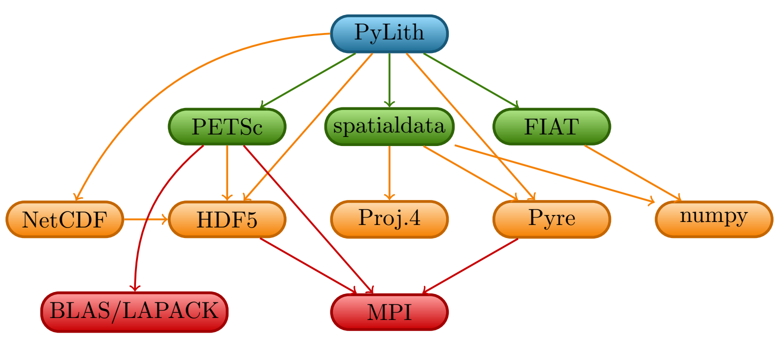



Figure 1.2: PyLith dependencies. PyLith makes direct use of several other packages, some of which have their own dependencies.

In writing PyLith 1.0, the code was designed to be object-oriented and modular. Each type of module is accessed through a specified interface (set of functions). This permits adding, replacing, and rewriting modules without affecting other parts of the code. This code structure simplifies code maintenance and development. Extending the set of code features is also easier, since developers can create new modules derived from the existing ones.

The current code design leverages Pyre and PETSc extensively. Pyre glues together the various modules used to construct a simulation and specify the parameters. PETSc provides the finite-element data structures and handles the creation and manipulation of matrices and vectors. As a result, most of the PyLith source code pertains to implementing the geodynamics, such as bulk rheology, boundary conditions, and slip on faults.

PyLith also uses FIAT to tabulate the finite-element basis functions at the numerical integration (quadrature) points. Nemesis allows PyLith to run Python using the Message Passing Interface (MPI) for parallel processing. Additional, indirect dependencies (see Figure 1.2 on the preceding page) include numpy (efficient operations on numerical arrays in Python), Proj.4 (geographic projections), and SWIG (calling C++ functions from Python).

During development, tests were constructed for nearly every module function. These unit tests are distributed with the source code. These tests are run throughout the development cycle to expose bugs and isolate their origin. As additional changes are made to the code, the tests are rerun to help prevent introduction of new bugs. A number of simple, full-scale tests, such as axial compression and extension, simple shear, and slip on through-going faults, have been used to test the code. Additionally, we have run the Southern California Earthquake Center crustal deformation and several of the spontaneous rupture benchmarks for strike-slip and reverse-slip to determine the relative local and global error (see Chapter 8 on page 241).

### 1.5.1 Pyre

Pyre is an object-oriented environment capable of specifying and launching numerical simulations on multiple platforms, including Beowulf-class parallel computers and grid computing systems. Pyre allows the binding of multiple components such as solid and fluid models used in Earth science simulations, and different meshers. The Pyre framework enables the elegant setup, modification and launching of massively parallel solver applications.

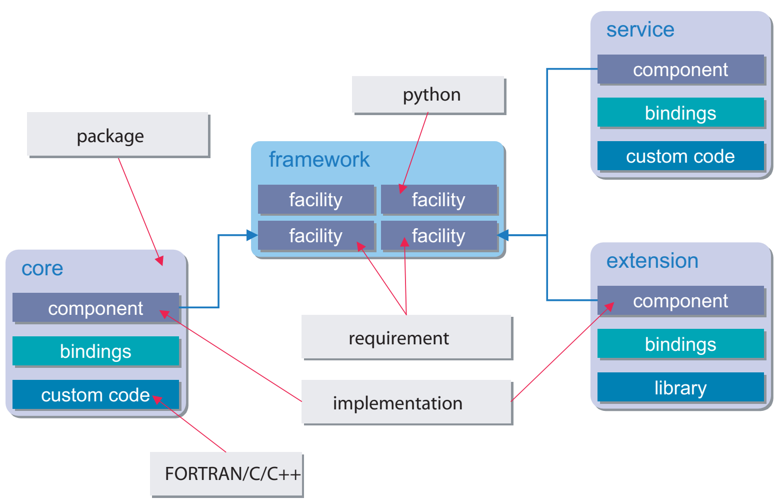



Figure 1.3: Pyre Architecture. The integration framework is a set of cooperating abstract services.

Pyre is a framework, a combination of software and design philosophy that promotes the reuse of code. In their canonical software design book, *Design Patterns*, Erich Gamma *et al.* condense the concept of a framework concept down to, "When you use a framework, you reuse the main body and write the code it calls." In the context of frameworks and object-oriented programming, Pyre can be thought of as a collection of classes and the way their instances interact. Programming applications based on Pyre will look similar to those written in any other object-oriented language. The Pyre framework contains a subset of parts that make up the overall framework. Each of those parts is designed to solve a specific problem.

The framework approach to computation offers many advantages. It permits the exchange of codes and promotes the reuse of standardized software while preserving efficiency. Frameworks are also an efficient way to handle changes in computer architecture. They present programmers and scientists with a unified and well-defined task and allow for shared costs of the housekeeping aspects of software development. They provide greater institutional continuity to model development than piecemeal approaches.

The Pyre framework incorporates features aimed at enabling the scientific non-expert to perform tasks easily without hindering the expert. Target features for end users allow complete and intuitive simulation specification, reasonable defaults, consistency checks of input, good diagnostics, easy access to remote facilities, and status monitoring. Target features for developers include easy access to user input, a shorter development cycle, and good debugging support.

### 1.5.2 PETSc

PyLith 2.x makes use of a set of data structures and routines in PETSc called DMPlex, which is still under active development. DMPlex provides data structures and routines for for representing and manipulating computational meshes, and it greatly simplifies finite-element computations.DMPlex represents the topology of the domain. Zero volume elements are inserted along all fault surfaces to implement kinematic (prescribed) or dynamic (constitutive model) implementations of fault slip. Material properties and other parameters are represented as scalar and vector fields over the mesh using vectors to store the values and sections to map vertices, edges, faces, and cells to indices in the vector. For each problem, functions are provided to calculate the residual and its Jacobian. All numerical integration is done in these functions, and parallel assembly is accomplished using the get/set closure paradigm of the DMPlex framework. We assemble into PETSc linear algebra objects and then call PETSc solvers.

PETSc `www-unix.mcs.anl.gov/petsc/petsc-as`, the Portable, Extensible Toolkit for Scientific computation, provides a suite of routines for parallel, numerical solution of partial differential equations for linear and nonlinear systems with large, sparse systems of equations. PETSc includes solvers that implement a variety of Newton and Krylov subspace methods. It can also interface with many external packages, including ESSL, MUMPS, Matlab, ParMETIS, PVODE, and Hypre, thereby providing additional solvers and interaction with other software packages.

PETSc includes interfaces for FORTRAN 77/90, C, C++, and Python for nearly all of the routines, and PETSc can be installed on most Unix systems. PETSc can be built with user-supplied, highly optimized linear algebra routines (e.g., ATLAS and commercial versions of BLAS/LAPACK), thereby improving application performance. Users can use PETSc parallel matrices, vectors, and other data structures for most parallel operations, eliminating the need for explicit calls to Message Passing Interface (MPI) routines. Many settings and options can be controlled with PETSc-specific command-line arguments, including selection of preconditions, solvers, and generation of performance logs.

# Chapter 2

# Governing Equations

We present here a brief derivation of the equations for both quasi-static and dynamic computations. Since the general equations are the same (except for the absence of inertial terms in the quasi-static case), we first derive these equations. We then present solution methods for each specific case. In all of our derivations, we use the notation described in Table 2.1 for both index and vector notation. When using index notation, we use the common convention that repeated indices indicate summation over the range of the index.

Table 2.1: Mathematical notation

| **Symbol** | | **Description** |
|---|---|---|
| **Index notation** | **Vector Notation** | |
| $a_i$ | $\vec{a}$ | Vector field a |
| $a_{ij}$ | $\underline{a}$ | Second order tensor field a |
| $u_i$ | $\vec{u}$ | Displacement vector field |
| $d_i$ | $\vec{d}$ | Fault slip vector field |
| $f_i$ | $\vec{f}$ | Body force vector field |
| $T_i$ | $\vec{T}$ | Traction vector field |
| $\sigma_{ij}$ | $\underline{\sigma}$ | Stress tensor field |
| $n_i$ | $\vec{n}$ | Normal vector field |
| $\rho$ | $\rho$ | Mass density scalar field |

## 2.1 Derivation of Elasticity Equation

### 2.1.1 Index Notation

Consider volume $V$ bounded by surface $S$. Applying a Lagrangian description of the conservation of momentum gives

$$\frac{\partial}{\partial t}\int_V \rho \frac{\partial u_i}{\partial t}\, dV = \int_V f_i\, dV + \int_S T_i\, dS. \tag{2.1}$$

The traction vector field is related to the stress tensor through

$$T_i = \sigma_{ij} n_j, \tag{2.2}$$

where $n_j$ is the vector normal to $S$. Substituting into equation 2.1 yields

$$\frac{\partial}{\partial t}\int_V \rho \frac{\partial u_i}{\partial t}\, dV = \int_V f_i\, dV + \int_S \sigma_{ij} n_j\, dS. \tag{2.3}$$

Applying the divergence theorem,

$$\int_V a_{i,j}\,dV = \int_S a_j n_j\,dS, \tag{2.4}$$

to the surface integral results in

$$\frac{\partial}{\partial t}\int_V \rho \frac{\partial u_i}{\partial t}\,dV = \int_V f_i\,dV + \int_V \sigma_{ij,j}\,dV, \tag{2.5}$$

which we can rewrite as

$$\int_V \left(\rho \frac{\partial^2 u_i}{\partial t^2} - f_i - \sigma_{ij,j}\right) dV = 0. \tag{2.6}$$

Because the volume $V$ is arbitrary, the integrand must be zero at every location in the volume, so that we end up with

$$\rho \frac{\partial^2 u_i}{\partial t^2} - f_i - \sigma_{ij,j} = 0 \text{ in } V, \tag{2.7}$$

$$\sigma_{ij} n_j = T_i \text{ on } S_T, \tag{2.8}$$

$$u_i = u_i^o \text{ on } S_u, \text{ and} \tag{2.9}$$

$$R_{ki}(u_i^+ - u_i^-) = d_k \text{ on } S_f. \tag{2.10}$$

We specify tractions, $T_i$, on surface $S_f$, displacements, $u_i^o$, on surface $S_u$, and slip, $d_k$, on fault surface $S_f$ (we will consider the case of fault constitutive models in Section 6.4 on page 92). The rotation matrix $R_{ki}$ transforms vectors from the global coordinate system to the fault coordinate system. Note that since both $T_i$ and $u_i$ are vector quantities, there can be some spatial overlap of the surfaces $S_T$ and $S_u$; however, the same degree of freedom cannot simultaneously have both types of boundary conditions.

### 2.1.2 Vector Notation

Consider volume $V$ bounded by surface $S$. Applying a Lagrangian description of the conservation of momentum gives

$$\frac{\partial}{\partial t}\int_V \rho \frac{\partial \vec{u}}{\partial t}\,dV = \int_V \vec{f}\,dV + \int_S \vec{T}\,dS. \tag{2.11}$$

The traction vector field is related to the stress tensor through

$$\vec{T} = \underline{\sigma} \cdot \vec{n}, \tag{2.12}$$

where $\vec{n}$ is the vector normal to $S$. Substituting into equation 2.11 yields

$$\frac{\partial}{\partial t}\int_V \rho \frac{\partial \vec{u}}{\partial t}\,dV = \int_V \vec{f}\,dV + \int_S \underline{\sigma} \cdot \vec{n}\,dS. \tag{2.13}$$

Applying the divergence theorem,

$$\int_V \nabla \cdot \vec{a}\,dV = \int_S \vec{a} \cdot \vec{n}\,dS, \tag{2.14}$$

to the surface integral results in

$$\frac{\partial}{\partial t}\int_V \rho \frac{\partial \vec{u}}{\partial t}\,dV = \int_V \vec{f}\,dV + \int_V \nabla \cdot \underline{\sigma}\,dV, \tag{2.15}$$

which we can rewrite as

$$\int_V \left(\rho \frac{\partial^2 \vec{u}}{\partial t^2} - \vec{f} - \nabla \cdot \vec{\sigma}\right) dV = \vec{0}. \tag{2.16}$$

Because the volume $V$ is arbitrary, the integrand must be the zero vector at every location in the volume, so that we end up with

$$\rho \frac{\partial^2 \vec{u}}{\partial t^2} - \vec{f} - \nabla \cdot \vec{\sigma} = \vec{0} \text{ in } V, \tag{2.17}$$

$$\underline{\sigma} \cdot \vec{n} = \vec{T} \text{ on } S_T, \tag{2.18}$$

$$\vec{u} = \vec{u^o} \text{ on } S_u, \text{ and} \tag{2.19}$$

$$\underline{R} \cdot (\vec{u^+} - \vec{u^-}) = \vec{d} \text{ on } S_f. \tag{2.20}$$

We specify tractions, $\vec{T}$, on surface $S_f$, displacements, $\overrightarrow{u^o}$, on surface $S_u$, and slip, $\vec{d}$, on fault surface $S_f$ (we will consider the case of fault constitutive models in Section 6.4 on page 92). The rotation matrix $\underline{R}$ transforms vectors from the global coordinate system to the fault coordinate system. Note that since both $\overrightarrow{T}$ and $\overrightarrow{u}$ are vector quantities, there can be some spatial overlap of the surfaces $S_T$ and $S_u$; however, the same degree of freedom cannot simultaneously have both types of boundary conditions.

## 2.2 Finite-Element Formulation of Elasticity Equation

We formulate a set of algebraic equations using Galerkin's method. We consider (1) a trial solution, $\vec{u}$, that is a piecewise differentiable vector field and satisfies the Dirichlet boundary conditions on $S_u$, and (2) a weighting function, $\vec{\phi}$, that is a piecewise differentiable vector field and is zero on $S_u$.

### 2.2.1 Index Notation

We start with the wave equation (strong form),

$$\sigma_{ij,j} + f_i = \rho \ddot{u}_i \text{ in } V, \tag{2.21}$$

$$\sigma_{ij} n_j = T_i \text{ on } S_T, \tag{2.22}$$

$$u_i = u_i^o \text{ on } S_u, \tag{2.23}$$

$$R_{ki}(u_i^+ - u_i^-) = d_k \text{ on } S_f, \text{ and} \tag{2.24}$$

$$\sigma_{ij} = \sigma_{ji} \text{ (symmetric).} \tag{2.25}$$

We construct the weak form by computing the dot product of the wave equation and weighting function and setting the integral over the domain to zero:

$$\int_V \left(\sigma_{ij,j} + f_i - \rho \ddot{u}_i\right) \phi_i \, dV = 0, \text{ or} \tag{2.26}$$

$$\int_V \sigma_{ij,j} \phi_i \, dV + \int_V f_i \phi_i \, dV - \int_V \rho \ddot{u}_i \phi_i \, dV = 0. \tag{2.27}$$

Consider the divergence theorem applied to the dot product of the stress tensor and the weighting function, $\sigma_{ij}\phi_i$,

$$\int_V (\sigma_{ij}\phi_i)_{,j} \, dV = \int_S (\sigma_{ij}\phi_i) n_i \, dS. \tag{2.28}$$

Expanding the left-hand side yields

$$\int_V \sigma_{ij,j} \phi_i \, dV + \int_V \sigma_{ij} \phi_{i,j} \, dV = \int_S \sigma_{ij} \phi_i n_i \, dS, \text{ or} \tag{2.29}$$

$$\int_V \sigma_{ij,j} \phi_i \, dV = -\int_V \sigma_{ij} \phi_{i,j} \, dV + \int_S \sigma_{ij} \phi_i n_i \, dS. \tag{2.30}$$

Substituting into the weak form gives

$$-\int_V \sigma_{ij} \phi_{i,j} \, dV + \int_S \sigma_{ij} \phi_i n_i \, dS + \int_V f_i \phi_i \, dV - \int_V \rho \ddot{u}_i \phi_i \, dV = 0. \tag{2.31}$$

Turning our attention to the second term, we separate the integration over $S$ into integration over $S_T$ and $S_u$ (we will consider tractions over the fault surface, $S_f$, associated with the fault constitutive model in Section 6.4 on page 92),

$$-\int_V \sigma_{ij} \phi_{i,j} \, dV + \int_{S_T} \sigma_{ij} \phi_i n_i \, dS + \int_{S_u} \sigma_{ij} \phi_i n_i \, dS + \int_V f_i \phi_i \, dV - \int_V \rho \ddot{u}_i \phi_i \, dV = 0, \tag{2.32}$$

and recognize that

$$\sigma_{ij} n_i = T_i \text{ on } S_T \text{ and} \tag{2.33}$$

$$\phi_i = 0 \text{ on } S_u, \tag{2.34}$$

so that the equation reduces to

$$-\int_V \sigma_{ij}\phi_{i,j}\,dV + \int_{S_T} T_i\phi_i\,dS + \int_V f_i\phi_i\,dV - \int_V \rho\ddot{u}_i\phi_i\,dV = 0. \tag{2.35}$$

We express the trial solution and weighting function as linear combinations of basis functions,

$$u_i = \sum_m a_i^m N^m, \tag{2.36}$$

$$\phi_i = \sum_n c_i^n N^n. \tag{2.37}$$

Note that because the trial solution satisfies the Dirichlet boundary condition, the number of basis functions for $u$ is generally greater than the number of basis functions for $\phi$, i.e., $m > n$. Substituting in the expressions for the trial solution and weighting function yields

$$-\int_V \sigma_{ij}\sum_n c_i^n N_{,j}^n\,dV + \int_{S_T} T_i\sum_n c_i^n N^n\,dS + \int_V f_i\sum_n c_i^n N^n\,dV - \int_V \rho\sum_m \ddot{a}_i^m N^m \sum_n c_i^n N^n\ dV = 0, \text{ or} \tag{2.38}$$

$$\sum_n c_i^n\left(-\int_V \sigma_{ij}N_{,j}^n\,dV + \int_{S_T} T_i N^n\,dS + \int_V f_i N^n\,dV - \int_V \rho\sum_m \ddot{a}_i^m N^m N^n\ dV\right) = 0. \tag{2.39}$$

Because the weighting function is arbitrary, this equation must hold for all $c_i^n$, so that the quantity in parenthesis is zero for each $c_i^n$

$$-\int_V \sigma_{ij}N_{,j}^n\,dV + \int_{S_T} T_i N^n\,dS + \int_V f_i N^n\,dV - \int_V \rho\sum_m \ddot{a}_i^m N^m N^n\ dV = \vec{0}. \tag{2.40}$$

We want to solve this equation for the unknown coefficients $a_i^m$ subject to

$$u_i = u_i^o \text{ on } S_u, \text{ and} \tag{2.41}$$

$$R_{ki}(u_i^+ - u_i^-) = d_k \text{ on } S_f, \tag{2.42}$$

### 2.2.2 Vector Notation

We start with the wave equation (strong form),

$$\nabla\cdot\underline{\sigma} + \overrightarrow{f} = \rho\frac{\partial^2\overrightarrow{u}}{\partial t^2} \text{ in } V, \tag{2.43}$$

$$\underline{\sigma}\cdot\overrightarrow{n} = \overrightarrow{T} \text{ on } S_T, \tag{2.44}$$

$$\overrightarrow{u} = \overrightarrow{u^o} \text{ on } S_u, \tag{2.45}$$

$$\underline{\mathrm{R}}\cdot(\overrightarrow{u^+} - \overrightarrow{u^-}) = \vec{d} \text{ on } S_f \tag{2.46}$$

$$\underline{\sigma} = \underline{\sigma}^T \text{ (symmetric)}. \tag{2.47}$$

We construct the weak form by multiplying the wave equation by a weighting function and setting the integral over the domain to zero. The weighting function is a piecewise differential vector field, $\overrightarrow{\phi}$, where $\overrightarrow{\phi} = 0$ on $S_u$. Hence our weak form is

$$\int_V \left(\nabla\cdot\underline{\sigma} + \overrightarrow{f} - \rho\frac{\partial^2\overrightarrow{u}}{\partial t^2}\right)\cdot\overrightarrow{\phi}\,dV = 0, \text{ or} \tag{2.48}$$

$$\int_V (\nabla\cdot\underline{\sigma})\cdot\overrightarrow{\phi}\,dV + \int_V \overrightarrow{f}\cdot\overrightarrow{\phi}\,dV - \int_V \rho\frac{\partial^2\overrightarrow{u}}{\partial t^2}\cdot\overrightarrow{\phi}\,dV = 0. \tag{2.49}$$

Consider the divergence theorem applied to the dot product of the stress tensor and the trial function, $\underline{\sigma}\cdot\overrightarrow{\phi}$,

$$\int_V \nabla\cdot(\underline{\sigma}\cdot\overrightarrow{\phi})\,dV = \int_S (\underline{\sigma}\cdot\overrightarrow{\phi})\cdot\overrightarrow{n}\,dS. \tag{2.50}$$

Expanding the left-hand side yields

$$\int_V (\nabla \cdot \underline{\sigma}) \cdot \overrightarrow{\phi}\, dV + \int_V \underline{\sigma} : \nabla \overrightarrow{\phi}\, dV = \int_S (\underline{\sigma} \cdot \overrightarrow{\phi}) \cdot \overrightarrow{n}\, dS, \text{ or} \tag{2.51}$$

$$\int_V (\nabla \cdot \underline{\sigma}) \cdot \overrightarrow{\phi}\, dV = -\int_V \underline{\sigma} : \nabla \overrightarrow{\phi}\, dV + \int_S \underline{\sigma} \cdot \overrightarrow{n} \cdot \overrightarrow{\phi}\, dS. \tag{2.52}$$

Substituting into the weak form gives

$$-\int_V \underline{\sigma} : \nabla \overrightarrow{\phi}\, dV + \int_S \underline{\sigma} \cdot \overrightarrow{n} \cdot \overrightarrow{\phi}\, dS + \int_V \overrightarrow{f} \cdot \overrightarrow{\phi}\, dV - \int_V \rho \frac{\partial^2 \overrightarrow{u}}{\partial t^2} \cdot \overrightarrow{\phi}\, dV = 0. \tag{2.53}$$

We separate the integration over $S$ into integration over $S_T$ and $S_u$,

$$-\int_V \underline{\sigma} : \nabla \overrightarrow{\phi}\, dV + \int_{S_T} \underline{\sigma} \cdot \overrightarrow{n} \cdot \overrightarrow{\phi}\, dS + \int_{S_u} \underline{\sigma} \cdot \overrightarrow{n} \cdot \overrightarrow{\phi}\, dS + \int_V \overrightarrow{f} \cdot \overrightarrow{\phi}\, dV - \int_V \rho \frac{\partial^2 \overrightarrow{u}}{\partial t^2} \cdot \overrightarrow{\phi}\, dV = 0, \tag{2.54}$$

and recognize that

$$\underline{\sigma} \cdot \overrightarrow{n} = \overrightarrow{T} \text{ on } S_T \text{ and} \tag{2.55}$$

$$\overrightarrow{\phi} = 0 \text{ on } S_u, \tag{2.56}$$

so that the equation reduces to

$$-\int_V \underline{\sigma} : \nabla \overrightarrow{\phi}\, dV + \int_{S_T} \overrightarrow{T} \cdot \overrightarrow{\phi}\, dS + \int_V \overrightarrow{f} \cdot \overrightarrow{\phi}\, dV - \int_V \rho \frac{\partial^2 \overrightarrow{u}}{\partial t^2} \cdot \overrightarrow{\phi}\, dV = 0. \tag{2.57}$$

We express the trial solution and weighting function as linear combinations of basis functions,

$$\vec{u} = \sum_m \overrightarrow{a^m} N^m, \tag{2.58}$$

$$\vec{\phi} = \sum_n \overrightarrow{c^n} N^n. \tag{2.59}$$

Note that because the weighting function is zero on $S_u$, the number of basis functions for $\vec{u}$ is generally greater than the number of basis functions for $\vec{\phi}$, i.e., $m > n$. Substituting in the expressions for the trial solution and weighting function yields

$$-\int_V \underline{\sigma} : \sum_n \overrightarrow{c^n} \nabla N^n_{,}\, dV + \int_{S_T} \vec{T} \cdot \sum_n \overrightarrow{c^n} N^n\, dS + \int_V \vec{f} \cdot \sum_n \overrightarrow{c^n} N^n\, dV$$

$$-\int_V \rho \sum_m \frac{\partial^2 \overrightarrow{a^m}}{\partial t^2} N^m \cdot \sum_n \overrightarrow{c^n} N^n\; dV = 0. \tag{2.60}$$

Because the weighting function is arbitrary, this equation must hold for all $\overrightarrow{c^n}$, so that

$$-\int_V \underline{\sigma} : \nabla N^n\, dV + \int_{S_T} \vec{T} N^n\, dS + \int_V \vec{f} N^n\, dV - \int_V \rho \sum_m \frac{\partial^2 \overrightarrow{a^m}}{\partial t^2} N^m N^n\, dV = \vec{0}. \tag{2.61}$$

We want to solve this equation for the unknown coefficients $\overrightarrow{a^m}$ subject to

$$\vec{u} = u^o \rightarrow \text{ on } S_u, \text{ and} \tag{2.62}$$

$$\underline{R}(\overrightarrow{u^+} - \overrightarrow{u^-}) = \vec{d} \text{ on } S_f, \tag{2.63}$$

## 2.3 Solution Method for Quasi-Static Problems

For brevity we outline the solution method for quasi-static problems using only index notation. In quasi-static problems we neglect the inertial terms, so equation (2.40) reduces to

$$-\int_V \sigma_{ij} N^n_{,j}\, dV + \int_{S_T} T_i N^n\, dS + \int_V f_i N^n\, dV = \vec{0}. \tag{2.64}$$

As a result, time-dependence only enters through the constitutive relationships and the loading conditions. We consider the deformation at time $t+\Delta t$,

$$-\int_V \sigma_{ij}(t+\Delta t) N^n_{,j}\, dV + \int_{S_T} T_i(t+\Delta t) N^n\, dS + \int_V f_i(t+\Delta t) N^n\, dV = \vec{0}. \tag{2.65}$$

We solve this equation through formulation of a linear algebraic system of equations ($Au = b$), involving the residual ($r = b - Au$) and Jacobian ($A$). The residual is simply

$$r^n_i = -\int_V \sigma_{ij}(t+\Delta t) N^n_{,j}\, dV + \int_{S_T} T_i(t+\Delta t) N^n\, dS + \int_V f_i(t+\Delta t) N^n\, dV. \tag{2.66}$$

We employ numerical quadrature in the finite-element discretization and replace the integrals with sums over the cells and quadrature points,

$$\begin{aligned} r^n_i = -\sum_{\text{vol cells}} \sum_{\text{quad pts}} \sigma_{ij}(x_q, t+\Delta t) N^n_{,j}(x_q)\, w_q |J_{cell}(x_q)| + \sum_{\text{vol cells}} \sum_{\text{quad pts}} f_i(x_q, t+\Delta t) N^n(x_q)\, w_q |J_{cell}(x_q)| \\ + \sum_{\text{tract cells}} \sum_{\text{quad pts}} T_i(x_q, t+\Delta t) N^n(x_q)\, w_q |J_{cell}(x_q)|, \end{aligned} \tag{2.67}$$

where $r^n_i$ is an $nd$ vector ($d$ is the dimension of the vector space) and $i$ is a vector space component, $x_q$ are the coordinates of the quadrature points, $w_q$ are the weights of the quadrature points, and $|J_{cell}(x_q)|$ is the determinant of the Jacobian matrix evaluated at the quadrature points associated with mapping the reference cell to the actual cell. The quadrature scheme for the integral over the tractions is one dimension lower than the one used in integrating the terms for the volume cells.

In order to find the Jacobian of the system, we let

$$\sigma_{ij}(t+\Delta t) = \sigma_{ij}(t) + d\sigma_{ij}(t). \tag{2.68}$$

Isolating the term associated with the increment in stresses yields

$$\int_V d\sigma_{ij}(t) N^n_j\, dV = -\int_V \sigma_{ij}(t) N^n_{,j}\, dV + \int_{S_T} T_i(t+\Delta t) N^n\, dS + \int_V f_i(t+\Delta t) N^n\, dV \tag{2.69}$$

We associate the term on the left-hand-side with the action of the system Jacobian on the increment of the displacement field. We approximate the increment in stresses using linear elasticity and infinitesimal strains,

$$d\sigma_{ij}(t) = C_{ijkl}(t) d\varepsilon_{kl}(t) \tag{2.70}$$

$$d\sigma_{ij}(t) = \frac{1}{2} C_{ijkl}(t)(du_{k.l}(t) + du_{l,k}(t)) \tag{2.71}$$

$$d\sigma_{ij}(t) = \frac{1}{2} C_{ijkl}(t)(\sum_m da^m_{k,l}(t) N^m + \sum_m da^m_{l,k}(t) N^m) \tag{2.72}$$

Now, $d\sigma_{ij}\phi_{i,j}$ is a scalar, so it is symmetric,

$$d\sigma_{ij}\phi_{i,j} = d\sigma_{ji}\phi_{j,i}, \tag{2.73}$$

and we know that $d\sigma_{ij}$ is symmetric, so

$$d\sigma_{ij}\phi_{i,j} = d\sigma_{ij}\phi_{j,i}, \tag{2.74}$$

which means

$$\phi_{i,j} = \phi_{j,i}, \tag{2.75}$$

which we can write as

$$\phi_{i,j} = \frac{1}{2}(\phi_{i,j} + \phi_{j,i}). \tag{2.76}$$

In terms of the basis functions, we have

$$\sum_n c_i^n N_{,j}^n = \frac{1}{2}(\sum_n c_i^n N_{,j}^n + \sum_n c_j^n N_{,i}^n). \tag{2.77}$$

Combining these expressions for the increment in stresses and making use of the symmetry of the weighting functions, we find the system Jacobian is

$$A_{ij}^{nm} = \int_V \frac{1}{4} C_{ijkl}(N_{,l}^m + N_{,k}^m)(N_{,j}^n + N_{,i}^n)\ dV. \tag{2.78}$$

We employ numerical quadrature in the finite-element discretization and replace the integral with a sum over the cells and quadrature points,

$$A_{ij}^{nm} = \sum_{\text{vol cells}} \sum_{\text{quad pts}} \frac{1}{4} C_{ijkl}(N_{,l}^m(x_q) + N_{,k}^m(x_q))(N_{,j}^n(x_q) + N_{,i}^n(x_q))\, w_q |J_{cell}(x_q). \tag{2.79}$$

## 2.4 Solution Method for Dynamic Problems

For brevity we outline the solution method for dynamic problems using only index notation. Time-dependence enters through the constitutive relationships, loading conditions, and the inertial terms. We consider the deformation at time $t$,

$$-\int_V \sigma_{ij}(t) N_{,j}^n\, dV + \int_{S_T} T_i(t) N^n\, dS + \int_V f_i(t) N^n\, dV - \int_V \rho \sum_m \ddot{a}_i^m(t) N^m N^n\ dV = \vec{0}. \tag{2.80}$$

We solve this equation through formulation of a linear algebraic system of equations ($Au = b$), involving the residual ($r = b - Au$) and Jacobian ($A$). The residual is simply

$$r_i^n = -\int_V \sigma_{ij}(t) N_{,j}^n\, dV + \int_{S_T} T_i(t) N^n\, dS + \int_V f_i(t) N^n\, dV - \int_V \rho \sum_m \ddot{a}_i^m(t) N^m N^n\ dV. \tag{2.81}$$

We employ numerical quadrature in the finite-element discretization and replace the integrals with sums over the cells and quadrature points,

$$\begin{aligned} r_i^n = -\sum_{\text{vol cells}} \sum_{\text{quad pts}} \sigma_{ij}(x_q,t) N^n(x_q)\, w_q |J_{cell}(x_q)| + \sum_{\text{vol cells}} \sum_{\text{quad pts}} f_i(x_q,t) N^n(x_q)\, w_q |J_{cell}(x_q)| \\ + \sum_{\text{tract cells}} \sum_{\text{quad pts}} T_i(x_q,t) N^n(x_q)\, w_q |J_{cell}(x_q)| - \sum_{\text{vol cells}} \sum_{\text{quad pts}} \rho \sum_m \ddot{a}_i^m(t) N^m N^n\ w_{q|J_{cell}(x_q)}, \end{aligned} \tag{2.82}$$

where $x_q$ are the coordinates of the quadrature points, $w_q$ are the weights of the quadrature points, and $|J_{cell}(x_q)|$ is the determinant of the Jacobian matrix evaluated at the quadrature points associated with mapping the reference cell to the actual cell. The quadrature scheme for the integral over the tractions is one dimension lower than the one used in integrating the terms for the volume cells.

We find the system Jacobian matrix by making use of the temporal discretization and isolating the term for the increment in the displacement field at time $t$. Using the central difference method to approximate the acceleration (and velocity),

$$\ddot{u}_i(t) = \frac{1}{\Delta t^2}\left(u_i(t+\Delta t) - 2u_i(t) + u_i(t-\Delta t)\right) \tag{2.83}$$

$$\dot{u}_i(t) = \frac{1}{2\Delta t}\left(u_i(t+\Delta t) - u_i(t-\Delta t)\right) \tag{2.84}$$

and writing the displacement at time $t+\Delta t$ in terms of the displacement at $t$ (for consistency with the displacement increment quasi-static formulation),

$$u_i(t+\Delta t) = u_i(t) + du_i(t), \tag{2.85}$$

$$\ddot{u}_i(t) = \frac{1}{\Delta t^2}\left(du_i(t) - u_i(t) + u_i(t-\Delta t)\right), \tag{2.86}$$

$$\dot{u}_i(t) = \frac{1}{2\Delta t}\left(du_i(t) + u_i(t) - u_i(t-\Delta t)\right). \tag{2.87}$$

Substituting into equation (2.80) yields

$$\frac{1}{\Delta t^2}\int_V \rho \sum_m da_i^m(t) N^m N^n \, dV = -\int_V \sigma_{ij} N^n_{,j} \, dV + \int_{S_T} T_i N^n \, dS + \int_V f_i N^n \, dV - \frac{1}{\Delta t^2}\int_V \rho \sum_m (a_i^m(t) - a_i^m(t-\Delta t)) N^m N^n \, dV. \tag{2.88}$$

Thus, the Jacobian for the system is

$$A_{ij}^{nm} = \delta_{ij} \frac{1}{\Delta t^2}\int_V \rho N^m N^n \, dV, \tag{2.89}$$

and using numerical quadrature in the finite-element discretization to replace the integrals with sums over the cells and quadrature points,

$$A_{ij}^{nm} = \delta_{ij} \frac{1}{\Delta t^2} \sum_{\text{vol cells}} \sum_{\text{quad pts}} \rho(x_q) N^m(x_q) N^n(x_q), \tag{2.90}$$

where $A_{ij}^{mn}$ is a $nd$ by $md$ matrix ($d$ is the dimension of the vector space), $m$ and $n$ refer to the basis functions and $i$ and $j$ are vector space components. We consider the contributions associated with the fault in section 6.4 on page 92 and with absorbing boundaries is section 6.3 on page 90.

## 2.5 Small (Finite) Strain Formulation

In some crustal deformation problems sufficient deformation may occur that the assumptions associated with infinitesimal strains no longer hold. This is often the case for problems when one wants to include the effects of gravitational body forces on vertical deformation. In such cases we want to account for both rigid body motion and small strains. We use a total Lagrangian formulation (quantities are associated with the undeformed configuration) based on the one presented by Bathe [Bathe, 1995].

Starting from the governing equation, written for the deformed configuration (denoted by the subscript $t$), we have

$$\int (\nabla_t \cdot \underline{\sigma}) \cdot \vec{\phi} \, dV_t + \int_{V_t} \vec{f}_t \cdot \vec{\phi} \, dV_t - \int_{V_t} \rho_t \frac{\partial^2 \vec{u}}{\partial t^2} \cdot \vec{\phi} \, dV_t = 0. \tag{2.91}$$

For the total Lagrangian formulation we want to transform these integrals over the deformed configuration to integrals over the undeformed configuration. We require that the deformed and undeformed configurations use the same coordinate system (origin and orientation). Conservation of mass requires that $\rho \, dV_t = \rho_0 \, dV_0$. We define the body force as a force per unit volume that does not depend on the configuration, which leads to $\vec{f}_t \, dV_t = \vec{f}_0 \, dV_0$.

The Green-Lagrange strain provides a measure of the strain relative to the original, undeformed configuration.

$$\underline{\epsilon} = \frac{1}{2}(\nabla u_{i,j} + (\nabla u)^T + u_{k,i} u_{k,j}), \text{ or} \tag{2.92}$$

$$\underline{\epsilon} = \underline{X}_0^T \underline{X}_0 - \underline{I}, \text{ where} \tag{2.93}$$

$$\underline{X_0} = \frac{\partial}{\partial x_j}(\vec{x}(0) + \vec{u}(t)), \tag{2.94}$$

and $\underline{X}$ is the deformation gradient tensor. The second Piola-Kirchhoff stress tensor, $\underline{S}$, is the work conjugate of the Green-Lagrange strain tensor. As a result, they are related through the elasticity constants,

$$\underline{S} = \underline{C}\,\varepsilon, \tag{2.95}$$

in the same manner as the Cauchy stress is related to the infinitesimal strain. The Cauchy stress is related to the second Piola-Kirchoff stress through the deformation gradient tensor,

$$\underline{\sigma} = \frac{1}{|\underline{X}_0|}\underline{X}_0\,\underline{S}\,\underline{X}_0^T, \tag{2.96}$$

where $det(\underline{X}_0) = |\underline{X}_0|$. Additionally, the first Piola-Kirhoff stress is define to be

$$\underline{P} = \underline{S}\,\underline{X}_0^T. \tag{2.97}$$

Applying the divergence theorem, making use of the fact that $dV_t = |\underline{X}_0|\,dV_0$, and recognizing that the gradient in the deformed configuration is related to the gradient in the undeformed configuration through the deformation gradient tensor, we can show that

$$\int_{V_t} \nabla_t \cdot \underline{\sigma} \cdot \vec{\phi}\, dV_t = -\int_{V_0} \underline{P} : \nabla\vec{\phi}\, dV_0 + \int_{S_0} \vec{T_0} \cdot \vec{\phi}\, dS_0, \tag{2.98}$$

where we assume the the tractions on the boundary do not depend on the configuration. That is, the normal and share traction components are defined in terms of the undeformed configuration. Incorporating the other relationships between the underformed and deformed configurations allows us to rewrite Equation 2.91 on the facing page in the undeformed configuration,

$$-\int_{V_0} \underline{P} : \nabla\vec{\phi}\, dV_0 + \int_{S_0} \vec{T_0} \cdot \vec{\phi}\, dS_0 + \int_{V_0} \vec{f_0} \cdot \vec{\phi}\, dV_0 - \int_{V_0} \rho_0 \frac{\partial^2 \vec{u}}{\partial t^2} \cdot \vec{\phi}\, dV_0 = 0. \tag{2.99}$$

### 2.5.1 Quasi-static Problems

The system Jacobian for quasi-static problems includes terms associated with elasticity. For the small strain formulation, we write the elasticity term at time $t+\Delta t$ and consider the first terms of the Taylor series expansion,

$$\int_v S_{ij}(t+\Delta t)\delta\varepsilon_{ij}(t+\Delta t)\, dV = \int_V (S_{ij}(t)\delta\varepsilon_{ij}(t) + dS_{ij}(t)\delta\varepsilon_{ij}(t) + S_{ij}(t)d\delta\varepsilon_{ij}(t))\, dV. \tag{2.100}$$

We approximate the increment in the stress tensor using the elastic constants,

$$dS_{ij} = C_{ijkl}d\varepsilon_{kl}, \tag{2.101}$$

and the increment in the “virtual” strain via

$$d\delta\varepsilon_{ij} = \frac{1}{2}(du_{k,i}\delta u_{k,j} + du_{k,j}\delta u_{k,i}). \tag{2.102}$$

We associate the system Jacobian with the terms involving the increment in displacements. After substituting in the expressions for the increment in the stresses and the increment in the “virtual” strains, we have

$$A_{ij}^{nm} = \int_V \frac{1}{4}C_{ijkl}(N_{,k}^m + (\sum_r a_p^r N_{,l}^r)N_{,k}^m)(N_{,i}^n + (\sum_r a_p^r N_{,j}^r)N_{,i}^n) + \frac{1}{2}S_{kl}N_{,l}^m N_{,l}^n \delta_{ij}\, dV. \tag{2.103}$$

The small strain formulation produces additional terms associated with the elastic constants and a new term associated with the stress tensor.

### 2.5.2 Dynamic Problems

The system Jacobian matrix in dynamic problems does not include any terms associated with elasticity, so the system Jacobian matrix in the small strain formulation matches the one used in the infinitesimal strain formulation.

# Chapter 3

# Installation and Getting Help

Figure 3.1 provides a guide to select the appropriate method for installing PyLith. Installation of PyLith on a desktop or laptop machine is, in most cases, very easy. Binary packages have been created for Linux and Mac OS X (Darwin) platforms. For Windows 10 users, we recommend installing the Windows Subsystem for Linux and using the Linux binary (see instructions in Section 3.1.2). You can also run PyLith inside a Docker container, which provides a virtual Linux environment on any platform that Docker supports, including Linux, Mac OS X, and Windows. Installation of PyLith on other operating systems – or installation on a cluster – requires building the software from the source code, which can be difficult for inexperienced users. We have created a small utility called PyLith Installer that makes installing PyLith and all of its dependencies from source much easier.

Help for installing and using PyLith is available from both a CIG mailing list and the GitHub issue tracking system `https://github.com/geodynamics/pylith/issues`. See Section 3.6 on page 26 for more information.

## 3.1 Installation of Binary Executable

The binaries are intended for users running on laptops or desktop computers (as opposed to clusters). The binaries contain the compilers and header files, so users wishing to extend the code can still use the binary and do not need to build PyLith and its dependencies from source. See Chapter 9 on page 253 for more information on extending PyLith.

Binary executables are available for Linux (glibc 2.12 and later) and Mac OS X (Intel 10.10 and later) from the PyLith web page `geodynamics.org/cig/software/packages/short/pylith/`. Users running Windows 10 build 14316 and later can install a Linux bash environment and use the PyLith binary for Linux (see Section 3.1.2 on page 19 for more information).

> ★ **Tip**
>
> On Linux systems you can check which version of glibc you have by running `ldd -version`

.

> ★ **Tip**
>
> On Darwin systems running OS X, you can check the operating system version by clicking on the Apple icon and About this Mac.

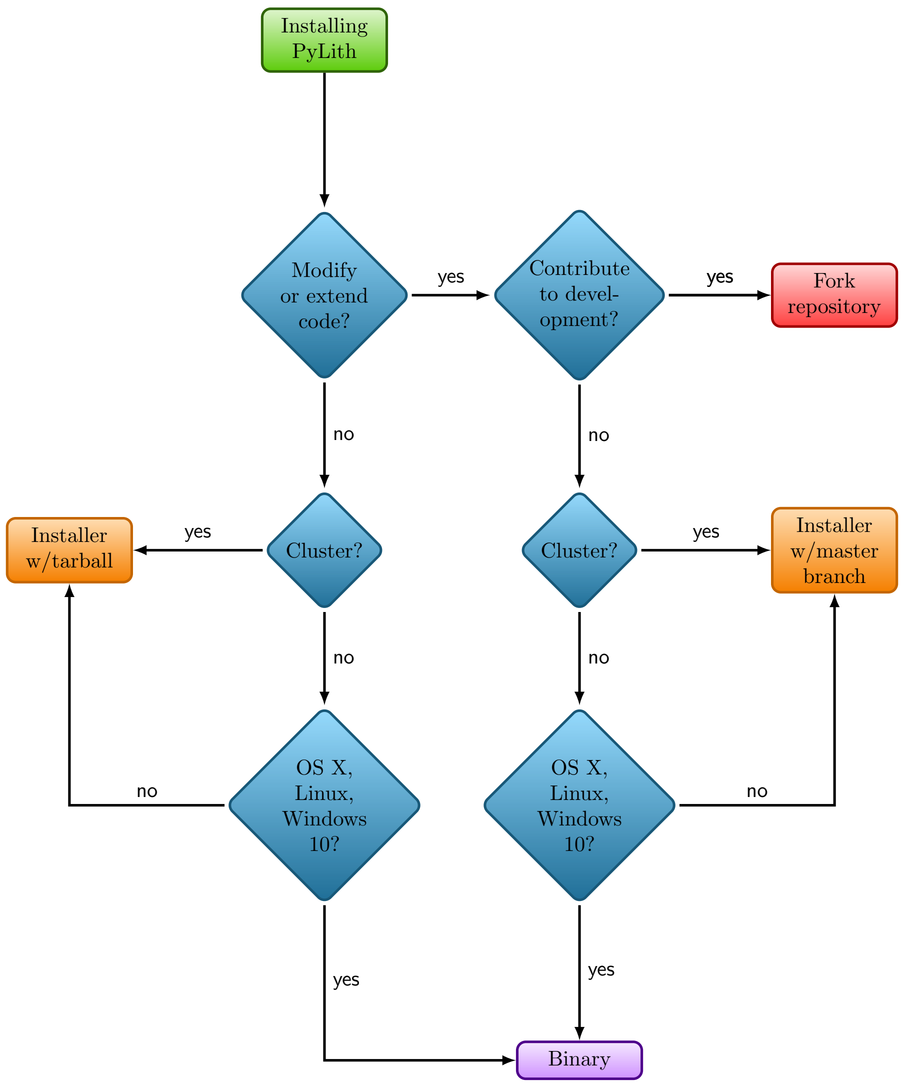



Figure 3.1: Guide for selecting the appropriate installation choice based on a hardware and intended use. The installation options are discussed in more detail in the following sections.

### 3.1.1 Linux and Mac OS X (Darwin)

1. Open a terminal window and change to the directory where you want to place the distribution.

```
$ cd  $HOME
$ mkdir pylith
$ cd pylith
```

2. Download the Linux or Mac OS X (Darwin) tarball from the PyLith web page geodynamics.org/cig/software/packages/short/pylith/, and save it to the desired location, e.g., `$HOME/pylith`.

3. Unpack the tarball.

```
# Linux 32-bit
$ tar -xzf pylith-2.2.1-linux-i686.tgz
# Linux 64-bit
$ tar -xzf pylith-2.2.1-linux-x86_64.tgz
# Mac OS X
$ tar -xzf pylith-2.2.1-darwin-10.11.6.tgz
```

4. Set environment variables. The provided `setup.sh` script only works if you are using bash shell. If you are using a different shell, you will need to alter how the environment variables are set in `setup.sh`.

```
$ source setup.sh
```

**Warning**

The binary distribution contains PyLith and all of its dependencies. If you have any of this software already installed on your system, you need to be careful in setting up your environment so that preexisting software does not conflict with the PyLith binary. By default the `setup.sh` script will prepend to the PATH and PYTHONPATH (for Darwin and Linux) and LD_LIBRARY_PATH (for Linux) environment variables. This will prevent most conflicts.

**Warning**

The PyLith binary distribution for **Darwin** systems is built using the system clang compiler suite and the system Python. **This means the system Python must be in your path to use the PyLith binary executable**; ensure `/bin` and `/usr/bin` are at the beginning of the PATH environment variable, which is done automatically if you use the `setup.sh` script. **This condition is often violated if you have Python installed from Anaconda, HomeBrew, MacPorts, etc. and set the PATH variable in your bash configuration file.**

### 3.1.2 Windows 10

PyLith is developed within the Unix/Linux framework, and we do not provide a native PyLith binary distribution for Windows. The preferred approach to installing PyLith on a computer running Windows 10 is to enable use of a Linux subsystem. This permits use of the PyLith Linux x86_64 binary within the bash environment.

To enable the Linux subsystem on Windows 10 build 14316 and later (users running an earlier Windows build should use the PyLith Docker container):

1. Go to Settings → Security.
2. Under For developers select Developer mode. This step should not be required for Windows build 16215 and later.
3. Go to Control Panel → Programs → Turn Windows Features On or Off.
4. Enable Windows Subsystem for Linux and click OK.
5. Restart the computer.
6. Go to Start → bash. You will be prompted to download "Bash on Ubuntu on Windows" from the Windows Store. Create a user account and password for the bash environment.
7. Install the PyLith Linux x86 binary within the bash environment following the instructions for installing the PyLith binary for Linux. You will run PyLith within the bash environment just like you would for a Linux operating system.

### 3.1.3 Extending PyLith and/or Integrating Other Software Into PyLith

**New in v.2.2.0**

We have constructed the binary package so that you can extend PyLith and/or build additional software for integration with PyLith using the binary distribution.

**Darwin** The binary package includes the header files for PyLith and all of its dependencies. Use the clang compiler and Python provided with the operating system. You will need to install XTools.

**Linux** The binary package includes the GNU compilers, Python, as well as header files for PyLith and all of its dependencies.

> ★ **Tip**
>
> We encourage anyone extending PyLith to fork the PyLith repository and build from source using the PyLith Installer Utility to facilitate contributing these features back into the CIG repository via pull requests.

## 3.2 Installation of PyLith Docker Container

As an alternative to installing a binary package, we provide a Docker container for running PyLith in a self-contained virtual environment. Docker containers provide a self-contained virtual environment that are a smaller, simpler alternative to a virtual machine. The PyLith Docker container provides a Debian Linux environment with a pre-built PyLith executable, vim text editor, iceweasel (GNU version of Firefox) web-browser, and the matplotlib Python module.

> ★ **Tip**
>
> In nearly all cases, installing a PyLith binary provides easier integration with mesh generation and post-processing tools, so binaries are the preferred approach to using the PyLith Docker container. This installation method targets users running Windows versions earlier than Windows 10 build 14316.

### 3.2.1 Setup (first time only)

1. Install Docker (See https://www.docker.com/products/docker)

2. Create a container to store persistent user data
   This container, called pylith-data, will hold a directory where all your user data can be stored for use with PyLith within Docker. The data can persist for different versions of PyLith; that is, you can update to a newer version of PyLith and your user data will still be available. This directory is not directly accessible from your host computer. However, you can copy files to/from your host filesystem using "docker cp" (see below).

```
# Create the container
$ docker create --name pylith-data geodynamics/pylith-data
# Run the docker container and copy examples to the persistent storage.
$ docker run -ti --volumes-from pylith-data geodynamics/pylith
# This next command is run WITHIN the docker container.
$ cp -R $HOME/pylith-VERSION/examples $HOME/data
```

### 3.2.2 Run Unix shell within Docker to use PyLith.

To run the container with a text only interface:

```
$ docker run -ti --volumes-from pylith-data geodynamics/pylith
```

To run the container and allow display of windows on the host computer (requires that X-Windows be installed):

```
# Darwin: Allow X connections
$ xhost +YOUR_IP_ADDRESS; DISPLAY=YOUR_IP_ADDRESS:0
# Linux: Allow X connections
$ xhost +local:root
# For Linux and Darwin, continue with the follow lines.
$ XSOCK=/tmp/.X11-unix
$ docker run -ti --volumes-from pylith-data \
    -e DISPLAY=$DISPLAY -v $XSOCK:$XSOCK geodynamics/pylith
```

In addition to a minimalist Debian Linux distribution and PyLith and all of its dependencies, the container includes the following useful utilities:

**vim** Lightweight text editor

**matplotlib** Python plotting module

**iceweasel** GNU version of Firefox

> ⚠ **Important**
>
> We do not yet include ParaView due to difficulties associated with setting up rendering on the host display outside the container. You will need to copy the output files to your host machine to view them in ParaView as described later.

#### 3.2.2.1 Using Docker containers

- To "pause" a container: `Control-p Control-q`
- To attach to a "paused" or "running" container.

```
# Get the container id.
$ docker ps
# Attach to the container
$ docker attach CONTAINER_ID
```

- To restart an existing container after it exited.

```
# Get the container id.
$ docker ps -a
# Start and then attach to the container
$ docker run CONTAINER_ID
$ docker attach CONTAINER_ID
```

### 3.2.3 Copy data to/from persistent storage volume.

These commands are run on the local host outside the container, not inside the Docker container. These commands are used to move files from your host machine into the PyLith Docker container and vice versa. For example, you will generate your mesh on the host, copy the mesh file into the Docker container, run PyLith within the container, and then copy the output files to the host to display in ParaView.

```
# Copy data FROM persistent storage volume TO local host
$ docker cp pylith-data:/data/pylith-user/PATH/FILENAME LOCAL_PATH
# Copy data FROM local host TO persistent storage volume
$ docker cp LOCAL_PATH pylith-data:/data/pylith-user/PATH/
```

### 3.2.4 Docker Quick Reference

```
# List local docker images.
$ docker images
# List all docker containers.
$ docker ps -a
# List running docker containers.
$ docker ps
# Remove docker container
$ docker rm CONTAINER_ID
# Remove docker image
$ docker rmi IMAGE_ID
```

## 3.3 Installation from Source

PyLith depends on a number of other packages (see Figure 1.2 on page 3). This complicates building the software from the source code. In many cases some of the packages required by PyLith are available as binary packages. On the one hand, using the binary packages for the dependencies removes the burden of configuring, building, and installing these dependencies, but that can come with its own host of complications if consistent compiler and configuration settings are not used across all of the packages on which PyLith depends. This is usually not an issue with Linux distributions, such as Fedora, Ubuntu, and Debian that have good quality control; it can be an issue with Darwin package managers, such as Fink, MacPorts, and Homebrew, where there is limited enforcement of consistency across packages. Nevertheless, PyLith can be built on most systems provided the instructions are followed carefully. PyLith is developed and tested on Linux and Mac OS X.

A small utility, PyLith Installer, removes most of the obstacles in building PyLith and its dependencies from source. For each package this utility downloads the source code, configures it, builds it, and installs it. This insures that the versions of the dependencies are consistent with PyLith and that the proper configure arguments are used. The minimum requirements for using the PyLith installer are a C compiler, `tar`, and `wget` or `curl`. Detailed instructions for how to install PyLith using the installer are included in the installer distribution, which is available from the PyLith web page geodynamics.org/cig/software/packages/short/pylith/.

## 3.4 Verifying PyLith is Installed Correctly

The easiest way to verify that PyLith has been installed correctly is to run one or more of the examples supplied with the binary and source code. In the binary distribution, the examples are located in `src/pylith-2.2.1/examples` while in the source distribution, they are located in `pylith-2.2.1/examples`. Chapter 7 on page 111 discusses how to run and visualize the results for the examples. To run the example discussed in Section 7.9.5 on page 141:

```
$ cd examples/3d/hex8
$ pylith step01.cfg
# A bunch of stuff will be written to stdout. The last few lines should be:
WARNING! There are options you set that were not used!
WARNING! could be spelling mistake, etc!
Option left: name:-snes_atol value: 1.0e-9
Option left: name:-snes_converged_reason (no value)
Option left: name:-snes_error_if_not_converged (no value)
Option left: name:-snes_linesearch_monitor (no value)
Option left: name:-snes_max_it value: 100
Option left: name:-snes_monitor (no value)
Option left: name:-snes_rtol value: 1.0e-10
```

If you run PyLith in a directory without any input, you will get the error message:

```
$ pylith
 >> {default}::
 -- pyre.inventory(error)
 -- meshimporter.meshioascii.filename <- ''
 -- Filename for ASCII input mesh not specified.
    To test PyLith, run an example as discussed in the manual.
 >> {default}::
 -- pyre.inventory(error)
 -- timedependent.homogeneous.elasticisotropic3d.label <- ''
 -- Descriptive label for material not specified.
 >> {default}::
 -- pyre.inventory(error)
 -- timedependent.homogeneous.elasticisotropic3d.simpledb.label <- ''
 -- Descriptive label for spatial database not specified.
 >> {default}::
 -- pyre.inventory(error)
 -- timedependent.homogeneous.elasticisotropic3d.simpledb.simpleioascii.filename <- ''
 -- Filename for spatial database not specified.
pylithapp: configuration error(s)
```

This indicates that a number of default settings must be set in order to run PyLith, including setting the filename for the finite-element mesh.

## 3.5 Configuration on a Cluster

If you are installing PyLith on a cluster with a batch system, you can configure Pyre such that the `pylith` command automatically submits jobs to the batch queue. Pyre contains support for the LSF, PBS, SGE, and Globus batch systems.

The command to submit a batch job depends upon the particular batch system used. Further, the command used in a batch script to launch an MPI program varies from one cluster to the next. This command can vary between two clusters, even if the clusters use the same batch system! On some systems, `mpirun` is invoked directly from the batch script. On others, a special wrapper is used instead.

Properly configured, Pyre can handle job submissions automatically, insulating users from the details of the batch system and the site configuration. This feature has the most value when the system administrator installs a global Pyre configuration file on the cluster (under /etc/pythia-0.8), for the benefit of all users and all Pyre-based applications.

### 3.5.1 Launchers and Schedulers

If you have used one of the batch systems, you will know that the batch system requires you to write a script to launch a job. Fortunately, launching a parallel PyLith job is simplified by Pyre's launcher and **scheduler** facilities. Many properties associated with **launcher** and **scheduler** are pertinent to the cluster you are on, and are best customized in a configuration file. Your personal PyLith configuration file ($HOME/.pyre/pylithapp/pylithapp.cfg) is suitable for this purpose. On a cluster, the ideal setup is to install a system-wide configuration file under /etc/pythia-0.8, for the benefit of all users.

Pyre's **scheduler** facility is used to specify the type of batch system you are using (if any):

```
[pylithapp]
# The valid values for scheduler are 'lsf", 'pbs', 'globus', and 'none.
scheduler = lsf
# Pyre's launcher facility is used to specify the MPI implementation.
# The valid values for launcher include 'mpich' and 'lam-mpi'.
launcher = mpich
```

You may find the 'dry' option useful while debugging the **launcher** and **scheduler** configuration. This option causes PyLith to perform a "dry run," dumping the batch script or mpirun command to the console, instead of actually submitting it for execution (the output is only meaningful if you're using a batch system).

```
# Display the bash script that would be submitted.
$ pylith --scheduler.dry
# Display the mpirun command.
$ pylith --launcher.dry
```

### 3.5.2 Running without a Batch System

On a cluster without a batch system, you need to explicitly specify the machines on which the job will run. Supposing the machines on your cluster are named n001, n002, . . . , etc., but you want to run the job on machines n001, n003, n004, and n005 (maybe n002 is down for the moment). To run an example, create a file named mymachines.cfg which specifies the machines to use:

```
[pylithapp.launcher]
nodegen = n%03d
nodelist = [1,3-5]
```

The **nodegen** property is a printf-style format string, used in conjunction with **nodelist** to generate the list of machine names. The nodelist property is a comma-separated list of machine names in square brackets.

Now, invoke the following:

```
$ pylith example.cfg mymachines.cfg
```

This strategy gives you the flexibility to create an assortment of cfg files (with one cfg file for each machine list) which can be easily paired with different parameter files.

If your machine list does not change often, you may find it more convenient to specify default values for **nodegen** and **nodelist** in $HOME/.pyre/pylithapp/pylithapp.cfg (which is read automatically). Then, you can run any simulation with no additional arguments:

```
$ pylith example.cfg
```

**Warning**

This assumes your machine list has enough nodes for the simulation in question.

You will notice that a machine file `mpirun.nodes` is generated. It will contain a list of the nodes where PyLith has run.

### 3.5.3 Using a Batch System

Many clusters use some implementation of a PBS (e.g., TORQUE/Maui) or LSF batch system. The examples below illustrate use of some of the more important settings. You may need to make use of more options or adjust these to submit jobs on various cluster. These settings are usually placed in `$HOME/.pyre/pylithapp/pylithapp.cfg` or in a system-wide configuration file. They can be overridden on the command line, where one typically specifies the number of compute nodes and number of processes per compute node, the job name, and the allotted time for the job:

```
$ pylith example1.cfg \
    --job.queue=debug \
    --job.name=example1 \
    --job.stdout=example1.log \
    --job.stderr=example1.err \
    --job.walltime=5*minute \
    --nodes=4
```

**Important**

The value for nodes is equal to the number of compute nodes times the number of processes (usually the number of cores) requested per compute node. Specifying the number of processes per compute node depends on the batch system. For more information on configuring Pyre for your batch system, see CIG's Pythia page geodynamics.org/cig/software/packages/cs/pythia.

#### 3.5.3.1 LSF Batch System

```
[pylithapp]
scheduler = lsf    ; the type of batch system

[pylithapp.lsf]
bsub-options = [-a mpich_gm]    ; special options for 'bsub'

[pylithapp.launcher]
command = mpirun.lsf    ; 'mpirun' command to use on our cluster

[pylithapp.job]
queue = normal    ; default queue for jobs
```

#### 3.5.3.2 PBS Batch System

```
[pylithapp]
scheduler = pbs     ; the type of batch system

[pylithapp.pbs]
```

```
shell = /bin/bash     ; submit the job using a bash shell script

# Export all environment variables to the batch job
# Send email to johndoe@mydomain.org when the job begins, ends, or aborts
qsub-options = -V -m bea -M johndoe@mydomain.org

[pylithapp.launcher]
command = mpirun -np ${nodes} -machinefile ${PBS_NODEFILE}
```

For most PBS batch systems you can specify N processes per compute node via the command line argument `--scheduler.ppn=N`.

## 3.6 Getting Help and Reporting Bugs

The CIG Short-Term Crustal Dynamics Mailing List `cig-short@geodynamics.org` is dedicated to CIG issues associated with short-term crustal dynamics, including the use of PyLith. You can subscribe to the mailing list and view messages at cig-short Mailing List `geodynamics.org/cig/lists/cig-short`.

CIG uses GitHub for source control and bug tracking. If you find a bug in PyLith, please submit a bug report to the GitHub issue tracking system for PyLith `https://github.com/geodynamics/pylith/issues`. Of course, it is helpful to first check to see if someone else already submitted a report related to the issue; one of the CIG developers may have posted a work around to the problem. You can reply to a current issue by clicking on the issue title. To submit a new issue, click on the New Issue button.

# Chapter 4

# Running PyLith

Figure 4.1 on the next page shows the workflow for running PyLith. There are essentially three main inputs needed to run a problem with PyLith:

1. Mesh information. This includes the topology of the finite-element mesh (coordinates of vertices and how the vertices are connected into cells), a material identifier for each cell, and sets of vertices associated with boundary conditions, faults, and output (for subsets of the mesh). This information can be provided using the PyLith mesh ASCII format (see Chapter 7 on page 111 for examples and Section C.1 on page 267 for the format specification) or by importing the information from the LaGriT or CUBIT meshing packages (see Chapter 7 on page 111 for examples).
2. A set of parameters describing the problem. These parameters describe the type of problem to be run, solver information, time-stepping information, boundary conditions, materials, etc. This information can be provided from the command-line or by using a `cfg` file.
3. Databases specifying the material property values and boundary condition values to be used. Arbitrarily complex spatial variations in boundary and fault conditions and material properties may be given in the spatial database (see Chapter 7 on page 111 for examples and Appendix C.2 on page 268 for the format specification).

PyLith writes solution information, such as solution fields and state variables, to either VTK files or HDF5/Xdmf files. ParaView and Visit can read both types of files. Post-processing of output is generally performed using HDF5 files accessed via a Python script and the h5py package or a Matlab script.

## 4.1 Defining the Simulation

The parameters for PyLith are specified as a hierarchy or tree of modules. The application assembles the hierarchy of modules from user input and then calls the main function in the top-level module in the same manner as a C or C++ program. The behavior of the application is determined by the modules included in the hierarchy as specified by the user. The Pyre framework provides the interface for defining this hierarchy. Pyre properties correspond to simple settings in the form of strings, integers, and real numbers. Pyre facilities correspond to software modules. Facilities may have their own facilities (branches in the tree) and any number of properties. See Figure 1.3 on page 4 for the general concept of Pyre facilities and properties. The top-level object is the PyLith application with three facilities: **`mesher`**, **`problem`**, and **`petsc`**. The **`mesher`** specifies how to import the mesh, the **`problem`** specifies the physical properties, boundary conditions, etc., and **`petsc`** is used to specify PETSc settings. Appendix B on page 261 contains a list of the components provided by PyLith and spatialdata.

### 4.1.1 Setting PyLith Parameters

There are several methods for setting input parameters for the `pylith` executable: via the command line or by using a text file in `cfg` or `pml` format. Both facilities and properties have default values provided, so you only need to set values when you

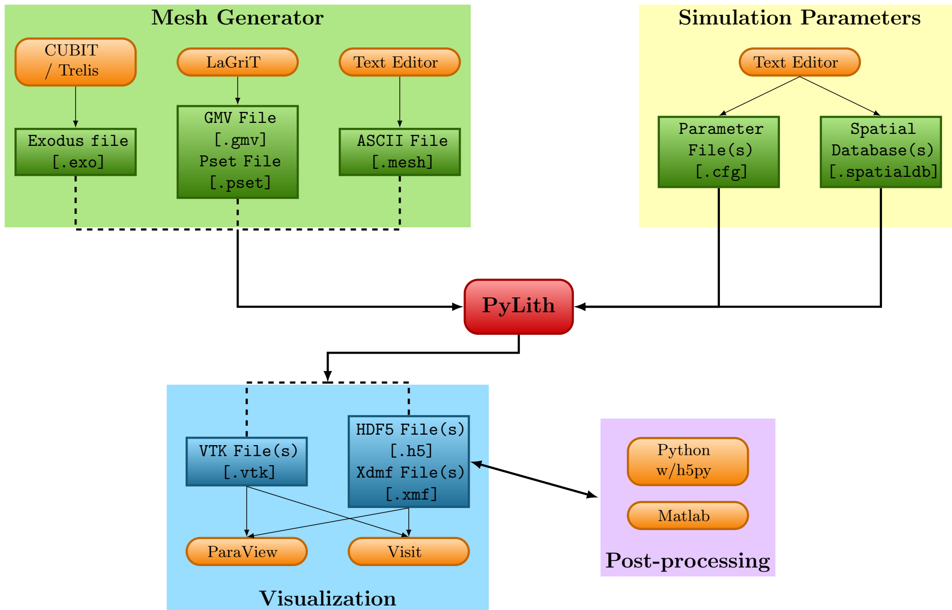



Figure 4.1: PyLith requires a finite-element mesh (three different mechanisms for generating a mesh are currently supported), simulation parameters, and spatial databases (defining the spatial variation of various parameters). PyLith writes the solution output to either VTK or HDF5/Xdmf files, which can be visualized with ParaView or Visit. Post-processing is generally done using the HDF5 files with Python or Matlab scripts.

want to deviate from the default behavior.

#### 4.1.1.1 Units

All dimensional parameters require units. The units are specified using Python and FORTRAN syntax, so square meters is m**2. Whitespace is not allowed in the string, for units and dimensioned quantities are multiplied by the units string; for example, two meters per second is 2.0*m/s. Available units are shown in Table 4.1

Table 4.1: Pyre supported units. Aliases are in parentheses.

| **Scale** | **Available Units** |
|---|---|
| length | meter (m), micrometer (um, micron), millimeter (mm), centimeter (cm), kilometer (km), inch, foot, yard, mile |
| time | second (s), nanosecond (ns), microsecond (us), millisecond (ms), minute, hour, day, year |
| mass | kilogram (kg), gram (g), centigram (cg), milligram (mg), ounce, pound, ton |
| pressure | pascal (Pa), kPa, MPa, GPa, bar, millibar, atmosphere (atm) |

#### 4.1.1.2 Using the Command Line

The `--help` command line argument displays links to useful resources for learning PyLith.

Pyre uses the following syntax to change properties from the command line. To change the value of a property of a component, use `--COMPONENT.PROPERTY=VALUE`. Each component is attached to a facility, so the option above can also be written as `--FACILITY.PROPERTY=VALUE`. Each facility has a default component attached to it. A different component can be attached to a facility by `--FACILITY=NEW_COMPONENT`.

PyLith's command-line arguments can control Pyre and PyLith properties and facilities, MPI settings, and PETSc settings. All PyLith-related properties are associated with the **`pylithapp`** component. You can get a list of all of these top-level properties along with a description of what they do by running PyLith with the `--help-properties` command-line argument.

To get information on user-configurable facilities and components, you can run PyLith with the `--help-components` command-line argument. To find out about the properties associated with a given component, you can run PyLith with the `--COMPONENT.help-properties` flag:

```
$ pylith --problem.help-properties

# Show problem components.
$ pylith --problem.help-components

# Show bc components (bc is a component of problem).
$ pylith --problem.bc.help-components

# Show bc properties.
$ pylith --problem.bc.help-properties
```

#### 4.1.1.3 Using a `.cfg` File

Entering all those parameters via the command line involves the risk of typographical errors. You will generally find it easier to collect parameters into a `cfg` file. The file is composed of one or more sections which are formatted as follows:

```
[pylithapp.COMPONENT1.COMPONENT2]
# This is a comment.

FACILITY3 = COMPONENT3
PROPERTY1 = VALUE1
PROPERTY2 = VALUE2 ; this is another comment
```

★ **Tip**

We strongly recommend that you use `cfg` files for your work. The files are syntax-highlighted in the vim editor.

#### 4.1.1.4 Using a `.pml` File

A `pml` file is an XML file that specifies parameter values in a highly structured format. It is composed of nested sections which are formatted as follows:

```
<component~name="COMPONENT1">
    <component~name="COMPONENT2">
        <property~name="PROPERTY1">VALUE1</property>
        <property~name="PROPERTY2">VALUE2</property>
    </component>
</component>
```

XML files are intended to be read and written by machines, not edited manually by humans. The `pml` file format is intended for applications in which PyLith input files are generated by another program, e.g., a GUI, web application, or a high-level structured editor. This file format will not be discussed further here, but if you are interested in using `pml` files, note that `pml` files and `cfg` files can be used interchangeably; in the following discussion, a file with a `pml` extension can be substituted anywhere a `cfg` file can be used.

#### 4.1.1.5 Specification and Placement of Configuration Files

Configuration files may be specified on the command line:

```
$ pylith example.cfg
```

In addition, the Pyre framework searches for configuration files named `pylithapp.cfg` in several predefined locations. You may put settings in any or all of these locations, depending on the scope you want the settings to have:

1. `$PREFIX/etc/pylithapp.cfg`, for system-wide settings;
2. `$HOME/.pyre/pylithapp/pylithapp.cfg`, for user settings and preferences;
3. the current directory (`./pylithapp.cfg`), for local overrides.

> ⚠ **Important**
>
> The Pyre framework will search these directories for `cfg` files matching the names of components (for example, `timedependent.cfg`, `faultcohesivekin.cfg`, `greensfns.cfg`, `pointforce.cfg`, etc) and will attempt to assign all parameters in those files to the respective component.

> ⚠ **Important**
>
> Parameters given directly on the command line will override any input contained in a configuration file. Configuration files given on the command line override all others. The `pylithapp.cfg` files placed in (3) will override those in (2), (2) overrides (1), and (1) overrides only the built-in defaults.

All of the example problems are set up using configuration files and specific problems are defined by including the appropriate configuration file on the command-line. Referring to the directory `examples/twocells/twohex8`, we have the following.

```
$ ls -1 *.cfg
axialdisp.cfg
dislocation.cfg
pylithapp.cfg
sheardisp.cfg
```

The settings in `pylithapp.cfg` will be read automatically, and additional settings are included by specifying one of the other files on the command-line:

```
$ pylith axialdisp.cfg
```

If you want to see what settings are being used, you can either examine the `cfg` files, or use the help flags as described above:

```
# Show components for the 'problem' facility.
$ pylith axialdisp.cfg --problem.help-components

# Show properties for the 'problem' facility.
$ pylith axialdisp.cfg --problem.help-properties

# Show components for the 'bc' facility.
$ pylith axialdisp.cfg --problem.bc.help-components

# Show properties for the 'bc' facility.
$ pylith axialdisp.cfg --problem.bc.help-properties
```

This is generally a more useful way of determining problem settings, since it includes default values as well as those that have been specified in the `cfg` file.

#### 4.1.1.6 List of PyLith Parameters (**`pylithinfo`**)

The Python application `pylithinfo` writes all of the current parameters to a text file or JSON file (default). The default name of the JSON is `pylith_parameters.json`. The usage synopsis is

```
$ pylithinfo [--verbose-false] [--format={ascii,json} [--filename=pylith_parameters.json] PYLITH_ARGS
```

where `--verbose-false` turns off printing the descriptions of the properties and components as well as the location where the current value was set, `--format=ascii` changes the output format to a simple ASCII file, and `--filename=pylith_parameter` sets the name of the output file. The PyLith Parameter Viewer (see Section 4.10) provides a graphic user interface for examining the JSON parameter file.

### 4.1.2 Mesh Information (**`mesher`**)

Geometrical and topological information for the finite element mesh may be provided by exporting an Exodus II format file from CUBIT/Trelis, by exporting a GMV file and an accompanying Pset file from LaGriT, or by specifying the information in PyLith mesh ASCII format. See Chapter 7 on page 111 for examples.

PyLith supports linear cells in 2D (Figure 4.2), and 3D (Figure 4.3 on the following page). The vertex ordering must follow the convention shown in Figures 4.2- 4.3 on the following page. PyLith no longer supports use of quadratic cells using the PyLith ASCII mesh format. In the next release, we plan to support higher order discretizations via PETSc finite-element features from meshes with linear cells as input.

The mesh information defines the vertex coordinates and specifies the vertices composing each cell in the mesh. The mesh information must also define at least one set of vertices for which displacement (Dirichlet) boundary conditions will be provided. In most realistic problems, there will be several vertex groups, each with a unique identifying label. For example, one group might define a surface of the mesh where displacement (Dirichlet) boundary conditions will be applied, another might define a surface where traction (Neumann) boundary conditions will be applied, while a third might specify a surface that defines a fault. Similarly, the mesh information contains cell labels that define the material type for each cell in the mesh. For a mesh with a single material type, there will only be a single label for every cell in the mesh. See Chapters 5 on page 61 and 6 on page 85 for more detailed discussions of setting the materials and boundary conditions.

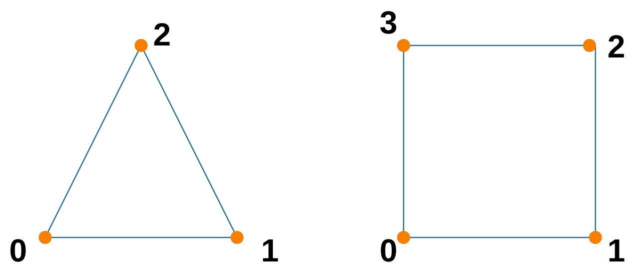



Figure 4.2: Linear cells available for 2D problems are the triangle (left) and the quadrilateral (right).

#### 4.1.2.1 Mesh Importer

The default mesher component is MeshImporter, which provides the capabilities of reading the mesh from files. The MeshImporter has several properties and facilities:

**`reorder_mesh`** Reorder the vertices and cells using the reverse Cuthill-McKee algorithm (default is False)

**`reader`** Reader for a given type of mesh (default is MeshIOAscii).

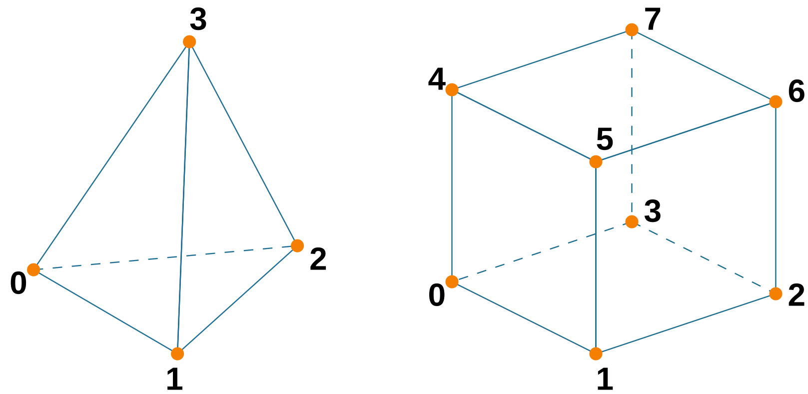



Figure 4.3: Linear cells available for 3D problems are the tetrahedron (left) and the hexahedron (right).

**distributor** Handles distribution of the mesh among processors.

**refiner** Perform global uniform mesh refinement after distribution among processors (default is no refinement).

Reordering the mesh so that vertices and cells connected topologically also reside close together in memory improves overall performance and can improve solver performance as well.

> **Warning**
>
> The coordinate system associated with the mesh must be a Cartesian coordinate system, such as a generic Cartesian coordinate system or a geographic projection.

#### 4.1.2.2 MeshIOAscii

The MeshIOAscii object is intended for reading small, simple ASCII files containing a mesh constructed by hand. We use this file format extensively in the examples. Appendix C.1 on page 267 describes the format of the files. The properties and facilities of the MeshIOAscii object include:

**filename** Name of the mesh file.

**coordsys** Coordinate system associated with the mesh.

#### 4.1.2.3 MeshIOCubit

The MeshIOCubit object reads the NetCDF Exodus II files output from CUBIT/Trelis. Beginning with CUBIT 11.0, the names of the nodesets are included in the Exodus II files and PyLith can use these nodeset names or revert to using the nodeset ids. The properties and facilities associated with the MeshIOCubit object are:

**filename** Name of the Exodus II file.

**use_nodeset_names** Identify nodesets by name rather than id (default is True).

**coordsys** Coordinate system associated with the mesh.

#### 4.1.2.4 MeshIOLagrit

The MeshIOLagrit object is used to read ASCII and binary GMV and PSET files output from LaGriT. PyLith will automatically detect whether the files are ASCII or binary. We attempt to provide support for experimental 64-bit versions of LaGriT via flags indicating whether the FORTRAN code is using 32-bit or 64-bit integers. The MeshIOLagrit properties and facilities are:

**filename_gmv** Name of GMV file.

**filename_pset** Name of the PSET file.

**flip_endian** Flip the endian of values when reading binary files (default is False).

**io_int32** Flag indicating that PSET files use 32-bit integers (default is True).

**record_header_32bt** Flag indicating FORTRAN record header is 32-bit (default is True).

**coordsys** Coordinate system associated with mesh.

> **Warning**
>
> The PyLith developers have not used LaGriT since around 2008 and the most recent release appears to have been in 2010.

#### 4.1.2.5 Distributor

The distributor uses a partitioner to compute which cells should be placed on each processor, computes the overlap among the processors, and then distributes the mesh among the processors. The type of partitioner is set via PETSc settings. The properties and facilities of the Distributor include:

**partitioner** Name of mesh partitioner ['chaco','parmetis'].

**write_partition** Flag indicating that the partition information should be written to a file (default is False).

**data_writer** Writer for partition information (default is DataWriterVTK for VTK output).

Distributor parameters in a `cfg` file

```
[pylithapp.mesh_generator.distributor]
partitioner = chaco ; Options are 'chaco' (default) and 'parmetis'.
```

METIS/ParMETIS are not included in the PyLith binaries due to licensing issues.

#### 4.1.2.6 Refiner

The refiner is used to decrease node spacing by a power of two by recursively subdividing each cell by a factor of two. In a 2D triangular mesh a node is inserted at the midpoint of each edge, splitting each cell into four cells (see Figure 4.4 on the following page). In a 2D quadrilateral mesh a node is inserted at the midpoint of each edge and at the centroid of the cell, splitting each cell into four cells. In a 3D tetrahedral mesh a node is inserted at the midpoint of each edge, splitting each cell into eight cells. In a 3D hexahedral mesh a node is inserted at the midpoint of each edge, the centroid of each face, and at the centroid of the cell, splitting each cell into eight cells.

Refinement occurs after distribution of the mesh among processors. This allows one to run much larger simulations by (1) permitting the mesh generator to construct a mesh with a node spacing largeer than that needed in the simulation and (2) operations performed in serial during the simulation setup phase, such as, adjusting the topology to insert cohesive cells and distribution of the mesh among processors uses this much smaller coarse mesh. For 2D problems the global mesh refinement increases the maximum problem size by a factor of $4^n$, and for 3D problems it increases the maximum problem size by a factor of $8^n$, where $n$ is the number of recursive refinement levels. For a tetrahedral mesh, the element quality decreases with refinement so $n$ should be limited to 1-2.

### 4.1.3 Problem Specification (`problem`)

The problem component specifies the basic parameters of the simulation, including the physical properties, the boundary conditions, and interface conditions (faults). The current release of PyLith contains two types of problems, TimeDependent

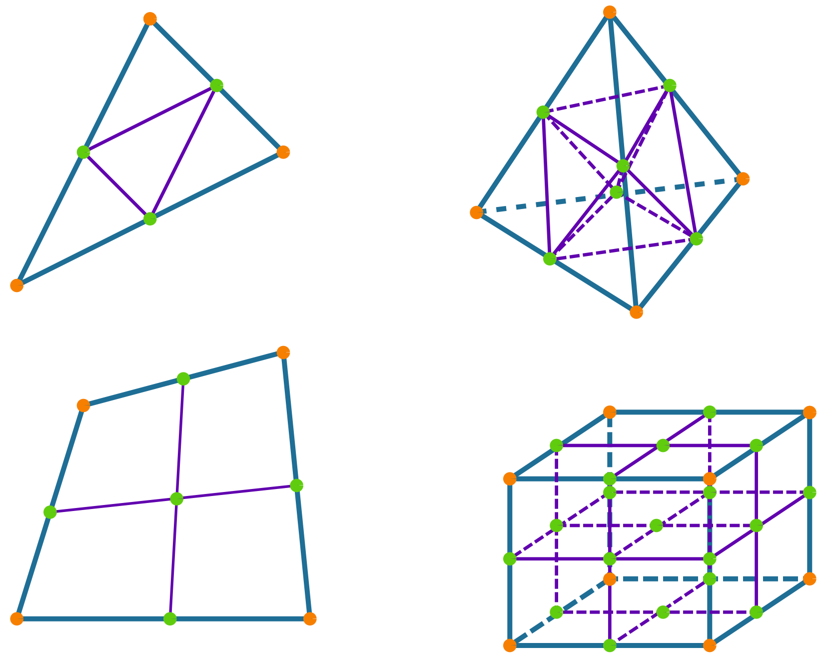


Figure 4.4: Global uniform mesh refinement of 2D and 3D linear cells. The blue lines and orange circles identify the edges and vertices in the original cells. The purple lines and green circles identify the new edges and vertices added to the original cells to refine the mesh by a factor of two.

for use in static, quasi-static, and dynamic simulations and GreensFns for computing static Green's functions. The general properties facilities include:

**dimension** Spatial dimension of problem space.

**normalizer** Scales used to nondimensionalize the problem (default is NondimElasticQuasistatic).

**materials** Array of materials comprising the domain (default is [material]).

**bc** Array of boundary conditions (default is none).

**interfaces** Array of interface conditions, i.e., faults (default is none).

**gravity_field** Gravity field used to construct body forces (default is none).

**progress_ monitor** Show progress of running simulation.

Problem parameters in a `cfg` file

```
[pylithapp.timedependent]
dimension = 3
normalizer = spatialdata.units.NondimElasticQuasistatic
materials = [elastic, viscoelastic]
bc = [boundary_east, boundary_bottom, boundary_west]
interfaces = [SanAndreas, SanJacinto]
gravity_field = spatialdata.spatialdb.GravityField
```

#### 4.1.3.1 Nondimensionalization (normalizer)

PyLith nondimensionalizes all parameters provided by the user so that the simulation solves the equations using nondimensional quantities. This permits application of PyLith to problems across a vast range of spatial and temporal scales. The scales used to nondimensionalize the problem are length, pressure, density, and time. PyLith provides two normalizer objects to make it easy to provide reasonable scales for the nondimensionalization. The NondimElasticQuasistatic normalizer (which is the default) has the following properties:

**length_scale** Distance to nondimensionalize length (default is 1.0 km).

**shear_modulus** Shear modulus to nondimensionalize pressure (default is 3.0e+10 Pa).

**relaxation_time** Relaxation time to nondimensionalize time (default is 1.0 year).

NondimElasticQuasistatic parameters in a cfg file

```
[pylithapp.timedependent.normalizer]
length_scale = 1.0*km
shear_modulus = 3.0e+10*Pa
relaxation_time = 1.0*yr
```

The NondimElasticDynamic normalizer has the following properties:

**shear_wave_speed** Shear wave speed used to nondimensionalize length and pressure (default is 3.0 km/s).

**mass_density** Mass density to nondimensionalize density and pressure (default is 3.0e+3 kg/m$^3$).

**wave_period** Period of seismic waves used to nondimensionalize time (default is 1.0 s).

NondimElasticDynamic parameters in a cfg file

```
[pylithapp.timedependent.normalizer]
shear_wave_speed = 3.0*km/s
mass_density = 3.0e+3*kg/m**3
wave_period = 1.0*s
```

**Important**

The default nondimensionalization is reasonable for many problems; however, it may be necessary to change the default values in some cases. When doing this, keep in mind that the nondimensionalization generally applies to the minimum values encountered for a problem. For example, in a quasistatic problem, the **length_scale** should be on the order of the minimum cell size. Similarly, the **relaxation_time** should be on the order of the minimum relaxation time.

### 4.1.4 Finite-Element Integration Settings

PyLith uses numerical quadrature to evaluate the finite-element integrals for the residual and system Jacobian (see Chapter 2 on page 7). PyLith employs FIAT (finite element automatic tabulator) to compute the basis functions and their derivatives at the quadrature points for various quadrature schemes and cell shapes. The parameters for Lagrange cells (lines, quadrilaterals, hexahedra) are specified using the FIATLagrange object, whereas the parameters for Simplex cells (lines, triangles, tetrahedra) are specified using the FIATSimplexx object. Both objects use the same set of parameters and PyLith will setup the basis functions and quadrature scheme appropriately for the two families of cells. The quadrature scheme and basis functions must be set for each material and boundary condition involving finite-element integrations (Dirichlet boundary conditions are constraints and do not involve integrations). Furthermore, the integration schemes can be set independently. The current version of PyLith supports basis functions with linear variations in the field (P1); support for higher order cells will be added in the future. The properties for the FIATLagrange and FIATSimplex objects are

**dimension** Dimension of the cell (0,1,2,3; default is 3).

**degree** Degree of the finite-element cell (default is 1).

**order** Order of quadrature rule (default is degree+1); hardwired to be equal to degree for faults.

**collocate_quad** Collocate quadrature points with vertices (default is False); hardwired to True for faults.

See Section 5.1.3 on page 62 for an example of setting these properties for a material.

### 4.1.5 PETSc Settings (`petsc`)

In quasi-static problems with implicit time-stepping, PyLith relies on PETSc for the linear algebra computations, including linear Krylov subspace solvers and nonlinear solvers. For dynamic problems, lumping the mass matrix and using explicit time-stepping is much more efficient; this permits solving the linear system with a trivial solver so we do not use a PETSc solver in this case (see Section 4.2.3 on page 40).

PETSc options can be set in `cfg` files in sections beginning with `[pylithapp.petsc]`. The options of primary interest in the case of PyLith are shown in Table 4.2 on the facing page. PETSc options are used to control the selection and settings for the solvers underlying the SolverLinear and SolverNonlinear objects discussed in Section 4.2.3 on page 40. A very wide range of elasticity problems in quasi-static simulations can be solved with reasonable runtimes by replacing the default Jacobi preconditioner with the Additive Schwarz Method (ASM) using Incomplete LU (ILU) factorization by default (see Table 4.3 on the facing page). A more advanced set of solver settings that may provide better performance in many elasticity problems are given in Table 4.4 on the next page. These are available in `$PYLITH_DIR/share/settings/solver_fault_fieldsplit.cfg`. These settings are limited to problems where we store the stiffness matrix as a nonsymmetric sparse matrix and require additional settings for the formulation,

```
[pylithapp.timedependent.formulation]
split_fields = True
use_custom_constraint_pc = True ; Use only if problem contains a fault
matrix_type = aij
```

**Important**

These settings are only available if you build PETSc with the ML package. These features are included in the PyLith binary packages.

**Warning**

The split fields and algebraic multigrid preconditioning currently fails in problems with a nonzero null space. This most often occurs when a problem contains multiple faults that extend through the entire domain and create subdomains without any Dirichlet boundary conditions. The current workaround is to use the Additive Schwarz preconditioner without split fields. See Section 4.8.2 on page 52 for the error message encountered in this situation.

These more advanced settings allow the displacement fields and Lagrange multipliers for fault tractions to be preconditioned separately. This usually results in a much stronger preconditioner. In simulations with fault slip, the degrees of freedom associated with the Lagrange multipliers should be preconditioned with a custom preconditioner that uses a diagonal approximation of the Schur complement.

#### 4.1.5.1 Model Verification with PETSc Direct Solvers

It is often useful to apply a direct solver so that solver convergence is decoupled from model verification for the purposes of testing. Unfortunately, the traditional LU factorization solvers cannot be directly applied in PyLith due to the saddle-point formulation used to accomodate the fault slip constraints. However, we can combine an LU factorization of the displacement sub-block with a full Schur complement factorization using the PETSc FieldSplit preconditioner. If the solver for the Schur complement S is given a very low tolerance, this is effectively a direct solver. The options given below will construct this solver in PyLith. These settings are available in `$PYLITH_DIR/share/settings/solver_fault_exact.cfg`.

Table 4.2: Useful command-line arguments for setting PETSc options.

| Property | Default Value | Description |
|---|---|---|
| `log_view` | *false* | Print logging objects and events. |
| `ksp_monitor` | *false* | Dump preconditioned residual norm to stdout. |
| `ksp_view` | *false* | Print linear solver parameters. |
| `ksp_rtol` | *1.0e-05* | Convergence tolerance for relative decrease in residual norm. |
| `snes_monitor` | *false* | Dump residual norm to stdout for each nonlinear solve iteration. |
| `snes_view` | *false* | Print nonlinear solver parameters. |
| `snes_rtol` | *1.0e-5* | Convergence tolerance for relative decrease in residual norm. |
| `pc_type` | *jacobi* | Set preconditioner type. See PETSc documentation for a list of all preconditioner types. |
| `ksp_type` | *gmres* | Set linear solver type. See PETSc documentation for a list of all solver types. |

Table 4.3: PETSc options that provide moderate performance in a wide range of quasi-static elasticity problems.

| Property | Value | Description |
|---|---|---|
| `pc_type` | *asm* | Additive Schwarz method. |
| `ksp_type` | *gmres* | GMRES method from Saad and Schultz. |
| `sub_pc_factor_shift_type` | *nonzero* | Turn on nonzero shifting for factorization. |
| `ksp_max_it` | *100* | Maximum number of iterations permitted in linear solve. Depends on problem size. |
| `ksp_gmres_restart` | *50* | Number of iterations after which Gram-Schmidt orthogonalization is restarted. |
| `ksp_rtol` | *1.0e-08* | Linear solve convergence tolerance for relative decrease in residual norm. |
| `ksp_atol` | 1.0e-12 | Linear solve convergence tolerance for absolute value of residual norm. |
| `ksp_converged_reason` | *true* | Indicate why iterating stopped in linear solve. |
| `snes_max_it` | *100* | Maximum number of iterations permitted in nonlinear solve. Depends on how nonlinear the problem is. |
| `snes_rtol` | *1.0e-08* | Nonlinear solve convergence tolerance for relative decrease in residual norm. |
| `snes_atol` | *1.0e-12* | Nonlinear solve convergence tolerance for absolute value of residual norm. |
| `snes_converged_reason` | *true* | Indicate why iterating stopped in nonlinear solve. |

Table 4.4: PETSc options used with split fields algebraic multigrid preconditioning that often provide improved performance in quasi-static elasticity problems with faults.

| Property | Value | Description |
|---|---|---|
| `fs_pc_type` | *field_split* | Precondition fields separately. |
| `fs_pc_use_amat` | *true* | Use diagonal blocks from the true operator, rather than the preconditioner. |
| `fs_pc_fieldsplit_type` | *multiplicative* | Apply each field preconditioning in sequence, which is stronger than all-at-once (additive). |
| `fs_fieldsplit_displacement_pc_type` | *ml* | Multilevel algebraic multigrid preconditioning using Trilinos/ML via PETSc. |
| `fs_fieldsplit_lagrange_multiplier_pc_type` | *jacobi* | Jacobi preconditioning for Lagrange multiplier block |
| `fs_fieldsplit_displacement_ksp_type` | *preonly* | Apply only the preconditioner. |
| `fs_fieldsplit_lagrange_multiplier_ksp_type` | *preonly* | Apply only the preconditioner. |

```
[pylithapp.timedependent.formulation]
split_fields = True
matrix_type = aij

[pylithapp.petsc]
fs_pc_type = fieldsplit
fs_pc_use_amat = True
fs_pc_fieldsplit_type = schur
fs_pc_fieldsplit_schur_factorization_type = full
fs_fieldsplit_displacement_ksp_type = preonly
fs_fieldsplit_displacement_pc_type = lu
fs_fieldsplit_lagrange_multiplier_pc_type = jacobi
fs_fieldsplit_lagrange_multiplier_ksp_type = gmres
fs_fieldsplit_lagrange_multiplier_ksp_rtol = 1.0e-11
```

## 4.2 Time-Dependent Problem (`formulation`)

This type of problem applies to transient static, quasi-static, and dynamic simulations. The time-dependent problem adds the `formulation` facility to the general-problem. The formulation specifies the time-stepping formulation to integrate the elasticity equation. PyLith provides several alternative formulations, each specific to a different type of problem.

**Implicit** Implicit time stepping for static and quasi-static problems with infinitesimal strains. The implicit formulation neglects inertial terms (see Section 2.65 on page 12).

**ImplicitLgDeform** Implicit time stepping for static and quasi-static problems including the effects of rigid body motion and small strains. This formulation requires the use of the nonlinear solver, which is selected automatically.

**Explicit** Explicit time stepping for dynamic problems with infinitesimal strains and lumped system Jacobian. The cell matrices are lumped before assembly, permitting use of a vector for the diagonal system Jacobian matrix. The built-in lumped solver is selected automatically.

**ExplicitLgDeform** Explicit time stepping for dynamic problems including the effects of rigid body motion and small strains. The cell matrices are lumped before assembly, permitting use of a vector for the diagonal system Jacobian matrix. The built-in lumped solver is selected automatically.

**ExplicitTri3** Optimized elasticity formulation for linear triangular cells with one point quadrature for dynamic problems with infinitesimal strains and lumped system Jacobian. The built-in lumped solver is selected automatically.

**ExplicitTet4** Optimized elasticity formulation for linear tetrahedral cells with one point quadrature for dynamic problems with infinitesimal strains and lumped system Jacobian. The built-in lumped solver is selected automatically.

In many quasi-static simulations it is convenient to compute a static problem with elastic deformation prior to computing a transient response. Up through PyLith version 1.6 this was hardwired into the Implicit Forumulation as advancing from time step $t = -\Delta t$ to $t = 0$, and it could not be turned off. PyLith now includes a property, `elastic_prestep` in the TimeDependent component to turn on/off this behavior (the default is to retain the previous behavior of computing the elastic deformation).

**Warning**

Turning off the elastic prestep calculation means the model only deforms when an *increment* in loading or deformation is applied, because the time-stepping formulation is implemented using the increment in displacement.

The TimeDependent properties and facilities include

**elastic_preset** If true, perform a static calculation with elastic behavior before time stepping (default is True).

**formulation** Formulation for solving the partial differential equation.

TimeDependent parameters in a cfg file

```
[pylithapp.timedependent]
formulation = pylith.problems.Implicit ; default
progres_monitor = pylith.problems.ProgressMonitorTime ; default
elastic_preset = True ; default
```

The formulation value can be set to the other formulations in a similar fashion.

### 4.2.1 Time-Stepping Formulation

The explicit and implicit time stepping formulations use a common set of facilities and properties. The properties and facilities include

**matrix_type** Type of PETSc matrix for the system Jacobian (sparse matrix, default is symmetric, block matrix with a block size of 1).

**view_jacobian** Flag to indicate if system Jacobian (sparse matrix) should be written to a file (default is false).

**split_fields** Split solution field into a displacement portion (fields 0..ndim-1) and a Lagrange multiplier portion (field ndim) to permit application of sophisticated PETSc preconditioners (default is false).

**time_step** Time step size specification (default is TimeStepUniform (uniform time step).

**solver** Type of solver to use (default is SolverLinear).

**output** Array of output managers for output of the solution (default is [output]).

**jacobian_viewer** Viewer to dump the system Jacobian (sparse matrix) to a file for analysis (default is PETSc binary).

Time-stepping formulation parameters in a cfg file

```
[pylithapp.timedependent.formulation]
matrix_type = sbaij ; Non-symmetric sparse matrix is 'aij'
view_jacobian = false

# Nonlinear solver is pylith.problems.SolverNonlinear
solver = pylith.problems.SolverLinear
output = [domain, ground_surface]
time_step = pylith.problems.TimeStepUniform
```

### 4.2.2 Numerical Damping in Explicit Time Stepping

In explicit time-stepping formulations for elasticity, boundary conditions and fault slip can excite short waveform elastic waves that are not accurately resolved by the discretization. We use numerical damping via an artificial viscosity[Knopoff and Ni, 2001, Day and Ely, 2002] to reduce these high frequency oscillations. In computing the strains for the elasticity term in equation 2.80 on page 13, we use an adjusted displacement rather than the actual displacement, where

$$\vec{u}^{adj}(t) = \vec{u}(t) + \eta^* \Delta t \vec{\dot{u}}(t), \tag{4.1}$$

$\vec{u}^{adj}(t)$ is the adjusted displacement at time t, $\vec{u}(t)$is the original displacement at time (t), $\eta^*$is the normalized artificial viscosity, $\Delta t$ is the time step, and $\vec{\dot{u}}(t)$ is the velocity at time $t$. The default value for the normalized artificial viscosity is 0.1. We have found values in the range 0.1-0.4 sufficiently suppress numerical noise while not excessively reducing the peak velocity. An example of setting the normalized artificial viscosity in a cfg file is

```
[pylithapp.timedependent.formulation]
norm_viscosity = 0.2
```

### 4.2.3 Solvers

PyLith supports three types of solvers. The linear solver, SolverLinear, corresponds to the PETSc KSP solver and is used in linear problems with linear elastic and viscoelastic bulk constitutive models and kinematic fault ruptures. The nonlinear solver, SolverNonlinear, corresponds to the PETSc SNES solver and is used in nonlinear problems with nonlinear viscoelastic or elastoplastic bulk constitutive models, dynamic fault ruptures, or problems involving finite strain (small strain formulation). The lumped solver (SolverLumped) is a specialized solver used with the lumped system Jacobian matrix. The options for the PETSc KSP and SNES solvers are set via the top-level PETSc options (see Section 4.1.5 on page 36 and the PETSc documentation `www.mcs.anl.gov/petsc/petsc-as/documentation/index.html`).

### 4.2.4 Time Stepping

PyLith provides three choices for controlling the time step in time-dependent simulations. These include (1) a uniform, user-specified time step (which is the default), (2) user-specified time steps (potentially nonuniform), and (3) automatically calculated (potentially nonuniform) time steps. The procedure for automatically selecting time steps requires that the material models provide a reasonable estimate of the time step for stable time integration. In general, quasi-static simulations with viscoelastic materials should use automatically calculated time steps and dynamic simulations should use a uniform, user-specified time step. Note that all three of the time stepping schemes make use of the computed stable time step (see 5.1.6 on page 65). When using user-specified time steps, the value is checked against the computed stable time step. The automatically calculated time step comes from the computed stable time step.

> **Warning**
>
> Varying the time step within a simulation requires recomputing the Jacobian of the system whenever the time step changes, which can greatly increase the runtime if the time-step size changes frequently.

#### 4.2.4.1 Uniform, User-Specified Time Step (TimeStepUniform)

With a uniform, user-specified time step, the user selects the time step that is used over the entire duration of the simulation. If this value exceeds the computed stable time step at any time, PyLith will terminate with an error. The properties for the uniform, user-specified time step are:

**total_time** Time duration for simulation (default is 0.0 s).

**start_time** Start time for simulation (default is 0.0 s).

**dt** Time step for simulation.

TimeStepUniform parameters in a `cfg` file

```
[pylithapp.problem.formulation]
time_step = pylith.problems.TimeStepUniform ; Default value

[pylithapp.problem.formulation.time_step]
total_time = 1000.0*year
dt = 0.5*year
```

#### 4.2.4.2 Nonuniform, User-Specified Time Step (TimeStepUser)

The nonuniform, user-specified, time-step implementation allows the user to specify the time steps in an ASCII file (see Section C.5 on page 273 for the format specification of the time-step file). If the total duration exceeds the time associated with the time steps, then a flag determines whether to cycle through the time steps or to use the last specified time step for the time remaining. Similar to the uniform time step, if the user-specified time step size exceeds the computed stable time step at any time, PyLith will terminate with an error. The properties for the nonuniform, user-specified time step are:

**total_time** Time duration for simulation.

**filename** Name of file with time-step sizes.

**loop_steps** If true, cycle through time steps, otherwise keep using last time-step size for any time remaining.

TimeStepUser parameters in a `cfg` file

```
[pylithapp.problem.formulation]
time_step = pylith.problems.TimeStepUser ; Change the time step algorithm

[pylithapp.problem.formulation.time_step]
total_time = 1000.0*year
filename = timesteps.txt
loop_steps = false ; Default value
```

#### 4.2.4.3 Nonuniform, Automatic Time Step (TimeStepAdapt)

This time-step implementation automatically calculates a time step size based on the constitutive model and rate of deformation. As a result, this choice for choosing the time step relies on accurate calculation of a stable time step within each finite-element cell by the constitutive models. To provide some control over the time-step selection, the user can control the frequency with which a new time step is calculated, the time step to use relative to the value determined by the constitutive models, and a maximum value for the time step. Note that the stability factor allows the computed time step size to exceed the computed stable time step. A stability factor of 1.0 would provide a time step size equal to the stable time step, while a value of 2.0 (default value) would provide a time step size equal to 1/2 the stable time step. Caution should be used when adjusting the stability factor to values less than 1.0, as the large time step size may result in inaccurate solutions. The properties for controlling the automatic time-step selection are:

**total_time** Time duration for simulation.

**max_dt** Maximum time step permitted.

**adapt_skip** Number of time steps to skip between calculating new stable time step.

**stability_factor** Safety factor for stable time step (default is 2.0).

TimeStepAdapt parameters in a `cfg` file

```
[pylithapp.problem.formulation]
time_step = pylith.problems.TimeStepAdapt ; Change the time step algorithm

[pylithapp.problem.formulation.time_step]
total_time = 1000.0*year
max_dt = 10.0*year
adapt_skip = 10 ; Default value
stability_factor = 2.0 ; Default value
```

## 4.3 Green's Functions Problem (GreensFns)

This type of problem applies to computing static Green's functions for elastic deformation. The GreensFns problem specializes the time-dependent facility to the case of static simulations with slip impulses on a fault. The default formulation is the Implicit formulation and should not be changed as the other formulations are not applicable to static Green's functions. In the output files, the deformation at each "time step" is the deformation for a different slip impulse. The properties provide the ability to select which fault to use for slip impulses. The only fault component available for use with the GreensFns problem is the FaultCohesiveImpulses component discussed in Section 6.4.6 on page 109. The GreensFns properties amd facilities include:

**fault_id** Id of fault on which to impose slip impulses.

**formulation** Formulation for solving the partial differential equation.

**progress_monitor** Simple progress monitor via text file.

GreensFns parameters in a `cfg` file

```
[pylithapp]
problem = pylith.problems.GreensFns ; Change problem type from the default

[pylithapp.greensfns]
fault_id = 100 ; Default value
formulation = pylith.problems.Implicit ; default
progres_monitor = pylith.problems.ProgressMonitorTime ; default
```

**Warning**

The GreensFns problem generates slip impulses on a fault. The current version of PyLith requires that impulses can only be applied to a single fault and the fault facility must be set to FaultCohesiveImpulses.

## 4.4 Progress Monitors

**New in v2.1.0**

The progress monitors make it easy to monitor the general progress of long simulations, especially on clusters where stdout is not always easily accessible. The progress monitors update a simulation's current progress by writing information to a text file. The information includes time stamps, percent completed, and an estimate of when the simulation will finish.

### 4.4.1 ProgressMonitorTime

This is the default progress monitor for time-stepping problems. The monitor calculates the percent completed based on the time at the current time step and the total simulated time of the simulation, not the total number of time steps (which may be unknown in simulations with adaptive time stepping). The ProgressMonitorTime properties include:

**update_percent** Frequency (in percent) of progress updates.

**filename** Name of output file.

**t_units** Units for simulation time in output.

ProgressMonitorTime parameters in a `cfg` file

```
[pylithapp.problem.progressmonitor]
update_percent = 5.0 ; default
filename = progress.txt ; default
t_units = year ; default
```

### 4.4.2 ProgressMonitorStep

This is the default progress monitor for problems with a specified number of steps, such as Green's function problems. The monitor calculates the percent completed based on the number of steps (e.g., Green's function impulses completed). The ProgressMonitorStep propertiles include:

**update_percent** Frequency (in percent) of progress updates.

**filename** Name of output file.

ProgressMonitorStep parameters in a `cfg` file

```
[pylithapp.problem.progressmonitor]
update_percent = 5.0 ; default
filename = progress.txt ; default
```

## 4.5 Databases for Boundaries, Interfaces, and Material Properties

Once the problem has been defined with PyLith parameters, and the mesh information has been provided, the final step is to specify the boundary conditions and material properties to be used. The mesh information provides labels defining sets of vertices to which boundary conditions or fault conditions will be applied, as well as cell labels that will be used to define the material type of each cell. For boundary conditions, the `cfg` file is used to associate boundary condition types and spatial databases with each vertex group (see Chapter 6 on page 85). For materials, the `cfg` file is used to associate material types and spatial databases with cells identified by the material identifier (see Figure 5.1 on page 69).

The spatial databases define how the boundary conditions or material property values vary spatially, and they can be arbitrarily complex. The simplest example for a material database would be a mesh where all the cells of a given type have uniform properties ("point" or 0D variation). A slightly more complex case would be a mesh where the cells of a given type have properties that vary linearly along a given direction ("line" or 1D variation). In more complex models, the material properties might have different values at each point in the mesh ("volume" or 3D variation). This might be the case, for example, if the material properties are provided by a database of seismic velocities and densities. For boundary conditions the simplest case would be where all vertices in a given group have the same boundary condition parameters ("point" or 0D variation). A more complex case might specify a variation in the conditions on a given surface ("area" or 2D variation). This sort of condition might be used, for example, to specify the variation of slip on a fault plane. The examples discussed in Chapter 7 on page 111 also contain more information regarding the specification and use of the spatial database files.

### 4.5.1 SimpleDB Spatial Database

In most cases the default type of spatial database for faults, boundary conditions, and materials is SimpleDB. Spatial database files provide specification of a field over some set of points. There is no topology associated with the points. Although multiple values can be specified at each point with more than one value included in a search query, the interpolation of each value will be done independently. Time dependent variations of a field are not supported in these files. Spatial database files can specify spatial variations over zero, one, two, and three dimensions. Zero dimensional variations correspond to uniform values. One-dimensional spatial variations correspond to piecewise linear variations, which need not coincide with coordinate axes. Likewise, two-dimensional spatial variations correspond to variations on a planar surface (which need not coincide with the coordinate axes) and three-dimensional spatial variations correspond to variations over a volume. In one, two, or three dimensions, queries can use a "nearest value" search or linear interpolation.

The spatial database files need not provide the data using the same coordinate system as the mesh coordinate system, provided the two coordinate systems are compatible. Examples of compatible coordinate systems include geographic coordinates (longitude/latitude/elevation), and projected coordinates (e.g., coordinates in a transverse Mercator projection). Spatial database queries use the Proj.4 Cartographic Projections library `proj.maptools.org` to convert between coordinate systems, so a large number of geographic projections are available with support for converting between NAD27 and WGS84 horizontal datums as well as several other frequently used datums. Because the interpolation is done in the coordinate system of the spatial database, geographic coordinates should only be used for very simple datasets, or undesirable results will occur. This is especially true when the spatial database coordinate system combines latitude, longitude, and elevation in meters (longitude and latitude in degrees are often much smaller than elevations in meters leading to distorted "distance" between locations and interpolation).

SimpleDB uses a simple ASCII file to specify the variation of values (e.g., displacement field, slip field, physical properties) in space. The file format is described in Section C.2 on page 268. The examples in Chapter 7 on page 111 use SimpleDB files to specify the values for the boundary conditions, physical properties, and fault slip.

As in the other Pyre objects, spatial database objects contain parameters that can be set from the command line or using `cfg` files. The properties and facilities for a spatial database are:

**label** Label for the database, which is used in diagnostic messages.

**query_type** Type of search query to perform. Values for this parameter are "linear" and "nearest" (default).

**iohandler** Database importer. Only one importer is implemented, so you do not need to change this setting.

**iohandler.filename** Filename for the spatial database.

SimpleDB parameters in a `cfg` file

```
label = Material properties
query_type = linear
iohandler.filename = mydb.spatialdb
```

### 4.5.2 UniformDB Spatial Database

The SimpleDB spatial database is quite general, but when the values are uniform, it is often easier to use the UniformDB spatial database instead. With the UniformDB, you specify the values directly either on the command line or in a parameter-setting (`cfg`) file. On the other hand, if the values are used in more than one place, it is easier to place the values in a SimpleDB file, because they can then be referred to using the filename of the spatial database rather than having to repeatedly list all of the values on the command line or in a parameter-setting (`cfg`) file. The properties for a UniformDB are:

**values** Array of names of values in spatial database.

**data** Array of values in spatial database.

UniformDB parameters in a `cfg` file

```
[pylithapp.timedependent.materials.material]
db_properties = spatialdata.spatialdb.UniformDB ; Set the db to a UniformDB
db_properties.values = [vp, vs, density] ; Set the names of the values in the database
db_properties.data = [5773.5*m/s, 3333.3*m/s, 2700.0*kg/m**3] ; Set the values in the database}
```

This example specifies the physical properties of a linearly elastic, isotropic material in a `cfg` file. The data values are dimensioned with the appropriate units using Python syntax.

#### 4.5.2.1 ZeroDispDB

The ZeroDispDB is a special case of the UniformDB for the Dirichlet boundary conditions. The values in the database are the ones requested by the Dirichlet boundary conditions, `displacement-x`, `displacement-y`, and `displacement-z`, and are all set to zero. This makes it trivial to set displacements to zero on a boundary. The examples discussed in Chapter 7 on page 111 use this database.

### 4.5.3 SimpleGridDB Spatial Database

The SimpleGridDB object provides a much more efficient query algorithm than SimpleDB in cases with a orthogonal grid. The points do not need to be uniformly spaced along each coordinate direction. Thus, in contrast to the SimpleDB there is an implicit topology. Nevertheless, the points can be specified in any order, as well as over a lower-dimension than the spatial dimension. For example, one can specify a 2-D grid in 3-D space provided that the 2-D grid is aligned with one of the coordinate axes.

SimpleGridDB uses a simple ASCII file to specify the variation of values (e.g., displacement field, slip field, physical properties) in space. The file format is described in Section C.3 on page 271.

As in the other Pyre objects, spatial database objects contain parameters that can be set from the command line or using `cfg` files. The parameters for a spatial database are:

**label** Label for the database, which is used in diagnostic messages.

**query_type** Type of search query to perform. Values for this parameter are "linear" and "nearest" (default).

**filename** Filename for the spatial database.

SimpleGridDB parameters in a `cfg` file

```
label = Material properties
query_type = linear
filename = mydb_grid.spatialdb
```

### 4.5.4 SCEC CVM-H Spatial Database (SCECCVMH)

Although the SimpleDB implementation is able to specify arbitrarily complex spatial variations, there are existing databases for physical properties, and when they are available, it is desirable to access these directly. One such database is the SCEC CVM-H database, which provides seismic velocities and density information for much of southern California. Spatialdata provides a direct interface to this database. See Section 7.7 on page 129 for an example of using the SCEC CVM-H database for physical properties of an elastic material. The interface is known to work with versions 5.2 and 5.3 of the SCEC CVM-H. Setting a minimum wave speed can be used to replace water and very soft soils that are incompressible or nearly incompressible with stiffer, compressible materials. The Pyre properties for the SCEC CVM-H are:

**data_dir** Directory containing the SCEC CVM-H data files.

**min_vs** Minimum shear wave speed. Corresponding minimum values for the dilatational wave speed (Vp) and density are computed. Default value is 500 m/s.

**squash** Squash topography/bathymetry to sea level (make the earth's surface flat).

**squash_limit** Elevation above which topography is squashed (geometry below this elevation remains undistorted).

SCECCVMH parameters in a `cfg` file

```
[pylithapp.timedependent.materials.material]
db_properties = spatialdata.spatialdb.SCECCVMH ; Set the database to the SCEC CVM-H

# Directory containing the database data files.
db_properties.data_dir = /home/johndoe/data/sceccvm-h/vx53

db_properties.min_vs = 500*m/s ; Default value
db_properties.squash = True ; Turn on squashing

# Only distort the geometry above z=-1km in flattening the earth
db_properties.squash_limit = -1000.0
```

### 4.5.5 CompositeDB Spatial Database

For some problems, a boundary condition or material property may have subsets with different spatial variations. One example would be when we have separate databases to describe the elastic and inelastic bulk material properties for a region. In this case, it would be useful to have two different spatial databases, e.g., a seismic velocity model with Vp, Vs, and density values, and another database with the inelastic physical properties. We can use the CompositeDB spatial database for these cases.

CompositeDB parameters in a `cfg` file

```
[pylithapp.timedependent.materials.maxwell]
label = Maxwell material
id = 1
db_properties = spatialdata.spatialdb.CompositeDB
```

```
db_properties.db_A = spatialdata.spatialdb.SCECCVMH
db_properties.db_B = spatialdata.spatialdb.SimpleDB
quadrature.cell = pylith.feassemble.FIATSimplex
quadrature.cell.dimension = 3

[pylithapp.timedependent.materials.maxwell.db_properties]
values_A = [density, vs, vp]
db_A.label = Elastic properties from CVM-H
db_A.data_dir = /home/john/tools/vx53/bin
db_A.squash = False
values_B = [viscosity]
db_B.label = Vertically varying Maxwell material
db_B.iohandler.filename<p> = ../spatialdb/mat_maxwell.spatialdb
```

Here we have specified a CompositeDB where the elastic properties (density, vs, vp) are given by the SCEC CVM-H, and viscosity is described by a SimpleDB (`mat_maxwell.spatialdb`). The user must first specify **db_properties** as a CompositeDB, and must then give the two components of this database (in this case, SCECCVMH and SimpleDB). The values to query in each of these databases is also required. This is followed by the usual parameters for each of the spatial databases. The CompositeDB provides a flexible mechanism for specifying material properties or boundary conditions where the variations come from two different sources.

### 4.5.6 TimeHistory Database

The TimeHistory database specifies the temporal variation in the amplitude of a field associated with a boundary condition. It is used in conjunction with spatial databases to provide spatial and temporal variation of parameters for boundary conditions. The same time history is applied to all of the locations, but the time history may be shifted with a spatial variation in the onset time and scaled with a spatial variation in the amplitude. The time history database uses a simple ASCII file which is simpler than the one used by the SimpleDB spatial database. The file format is described in Section C.4 on page 272.

As in the other Pyre objects, spatial database objects contain parameters that can be set from the command line or using `cfg` files. The parameters for a spatial database are:

**label** Label for the time history database, which is used in diagnostic messages.

**filename** Filename for the time history database.

TimeHistory parameters in a `cfg` file

```
label = Displacement time history
filename = mytimehistory.timedb
```

## 4.6 Labels and Identifiers for Materials, Boundary Conditions, and Faults

For materials, the "label" is a string used only for error messages. The "id" is an integer that corresponds to the material identifier in LaGriT (itetclr) and CUBIT/Trelis (block id). The id also tags the cells in the mesh for associating cells with a specific material model and quadrature rule. For boundary conditions, the "label" is a string used to associate groups of vertices (psets in LaGriT and nodesets in CUBIT/Trelis) with a boundary condition. Some mesh generators use strings (LaGriT) to identify groups of nodes while others (CUBIT/Trelis) use strings and integers. The default behavior in PyLith is to use strings to identify groups for both LaGriT and CUBIT/Trelis meshes, but the behavior for CUBIT/Trelis meshes can be changed to use the nodeset id (see Section 4.1.2.3 on page 32). PyLith 1.0 had an "id" for boundary conditions, but we removed it from subsequent releases because it was not used. For faults the "label" is used in the same manner as the "label" for boundary conditions. That is, it associates a string with a group of vertices (pset in LaGriT and nodeset in CUBIT/Trelis). The fault "id" is a integer used to tag the cohesive cells in the mesh with a specific fault and quadrature rule. Because we use the fault "id" to tag cohesive cells in the mesh the same way we tag normal cells to materials, it must be unique among the faults as well as the materials.

## 4.7 PyLith Output

PyLith currently supports output to VTK and HDF5/Xdmf files, which can be imported directly into a number of visualization tools, such as ParaView, Visit, and MayaVi. The HDF5 files can also be directly accessed via Matlab and PyTables. PyLith v1.1 significantly expanded the information available for output, including fault information and state variables. Output of solution information for the domain, faults, materials, and boundary conditions is controlled by an output manager for each module. This allows the user to tailor the output to the problem. By default PyLith will write a number of files. Diagnostic information for each fault and material is written into a separate file as are the solution and state variables for the domain, each fault, and each material. For a fault the diagnostic fields include the final slip, the slip initiation time, and the fault normal vector. For a material the diagnostic fields include the density and the elastic constants. Additional diagnostic information can be included by setting the appropriate output parameters. See Chapters 5 on page 61 and 6 on page 85 for more information on the available fields and the next section for output parameters. The other files for each fault and material include solution information at each time step where output was requested (also customizable by the user). For a fault the solution information includes the slip and the change in tractions on the fault surface. For a material the solution information includes the total strain and stress. For some materials fields for additional state variables may be available. For output via VTK files, each time step is written to a separate file, whereas for HDF5 files all of the time steps for a given domain, fault, or material are written into the same file. A single Xdmf metadata file is created for each HDF5 file.

### 4.7.1 Output Manager

The OutputManager object controls the type of files written, the fields included in the output, and how often output is written. PyLith includes some specialized OutputManagers that prescribe what fields are output by default. In some cases, additional fields are available but not included by default. For example, in 3D problems, the along-strike and up-dip directions over the fault surface can be included in the diagnostic information. These are not included by default, because 1D problems have neither an along-strike nor up-dip direction and 2D problems do not have an up-dip direction.

The parameters for the OutputManager are:

**output_freq** Flag indicating whether to write output based on the time or number of time steps since the last output. Permissible values are “time_step” and “skip” (default).

**time_step** Minimum time between output if **output_freq** is set to “time_step”.

**skip** Number of time steps between output if **output_freq** is set to “skip”. A value of 0 means every time step is written.

**writer** Writer for data (VTK writer or HDF5 writer).

**coordsys** Coordinate system for vertex coordinates (currently ignored).

**vertex_filter** Filter to apply to all vertex fields (see Section 4.7.3.1 on the next page).

**cell_filter** Filter to apply to all cell fields (see Section 4.7.3.2 on the following page).

OutputManager parameters in a `cfg` file

```
[pylithapp.timedependent.materials.elastic.output]
output_freq = time_step
time_step = 1.0*yr
cell_filter = pylith.meshio.CellFilterAvg
cell_info_fields = [density] ; limit diagnostic data to density
cell_data_fields = [total-strain, stress] ; default
writer.filename = dislocation-elastic.vtk
```

#### 4.7.1.1 Output Over Subdomain

Output of the solution over the entire domain for large problems generates very large data files. In some cases one is primarily interested in the solution over the ground surface. PyLith supports output of the solution on any boundary of the domain by associating an output manager with a group of vertices corresponding to the surface of the boundary. As with several of the boundary conditions, the boundary must be a simply-connected surface. The OutputSolnSubset is the specialized OutputManager that implements this feature and, by default, includes the displacement field in the output. In addition to the OutputManager parameters, the OutputSolnSubset includes:

**label** Label of group of vertices defining boundary surface.

**vertex_data_fields** Names of vertex data fields to output (default is [displacement]).

### 4.7.2 Output at Arbitrary Points

In many situations with recorded observations, one would like to extract the solution at the same locations as the recorded observation. Rather than forcing the finite-element discretization to be consistent with the observation points, PyLith includes a specialized output manager, OutputSolnPoints, to interpolate the solution to arbitrary points. By default, the output manager will include the displaceent time histories in the output. The locations are specified in a text file. In addition to the OutputManager parameters, the OutputSolnSubset includes:

**vertex_data_fields** Names of vertex data fields to output (default is [displacement]).

**reader** Reader for points list (default is PointsList).

**writer** Writer for output (default is DataWriterVTKPoints). In most cases users will want to use the DataWriterHDF5.

#### 4.7.2.1 PointsList Reader

This object corresponds to a simple text file containing a list of points (one per line) where output is desired. See C.6 on page 273 for file format specifications. The points are specified in the coordinate system specified by OutputSolnPoints. The coordinates will be transformed into the coordinate system of the mesh prior to interpolation. The properties available to customize the behavior of PointsList are:

**filename** Names of file containing list of points.

**comment_delimiter** Delimiter at beginning of line to identify comments (default is #).

**value_delimiter** Delimiter used to separate values (default is whitespace).

### 4.7.3 Output Field Filters

Output fields may not directly correspond to the information a user desires. For example, the default output for the state variables includes the values at each quadrature point. Most visualization packages cannot handle cell fields with multiple points in a cell (the locations of the points within the cell are not included in the data file). In order to reduce the field to a single point within the cell, we would like to average the values. This is best done within PyLith before output, because it reduces the file size and the quadrature information provides the information necessary (the weights of the quadrature points) to compute the appropriate average over the cell.

#### 4.7.3.1 Vertex Field Filters

Currently the only filter available for vertex fields computes the magnitude of a vector at each location. Most visualization packages support this operation, so this filter is not used very often.

**VertexFilterVecNorm** Computes the magnitude of a vector field at each location.

#### 4.7.3.2 Cell Field Filters

Most users will want to apply a filter to cell fields to average the fields over the cell, producing values at one location per cell for visualization.

**CellFilterAvg** Compute the weighted average of the values within a cell. The weights are determined from the quadrature associated with the cells.

### 4.7.4 VTK Output (DataWriterVTK)

PyLith writes legacy (non-XML) VTK files. These are simple files with vertex coordinates, the mesh topology, and fields over vertices and/or cells. Each time step is written to a different file. The time stamp is included in the filename with the decimal point removed. This allows automatic generation of animations with many visualization packages that use VTK files. The default time stamp is the time in seconds, but this can be changed using the normalization constant to give a time stamp in years, tens of years, or any other value.

The parameters for the VTK writer are:

**filename** Name of VTK file.

**time_format** C-style format string for time stamp in filename. The decimal point in the time stamp will be removed for compatibility with VTK visualization packages that provide seamless animation of data from multiple VTK files.

**time_constant** Value used to normalize time stamp in VTK files (default is 1.0 s).

### 4.7.5 HDF5/Xdmf Output (DataWriterHDF5, DataWriterHDF5Ext)

HDF5 files provide a flexible framework for storing simulation data with datasets in groups logically organized in a tree structure analogous to files in directories. HDF5 output offers parallel, multi-dimensional array output in binary files, so it is much faster and more convenient than the VTK output which uses ASCII files and separate files for each time step. Standards for organizing datasets and groups in HDF5 files do not exist for general finite-element software in geodynamics. Consequently, PyLith uses its own simple layout show in Figure 4.5 on the following page. In order for visualization tools, such as ParaView, to determine which datasets to read and where to find them in the hierarchy of groups within the HDF5 file, we create an Xdmf (eXtensible Data Model and Format, `www.xdmf.org`) metadata file that provides this information. This file is written when PyLith closes the HDF5 file at the end of the simulation. In order to visualize the datasets in an HDF5 file, one simply opens the corresponding Xdmf file (the extension is `xmf`) in ParaView or Visit. The Xdmf file contains the relative path to the HDF5 file so the files can be moved but must be located together in the same directory.

**Important**

The Xdmf format supports representation of two- and three-dimensional coordinates of points, scalar fields, and three-dimensional vector and tensor fields but not two-dimensional vector or tensor fields. Consequently, for two-dimensional vector fields we build a three-component vector from the two-component vector (x and y components) and a separate zero scalar field (z component). For tensor fields, we create a scalar field for each of the tensor components, adding the component as a suffix to the name of the field.

See Table 5.2 on page 63 in Section 5.1.3 on page 62 for a table of component values for tensor output in HDF5 files. To avoid confusion about the ordering of components for tensor data, we separate the components in the Xdmf file.

HDF5 files do not contain self-correcting features that allow a file to be read if part of a dataset is corrupted. This type of error can occur if a job terminates abnormally in the middle or at the end of a simulation on a large cluster or other parallel machine. Fortunately, HDF5 also offers the ability to store datasets in external binary files with the locations specified by links in the HDF5 file. Note that the use of external data files results in one data file per dataset in addition to the HDF5 and Xdmf files. The external data files use the name of the HDF5 file with the dataset name added to the prefix and the `h5` suffix replaced by `dat`. The HDF5 files include relative paths to the external data files, so these files can also be moved, but they, too, must

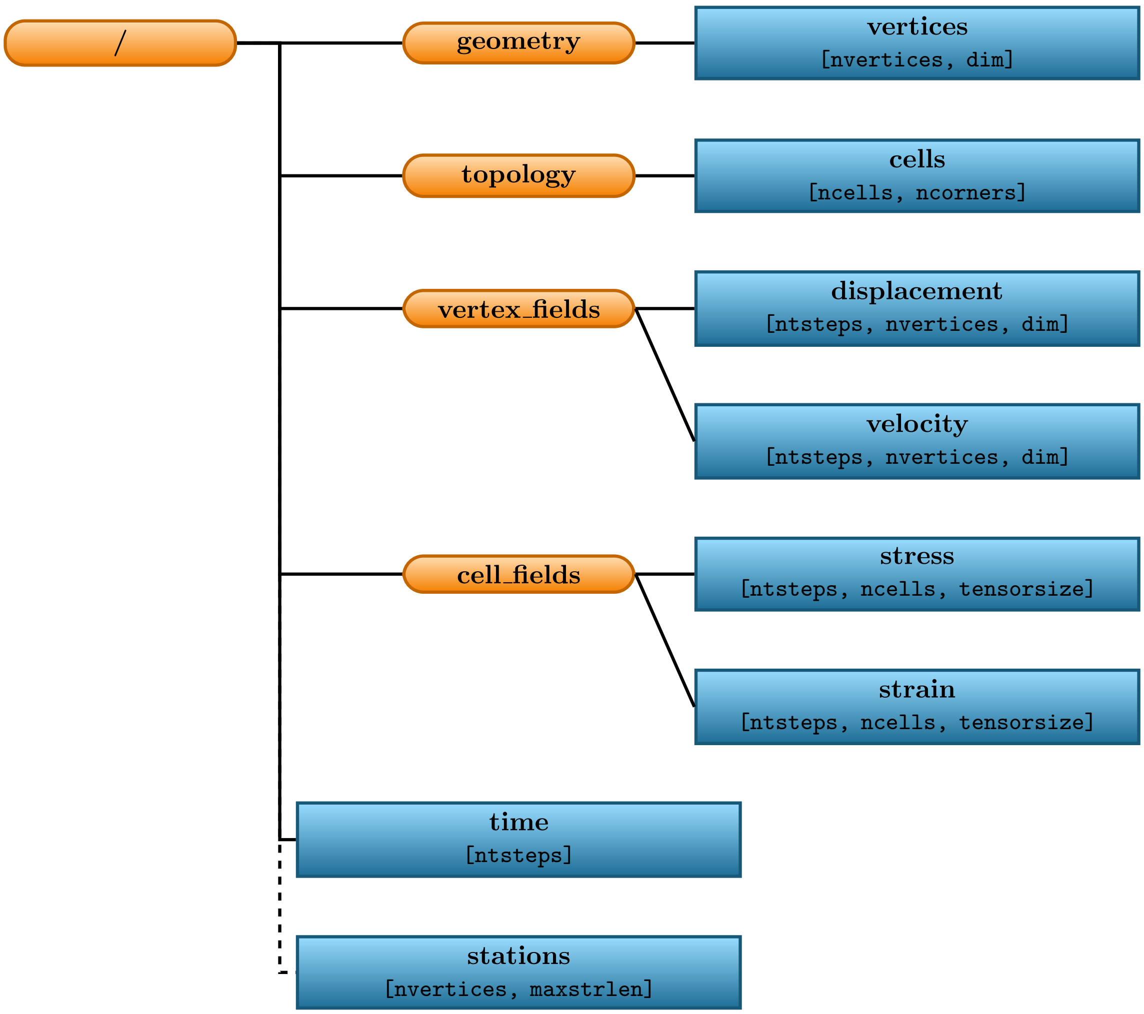



Figure 4.5: General layout of a PyLith HDF5 file. The orange rectangles with rounded corners identify the groups and the blue rectangles with sharp corners identify the datasets. The dimensions of the data sets are shown in parentheses. Most HDF5 files will contain either `vertex_fields` or `cell_fields` but not both.

be kept together in the same directory. This provides a more robust method of output because one can generate an HDF5 file associated with the uncorrupted portions of the external data files should an error occur. Currently, PyLith does not include a utility to do this, but we plan to add one in a future release. Thus, there are two options when writing PyLith output to HDF5 files: (1) including the datasets directly in the HDF5 files themselves using the DataWriterHDF5 object or (2) storing the datasets in external binary files with just metadata in the HDF5 files using the DataWriterHDF5Ext object. Both methods provide similar performance because they will use MPI I/O if it is available.

**Warning**

Storing the datasets within the HDF5 file in a parallel simulation requires that the HDF5 library be configured with the `--enable-parallel` option. The binary PyLith packages include this feature and it is a default setting in building HDF5 via the PyLith Installer.

Accessing the datasets for additional analysis or visualization is nearly identical in the two methods because the use of external data files is completely transparent to the user except for the presence of the additional files. Note that in order for ParaView to find the HDF5 and external data files, it must be run from the same relative location where the simulation was run. For example, if the simulation was run from a directory called "work" and the HDF5/Xdmf files were written to "work/output", then ParaView should be run from the "work" directory. See Table 5.2 on page 63 in Section 5.1.3 on page 62 for a table of component values for tensor output.

#### 4.7.5.1 Parameters

The parametesr for the DataWriterHDF5 and DataWriterHDF5Ext objects is identical:

**filename** Name of HDF5 file (the Xdmf filename is generated from the same prefix).

DataWriterHDF5Ext parameters in a `cfg` file

```
[pylithapp.timedependent.domain.output]
output_freq = time_step
time_step = 1.0*yr
cell_data_fields = [displacement, velocity]
writer = pylith.meshio.DataWriterHDF5Ext
writer.filename = dislocation.h5
```

In this example, we change the writer from the default VTK writer to the HDF5 writer with external datasets (DataWriter-HDF5Ext) for output over the domain.

#### 4.7.5.2 HDF5 Utilities

HDF5 includes several utilities for examining the contents of HDF5 files. `h5dump` is very handy for dumping the hierarchy, dimensions of datasets, attributes, and even the dataset values to stdout.

```
# Dump the entire HDF5 file (not useful for large files).
$ h5dump mydata.h5

# Dump the hierarchy of an HDF5 file.
$ h5dump -n mydata.h5

# Dump the hierarchy with dataset dimensions and attributes.
$ h5dump -H mydata.h5

# Dump dataset 'vertices' in group '/geometry' to stdout.
$ h5dump -d /geometry/vertices mydata.h5
```

We have also include a utility `pylith_genxdmf` (see Section 4.9.2 on page 54) that generates an appropriate Xdmf file from a PyLith HDF5 file. This is very useful if you add fields to HDF5 files in post-processing and wish to view the results in ParaView or Visit.

## 4.8 Tips and Hints

### 4.8.1 Tips and Hints For Running PyLith

- Examine the examples for a problem similar to the one you want to run and dissect it in detail.
- Start with a uniform-resolution coarse mesh to debug the problem setup. Increase the resolution as necessary to resolve the solution fields of interest (resolving stresses/strains may require a higher resolution than that for resolving displacements).
- Merge materials using the same material model. This will result in only one VTK or HDF5 file for each material model rather than several files.
- The rate of convergence in quasi-static (implicit) problems can sometimes be improved by renumbering the vertices in the finite-element mesh to reduce the bandwidth of the sparse matrix. PyLith can use the reverse Cuthill-McKee algorithm to reorder the vertices and cells.
- If you encounter errors or warnings, run `pylithinfo` or use the `--help`, `--help-components`, and `--help-properties` command-line arguments when running PyLith to check the parameters to make sure PyLith is using the parameters you intended.
- Use the `--petsc.log_view`, `--petsc.ksp_monitor`, `--petsc.ksp_view`, `--petsc.ksp_converged_reason`, and `--petsc.snes_converged_reason` command-line arguments (or set them in a parameter file) to view PyLith performance and monitor the convergence.
- Turn on the journals (see the examples) to monitor the progress of the code.

### 4.8.2 Troubleshooting

Consult the PyLith FAQ webpage (https://wiki.geodynamics.org/software:pylith:help:hints) which contains a growing list of common problems and their corresponding solutions.

#### 4.8.2.1 Import Error and Missing Library

```
ImportError: liblapack.so.2: cannot open shared object file: No such file or directory
```

PyLith cannot find one of the libraries. You need to set up your environment variables (e.g., PATH, PYTHONPATH, and LD_LIBRARY_PATH) to match your installation. If you are using the PyLith binary on Linux or Mac OS X, run the command `source setup.sh` in the directory where you unpacked the distribution. This will set up your environment variables for you. If you are building PyLith from source, please consult the instructions for building from source.

#### 4.8.2.2 Unrecognized Property 'p4wd'

```
-- pyre.inventory(error) } \\
-- p4wd <- 'true' } \\
-- unrecognized property 'p4wd' } \\
>> command line:: } \\
-- pyre.inventory(error) } \\
-- p4pg <- 'true' } \\
-- unrecognized property ' p4pg'}
```

Verify that the filenamempirun command included in the PyLith package is the first one on your PATH: `which mpirun`. If it is not, adjust your PATH environment variable accordingly.

#### 4.8.2.3 Detected zero pivor in LU factorization

```
-- Solving equations.
[0] PETSC ERROR: ----------------
Error Message ------------------------------
[0] PETSC ERROR: Detected zero pivot in LU factorization
see http://www.mcs.anl.gov/petsc/petsc-as/documentation/faq.html\#ZeroPivot!
```

This usually occurs when the null space of the system Jacobian is nonzero, such as the case of a problem without Dirichlet boundary conditions on any boundary. If this arises when using the split fields and algebraic multigrid preconditioning, and no additional Dirichlet boundary conditions are desired, then the workaround is to revert to using the Additive Schwarz preconditioning without split fields as discussed in Section 4.1.5 on page 36.

#### 4.8.2.4 Bus Error

This often indicates that PyLith is using incompatible versions of libraries. This can result from changing your environment variables after configuring or installing PyLith (when building from source) or from errors in setting the environment variables (`PATH`, `LD_LIBRARY_PATH`, and `PYTHONPATH`). If the former case, simply reconfigure and rebuild PyLith. In the latter case, check your environment variables (order matters!) to make sure PyLith finds the desired directories before system directories.

#### 4.8.2.5 Segmentation Fault

A segmentation fault usually results from an invalid read/write to memory. It might be caused by an error that wasn't trapped or a bug in the code. Please report these cases so that we can fix these problems (either trap the error and provide the user with an informative error message, or fix the bug). If this occurs with any of the problems distributed with PyLith, simply submit a bug report (see Section 3.6 on page 26) indicating which problem you ran and your platform. If the crash occurs for a problem you created, it is a great help if you can try to reproduce the crash with a very simple problem (e.g., adjust the boundary conditions or other parameters of one of the examples to reproduce the segmentation fault). Submit a bug report along with log files showing the backtrace from a debugger (e.g., gdb) and the valgrind log file (only available on Linux platforms). You can generate a backtrace using the debugger by using the `--petsc.start_in_debugger` command-line argument:

```
$ pylith [..args..] --petsc.start_in_debugger
(gdb) continue
(gdb) backtrace
```

To use valgrind to detect the memory error, first go to your working directory and run the problem with `--launcher.dry`:

```
$ pylith [..args..] --launcher.dry
```

Instead of actually running the problem, this causes PyLith to dump the mpirun/mpiexec command it will execute. Copy and paste this command into your shell so you can run it directly. Insert the full path to valgrind before the full path to mpinemesis and tell valgrind to use a log file:

```
$ mpirun /path/to/valgrind --log-file=valgrind-log /path/to/mpinemesis --pyre-start
  [..lots of junk..]
```

## 4.9 Post-Processing Utilities

The PyLith distribution includes a few post-processing utilities. These are Python scripts that are installed into the same bin directory as the `pylith` executable.

### 4.9.1 pylith_eqinfo

This utility computes the moment magnitude, seismic moment, seismic potency, and average slip at user-specified time snapshots from PyLith fault HDF5 output. The utility works with output from simulations with either prescribed slip and/or spontaneous rupture. Currently, we compute the shear modulus from a user-specified spatial database at the centroid of the fault cells. In the future we plan to account for lateral variations in shear modulus across the fault when calculating the seismic moment. The Python script is a Pyre application, so its parameters can be specified using `cfg` and command line arguments just like PyLith. The Pyre properties and facilities include:

**output_filename** Filename for output of slip information.

**faults** Array of fault names.

**filename_pattern** Filename pattern in C/Python format for creating filename for each fault. Default is `output/fault_%s.h5`.

**snapshots** Array of timestamps for slip snapshosts ([-1] means use last time step in file, which is the default).

**snapshot_units** Units for timestamps in array of snapshots.

**db_properties** Spatial database for elastic properties.

**coordsys** Coordinate system associated with mesh in simulation.

### 4.9.2 pylith_genxdmf

This utility generates Xdmf files from HDF5 files that conform to the layout used by PyLith. It is a simple Python script with a single command line argument with the file pattern of HDF5 files for which Xdmf files should be generated. Typically, it is used to regenerate Xdmf files that get corrupted or lost due to renaming and moving. It is also useful in updating Xdmf files when users add fields to HDF5 files during post-processing.

```
$ pylith_genxdmf --files=FILE_OR_FILE_PATTERN
```

The default value for `FILE_OR_FILE_PATTERN` is `*.h5`.

**Warning**

If the HDF5 files contain external datasets, then this utility should be run from the same relative path to the HDF5 files as when they were created. For example, if a PyLith simulation was run from directory `work` and HDF5 files were generated in `output/work`, then the utility should be run from the directory `work`. Furthermore, a visualization tool, such as ParaView, should also be started from the working directory `work`.

## 4.10 PyLith Parameter Viewer

**New in v2.2.0**

The PyLith Parameter Viewer provides a graphical user interface for viewing the parameters associated with a PyLith simulation and the version information for PyLith and its dependencies. This viewer is an updated and interactive interface to the information generated by the `pylithinfo` script. It displays the hiearchy of components and the parameters for each one, including default values.

## 4.11 Installation

The PyLith Parameter Viewer is included in the PyLith binary distributions and PyLith Docker container for versions 2.1.5 and later. Additionally, the PyLith Installer will install the Parameter Viewer by default. For manual installation you can download the PyLith Parameter Viewer tarball from the PyLith software page (`https://geodynamics.org/cig/software/pylith/`). After downloading the tarball, unpack it. We recommend unpacking the tarball in the top-level PyLith directory.

```
$ tar -xvf pylith_parameters-1.1.0.tgz
```

## 4.12 Running the Parameter Viewer

The steps to run the parameter viewer are:

1. Generate the parameter JSON file.
2. Start the web server (if not already running).
3. Load the parameter JSON file.

### 4.12.1 Generate the parameter JSON file

The parameter viewer uses a JSON file with all of the parameters collected from `cfg` files, command line arguments, etc as input. This file can be generated using `pylithinfo` (see Section 4.1.1.6) and, by default, it will be generated whenever a `pylith` simulation is run. When using `pylithinfo`, the name of the parameter file can be set via a command line argument. When using `pylith`, the DumpParametersJSON component contains a property for the name of the file. You can set the filename on the command line

```
$ pylith --dump_parameters.filename=FILENAME.json
```

or within a .cfg file

```
[pylithapp.dump_parameters]
filename = FILENAME.json
```

Currently, the JSON parameter file cannot be used to run a PyLith simulation. This feature will be added in an upcoming release.

### 4.12.2 Start the web server

Change to the directory containing the `pylith_paramviewer` script (usually the `parametersgui` directory under the top-level `pylith` directory), and run the `pylith_paramviewer` script. This will start a simple Python-based web server on your local computer.

```
$ cd parametersgui
$ ./pylith_paramviewer
```

The script will instruct you to point your web browswer to a local port on your computer. The default is `http://127.0.0.1:9000`. You can change the default port using the `--port` command line argument to the `pylith_paramviewer` script.

## 4.13 Using the Parameter Viewer

When you point your web browser to the correct port, you should see the PyLith Parameter Viewer as shown in Figure 4.6. Click the Choose File button and navigate to the desired JSON parameter file. The viewer tarball includes a sample parameter file `sample_parameters.json`. Click the Reload button to reload the same JSON parameter file if you regenerate it. To select a new JSON parameter file, click the Choose File button and navigate to the desired file.

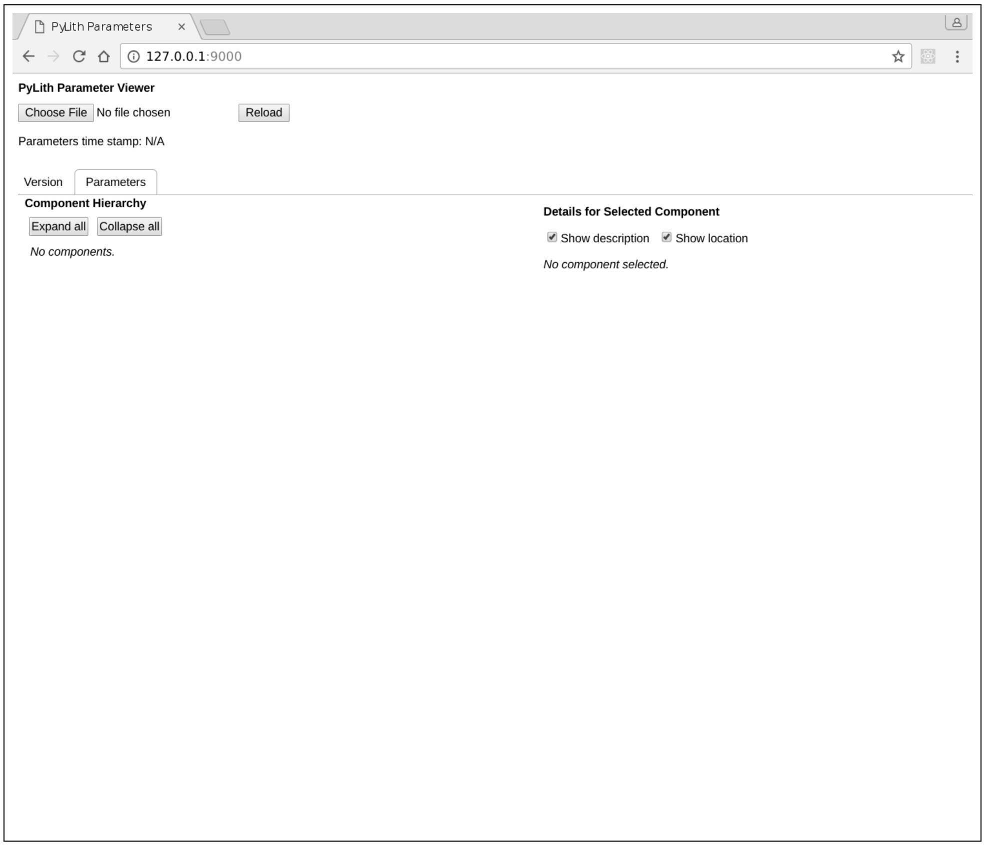



Figure 4.6: Screenshot of PyLith Parameter Viewer in web browser upon startup.

### 4.13.1 Version Information

Click on the Version tab to examine the version information. This tab displays the same version information shown with the `--version` command line argument to `pylith` in an easy to read layout. This includes information about the platform on which `pylith` or `pylithinfo` was run, the PyLith version, and versions of the dependencies, as shown in Figure 4.7.

### 4.13.2 Parameter Information

Click on the Parameters tab to examine the hiearchy of components and the parameters for each. You can expand/collapse the Component Hierarchy tree in the left panel by clicking on the triangles or facility name in blue to the left of the equals sign (Figure 4.8). Clicking on the component in red to the right of the equals sign will show its parameters in the right panel (Figure 4.8). The selected facility in the left panel whose parameters are shown in the right panel will be highlighted via a gray background (Figure 4.9).

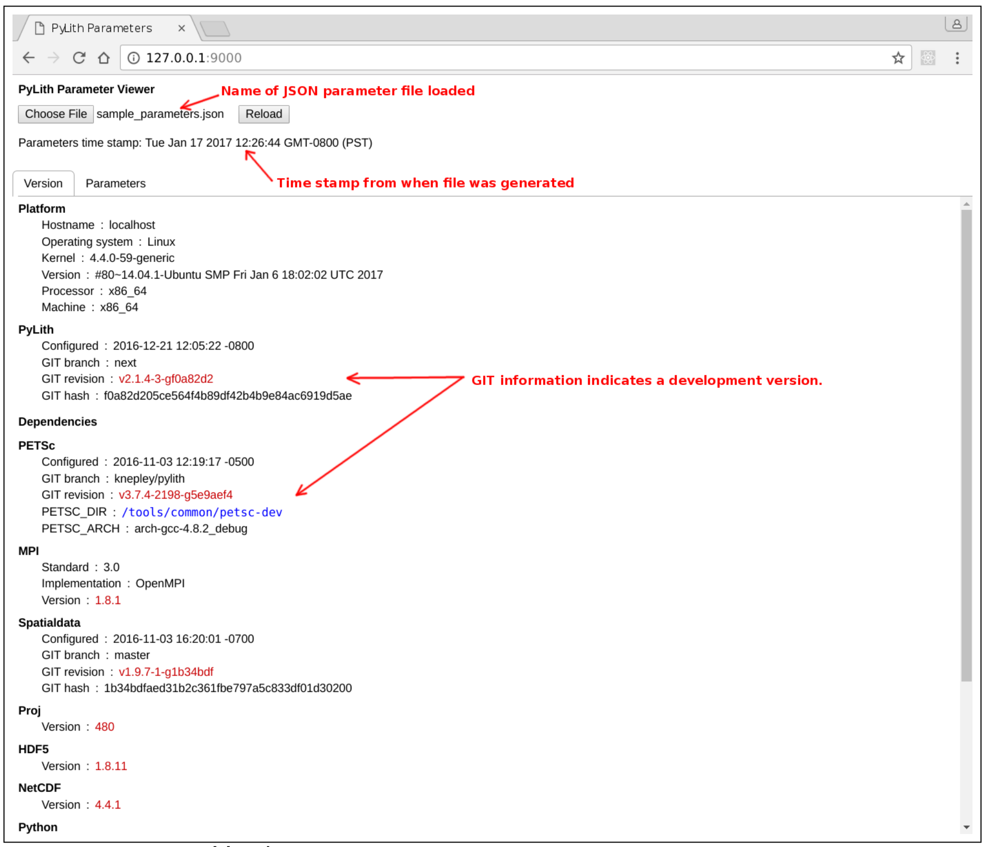


Figure 4.7: Screenshot of Version tab of the PyLith Parameter Viewer with sample JSON parameter file.

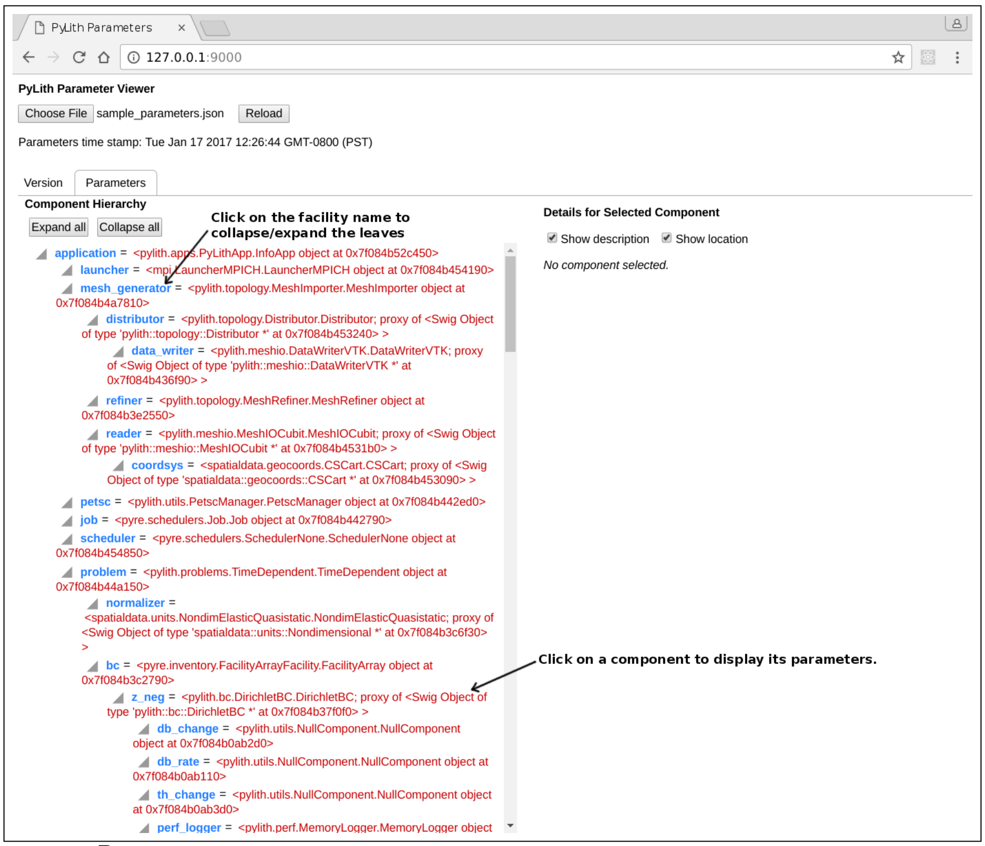



Figure 4.8: Screenshot of Parameters tab of the PyLith Parameter Viewer with sample JSON parameter file before selecting a component in the left panel.

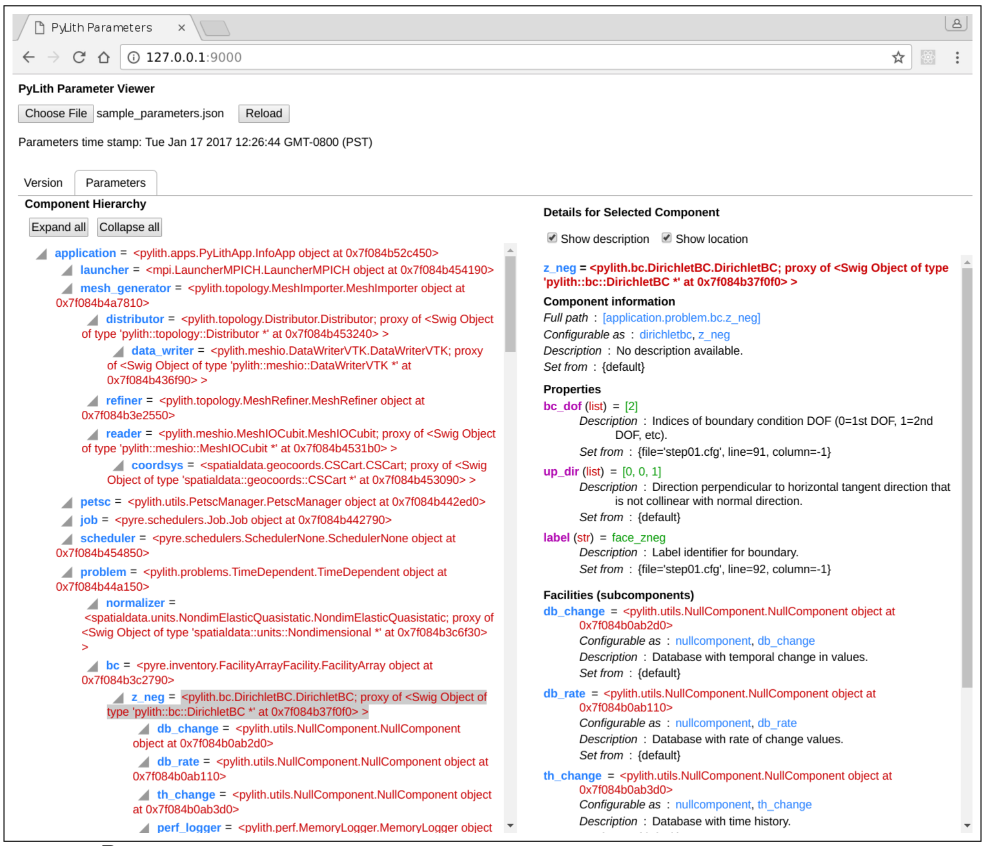


Figure 4.9: Screenshot of Parameters tab of the PyLith Parameter Viewer with sample JSON parameter file with the **z_neg** facility selected.

# Chapter 5

# Material Models

## 5.1 Specifying Material Properties

Associating material properties with a given cell involves several steps.

1. In the mesh generation process, assign a material identifier to each cell.
2. Define material property groups corresponding to each material identifier.
3. Set the parameters for each material group using `cfg` and/or command-line arguments.
4. Specify the spatial variation in material property parameters using a spatial database file.

### 5.1.1 Setting the Material Identifier

Each cell in the finite-element mesh must have a material identifier. This integer value is associated with a bulk material model. The parameters of the material model need not be uniform for cells with the same material identifier. The bulk constitutive model and numerical integration (quadrature) scheme will, however, be the same for all cells with the same material identifier value. The material identifier is set during the mesh generation process. The procedure for assigning this integer value to a cell depends on the mesh generator. For example, in the PyLith mesh ASCII format, the identifiers are listed in the cells group using the material-id data; in CUBIT materials are defined using blocks; in LaGriT materials are defined by the attribute `imt1` and the mregion command.

### 5.1.2 Material Property Groups

The material property group associates a material model (label for the material, a bulk constitutive model, and parameters for the constitutive model) with a material identifier. In previous versions of PyLith it was necessary to specify containers that defined the number of groups and associated information for each group. This was necessary because previous versions of Pyre did not support dynamic arrays of components, and it was necessary to predefine these arrays. More recent versions of Pythia do support this, however, and it is now possible to define material property groups using a `cfg` file or on the command-line. User-defined containers are no longer necessary, and the predefined containers are no longer available (or necessary). If a set of material groups is not specified, a single material model is used for the entire problem. See Sections 7.9 on page 139 and 7.8 on page 131 for examples that demonstrate how to specify more than one material model.

### 5.1.3 Material Parameters

For each material group, there is a single component defining the material model to be used. The default material model is ElasticIsotropic3D. For each material model, the available properties and facilities are:

**id** This is the material identifier that matches the integer value assigned to each cell in the mesh generation process.

**label** Name or label for the material. This is used in error and diagnostic reports.

**db_properties** Spatial database specifying the spatial variation in the parameters of the bulk constitutive model (default is a SimpleDB).

**db_initial_stress** Spatial database specifying the spatial variation in the initial stress (default is none).

**db_initial_strain** Spatial database specifying the spatial variation in the initial strain (default is none).

**db_initial_state** Spatial database specifying the spatial variation in the other initial state variables (default is none).

**output** The output manager used for outputting material information.

**quadrature** Numerical integration scheme used in integrating fields over each cell.

Parameters for two materials in a `cfg` file

```
[pylithapp.timedependent]
materials = [elastic, viscoelastic]

[pylithapp.timedependent.materials.elastic]
label = Elastic material
id = 1
db_properties.iohandler.filename = mat\_elastic.spatialdb
quadrature.cell = pylith.feassemble.FIATLagrange
quadrature.cell.dimension = 3

[pylithapp.timedependent.materials.viscoelastic]
label = Viscoelastic material
id = 2
db_properties.iohandler.filename = mat_viscoelastic.spatialdb
quadrature.cell = pylith.feassemble.FIATLagrange
quadrature.cell.dimension = 3
```

These settings correspond to the the problem in Section 7.9 on page 139. The parameters for the bulk constitutive models are specified using the spatial databases `mat_elastic.spatialdb` and `mat_viscoelastic.spatialdb`. Refer to the discussion of each material model to find the parameters that must be specified in the spatial database. Appendix C.2 on page 268 describes the format of the SimpleDB spatial database files. In a more realistic problem, a different spatial database, and possibly a different material model, would be used for each material group.

In general, we average the output over the quadrature points within a cell and specify the name of the output files for each material group:

```
[pylithapp.timedependent.materials.elastic.output]
cell_filter = pylith.meshio.CellFilterAvg
writer.filename = dislocation-elastic.vtk

[pylithapp.timedependent.materials.viscoelastic.output]
cell_filter = pylith.meshio.CellFilterAvg
writer.filename = dislocation-viscoelastic.vtk
```

These settings again correspond to the problem in Section 7.9 on page 139. The specification of a state variable base filename (`writer.filename` settings) will cause two files to be created for each material group: an info file, which describes the material property parameters used in the model, and a state variables file, which contains the state variable information. Note that the material property parameters described by the info file are the parameters used internally by PyLith. In some cases they are parameters convenient for use in the constitutive models and are derived from the parameters specified by the user via the spatial database. If the problem has more than one time step, a state variable output file will be created for each requested time step. We have requested that the values be averaged over each cell. Otherwise, output would be produced for each quadrature point, which can cause problems with some visualization packages. For this example problem, the material is three-dimensional isotropic elastic, and is thus described by three parameters ($\lambda$, $\mu$, $\rho$), as described below. These properties are output by default. Other material models require additional parameters, and if users want these to be output, they must be specified. Similarly, other material models require state variables in addition to the default stress and strain variables that are used by all material models. Additional output may be requested for a material model, as in this example (see Section 7.6 on page 125):

```
[pylithapp.timedependent.materials.material.output]
cell_data_fields = [total_strain, viscous_strain, stress]
cell_info_fields = [mu, lambda, density, maxwell_time]
```

The properties and state variables available for output in each material model are listed in Table 5.1. The order of the state variables in the output arrays is given in Table 5.2. For the generalized Maxwell model, values of `shear_ratio` and `maxwell_time` are given for each Maxwell element in the model (there are presently three, as described below). Similarly, there are three sets of `viscous_strain` values for the generalized Maxwell model.

Table 5.1: Properties and state variables available for output for existing material models. Physical properties are available for output as **`cell_info_fields`** and state variables are available for output as **`cell_data_fields`**.

| Model | Physical Properties | State Variables | Requires nonlinear solver? |
|---|---|---|---|
| Elastic | `mu, lambda, density` | `total_strain, stress, cauchy_stress` | No |
| Maxwell Viscoelastic | `mu, lambda, density, maxwell_time` | `total_strain, stress, cauchy_stress, viscous_strain` | No |
| Generalized Maxwell Viscoelastic | `mu, lambda, density, shear_ratio, maxwell_time` | `total_strain, stress, cauchy_stress, viscous_strain_1, viscous_strain_2, viscous_strain_3` | No |
| Power-law Viscoelastic | `mu, lambda, density, reference_strain_rate, reference_stress, power_law_exponent` | `total_strain, stress, cauchy_stress, viscous_strain` | Yes |
| Drucker-Prager Elastoplastic | `mu, lambda, density, alpha_yield, beta, alpha_flow` | `total_strain, stress, cauchy_stress, plastic_strain` | Yes |

Table 5.2: Order of components in tensor state-variables for material models.

| State Variable | 2D | 3D |
|---|---|---|
| `total_strain` | $\epsilon_{xx}$, $\epsilon_{yy}$, $\epsilon_{xy}$ | $\epsilon_{xx}$, $\epsilon_{yy}$, $\epsilon_{zz}$, $\epsilon_{xy}$, $\epsilon_{yz}$, $\epsilon_{xz}$ |
| `stress, cauchy_stress` | $\sigma_{xx}$, $\sigma_{yy}$, $\sigma_{xy}$ | $\sigma_{xx}$, $\sigma_{yy}$, $\sigma_{zz}$, $\sigma_{xy}$, $\sigma_{yz}$, $\sigma_{xz}$ |
| `viscous_strain, plastic_strain` | $\epsilon_{xx}$, $\epsilon_{yy}$, $\epsilon_{zz}$, $\epsilon_{xy}$ | $\epsilon_{xx}$, $\epsilon_{yy}$, $\epsilon_{zz}$, $\epsilon_{xy}$, $\epsilon_{yz}$, $\epsilon_{xz}$ |
| `stress4` | $\sigma_{xx}$, $\sigma_{yy}$, $\sigma_{zz}$, $\sigma_{xy}$ | |

### 5.1.4 Initial State Variables

In many problems of interest, the state variables describing a material model may already have nonzero values prior to the application of any boundary conditions. For problems in geophysics, the most common example is a problem that includes the effects of gravitational body forces. In the real earth, rocks were emplaced and formed under the influence of gravity. When performing numerical simulations, however, it is not possible to represent the entire time history of rock emplacement. Instead, gravity must be "turned on" at the beginning of the simulation. Unfortunately, this results in unrealistic amounts of deformation at the beginning of a simulation. An alternative is to provide initial state variables for the region under consideration. This allows the specification of a set of state variables that is consistent with the prior application of gravitational body forces. In a more general sense, initial values for state variables may be used to provide values that are consistent with any set of conditions that occurred prior to the beginning of a simulation. The current release of PyLith allows the specification of initial stresses, strains, and state variables for all materials; however, not all of the initial state variables are presently used. For example, `cauchy_stress` is available as a state variable for all materials, but specifying an initial value would not make sense for most problems.

#### 5.1.4.1 Specification of Initial State Variables

State variables are specific to a given material, so initial values for state variables are specified as part of the material description. The default is that no initial state variables are specified. In computing the elastic prestep, appropriate values for the state variables are set; otherwise the state variables are set to zero. To override this behavior, specify a spatial database for the initial stress, strain, and/or state variables as in the example from the example in Section 7.9 on page 139:

Excerpt from `examples/3d/hex8/step16.cfg`

```
[pylithapp.timedependent.materials.elastic]
db_initial_stress = spatialdata.spatialdb.SimpleDB
db_initial_stress.iohandler.filename<p> = initial\_stress.spatialdb
```

> **Warning**
>
> Using the elastic prestep with initial state variables will generally lead to the state variables being ignored (the initial out of plane stress is the exception), because the elastic prestep will set the state variables based on the elastic solution.

> **Warning**
>
> Currently, PyLith assumes initial displacements and velocities of zero, so any initial strain and state variables should be consistent with these initial conditions. This limitation will be removed in future releases.

As mentioned in section D.1 on page 275, plane strain problems do not include the out-of-plane stress component ($\sigma_{zz}$), and an additional state variable (`stress-zz-initial`) is provided for all two-dimensional viscoelastic and elastoplastic models. To completely specify the initial stresses, the user must provide two spatial databases: an initial stress database that includes the three 2D stress components ($\sigma_{xx}$, $\sigma_{yy}$, and $\sigma_{xy}$) and an additional database containing the out of plane stress and initial values for all other state variables for the given material. The complete initial stress field may then be defined in the `cfg` file as:

```
[pylithapp.problem.materials.powerlaw]
# First specify initial 2D stresses
db_initial_stress = spatialdata.spatialdb.SimpleDB
db_initial_stress.label = 2D initial stress
db_initial_stress.iohandler.filename = inititial_stress_2d.spatialdb

# Now specify out-of-plane initial stresses (and all other state variables)
db_initial_state = spatialdata.spatialdb.SimpleDB
```

```
db_initial_state.label = Out of plane strain initial stress
db_initial_state.iohandler.filename = initial_state_2d.spatialdb
```

Table 5.3: Values in spatial database for initial state variables for 3D problems. 2D problems use only the relevant values. Note that initial stress and strain are available for all material models. Some models have additional state variables (Table 5.1 on page 63) and initial values for these may also be provided.

| State Variable | Values in Spatial Database |
|---|---|
| initial stress | stress-xx, stress-yy, stress-zz, stress-xy, stress-yz, stress-xz |
| initial strain | total-strain-xx, total-strain-yy, total-strain-zz, total-strain-xy, total-strain-yz, total-strain-xz |

### 5.1.5 Cauchy Stress Tensor and Second Piola-Kirchoff Stress Tensor

In outputting the stress tensor (see Tables 5.1 on page 63 and 5.2 on page 63), the tensor used internally in the formulation of the governing equation is the stress field available for output. For the infinitesimal strain formulation this is the Cauchy stress tensor; for the finite strain formulation, this is the second Piola-Kirchoff stress tensor. The user may also explicitly request output of the Cauchy stress tensor (cauchy_stress field). Obviously, this is identical to the stress field when using the infinitesimal strain formulation. See section 2.5 on page 14 for a discussion of the relationship between the Cauchy stress tensor and the second Piola-Kirchhoff stress tensor.

**Important**

Although the second Piola-Kirchoff stress tensor has little physical meaning, the second Piola-Kirchoff stress tensor (not the Cauchy stress tensor) values should be specified in the initial stress database when using the finite strain formulation.

### 5.1.6 Stable time step

PyLith computes the stable time step in both quasi-static and dynamic simulations. In quasi-static simulations the stability of the implicit time stepping scheme does not depend on the time step; instead, the stable time step is associated with the accuracy of the solution. For viscoelastic materials the stable time step uses 1/5 of the minimum viscoelastic relaxation time. In purely elastic materials, the accuracy is independent of the time step, so the stable time step is infinite. The same is true for elastoplastic materials, since there is no inherent time scale for these problems. Depending on the loading rate, however, it is possible to impose a load increment that is large enough so that the resulting solution may be inaccurate or divergent. In quasi-static simulations we recompute the stable time step at every time step.

**Warning**

Caution must be used in assigning time step sizes for elastoplastic problems, and the linear and nonlinear convergence should be monitored closely.

In dynamic simulations the stability of the explicit time-stepping scheme integration does depend on the time step via the Courant-Friderichs-Lewy condition [Courant et al., 1967]. This condition states that the critical time step is the time it takes for the P wave to travel across the shortest dimension of a cell. In most cases this is the shortest edge length. However, distorted cells which have relatively small areas in 2-D or relatively small volumes in 3-D for the given edge lengths also require small time steps due to the artificially high stiffness associated with the distorted shape. As a result, we set the stable time step to be the smaller of the shortest edge length and a scaling factor times the radius of an inscribed circle (in 2-D),

$$dt = \min(e_{min}, 3.0 r_{inscribed}) \tag{5.1}$$

$$r_{inscribed} = \sqrt{\frac{k(k-e_0)(k-e_1)(k-e_2)}{k}} \tag{5.2}$$

$$k = \frac{1}{2}(e_0 + e_1 + e_2) \tag{5.3}$$

and sphere (in 3-D),

$$dt = \min(e_{min}, 6.38 r_{inscribed}) \tag{5.4}$$

$$r_{inscribed} = 3V/(A_0 + A_1 + A_2 + A_3), \tag{5.5}$$

where $e_i$ denotes the length of edge $i$, $A_i$ denotes the area of face $i$, and $V$ is the volume of the cell. We determined the scaling factoring empirically using several benchmarks. In dynamic simulations we check the stable time step only at the beginning of the simulation. That is, we assume the elastic properties and mesh do not change, so that the stable time step is constant throughout the simulation.

The stable time step is used in all three of the time stepping schemes used by PyLith (see section 4.2.4 on page 40). In general, an error is generated if the user attempts to use a time step size larger than the stable time step. The stable time steps for each cell can be included in the output with the other **`cell_info_fields`**. For implicit time stepping the field is `stable_dt_implicit` and for explicit time stepping the field is `stable_dt_explicit`.

## 5.2 Elastic Material Models

The generalized form of Hooke's law relating stress and strain for linear elastic materials is

$$\sigma_{ij} = C_{ijkl}\left(\epsilon_{kl} - \epsilon^I_{kl}\right) + \sigma^I_{ij}, \tag{5.6}$$

where we have included both initial strains and initial stresses, denoted with the superscript *I*. Due to symmetry considerations, however, the 81 components of the elasticity matrix are reduced to 21 independent components for the most general case of anisotropic elasticity. Representing the stress and strain in terms of vectors, the constitutive relation may be written

$$\vec{\sigma} = \underline{C}\left(\vec{\epsilon} - \vec{\epsilon}^I\right) + \vec{\sigma}^I, \tag{5.7}$$

where

$$\underline{C} = \begin{bmatrix} C_{1111} & C_{1122} & C_{1133} & C_{1112} & C_{1123} & C_{1113} \\ C_{1122} & C_{2222} & C_{2233} & C_{2212} & C_{2223} & C_{2213} \\ C_{1133} & C_{2233} & C_{3333} & C_{3312} & C_{3323} & C_{3313} \\ C_{1112} & C_{2212} & C_{3312} & C_{1212} & C_{1223} & C_{1213} \\ C_{1123} & C_{2223} & C_{3323} & C_{1223} & C_{2323} & C_{2313} \\ C_{1113} & C_{2213} & C_{3313} & C_{1213} & C_{2313} & C_{1313} \end{bmatrix}. \tag{5.8}$$

For the case of isotropic elasticity, the number of independent components reduces to two, and the model can be characterized by two parameters, Lame's constants $\mu$ and $\lambda$. Lame's constants are related to the density ($\rho$), shear wave speed ($v_s$), and compressional wave speed ($v_p$) via

$$\begin{aligned} \mu &= \rho v_s^2 \\ \lambda &= \rho v_p^2 - 2\mu \end{aligned} \tag{5.9}$$

Table 5.4: Values in spatial databases for the elastic material constitutive models.

| Spatial database | Value | Description |
|---|---|---|
| **db_properties** | vp | Compressional wave speed, $v_p$ |
| | vs | Shear wave speed, $v_s$ |
| | density | Density, $\rho$ |
| **db_initial_stress** | stress-xx, ... | Initial stress components |
| **db_initial_strain** | total-strain-xx, ... | Initial strain components |

### 5.2.1 2D Elastic Material Models

In 2D we can write Hooke's law as

$$\begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{12} \end{bmatrix} = \begin{bmatrix} C_{1111} & C_{1122} & C_{1112} \\ C_{1122} & C_{2222} & C_{2212} \\ C_{1112} & C_{2212} & C_{1212} \end{bmatrix} \begin{bmatrix} \epsilon_{11} - \epsilon_{11}^I \\ \epsilon_{22} - \epsilon_{22}^I \\ \epsilon_{12} - \epsilon_{12}^I \end{bmatrix} + \begin{bmatrix} \sigma_{11}^I \\ \sigma_{22}^I \\ \sigma_{12}^I \end{bmatrix}. \tag{5.10}$$

#### 5.2.1.1 Elastic Plane Strain

If the gradient in deformation with respect to the $x_3$ axis is zero, then $\epsilon_{33} = \epsilon_{13} = \epsilon_{23} = 0$ and plane strain conditions apply, so we have

$$\begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{12} \end{bmatrix} = \begin{bmatrix} \lambda + 2\mu & \lambda & 0 \\ \lambda & \lambda + 2\mu & 0 \\ 0 & 0 & 2\mu \end{bmatrix} \begin{bmatrix} \epsilon_{11} - \epsilon_{11}^I \\ \epsilon_{22} - \epsilon_{22}^I \\ \epsilon_{12} - \epsilon_{12}^I \end{bmatrix} + \begin{bmatrix} \sigma_{11}^I \\ \sigma_{22}^I \\ \sigma_{12}^I \end{bmatrix}. \tag{5.11}$$

#### 5.2.1.2 Elastic Plane Stress

If the $x_1 x_2$ plane is traction free, then $\sigma_{33} = \sigma_{13} = \sigma_{23} = 0$ and plane stress conditions apply, so we have

$$\begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{12} \end{bmatrix} = \begin{bmatrix} \frac{4\mu(\lambda+\mu)}{\lambda+2\mu} & \frac{2\mu\lambda}{\lambda+2\mu} & 0 \\ \frac{2\mu\lambda}{\lambda+2} & \frac{4\mu(\lambda+\mu)}{\lambda+2\mu} & 0 \\ 0 & 0 & 2\mu \end{bmatrix} \begin{bmatrix} \epsilon_{11} - \epsilon_{11}^I \\ \epsilon_{22} - \epsilon_{22}^I \\ \epsilon_{12} - \epsilon_{12}^I \end{bmatrix} + \begin{bmatrix} \sigma_{11}^I \\ \sigma_{22}^I \\ \sigma_{12}^I \end{bmatrix}, \tag{5.12}$$

where

$$\begin{gathered} \epsilon_{33} = -\frac{\lambda}{\lambda + 2\mu}(\epsilon_{11} + \epsilon_{22}) + \epsilon_{33}^I \\ \epsilon_{13} = \epsilon_{23} = 0. \end{gathered} \tag{5.13}$$

### 5.2.2 3D Elastic Material Models

#### 5.2.2.1 Isotropic

For this case the stress-strain matrix, $\underline{C}$, becomes

$$\underline{C} = \begin{bmatrix} \lambda + 2\mu & \lambda & \lambda & 0 & 0 & 0 \\ \lambda & \lambda + 2\mu & \lambda & 0 & 0 & 0 \\ \lambda & \lambda & \lambda + 2\mu & 0 & 0 & 0 \\ 0 & 0 & 0 & 2\mu & 0 & 0 \\ 0 & 0 & 0 & 0 & 2\mu & 0 \\ 0 & 0 & 0 & 0 & 0 & 2\mu \end{bmatrix}. \tag{5.14}$$

## 5.3 Viscoelastic Materials

At present, there are six viscoelastic material models available in PyLith (Table 5.5 and Figure 5.1 on the next page). Future code versions may include alternative formulations for the various material models (Appendix D on page 275), so that users may use the most efficient formulation for a particular problem. Note that both 2D and 3D viscoelastic models are described, but we present below only the 3D formulations. The 2D formulations are easily obtained from the plane strain definition. The one aspect of the 2D formulations that is different is the specification of initial stresses. Since 2D models only have three tensor components, it is not possible to specify the normal stress in the out-of-plane direction ($\sigma_{33}$), which is generally nonzero, using the same method as the other tensor components. To allow for the specification of this initial stress component, an additional state variable corresponding to $\sigma_{33}^{I}$ is provided (`stress_zz_initial`). This state variable is provided for all of the viscoelastic material models as well as the plane strain Drucker-Prager elastoplastic model. See section 5.1.4 on page 63 for additional information on specifying initial stresses for plane strain problems. For the PowerLawPlaneStrain model, all four of the stress components are needed, so a 4-component stress state variable (`stress4`) is provided in addition to the normal 3-component `stress` state variable (see Table 5.1 on page 63).

Table 5.5: Available viscoelastic materials for PyLith.

| Model Name | Description |
|---|---|
| MaxwellPlaneStrain | Plane strain Maxwell material with linear viscous rheology |
| GenMaxwellPlaneStrain | Plane strain generalized Maxwell material (3 Maxwell models in parallel) |
| PowerLawPlaneStrain | Plane strain Maxwell material with power-law viscous rheology |
| MaxwellIsotropic3D | Isotropic Maxwell material with linear viscous rheology |
| GenMaxwellIsotropic3D | Generalized model consisting of 3 Maxwell models in parallel |
| PowerLaw3D | Isotropic Maxwell material with power-law viscous rheology |

### 5.3.1 Definitions

In the following sections, we use a combination of vector and index notation (our notation conventions are shown in Table 2.1 on page 7). When using index notation, we use the common convention where repeated indices indicate summation over the range of the index. We also make frequent use of the scalar inner product. The scalar inner product of two second-order tensors may be written

$$\underline{a} \cdot \underline{b} = a_{ij} b_{ij} \,. \tag{5.15}$$

Although the general constitutive relations are formulated in terms of the stress and strain, we frequently make use of the deviatoric stress and strain in our formulation. We first define the mean stress, $P$, and mean strain, $\theta$:

$$P = \frac{\sigma_{ii}}{3}, \;\; \theta = \frac{\epsilon_{ii}}{3}, \tag{5.16}$$

where the $\sigma_{ii}$ and $\epsilon_{ii}$ represent the trace of the stress and strain tensors, respectively. We then define the deviatoric components of stress and strain as

$$S_{ij} = \sigma_{ij} - P\delta_{ij}, \;\; e_{ij} = \epsilon_{ij} - \theta\delta_{ij}, \tag{5.17}$$

where $\delta_{ij}$ is the Kronecker delta. Using the deviatoric components, we define the effective stress, $\overline{\sigma}$, the second deviatoric stress invariant, $J_2'$, the effective deviatoric strain, $\overline{e}$, and the second deviatoric strain invariant, $L_2'$, as

$$\begin{aligned} \overline{\sigma} &= \sqrt{\frac{3}{2}\underline{S}\cdot\underline{S}} \\ J_2' &= \frac{1}{2}\underline{S}\cdot\underline{S}. \\ \overline{e} &= \sqrt{\frac{2}{3}\underline{e}\cdot\underline{e}} \\ L_2' &= \frac{1}{2}\underline{e}\cdot\underline{e} \end{aligned} \tag{5.18}$$

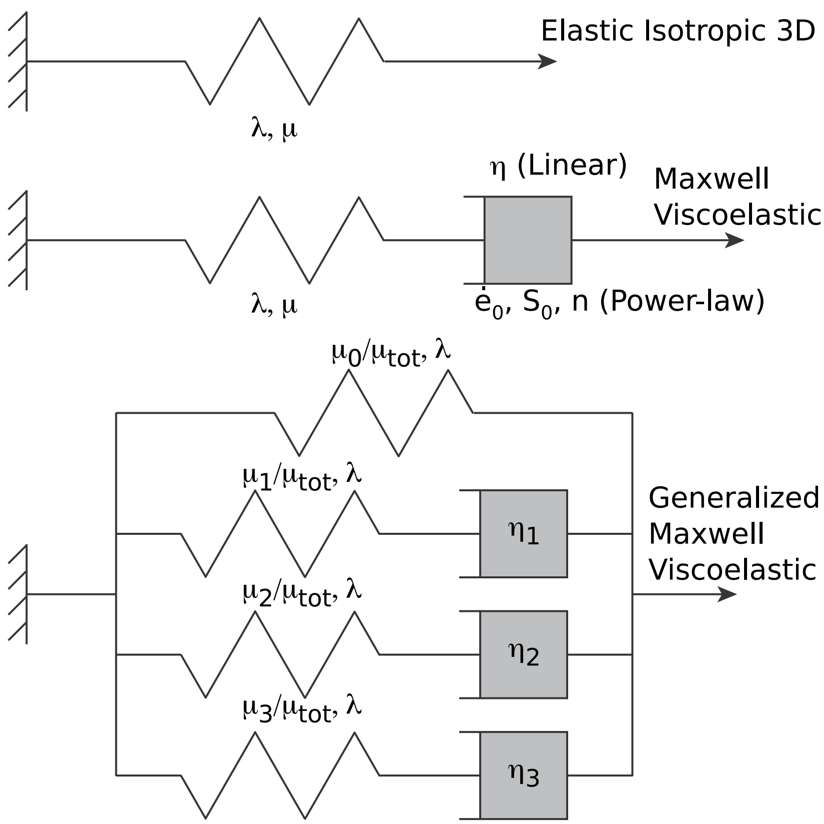



Figure 5.1: Spring-dashpot 1D representations of the available 3D elastic and 2D/3D viscoelastic material models for PyLith. The top model is a linear elastic model, the middle model is a Maxwell model, and the bottom model is a generalized Maxwell model. For the generalized Maxwell model, $\lambda$ and $\mu_{tot}$ are specified for the entire model, and then the ratio $\mu_i/\mu_{tot}$ is specified for each Maxwell model. For the power-law model, the linear dashpot in the Maxwell model is replaced by a nonlinear dashpot obeying a power-law.

Due to the symmetry of the stress and strain tensors, it is sometimes convenient to represent them as vectors:

$$\overrightarrow{\sigma}^T = \begin{bmatrix} \sigma_{11} & \sigma_{22} & \sigma_{33} & \sigma_{12} & \sigma_{23} & \sigma_{31} \end{bmatrix} \tag{5.19}$$
$$\overrightarrow{\epsilon}^T = \begin{bmatrix} \epsilon_{11} & \epsilon_{22} & \epsilon_{33} & \epsilon_{12} & \epsilon_{23} & \epsilon_{31} \end{bmatrix}.$$

Note that when taking the scalar inner product of two tensors represented as vectors, it is necessary to double the products representing off-diagonal terms.

For quantities evaluated over a specific time period, we represent the initial time as a prefixed subscript and the end time as a prefixed superscript. In cases where the initial time does not appear, it is understood to be $-\infty$.

### 5.3.2 Linear Viscoelastic Models

Linear viscoelastic models are obtained by various combinations of a linear elastic spring and a linear viscous dashpot in series or parallel. The simplest example is probably the linear Maxwell model, which consists of a spring in series with a dashpot, as shown in Figure 5.1 on the preceding page. For a one-dimensional model, the response is given by

$$\frac{d\epsilon_{Total}}{dt} = \frac{d\epsilon_D}{dt} + \frac{d\epsilon_S}{dt} = \frac{\sigma}{\eta} + \frac{1}{E}\frac{d\sigma}{dt}, \tag{5.20}$$

where $\epsilon_{Total}$ is the total strain, $\epsilon_D$ is the strain in the dashpot, $\epsilon_S$ is the strain in the spring, $\sigma$ is the stress, $\eta$ is the viscosity of the dashpot, and $E$ is the spring constant. When a Maxwell material is subjected to constant strain, the stresses relax exponentially with time. When a Maxwell material is subjected to a constant stress, there is an immediate elastic strain, corresponding to the response of the spring, and a viscous strain that increases linearly with time. Since the strain response is unbounded, the Maxwell model actually represents a fluid.

Another simple model is the Kelvin-Voigt model, which consists of a spring in parallel with a dashpot. In this case, the one-dimensional response is given by

$$\sigma(t) = E\epsilon(t) + \eta\frac{d\epsilon(t)}{dt}. \tag{5.21}$$

As opposed to the Maxwell model, which represents a fluid, the Kelvin-Voigt model represents a solid undergoing reversible, viscoelastic strain. If the material is subjected to a constant stress, it deforms at a decreasing rate, gradually approaching the strain that would occur for a purely elastic material. When the stress is released, the material gradually relaxes back to its undeformed state.

The most general form of linear viscoelastic model is the generalized Maxwell model, which consists of a spring in parallel with a number of Maxwell models (see Figure 5.1 on the previous page). Using this model, it is possible to represent a number of simpler viscoelastic models. For example, a simple Maxwell model is obtained by setting the elastic constants of all springs to zero, with the exception of the spring contained in the first Maxwell model ($\mu_1$). Similarly, the Kelvin-Voigt model may be obtained by setting the elastic constants $\mu_2 = \mu_3 = 0$, and setting $\mu_1 = \infty$ (or a very large number).

### 5.3.3 Formulation for Generalized Maxwell Models

As described above, the generalized Maxwell viscoelastic model consists of a number of Maxwell linear viscoelastic models in parallel with a spring, as shown in Figure 5.1 on the preceding page. PyLith includes the specific case of a spring in parallel with three Maxwell models. As described in the previous paragraph, a number of common material models may be obtained from this model by setting the shear moduli of various springs to zero or infinity (or a large number), such as the Maxwell model, the Kelvin model, and the standard linear solid. We follow formulations similar to those used by Zienkiewicz and Taylor [Zienkiewicz and Taylor, 2000] and Taylor [Taylor, 2003]. In this formulation, we specify the total shear modulus of the model ($\mu_{tot}$) and Lame's constant ($\lambda$). We then provide the fractional shear modulus for each Maxwell element spring in the model. It is not necessary to specify the fractional modulus for $\mu_0$, since this is obtained by subtracting the sum of the other ratios from 1. Note that the sum of all these fractions must equal 1. We use a similar formulation for our linear Maxwell viscoelastic model, but in that case $\mu_0$ is always zero and we only use a single Maxwell model. The parameters defining the standard Maxwell model are shown in Table 5.6 on page 73, and those defining the generalized Maxwell model are shown in Table 5.7 on page 73.

As for all our viscoelastic models, the volumetric strain is completely elastic, and the viscoelastic deformation may be expressed purely in terms of the deviatoric components:

$$\underline{S} = 2\mu_{tot}\left[\mu_0\underline{e} + \sum_{i=1}^{N}\mu_i\underline{q}^i - \underline{e}^I\right] + \underline{S}^I \,;\, P = 3K\left(\theta - \theta^I\right) + P^I\,, \tag{5.22}$$

where $K$ is the bulk modulus, $N$ is the number of Maxwell models, and the variable $\underline{q}^i$ follows the evolution equations

$$\underline{\dot{q}}^i + \frac{1}{\tau_i}\underline{q}^i = \underline{\dot{e}}. \tag{5.23}$$

The $\tau_i$ are the relaxation times for each Maxwell model:

$$\tau_i = \frac{\eta_i}{\mu_{tot}\mu_i}\,. \tag{5.24}$$

An alternative to the differential equation form above is an integral equation form expressed in terms of the relaxation modulus function. This function is defined in terms of an idealized experiment in which, at time labeled zero ($t = 0$), a specimen is subjected to a constant strain, $\underline{e}_0$, and the stress response, $\underline{S}(t)$, is measured. For a linear material we obtain:

$$\underline{S}(t) = 2\mu(t)\left(\underline{e}_0 - \underline{e}^I\right) + \underline{S}^I\,, \tag{5.25}$$

where $\mu(t)$ is the shear relaxation modulus function. Using linearity and superposition for an arbitrary state of strain yields an integral equation:

$$\underline{S}(t) = \int_{-\infty}^{t}\mu(t-T)\,\underline{\dot{e}}\,dT\,. \tag{5.26}$$

If we assume the modulus function in Prony series form we obtain

$$\mu(t) = \mu_{tot}\left(\mu_0 + \sum_{i=1}^{N}\mu_i\exp\frac{-t}{\tau_i}\right), \tag{5.27}$$

where

$$\mu_0 + \sum_{i=1}^{N}\mu_i = 1\,. \tag{5.28}$$

With the form in Equation 5.27, the integral equation form is identical to the differential equation form.

If we assume the material is undisturbed until a strain is suddenly applied at time zero, we can divide the integral into

$$\int_{-\infty}^{t}(\cdot)\,dT = \int_{-\infty}^{0^-}(\cdot)\,dT + \int_{0^-}^{0^+}(\cdot)\,dT + \int_{0^+}^{t}(\cdot)\,dT\,. \tag{5.29}$$

The first term is zero, the second term includes a jump term associated with $\underline{e}_0$ at time zero, and the last term covers the subsequent history of strain. Applying this separation to Equation 5.26,

$$\underline{S}(t) = 2\mu(t)\left(\underline{e}_0 - \underline{e}^I\right) + \underline{S}^I + 2\int_0^t\mu(t-T)\,\underline{\dot{e}}(T)\,dT\,, \tag{5.30}$$

where we have left the sign off of the lower limit on the integral.

Substituting Equation 5.27 into 5.30, we obtain

$$\underline{S}(t) = 2\mu_{tot}\left\{\mu_0\underline{e}(t) + \sum_{i=1}^{N}\left[\mu_i\exp\frac{-t}{\tau_i}\left(\underline{e}_0 + \int_0^t\exp\frac{t}{\tau_i}\underline{\dot{e}}(T)\,dT\right)\right] - \underline{e}^I\right\} + \underline{S}^I\,. \tag{5.31}$$

We then split each integral into two ranges: from 0 to $t_n$, and from $t_n$ to $t$, and define each integral as

$$\underline{i}_i^1(t) = \int_0^t \exp\frac{T}{\tau_i}\underline{\dot{e}}(T)\,dT. \tag{5.32}$$

The integral then becomes

$$\underline{i}_i^1(t) = \underline{i}_i^1(t_n) + \int_{t_n}^t \exp\frac{T}{\tau_i}\underline{\dot{e}}(T)\,dT. \tag{5.33}$$

Including the negative exponential multiplier:

$$\underline{h}_i^1(t) = \exp\frac{-t}{\tau_i}\underline{i}_i^1. \tag{5.34}$$

Then

$$\underline{h}_i^1(t) = \exp\frac{-\Delta t}{\tau_i}\underline{h}_i^1(t_n) + \Delta\underline{h}_i, \tag{5.35}$$

where

$$\Delta\underline{h}_i = \exp\frac{-t}{\tau_i}\int_{t_n}^t \exp\frac{T}{\tau_i}\underline{\dot{e}}(T)\,dT. \tag{5.36}$$

Approximating the strain rate as constant over each time step, the solution may be found as

$$\Delta\underline{h}_i = \frac{\tau_i}{\Delta t}\left(1 - \exp\frac{-\Delta t}{\tau_i}\right)\left(\underline{e} - \underline{e}_n\right) = \Delta h_i\left(\underline{e} - \underline{e}_n\right). \tag{5.37}$$

The approximation is singular for zero time steps, but a series expansion may be used for small time-step sizes:

$$\Delta h_i \approx 1 - \frac{1}{2}\left(\frac{\Delta t}{\tau_i}\right) + \frac{1}{3!}\left(\frac{\Delta t}{\tau_i}\right)^2 - \frac{1}{4!}\left(\frac{\Delta t}{\tau_i}\right)^3 + \cdots. \tag{5.38}$$

This converges with only a few terms. With this formulation, the constitutive relation now has the simple form:

$$\underline{S}(t) = 2\mu_{tot}\left(\mu_0\underline{e}(t) + \sum_{i=1}^N \mu_i\underline{h}_i^1(t) - \underline{e}^I\right) + \underline{S}^I. \tag{5.39}$$

We need to compute the tangent constitutive matrix when forming the stiffness matrix. In addition to the volumetric contribution to the tangent constitutive matrix, we require the deviatoric part:

$$\frac{\partial\underline{S}}{\partial\underline{\epsilon}} = \frac{\partial\underline{S}}{\partial\underline{e}}\frac{\partial\underline{e}}{\partial\underline{\epsilon}}, \tag{5.40}$$

where the second derivative on the right may be easily deduced from Equation 5.17 on page 68. The other derivative is given by

$$\frac{\partial\underline{S}}{\partial\underline{e}} = 2\mu_{tot}\left[\mu_0\underline{I} + \sum_{i=1}^N \mu_i\frac{\partial\underline{h}_i^1}{\partial\underline{e}}\right], \tag{5.41}$$

where $\underline{I}$ is the identity matrix. From Equations 5.35 through 5.37, the derivative inside the brackets is

$$\frac{\partial\underline{h}_i^1}{\partial\underline{e}} = \Delta h_i(\Delta t)\,\underline{I}. \tag{5.42}$$

The complete deviatoric tangent relation is then

$$\frac{\partial\underline{S}}{\partial\underline{\epsilon}} = 2\mu_{tot}\left[\mu_0 + \sum_{i=1}^N \mu_i\Delta h_i(\Delta t)\right]\frac{\partial\underline{e}}{\partial\underline{\epsilon}}. \tag{5.43}$$

We use this formulation for both our Maxwell and generalized Maxwell viscoelastic models. For the Maxwell model, $\mu_0 = 0$ and $N = 1$. For the generalized Maxwell model, $N = 3$. The stable time step is equal to 1/5 of the minimum relaxation time for all of the Maxwell models (equation 5.24 on the preceding page).

Table 5.6: Values in spatial databases for the linear Maxwell viscoelastic material constitutive model.

| Spatial database | Value | Description |
|---|---|---|
| **db_initial_stress** | | |
| **db_properties** | `vp` | Compressional wave speed, $v_p$ |
| | `vs` | Shear wave speed, $v_s$ |
| | `density` | Density, $\rho$ |
| | `viscosity` | Viscosity, $\eta$ |
| **db_initial_stress** | `stress-xx`, ... | Initial stress components |
| **db_initial_strain** | `total-strain-xx`, ... | Initial strain components |
| **db_initial_state** | `viscous-strain-xx`, ... | Initial viscous strain components |
| | `stress-zz-initial` | Initial out-of-plane stress (2D only) |

Table 5.7: Values in spatial database used as parameters in the generalized linear Maxwell viscoelastic material constitutive model.

| Spatial database | Value | Description |
|---|---|---|
| **db_properties** | `vp` | Compressional wave speed, $v_p$ |
| | `vs` | Shear wave speed, $v_s$ |
| | `density` | Density, $\rho$ |
| | `shear-ratio-1` | Shear ratio for Maxwell model 1, $\mu_1/\mu_{tot}$ |
| | `shear-ratio-2` | Shear ratio for Maxwell model 2, $\mu_2/\mu_{tot}$ |
| | `shear-ratio-3` | Shear ratio for Maxwell model 3, $\mu_3/\mu_{tot}$ |
| | `viscosity-1` | Viscosity for Maxwell model 1, $\eta_1$ |
| | `viscosity-2` | Viscosity for Maxwell model 2, $\eta_2$ |
| | `viscosity-3` | Viscosity for Maxwell model 3, $\eta_3$ |
| **db_initial_stress** | `stress-xx`, ... | Initial stress components |
| **db_initial_strain** | `total-strain-xx`, ... | Initial strain components |
| **db_initial_state** | `viscous-strain-1-xx`, ... | Initial viscous strain components for Maxwell model 1 |
| | `viscous-strain-2-xx`, ... | Initial viscous strain components for Maxwell model 2 |
| | `viscous-strain-3-xx`, ... | Initial viscous strain components for Maxwell model 3 |
| | `stress-zz-initial` | Initial out-of-plane stress (2D only) |

### 5.3.4 Effective Stress Formulations for Viscoelastic Materials

As an alternative to the approach outlined above, an effective stress function formulation [Kojic and Bathe, 1987] may be employed for both a linear Maxwell model and a power-law Maxwell model. Note that this formulation is not presently employed for linear viscoelastic models (see Appendix D on page 275), but it is used for power-law viscoelastic materials. For the viscoelastic materials considered here, the viscous volumetric strains are zero (incompressible flow), and it is convenient to separate the general stress-strain relationship at time $t+\Delta t$ into deviatoric and volumetric parts:

$$\begin{aligned} {}^{t+\Delta t}\underline{S} &= \frac{E}{1+\nu}\left({}^{t+\Delta t}\underline{e} - {}^{t+\Delta t}\underline{e}^C - \underline{e}^I\right) + \underline{S}^I = \frac{1}{a_E}\left({}^{t+\Delta t}\underline{e} - {}^{t+\Delta t}\underline{e}^C - \underline{e}^I\right) \\ {}^{t+\Delta t}P &= \frac{E}{1-2\nu}\left({}^{t+\Delta t}\theta - \theta^I\right) + P^I = \frac{1}{a_m}\left({}^{t+\Delta t}\theta - \theta^I\right), \end{aligned} \tag{5.44}$$

where ${}^{t+\Delta t}\underline{e}$ is the total deviatoric strain, ${}^{t+\Delta t}\underline{e}^C$ is the total viscous strain, $\underline{e}^I$ is the initial deviatoric strain, ${}^{t+\Delta t}P$ is the pressure, ${}^{t+\Delta t}\theta$ is the mean strain evaluated at time $t+\Delta t$ , and $\theta^I$ is the initial mean strain. The initial deviatoric stress and initial pressure are given by $\underline{S}^I$ and $P^I$, respectively. The topmost equation in Equation 5.44 may also be written as

$${}^{t+\Delta t}\underline{S} = \frac{1}{a_E}({}^{t+\Delta t}\underline{e}' - \underline{\Delta e}^C) + \underline{S}^I, \tag{5.45}$$

where

$${}^{t+\Delta t}\underline{e}' = {}^{t+\Delta t}\underline{e} - {}^{t}\underline{e}^C - \underline{e}^I,\ \underline{\Delta e}^C = {}^{t+\Delta t}\underline{e}^C - {}^{t}\underline{e}^C. \tag{5.46}$$

The creep strain increment is approximated using

$$\underline{\Delta e}^C = \Delta t^\tau \gamma^\tau \underline{S}, \tag{5.47}$$

where, using the $\alpha$-method of time integration,

$${}^\tau\underline{S} = (1-\alpha)^t_I\underline{S} + \alpha^{t+\Delta t}_I\underline{S} + \underline{S}^I = (1-\alpha)^t\underline{S} + \alpha^{t+\Delta t}\underline{S}, \tag{5.48}$$

and

$${}^\tau\gamma = \frac{3\Delta\bar{e}^C}{2\Delta t^\tau\bar{\sigma}}, \tag{5.49}$$

where

$$\Delta\bar{e}^C = \sqrt{\frac{2}{3}\underline{\Delta e}^C\cdot\underline{\Delta e}^C} \tag{5.50}$$

and

$${}^\tau\bar{\sigma} = (1-\alpha)^t_I\bar{\sigma} + \alpha^{t+\Delta t}_I\bar{\sigma} + \bar{\sigma}^I = \sqrt{3^\tau J_2'}. \tag{5.51}$$

To form the global stiffness matrix, it is necessary to provide a relationship for the viscoelastic tangent material matrix relating stress and strain. If we use vectors composed of the stresses and tensor strains, this relationship is

$$\underline{C}^{VE} = \frac{\partial^{t+\Delta t}\vec{\sigma}}{\partial^{t+\Delta t}\vec{\epsilon}}. \tag{5.52}$$

In terms of the vectors, we have

$$\begin{aligned} {}^{t+\Delta t}\sigma_i &= {}^{t+\Delta t}S_i + {}^{t+\Delta t}P;\ \ i = 1,2,3 \\ {}^{t+\Delta t}\sigma_i &= {}^{t+\Delta t}S_i;\qquad\quad i = 4,5,6 \end{aligned} \tag{5.53}$$

Therefore,

$$C^{VE}_{ij} = C'_{ij} + \frac{1}{3a_m}\,; \; 1 \le i, j \le 3\,. \qquad (5.54)$$
$$C^{VE}_{ij} = C'_{ij}\,; \qquad \text{otherwise}$$

Using the chain rule,

$$C'_{ij} = \frac{\partial^{t+\Delta t} S_i}{\partial^{t+\Delta t} \epsilon_j} = \frac{\partial^{t+\Delta t} S_i}{\partial^{t+\Delta t} e'_k} \frac{\partial^{t+\Delta t} e'_k}{\partial^{t+\Delta t} e_l} \frac{\partial^{t+\Delta t} e_l}{\partial^{t+\Delta t} \epsilon_j}\,. \qquad (5.55)$$

From Equation 5.46 on the preceding page, we obtain

$$\frac{\partial^{t+\Delta t} e'_k}{\partial^{t+\Delta t} e_l} = \delta_{kl}\,, \qquad (5.56)$$

and from Equation 5.17 on page 68:

$$\frac{\partial^{t+\Delta t} e_l}{\partial^{t+\Delta t} \epsilon_j} = \frac{1}{3}\begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix}; \; 1 \le l, j \le 3 \qquad (5.57)$$
$$\frac{\partial^{t+\Delta t} e_l}{\partial^{t+\Delta t} \epsilon_j} = \delta_{lj}\,; \qquad \text{otherwise.}$$

The first term of Equation 5.55 depends on the particular constitutive relationship, and the complete tangent matrix may then be obtained from Equation 5.54.

#### 5.3.4.1 Power-Law Maxwell Viscoelastic Material

Laboratory results on rock rheology are typically performed using a triaxial experiment, and the creep data are fit to a power-law equation of the form (e.g., [Kirby and Kronenberg, 1987]):

$$\dot{\epsilon}^C_{11} = A_E \exp\left(\frac{-Q}{RT}\right)(\sigma_1 - \sigma_3)^n = A_E \exp\left(\frac{-Q}{RT}\right)\sigma^n_d\,, \qquad (5.58)$$

where $\dot{\epsilon}^C_{11}$ is the strain rate in the direction of the maximum principal stress ($\sigma_1$), $A_E$ is the experimentally-derived pre-exponential constant, $Q$ is the activation enthalpy, $R$ is the universal gas constant, $T$ is the absolute temperature, $n$ is the power-law exponent, $\sigma_3$ $(= \sigma_2)$ is equal to the confining pressure, and $\sigma_d$ is the differential stress. To properly formulate the flow law, it must be generalized so that the results are not influenced by the experiment type or the choice of coordinate systems (e.g., [Paterson, 1994]). The flow law may then be generalized in terms of the deviatoric stress and strain rate invariants:

$$\sqrt{\dot{L}'^C_2} = A_M \exp\left(\frac{-Q}{RT}\right)\sqrt{J'_2}^{\,n}\,, \qquad (5.59)$$

where $A_M$ is now a pre-exponential constant used in the formulation for modeling. In practice, it is necessary to compute each strain rate component using the flow law. This is accomplished using:

$$\dot{e}^C_{ij} = A_M \exp\left(\frac{-Q}{RT}\right)\sqrt{J'_2}^{\,n-1} S_{ij}\,. \qquad (5.60)$$

Note that Equations 5.59 and 5.60 are consistent, since Equation 5.59 may be obtained from Equation 5.60 by taking the scalar inner product of both sides, multiplying by 1/2, and taking the square root.

In a triaxial experiment with confining pressure $P_c$, we have

$$\begin{aligned} \sigma_2 &= \sigma_3 = P_c \\ \sigma_1 &= \sigma^{app}_1 \\ P &= \frac{\sigma_1 + 2P_c}{3}\,, \end{aligned} \qquad (5.61)$$

where $\sigma_1^{app}$ is the applied load. The deviatoric stresses are then:

$$S_1 = \frac{2}{3}(\sigma_1 - P_c)$$
$$S_2 = S_3 = -\frac{1}{3}(\sigma_1 - P_c)\,. \tag{5.62}$$

This gives

$$S_1 = \frac{2}{3}(\sigma_1 - \sigma_3) = \frac{2}{3}\sigma_d$$
$$S_2 = S_3 = -\frac{1}{3}(\sigma_1 - \sigma_3) = -\frac{1}{3}\sigma_d\,. \tag{5.63}$$

In terms of the second deviatoric stress invariant, we then have

$$\sqrt{J_2'} = \frac{\sigma_d}{\sqrt{3}}\,. \tag{5.64}$$

Under the assumption that the creep measured in the laboratory experiments is incompressible, we have

$$\dot{e}_{11}^C = \dot{\epsilon}_{11}$$
$$\dot{e}_{22}^C = \dot{e}_{33}^C = -\frac{1}{2}\dot{\epsilon}_{11}\,. \tag{5.65}$$

In terms of the second deviatoric strain rate invariant we then have

$$\sqrt{\dot{L}_2'^C} = \frac{\sqrt{3}}{2}\dot{\epsilon}_{11}\,. \tag{5.66}$$

Substituting Equations 5.64 and 5.66 into Equation 5.58 on the previous page, we obtain

$$\sqrt{\dot{L}_2'^C} = A_E \frac{\sqrt{3}^{n+1}}{2} \exp\left(\frac{-Q}{RT}\right)\sqrt{J_2'}^n\,, \tag{5.67}$$

and therefore,

$$A_M = \frac{\sqrt{3}^{n+1}}{2} A_E\,. \tag{5.68}$$

When the exponential factor is included, we define a new parameter:

$$A_T = A_M \exp\left(\frac{-Q}{RT}\right) = \frac{\sqrt{3}^{n+1}}{2} A_E \exp\left(\frac{-Q}{RT}\right)\,. \tag{5.69}$$

There is a problem with the usage of parameters $A_E$, $A_M$, and $A_T$. Since the dimensions of these parameters are dependent on the value of the power-law exponent, they are not really constants. In addition to being logically inconsistent, this presents problems when specifying parameters for PyLith, since the power-law exponent must be known before the units can be determined. An alternative way of writing the flow rule is (e.g., [Prentice, 1968]):

$$\frac{\sqrt{\dot{L}_2'^C}}{\dot{e}_0} = \left(\frac{\sqrt{J_2'}}{S_0}\right)^n\,, \tag{5.70}$$

where $\dot{e}_0$ and $S_0$ are reference values for the strain rate and deviatoric stress. This means that

$$\frac{\dot{e}_0}{S_0^n} = A_T\,. \tag{5.71}$$

Users must therefore specify three parameters for a power-law material. The properties `reference-strain-rate`, `reference-stress`, and `power-law-exponent` in Table 5.8 on page 79 refer to $\dot{e}_0$, $S_0$, and $n$, respectively. To specify the power-law properties for PyLith using laboratory results, the user must first compute $A_T$ using Equation 5.69 on the preceding page. Then, values for $\dot{e}_0$ and $S_0$ must be provided. The simplest method is probably to assume a reasonable value for the reference strain rate, and then compute $S_0$ as

$$S_0 = \left(\frac{\dot{e}_0}{A_T}\right)^{\frac{1}{n}} . \tag{5.72}$$

A utility code (`powerlaw_gendb.py`) is provided to convert laboratory results to the properties used by PyLith. To use the code, users must specify the spatial variation of $A_E$, $Q$, $n$, and $T$. An additional parameter is given to define the units of $A_E$. The user then specifies either a reference stress or a reference strain rate, and a database suitable for PyLith is generated. This utility is described more fully in Section 7.9.6.6 on page 154.

The flow law in component form is

$$\dot{e}^C_{ij} = \frac{\dot{e}_0 \sqrt{J'_2}^{\,n-1} S_{ij}}{S^n_0} , \tag{5.73}$$

and the creep strain increment is approximated as

$$\underline{\Delta e}^C \approx \frac{\Delta t \dot{e}_0 \sqrt{{}^\tau J'_2}^{\,n-1}\, {}^\tau\underline{S}}{S^n_0} = \frac{\Delta t \dot{e}_0 {}^\tau\overline{\sigma}^{n-1}\, {}^\tau\underline{S}}{\sqrt{3} S^n_0} . \tag{5.74}$$

Therefore,

$$\Delta\bar{e}^C \approx \frac{2\Delta t \dot{e}_0 \sqrt{{}^\tau J'_2}^{\,n}}{\sqrt{3} S^n_0} = \frac{2\Delta t \dot{e}_0 {}^\tau\overline{\sigma}^n}{\sqrt{3}^{n+1} S^n_0} , \text{and}\, {}^\tau\gamma = \frac{\dot{e}_0 \sqrt{{}^\tau J'_2}^{\,n-1}}{S^n_0} . \tag{5.75}$$

substituting Equations 5.48 on page 74, 5.74, and 5.75 into 5.45 on page 74, we obtain:

$$ {}^{t+\Delta t}\underline{S} = \frac{1}{a_E}\left\{ {}^{t+\Delta t}\underline{e}' - \Delta t^\tau\gamma\left[(1-\alpha)\, {}^t\underline{S} + \alpha\, {}^{t+\Delta t}\underline{S}\right]\right\} + \underline{S}^I , \tag{5.76}$$

which may be rewritten:

$$ {}^{t+\Delta t}\underline{S}\left(a_E + \alpha\Delta t^\tau\gamma\right) = {}^{t+\Delta t}\underline{e}' - \Delta t^\tau\gamma(1-\alpha)\, {}^t\underline{S} + a_E \underline{S}^I . \tag{5.77}$$

Taking the scalar inner product of both sides we obtain:

$$a^2\, {}^{t+\Delta t}J'_2 - b + c^\tau\gamma - d^2\, {}^\tau\gamma^2 = F = 0 , \tag{5.78}$$

where

$$\begin{gathered} a = a_E + \alpha\Delta t^\tau\gamma \\ b = \frac{1}{2}\, {}^{t+\Delta t}\underline{e}' \cdot {}^{t+\Delta t}\underline{e}' + a_E\, {}^{t+\Delta t}\underline{e}' \cdot \underline{S}^I + a_E^2\, {}^I J'_2 . \\ c = \Delta t(1-\alpha)\, {}^{t+\Delta t}\underline{e}' \cdot {}^t\underline{S} + \Delta t(1-\alpha)\, a_E\, {}^t\underline{S} \cdot \underline{S}^I \\ d = \Delta t(1-\alpha)\sqrt{{}^t J'_2} \end{gathered} \tag{5.79}$$

Equation 5.78 is a function of a single unknown – the square root of the second deviatoric stress invariant at time $t + \Delta t$ – and may be solved by bisection or by Newton's method. Once this parameter has been found, the deviatoric stresses for the current time step may be found from Equations 5.51 on page 74, 5.75, and 5.76, and the total stresses may be found by combining the deviatoric and volumetric components from Equation 5.44 on page 74.

Once the stresses are computed for the current time step, we can compute the relaxation time (used in computing the stable time step) by first computing the effective viscous strain rate from Equation 5.73:

$$\dot{\bar{e}}^C = \frac{2\dot{e}_0\left(\frac{\bar{\sigma}}{\sqrt{3}}\right)^n}{\sqrt{3} S^n_0} . \tag{5.80}$$

Similarly, the effective elastic strain is computed as:

$$\bar{e}^E = \frac{\bar{\sigma}}{3\mu}. \tag{5.81}$$

The relaxation time is then the ratio between these two:

$$\tau = \frac{\bar{e}^E}{\bar{\dot{e}}^C} = \left(\frac{S_0}{\sqrt{J_2'}}\right)^{n-1} \frac{S_0}{6\mu \dot{e}_0}. \tag{5.82}$$

The stable time step returned by PyLith is 1/5 of the value computed from Equation 5.82.

To compute the tangent stress-strain relation, we need to compute the first term in Equation 5.55 on page 75. We begin by rewriting Equation 5.77 on the previous page as

$$F = {}^{t+\Delta t}S_i\left(a_E + \alpha \Delta t^\tau \gamma\right) - {}^{t+\Delta t}e_i' + \Delta t^\tau \gamma (1-\alpha)^t S_i - a_E S_i^I = 0. \tag{5.83}$$

The derivative of this function with respect to ${}^{t+\Delta t}e_k''$ is

$$\frac{\partial F}{\partial^{t+\Delta t}e_k'} = -\delta_{ik}, \tag{5.84}$$

and the derivative with respect to ${}^{t+\Delta t}S_i$ is

$$\frac{\partial F}{\partial^{t+\Delta t}S_i} = a_E + \alpha \Delta t^\tau \gamma + \frac{\partial^\tau \gamma}{\partial^{t+\Delta t}S_i} \Delta t \left[\alpha^{t+\Delta t}S_i + (1-\alpha)^t S_i\right]. \tag{5.85}$$

From Equation 5.75 on the preceding page and Equation 5.51 on page 74,

$${}^\tau\gamma = \frac{\dot{e}_0}{S_0^n}\left[\alpha\sqrt{{}^{t+\Delta t}J_2'} + (1-\alpha)\sqrt{{}^t J_2'}\right]^{n-1}. \tag{5.86}$$

Then

$$\begin{aligned} \frac{\partial^\tau \gamma}{\partial^{t+\Delta t}S_i} &= \frac{\partial^\tau \gamma}{\partial\sqrt{{}^{t+\Delta t}J_2'}} \frac{\partial\sqrt{{}^{t+\Delta t}J_2'}}{\partial^{t+\Delta t}S_l} \\ &= \frac{\dot{e}_0 \alpha (n-1)\sqrt{{}^\tau J_2'}^{\,n-2}\,{}^{t+\Delta t}T_i}{2S_0^n}, \end{aligned} \tag{5.87}$$

where

$$\begin{aligned} {}^{t+\Delta t}T_i &= {}^{t+\Delta t}S_i;\ 1 \le i \le 3 \\ {}^{t+\Delta t}T_i &= 2^{t+\Delta t}S_i;\ \text{otherwise.} \end{aligned} \tag{5.88}$$

Then using Equations 5.84, 5.85, 5.87, and the quotient rule for derivatives of an implicit function,

$$\frac{\partial^{t+\Delta t}S_i}{\partial^{t+\Delta t}e_k'} = \frac{\delta_{ik}}{a_E + \alpha\Delta t\left[{}^\tau\gamma + \frac{\dot{e}_0{}^\tau S_i (n-1)^{t+\Delta t}T_i \sqrt{{}^\tau J_2'}^{\,n-2}}{2\sqrt{{}^{t+\Delta t}J_2'}S_0^n}\right]}. \tag{5.89}$$

Note that for a linear material ($n = 1$), this equation is identical to the linear formulation in Section D.1.1 on page 275 (making the appropriate substitution for ${}^\tau\gamma$). Then, using Equations 5.54 on page 75 through 5.57 on page 75,

$$C_{ij}^{VE} = \frac{1}{3a_m}\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} + \frac{1}{3}\frac{\partial^{t+\Delta t}S_i}{\partial^{t+\Delta t}e_k'}\begin{bmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ -1 & -1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 \end{bmatrix}. \tag{5.90}$$

Note that if there are no deviatoric stresses at the beginning and end of a time step (or if $\dot{e}_0/S_0^n$ approaches zero), Equations 5.89 on the facing page and 5.90 on the preceding page reduce to the elastic constitutive matrix, as expected.

To compute the zero of the effective stress function using Newton's method, we require the derivative of Equation 5.78 on page 77, which may be written:

$$\frac{\partial F}{\partial\sqrt{^{t+\Delta t}J_2'}} = 2a^2\sqrt{^{t+\Delta t}J_2'} + \frac{\dot{e}_0\alpha\,(n-1)\sqrt{^{\tau}J_2'}^{\,n-2}}{S_0^n}\left(2a\alpha\Delta t^{t+\Delta t}J_2' + c - 2d^{2\,\tau}\gamma\right). \tag{5.91}$$

Table 5.8: Values in spatial database used as parameters in the nonlinear power-law viscoelastic material constitutive model.

| Spatial database | Value | Description |
|---|---|---|
| **db_properties** | `vp` | Compressional wave speed, $v_p$ |
| | `vs` | Shear wave speed, $v_s$ |
| | `density` | Density, $\rho$ |
| | `reference-strain-rate` | Reference strain rate, $\dot{e}_0$ |
| | `reference-stress` | Reference stress, $S_0$ |
| | `power-law-exponent` | Power-law exponent, $n$ |
| **db_initial_stress** | `stress-xx, ...` | Initial stress components |
| **db_initial_strain** | `total-strain-xx, ...` | Initial strain components |
| **db_initial_state** | `viscous-strain-xx, ...` | Initial viscous strain components |
| | `stress-zz-initial` | Initial out-of-plane stress (2D only) |

## 5.4 Elastoplastic Materials

PyLith presently contains just a single elastoplastic material that implements the Drucker-Prager yield criterion. Future releases of PyLith may contain additional elastoplastic materials, such as Drucker-Prager with hardening/softening.

### 5.4.1 General Elastoplasticity Formulation

The elastoplasticity formulation in PyLith is based on an additive decomposition of the total strain into elastic and plastic parts:

$$d\epsilon_{ij} = d\epsilon_{ij}^E + d\epsilon_{ij}^P. \tag{5.92}$$

The stress increment is then given by

$$d\sigma_{ij} = C_{ijrs}^E\left(d\epsilon_{rs} - d\epsilon_{rs}^P\right), \tag{5.93}$$

where $C_{ijrs}^E$ are the components of the elastic constitutive tensor. To completely specify an elastoplastic problem, three components are needed. We first require a yield condition, which specifies the state of stress at which plastic flow initiates. This is generally given in the form:

$$f\left(\underline{\sigma}, k\right) = 0, \tag{5.94}$$

where $k$ is an internal state parameter. It is then necessary to specify a flow rule, which describes the relationship between plastic strain and stress. The flow rule is given in the form:

$$g\left(\underline{\sigma}, k\right) = 0. \tag{5.95}$$

The plastic strain increment is then given as

$$d\epsilon_{ij}^P = d\lambda\frac{\partial g}{\partial\sigma_{ij}}, \tag{5.96}$$

where $d\lambda$ is the scalar plastic multiplier. When the flow rule is identical to the yield criterion ($f \equiv g$), the plasticity is described as associated. Otherwise, it is non-associated. The final component needed is a hardening hypothesis, which describes how the

Table 5.9: Options for fitting the Drucker-Prager plastic parameters to a Mohr-Coulomb model using **fit_mohr_coulomb**.

| **Parameter Value** | $\alpha_f$ | $\beta$ | $\alpha_g$ |
|---|---|---|---|
| inscribed | $\frac{2\sin\phi(k)}{\sqrt{3}(3-\sin\phi(k))}$ | $\frac{6\bar{c}(k)\cos\phi_0}{\sqrt{3}(3-\sin\phi_0)}$ | $\frac{2\sin\psi(k)}{\sqrt{3}(3-\sin\psi(k))}$ |
| middle | $\frac{\sin\phi(k)}{3}$ | $\bar{c}(k)\cos(\phi_0)$ | $\frac{\sin\psi(k)}{3}$ |
| circumscribed | $\frac{2\sin\phi(k)}{\sqrt{3}(3+\sin\phi(k))}$ | $\frac{6\bar{c}(k)\cos\phi_0}{\sqrt{3}(3+\sin\phi_0)}$ | $\frac{2\sin\psi(k)}{\sqrt{3}(3+\sin\psi(k))}$ |

yield condition and flow rule are modified during plastic flow. When the yield condition and flow rule remain constant during plastic flow (e.g., no hardening), the material is referred to as perfectly plastic.

To perform the solution, the yield condition (Equation 5.94 on the previous page) is first evaluated under the assumption of elastic behavior. If ${}^{t+\Delta t}f < 0$, the material behavior is elastic and no plastic flow occurs. Otherwise, the behavior is plastic and a plastic strain increment must be computed to return the stress state to the yield envelope. This procedure is known as an elastic predictor-plastic corrector algorithm.

### 5.4.2 Drucker-Prager Elastoplastic Material

PyLith includes an elastoplastic implementation of the Drucker-Prager yield criterion [Drucker and Prager, 1952]. This criterion was originally devised to model plastic deformation of soils, and it has also been used to model rock deformation. It is intended to be a smooth approximation of the Mohr-Coulomb yield criterion. The implementation used in PyLith includes non-associated plastic flow, which allows control over the unreasonable amounts of dilatation that are sometimes predicted by the associated model. The model is described by the following yield condition:

$$f\left(\underline{\sigma},k\right) = \alpha_f I_1 + \sqrt{J_2'} - \beta\,, \tag{5.97}$$

and a flow rule given by:

$$g\left(\underline{\sigma},k\right) = \sqrt{J_2'} + \alpha_g I_1\,. \tag{5.98}$$

The yield surface represents a circular cone in principal stress space, and the parameters can be related to the friction angle, $\phi$, and the cohesion, $\bar{c}$, of the Mohr-Coulomb model. The yield surface in Haigh-Westergaard space ($\zeta = \frac{1}{\sqrt{3}}I_1, p = \sqrt{2J_2}, \cos(3\theta) = \frac{3\sqrt{3}}{2}\frac{J_3}{J_2^{3/2}}$) is

$$\left(\sqrt{3}\sin\left(\theta+\frac{\pi}{3}\right) - \sin\phi\cos\left(\theta+\frac{\pi}{3}\right)\right)p - \sqrt{2}\sin\phi\zeta = \sqrt{6}\bar{c}\cos\theta. \tag{5.99}$$

The yield surface can be fit to the Mohr-Coulomb model in several different ways. The yield surface can touch the outer apices ($\theta = \pi/3$) of the Mohr-Coulomb model (inscribed version), the inner apices ($\theta = 0$) of the Mohr-Coulomb model (circumscribed version), or halfway between the two ($\theta = pi/6$,middle version). Substituting these values for $\theta$ into Equation ( 5.99) and casting it into the same form as Equation ( 5.98) yields the values of $\alpha_f$, $\beta$, and $\alpha_g$ given in Table 5.9, where $\phi_0$ refers to the initial friction angle. Similarly, the flow rule can be related to the dilatation angle, $\psi$, of a Mohr-Coulomb model. It is also possible for the Mohr-Coulomb parameters to be functions of the internal state parameter, $k$. In PyLith, the fit to the Mohr-Coulomb yield surface and flow rule is controlled by the `fit_mohr_coulomb` property.

As for the viscoelastic models, it is convenient to separate the deformation into deviatoric and volumetric parts:

$$\begin{aligned}
{}^{t+\Delta t}S_{ij} &= \frac{1}{a_E}\left({}^{t+\Delta t}e_{ij} - {}^{t+\Delta t}e^P_{ij} - e^I_{ij}\right) + S^I_{ij} = \frac{1}{a_E}\left({}^{t+\Delta t}e'_{ij} - \Delta e^P_{ij}\right) + S^I_{ij} \\
{}^{t+\Delta t}P &= \frac{1}{a_m}\left({}^{t+\Delta t}\theta - {}^{t+\Delta t}\theta^P - \theta^I\right) + P^I = \frac{1}{a_m}\left({}^{t+\Delta t}\theta' - \Delta\theta^P\right) + P^I\,,
\end{aligned} \tag{5.100}$$

where

$$\begin{aligned}
{}^{t+\Delta t}e'_{ij} &= {}^{t+\Delta t}e_{ij} - {}^{t}e^P_{ij} - e^I_{ij} \\
\Delta e^P_{ij} &= {}^{t+\Delta t}e^P_{ij} - {}^{t}e^P_{ij} \\
{}^{t+\Delta t}\theta' &= {}^{t+\Delta t}\theta - {}^{t}\theta^P - \theta^I \\
\Delta\theta^P &= {}^{t+\Delta t}\theta^P - {}^{t}\theta^P\,.
\end{aligned} \tag{5.101}$$

Since the plasticity is pressure-dependent, there are volumetric plastic strains, unlike the viscous strains in the previous section. From Equation 5.96 on page 79, the plastic strain increment is

$$\Delta\epsilon^P_{ij} = \lambda\frac{\partial^{t+\Delta t}g}{\partial^{t+\Delta t}\sigma_{ij}} = \lambda\alpha_g\delta_{ij} + \lambda\frac{^{t+\Delta t}S_{ij}}{2\sqrt{^{t+\Delta t}J'_2}}\,. \tag{5.102}$$

The volumetric part is

$$\Delta\theta^P = \frac{1}{3}\Delta\epsilon^P_{ii} = \lambda\alpha_g\,, \tag{5.103}$$

and the deviatoric part is

$$\Delta e^P_{ij} = \Delta\epsilon^P_{ij} - \Delta\epsilon^P_m\delta_{ij} = \lambda\frac{^{t+\Delta t}S_{ij}}{2\sqrt{^{t+\Delta t}J'_2}}\,. \tag{5.104}$$

The problem is reduced to solving for $\lambda$. The procedure is different depending on whether hardening is included.

#### 5.4.2.1 Drucker-Prager Elastoplastic With No Hardening (Perfectly Plastic)

When there is no hardening (perfect plasticity), the Drucker-Prager elastoplastic model may be parameterized with just three parameters, in addition to the normal elasticity parameters. The parameters `friction-angle`, `cohesion`, and `dilatation-angle` in Table 5.10 on page 83 refer respectively to $\phi$, $\bar{c}$, and $\psi$ in Table 5.9 on the preceding page. These are then converted to the properties $\alpha_f$ (`alpha-yield`), $\beta$ (`beta`), and $\alpha_g$ (`alpha-flow`), as shown in Table 5.1 on page 63.

For perfect plasticity the yield and flow functions do not vary, and we can solve for $\lambda$ by substituting Equation 5.104 into Equation 5.100 on the facing page and taking the scalar inner product of both sides:

$$\lambda = \sqrt{2}\,^{t+\Delta t}d - 2a_E\sqrt{^{t+\Delta t}J'_2}\,, \tag{5.105}$$

where

$$^{t+\Delta t}d^2 = 2a_E^2J'^I_2 + 2a_ES^I_{ij}\,^{t+\Delta t}e'_{ij} + \,^{t+\Delta t}e'_{ij}\,^{t+\Delta t}e'_{ij}\,. \tag{5.106}$$

The second deviatoric stress invariant is therefore

$$\sqrt{^{t+\Delta t}J'_2} = \frac{\sqrt{2}\,^{t+\Delta t}d - \lambda}{2a_E}\,, \tag{5.107}$$

and the pressure is computed from Equations 5.100 on the preceding page and 5.103 as:

$$^{t+\Delta t}P = \frac{^{t+\Delta t}I_1}{3} = \frac{1}{a_m}\left(^{t+\Delta t}\theta' - \lambda\alpha_g\right) + P^I\,. \tag{5.108}$$

We then use the yield condition ($^{t+\Delta t}f = 0$) and substitute for the stress invariants at $t + \Delta t$ to obtain:

$$\lambda = \frac{2a_Ea_m\left(\frac{3\alpha_f}{a_m}\,^{t+\Delta t}\theta' + \frac{^{t+\Delta t}d}{\sqrt{2}a_E} - \beta + 3\alpha_fP^I\right)}{6\alpha_f\alpha_ga_E + a_m}\,. \tag{5.109}$$

Since $\lambda$ is now known, we can substitute 5.107 into 5.104 to obtain

$$^{t+\Delta t}S_{ij} = \frac{\Delta e^P_{ij}\left(\sqrt{2}\,^{t+\Delta t}d - \lambda\right)}{\lambda a_E}\,. \tag{5.110}$$

Substituting this into Equation 5.100 on the preceding page, we obtain the deviatoric plastic strain increment:

$$\Delta e^P_{ij} = \frac{\lambda}{\sqrt{2}\,^{t+\Delta t}d}\left(^{t+\Delta t}e'_{ij} + a_ES^I_{ij}\right)\,. \tag{5.111}$$

We then use Equation 5.103 on the preceding page and the second line of Equation 5.100 on page 80 to obtain the volumetric plastic strains and the pressure, and we use 5.111 on the preceding page and the first line of Equation 5.100 on page 80 to obtain the deviatoric plastic strains and the deviatoric stresses.

In certain cases where the mean stress is tensile, it is possible that the flow rule will not allow the stresses to project back to the yield surface, since they would project beyond the tip of the cone. Although this stress state is not likely to be encountered for quasi-static tectonic problems, it can occur for dynamic problems. One simple solution is to redefine the plastic multiplier, $\lambda$. We do this by taking the smaller of the values yielded by Equation 5.109 on the preceding page or by the following relation:

$$\lambda = \sqrt{2}\,{}^{t+\Delta t}d\,. \tag{5.112}$$

This is equivalent to setting the second deviatoric stress invariant to zero in Equation 5.105 on the previous page. By default, PyLith does not allow such tensile yield, since this would generally represent an error in problem setup for tectonic problems; however, for cases where such behavior is necessary, the material flag `allow_tensile_yield` may be set to `True`. This same criterion is used to determine whether a feasible stress state is attainable in cases where `allow_tensile_yield` is `False`. If Equation 5.109 on the preceding page yields a smaller value than Equation 5.112, this implies $\sqrt{{}^{t+\Delta t}J_2'} < 0$, which is not a feasible stress state (see Equation 5.105 on the preceding page).

To compute the elastoplastic tangent matrix we begin by writing Equation 5.100 on page 80 as a single expression in terms of stress and strain vectors:

$${}^{t+\Delta t}\sigma_i = \frac{1}{a_E}\left({}^{t+\Delta t}e_i' - \Delta e_i^P\right) + S_i^I + \frac{R_i}{a_m}\left({}^{t+\Delta t}\theta' - \Delta\theta^P\right) + R_i P^I \tag{5.113}$$

where

$$\begin{aligned} R_i &= 1\,;\ i = 1,2,3 \\ R_i &= 0\,;\ i = 4,5,6\,. \end{aligned} \tag{5.114}$$

The elastoplastic tangent matrix is then given by

$$C_{ij}^{EP} = \frac{\partial\,{}^{t+\Delta t}\sigma_i}{\partial\,{}^{t+\Delta t}\epsilon_j} = \frac{1}{a_E}\left(\frac{\partial\,{}^{t+\Delta t}e_i'}{\partial\,{}^{t+\Delta t}\epsilon_j} - \frac{\partial\Delta e_i^P}{\partial\,{}^{t+\Delta t}\epsilon_j}\right) + \frac{R_i}{a_m}\left(\frac{\partial\,{}^{t+\Delta t}\theta'}{\partial\,{}^{t+\Delta t}\epsilon_j} - \frac{\partial\Delta\theta^P}{\partial\,{}^{t+\Delta t}\epsilon_j}\right). \tag{5.115}$$

From Equations 5.17 on page 68 and 5.101 on page 80, we have

$$\frac{\partial\,{}^{t+\Delta t}e_i'}{\partial\,{}^{t+\Delta t}\epsilon_j} = \frac{1}{3}\begin{bmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ -1 & -1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 \end{bmatrix}, \tag{5.116}$$

and from Equations 5.16 on page 68 and 5.101 on page 80 we have

$$\frac{\partial\,{}^{t+\Delta t}\theta'}{\partial\,{}^{t+\Delta t}\epsilon_j} = \frac{R_j}{3}\,. \tag{5.117}$$

From Equation 5.111 on the preceding page we have

$$\frac{\partial\Delta e_i^P}{\partial\,{}^{t+\Delta t}\epsilon_j} = \frac{1}{\sqrt{2}\,{}^{t+\Delta t}d}\left[\left({}^{t+\Delta t}e_i' + a_E S_i^I\right)\left(\frac{\partial\lambda}{\partial\,{}^{t+\Delta t}\epsilon_j} - \frac{\lambda}{{}^{t+\Delta t}d}\frac{\partial\,{}^{t+\Delta t}d}{\partial\,{}^{t+\Delta t}\epsilon_j}\right) + \lambda\frac{\partial\,{}^{t+\Delta t}e_i'}{\partial\,{}^{t+\Delta t}\epsilon_j}\right]. \tag{5.118}$$

The derivative of ${}^{t+\Delta t}d$ is

$$\frac{\partial\,{}^{t+\Delta t}d}{\partial\,{}^{t+\Delta t}\epsilon_j} = \frac{a_E T_j^I + {}^{t+\Delta t}E_j}{{}^{t+\Delta t}d}\,, \tag{5.119}$$

where

$$T_j^I = S_j^I \text{ and } {}^{t+\Delta t}E_j = {}^{t+\Delta t}e'_j\,;\ j = 1,2,3$$
$$T_j^I = 2S_j^I \text{ and } {}^{t+\Delta t}E_j = 2{}^{t+\Delta t}e'_j\,;\ j = 4,5,6\,. \tag{5.120}$$

The derivative of ${}^{t+\Delta t}\lambda$ is a function of derivatives already computed:

$$\begin{aligned}\frac{\partial \lambda}{\partial^{t+\Delta t}\epsilon_j} &= \frac{2a_E a_m}{6\alpha_f\alpha_g a_E + a_m}\left(\frac{3\alpha_f}{a_m}\frac{\partial^{t+\Delta t}\theta'}{\partial^{t+\Delta t}\epsilon_j} + \frac{1}{\sqrt{2}a_E}\frac{\partial^{t+\Delta t}d}{\partial^{t+\Delta t}\epsilon_j}\right)\\ &= \frac{2a_E a_m}{6\alpha_f\alpha_g a_E + a_m}\left(\frac{\alpha_f R_j}{a_m} + \frac{a_E T_j^I + {}^{t+\Delta t}E_j}{\sqrt{2}a_E{}^{t+\Delta t}d}\right)\,.\end{aligned} \tag{5.121}$$

Finally, from Equation 5.103 on page 81, the derivative of the volumetric plastic strain increment is:

$$\frac{\partial \Delta\theta^P}{\partial^{t+\Delta t}\epsilon_j} = \alpha_g \frac{\partial \lambda}{\partial^{t+\Delta t}\epsilon_j}\,. \tag{5.122}$$

Table 5.10: Values in spatial database used as parameters in the Drucker-Prager elastoplastic model with perfect plasticity.

| Spatial database | Value | Description |
|---|---|---|
| **db_properties** | vp | Compressional wave speed, $v_p$ |
| | vs | Shear wave speed, $v_s$ |
| | density | Density, $\rho$ |
| | friction-angle | Friction angle, $\phi$ |
| | cohesion | Cohesion, $\bar{c}$ |
| | dilatation-angle | Dilatation angle, $\psi$ |
| **db_initial_stress** | stress-xx, ... | Initial stress components |
| **db_initial_strain** | total-strain-xx, ... | Initial strain components |
| **db_initial_state** | plastic-strain-xx, ... | Initial plastic strain components |
| | stress-zz-initial | Initial out-of-plane stress (2D only) |

In addition to the properties available for every material, the properties for the Drucker-Prager model also includes:

**fit_mohr_coulomb** Fit to the yield surface to the Mohr-Coulomb model (default is inscribed).

allow_tensile_yieldIf true, allow yield beyond tensile strength; otherwise an error message will occur when the model fails beyond the tensile strength (default is false).

DruckerPrager3D parameters in a cfg file

```
[pylithapp.timedependent]
materials = [plastic]
materials.plastic = pylith.materials.DruckerPrager3D

[pylithapp.timedependent.materials.plastic]
fit_mohr_coulomb = inscribed ; default
allow_tensile_yield = False ; default
```

# Chapter 6

# Boundary and Interface Conditions

## 6.1 Assigning Boundary Conditions

There are four basic steps in assigning a specific boundary condition to a portion of the domain.

1. Create sets of vertices in the mesh generation process for each boundary condition.
2. Define boundary condition groups corresponding to the vertex sets.
3. Set the parameters for each boundary condition group using `cfg` files and/or command line arguments.
4. Specify the spatial variation in parameters for the boundary condition using a spatial database file.

### 6.1.1 Creating Sets of Vertices

The procedure for creating sets of vertices differs depending on the mesh generator. For meshes specified using the PyLith mesh ASCII format, the sets of vertices are specified using groups (see Appendix C.1 on page 267). In CUBIT/Trelis the groups of vertices are created using nodesets. Similarly, in LaGriT, psets are used. Note that we chose to associate boundary conditions with groups of vertices because nearly every mesh generation package supports associating a string or integer with groups of vertices. Note also that we currently associate boundary conditions with string identifiers, so even if the mesh generator uses integers, the name is specified as the digits of the integer value. Finally, note that every vertex set that ultimately is associated with a boundary condition on a cell face (e.g., Neumann boundary conditions and fault interface conditions) must correspond to a simply-connected surface.

### 6.1.2 Arrays of Boundary Condition Components

A dynamic array of boundary condition components associates a name (string) with each boundary condition. This dynamic array of boundary conditions replaces the boundary condition container in PyLith v1.0. User-defined containers are no longer necessary, and the predefined containers are no longer available (or necessary). The default boundary condition for each component in the array is the DirichletBC object. Other boundary conditions can be bound to the named items in the array via a `cfg` file, `pml` file, or the command line. The parameters for the boundary condition are set using the name of the boundary condition.

Array of boundary conditions in a `cfg` file

```
[pylithapp.problem]
bc = [x_neg, x_pos, y_pos, z_neg] ; Array of boundary conditions

# Default boundary condition is DirichletBC
```

```
# Keep default value for bc.x_neg
bc.x_pos = pylith.bc.DirichletBoundary ; change BC type to DirichletBoundary
bc.y_pos = pylith.bc.AbsorbingDampers ; change BC type to AbsorbingDampers
bc.z_neg = pylith.bc.Neumann ; change BC type to Neumann (traction)
```

## 6.2 Time-Dependent Boundary Conditions

Several boundary conditions use a common formulation for the spatial and temporal variation of the boundary condition parameters,

$$f(\vec{x}) = f_0(\vec{x}) + \dot{f}_0(\vec{x})(t - t_0(\vec{x})) + f_1(\vec{x})a(t - t_1(\vec{x})), \tag{6.1}$$

where $f(\vec{x})$ may be a scalar or vector parameter, $f_0(\vec{x})$ is a constant value, $\dot{f}_0(\vec{x})$ is a constant rate of change in the value, $t_0(\vec{x})$ is the onset time for the constant rate of change, $f_1(\vec{x})$ is the amplitude for the temporal modulation, $a(t)$ is the variation in amplitude with time, $t_1(\vec{x})$ is the onset time for the temporal modulation, and $\vec{x}$ is the position of a location in space. This common formulation permits easy specification of a scalar or vector with a constant value, constant rate of change of a value, and/or modulation of a value in time. One can specify just the initial value, just the rate of change of the value (along with the corresponding onset time), or just the modulation in amplitude (along with the corresponding temporal variation and onset time), or any combination of the three. The facilities associated with this formulation are:

**db_initial** Spatial database specifying the spatial variation in the initial value (default is none).

**db_rate** Spatial database specifying rate of change in the value (default is none).

**db_change** Spatial database specifying the amplitude of the temporal modulation (default is none).

**th_change** Time history database specifying the temporal change in amplitude (default is none).

### 6.2.1 Dirichlet Boundary Conditions

Dirichlet boundary conditions in PyLith prescribe the displacement of a subset of the vertices of the finite-element mesh. While Dirichlet boundary conditions can be applied to any vertex, usually they are applied to vertices on the lateral and bottom boundaries of the domain. There are two types of Dirichlet boundary conditions, DirichletBC and DirichletBoundary. Both provide identical constraints on the solution, but DirichletBoundary is limited to vertices of a simply-connected surface, which allows diagnostic output of the prescribed displacements. DirichletBC can be applied to a set of unconnected vertices.

The properties and components common to both the DirichletBC and DirichletBoundary boundary conditions are:

**label** Label of the group of vertices associated with the boundary condition.

**bc_dof** Array of degrees of freedom to be fixed (first degree of freedom is 0).

DirichletBoundary contains an additional component:

**output** Manager for output of displacements on boundary with specified displacements.

By default the output manager does not output any information. The specified displacements and velocities can be output by including "displacement" and "velocity" in the output manager's **vertex_info_fields** array parameter.

DirichletBC parameters in a `cfg` file

```
[pylithapp.problem]
bc = [mybc]

[pylithapp.problem.bc.mybc]
label = group A
bc_dof = [2] ; fixed displacement in z direction
db_initial = spatialdata.spatialdb.SimpleDB
db_initial.iohandler.filename = disp_A.spatialdb
db_initial.query_type = nearest ; change query type to nearest point algorithm
db_rate = spatialdata.spatialdb.UniformDB
db_rate.values = [displacement-rate-z]
```

```
db_rate.data = [1.0e-06*m/s] ; velocity is 1.0e-06 m/s
```

We have created an array with one boundary condition, mybc. The group of vertices associated with the boundary condition is group A. For the database associated with the constant displacement, we use a SimpleDB. We set the filename and query type for the database. For the rate of change of values, we use a UniformDB and specify the velocity in the z-direction to be 1.0e-06 m/s. See Section 4.5 on page 43 for a discussion of the different types of spatial databases available.

Table 6.1: Fields available in output of DirichletBoundary boundary condition information.

| Field Type | Field | Description |
|---|---|---|
| vertex_info_fields | displacement_initial | Initial displacement field in global coordinate system |
| | velocity | Rate of change of displacement field in global coordinate system |
| | velocity_start_time | Onset time in seconds for rate of change in displacement field |
| | displacement_change | Amplitude of change in displacement field in global coordinate system |
| | change_start_time | Onset time in seconds for the amplitude change in the displacement field |

#### 6.2.1.1 Dirichlet Boundary Condition Spatial Database Files

The spatial database files for the Dirichlet boundary condition specify the fixed displacements. The spatial database file may contain displacements at more degrees of freedom than those specified in the Dirichlet boundary condition settings using the **bc_dof** setting. Only those listed in **bc_dof** will be used. This permits using the same spatial database file for multiple Dirichlet boundary conditions with the same displacement field.

Table 6.2: Values in the spatial databases used for Dirichlet boundary conditions.

| Spatial database | Name in Spatial Database |
|---|---|
| db_initial | displacement-x, displacement-y, displacement-z |
| db_rate | displacement-rate-x, displacement-rate-y, displacement-rate-z, rate-start-time |
| db_change | displacement-x, displacement-y, displacement-z, change-start-time |

### 6.2.2 Neumann Boundary Conditions

Neumann boundary conditions are surface tractions applied over a subset of the mesh. As with the DirichletBoundary condition, each Neumann boundary condition can only be applied to a simply-connected surface. The surface over which the tractions are applied always has a spatial dimension that is one less than the dimension of the finite-element mesh. Traction values are computed at the integration points of each cell on the surface, using values from a spatial database. The tractions are integrated over each cell and assembled to obtain the forces applied at the vertices. See Section 7.4.7 on page 121 for a example that uses Neumann boundary conditions.

**Important**

In the small (finite) strain formulation, we assume that the normal and shear tractions are prescribed in terms of the undeformed configuration as described in section 2.5 on page 14.

The Neumann boundary condition properties and components are:

**label** Name of the group of vertices defining the mesh boundary for the Neumann boundary condition.

**up_dir** This is a 3-vector that provides a hint for the direction perpendicular to the horizontal tangent direction that is not collinear with the direction normal to the surface. The default value is (0,0,1), which assumes that the z-axis is positive upward. This vector is only needed for three-dimensional problems where the positive upward direction differs from the default.

**output** The output manager associated with diagnostic output (traction vector).

**quadrature** The quadrature object to be used for numerical integration. Since we are integrating over a surface that is one dimension lower than the problem domain, this would typically be set to something like `Quadrature2Din3D` (for a three-dimensional problem).

By default the output manager does not output any information. The specified tractions can be output in global coordinates by including "tractions" in the output manager's **cell_info_fields** array parameter.

Neumann parameters in a `cfg` file

```
pylithapp.timedependent]
bc = [x_neg, x_pos, y_neg]
bc.x_pos = pylith.bc.Neumann ; Change BC type to Neumann

[pylithapp.timedependent.bc.x_pos]
label = x_pos ; Name of group of vertices for +x boundary
db_initial = spatialdata.spatialdb.SimpleDB
db_initial.label = Neumann BC +x edge
db_initial.iohandler.filename = axialtract.spatialdb
db_initial.query_type = nearest
quadrature.cell = pylith.feassemble.FIATLagrange
quadrature.cell.dimension = 1
quadrature.cell.quad_order = 2
```

These settings correspond to the example problem described in Section 7.4.7 on page 121. It is necessary to set the boundary condition type to pylith.bc.Neumann, since the default value is DirichletBC. Constant tractions are used for this particular problem, so a quadrature order of one would have been sufficient; however, for problems involving more complex variations (e.g., a linear variation), a quadrature order of two will provide more accurate results. Note that there is no advantage to specifying an integration order higher than two, since linear elements are being used for this problem.

Table 6.3: Fields available in output of Neumann boundary condition information.

| Field Type | Field | Description |
|---|---|---|
| **cell_info_fields** | `tracton_initial` | Initial traction field in global coordinate system |
| | `traction_rate` | Rate of change of traction field in global coordinate system |
| | `rate_start_time` | Onset time in seconds for rate of change in traction field |
| | `traction_change` | Amplitude of change in traction field in global coordinate system |
| | `change_start_time` | Onset time in seconds for the amplitude change in the traction field |

#### 6.2.2.1 Neumann Boundary Condition Spatial Database Files

The spatial database files for the Neumann boundary condition specify the applied tractions. The number of traction components is equal to the spatial dimension for the problem. The tractions are specified in a local coordinate system for the boundary. The names of the components of the traction vector are:

**one-dimensional** `normal`

**two-dimensional** `shear`, `normal`

**three-dimensional** `horiz-shear`, `vert-shear`, `normal`

Ambiguities in specifying the shear tractions in 3D problems are resolved using the **up_dir** parameter. In the case of a horizontal surface, users will need to pick an alternative vector, as the default **up_dir** would coincide with the normal direction. In this case, the orientation for the `vert-shear-traction` component will correspond to whatever the user specifies for **up_dir**, rather than the actual vertical direction.

Table 6.4: Values in the spatial databases used for Dirichlet boundary conditions in three dimensions. In one- and two-dimensional problems, the names of the components are slightly different as described earlier in this section.

| Spatial database | Name in Spatial Database |
|---|---|
| **db_initial** | `traction-shear-horiz, traction-shear-vert, traction-normal` |
| **db_rate** | `traction-rate-horiz-shear, traction-rate-vert-shear, traction-rate-normal,rate-start-time` |
| **db_change** | `traction-horiz-shear, traction-vert-shear, traction-normal,change-start-time` |

### 6.2.3 Point Force Boundary Conditions

Point force boundary conditions in PyLith prescribe the application of point forces to a subset of the vertices of the finite-element mesh. While point force boundary conditions can be applied to any vertex, usually they are applied to vertices on the lateral, top, and bottom boundaries of the domain.

#### 6.2.3.1 Point Force Parameters

The properties and components for the PointForce boundary condition are:

**label** Label of the group of vertices associated with the boundary condition.

**bc_dof** Array of degrees of freedom to which forces are applied (first degree of freedom is 0).

PointForce parameters in a `cfg` file

```
[pylithapp.problem]
bc = [mybc]
bc.mybc = pylith.bc.PointForce

<h<[pylithapp.problem.bc.mybc]</h>
label = group A
bc_dof = [2] ; force in z direction
db_initial = spatialdata.spatialdb.SimpleDB
db_initial.iohandler.filename = force\_A.spatialdb
db_initial.query_type = nearest ; change query type to nearest point algorithm
db_rate = spatialdata.spatialdb.UniformDB
db_rate.values = [force-rate-z]
db_rate.data = [1.0e+5*newton/s]
```

We have created an array with one boundary condition, mybc. The group of vertices associated with the boundary condition is group A. For the database associated with the constant force, we use a SimpleDB. We set the filename and query type for the database. For the rate of change of values, we use a UniformDB and specify the rate of change in the force to be 1.0e+5 Newton/s. See Section 4.5 on page 43 for a discussion of the different types of spatial databases available.

#### 6.2.3.2 Point Force Spatial Database Files

The spatial database files for the point force boundary condition specify the forces applied.

Table 6.5: Values in the spatial databases used for point force boundary conditions.

| Spatial database | Name in Spatial Database |
|---|---|
| db_initial | force-x, force-y, force-z |
| db_rate | force-rate-x, force-rate-y, force-rate-z, rate-start-time |
| db_change | force-x, force-y, force-z, change-start-time |

## 6.3 Absorbing Boundary Conditions

This AbsorbingDampers boundary condition attempts to prevent seismic waves reflecting off of a boundary by placing simple dashpots on the boundary. Normally incident dilatational and shear waves are perfectly absorbed. Waves incident at other angles are only partially absorbed. This boundary condition is simpler than a perfectly matched layer (PML) boundary condition but does not perform quite as well, especially for surface waves. If the waves arriving at the absorbing boundary are relatively small in amplitude compared to the amplitudes of primary interest, this boundary condition gives reasonable results.

The AbsorbingDampers boundary condition properties and components are:

**label** Name of the group of vertices defining the mesh boundary for the absorbing boundary condition.

**up_dir** This is a 3-vector that provides a hint for the direction perpendicular to the horizontal tangent direction that is not collinear with the direction normal to the surface. The default value is (0,0,1), which assumes that the z-axis is positive upward. This vector is only needed for three-dimensional problems where the positive upward direction differs from the default.

**db** The spatial database specifying the material properties for the seismic velocities.

**quadrature** The quadrature object to be used for numerical integration. Since we are integrating over a surface that is one dimension lower than the problem domain, this would typically be set to something like **Quadrature2Din3D** (for a three-dimensional problem).

### 6.3.1 Finite-Element Implementation of Absorbing Boundary

Consider a plane wave propagating at a velocity $c$. We can write the displacement field as

$$\vec{u}(\vec{x},t) = \vec{u^t}(t - \frac{\vec{x}}{c}), \tag{6.2}$$

where $\vec{x}$ is position, $t$ is time, and $\vec{u^t}$ is the shape of the propagating wave. For an absorbing boundary we want the traction on the boundary to be equal to the traction associated with the wave propagating out of the domain. Starting with the expression for the traction on a boundary, $T_i = \sigma_{ij} n_j$, and using the local coordinate system for the boundary $s_h s_v n$, where $\vec{n}$ is the direction normal to the boundary, $\vec{s}_h$ is the horizontal direction tangent to the boundary, and $\vec{s}_v$ is the vertical direction tangent to the boundary, the tractions on the boundary are

$$T_{s_h} = \sigma_{s_h n} \tag{6.3}$$

$$T_{s_v} = \sigma_{s_v n} \tag{6.4}$$

$$T_n = \sigma_{nn}. \tag{6.5}$$

In the case of a horizontal boundary, we can define an auxiliary direction in order to assign unique tangential directions. For a linear elastic isotropic material, $\sigma_{ij} = \lambda\epsilon_{kk}\delta_{ij} + 2\mu\epsilon_{ij}$, and we can write the tractions as

$$T_{s_h} = 2\mu\epsilon_{s_h n} \tag{6.6}$$
$$T_{s_v} = 2\epsilon_{s_v n} \tag{6.7}$$
$$T_n = (\lambda + 2\mu)\epsilon_{nn} + \lambda(\epsilon_{s_h s_h} + \epsilon_{s_v s_v}). \tag{6.8}$$

For infinitesimal strains, $\epsilon_{ij} = \frac{1}{2}(u_{i,j} + u_{j,i})$ and we have

$$\epsilon_{s_h n} = \frac{1}{2}(u_{s_h,n} + u_{n,s_h}) \tag{6.9}$$
$$\epsilon_{s_v n} = \frac{1}{2}(u_{s_v,n} + u_{n,s_v}) \tag{6.10}$$
$$\epsilon_{nn} = u_{n,n}. \tag{6.11}$$

For our propagating plane wave, we recognize that

$$\frac{\partial \vec{u}^t(t - \frac{\vec{x}}{c})}{\partial x_i} = -\frac{1}{c}\frac{\partial \vec{u}^t(t - \frac{\vec{x}}{c})}{\partial t}, \tag{6.12}$$

so that our expressions for the tractions become

$$T_{s_h} = -\frac{\mu}{c}\left(\frac{\partial u^t_{s_h}(t - \frac{\vec{x}}{c})}{\partial t} + \frac{\partial u^t_n(t - \frac{\vec{x}}{c})}{\partial t}\right), \tag{6.13}$$
$$T_{s_v} = -\frac{\mu}{c}\left(\frac{\partial u^t_{s_v}(t - \frac{\vec{x}}{c})}{\partial t} + \frac{\partial u^t_n(t - \frac{\vec{x}}{c})}{\partial t}\right). \tag{6.14}$$

For the normal traction, consider a dilatational wave propagating normal to the boundary at speed $v_p$; in this case $u_{s_h} = u_{s_v} = 0$ and $c = v_p$. For the shear tractions, consider a shear wave propagating normal to the boundary at speed $v_s$; we can decompose this into one case where $u_n = u_{s_v} = 0$ and another case where $u_n = u_{s_h} = 0$, with $c = v_s$ in both cases. We also recognize that $\mu = \rho v_s^2$ and $\lambda + 2\mu = \rho v_p^2$. This leads to the following expressions for the tractions:

$$T_{s_h} = -\rho v_s \frac{\partial u^t_{s_h}(t - \frac{\vec{x}}{c})}{\partial t} \tag{6.15}$$
$$T_{s_v} = -\rho v_s \frac{\partial u^t_{v}(t - \frac{\vec{x}}{c})}{\partial t} \tag{6.16}$$
$$T_n = -\rho v_p \frac{\partial u^t_n(t - \frac{\vec{x}}{c})}{\partial t} \tag{6.17}$$

We write the weak form of the boundary condition as

$$\int_{S_T} T_i \phi_i \, dS = \int_{S_T} -\rho c_i \frac{\partial u_i}{\partial t} \phi_i \, dS,$$

where $c_i$ equals $v_p$ for the normal traction and $v_s$ for the shear tractions, and $\phi_i$ is our weighting function. We express the trial solution and weighting function as linear combinations of basis functions,

$$u_i = \sum_m a_i^m N^m, \tag{6.18}$$
$$\phi_i = \sum_n c_i^n N^n. \tag{6.19}$$

Substituting into our integral over the absorbing boundaries yields

$$\int_{S_T} T_i \phi_i \, dS = \int_{S_T} -\rho c_i \sum_m \dot{a}_i^m N^m \sum_n c_i^n N^n \, dS. \tag{6.20}$$

In the derivation of the governing equations, we recognized that the weighting function is arbitrary, so we form the residual by setting the terms associated with the coefficients $c_i^n$ to zero,

$$r_i^n = \sum_{\text{tract cells}} \sum_{\text{quad pts}} -\rho(x_q) c_i(x_q) \sum_m \dot{a}_i^m N^m(x_q) N^n(x_q) w_q |J_{cell}(x_q)|, \tag{6.21}$$

where $x_q$ are the coordinates of the quadrature points, $w_q$ are the weights of the quadrature points, and $|J_{cell}(x_q)|$ is the determinant of the Jacobian matrix evaluated at the quadrature points associated with mapping the reference cell to the actual cell.

The appearance of velocity in the expression for the residual means that the absorbing dampers also contribute to the system Jacobian matrix. Using the central difference method, the velocity is written in terms of the displacements,

$$\dot{u}_i(t) = \frac{1}{2\Delta t}(u_i(t+\Delta t) - u_i(t-\Delta t)). \tag{6.22}$$

Expressing the displacement at time $t+\Delta t$ in terms of the displacement at time $t$ ($u_i(t)$) and the increment in the displacement at time $t$ ($du_i(t)$) leads to

$$\dot{u}_i(t) = \frac{1}{2\Delta t}(du_i(t) + u_i(t) - u_i(t-\Delta t)) \tag{6.23}$$

The terms contributing to the system Jacobian are associated with the increment in the displacement at time $t$. Substituting into the governing equations and isolating the term associated with the increment in the displacement at time t yields

$$A_{ij}^{nm} = \sum_{\text{tract cells}} \sum_{\text{quad pts}} \delta_{ij} \frac{1}{2\Delta t} \rho(x_q) v_i(x_q) N^m(x_q) N^n(x_q) w_q |J_{cells}(x_q)|, \tag{6.24}$$

where $A_{ij}^{mn}$ is an $nd$ by $md$ matrix ($d$ is the dimension of the vector space), $m$ and $n$ refer to the basis functions and $i$ and $j$ are vector space components.

## 6.4 Fault Interface Conditions

Fault interfaces are used to create dislocations (jumps in the displacement field) in the model. The dislocations arise from slip across a fault surface. Both shear and tensile dislocations are supported. For fault interfaces, dislocations in 1D correspond to fault-opening (and closing), in 2D lateral-slip and fault opening, and in 3D lateral-slip, reverse-slip, and fault opening. PyLith supports kinematic (prescribed) slip and dynamic (spontaneous) rupture simulations.

### 6.4.1 Conventions

Slip corresponds to relative motion across a fault surface. Figure 6.1 on the facing page shows the orientation of the slip vector in 3D with respect to the fault surface and coordinate axes. PyLith automatically determines the orientation of the fault surface. This alleviates the user from having to compute the strike, dip, and rake angles over potentially complex, nonplanar fault surfaces. Instead, the user specifies fault parameters in terms of lateral motion, reverse motion, and fault opening as shown in Figure 6.2 on the next page.

### 6.4.2 Fault Implementation

In order to create relative motion across the fault surface in the finite-element mesh, additional degrees of freedom are added along with adjustment of the topology of the mesh. These additional degrees of freedom are associated with cohesive cells. These zero-volume cells allow control of the relative motion between vertices on the two sides of the fault. PyLith automatically adds cohesive cells for each fault surface. Figure 6.3 on page 94 illustrates the results of inserting cohesive cells in a mesh consisting of triangular cells. This example also shows the distinction between how buried fault edges are handled differently than fault edges that reach the edge of the domain, such as the ground surface.

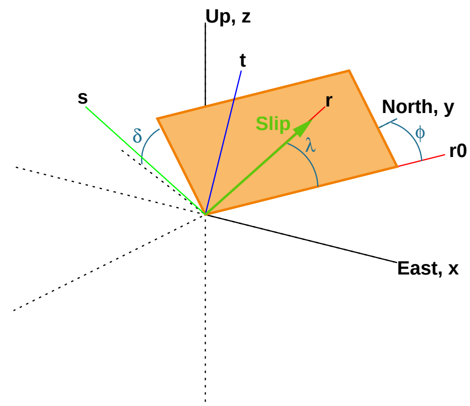



Figure 6.1: Orientation of a fault surface in 3D, where $\phi$ denotes the angle of the fault strike, $\delta$ denotes the angle of the fault dip, and $\lambda$ the rake angle.

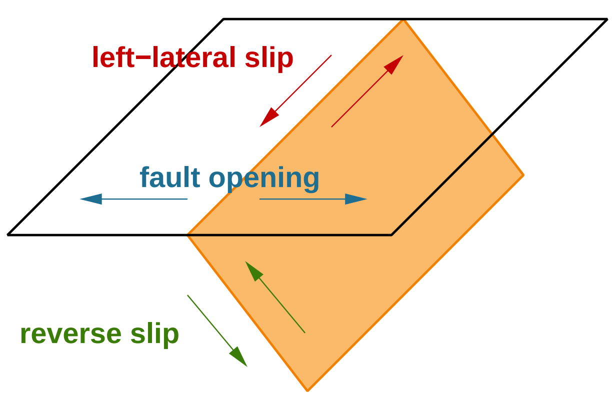



Figure 6.2: Sign conventions associated with fault slip. Positive values are associated with left-lateral, reverse, and fault opening motions.

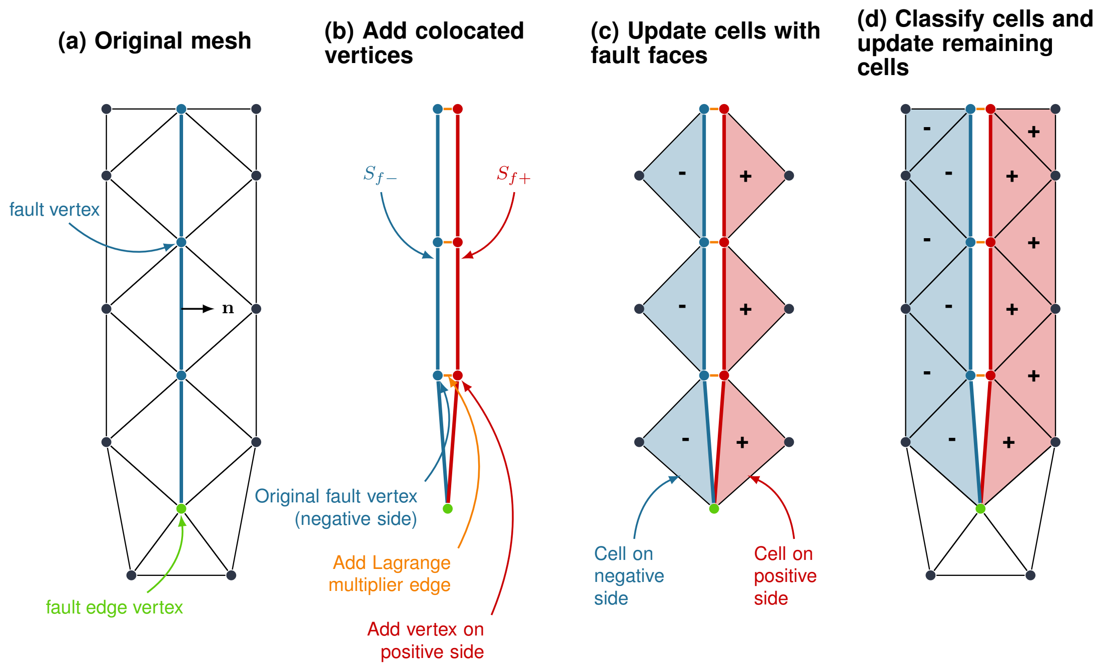



Figure 6.3: Example of cohesive cells inserted into a mesh of triangular cells. The zero thickness cohesive cells control slip on the fault via the relative motion between the vertices on the positive and negative sides of the fault.

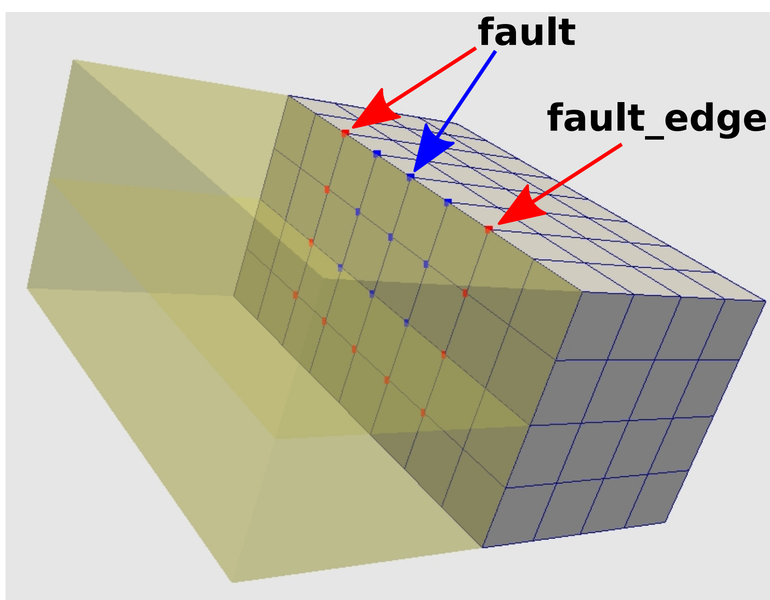



Figure 6.4: Example of how faults with buried edges must be described with two sets of vertices. All of the vertices on the fault are included in the `fault` group; the subset of vertices along the buried edges are included in the `fault_edge` group. In 2-D the fault edges are just a single vertex as shown in Figure 6.3(a).

For faults that have buried edges, splitting the mesh apart and inserting the cohesive cells becomes complex at the buried edges due to the ambiguity of defining where the fault ends and how to insert the cohesive cell. In PyLith v2.0.0 we have changed how the buried edges of the fault are managed. An additional group of fault nodes is specified (e.g., via a nodeset from CUBIT) that marks the buried edges of the fault (see Figure 6.4 on the preceding page). This allows the cohesive cell insertion algorithm to adjust the topology so that cohseive cells are inserted up to the buried edge of the fault but no additional degrees of freedom are added on the fault edge. This naturally forces slip to zero along the buried edges.

### 6.4.3 Fault Parameters

The principal parameters for fault interface conditions are:

**id** This is an integer identifier for the fault surface. It is used to specify the **material-id** of the cohesive cells in the mesh. Material identifiers must be unique across all materials and fault interfaces. Because PyLith creates the cohesive cells at runtime, there is no correspondence between the **id** property and information in the input mesh like there is for materials.

**label** Name of group of vertices associated with the fault surface. This label is also used in error and diagnostic reports.

**edge** Name of group of vertices marking the buried edges of the fault.

**up_dir** Up-dir or up direction (used in 2D and 3D simulations). In 2D the default in-plane slip is left-lateral, so we use the up-direction to resolve the ambiguity in specifying reverse slip. In 3D the up-direction is used to resolve the ambiguity in the along-strike and dip-dir directions. If the fault plane is horizontal, then the up-dir corresponds to the reverse-motion on the +z side of the fault. The only requirement for this direction is that it not be collinear with the fault normal direction. The default value of [0, 0, 1] is appropriate for most 3D problems.

**quadrature** Quadrature object used in integrating fault quantities.

**output** Manager for output of diagnostic and data fields for the fault.

By default the output manager outputs both diagnostic information (e.g., fault normal direction) and the slip at each time step. Tables 6.6 on page 98 and 6.12 on page 105 list the fields available for output for a fault with kinematic (prescribed) earthquake rupture and a fault with dynamic rupture, respectively. The fault coordinate system is shown in Figure 6.2 on page 93. The vectors in the fault coordinate system can be transformed to the global coordinate system using the direction vectors in the diagnostic output.

Fault parameters in a `cfg` file

```
[pylithapp.problem]
interfaces = [fault]

[pylithapp.problem.interfaces]
fault = pylith.faults.FaultCohesiveKin ; default
label = fault_A ; Group of vertices defining the fault surface
edge = fault_edge ; Group of vertices defining the buried edges
id = 100 ; Value for material identifier associated with fault's cohesive cells
up_dir = [0, 0, 1] ; default
quadrature.cell = pylith.feassemble.FIATLagrange
quadrature.cell.dimension = 2
```

The group of vertices has the label “fault A.” We replicate the default values for the fault “up” direction. These settings apply to a 2D fault surface embedded within a 3D mesh, so we use 2D Lagrange reference cells. The spatial database for elastic properties is used to determine the approximate shear modulus and condition the equations for faster convergence rates.

### 6.4.4 Kinematic Earthquake Rupture

Kinematic earthquake ruptures use the FaultCohesiveKin object to specify the slip as a function of time on the fault surface. Slip may evolve simultaneously over the fault surface instantaneously in a single time step (as is usually done in quasi-static simulations) or propagate over the fault surface over hundreds and up to thousands of time steps (as is usually done in a dynamic simulation).

#### 6.4.4.1 Governing Equations

The insertion of cohesive cells into the finite-element mesh has the effect of decoupling the motion of the two sides of the fault surface. In order to impose the desired relative motion, we must adjust the governing equations. PyLith employs Lagrange multiplier constraints to enforce the constraint of the relative motion in the strong sense. That is, we enforce the slip across the fault at each degree of freedom.

In conventional implementations the additional degrees of freedom associated with the Lagrange multipliers result in a complex implementation. However, the use of Lagrange multiplier constraints with cohesive cells provides for a simple formulation; we simply add the additional degrees of freedom associated with the Lagrange multipliers to the cohesive cells as shown in Figure 6.3 on page 94. As a result, the fault implementation is completely confined to the cohesive cell. Furthermore, the Lagrange multiplier constraints correspond to forces required to impose the relative motions, so they are related to the change in stress on the fault surface associated with fault slip. If we write the algebraic system of equations associated with elasticity in the form

$$\underline{A}\vec{u} = \vec{b}, \tag{6.25}$$

then adding in the Lagrange multiplier constraints associated with fault slip leads to a new system of equations of the form

$$\begin{bmatrix} \underline{A} & \underline{C}^T \\ \underline{C} & 0 \end{bmatrix} \begin{bmatrix} \vec{u} \\ \vec{l} \end{bmatrix} = \begin{bmatrix} \vec{b} \\ \vec{d} \end{bmatrix}, \tag{6.26}$$

where $\vec{l}$ is the vector of Lagrange multipliers and $\underline{C}$ is composed of rotation submatrices, $\underline{R}$, associated with the direction cosines relating the relative displacements across the fault to the vector of fault slip, $\vec{d}$. Note that by using the direction cosines to relate the relative motion across the fault, the slip vector and Lagrange multipliers (forces required to impose the slip) are in the local fault coordinate system (lateral motion, reverse motion, and fault opening).

**Non-diagonal A** The Lagrange multipliers contribute to both the system Jacobian matrix and the residual. Because we enforce the constraints in a strong sense, the terms do not involve integrals over the fault surface. The additional terms in the residual are

$$r_i^n = -C_{ji}^{pn} l_j^p, \tag{6.27}$$

$$r_i^p = d_i^p - C_{ij}^{pn} u_j^n, \tag{6.28}$$

where $n$ denotes a conventional degree of freedom and $p$ denotes a degree of freedom associated with a Lagrange multiplier. The additional terms in the system Jacobian matrix are simply the direction cosines,

$$J_{ij}^{np} = C_{ji}^{pn}, \tag{6.29}$$

$$J_{ij}^{pn} = C_{ij}^{pn}. \tag{6.30}$$

**Diagonal A** When we use a lumped system Jacobian matrix, we cannot lump the terms associated with the Lagrange multipliers. Instead, we formulate the Jacobian ignoring the contributions from the Lagrange multipliers, and then adjust the solution after the solve to account for their presence. Including the Lagrange multipliers in the general expression for the residual at time $t+\Delta t$, we have

$$r_i^n(t+\Delta t) = A_{ij}^{nm}(u_j^m(t) + du_j^m(t)) + C_{ki}^{pn}(l_k^p(t) + dl_k^p(t)), \tag{6.31}$$

where we have written the displacements and Lagrange multipliers at time $t+\Delta t$ in terms of the values at time $t$ and the increment from time $t$ to $t+\Delta t$. When we solve the lumped system ignoring the Lagrange multipliers contributions to the Jacobian, we formulate the residual assuming the values $du_i^n$(t) and $dl_k^p(t)$ are zero. So our task is to determine the increment in the Lagrange multiplier, $dl_k^p$, and the correction to the displacement increment, $du_i^n$, and by setting the residual with all terms included to zero; thus, we have

$$A_{ij}^{nm}(u_j^m(t)+du_j^m(t))+C_{ki}^{pn}(l_k^p(t)+dl_k^p(t))=0 \text{ subject to} \tag{6.32}$$

$$C_{ij}^{pn}(u_j^n(t)+du_j^n(t))=d_i^p. \tag{6.33}$$

Making use of the residual computed with $du_i^n(t)=0$ and $dl_k^p(t)=0$,

$$r_i^n+A_{ij}^{nm}du_j^m+C_{ki}^{pn}dl_k^p=0 \text{ subject to} \tag{6.34}$$

$$C_{ij}^{pn}(u_j^n(t)+du_j^n(t))=d_i^p. \tag{6.35}$$

Explicitly writing the equations for the vertices on the negative and positive sides of the fault yields

$$r_i^{n-}+A_{ij}^{nm-}du_j^{m-}+R_{ki}^{pn}dl_k^p=0, \tag{6.36}$$

$$r_i^{n+}+A_{ij}^{nm+}du_j^{m+}+R_{ki}^{pn}dl_k^p=0, \tag{6.37}$$

$$R_{ij}^{pn}(u_j^{n+}+du_j^{n+}-u_j^{n-}-du_j^{n-})=d_i^p. \tag{6.38}$$

Solving the first two equations for $du_j^{m-}$ and $du_j^{m+}$ and combining them using the third equation leads to

$$R_{ij}^{pn}\left((A_{ij}^{nm+})^{-1}+(A_{ij}^{nm+})^{-1}\right)R_{ki}^{pn}dl_k^p=d_i^p-R_{ij}^{pn}(u_j^{n+}-u_j^{n-}) \\ +R_{ij}^{pn}\left((A_{ij}^{nm+})^{-1}r_i^{n+}-(A_{ij}^{nm-})^{-1}r_i^{n-}\right). \tag{6.39}$$

We do not allow overlap between the fault interface and the absorbing boundary, so $A_{ij}^{nm}$ is the same for all components at a vertex. As a result the matrix on the left hand side simplifies to

$$S_{ik}^{pn}=\delta_{ik}\left(\frac{1}{A^{nm+}}+\frac{1}{A^{nm-}}\right), \tag{6.40}$$

and

$$dl_k^p=(S_{ik}^{pn})^{-1}\left(d_i^p-R_{ij}^{pn}(u_j^{n+}-u_j^{n-})+R_{ij}^{pn}\left((A_{ij}^{nm+})^{-1}r_i^{n+}-(A_{ij}^{nm-})^{-1}r_i^{n-}\right)\right). \tag{6.41}$$

Now that we know the value of the increment in the Lagrange multiplier from time $t$ to time $t+\Delta t$, we can correct the value for the displacement increment from time $t$ to $t+\Delta t$ using

$$\Delta du_j^{n-}=(A_{ij}^{nm-})^{-1}C_{ki}^{pn}dl_k^p \text{ and} \tag{6.42}$$

$$\Delta du_j^{n+}=-(A_{ij}^{nm+})^{-1}C_{ki}^{pn}dl_k^p. \tag{6.43}$$

#### 6.4.4.2 Arrays of Kinematic Rupture Components

Multiple earthquake ruptures can be specified on a single fault surface. This permits repeatedly rupturing the same portion of a fault or combining earthquake rupture on one subset of the fault surface with steady aseismic slip on another subset (the two subsets may overlap in both time and space). A dynamic array of kinematic earthquake rupture components associates a name (string) with each kinematic rupture. The default dynamic array contains a single earthquake rupture, "rupture". The **eq_srcs** is the FaultCohesiveKin facility for this dynamic array.

Array of kinematic rupture components in a `cfg` file

```
[pylithapp.problem.interfaces.fault]
eq_srcs = [earthquake,creep]
```

The output manager includes generic fault information (orientation) as well as the final slip or slip rate (as in the case of the constant slip rate slip time function) and slip initiation time for each kinematic rupture. The name of the slip and slip initiation time vertex fields are of the form `final_slip_NAME` and `slip_time_NAME`, respectively, where `NAME` refers to the name used in the dynamic array of kinematic ruptures, **eq_srcs**.

Table 6.6: Fields available in output of fault information.

| Field Type | Field | Description |
|---|---|---|
| **vertex_info_fields** | normal_dir | Direction of fault normal in global coordinate system |
| | strike_dir | Direction of fault strike in global coordinate system |
| | dip_dir | Up-dip direction on hanging wall in global coordinate system |
| | final_slip_NAME | Vector of final slip (in fault coordinate system) in meters |
| | slip_time_NAME | Time at which slip begins in seconds |
| **vertex_data_fields** | slip | Slip vector at time step (in fault coordinate system) in meters |
| | traction_change | Change in fault tractions (in fault coordinate system) in Pa |

#### 6.4.4.3 Kinematic Rupture Parameters

The kinematic rupture parameters include the origin time and slip time function. The slip initiation time in the slip time function is relative to the origin time (default is 0). This means that slip initiates at a point at a time corresponding to the sum of the kinematic rupture's origin time and the slip initiation time for that point.

FaultCohesiveKin parameters in a `cfg` file

```
[pylithapp.problem.interfaces.fault]

eq_srcs = [earthquake,creep]

[pylithapp.problem.interfaces.fault.eq_srcs.earthquake]
origin_time = 0.0*s ; default origin time
slip_function = pylith.faults.StepSlipFn ; default slip time function

[pylithapp.problem.interfaces.fault.eq_srcs.creep]
origin_time = 10.0*year ; start creep at 10.0 years

slip_function = pylith.faults.ConstRateSlipFn ; switch to constant slip rate slip function
```

#### 6.4.4.4 Slip Time Function

The current release of PyLith supports specification of the evolution of fault slip using analytical expressions for the slip time history at each point, where the parameters for the slip time function may vary over the fault surface. Currently, three slip time functions are available: (1) a step-function for quasi-static modeling of earthquake rupture, (2) a constant slip rate time function for modeling steady aseismic slip, and (3) the integral of Brune's far-field time function [Brune, 1970] for modeling the dynamics of earthquake rupture. Additional slip time functions will likely be available in future releases. The default slip time function is the step-function slip function.

**Step-Function Slip Time Function** This slip function prescribes a step in slip at a given time at a point:

$$D(t) = \begin{cases} 0 & 0 \leq t < t_r \\ D_{final} & t \geq t_r \end{cases}, \tag{6.44}$$

where $D(t)$ is slip at time $t$, $D_{final}$ is the final slip, and $t_r$ is the slip initiation time (time when rupture reaches the location). The slip is specified independently for each of the components of slip, and the slip and slip starting time may vary over the fault surface.

**final_slip** Spatial database of slip ($D_{final}$).

**slip_time** Spatial database of slip initiation times ($t_r$).

StepSlipFn parameters in a `cfg` file

```
[pylithapp.problem.interfaces.fault.eq_srcs.rupture]
slip_function = pylith.faults.StepSlipFn
```

```
[pylithapp.problem.interfaces.fault.eq_srcs.rupture.slip_function]
final_slip.iohandler.filename = final_slip.spatialdb
slip_time.iohandler.filename = sliptime.spatialdb
```

The spatial database files for the slip time function specify the spatial variation in the parameters for the slip time function, as shown in Table 6.7.

Table 6.7: Values in spatial database used as parameters in the step function slip time function.

| Spatial database | Value | Description |
|---|---|---|
| **final_slip** | left-lateral-slip | Amount of left-lateral final slip in meters. Use negative values for right-lateral slip. |
| | reverse-slip | Amount of reverse slip in meters. Use negative values for normal slip. |
| | fault-opening | Amount of fault opening in meters. Negative values imply penetration. |
| **slip_time** | slip-time | Slip initiation time ($t_t$) in seconds. |

**Constant Slip Rate Slip Time Function** This slip function prescribes a constant slip rate for the evolution of slip at a point:

$$D(t) = \left\{ \begin{array}{cc} 0 & 0 \le t < t_r \\ V(t - t_r) & t \ge t_r \end{array} \right. , \tag{6.45}$$

where $D(t)$ is slip at time $t$, $V$ is the slip rate, and $t_r$ is the slip initiation time (time when rupture reaches the location). The slip rate is specified independently for each of the components of slip, and the slip rate and slip starting time may vary over the fault surface.

**slip_rate** Spatial database of slip rate ($V$).

**slip_time** Spatial database of slip initiation times ($t_r$).

ConstRateSlipFn parameters in a cfg file

```
[pylithapp.problem.interfaces.fault.eq_srcs.ruptures]
slip_function = pylith.faults.ConstRateSlipFn

[pylithapp.problem.interfaces.fault.eq_srcs.ruptures.slip_function]
slip_rate.iohandler.filename = slip_rate.spatialdb
slip_time.iohandler.filename = sliptime.spatialdb
```

The spatial database files for the slip time function specify the spatial variation in the parameters for the slip time function, as shown in Table 6.8.

Table 6.8: Values in spatial database used as parameters in the constant slip rate slip time function.

| Spatial database | Value | Description |
|---|---|---|
| **slip_rate** | left-lateral-slip | Slip rate for left-lateral final slip in meters per second. Use negative values for right-lateral slip. |
| | reverse-slip | Slip rate for reverse slip in meters per second. Use negative values for normal slip. |
| | fault-opening | Slip rate for fault opening in meters per second. Negative values imply penetration. |
| **slip_time** | slip-time | Slip initiation time ($t_t$) in seconds. |

**Brune Slip Time Function** We use an integral of Brune's far-field time function [Brune, 1970] to describe the evolution in time of slip at a point:

$$D(t) = \begin{cases} 0 & 0 \leq t < t_r \\ D_{final}\left(1 - exp\left(-\frac{t-t_r}{t_0}\right)\left(1 + \frac{t-t_r}{t_0}\right)\right) & t \geq t_r \end{cases}, \tag{6.46}$$

$$t_0 = 0.6195 t_{rise}, \tag{6.47}$$

where $D(t)$ is slip at time $t$, $D_{final}$ is the final slip at the location, $t_r$ is the slip initiation time (time when rupture reaches the location), and $t_{rise}$ is the rise time.

**slip** Spatial database of final slip distribution ($D_{final}$).

**slip_time** Spatial database of slip initiation times ($t_r$).

**rise_time** Spatial database for rise time ($t_{rise}$).

BruneSlipFn parameters in a cfg file

```
[pylithapp.problem.interfaces.fault.eq_srcs.ruptures]
slip_function = pylith.faults.BruneSlipFn

[pylithapp.problem.interfaces.fault.eq_srcs.rupture.slip_function]
slip.iohandler.filename = finalslip.spatialdb
rise_time.iohandler.filename = risetime.spatialdb
slip_time.iohandler.filename = sliptime.spatialdb
```

The spatial database files for the slip time function specify the spatial variation in the parameters for the slip time function, as shown in Table 6.9.

Table 6.9: Values in spatial database used as parameters in the Brune slip time function.

| Spatial database | Value | Description |
|---|---|---|
| **slip** | left-lateral-slip | Amount of left-lateral final slip in meters. Use negative values for right-lateral slip. |
| | reverse-slip | Amount of reverse slip in meters. Use negative values for normal slip. |
| | fault-opening | Amount of fault opening in meters. Negative values imply penetration. |
| **rise_time** | rise-time | Rise time ($t_r$) in seconds. |
| **slip_time** | slip-time | Slip initiation time ($t_t$) in meters. |

**Liu-Cosine Slip Time Function** This slip time function, proposed by Liu, Archuleta, and Hartzell for use in ground-motion modeling[Liu et al., 2006], combines several cosine and sine functions together to create a slip time history with a sharp rise and gradual termination with a finite duration of slip. The evolution of slip at a point follows:

$$D(t) = \begin{cases} D_{final}C_n\left(0.7t - 0.7\frac{t_1}{\pi}\sin\frac{\pi t}{t_1} - 1.2\frac{t_1}{\pi}\left(\cos\frac{\pi t}{2t_1} - 1\right)\right) & 0 \leq t < t_1 \\ D_{final}C_n\left(1.0t - 0.7\frac{t1}{\pi}\sin\frac{\pi t}{t_1} + 0.3\frac{t2}{\pi}\sin\frac{\pi(t-t1)}{t_2} + \frac{1.2}{\pi}t_1 - 0.3t_1\right) & t_1 \leq t < 2t_1 \\ D_{final}C_n\left(0.7 - 0.7\cos\frac{\pi t}{t_1} + 0.6\sin\frac{\pi t}{2t_1}\right) & 2t_1 \leq t \leq t_0 \end{cases}, \tag{6.48}$$

$$C_n = \frac{\pi}{1.4\pi t_1 + 1.2 t_1 + 0.3\pi t_2}, \tag{6.49}$$

$$t_0 = 1.525 t_{rise}, \tag{6.50}$$

$$t_1 = 0.13 t_0, \tag{6.51}$$

$$t_2 = t_0 - t_1, \tag{6.52}$$

where $D(t)$ is slip at time $t$, $D_{final}$ is the final slip at the location, $t_r$ is the slip initiation time (time when rupture reaches the location), and $t_{rise}$ is the rise time.

**slip** Spatial database of final slip distribution ($D_{final}$).

**slip_time** Spatial database of slip initiation times ($t_r$).

**rise_time** Spatial database for rise time ($t_{rise}$).

The spatial database files for the slip time function use the same parameters for the slip time function as the Brune slip time function shown in Table 6.9 on the preceding page.

**Time-History Slip Time Function** This slip time function reads the slip time function from a data file, so it can have an arbitrary shape. The slip and slip initiation times are specified using spatial databases, so the slip time function, in general, will use a normalized amplitude.

**slip** Spatial database of final slip distribution ($D_{final}$).

**slip_time** Spatial database of slip initiation times ($t_r$).

**time_history** Temporal database for slip evolution.

TimeHistorySlipFn parameters in a cfg file

```
[pylithapp.problem.interfaces.fault.eq_srcs.ruptures]
slip_function = pylith.faults.TimeHistorySlipFn

[pylithapp.problem.interfaces.fault.eq_srcs.rupture.slip_function]
slip.iohandler.filename = finalslip.spatialdb
slip_time.iohandler.filename = sliptime.spatialdb
time_history.iohandler.filename = myfunction.timedb
```

The spatial database files for the slip time function specify the spatial variation in the parameters for the slip time function, as shown in Table 6.10.

Table 6.10: Values in spatial database used as parameters in the time history slip time function.

| Spatial database | Value | Description |
|---|---|---|
| **slip** | left-lateral-slip | Amount of left-lateral final slip in meters. Use negative values for right-lateral slip. |
| | reverse-slip | Amount of reverse slip in meters. Use negative values for normal slip. |
| | fault-opening | Amount of fault opening in meters. Negative values imply penetration. |
| **rise_time** | rise-time | Rise time ($t_r$) in seconds. |
| **slip_time** | slip-time | Slip initiation time ($t_t$) in meters. |

### 6.4.5 Dynamic Earthquake Rupture

Dynamic fault interfaces use the FaultCohesiveDyn object to specify a fault constitutive model to govern the fault tractions (friction) and the resulting slip. When friction is large enough such that there is no sliding on the fault, the fault is locked (slip is zero) and the Lagrange multipliers assume their values just as they do in kinematic ruptures. In this case, the Lagrange multipliers correspond to the forces necessary to keep the slip zero. When the driving forces exceed those allowed by friction, we reduce the values of the Lagrange multipliers to those consistent with friction from the fault constitutive model. When we reduce the Lagrange multipliers, we must increment the slip accordingly to maintain consistency in the algebraic system of equations.

#### 6.4.5.1 Governing Equations

The algebraic systems of equations for dynamic earthquake rupture are the same as those for kinematic rupture

$$\left[\begin{array}{cc} \underline{A} & \underline{C}^T \\ \underline{C} & 0 \end{array}\right] \left[\begin{array}{c} \vec{u} \\ \vec{l} \end{array}\right] = \left[\begin{array}{c} \vec{b} \\ \vec{d} \end{array}\right]. \tag{6.53}$$

Enforcing the limits imposed on the Lagrange multipliers by the fault constitutive model requires determining the increment in slip for an increment in the Lagrange multipliers. The increment in the Lagrange multipliers is the difference between the value computed for the current slip (either zero or the slip at the previous time step) and the value computed from the fault constitutive model. Starting from our system of algebraic equations,

$$A_{ij}^{nm} u_j^m + C_{ji}^{pn} l_j^p = b_i^n, \tag{6.54}$$

we compute the sensitivity for the given loading and boundary conditions,

$$A_{ij}^{nm} \partial u_j^m = -C_{ji}^{pn} \partial l_j^p. \tag{6.55}$$

Computing the increment in the slip requires computing the increment in the displacements. Solving this equation rigorously would require inverting the system Jacobian, which we do not want to do unless it is diagonal (as it is in the case of the lumped formulations).

**Non-Diagonal A** In general A is a sparse matrix with off-diagonal terms of the form

$$A = \left( \begin{array}{ccc} A_0 & A_1 & A_2 \\ A_3 & A_{n-} & 0 \\ A_4 & 0 & A_{n+} \end{array} \right), \tag{6.56}$$

where the degrees of freedom on either side of the fault are uncoupled. We formulate two small linear systems involving just the degrees of freedom associated with vertices on either the positive or negative sides of the fault,

$$A_{ij}^{nm-} \partial u_j^{m-} = -R_{ij}^{pn} \partial l_j^p, \tag{6.57}$$

$$A_{ij}^{nm+} \partial u_j^{m+} = R_{ij}^{pn} \partial l_j^p, \tag{6.58}$$

where we have replaced $\underline{C}$ with $\underline{R}$ to denote the explicit inclusion of the signs for the terms in $\underline{C}$ associated with the positive ($n^+$) and negative ($n^-$) sides of the fault. After solving these two linear systems of equations, we compute the increment in slip using

$$\partial d_i^p = R_{ij}^{pn} (\partial u_j^{n+} - \partial u_j^{n-}). \tag{6.59}$$

The solution of these two linear systems gives the increment in slip assuming all the degrees of freedom except those immediately adjacent to the fault remain fixed. In real applications where the deformation associated with fault slip is localized around the fault, this provides good enough approximations so that the nonlinear solver converges quickly. In problems where deformation associated with slip on the fault is not localized (as in the case in some of the example problems), the increment in slip computed by solving these two linear systems is not a good approximation and the nonlinear solve requires a large number of iterations.

We use the PETSc Krylov subspace solver (KSP) to solve these two linear systems. The PETSc settings for the KSP object are set in the same manner as the main solver, except we use the prefix `friction_` in all of the settings related to the KSP solver for these two linear systems. For example, to use the recommended additive Schwarz preconditioner in the friction sensitivity solves, the settings in a `cfg` file are:

```
[pylithapp.petsc]
friction_pc_type = asm
```

See the examples in Sections 7.9.7 on page 158 and 7.13 on page 187 for details.

**Diagonal A** With a lumped Jacobian matrix, we can solve for the increment in slip directly,

$$\partial d_i^p = -C_{ij}^{pn}(A_{jk}^{nm})^{-1} C_{lk}^{pm} \partial l_l^p. \tag{6.60}$$

By not allowing the fault interface to overlap with the absorbing boundary, the terms in $A$ for a given vertex are identical and the expression on the right-hand side reduces to

$$\partial d_i^p = -\left(\frac{1}{A^{n+}} + \frac{1}{A^{n-}}\right) \partial l_i^p. \tag{6.61}$$

#### 6.4.5.2 Dynamic Rupture Parameters

The properties and facilities of the FaultCohesiveDyn object include

**open_free_surface** If true, enforce traction free surface when the fault opens, otherwise apply prescribed tractions even when the fault opens (default is true); to mimic a dike opening, use false.

**zero_tolerance** Tolerance for detecting zero values (default is 1.0e-10); should be larger than absolute tolerance in KSP solves.

**zero_tolerance_normal** Tolerance for suppressing near zero fault opening values (default is 1.0e-10); should be larger than absolute tolerance in KSP solves.

**traction_perturbation** Prescribed tractions on fault surface (generally used for nucleating earthquake ruptures; default is none).

**friction** Fault constitutive model.

FaultCohesiveDyn parameters in a `cfg` file

```
[pylithapp.problem.interfaces.fault]
open_free_surface = True ; default

traction_perturbation = pylith.faults.TractPerturbation ; not default
traction_perturbation.db_initial = spatialdata.spatialdb.SimpleDB
traction_perturbation.db_initial.iohandler.filename = tractions.spatialdb

friction<f> = pylith.friction.StaticFriction
<f>friction.db_properties = spatialdata.spatialdb.SimpleDB
friction.db_properties.iohandler.filename = friction.spatialdb
```

**Warning**

Use of the dynamic rupture implementation in a quasi-static simulations requires use of the nonlinear solver.

**Important**

The dynamic rupture implementation requires careful selection of linear and nonlinear solver tolerances. A key issue is making sure the linear solver toleance is tighter (smaller) than the tolerance used to detect slip (fault **zero_toelerance** and **zero_toelerance_normal**). As a result, the linear and solver absolute tolerances should be used to for convergence, not the relative tolerances. The settings below illustrates the relevant parameters and example values. The values can be scaled to change the overall desired tolerances. The separate tolerance for near zero values of fault opening was added in v2.2.1. This provides the solver greater flexibility to prevent fault opening with nonplanar faults.

Sample tolerance settings for fault friction

```
[pylithapp.problem.interfaces.fault]
zero_tolerance = 1.0e-11
zero_tolerance_normal = 2.0e-11

[pylithapp.petsc]
# Linear solver tolerances
ksp_rtol = 1.0e-20
ksp_atol = 1.0e-12

# Nonlinear solver tolerances
snes_rtol = 1.0e-20
snes_atol = 1.0e-10

# Set preconditioner for friction sensitivity solve
friction_pc_type = asm
friction_sub_pc_factor_shift_type<p> = nonzero
```

The prescribed traction perturbation is specified using the same fault coordinate system as the slip directions in the kinematic ruptures. The perurbation has the same functional form as the time-dependent boundary conditions (and same spatial databases). Table 6.11 gives the values in the spatial database for the prescribed tractions. Table 6.12 on the facing page shows the fields available for output. Additional fields are available depending on the fault constitutive model.

Table 6.11: Values in spatial databases for prescribed tractions.

| Spatial database | Dimension | Value | Description |
|---|---|---|---|
| **db_initial** | 2D | traction-shear | Left-lateral shear traction (reverse shear for dipping faults) |
| | | traction-normal | Normal traction (tension is positive) |
| | 3D | traction-shear-leftlateral | Left-lateral shear traction |
| | | traction-shear-updip | Reverse shear traction |
| | | traction-normal | Normal traction (tension is positive) |
| **db_rate** | 2D | traction-rate-shear | Rate of change of left-lateral shear traction (reverse shear for dipping faults) |
| | | traction-rate-normal | Rate of change of normal traction (tension is positive) |
| | 3D | traction-rate-leftlateral | Rate of change of left-lateral shear traction |
| | | traction-rate-shear-updip | Rate of change of reverse shear traction |
| | | traction-rate-normal | Rate of change of normal traction (tension is positive) |
| | all | rate-start-time | Time at which rate of change begins |
| **db_change** | 2D | traction-shear | Change in left-lateral shear traction (reverse shear for dipping faults) |
| | | traction-normal | Change in normal traction (tension is positive) |
| | 3D | traction-leftlateral | Change in left-lateral shear traction |
| | | traction-shear-updip | Change in reverse shear traction |
| | | traction-normal | Change in normal traction (tension is positive) |
| | all | change-start-time | Time at which change begins |
| **th_change** | all | None | Time history for change |

Table 6.12: Fields available in output of fault information.

| Field Type | Field | Description |
|---|---|---|
| **vertex_info_fields** | normal_dir | Direction of fault normal in global coordinate system |
| | strike_dir | Direction of fault strike in global coordinate system |
| | dip_dir | Up-dip direction on hanging wall in global coordinate system |
| | traction_initial | Initial tractions (if specified) in fault coordinate system |
| | traction_rate | Rate of change in tractions (if specified) in fault coordinate system |
| | rate_start_time | Time at which rate of change begins (if specified) |
| | traction_change | Change in tractions (if specified) in fault coordinate system |
| | change_start_time | Time at which change occurs (if specified) |
| **vertex_data_fields** | slip | Slip vector at time step (in fault coordinate system) in meters |
| | traction | Fault tractions (in fault coordinate system) in Pa |

#### 6.4.5.3 Fault Constitutive Models

PyLith provides four fault constitutive models. Future releases may contain additional models, and a template is provided for you to construct your own (see Section 9.3 on page 256). The fault constitutive model implementations are independent of dimension and work in both 2D and 3D. In solving the governing equations, PyLith will use a scalar representation of the shear traction in 2D and a vector representation of the shear traction in 3D, with the shear traction resolved in the direction of current slip. The fault constitutive models contain a common set of properties and components:

**label** Name of the friction model.

**db_properties** Spatial database of the friction model parameters (default is SimpleDB).

**db_initial_state** Spatial database for initial state variables. A warning will be given when a spatial database for the initial state is not specified. The default is none which results in initial state values of 0.0. For some friction models, we provide more meaningful values for default values.

**Static Friction** The static friction model produces shear tractions proportional to the fault normal traction plus a cohesive stress,

$$T_f = \begin{cases} T_c - \mu_f T_n & T_n \le 0 \\ 0 & T_n > 0 \end{cases}. \tag{6.62}$$

The spatial database file for the static friction model properties specifies the spatial variation of the parameters given in Table 6.13.

Table 6.13: Values in the spatial database for constant friction parameters.

| Value | Description |
|---|---|
| friction-coefficient | Coefficient of friction, $\mu_f$ |
| cohesion | Cohesive stress, $T_c$ |

**Slip-Weakening Friction** The linear slip-weakening friction model produces shear tractions equal to the cohesive stress plus a contribution proportional to the fault normal traction that decreases from a static value to a dynamic value as slip progresses,

$$T_f = \begin{cases} T_c - (\mu_s - (\mu_s - \mu_d)\frac{d}{d_0})T_n & d \le d_0 \text{ and } T_n \le 0 \\ T_c - \mu_d T_n & d > d_0 \text{ and } T_n \le 0 \\ 0 & T_n > 0 \end{cases} \tag{6.63}$$

The spatial database files for the slip-weakening friction model properties and state variables specify the spatial variation of the fault constitutive model parameters given in Table 6.14 on the next page. As long as the fault is locked, the initial state variables are zero, so specifying the initial state variables for slip-weakening friction is rare. The slip-weakening friction also includes a parameter, **force_healing**, to control healing. In quasi-static simulations, one usually wants slip confined to a single time step (**force_healing** = True), whereas in a dynamic simulation slip occurs over many time steps (**force_healing** = False; default behavior) and fault healing is often neglected. The properties include:

**force_healing** Flag indicating whether healing (cumalative slip state variable reset to zero) is forced after every time step.

SlipWeakening parameters in a `cfg` file

```
[pylithapp.problem.interfaces.fault]
friction = pylith.friction.SlipWeakening ; Change from the default

friction.force_healing = False ; default value
```

Table 6.14: Values in spatial databases for slip-weakening friction.

| Spatial database | Value | Description |
|---|---|---|
| **db_properties** | `static-coefficient` | Static coefficient of friction, $\mu_s$ |
| | `dynamic-coefficient` | Dynamic coefficient of friction, $\mu_d$ |
| | `slip-weakening-parameter` | Slip-weakening parameter, $d_0$ |
| | `cohesion` | Cohesive stress, $T_c$ |
| **db_initial_state** | `cumulative-slip` | Cumulative slip, $d$ |
| | `previous-slip` | Slip at previous time step, $d(t-\Delta t)$ |

**Time-Weakening Friction** The linear time-weakening friction model is analogous to the linear slip-weakening friction model with time replacing slip. It produces shear tractions equal to the cohesive stress plus a contribution proportional to the fault normal traction that decreases from a static value to a dynamic value as time progresses,

$$T_f = \begin{cases} T_c - (\mu_s - (\mu_s - \mu_d)\frac{t}{t_0})T_n & t \leq t_0 \text{ and } T_n \leq 0 \\ T_c - \mu_d T_n & t > t_0 \text{ and } T_n \leq 0 \\ 0 & T_n > 0 \end{cases} \tag{6.64}$$

The spatial database files for the time-weakening friction model properties and state variables specify the spatial variation of the fault constitutive model parameters given in Table 6.15. As long as the fault is locked, the initial state variable is zero, so specifying the initial state variable for time-weakening friction is rare.

Table 6.15: Values in spatial databases for time-weakening friction.

| Database | Value | Description |
|---|---|---|
| **db_properties** | `static-coefficient` | Static coefficient of friction, $\mu_s$ |
| | `dynamic-coefficient` | Dynamic coefficient of friction, $\mu_d$ |
| | `time-weakening-parameter` | Time-weakening parameter, $t_0$ |
| | `cohesion` | Cohesive stress, $T_c$ |
| **db_initial_state** | `elapsed-time` | Elasped time of slip, $t$ |

**Slip- and Time-Weakening Friction I** This friction model, used in a few SCEC Spontaneous Rupture benchmarks, combines characteristics of slip-weakening and time-weakening friction. The time-weakening portion is generally used to force nucleation of the rupture. The model produces shear tractions equal to the cohesive stress plus a contribution proportional to the fault normal traction that decreases from a static value to a dynamic value as slip progresses or when a weakening time is reached,

$$T_f = \begin{cases} T_c - (\mu_s - (\mu_s - \mu_d)\frac{d}{d_0})T_n & d \leq d_0 \text{ and } t < t_w \text{ and } T_n \leq 0 \\ T_c - \mu_d T_n & (d > d_0 \text{ or } t \geq t_w) \text{ and } T_n \leq 0 \\ 0 & T_n > 0 \end{cases} \tag{6.65}$$

The spatial database files for the slip- and time-weakening friction model properties and state variables specify the spatial variation of the fault constitutive model parameters given in Table 6.16 on the next page. As long as the fault is locked, the initial state variables are zero, so specifying the initial state variables for slip-weakening friction is rare. This variation of slip-weakening friction does not include the `force_healing` parameter, because this friction model was developed for dynamic simulations.

SlipWeakeningTime parameters in a `cfg` file

```
[pylithapp.problem.interfaces.fault]
friction = pylith.friction.SlipWeakeningTime ; Change from the default
```

Table 6.16: Values in spatial databases for a simple slip- and time-weakening friction model.

| Spatial database | Value | Description |
|---|---|---|
| **db_properties** | static-coefficient | Static coefficient of friction, $\mu_s$ |
| | dynamic-coefficient | Dynamic coefficient of friction, $\mu_d$ |
| | slip-weakening-parameter | Slip-weakening parameter, $d_0$ |
| | weakening-time | Weakening time, $t_w$ |
| | cohesion | Cohesive stress, $T_c$ |
| **db_initial_state** | cumulative-slip | Cumulative slip, $d$ |
| | previous-slip | Slip at previous time step, $d(t-\Delta t)$ |

**Slip- and Time-Weakening Friction II** This friction model, used in a few SCEC Spontaneous Rupture benchmarks, merges features of slip-weakening and time-weakening to provide a more numerically stable version of the Slip- and Time-Weakening Friction I model. Rather than an instantaneous drop in the coefficient of friction from the static value to the dynamic value when the weakening time is reached, the weakening progresses linearly with time. As in the other slip- and time-weakening friction model, the time-weakening portion is generally used to force nucleation of the rupture. The model produces shear tractions equal to the cohesive stress plus a contribution proportional to the fault normal traction that decreases from a static value to a dynamic value as slip and time progress,

$$T_f = \begin{cases} T_c - (\mu_s - (\mu_s - \mu_d) max(f_1, f_2)) T_n & T_n \leq 0 \\ 0 & T_n > 0 \end{cases} \tag{6.66}$$

$$f_1 = \begin{cases} d/d_0 & d \leq d_0 \\ 1 & d \geq d_0 \end{cases} \tag{6.67}$$

$$f_2 = \begin{cases} 0 & t \leq t_w \\ (t - t_w)/t_0 & t_w < t \leq t_w + t_0 \\ 1 & t > t_w + t_0 \end{cases} \tag{6.68}$$

The spatial database files for the slip- and time-weakening friction model properties and state variables specify the spatial variation of the fault constitutive model parameters given in Table 6.17 on the following page. As long as the fault is locked, the initial state variables are zero, so specifying the initial state variables for slip-weakening friction is rare. This variation of slip-weakening friction does not include the `force_healing` parameter, because this friction model was developed for dynamic simulations.

SlipWeakeningTimeStable parameters in a `cfg` file

```
[pylithapp.problem.interfaces.fault]
friction = pylith.friction.SlipWeakeningTimeStable ; Change from the default
```

Table 6.17: Values in spatial databases for a second slip- and time-weakening friction model.

| Spatial database | Value | Description |
|---|---|---|
| `db_properties` | `static-coefficient` | Static coefficient of friction, $\mu_s$ |
| | `dynamic-coefficient` | Dynamic coefficient of friction, $\mu_d$ |
| | `slip-weakening-parameter` | Slip-weakening parameter, $d_0$ |
| | `time-weakening-time` | Weakening time, $t_w$ |
| | `time-weakening-parameter` | Time-weakening parameter, $t_0$ |
| | `cohesion` | Cohesive stress, $T_c$ |
| `db_initial_state` | `cumulative-slip` | Cumulative slip, $d$ |
| | `previous-slip` | Slip at previous time step, $d(t-\Delta t)$ |

**Rate- and State-Friction with Ageing Law** The Dieterich-Ruina rate and state friction model produces shear tractions equal to the cohesive stress plus a contribution proportional to the fault normal traction that depends on a state variable,

$$T_f = \begin{cases} T_c - \mu_f T_n & T_n \le 0 \\ 0 & T_n > 0 \end{cases} \tag{6.69}$$

$$\mu_f = \begin{cases} \mu_0 + a\ln\left(\frac{V}{V_0}\right) + b\ln\left(\frac{V_0\theta}{L}\right) & V \ge V_{linear} \\ \mu_0 + a\ln\left(\frac{V_{linear}}{V_0}\right) + b\ln\left(\frac{V_0\theta}{L}\right) - a\left(1 - \frac{V}{V_{linear}}\right) & V < V_{linear} \end{cases} \tag{6.70}$$

$$\frac{d\theta}{dt} = 1 - \frac{V\theta}{L} \tag{6.71}$$

where $V$ is slip rate, $V_{linear}$ is a cutoff for a linear slip rate dependence, $a$ and $b$ are coefficients, $L$ is the characteristic slip distance, $\theta$ is a state variable. With an interative solver in quasi-static simulations with its small, but nonzero residual tolerance we never encounter zero slip rates in quasi-static simulations. Instead we want to avoid significant variations in the coefficient of friction for slip rates on the same order as our residual tolerance. We regularize the rate and state friction model by imposing a linearization of the variation of the coefficient of friction with slip rate when the slip rate drops below a cutoff slip rate, $V_{linear}$ (**`linear_slip_rate`** property with a default value of 1.0e-12). Note that this is different than the popular inverse hyperbolic sine regularization proposed by Ben-Zion and Rice [Ben-Zion and Rice, 1997] to permit zero slip rates. Following Kaneko *et al.* [Kaneko et al., 2008], we integrate the evolution equation for the state variable, keeping slip rate constant, to get

$$\theta(t+\Delta t) = \theta(t)\exp\left(\frac{-V(t)\Delta t}{L}\right) + \frac{L}{V(t)}\left(1 - \exp\left(-\frac{V(t)\Delta t}{L}\right)\right). \tag{6.72}$$

As the slip rate approaches zero, the first exponential term approaches 1. Using the first three terms of the Taylor series expansion of the second exponential yields

$$\theta(t+\Delta t) = \begin{cases} \theta(t)\exp\left(-\frac{V(t)\Delta t}{L}\right) + \Delta t - \frac{1}{2}\frac{V(t)\Delta t^2}{L} & \frac{V(t)\Delta t}{L} < 0.00001 \\ \theta(t)\exp\left(-\frac{V(t)\Delta t}{L}\right) + \frac{L}{V(t)}\left(1 - \exp\left(-\frac{V(t)\Delta t}{L}\right)\right) & \frac{V(t)\Delta t}{L} \ge 0.00001 \end{cases} \tag{6.73}$$

A zero value for the initial state results in infinite values for the coefficient of friction. To avoid such behavior when the user fails to provide nonzero values for the initial state, we set the state variable to $L/V_0$.

The properties include:

**`linear_slip_rate`** Nondimensional slip rate at which linearization occurs, $V_{linear}$. In quasi-static simulations it should be about one order of magnitude larger than absolute tolerance in solve.

RateStateAgeing parameters in a `cfg` file

```
[pylithapp.problem.interfaces.fault]
friction = pylith.friction.RateStateAgeing ; Change from the default
friction.linear_slip_rate = 1.0e-12 ; default value
```

The spatial database files for the rate- and state-friction model properties and state variables specify the spatial variation of the fault constitutive model parameters given in Table 6.18 on the next page.

Table 6.18: Values in spatial databases for Dieterich-Ruina rate-state friction.

| Database | Value | Description |
|---|---|---|
| **db_properties** | reference-friction-coefficient | Steady-state coefficient of friction at slip rate $V_0$, $\mu_s$ |
| | reference-slip-rate | Reference slip rate, $V_0$ |
| | characteristic-slip-distance | Slip-weakening parameter, $L$ |
| | constitutive-parameter-a | Coefficient for the ln slip rate term, $a$ |
| | constitutive-parameter-b | Coefficient for the ln state variable term, $b$ |
| | cohesion | Cohesive stress, $T_c$ |
| **db_initial_state** | state-variable | State variable, $\theta$ |

### 6.4.6 Slip Impulses for Green's Functions

Computing static Green's functions using the GreensFns problem requires a specialized fault implementation, FaultCohesiveImpulses, to set up the slip impulses. The parameters controlling the slip impulses include the components involved (lateral, reverse, and/or fault opening) and the amplitude of the pulses (e.g., selecting a subset of a fault or including a spatial variation). The FaultCohesiveImpulses properties and facilities include:

**threshold** Threshold for non-zero amplitude; impulses will only be generated at locations on the fault where the amplitude exceeds this threshold.

**impulse_dof** Array of components associated with impulses, e.g., [0, 1, 2] for slip involving the left-lateral, reverse, and opening components, respectively.

**db_impulse_amplitude** Spatial database for amplitude of slip impulse (scalar field). Default is SimpleDB.

FaultCohesiveImpulses parameters in a cfg file

```
[pylithapp.problem.interfaces]
fault = pylith.faults.FaultCohesiveImpulses ; Change from the default

[pylithapp.problem.interfaces.fault]
threshold = 1.0e-6*m ; default
impulse_dof = [0] ; lateral slip-only
db_impulse_amplitude.iohandler.filename = myimpulse.spatialdb
db_impulse_amplitude.label = Impulse amplitude
```

## 6.5 Gravitational Body Forces

Many problems in geophysics require the consideration of gravitational body forces. For example, it is often important to include the effects of the lithostatic (overburden) pressure. In future releases of PyLith that permit nonlinear bulk rheologies, body forces will affect plastic yield criteria and the deformation field for large deformation/finite strain problems. As described in Chapter 2 on page 7, the body forces contribute to the residual,

$$r_i^n = \int_V f_i N^n \, dV. \tag{6.74}$$

For gravitational body forces, the body force per unit volume, $f_i$, is given as the product of the mass density, $\rho$, the scalar gravitational acceleration value, $g$, and the gravitational acceleration orientation vector, $a_i$:

$$f_i = \rho g a_i. \tag{6.75}$$

The mass density is a property of every material model, and is thus included in the spatial database with the physical properties for each material. The gravitational acceleration is assumed to be uniform and constant for a given problem, with a default value of 9.80665 m/s$^2$. The orientation vector will depend on the dimension of the problem as well as the coordinate system being used. The default orientation vector has components (0, 0, -1). This is appropriate for three-dimensional problems where the gravity vector is aligned with the negative z-axis, as would be the case in a geographic-projected coordinate system or a generic Cartesian coordinate system. For cases in which the curvature of the earth should be considered, the spatialdata package provides an earth-centered, earth-fixed (ECEF) coordinate system and a local georeferenced Cartesian system; in each of these cases the orientation vector is computed automatically, although this feature has not been tested. For problems in one or two dimensions where the orientation vector is constant, the vector will need to be explicitly specified. For example, in a two-dimensional problem, the vector might be specified as (0, -1, 0). The vector still has three components, although the extra component is not used.

Turning on gravitational body forces in a `cfg` file

```
[pylithapp.timedependent]
gravity_field = spatialdata.spatialdb.GravityField

[pylithapp.timedependent.gravity_field]
acceleration = 100.0*m*s**-2  ; default is 9.80665*m*s**-2
gravity_dir = [0, -1, 0]  ; default is [0, 0, -1]
```

Examples using gravity are described in Sections 7.9.8 on page 164 and 7.17 on page 200.

# Chapter 7

# Examples

## 7.1 Overview

This chapter includes several suites of examples. Each suite includes several "steps" which are examples that increase in complexity from one "step" to the next. In some cases, a later step may make use of output from an earlier step; these cases are clearly documented. Table 7.1 classifies the level of difficulty of each example suite and provides a general description of the type of problems discussed.

Table 7.1: Overview of example suites.

| Directory | Section(s) | Difficulty | Description |
|---|---|---|---|
| twocells | 7.3–7.7 | novice | Toy problems with ASCII two-cell meshes. |
| 3d/hex8 | 7.9 | beginner | Illustration of most features using simple CUBIT box mesh. |
| 3d/tet4 | 7.8 | beginner | Illustration of refinement using simple LaGriT box mesh. |
| bar_shearwave | 7.12–7.15 | beginner | Illustration of wave propagation using simple shear beam. |
| 2d/subduction | 7.10 | intermediate | Illustration of coseismic, postseismic, and creep deformation using a 2-D subduction zone cross-section with a CUBIT mesh. |
| 2d/greensfns | 7.16 | intermediate | Illustration of computing static Green's functions for a strike-slip and reverse fault using a CUBIT mesh. |
| 3d/subduction | 7.18 | intermediate | Illustration of most PyLith features for quasi-static deformation using a 3-D subduction zone with a CUBIT mesh. |

The 3d/subduction example suite is the newest and most comprehensive. Users wanting to use PyLith in their research should work through relevant beginner examples and then the 3d/subduction examples.

### 7.1.1 Prerequisites

Before you begin any of the examples, you will need to install PyLith following the instructions in Chapter 3 on page 17. For more complex examples, you will also need either Trelis (available from csimsoft.com), CUBIT (available to US federal government agencies from cubit.sandia.gov) or LaGriT (available form meshing.lanl.gov) mesh generation software to create the meshes. If you do not wish to create your own mesh at this time, the meshes are also provided as part of the example. The ParaView www.paraview.org visualization package may be used to view simulation results. ParaView 3 includes built-in documentation that is accessed by clicking on the Help menu item. Some additional documentation is available on the ParaView Wiki site paraview.org/Wiki/ParaView. You may use other visualization software, but some adaption from what is described here will be necessary. Furthermore, you can complete a subset of the example using files provided (as described below), skipping the steps for which you do not have the proper software packages installed.

### 7.1.2 Input Files

The files needed to work through the examples are found in the `examples` directory under the top-level PyLith directory. There are five examples in `examples/twocells`, each consisting of just two cells (elements). These very simple examples make use of PyLith mesh ASCII format to define the mesh. This format is useful for understanding the basics of how PyLith works, since it is easy to create these files by hand. More complex problems, such as those found in `examples/3d`, use external mesh generation software to create the meshes. All of the files used in the example problems are extensively documented with comments.

## 7.2 ParaView Python Scripts

**New in v2.2.1**

In some of the examples (currently only the 2D and 3D subduction zone examples) we provide ParaView Python scripts for visualizing the input finite-element mesh and the PyLith simulation results. Some of these scripts are very generic and are easily reused; others are more specific to the examples. The primary advantage of the ParaView Python scripts is that they make it easy to replicate visualizations, whether they are produced by the developers and regenerated by users.

There are several different ways to run the ParaView Python scripts:

- Within the ParaView GUI, select Tools→Python Shell. Override the default parameters as desired (which we will discuss later in this section). Click on the Run Script button, and navigate to the select the script you want to run.
- From a shell (terminal window) start ParaView from the command line with the `--script=FILENAME` where `FILENAME` is the relative or absolute path to the ParaView Python script. Note that this method does not provide a mechanism for overriding the default parameters.
- Run the ParaView Python script directly from a shell (terminal window) via the command line. You can use command line arguments to override the default values for the parameters. If pvpython is not in your PATH, then you can run a script called `MY_SCRIPT.py` using: `PATH_TO_PVPYTHON/pvpython MY_SCRIPT.py`

> ★ **Tip**
>
> Running the ParaView Python script from within the ParaView GUI allows further manipulation of the data, which is not possible when running the ParaView Python script outside the ParaView GUI. When run outside the ParaView GUI, the interaction is limited to rotating, translating, and zooming.

> ⚠ **Important**
>
> The ParaView Python scripts run Python via `pvpython`, which is a customized version of the Python interpreter included in the ParaView distribution. This is different from Python provided with your operating system and/or the one included in the PyLith distribution. This means you cannot, in general, import Python modules provided with the PyLith distribution into ParaView.

★ **Tip**

In creating the ParaView Python scripts, we performed the steps within the GUI while capturing the commands using Tools→Start Trace and then Tools→Start Trace. This makes it very easy to create the Python script. Note that we have omitted supefluous commands in the trace when transferring the trace into a Python script. See the ParaView documentation for additional information about the Python API.

### 7.2.1 Overriding Default Parameters

We setup the ParaView Python scripts, so that when they are run from the command line in the main directory for a given example, e.g., `examples/3d/subduction`, the script will produce the output discussed in the manual. If you start ParaView from the OS X Dock or a similar method, like a shortcut, then you will need to override at least the default values for the data file(s).

In order to override the default values from within the ParaView GUI, simply set the values within the Python shell. For example, to set the value of the variable EXODUS_FILE to the absolute path of the input file,

ParaView Python shell

```
>>> EXODUS_FILE = "/home/johndoe/pylith/examples/3d/subduction/mesh/mesh_tet.exo"
```

In this case, we use the Python os module to get the absolute path of the home directory and append the path to the Exodus file with the appropriate separators for the operating system.

⚠ **Important**

In each of the ParaView Python scripts, the names of the variables and their default values are given by the DEFAULTS dictionary near the top of the file.

## 7.3 Examples Using Two Triangles

PyLith features discussed in this example:

- Quasi-static solution
- Mesh ASCII format
- Dirichlet boundary conditions
- Kinematic fault interface conditions
- Plane strain linearly elastic material
- VTK output
- Linear triangular cells
- SimpleDB spatial database
- ZeroDispDB spatial database

All of the files necessary to run the examples are contained in the directory `examples/twocells/twotri3`.

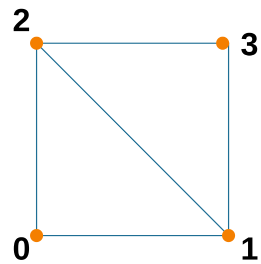



Figure 7.1: Mesh composed of two linear triangular cells used in the example problems.

### 7.3.1 Overview

This example is the simplest 2D example of a quasi-static finite element problem (a simpler problem would consist of a 1D bar). It is a mesh composed of two linear triangles subject to displacement boundary conditions, assuming plane-strain linear elastic behavior. Due to the simple geometry of the problem, the mesh may be constructed by hand, using PyLith mesh ASCII format. In this example, we will walk through the steps necessary to construct, run, and view three problems that use the same mesh. In addition to this manual, each of the files for the example problem includes extensive comments.

### 7.3.2 Mesh Description

The mesh consists of two triangles forming a square with edge lengths of one unit (Figure 7.1). The mesh geometry and topology are described in the file `twotri3.mesh`, which is in PyLith mesh ASCII format. This file format is described in Appendix C on page 267. This file describes the dimensionality of the problem (1D, 2D, or 3D), the coordinates of the vertices (nodes), the vertices composing each cell (element), the material ID to be associated with each cell, and groups of vertices that may be used to define faults or surfaces to which boundary conditions may be applied.

### 7.3.3 Additional Common Information

In addition to the mesh, the three example problems share additional information. For problems of this type, it is generally useful to create a file named `pylithapp.cfg` in the working directory, since this file is read automatically every time PyLith is run. Settings specific to a particular problem may be placed in other `cfg` files, as described later, and then those files are placed on the command line. The settings contained in `pylithapp.cfg` for this problem consist of:

**`pylithapp.journal.info`** Settings that control the verbosity of the output for the different components.

**`pylithapp.mesh_generator`** Settings that control mesh importing, such as the importer type, the filename, and the spatial dimension of the mesh.

**`pylithapp.timedependent`** Settings that control the problem, such as the total time, time step size, and spatial dimension.

**`pylithapp.timedependent.materials`** Settings that control the material type, specify which material IDs are to be associated with a particular material type, and give the name of the spatial database containing the physical properties for the material. The quadrature information is also given.

**`pylithapp.petsc`** PETSc settings to use for the problem, such as the preconditioner type.

All of the problems in this directory use the same material database, as specified under pylithapp.timedependent.materials in `pylithapp.cfg`. This information is contained in the file `matprops.spatialdb`. Although the material model is specified in `pylithapp.cfg`, the values for the physical properties of the material are given in `matprops.spatialdb`. For this example, values describing elastic plane strain material properties are given at a single point, resulting in uniform material properties.

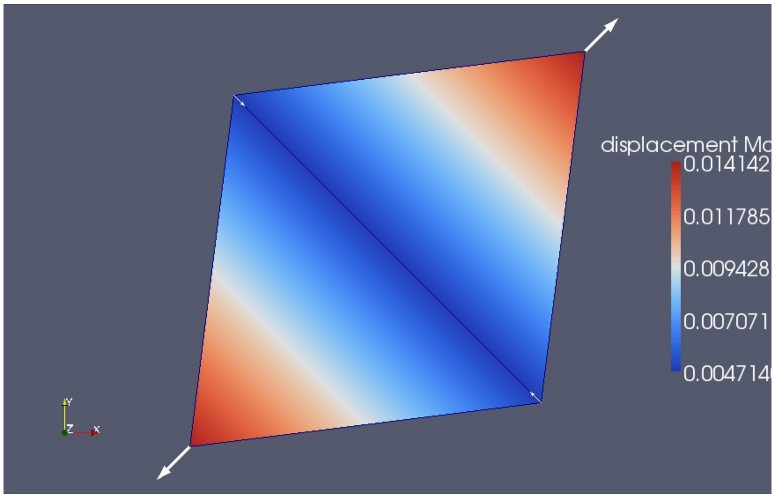



Figure 7.2: Color contours and vectors of displacement for the axial displacement example using a mesh composed of two linear triangular cells.

### 7.3.4 Axial Displacement Example

The first example problem is extension of the mesh along the diagonal extending from the lower left to the upper right of the square mesh. Parameter settings that augment those in `pylithapp.cfg` are contained in the file `axialdisp.cfg`. These settings are:

**pylithapp.timedependent** Specifies an implicit formulation for the problem and specifies the array of boundary conditions.

**pylithapp.timedependent.bc.bc** Defines which degrees of freedom are being constrained (x and y), gives the label (defined in `twotri3.mesh`) defining the points desired, assigns a label to the boundary condition set, and gives the name of the spatial database with the values for the Dirichlet boundary condition (`axialdisp.spatialdb`).

**pylithapp.problem.formulation.output.output.writer** Gives the base filename for VTK output (`axialdisp.vtk`).

**pylithapp.timedependent.materials.material.output** Gives the base filename for state variable output files (`axialdisp-statevars.vtk`).

The values for the Dirichlet boundary condition are given in the file `axialdisp.spatialdb`, as specified in `axialdisp.cfg`. The format of all spatial database files is similar. In this case, the desired displacement values are given at two points (lower left and upper right). Since data are being specified at points (rather than being uniform over the mesh, for example), the data dimension is one.

The files containing common information (`twotri3.mesh`, `pylithapp.cfg`, `matprops.spatialdb`) along with the problem-specific files (`axialdisp.cfg`, `axialdisp.spatialdb`) provide a complete description of the problem, and we can then run this example by typing

```
$ pylith axialdisp.cfg
```

Once the problem has run, three files will be produced. The first file is named `axialdisp_t0000000.vtk`. The `t0000000` indicates that the output is for the first (and only) time step, corresponding to an elastic solution. This file contains mesh information as well as displacement values at the mesh vertices. The second file is named `axialdisp-statevars_t0000000.vtk`. This file contains the state variables for each cell. The default fields are the total strain and stress fields. Since the cells are linear triangles, there is a single quadrature point for each cell and thus a single set of stress and strain values for each cell. The final file (`axialdisp-statevars_info.vtk`) gives the material properties used for the problem. Since we have not specified which properties to write, the default properties (`mu`, `lambda`, `density`) are written. All of the `vtk` files may be used with a number of visualization packages. If the problem ran correctly, you should be able to generate a figure such as Figure 7.2, which was generated using ParaView.

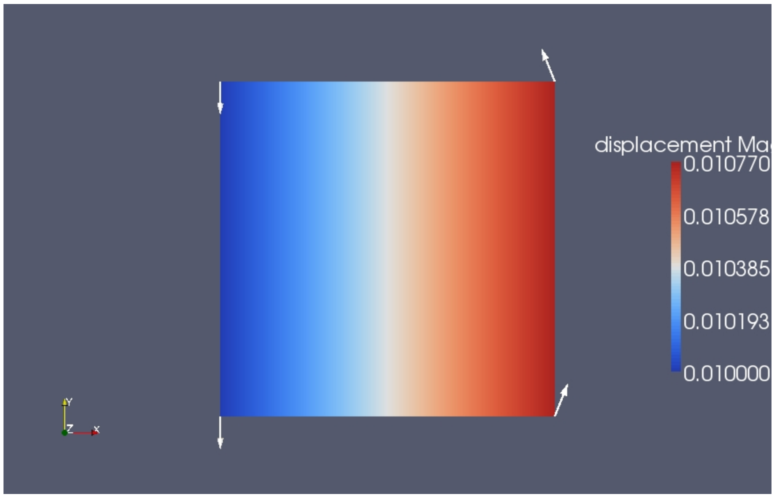



Figure 7.3: Color contours and vectors of displacement for the shear displacement example using a mesh composed of two linear triangular cells.

### 7.3.5 Shear Displacement Example

The next example problem is shearing of the mesh in the y direction using displacements applied along the positive and negative x boundaries. Parameter settings that augment those in `pylithapp.cfg` are contained in the file `sheardisp.cfg`. These settings include:

**`pylithapp.timedependent.bc.x_neg`** Specifies the boundary conditions for the left side of the mesh, defining which degrees of freedom are being constrained (x and y), giving the label (**`x_neg`**, defined in `twotri3.mesh`) defining the points desired, assigning a label to the boundary condition set, and giving the name of the spatial database with the values for the Dirichlet boundary condition (`sheardisp.spatialdb`).

**`pylithapp.timedependent.bc.x_pos`** Specifies the boundary conditions for the left side of the mesh, defining which degrees of freedom are being constrained (y only), giving the label (**`x_pos`**, defined in `twotri3.mesh`) defining the points desired, assigning a label to the boundary condition set, and giving the name of the spatial database with the values for the Dirichlet boundary condition (`sheardisp.spatialdb`).

The files containing common information (`twotri3.mesh`, `pylithapp.cfg`, `matprops.spatialdb`) along with the problem-specific files (`sheardisp.cfg`, `sheardisp.spatialdb`) provide a complete description of the problem, and we can then run this example by typing

```
$ pylith sheardisp.cfg
```

Once the problem has run, three files will be produced as in the previous example. If the problem ran correctly, you should be able to generate a figure such as Figure 7.3, which was generated using ParaView.

### 7.3.6 Kinematic Fault Slip Example

The next example problem is left-lateral fault slip applied between the two triangular cells using kinematic cohesive cells. The lower left and upper right boundaries are held fixed in the x and y directions. Parameter settings that augment those in `pylithapp.cfg` are contained in the file `dislocation.cfg`. The solution corresponds to rigid body rotation of each triangular cell. As a result, the tractions on the fault are zero. These settings include:

**`pylithapp.journal.info`** Turns on journaling for 1D quadrature (used for 2D faults) and for cohesive kinematic faults.

**`pylithapp.timedependent.bc.bc`** Defines which degrees of freedom are being constrained (x and y), gives the label (defined in `twotri3.mesh`) defining the points desired, and assigns a label to the boundary condition set. In this case, rather than specifying a spatial database file with the values for the Dirichlet boundary condition,

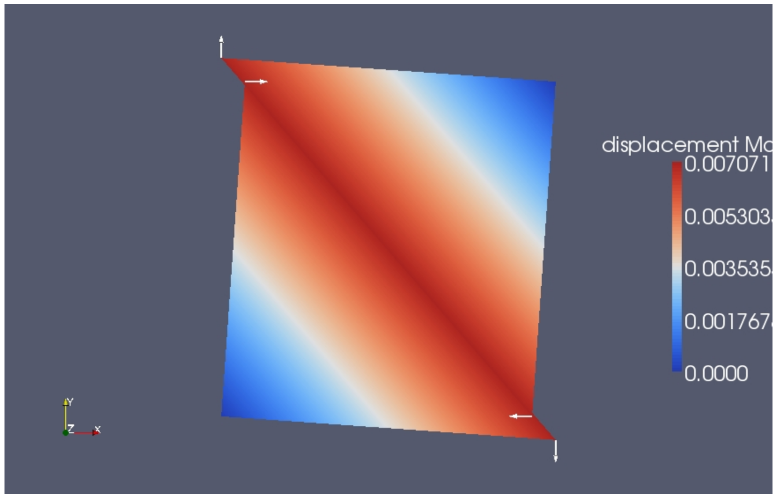



Figure 7.4: Color contours and vectors of displacement for the kinematic fault example using a mesh composed of two linear triangular cells.

the default database (ZeroDispDB) for Dirichlet boundary conditions is used, which sets the displacements to zero.

**pylithapp.timedependent.interfaces** Gives the label (defined in `twotri3.mesh`) defining the points on the fault, provides quadrature information, and then gives database names for material properties (needed for conditioning), fault slip, peak fault slip rate, and fault slip time.

**pylithapp.timedependent.interfaces.fault.output.writer** Gives the base filename for cohesive cell output files (`dislocation-fault.vtk`).

Rather than specifying the displacement boundary conditions in a spatial database file, we use the default behavior for Dirichlet boundary conditions, which is a uniform, constant displacement of zero.

The fault example requires three additional database files that were not needed for the simple displacement examples. The first file (`dislocation_slip.spatialdb`) specifies 0.01 m of left-lateral fault slip for the entire fault. The data dimension is zero since the same data are applied to all points in the set. The default slip time function is a step-function, so we also must provide the time at which slip begins. The elastic solution is associated with advancing from $t = -dt$ to $t = 0$, so we set the slip initiation time for the step-function to 0 in `dislocation_sliptime.spatialdb`.

The files containing common information (`twotri3.mesh`, `pylithapp.cfg`, `matprops.spatialdb`) along with the problem-specific files (`dislocation.cfg`, `dislocation_slip.spatialdb`, `dislocation_sliptime.spatialdb`) provide a complete description of the problem, and we can then run this example by typing

```
$ pylith dislocation.cfg
```

Once the problem has run, five files are produced. In addition to the files produced in the previous two examples, this example produces two files associated with the fault interface. The file `dislocation-fault_t0000000.vtk` gives the fault slip for each vertex on the fault along with the computed traction change for the cohesive cell. The file `dislocation-fault_info.vtk` provides information such as the normal direction, final slip, and slip time for each vertex on the fault. If the problem ran correctly, you should be able to generate a figure such as Figure 7.4, which was generated using ParaView.

## 7.4 Example Using Two Quadrilaterals

PyLith features discussed in this example:

- Quasi-static solution
- Mesh ASCII format

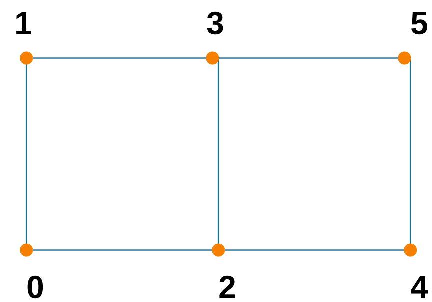



Figure 7.5: Mesh composed of two bilinear quadrilateral cells used for the example problems.

- Dirichlet boundary conditions
- Neumann boundary conditions
- Kinematic fault interface conditions
- Plane strain linearly elastic material
- VTK output
- Bilinear quadrilateral cells
- SimpleDB spatial database
- ZeroDispDB spatial database

All of the files necessary to run the examples are contained in the directory `examples/twocells/twoquad4.`

### 7.4.1 Overview

This example is another simple 2D example of a quasi-static finite element problem. It is a mesh composed of two bilinear quadrilaterals subject to displacement or traction boundary conditions, assuming plane-strain linear elastic behavior. Due to the simple geometry of the problem, the mesh may be constructed by hand, using PyLith mesh ASCII format to describe the mesh. In this example, we will walk through the steps necessary to construct, run, and view four problems that use the same mesh. In addition to this manual, each of the files for the example problem includes extensive comments.

### 7.4.2 Mesh Description

The mesh consists of two square cells with edge lengths of one unit forming a regular region (Figure 7.5). The mesh geometry and topology are described in the file `twoquad4.mesh`, which is in PyLith mesh ASCII format. This file describes the dimensionality of the problem (in this case 2D), the coordinates of the vertices (nodes), the vertices composing each cell (element), the material ID to be associated with each cell, and then provides groups of vertices that may be used to define faults or surfaces to which boundary conditions may be applied.

### 7.4.3 Additional Common Information

In addition to the mesh, the four example problems share additional information. As in the previous examples, we place this information in `pylithapp.cfg`, since this file is read automatically every time PyLith is run. Settings specific to a particular problem may be placed in other `cfg` files, as described later, and then those files are placed on the command line.

### 7.4.4 Axial Displacement Example

The first example problem is extension of the mesh along the x axis. Parameter settings that override or augment those in `pylithapp.cfg` are contained in the file `axialdisp.cfg`. These include:

**pylithapp.timedependent.bc.x_neg** Specifies the boundary conditions for the left side of the mesh, defining which degrees of freedom are being constrained (x), giving the label (defined in `twoquad4.mesh`) defining the points desired, assigning a label to the boundary condition set, and giving the name of the spatial database with the values for the Dirichlet boundary condition (`axialdisp.spatialdb`).

**pylithapp.timedependent.bc.x_pos** Specifies the boundary conditions for the right side of the mesh, defining which degrees of freedom are being constrained (x), giving the label (defined in `twoquad4.mesh`) defining the points desired, assigning a label to the boundary condition set, and giving the name of the spatial database defining the boundary conditions (`axialdisp.spatialdb`).

**pylithapp.timedependent.bc.y_neg** Specifies the boundary conditions for the bottom two corners of the mesh, defining which degrees of freedom are being constrained (y), giving the label (defined in `twoquad4.mesh`) defining the points desired, assigning a label to the boundary condition set, and giving the name of the spatial database with the values for the Dirichlet boundary condition (`axialdisp.spatialdb`).

The values for the Dirichlet boundary condition are given in the file `axialdisp.spatialdb`, as specified in `axialdisp.cfg`. Because the data are being specified using two control points with a linear variation in the values between the two (rather than being uniform over the mesh, for example), the data dimension is one.

The files containing common information (`twoquad4.mesh`, `pylithapp.cfg`, `matprops.spatialdb`) along with the problem-specific files (`axialdisp.cfg`, `axialdisp.spatialdb`) provide a complete description of the problem, and we can then run this example by typing

```
$ pylith axialdisp.cfg
```

As in the two triangle axial displacement example, three files will be produced. If the problem ran correctly, you should be able to produce a figure such as Figure 7.6, which was generated using ParaView.

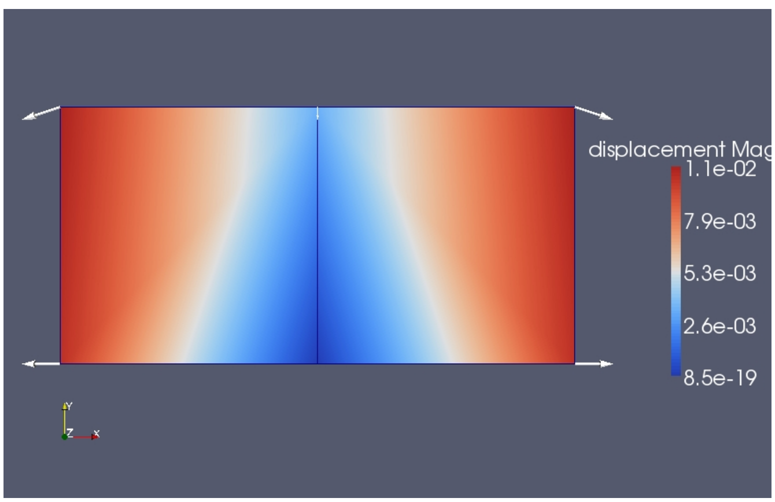



Figure 7.6: Color contours and vectors of displacement for the axial displacement example using a mesh composed of two bilinear quadrilateral cells.

### 7.4.5 Shear Displacement Example

The next example problem is shearing of the mesh in the y direction using displacements applied along the positive and negative x boundaries. Parameter settings that override or augment those in `pylithapp.cfg` are contained in the file `sheardisp.cfg`. These include:

**pylithapp.timedependent.bc.x_neg** Specifies the boundary conditions for the left side of the mesh, defining which degrees of freedom are being constrained (x and y), giving the label (**x_neg**, defined in `twoquad4.mesh`) defining the points desired, assigning a label to the boundary condition set, and giving the name of the spatial

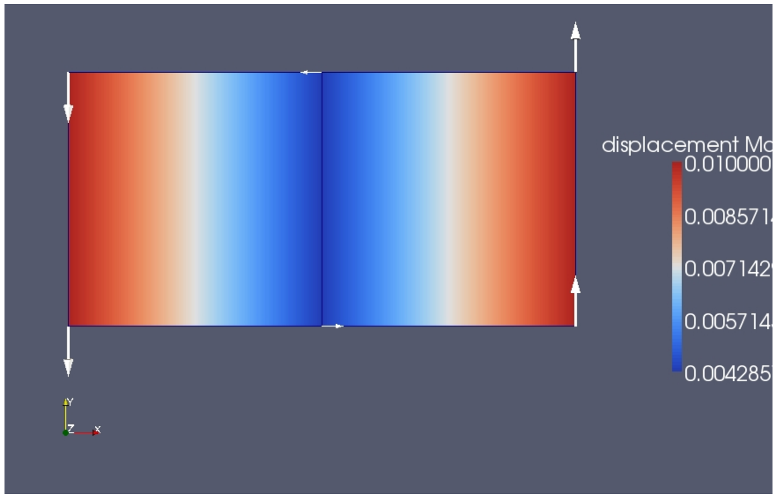



Figure 7.7: Color contours and vectors of displacement for the shear displacement example using a mesh composed of two bilinear quadrilateral cells.

database with the values for the Dirichlet boundary condition (`sheardisp.spatialdb`).

**pylithapp.timedependent.bc.x_pos** Specifies the boundary conditions for the left side of the mesh, defining which degrees of freedom are being constrained (y only), giving the label (**x_pos**, defined in `twoquad4.mesh`) defining the points desired, assigning a label to the boundary condition set, and giving the name of the spatial database with the values for the Dirichlet boundary condition (`sheardisp.spatialdb`).

The values for the Dirichlet boundary conditions are described in the file `sheardisp.spatialdb`, as specified in `sheardisp.cfg`. In this case, the desired displacement values are given at two control points, corresponding to the two edges we want to constrain. Since data are being specified at two points with a linear variations in the values between the points (rather than being uniform over the mesh, for example), the data dimension is one.

The files containing common information (`twoquad4.mesh`, `pylithapp.cfg`, `matprops.spatialdb`) along with the problem-specific files (`sheardisp.cfg`, `sheardisp.spatialdb`) provide a complete description of the problem, and we can then run this example by typing

```
$ pylith sheardisp.cfg
```

As in the previous example, three files will be produced. If the problem ran correctly, you should be able to produce a figure such as Figure 7.7, which was generated using ParaView.

### 7.4.6 Kinematic Fault Slip Example

The next example problem is a left-lateral fault slip applied between the two square cells using kinematic cohesive cells. The left and right boundaries are held fixed in the x and y directions. Parameter settings that override or augment those in `pylithapp.cfg` are contained in the file `dislocation.cfg`. These settings include:

**pylithapp.timedependent.bc.x_neg** Specifies the boundary conditions for the left side of the mesh, defining which degrees of freedom are being constrained (x and y), giving the label (**x_neg**, defined in `twoquad4.mesh`) defining the points desired, and assigning a label to the boundary condition set. Instead of specifying a spatial database file for the values of the Dirichlet boundary condition, we use the default spatial database (ZeroDispDB) for the Dirichlet boundary condition, which sets the displacements to zero for all time.

**pylithapp.timedependent.bc.x_pos** Specifies the boundary conditions for the right side of the mesh, defining which degrees of freedom are being constrained (x and y), giving the label (**x_neg**, defined in `twoquad4.mesh`) defining the points desired, and assigning a label to the boundary condition set. We use the ZeroDispDB for this boundary condition as well, which sets the displacements to zero for all time.

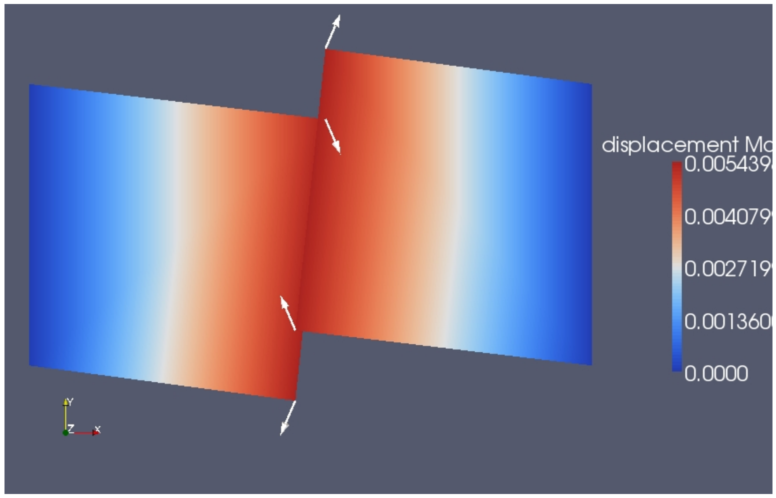



Figure 7.8: Color contours and vectors of displacement for the kinematic fault example using a mesh composed of two bilinear quadrilateral cells.

**pylithapp.timedependent.interfaces** Gives the label (defined in `twoquad4.mesh`) defining the points on the fault, provides quadrature information, and then gives database names for material properties (needed for conditioning), fault slip, peak fault slip rate, and fault slip time.

The fault example requires three additional database files that were not needed for the simple displacement examples. The first file (`dislocation_slip.spatialdb`) specifies 0.01 m of left-lateral fault slip for the entire fault. The data dimension is zero since the same data are applied to all points in the set. The default slip time function is a step-function, so we also must provide the time at which slip begins. The elastic solution is associated with advancing from $t = -dt$ to $t = 0$, so we set the slip initiation time for the step-function to 0 in `dislocation_sliptime.spatialdb`.

The files containing common information (`twoquad4.mesh`, `pylithapp.cfg`, `matprops.spatialdb`) along with the problem-specific files (`dislocation.cfg`, `dislocation_slip.spatialdb`, `dislocation_sliptime.spatialdb` provide a complete description of the problem, and we can then run this example by typing

```
$ pylith dislocation.cfg
```

The addition of a fault results in two additional output files (as in the two triangle fault example), `dislocation-fault_t0000000.vtk` and `dislocation-fault_info.vtk`. These files provide output of fault slip, change in tractions, and diagnostic information such as the normal direction, final slip, and slip time for each vertex on the fault. If the problem ran correctly, you should be able to produce a figure such as Figure 7.8, which was generated using ParaView.

### 7.4.7 Axial Traction Example

The fourth example demonstrates the use of Neumann (traction) boundary conditions. Constant tractions are applied to the right edge of the mesh, while displacements normal to the boundaries are held fixed along the left and bottom edges of the mesh. Parameter settings that override or augment those in `pylithapp.cfg` are contained in the file `axialtract.cfg`. These settings include:

**pylithapp.timedependent** Specifies an implicit formulation for the problem and specifies the array of boundary conditions. The boundary condition type for **x_pos** is explicitly set to Neumann, since the default boundary condition type is DirichletBC.

**pylithapp.timedependent.bc.x_neg** Specifies the boundary conditions for the left side of the mesh, defining which degrees of freedom are being constrained (x) and giving the label (defined in `twoquad4.mesh`) defining the points desired. In this case, rather than specifying a spatial database file with values for the Dirichlet boundary conditions, we use the default spatial database (ZeroDispDB) for the Dirichlet boundary condition, which sets the displacements to zero for all time.

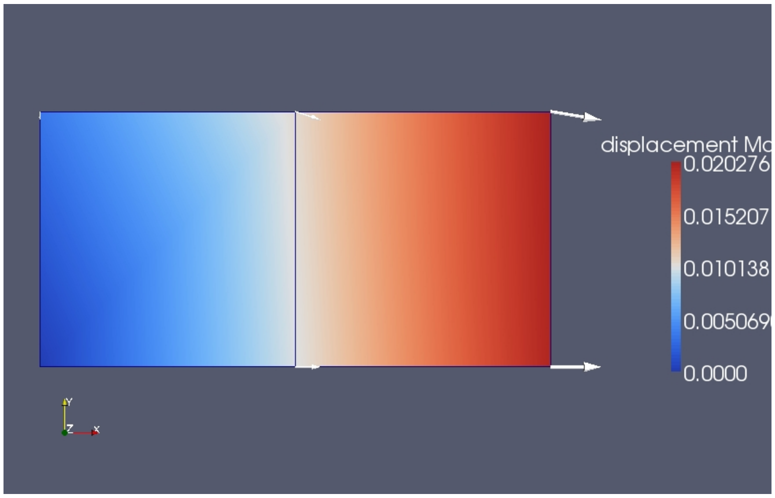



Figure 7.9: Color contours and vectors of displacement for the axial traction example using a mesh composed of two bilinear quadrilateral cells.

**pylithapp.timedependent.bc.x_pos** Specifies the Neumann boundary conditions for the right side of the mesh, giving the label (defined in `twoquad4.mesh`) defining the points desired, assigning a label to the boundary condition set, and giving the name of the spatial database with the traction vectors for the Neumann boundary condition (`axialtract.spatialdb`).

**pylithapp.timedependent.bc.y_neg** Specifies the boundary conditions for the bottom two corners of the mesh, defining which degrees of freedom are being constrained (y) and giving the label (defined in `twoquad4.mesh`) defining the points desired. In this case, we again use the ZeroDispDB, which sets the displacements to zero for all time.

**pylithapp.problem.formulation.output.output.writer** Gives the base filename for VTK output (`axialtract.vtk`).

**pylithapp.timedependent.bc.x_pos.output** Gives the field to be output for the **x_pos** boundary (`tractions`), and gives the base filename for **x_pos** boundary output (`axialtract-tractions.vtk`).

The traction vectors for the Neumann boundary conditions are given in the file `axialtract.spatialdb`, as specified in `axialtract.cfg`. The files containing common information (`twoquad4.mesh`, `pylithapp.cfg`, `matprops.spatialdb`) along with the problem-specific files (`axialtract.cfg`, `axialtract.spatialdb`) provide a complete description of the problem, and we can then run this example by typing

```
$ pylith axialtract.cfg
```

Once the problem has run, six files will be produced. This includes the five files as in the previous example plus `axialtract-tractions_` which gives the `x` and `y` components of traction applied at each integration point. If the problem ran correctly, you should be able to produce a figure such as Figure 7.9, which was generated using ParaView. The results may be compared against the analytical solution derived in Section E.1.2 on page 278.

## 7.5 Example Using Two Tetrahedra

PyLith features discussed in this example:

- Quasi-static solution
- Mesh ASCII format
- Dirichlet boundary conditions
- Kinematic fault interface conditions
- Linearly elastic isotropic material

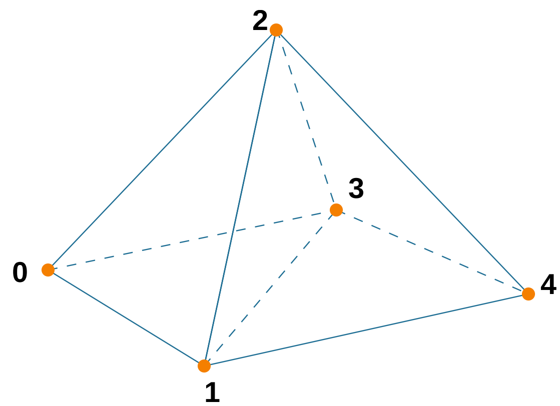



Figure 7.10: Mesh composed of two linear tetrahedral cells used for example problems.

- VTK output
- Linear tetrahedral cells
- SimpleDB spatial database
- ZeroDispDB spatial database

All of the files necessary to run the examples are contained in the directory `examples/twocells/twotet4.`

### 7.5.1 Overview

This example is a simple 3D example of a quasi-static finite element problem. It is a mesh composed of two linear tetrahedra subject to displacement boundary conditions, and is probably the simplest example of a 3D elastic problem. Due to the simple geometry of the problem, the mesh may be constructed by hand, using PyLith mesh ASCII format. In this example, we will walk through the steps necessary to construct, run, and view two problems that use the same mesh. In addition to this manual, each of the files for the example problem includes extensive comments.

### 7.5.2 Mesh Description

The mesh consists of two tetrahedra forming a pyramid shape (Figure 7.10). The mesh geometry and topology is described in the file `twotet4.mesh`, which is in PyLith mesh ASCII format.

### 7.5.3 Additional Common Information

In addition to the mesh, the two example problems share additional information, which we place in `pylithapp.cfg`.

### 7.5.4 Axial Displacement Example

The first example problem is extension of the mesh along the diagonal, extending along the base of the pyramid between two opposing vertices. Parameter settings that override or augment those in `pylithapp.cfg` are contained in the file `axialdisp.cfg`. These settings include:

**`pylithapp.timedependent.bc.bc`** Defines which degrees of freedom are being constrained (`x`, `y`, and `z`), gives the label (defined in `twotet4.mesh`) defining the points desired, assigns a label to the boundary condition set, and gives the name of the spatial database defining the boundary conditions (`axialdisp.spatialdb`).

The values for the Dirichlet boundary conditions are described in the file `axialdisp.spatialdb`, as specified in `axialdisp.cfg`. Because data are being specified using two control points (rather than being uniform over the mesh), the data dimension is one.

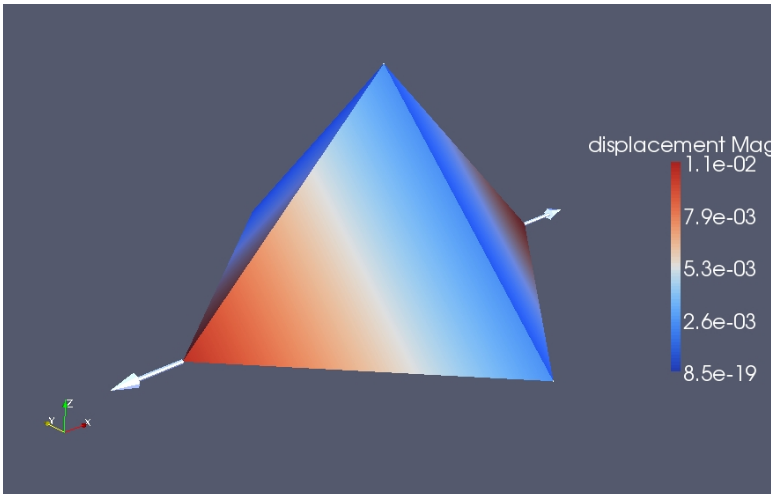



Figure 7.11: Color contours and vectors of displacement for the axial displacement example using a mesh composed of two linear tetrahedral cells.

The files containing common information (`twotet4.mesh`, `pylithapp.cfg`, `matprops.spatialdb`) along with the problem-specific files (`axialdisp.cfg`, `axialdisp.spatialdb`) provide a complete description of the problem, and we can then run this example by typing

```
$ pylith axialdisp.cfg
```

If the problem ran correctly, you should be able to produce a figure such as Figure 7.11, which was generated using ParaView.

### 7.5.5 Kinematic Fault Slip Example

The next example problem is a left-lateral fault slip applied between the two tetrahedral cells using kinematic cohesive cells. The vertices away from the fault are held fixed in the `x`, `y`, and `z` directions. Parameter settings that override or augment those in `pylithapp.cfg` are contained in the file `dislocation.cfg`. These settings include:

**pylithapp.timedependent.bc.bc** Defines which degrees of freedom are being constrained (`x`, `y`, and `z`), gives the label (defined in `twotet4.mesh`) defining the points desired, and assigns a label to the boundary condition set. Rather than specifying a spatial database file to define the boundary conditions, we use the default spatial database (ZeroDispDB) for the Dirichlet boundary condition, which sets the displacements to zero.

**pylithapp.timedependent.interfaces** Gives the label (defined in `twotet4.mesh`) defining the points on the fault, provides quadrature information, and then gives database names for material properties (needed for conditioning), fault slip, peak fault slip rate, and fault slip time.

The fault example requires three additional database files that were not needed for the simple displacement examples. The first file (`dislocation_slip.spatialdb`) specifies 0.01 m of left-lateral fault slip for the entire fault. The data dimension is zero since the same data are applied to all points in the set. The default slip time function is a step-function, so we also must provide the time at which slip begins. The elastic solution is associated with advancing from $t = -dt$ to $t = 0$, so we set the slip initiation time for the step-function to 0 in `dislocation_sliptime.spatialdb`.

The files containing common information (`twotet4.mesh`, `pylithapp.cfg`, `matprops.spatialdb`) along with the problem-specific files (`dislocation.cfg, dislocation_slip.spatialdb`, `dislocation_sliptime.spatialdb`) provide a complete description of the problem, and we can then run this example by typing

```
$ pylith dislocation.cfg
```

If the problem ran correctly, you should be able to generate a figure such as Figure 7.12 on the next page, which was generated using ParaView.

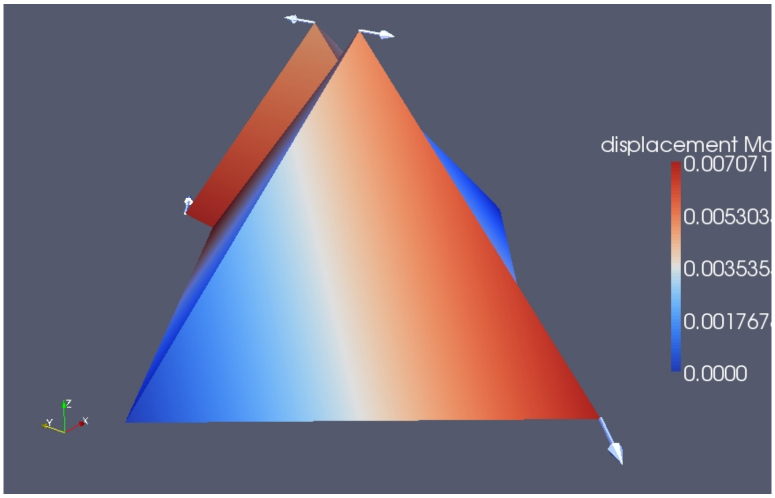



Figure 7.12: Color contours and vectors of displacement for the kinematic fault example using a mesh composed of two linear tetrahedral cells.

## 7.6 Example Using Two Hexahedra

PyLith features discussed in this example:

- Quasi-static solution
- Mesh ASCII format
- Dirichlet boundary conditions
- Kinematic fault interface conditions
- Maxwell viscoelastic material
- VTK output
- Trilinear hexahedral cells
- SimpleDB spatial database
- ZeroDispDB spatial database
- UniformDB spatial database
- Filtering of cell output fields

All of the files necessary to run the examples are contained in the directory `examples/twocells/twohex8.`

### 7.6.1 Overview

This example is a simple 3D example of a quasi-static finite element problem. It is a mesh composed of two trilinear hexahedra subject to displacement boundary conditions. One primary difference between this example and the example with two tetrahedra is that we use a Maxwell viscoelastic material model, and run the model for 10 time steps of 0.1 year each. Due to the simple geometry of the problem, the mesh may be constructed by hand, using PyLith mesh ASCII format to describe the mesh. In this example, we will walk through the steps necessary to construct, run, and view three problems that use the same mesh. In addition to this manual, each of the files for the example problems includes extensive comments.

### 7.6.2 Mesh Description

The mesh consists of two hexahedra forming a rectangular prism (Figure 7.13 on the following page). The mesh geometry and topology are described in the file `twohex8.mesh`, which is in PyLith mesh ASCII format.

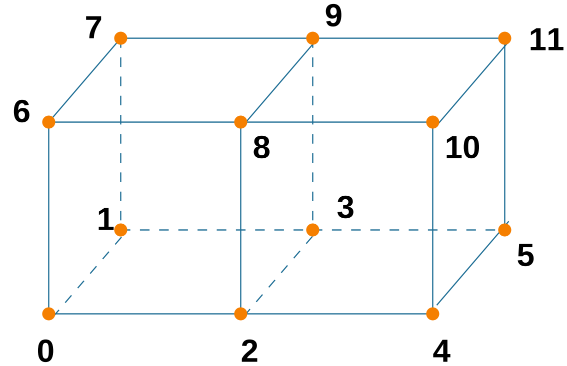



Figure 7.13: Mesh composed of two trilinear hexahedral cells used for the example problems.

### 7.6.3 Additional Common Information

In addition to the mesh, the three example problems share additional information, which we place in `pylithapp.cfg`. Note that in this example we make use of the UniformDB spatial database, rather than the SimpleDB implementation used to specify the physical properties in the other example problems. For simple distributions of material properties (or boundary conditions), this implementation is often easier to use. Examining `pylithapp.cfg`, we specify the material information with the following set of parameters:

```
[pylithapp.timedependent.materials]
material = pylith.materials.MaxwellIsotropic3D

[pylithapp.timedependent.materials.material]
label = viscoelastic material
id = 1
db = spatialdata.spatialdb.UniformDB
db.values = [vp, vs, density, viscosity]
db.data = [5773.502691896258*m/s, 3333.333333333333*m/s, 2700.0*kg/m**3, 1.0e18*Pa*s]

quadrature.cell = pylith.feassemble.FIATLagrange
quadrature.cell.dimension = 3
```

### 7.6.4 Axial Displacement Example

The first example problem is extension of the mesh along the long axis of the prism. Parameter settings that override or augment those in `pylithapp.cfg` are contained in the file `axialdisp.cfg`. These settings include:

**`pylithapp.timedependent.bc.x_neg`** Defines which degrees of freedom are being constrained (x, y, and z), gives the label (**`x_neg`**, defined in `twohex8.mesh`) defining the points desired, assigns a label to the boundary condition set, and gives the name of the spatial database with the values for the Dirichlet boundary conditions (`axialdisp.spatialdb`).

**`pylithapp.timedependent.bc.x_pos`** Defines which degrees of freedom are being constrained (x, y, and z), gives the label (**`x_pos`**, defined in `twohex8.mesh`) defining the points desired, assigns a label to the boundary condition set, and gives the name of the spatial database with the values for the Dirichlet boundary conditions (`axialdisp.spatialdb`).

**`pylithapp.timedependent.materials.material.output`** Defines the filter to be used when writing cell state variables (average over the quadrature points of the cell), specifies which state variables and properties to output, gives the base filename for state variable output files, and defines the format to use when defining the output filenames for each time step.

The values for the Dirichlet boundary conditions are given in the file `axialdisp.spatialdb`, as specified in `axialdisp.cfg`. Since data are being specified using two control points (rather than being uniform over the mesh, for example), the data dimen-

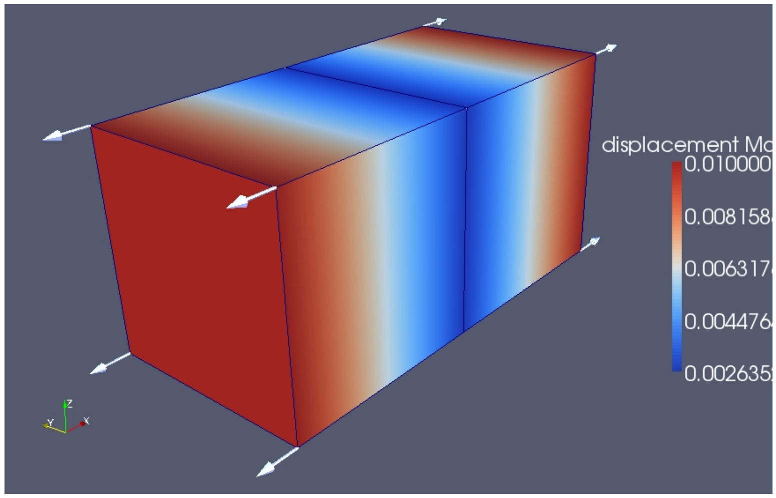



Figure 7.14: Color contours and vectors of displacement for the axial displacement example using a mesh composed of two trilinear hexahedral cells.

sion is one. Note that since we are using a Maxwell viscoelastic model, we request that additional state variables and properties be output:

```
[pylithapp.timedependent.materials.material.output]
cell_data_fields = [total_strain, viscous_strain, stress]
cell_info_fields = [mu, lambda, density, maxwell_time]
```

The files containing common information (`twohex8.mesh`, `pylithapp.cfg`) along with the problem-specific files (`axialdisp.cfg`, `axialdisp.spatialdb`) provide a complete description of the problem, and we can then run this example by typing

```
$ pylith axialdisp.cfg
```

Once the problem has run, two sets of files will be produced, along with one additional file. The first set will have filenames such as `axialdisp_txxxx.vtk`, where `xxxx` is the time for which output has been produced. In `axialdisp.cfg` we specify that the time stamp should be normalized by a value of 1.0 years and the time stamp should be of the form `xxx.x` (recall that the decimal point is removed in the filename). As a result, the filenames contain the time in tenths of a year. These files will contain mesh information as well as displacement values for the mesh vertices at the given time. The second set of files will have names such as `axialdisp-statevars_txxxx.vtk`, where `xxxx` is the time in tenths of a year (as above) for which output has been produced. These files contain the state variables for each cell at the given time. The default fields are the total strain and stress fields; however, we have also requested the viscous strains. As specified in `axialdisp.cfg`, these values are averaged over each cell. The final file (`axialdisp-statevars_info.vtk`) gives the material properties used for the problem. We have requested all of the properties available for this material model (`mu`, `lambda`, `density`, `maxwell_time`). If the problem ran correctly, you should be able to produce a figure such as Figure 7.14, which was generated using ParaView.

### 7.6.5 Shear Displacement Example

The second example problem is shearing of the mesh in the y direction. Parameter settings that override or augment those in `pylithapp.cfg` are contained in the file `sheardisp.cfg`. These settings include:

**pylithapp.timedependent.bc.x_neg** Defines which degrees of freedom are being constrained (x, y, and z), gives the label (**x_neg**, defined in `twohex8.mesh`) defining the points desired, assigns a label to the boundary condition set, and gives the name of the spatial database with the values for the Dirichlet boundary conditions (`sheardisp.spatialdb`).

**pylithapp.timedependent.bc.x_pos** Defines which degrees of freedom are being constrained (x, y, and z), gives the label (**x_pos**, defined in `twohex8.mesh`) defining the points desired, assigns a label to the boundary

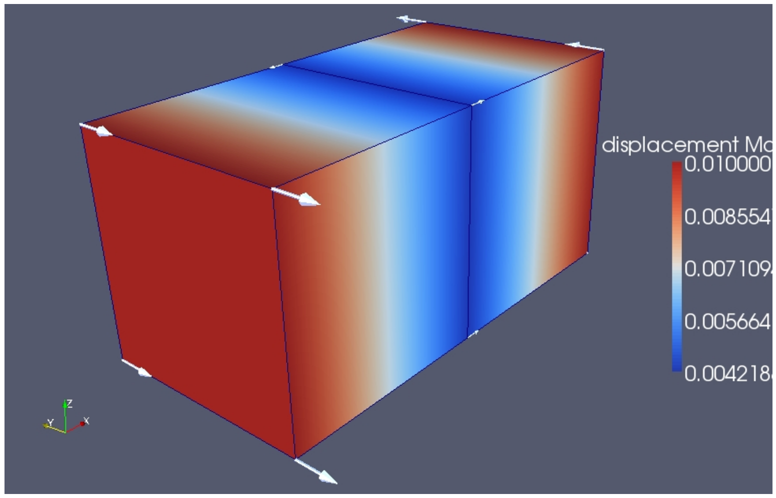



Figure 7.15: Color contours and vectors of displacement for the shear displacement example using a mesh composed of two trilinear hexahedral cells.

condition set, and gives the name of the spatial database with the values for the Dirichlet boundary conditions (`sheardisp.spatialdb`).

The values for the Dirichlet boundary conditions are given in the file `sheardisp.spatialdb`, as specified in `sheardisp.cfg`. Data are being specified at two control points (rather than being uniform over the mesh, for example), so the data dimension is one. The files containing common information (`twohex8.mesh`, `pylithapp.cfg`) along with the problem-specific files (`sheardisp.cfg`, `sheardisp.spatialdb`) provide a complete description of the problem, and we can then run this example by typing

```
$ pylith sheardisp.cfg
```

If the problem ran correctly, you should be able to generate a figure such as Figure 7.15, which was generated using ParaView.

### 7.6.6 Kinematic Fault Slip Example

The next example problem is left-lateral fault slip applied between the two hexahedral cells using kinematic cohesive cells. The vertices away from the fault are held fixed in the x, y, and z directions. Parameter settings that override or augment those in `pylithapp.cfg` are contained in the file `dislocation.cfg`. These settings include:

**`pylithapp.timedependent.bc.x_neg`** Defines which degrees of freedom are being constrained (x, y, and z), gives the label (**`x_neg`**, defined in `twohex8.mesh`) defining the points desired, and assigns a label to the boundary condition set. In this case, we use the default spatial database (ZeroDispDB) for the Dirichlet boundary condition, which sets the displacements to zero.

**`pylithapp.timedependent.bc.x_pos`** Defines which degrees of freedom are being constrained (x, y, and z), gives the label (**`x_pos`**, defined in `twohex8.mesh`) defining the points desired, and assigns a label to the boundary condition set.

**`pylithapp.timedependent.interfaces`** Gives the label (defined in `twohex8.mesh`) defining the points on the fault, provides quadrature information, and then gives database names for material properties (needed for conditioning), fault slip, peak fault slip rate, and fault slip time.

The fault example requires three additional database files that were not needed for the simple displacement examples. The first file (`dislocation_slip.spatialdb`) specifies 0.01 m of left-lateral fault slip for the entire fault. The data dimension is zero since the same data are applied to all points in the set. The default slip time function is a step-function, so we also must provide the time at which slip begins. The elastic solution is associated with advancing from $t = -dt$ to $t = 0$, so we set the slip initiation time for the step-function to 0 in `dislocation_sliptime.spatialdb`. The files containing common information (`twohex8.mesh`, `pylithapp.cfg`) along with the problem-specific files (`dislocation.cfg`,

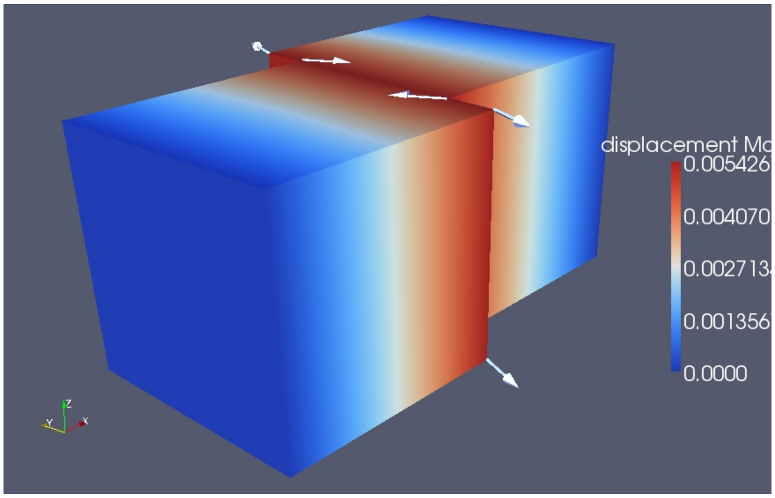



Figure 7.16: Color contours and vectors of displacement for the kinematic fault example using a mesh composed of two trilinear hexahedral cells.

`dislocation_slip.spatialdb`, `dislocation_sliptime.spatialdb`) provide a complete description of the problem, and we can then run this example by typing

```
$ pylith dislocation.cfg
```

If the problem ran correctly, you should be able to generate a figure such as Figure 7.16, which was generated using ParaView.

## 7.7 Example Using Two Tetrahedra with Georeferenced Coordinate System Mesh

PyLith features discussed in this example:

- Quasi-static solution
- Mesh ASCII format
- Dirichlet boundary conditions
- Kinematic fault interface conditions
- Linearly elastic isotropic material
- VTK output
- Linear tetrahedral cells
- SimpleDB spatial database with geographic coordinates
- SCEC CVM-H spatial database
- ZeroDispDB spatial database

All of the files necessary to run the examples are contained in the directory `examples/twocells/twotet4-geoproj`.

### 7.7.1 Overview

This example is virtually identical to the other example using two linear tetrahedra (See Section 7.5 on page 122). The primary difference is in how the material properties are assigned. For this example, the physical properties come from the SCEC CVM-H database (described in Section 4.5.4 on page 45). Using the SCEC CVM-H database is straightforward, requiring only a few modifications to `pylithapp.cfg`. Because the SCEC CVM-H database uses geographic coordinates, we must also use geographic coordinates in the PyLith mesh ASCII file and other spatial databases. Note that all of these geographic coordinate systems do not need to be the same. PyLith will automatically transform from one geographic coordinate system to another

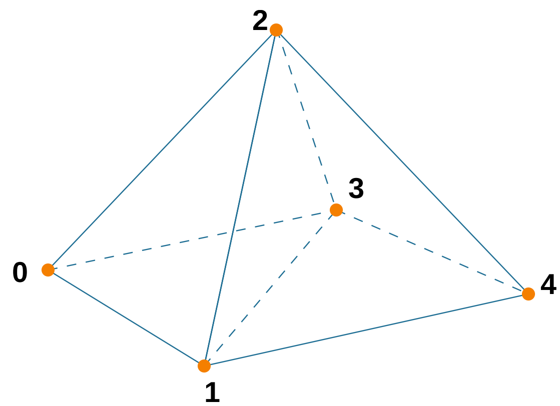



Figure 7.17: Mesh composed of two linear tetrahedral cells in a georeferenced coordinate system used for the example problems.

using the spatialdata package. The spatial databases should all use a georeferenced Cartesian coordinate system, such as a geographic projection to insure interpolation is performed properly. Since all aspects of this problem other than the material database and the coordinate system are identical to the examples in Section 7.5 on page 122, we only describe the kinematic fault problem in this example.

### 7.7.2 Mesh Description

The mesh consists of two tetrahedra forming a pyramid shape (Figure 7.17). The mesh geometry and topology are described in the file `twotet4.mesh`, which is in PyLith mesh ASCII format. If you compare this mesh against the one used in 7.5 on page 122, you will notice that, although the mesh topology is the same, the vertex coordinates are significantly different. We use zone 11 UTM coordinates with the NAD27 datum for the mesh. Although we used the same coordinate system as the SCEC CVM-H, we could have also used any other geographic projection supported by spatialdata and Proj.4. See Appendix C.2 on page 268 for other examples of using geographic coordinates.

### 7.7.3 Additional Common Information

This problem has some unique aspects compared to the other examples. First, all of the other examples use a Cartesian coordinate system, while this one uses a geographic coordinate system. In addition to using different vertex coordinates, we also define the coordinate system for the mesh in `pylithapp.cfg`:

```
pylithapp.mesh_generator.importer]
coordsys = spatialdata.geocoords.CSGeoProj
filename = twotet4.mesh
coordsys.space_dim = 3

[pylithapp.mesh_generator.importer.coordsys]
datum_horiz = NAD27
datum_vert = mean sea level
ellipsoid = clrk66

[pylithapp.mesh\_generator.importer.coordsys.projector]
projection = utm
proj-options = +zone=11
```

At the top level, we define the type of coordinate system, give the file describing the mesh, and give the number of spatial dimensions for the coordinate system. We then provide the horizontal datum and vertical datum for the coordinate system, along with the ellipsoid to be used. Finally, we specify a UTM projection, and specify zone 11 as the zone to be used.

In addition to the usual material information, we must specify that we want to use the SCECCVMH database implementation:

```
[pylithapp.timedependent.materials.material]
db = spatialdata.spatialdb.SCECCVMH
db.data_dir = /home/john/data/sceccvm-h/vx53/bin
```

The first db option defines SCECCVMH as the spatial database to be used. The next line defines the location of the vx53 data files, and must be changed to the location specified by the user when the package is installed. The package may be obtained from Harvard's Structural Geology and Tectonics structure.harvard.edu/cvm-h.

The final difference with the other examples is in the description of the spatial databases. They must also use geographic coordinates. Examining dislocation_slip.spatialdb, we find:

```
// We are specifying the data in a projected geographic coordinate system.
cs-data = geo-projected {
  to-meters = 1.0
  ellipsoid = clrk66
  datum-horiz = NAD27
  datum-vert = mean sea level
  projector = projection {
    projection = utm
    units = m
    proj-options = +zone=11
  }
}
```

### 7.7.4 Kinematic Fault Slip Example

This example problem is a left-lateral fault slip applied between the two tetrahedral cells using kinematic cohesive cells. Note that we vary the amount of fault slip for each vertex with this example, as described in dislocation_slip.spatialdb. The vertices away from the fault are held fixed in the x, y, and z directions. Parameter settings that override or augment those in pylithapp.cfg are contained in the file dislocation.cfg.

Recall that we condition problems with the kinematic fault interface using the material properties. Since the material properties are being defined using the SCEC CVM-H database, this same database should be used as the material database for the faults. This also applies to the AbsorbingDampers boundary condition.

The files containing common information (twotet4.mesh, pylithapp.cfg) along with the problem-specific files (dislocation.cfg, dislocation_slip.spatialdb, dislocation_sliptime.spatialdb) provide a complete description of the problem, and we can then run this example by typing

```
$ pylith dislocation.cfg
```

If the problem ran correctly, you should be able to generate a figure such as Figure 7.18 on the following page, which was generated using ParaView.

## 7.8 Example Using Tetrahedral Mesh Created by LaGriT

PyLith features discussed in this example:

- Quasi-static solution
- LaGriT mesh format
- Dirichlet boundary conditions
- Kinematic fault interface conditions
- Linearly elastic isotropic material
- Maxwell linear viscoelastic material

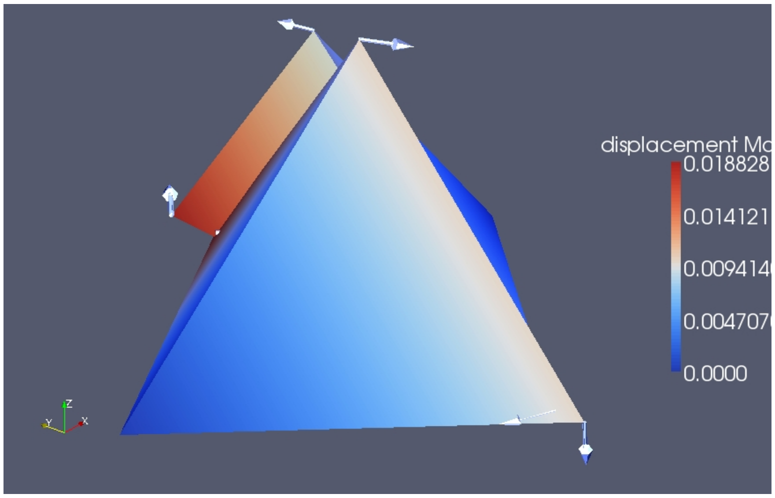



Figure 7.18: Color contours and vectors of displacement for the kinematic fault example using a mesh composed of two linear tetrahedral cells in a georeferenced coordinate system.

- Specifying more than one material
- VTK output
- Linear tetrahedral cells
- SimpleDB spatial database
- ZeroDispDB spatial database
- Custom algebraic multigrid preconditioner with split fields
- Global uniform mesh refinement

All of the files necessary to run the examples are contained in the directory `examples/3d/tet4`.

### 7.8.1 Overview

This example is a simple 3D example of a quasi-static finite element problem. It is a mesh composed of 852 linear tetrahedra subject to displacement boundary conditions. This example demonstrates the usage of the LaGriT mesh generation package `lagrit.lanl.gov` to create a mesh, as well as describing how to use a LaGriT-generated mesh in PyLith. In this example we will walk through the steps necessary to construct, run, and visualize the results for two problems that use the same mesh. For each of these problems we also consider a simulation using a custom algebraic multigrid preconditioner with a globally uniformly refined mesh that reduces the node spacing by a factor of two. In addition to this manual, each of the files for the example problems includes extensive comments.

### 7.8.2 Mesh Generation and Description

The mesh for these examples is a simple rectangular prism (Figure 7.19 on the next page). This mesh would be quite difficult to generate by hand, so we use the LaGriT mesh generation package. For this example, we provide a documented command file in `examples/3d/tet4`. Examination of this command file should provide some insight into how to use LaGriT with PyLith. For more detailed information refer to the LaGriT website `lagrit.lanl.gov`. If you have LaGriT installed on your machine, you can use the command file to create your own mesh. Otherwise, you can use the mesh that has already been created.

There are two ways to use the command file. The simplest method is to go to the `examples/3d/tet4` directory, start LaGriT, and then type:

```
input mesh_tet4_1000m.lagrit
```

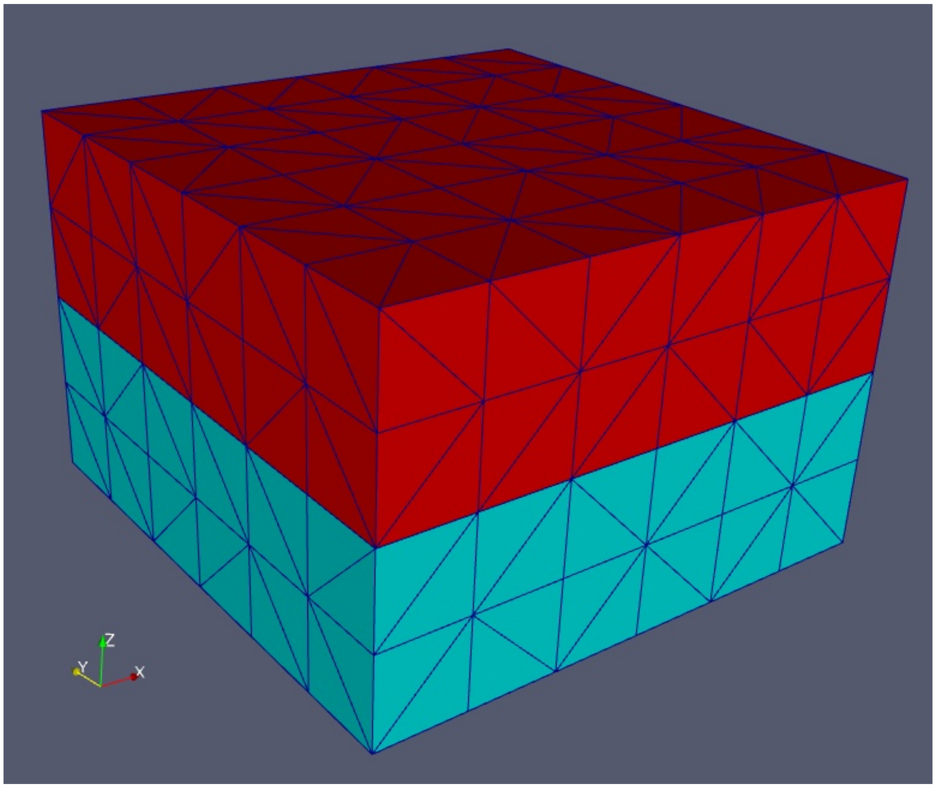



Figure 7.19: Mesh composed of linear tetrahedral cells generated by LaGriT used for the example problems. The different colors represent the different materials.

This will run the commands in that file, which will produce the necessary files to run the example. This method will create the mesh, but you will gain very little insight into what is being done. A more informative approach is to input each command directly. That way, you will see what each command does. You can simply copy and paste the commands from `mesh_tet4_1000m.lagrit`. For example, the first six commands, which define the block shape, are

```
define / domain_xm / -3.0e+3
define / domain_xp /  3.0e+3
define / domain_ym / -3.0e+3
define / domain_yp /  3.0e+3
define / domain_zm / -4.0e+3
define / domain_zp /  0.0e+3
```

Continuing through the remainder of the commands in `mesh_tet4_1000m.lagrit`, you will eventually end up with the files `tet4_1000m_binary.gmv`, `tet4_1000m_ascii.gmv`, `tet4_1000m_ascii.pset`, and `tet4_1000m_binary.pset`. The ASCII files are not actually needed, but we create them so users can see what is contained in the files. These files may also be used instead of the binary versions, if desired. The `gmv` files define the mesh information, and they may be read directly by the GMV `laws.lanl.gov/XCM/gmv/GMVHome.html` mesh visualization package. The `pset` files specify the vertices corresponding to each set of vertices on a surface used in the problem, including the fault as well as external boundaries to which boundary conditions are applied.

### 7.8.3 Additional Common Information

In addition to the mesh, the example problems share additional information. In such cases it is generally useful to create a file named `pylithapp.cfg` in the run directory, since this file is read automatically every time PyLith is run. Settings specific to a particular problem may be placed in other `cfg` files, as described later, and then those files are placed on the command line. The settings contained in `pylithapp.cfg` for this problem consist of:

**pylithapp.journal.info** Settings that control the verbosity of the output for the different components.

**pylithapp.mesh_generator** Settings that control mesh importing, such as the importer type, the filenames, and the spatial dimension of the mesh.

**pylithapp.timedependent** Settings that control the problem, such as the total time, time-step size, and number of entries in the material array.

**pylithapp.timedependent.materials** Settings that control the material type, specify which material IDs are to be associated with a particular material type, and give the name of the spatial database containing material parameters for the mesh. The quadrature information is also given.

**pylithapp.petsc** PETSc settings to use for the problem, such as the preconditioner type.

Since these examples use a mesh from LaGriT, we set the importer to MeshIOLagrit:

```
[pylithapp.mesh_generator]
reader = pylith.meshio.MeshIOLagrit

[pylithapp.mesh_generator.reader]
filename_gmv = mesh/tet4_1000m_binary.gmv
filename_pset = mesh/tet4_1000m_binary.pset
flip_endian = True
# record_header_32bit = False
```

Notice that there are a couple of settings pertinent to binary files. The first flag (`flip_endian`) is used if the binary files were produced on a machine with a different endianness than the machine on which they are being read. If you get an error when attempting to run an example, you may need to change the setting of this flag. The second flag (`record_header_32bit`) may need to be set to `False` if the version of LaGriT being used has 64-bit Fortran record headers.

This example differs from previous examples, because we specify two material groups:

```
[pylithapp.timedependent]
materials = [elastic, viscoelastic]

[pylithapp.timedependent.materials.elastic]
label = Elastic material
id = 1
db.iohandler.filename = spatialdb/mat_elastic.spatialdb
quadrature.cell = pylith.feassemble.FIATSimplex
quadrature.cell.dimension = 3

[pylithapp.timedependent.materials.viscoelastic]
label = Viscoelastic material
id = 2
db.iohandler.filename = spatialdb/mat_viscoelastic.spatialdb
quadrature.cell = pylith.feassemble.FIATSimplex
quadrature.cell.dimension = 3
```

The two materials correspond to the two different colors in Figure 7.19 on the preceding page. Each material uses a different spatial database because the physical parameters are different. In generating the mesh within LaGriT, the mesh contains four materials as a result of how LaGriT handles materials and interior interfaces. Near the end of the LaGriT command file, we merge the materials on each side of the fault into a single material to simplify the input and output from PyLith. For this example, values describing three-dimensional elastic material properties are given by the single point in the spatial databases, resulting in uniform physical properties within each material.

### 7.8.4 Shear Displacement Example

The first example problem is shearing of the mesh along the y-direction, with displacement boundary conditions applied on the planes corresponding to the minimum and maximum x-values. Parameter settings that override or augment those in `pylithapp.cfg` are contained in the file `step01.cfg`. These settings are:

**pylithapp.timedependent** Specifies an implicit formulation for the problem and specifies the array of boundary conditions.

**pylithapp.timedependent.implicit** Specifies an array of two output managers, one for the full domain, and another for a subdomain corresponding to the ground surface.

**pylithapp.timedependent.bc.x_pos** Specifies the boundary conditions for the right side of the mesh, defining which degrees of freedom are being constrained (`x` and `y`), providing the label (defined in `tet4_1000m_binary.pset`) defining the points desired, assigning a label to the boundary condition set, and giving the name of the spatial database defining the boundary conditions (`fixeddisp_shear.spatialdb`).

**pylithapp.timedependent.bc.x_neg** Specifies the boundary conditions for the left side of the mesh, defining which degrees of freedom are being constrained (`x` and `y`), providing the label (defined in `tet4_1000m_binary`.pset) defining the points desired, assigning a label to the boundary condition set, and giving the name of the spatial database defining the boundary conditions (`fixeddisp_shear.spatialdb`).

**pylithapp.timedependent.bc.z_neg** Specifies the boundary conditions for the bottom of the mesh, defining which degrees of freedom are being constrained (`x` and `y`), providing the label (defined in `tet4_1000m_binary`.pset) defining the points desired, assigning a label to the boundary condition set, and giving the name of the spatial database defining the boundary conditions (`fixeddisp_shear.spatialdb`).

**pylithapp.problem.formulation.output.domain.writer** Gives the base filename for VTK output over the entire domain (`shearxy.vtk`).

**pylithapp.problem.formulation.output.subdomain** Gives the label of the nodeset defining the subdomain and gives the base filename for VTK output over the subdomain corresponding to the ground surface (`step01-groundsurf.vtk`).

**pylithapp.timedependent.materials.elastic.output** Gives the base filename for state variable output files for the `elastic` material set (`step01-elastic.vtk`), and causes state variables to be averaged over all quadrature points in each cell.

**pylithapp.timedependent.materials.viscoelastic.output** Gives the base filename for state variable output files for the `viscoelastic` material set (`step01-viscoelastic.vtk`), and causes state variables to be averaged over all quadrature points in each cell.

The values for the Dirichlet boundary conditions are described in the file `fixeddisp_shear.spatialdb`, as specified in `step01.cfg`. The format of all spatial database files is similar. Because data are being specified using two control points (rather than being uniform over the mesh, for example), the data dimension is one.

The files containing common information (`tet4_1000m_binary.gmv`, `tet4_1000m_binary.pset`, `pylithapp.cfg`, `mat_elastic.spatialdb`, and `mat_viscoelastic.spatialdb`) along with the problem-specific files (`step01.cfg` and `fixeddisp_shear.spatialdb`) provide a complete description of the problem, and we can then run this example by typing

```
$ pylith step01.cfg
```

Once the problem has run, six files will be produced. The first file is named `step01_t0000000.vtk`. The `t0000000` indicates that the output is for the first (and only) time step, corresponding to an elastic solution. This file contains mesh information as well as displacement values at the mesh vertices. The second file is named `step01-statevars-elastic_t0000000.vtk`. This file contains the state variables for each cell in the material group `elastic`. The default fields are the total strain and stress fields. These values are computed at each quadrature point in the cell. We have requested that the values be averaged over all quadrature points for each cell; however, since we only have a single quadrature point for each linear tetrahedron, this will have no effect. The third file (`step01-statevars-viscoelastic_info.vtk`) gives the material properties used for the `viscoelastic` material set. Since we have not specified which properties to write, the default properties (`mu`, `lambda`, `density`) are written. There are two additional files containing the state variables for each of the material sets. The final file (`step01-groundsurf_t0000000.vtk`) is analogous to `step01_t0000000.vtk`, but in this case the results are only given for a subset of the mesh corresponding to the ground surface. Also, the cells in this file are one dimension lower than the cells described in `step01_t0000000.vtk`, so they are triangles rather than tetrahedra. All of the `vtk` files may be used with a number of visualization packages. If the problem ran correctly, you should be able to generate a figure such as Figure 7.20 on the next page, which was generated using ParaView.

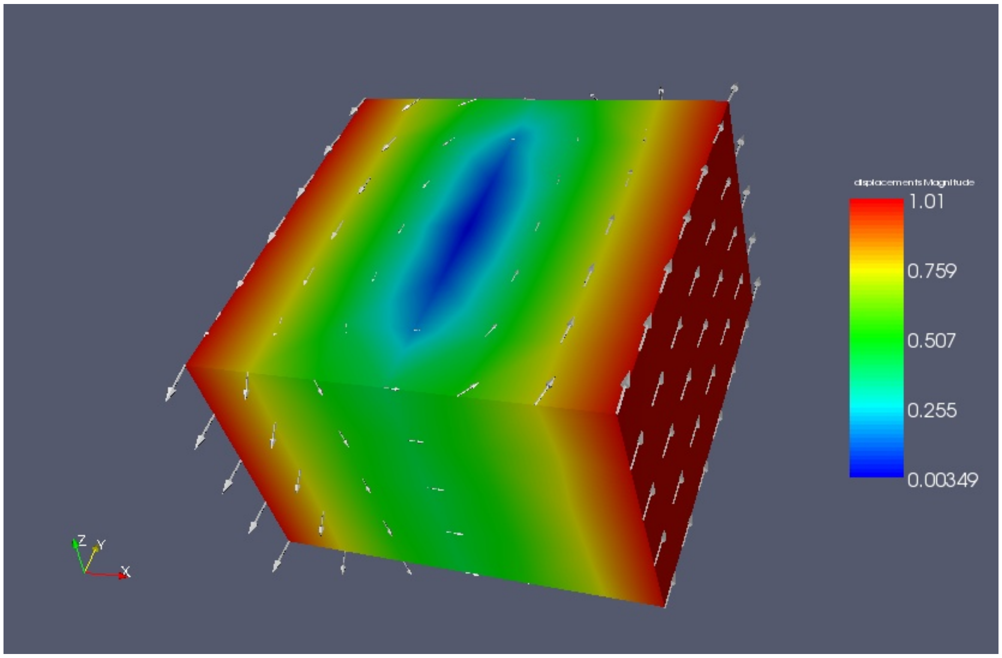



Figure 7.20: Color contours and vectors of displacement for the axial displacement example using a mesh composed of linear tetrahedral cells generated by LaGriT.

#### 7.8.4.1 Alternative Solver and Discretization Settings

Example `step01.cfg` uses the additive Schwarz preconditioner in conjunction with a classical Gram-Schmidt orthogonalization iterative solver. This preconditioner works reasonably well but the number of iterations generally scales with problem size. Even this small, simple problem requires 24 iterations. In this example (`step02.cfg`), we use a more sophisticated preconditioner that preconditions the degrees of freedom associated with the displacement field with an algebraic multigrid algorithm (see Section 4.1.5 on page 36 for details). Additionally, we illustrate the use of global uniform mesh refinement to increase the resolution of the solution by a factor of two. Because the mesh is refined in parallel after distribution, this technique can be used to run a larger problem than would be possible if the full resolution mesh had to be generated by the mesh generator. LaGriT runs only in serial and CUBIT has extremely limited parallel mesh generation capabilities. Table 7.2 shows the improved efficiency of the solver using the split fields with the algebraic multigrid preconditioner, especially as the problem size becomes larger. We have found similar results for other problems.

Table 7.2: Number of iterations in linear solve for the Shear Displacement and Kinematic Fault Slip problems discussed in this section. The preconditioner using split fields and an algebraic multigrid algorithm solves the linear system with fewer iterations with only a small to moderate increase as the problem size grows.

| Problem | Preconditioner | Refinement | # DOF | # Solve Iterations |
|---|---|---|---|---|
| Shear Displacement | additive Schwarz | none | 546 | 24 (step01) |
| | | 2x | 3890 | 47 |
| | split fields with algebraic multigrid | none | 546 | 13 |
| | | 2x | 3890 | 28 (step02) |
| Kinematic Fault Slip | additive Schwarz | none | 735 | 28 (step03) |
| | | 2x | 4527 | 63 |
| | split fields with algebraic multigrid | none | 735 | 28 |
| | | 2x | 4527 | 38 (step04) |

The field splitting and algebraic multigrid preconditioning are set up in `step02.cfg` with the following parameters:

```
[pylithapp.timedependent.formulation]
matrix_type = aij
```

```
[pylithapp.petsc]
pc_type = ml
```

The uniform global refinement requires changing just a single parameter:

```
[pylithapp.mesh_generator]
refiner = pylith.topology.RefineUniform
```

### 7.8.5 Kinematic Fault Slip Example

The next example problem is a right-lateral fault slip applied on the vertical fault defined by `x = 0`. The left and right sides of the mesh are fixed in the `x`, `y`, and `z` directions. Parameter settings that override or augment those in `pylithapp.cfg` are contained in the file `step03.cfg`. These settings are:

**pylithapp.timedependent** Specifies an implicit formulation for the problem, the array of boundary conditions, and the array of interfaces.

**pylithapp.timedependent.implicit** Specifies an array of two output managers, one for the full domain, and another for a subdomain corresponding to the ground surface.

**pylithapp.timedependent.bc.x_pos** Specifies the boundary conditions for the right side of the mesh, defining which degrees of freedom are being constrained (`x`, `y`, and `z`), providing the label (defined in `tet4_1000m_binary.ps` defining the points desired, and assigning a label to the boundary condition set. Rather than specifying a spatial database file to define the boundary conditions, we use the default spatial database (ZeroDispDB) for the Dirichlet boundary condition, which sets the displacements to zero.

**pylithapp.timedependent.bc.x_neg** Specifies the boundary conditions for the left side of the mesh, defining which degrees of freedom are being constrained (`x`, `y`, and `z`), providing the label (defined in `tet4_1000m_binary.pset`) defining the points desired, and assigning a label to the boundary condition set. Rather than specifying a spatial database file to define the boundary conditions, we use the default spatial database (ZeroDispDB) for the Dirichlet boundary condition, which sets the displacements to zero.

**pylithapp.timedependent.interfaces** Gives the label (defined in `tet4_1000m_binary.pset`) defining the points on the fault, provides quadrature information, and then gives database names for material properties (needed for conditioning), fault slip, peak fault slip rate, and fault slip time.

**pylithapp.problem.formulation.output.output.writer** Gives the base filename for VTK output over the entire domain (`step03.vtk`).

**pylithapp.problem.formulation.output.subdomain** Gives the label of the nodeset defining the subdomain and gives the base filename for VTK output over the subdomain corresponding to the ground surface (`step03-groundsurf.vtk`).

**pylithapp.timedependent.interfaces.fault.output.writer** Gives the base filename for cohesive cell output files (`step03-fault.vtk`).

**pylithapp.timedependent.materials.elastic.output** Gives the base filename for state variable output files for the `elastic` material set (`step03-statevars-elastic.vtk`), and causes state variables to be averaged over all quadrature points in each cell.

**pylithapp.timedependent.materials.viscoelastic.output** Gives the base filename for state variable output files for the `viscoelastic` material set (`step03-statevars-viscoelastic.vtk`), and causes state variables to be averaged over all quadrature points in each cell.

The fault example requires three additional database files that were not needed for the simple displacement example. The first file (`finalslip.spatialdb`) specifies a constant value of 2 m of right-lateral fault slip that then tapers linearly to zero from 2 km to 4 km depth, and a linearly-varying amount of reverse slip, with a maximum of 0.25 m at the surface, linearly tapering to 0 m at 2 km depth. The data dimension is one since the data vary linearly along a vertical line. The default slip time function is a step-function, so we also must provide the time at which slip begins. The elastic solution is associated with advancing from $t = -dt$ to $t = 0$, so we set the slip initiation time for the step-function to 0 in `dislocation_sliptime.spatialdb`.

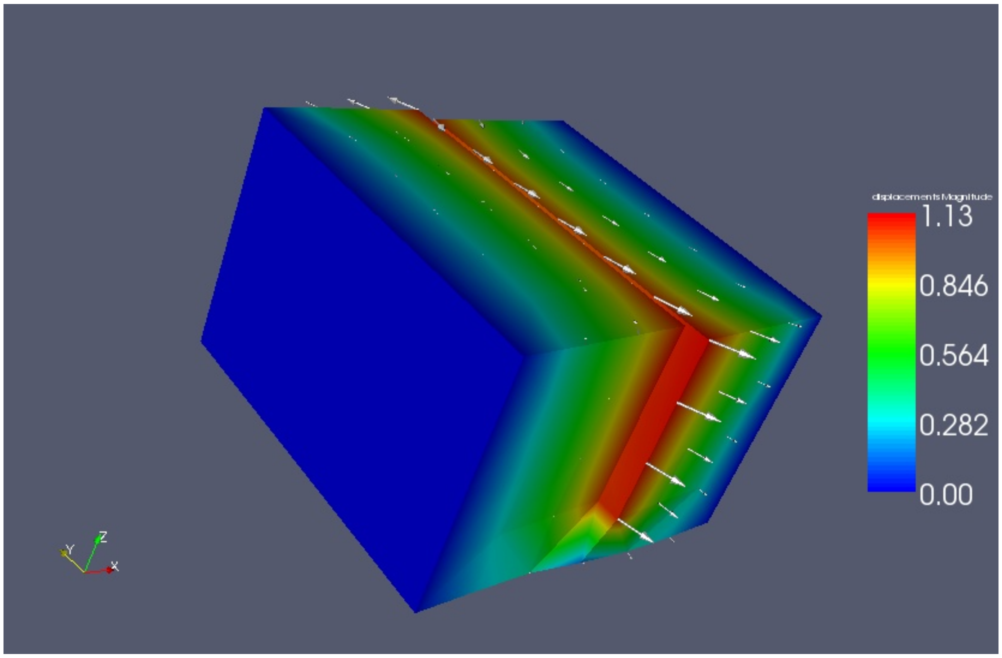



Figure 7.21: Color contours and vectors of displacement for the kinematic fault example using a mesh composed of linear tetrahedral cells generated by LaGriT.

The files containing common information (`tet4_1000m_binary.gmv`, `tet4_1000m_binary.pset`, `pylithapp.cfg`, `mat_elastic.spatialdb`, and `mat_viscoelastic.spatialdb`) along with the problem-specific files (`step03.cfg`, `finalslip.spatialdb`, and `sliptime.spatialdb`) provide a complete description of the problem, and we can then run this example by typing

```
$ pylith step03.cfg
```

Once the problem has run, eight files will be produced. The first file is named `step03_t0000000.vtk`. The `t0000000` indicates that the output is for the first (and only) time step, corresponding to an elastic solution. This file contains mesh information as well as displacement values at the mesh vertices. The second file is named `step03-statevars-elastic_t0000000.vtk`. This file contains the state variables for each cell in the material group `elastic`. The default fields are the total strain and stress fields. We have requested that the values be averaged over all quadrature points for each cell; however, since we only have a single quadrature point for each linear tetrahedron, this will have no effect. The third file (`step03-statevars-viscoelastic_info.vt` gives the material properties used for the `viscoelastic` material set. Since we have not specified which properties to write, the default properties (`mu`, `lambda`, `density`) are written. There are two additional files containing the state variables for each of the material sets. The file `step03-groundsurf_t0000000.vtk` is analogous to `step03_t0000000.vtk`, but in this case the results are only given for a subset of the mesh corresponding to the ground surface. Also, the cells in this file are one dimension lower than the cells described in `step03_t0000000.vtk`, so they are triangles rather than tetrahedra. The file `step03-fault_t0000000.vtk` gives the specified fault slip for each vertex on the fault, along with the computed traction change for the cohesive cell. The final file, `step03-fault_info.vtk`, provides information such as the normal direction, final slip, and slip time for each vertex on the fault. All of the `vtk` files may be used with a number of visualization packages. If the problem ran correctly, you should be able to generate a figure such as Figure 7.21, which was generated using ParaView.

#### 7.8.5.1 Alternative Solver and Discretization Settings

As we did for the Shear Dislocation examples, in `step04.cfg` we switch to using the split fields and algebraic multigrid preconditioner along with global uniform mesh refinement. Because PyLith implements fault slip using Lagrange multipliers, we make a few small adjusments to the solver settings. As discussed in Section 4.1.5 on page 36, we use a custom preconditioner for the Lagrange multiplier degrees of freedom when preconditioning with field splitting. Within `step04.cfg` we turn on the use of the custom preconditioner for the Lagrange multiplier degrees of freedom and add the corresponding settings for the fourth field for the algebraic multigrid algorithm,

```
[pylithapp.timedependent.formulation]
split_fields = True
use_custom_constraint_pc = True
matrix_type = aij

[pylithapp.petsc]
fs_pc_type = fieldsplit
fs_pc_use_amat = true
fs_pc_fieldsplit_type = multiplicative
fs_fieldsplit_displacement_pc_type = ml
fs_fieldsplit_lagrange_multiplier_pc_type = jacobi
fs_fieldsplit_displacement_ksp_type = preonly
fs_fieldsplit_lagrange_multiplier_ksp_type = preonly
```

Table 7.2 on page 136 shows the improved efficiency of the solver using the split fields with the algebraic multigrid preconditioner.

## 7.9 Examples Using Hexahedral Mesh Created by CUBIT/Trelis

PyLith features discussed in this set of examples:

- Static solution
- Quasi-static solution
- CUBIT/Trelis mesh format
- Trilinear hexahedral cells
- VTK output
- HDF5 output
- Dirichlet displacement and velocity boundary conditions
- Neumann traction boundary conditions and time-varying tractions
- ZeroDispDB spatial database
- SimpleDB spatial database
- UniformDB spatial database
- Static fault rupture
- Multiple kinematic fault ruptures
- Specifying more than one material
- Nonlinear solver
- Linearly elastic isotropic material
- Maxwell linear viscoelastic material
- Generalized Maxwell linear viscoelastic material
- Power-law viscoelastic material
- Drucker-Prager elastoplastic material
- Adaptive time stepping
- Static fault friction
- Slip-weakening fault friction
- Rate-and-state fault friction
- Gravitational body forces
- Initial stresses
- Finite strain

All of the files necessary to run the examples are contained in the directory `examples/3d/hex8`.

### 7.9.1 Overview

This example is meant to demonstrate most of the important features of PyLith as a quasi-static finite-element code, using a sequence of example problems. All problems use the same 3D hexahedral mesh generated using CUBIT/Trelis (CUBIT is available to employees of the United States government through `cubit.sandia.gov` and Trelis licenses are available through `www.csimsoft.com/trelis`). Each example builds on the previous examples, as we demonstrate new features. As in the other examples, the files include extensive comments. We start with the generation of the mesh, which is composed of 144 hexahedra. Following the discussion of how to generate the mesh, we discuss the `pylithapp.cfg` file, which contains information common to all the simulations. We group the examples into four sections, each pertaining to a particular set of PyLith features. We suggest users go through each of these sections in order as the complexity increases at each step.

### 7.9.2 Mesh Generation and Description

The mesh for these examples is a simple rectangular solid (Figure 7.22 on the next page). Although it would be possible to generate this mesh by hand, we use this example to illustrate the use of CUBIT/Trelis for mesh generation. We provide documented journal files in `examples/3d/hex8/mesh`. Dissection of these journal files should provide some insight into how to use CUBIT/Trelis with PyLith. For more detailed information on using CUBIT/Trelis, refer to the CUBIT/Trelis documentation. If you have CUBIT/Trelis installed on your machine, you can use the journal files to create your own mesh. Otherwise, you can use the mesh that has already been created.

If you are using CUBIT/Trelis to generate your own mesh, there are two ways to use the journal files. The simplest method is to go to the Tools menu, select Play Journal File, and then select the file `mesh_hex8_1000m.jou`. This will run the commands in that file as well as the commands in `geometry.jou`, which is referenced from `mesh_hex8_1000m.jou`. Prior to doing this, you should set your directory to the one containing the journal files. This method will create the mesh, but you will gain very little insight into what is being done. A more informative approach is to input each journal command into the CUBIT command window directly. That way, you will see what each command does. The first command in `mesh_hex8_1000m.jou` is playback geometry.jou, so you should start with the commands in `geometry.jou`. The first three commands, which define the block shape, are

```
reset
brick x 6000 y 6000 z 4000
volume 1 move x 0 y 0 z -2000
```

Continuing through the remainder of the commands in `geometry.jou`, and then using the additional commands in `mesh_hex8_1000m.j` you will eventually end up with the file `box_hex8_1000m.exo`, which contains all of the mesh information. This information is similar to that included in PyLith mesh ASCII format, but the information is contained in an Exodus file, which is a specialized netCDF file. If you have the `ncdump` command available, you can see what is in the file by typing:

```
$ ncdump box_hex8_1000m.exo
```

### 7.9.3 Additional Common Information

In addition to the mesh, the example problems share other information. As in previous examples, we place this information in `pylithapp.cfg`. Since these examples use a mesh from CUBIT, in this file we set the importer to MeshIOCubit:

Excerpt from `pylithapp.cfg`

```
[pylithapp.mesh_generator]
reader = pylith.meshio.MeshIOCubit

[pylithapp.mesh_generator.reader]
filename = mesh/box_hex8_1000m.exo
```

This example differs from some earlier examples, because we specify two material groups:

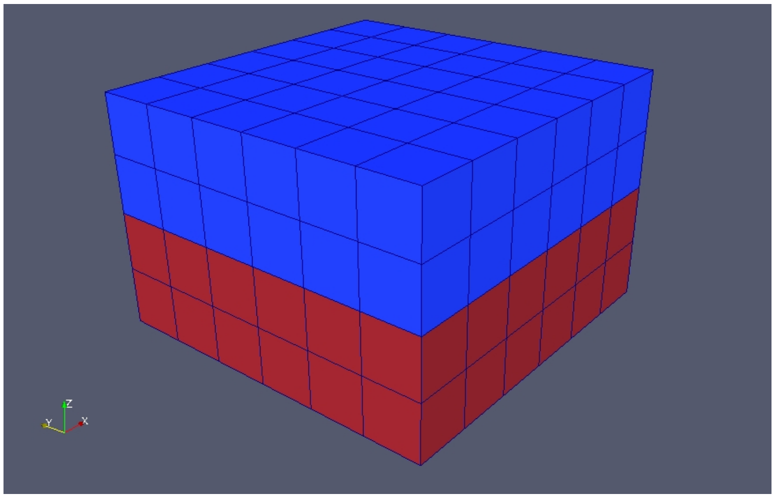



Figure 7.22: Mesh composed of trilinear hexahedral cells generated by CUBIT used for the suite of example problems. The different colors represent the two different materials.

Excerpt from `pylithapp.cfg`

```
[pylithapp.timedependent]
materials = [upper_crust, lower_crust]</h>

[pylithapp.timedependent.materials.upper_crust]
label = Upper crust material
id = 1
db.iohandler.filename = spatialdb/mat_elastic.spatialdb
quadrature.cell = pylith.feassemble.FIATLagrange
quadrature.cell.dimension = 3

[pylithapp.timedependent.materials.lower_crust]
label = Lower crust material
id = 2
db.iohandler.filename = spatialdb/mat\_elastic.spatialdb
quadrature.cell = pylith.feassemble.FIATLagrange
quadrature.cell.dimension = 3
```

The two material groups correspond to the two different colored regions in Figure 7.22. Using two material groups allows us to specify different material types or material variations for the upper crust and lower crust, if desired. For now, we retain the default ElasticIsotropic3D material type for both materials. This behavior will be overridden by example-specific `cfg` files in some of the examples. Although the material groups are specified in `pylithapp.cfg`, the physical properties for the material models are given in `spatialdb/ mat_elastic.spatialdb`. This spatial database provides values at a single point, resulting in uniform properties within the material.

### 7.9.4 Example Problems

The example problems are divided into categories that roughly correspond to simple static problems, quasi-static problems, problems involving fault friction, and problems where gravity is used. For the most part, each successive example involves just adding or changing a few parameters from the previous example. For this reason, it is advisable to go through each example in order, starting with the simplest (static problems).

### 7.9.5 Static Examples

PyLith features discussed in this example:

- Static solution
- VTK output
- Dirichlet displacement boundary conditions
- Neumann traction boundary conditions
- ZeroDispDB spatial database
- SimpleDB spatial database
- UniformDB spatial database
- Static fault rupture
- Specifying more than one material
- Linearly elastic isotropic material

#### 7.9.5.1 Overview

This set of examples describe the simplest class of problems for PyLith. The problems are all purely elastic, and there is no time-dependence. This set of elastostatic examples primarily demonstrates the application of different types of boundary conditions in PyLith, as well as demonstrating the use of a kinematic fault for a static problem. All of the examples are contained in the directory `examples/3d/hex8`, and the corresponding `cfg` files are `step01.cfg`, `step02.cfg`, and `step03.cfg`. Run the examples as follows:

```
# Step01
$ pylith step01.cfg

# Step02
$ pylith step02.cfg

# Step03
$ pylith step03.cfg
```

This will cause PyLith to read the default parameters in `pylithapp.cfg`, and then override or augment them with the additional parameters in the `stepXX.cfg` file. Each `cfg` file is extensively documented to provide detailed information on the various parameters.

#### 7.9.5.2 Step01 - Pure Dirichlet Boundary Conditions

The `step01.cfg` file defines a problem with pure Dirichlet (displacement) boundary conditions corresponding to compression in the x-direction and shear in the y-direction. The bottom (minimum z) boundary is held fixed in the z-direction. On the positive and negative x-faces, compressional displacements of 1 m are applied in the x-direction and shear displacements yielding a left-lateral sense of shear are applied in the y-direction. In this example and in subsequent examples we would like to output the displacement solution over a subset of the domain corresponding to the ground surface.

Excerpt from `step01.cfg`

```
[pylithapp.timedependent.implicit]
# Set the output to an array of 2 output managers.
# We will output the solution over the domain and the ground surface.
output = [domain,subdomain]

# Set subdomain component to OutputSolnSubset (boundary of the domain).
output.subdomain = pylith.meshio.OutputSolnSubset

# Give basename for VTK domain output of solution over ground surface.
[pylithapp.problem.formulation.output.subdomain]
# Name of nodeset for ground surface.
label = face_zpos
writer.filename = output/step01-groundsurf.vtk
```

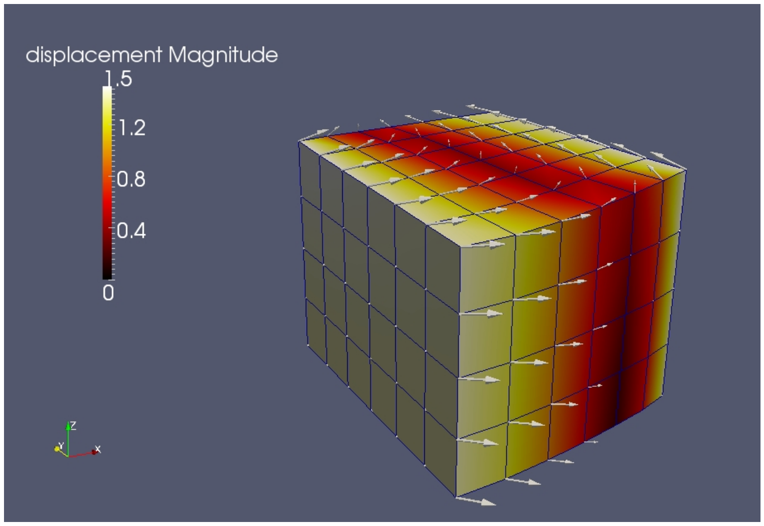



Figure 7.23: Displacement field for example step01 visualized using ParaView. The mesh has been distorted by the computed displacements (magnified by 500), and the vectors show the computed displacements.

For the boundary conditions, we must describe which degrees of freedom are being constrained (**bc_dof**), we must provide a the label associated with the CUBIT/Trelis nodeset associated with the BC, and we must specify the type of spatial database is being used to describe the boundary conditions. For the x-faces, we use a SimpleDB to provide the displacements on the x-faces:

Excerpt from `step01.cfg`

```
# Boundary condition on +x face
[pylithapp.timedependent.bc.x_pos]
bc_dof = [0, 1]
label = face_xpos
db_initial = spatialdata.spatialdb.SimpleDB
db_initial.label = Dirichlet BC on +x
db_initial.iohandler.filename = spatialdb/fixeddisp_axial_shear.spatialdb

# Boundary condition on -x face
[pylithapp.timedependent.bc.x_neg]
bc_dof = [0, 1]
label = face_xneg
db_initial = spatialdata.spatialdb.SimpleDB
db_initial.label = Dirichlet BC on -x
db_initial.iohandler.filename = spatialdb/fixeddisp_axial_shear.spatialdb
```

For a SimpleDB, we must provide a filename. The default spatial database for **db_initial** is ZeroDispBC, which automatically applies zero displacements to all vertices in the nodeset, and no filename is required (or needed).

Excerpt from `step01.cfg`

```
# Boundary condition on -z face
[pylithapp.timedependent.bc.z_neg]
bc_dof = [2]
label = face_zneg
db_initial.label = Dirichlet BC on -z
```

When we have run the simulation, the output VTK files will be contained in `examples/3d/hex8/output` (all with a prefix of `step01`). Results using ParaView are shown in Figure 7.23.

#### 7.9.5.3 Step02 - Dirichlet and Neumann Boundary Conditions

The `step02.cfg` file defines a problem with Dirichlet (displacement) boundary conditions corresponding to zero x and y-displacements applied on the negative x-face and Neumann (traction) boundary conditions corresponding to normal compression and horizontal shear applied on the positive x-face. The bottom (negative z) boundary is held fixed in the z-direction. The problem is similar to example step01, except that 1 MPa of normal compression and 1 MPa of shear (in a left-lateral sense) are applied on the positive x-face, and the negative x-face is pinned in both the x and y-directions.

For the boundary conditions, we must first change the boundary condition type for the positive x-face from the default Dirichlet to Neumann:

Excerpt from `step02.cfg`

```
# +x face -- first change bc type to Neumann
[pylithapp.timedependent.bc]
x_pos = pylith.bc.Neumann
```

We use a SimpleDB to describe the traction boundary conditions. When applying traction boundary conditions over a surface, it is also necessary to specify integration information for the surface. Since this is a three-dimensional problem, the dimension of the surface is 2. Since the cells being used are trilinear hexahedra, the cell type is FIATLagrange and we use an integration order of 2. A lower integration order would not provide sufficient accuracy while a higher integration order would offer no benefit (while requiring more computation time and storage):

Excerpt from `step02.cfg`

```
# Boundary condition on +x face
[pylithapp.timedependent.bc.x_pos]
label = face_xpos
db_initial = spatialdata.spatialdb.SimpleDB
db_initial.label = Neumann BC on +x
db_initial.iohandler.filename = spatialdb/tractions_axial_shear.spatialdb

# We must specify quadrature information for the cell faces.
quadrature.cell = pylith.feassemble.FIATLagrange
quadrature.cell.dimension = 2
quadrature.cell.quad_order = 2
```

The boundary conditions on the negative x-face are simpler than they were in example step01 (zero displacements in the x and y-directions), so we can use the default ZeroDispBC:

Excerpt from `step02.cfg`

```
# Boundary condition on -x face
[pylithapp.timedependent.bc.x_neg]
bc_dof = [0, 1]
label = face_xneg
db_initial.label = Dirichlet BC on -x
```

The boundary conditions on the negative z-face are supplied in the same manner as for example step01. When we have run the simulation, the output VTK files will be contained in `examples/3d/hex8/output` (all with a prefix of `step02`). Results using ParaView are shown in Figure 7.24 on the facing page.

#### 7.9.5.4 Step03 - Dirichlet Boundary Conditions with Kinematic Fault Slip

The `step03.cfg` file describes a problem with Dirichlet (displacement) boundary conditions corresponding to zero x and y-displacements applied on the negative and positive x-faces and a vertical fault with a combination of left-lateral and updip motion. The left-lateral component of fault slip has a constant value of 2 m in the upper crust, and then decreases linearly to zero at the base of the model. The reverse slip component has a value of 0.25 m at the surface, and then decreases linearly to zero at 2 km depth.

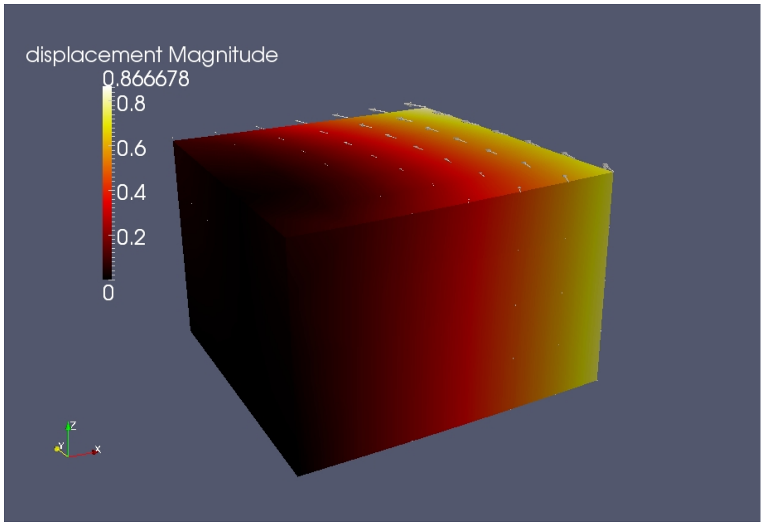



Figure 7.24: Displacement field for example step02 visualized using ParaView. The mesh has been distorted by the computed displacements (magnified by 500), and the vectors show the computed displacements.

Due to the simplicity of the boundary conditions, we are able to use the default ZeroDispBC for the positive and negative x-faces, as well as the negative z-face. To use a fault, we must first define a fault interface. We do this by providing an array containing a single interface. For this example we specify the fault slip, so we set the interface type to FaultCohesiveKin.

Excerpt from `step03.cfg`

```
[pylithapp.timedependent]
# Set interfaces to an array of 1 fault: 'fault'.
interfaces = [fault]

# Set the type of fault interface condition.
[pylithapp.timedependent.interfaces]
fault = pylith.faults.FaultCohesiveKin

[pylithapp.timedependent.interfaces.fault]
# The label corresponds to the name of the nodeset in CUBIT/Trelis.
label = fault

# We must define the quadrature information for fault cells.
# The fault cells are 2D (surface).
quadrature.cell = pylith.feassemble.FIATLagrange
quadrature.cell.dimension = 2
```

We retain the default StepSlipFn since we want static fault slip. Finally, we use one SimpleDB to define the spatial variation of fault slip, and another SimpleDB to define the spatial variation in slip initiation times (the start time is 0.0 everywhere since this is a static problem):

Excerpt from `step03.cfg`

```
# The slip time and final slip are defined in spatial databases.
[pylithapp.timedependent.interfaces.fault.eq\_srcs.rupture.slip\_function]
slip.iohandler.filename = spatialdb/finalslip.spatialdb
slip.query_type = linear
slip_time.iohandler.filename = spatialdb/sliptime.spatialdb

# Fault output, give the basename for the VTK file.
[pylithapp.problem.interfaces.fault.output]
writer.filename = output/step03-fault.vtk
```

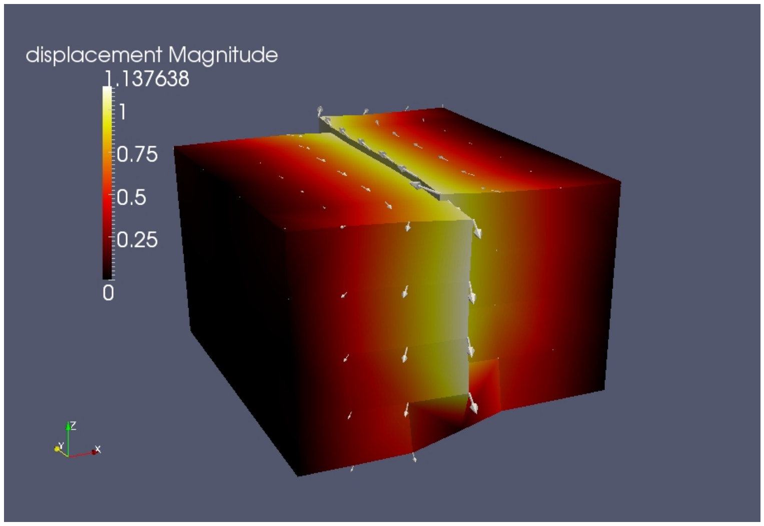



Figure 7.25: Displacement field for example step03 visualized using ParaView. The mesh has been distorted by the computed displacements (magnified by 500), and the vectors show the computed displacements.

This will result in two extra files being produced. The first file (`step03-fault_info.vtk`) contains information such as the normal directions to the fault surface, the applied fault slip, and the fault slip times. The second file (`step03-fault_t0000000.vtk`) contains the cumulative fault slip for the time step and the change in tractions on the fault surface due to the slip. When we have run the simulation, the output VTK files will be contained in `examples/3d/hex8/output` (all with a prefix of `step03`). Results using ParaView are shown in Figure 7.25.

### 7.9.6 Quasi-Static Examples

PyLith features discussed in this example:

- Quasi-static solution
- Formatting timestamps of VTK output files
- HDF5 output
- Output of velocity field
- Dirichlet displacement and velocity boundary conditions
- Neumann traction boundary conditions and time-varying tractions
- UniformDB spatial database
- CompositeDB spatial database
- Quasi-static fault rupture and fault creep
- Multiple kinematic fault ruptures
- Specifying more than one material
- Nonlinear solver
- Maxwell linear viscoelastic material
- Power-law viscoelastic material
- Drucker-Prager elastoplastic material
- Adaptive time stepping

#### 7.9.6.1 Overview

This set of examples describes a set of quasi-static problems for PyLith. These quasi-static problems primarily demonstrate the usage of time-dependent boundary conditions and fault slip, as well as different rheologies. Some of the examples also demonstrate the usage of the nonlinear solver, which is required by the nonlinear rheologies (power-law viscoelastic and Drucker-Prager elastoplastic). Some of the examples also demonstrate the usage of HDF5 output, which is an alternative to the default VTK output. All of the examples are contained in the directory `examples/3d/hex8`, and the corresponding `cfg` files are `step04.cfg`, `step05.cfg`, `step06.cfg`, `step07.cfg`, `step08.cfg`, and `step09.cfg`. Run the examples as follows:

```
# Step04
$ pylith step04.cfg

# Step05
$ pylith step05.cfg

# Step06
$ pylith step06.cfg

# Step07
$ pylith step07.cfg

# Step08
$ pylith step08.cfg

# Step09
$ pylith step09.cfg
```

This will cause PyLith to read the default parameters in `pylithapp.cfg`, and then override or augment them with the additional parameters in the `stepXX.cfg` file. Each `cfg` file is extensively documented, to provide detailed information on the various parameters.

#### 7.9.6.2 Step04 - Pure Dirichlet Velocity Boundary Conditions

The `step04.cfg` file defines a problem with x-displacements fixed at zero on the positive and negative x-faces while velocity boundary conditions are applied in the y-directions on the same faces, yielding a left-lateral sense of movement. The bottom (negative z) boundary is held fixed in the z-direction. We also use a Maxwell viscoelastic material for the lower crust, and the simulation is run for 200 years using a constant time-step size of 20 years. The default time stepping behavior is TimeStepUniform. We retain that behavior for this problem and provide the total simulation time and the time-step size:

Excerpt from `step04.cfg`

```
# Change the total simulation time to 200 years, and use a constant time
# step size of 20 years.
[pylithapp.timedependent.implicit.time_step]<.h>
<p>total_time</p> = 200.0*year
<p>dt</p> = 20.0*year
```

We then change the material type of the lower crust, provide a spatial database from which to obtain the material properties (using the default SimpleDB), and request additional output information for the material:

Excerpt from `step04.cfg`

```
# Change material type of lower crust to Maxwell viscoelastic.
[pylithapp.timedependent]
materials.lower_crust = pylith.materials.MaxwellIsotropic3D

# Provide a spatial database from which to obtain property values.
# Since there are additional properties and state variables for the Maxwell
# model, we explicitly request that they be output. Properties are named in
# cell_info_fields and state variables are named in cell_data_fields.
[pylithapp.timedependent.materials.lower_crust]
db_properties.iohandler.filename = spatialdb/mat_maxwell.spatialdb
output.cell_info_fields = [density, mu, lambda, maxwell_time]
output.cell_data_fields = [total_strain, stress, viscous_strain]
```

Note that the default **output.cell_info_fields** are those corresponding to an elastic material (`density`, `mu`, `lambda`), and the default **output.cell_data_fields** are `total_strain` and `stress`. For materials other than elastic, there are generally additional material properties and state variables, and the appropriate additional fields must be specifically requested for each material type.

This example has no displacements in the elastic solution (t = 0), so we retain the default ZeroDispDB for all instances of **db_initial**. To apply the velocity boundary conditions, we must specify **db_rate**, which is zero by default. We use a UniformDB to assign the velocities:

Excerpt from `step04.cfg`

```
# Boundary condition on +x face
[pylithapp.timedependent.bc.x_pos]
bc_dof = [0, 1]
label = face_xpos
db_initial.label = Dirichlet BC on +x
db_rate = spatialdata.spatialdb.UniformDB
db_rate.label = Dirichlet rate BC on +x
db_rate.values = [displacement-rate-x, displacement-rate-y, rate-start-time]
db_rate.data = [0.0*cm/year, 1.0*cm/year, 0.0*year]

# Boundary condition on -x face
[pylithapp.timedependent.bc.x_neg]
bc_dof = [0, 1]
label = face_xneg
db_initial.label = Dirichlet BC on -x
db_rate = spatialdata.spatialdb.UniformDB
db_rate.label = Dirichlet rate BC on +x
db_rate.values = [displacement-rate-x, displacement-rate-y, rate-start-time]
db_rate.data = [0.0*cm/year, -1.0*cm/year, 0.0*year]
```

Note that **db_rate** requires a start time, which allows the condition to be applied at any time during the simulation. For this example, we start the velocity boundary conditions at t = 0.

Finally, we must provide information on VTK output. This is slightly more complicated than the static case, because we must decide the frequency with which output occurs for each output manager. We also assign a more user-friendly format to the output file time stamp, and we request that the time stamp is in units of 1 year (rather than the default value of seconds):

Excerpt from `step04.cfg`

```
# Give basename for VTK domain output of solution over domain.
[pylithapp.problem.formulation.output.domain]
# We specify that output occurs in terms of a given time frequency, and
# ask for output every 40 years. The time stamps of the output files are
# in years (rather than the default of seconds), and we give a format for
# the time stamp.
output_freq = time_step
time_step = 40.0*year
writer.filename = output/step04.vtk
writer.time_format = \%04.0f
writer.time_constant = 1.0*year

# Give basename for VTK domain output of solution over ground surface.
[pylithapp.problem.formulation.output.subdomain]
label = face_zpos ; Name of nodeset for ground surface
# We keep the default output frequency behavior (skip every n steps), and
# ask to skip 0 steps between output, so that we get output every time step.
skip = 0
writer.filename = output/step04-groundsurf.vtk
writer.time_format = %04.0f
writer.time_constant = 1.0*year
```

We provide similar output information for the two materials (**upper_crust** and **lower_crust**). Note that for the domain

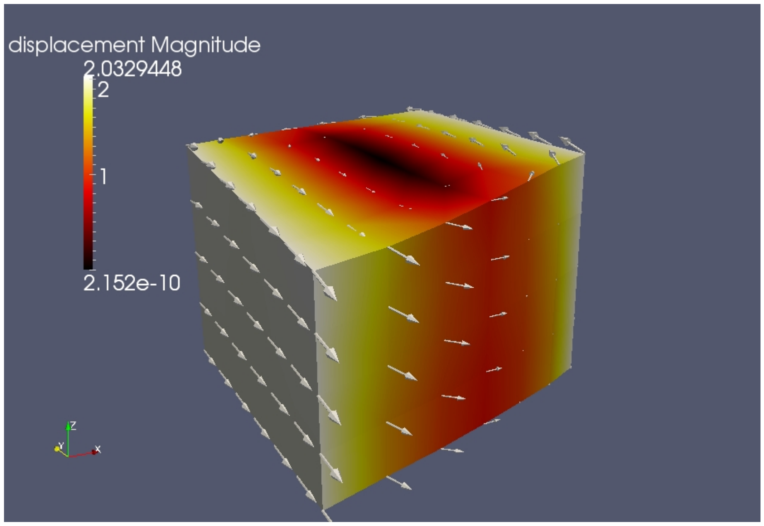



Figure 7.26: Displacement field for example step04 at t = 200 years visualized using ParaView. The mesh has been distorted by the computed displacements (magnified by 500), and the vectors show the computed displacements.

output, we requested output in terms of a given time frequency, while for the subdomain we requested output in terms of number of time steps. When we have run the simulation, the output VTK files will be contained in `examples/3d/hex8/output` (all with a prefix of `step04`). Results using ParaView are shown in Figure 7.26.

#### 7.9.6.3 Step05 - Time-Varying Dirichlet and Neumann Boundary Conditions

The `step05.cfg` file describes a problem with time-varying Dirichlet and Neumann boundary conditions. The example is similar to example step04, with a few important differences:

- The Dirichlet boundary conditions on the negative x-face include an initial displacement (applied in the elastic solution), as well as a constant velocity.
- Neumann (traction) boundary conditions are applied in the negative x-direction on the positive x-face, giving a compressive stress. An initial traction is applied in the elastic solution, and then at t = 100 years it begins decreasing linearly until it reaches zero at the end of the simulation (t = 200 years).

We again use a Maxwell viscoelastic material for the lower crust.

For the boundary conditions, we must first change the boundary condition type for the positive x-face from the default Dirichlet to Neumann:

Excerpt from `step05.cfg`

```
# +x face -- first change bc type to Neumann
[pylithapp.timedependent.bc]
x_pos = pylith.bc.Neumann
```

We provide quadrature information for this face as we did for example step02. We then use a UniformDB for both the initial tractions as well as the traction rates. We provide a start time of 100 years for the traction rates, and use a rate of 0.01 MPa/year, so that by the end of 200 years we have completely cancelled the initial traction of -1 MPa:

Excerpt from `step05.cfg`

```
[pylithapp.timedependent.bc.x_pos]
# First specify a UniformDB for the initial tractions, along with the values.
db_initial = spatialdata.spatialdb.UniformDB
db_initial.label = Neumann BC on +x
```

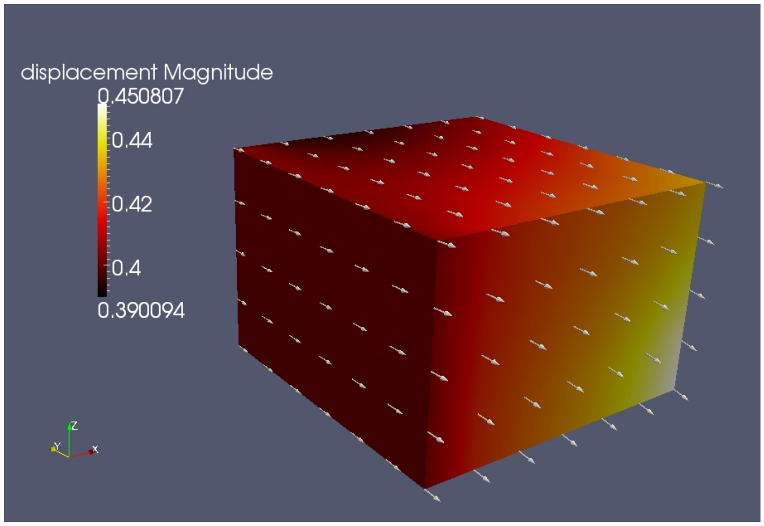



Figure 7.27: Displacement field for example step05 at t = 40 years visualized using ParaView. The mesh has been distorted by the computed displacements (magnified by 500), and the vectors show the computed displacements.

```
db_initial.values = [traction-shear-horiz, traction-shear-vert, traction-normal]
db_initial.data = [0.0*MPa, 0.0*MPa, -1.0*MPa]

# Provide information on traction rates.
db_rate = spatialdata.spatialdb.UniformDB
db_rate.label = Neumann rate BC on +x
db_rate.values = [traction-rate-shear-horiz, traction-rate-shear-vert, traction-rate-normal,rate-start-t
db_rate.data = [0.0*MPa/year, 0.0*MPa/year, 0.01*MPa/year, 100.0*year]
```

The boundary conditions on the negative x-face are analogous, but we are instead using Dirichlet boundary conditions, and the initial displacement is in the same direction as the applied velocities:

Excerpt from `step05.cfg`

```
# -x face
[pylithapp.timedependent.bc.x_neg]
bc_dof<p> = [0, 1]
<p>label = face_xneg

# Initial displacements.
db_initial = spatialdata.spatialdb.UniformDB
db_initial.label = Dirichlet BC on -x
db_initial.values = [displacement-x, displacement-y]
db_initial.data = [0.0*cm, -0.5*cm]

# Velocities.
db_rate = spatialdata.spatialdb.UniformDB
db_rate.label = Dirichlet rate BC on -x
db_rate.values = [displacement-rate-x,displacement-rate-y,rate-start-time]
db_rate.data = [0.0*cm/year, -1.0*cm/year, 0.0*year]
```

The boundary conditions on the negative z-face are supplied in the same manner as for example step04. When we have run the simulation, the output VTK files will be contained in `examples/3d/hex8/output` (all with a prefix of `step05`). Results using ParaView are shown in Figure 7.27.

#### 7.9.6.4 Step06 - Dirichlet Boundary Conditions with Time-Dependent Kinematic Fault Slip

The `step06.cfg` file defines a problem with Dirichlet (displacement) boundary conditions corresponding to zero x- and y-displacements applied on the negative and positive x-faces and a vertical fault that includes multiple earthquake ruptures as well as steady fault creep. The upper (locked) portion of the fault has 4 m of left-lateral slip every 200 years, while the lower (creeping) portion of the fault slips at a steady rate of 2 cm/year. The problem bears some similarity to the strike-slip fault model of Savage and Prescott [Savage and Prescott, 1978], except that the fault creep extends through the viscoelastic portion of the domain, and the far-field displacement boundary conditions are held fixed.

In this example and the remainder of the examples in this section, we change the time stepping behavior from the default TimeStepUniform to TimeStepAdapt. For adaptive time stepping, we provide the maximum permissible time-step size, along with a stability factor. The stability factor controls the time-step size relative to the stable time-step size provided by the different materials in the model. A **stability_factor** of 1.0 means we should use the stable time-step size, while a **stability_factor** greater than 1.0 means we want to use a smaller time-step size. A **stability_factor** less than 1.0 allows time-step sizes greater than the stable time-step size, which may provide inaccurate results. The adaptive time stepping information is provided as:

Excerpt from `step06.cfg`

```
# Change time stepping algorithm from uniform time step, to adaptive
# time stepping.
time_step = pylith.problems.TimeStepAdapt

# Change the total simulation time to 700 years, and set the maximum time
# step size to 10 years.
[pylithapp.timedependent.implicit.time_step]
total_time = 700.0*year
max_dt = 10.0*year
stability_factor = 1.0 ; use time step equal to stable value from materials
```

In this example and the remainder of the examples in this section, we also make use of HDF5 output rather than the default VTK output. HDF5 output is a new feature beginning with PyLith version 1.6, and it is much more efficient with the additional advantage that multiple time steps can be contained in a single file. PyLith also produces Xdmf files describing the contents of the HDF5 files, which allows the files to be read easily by applications such as ParaView. Since VTK output is still the default, we must change the value from the default. Also note that the filename suffix is `h5`:

Excerpt from `step06.cfg`

```
# Give basename for output of solution over domain.
[pylithapp.problem.formulation.output.domain]
# We specify that output occurs in terms of a given time frequency, and
# ask for output every 50 years.
output_freq = time_step
time_step = 50.0*year

# We are using HDF5 output so we must change the default writer.
writer = pylith.meshio.DataWriterHDF5
writer.filename = output/step06.h5
```

Note that we no longer need the **writer.time_format** or **writer.time_constant** properties, since all time steps are contained in a single file. The HDF5 writer does not have these properties, so if we attempt to define them an error will result.

We also set the writer for other output as well, since it is not the default. For subdomain output we use:

Excerpt from `step06.cfg`

```
# Give basename for output of solution over ground surface.
[pylithapp.problem.formulation.output.subdomain]
# Name of nodeset for ground surface.
label = face_zpos
```

```
# We keep the default output frequency behavior (skip every n steps), and
# ask to skip 0 steps between output, so that we get output every time step.
# We again switch the writer to produce HDF5 output.
skip = 0
writer = pylith.meshio.DataWriterHDF5
writer.filename = output/step06-groundsurf.h5

# Fault output
[pylithapp.problem.interfaces.fault.output]
# We keep the default output frequency behavior (skip every n steps), and
# ask to skip 0 steps between output, so that we get output every time step.
# We again switch the writer to produce HDF5 output.
skip = 0
writer = pylith.meshio.DataWriterHDF5
writer.filename = output/step06-fault.h5
```

Due to the simplicity of the boundary conditions, we are able to use the default ZeroDispBC for the positive and negative x-faces, as well as the negative z-face. As for example step03, we define a fault interface, we identify the nodeset corresponding to the fault, and we provide quadrature information for the fault. We then define an array of earthquake sources and provide an origin time for each:

Excerpt from `step06.cfg`

```
[pylithapp.timedependent.interfaces.fault]
# Set earthquake sources to an array consisting of creep and 3 ruptures.
eq_srcs = [creep, one, two, three]
eq_srcs.creep.origin_time = 00.0*year
eq_srcs.one.origin_time = 200.0*year
eq_srcs.two.origin_time = 400.0*year
eq_srcs.three.origin_time = 600.0*year
```

Note that the creep begins at t = 0 years, while the ruptures (**one**, **two**, **three**) occur at regular intervals of 200 years. We retain the default StepSlipFn for the ruptures. Each of the ruptures has the same amount of slip, and slip occurs simultaneously for the entire rupture region, so we can use the same SimpleDB files providing slip and slip time for each rupture:

Excerpt from `step06.cfg`

```
# Define slip and origin time for first rupture.
[pylithapp.timedependent.interfaces.fault.eq_srcs.one.slip_function]
slip.iohandler.filename = spatialdb/finalslip_rupture.spatialdb
slip_time.iohandler.filename = spatialdb/sliptime.spatialdb

# Define slip and origin time for second rupture.
[pylithapp.timedependent.interfaces.fault.eq_srcs.two.slip_function]
slip.iohandler.filename = spatialdb/finalslip_rupture.spatialdb
slip_time.iohandler.filename = spatialdb/sliptime.spatialdb

# Define slip and origin time for third rupture.
[pylithapp.timedependent.interfaces.fault.eq_srcs.three.slip_function]
slip.iohandler.filename = spatialdb/finalslip_rupture.spatialdb
slip_time.iohandler.filename = spatialdb/sliptime.spatialdb
```

For the creep source, we change the slip function to ConstRateSlipFn, and we use a SimpleDB for both the slip time and the slip rate:

Excerpt from `step06.cfg`

```
# Define slip rate and origin time for fault creep.
[pylithapp.timedependent.interfaces.fault.eq_srcs.creep]
slip_function = pylith.faults.ConstRateSlipFn
slip_function.slip_rate.iohandler.filename = spatialdb/sliprate_creep.spatialdb
slip_function.slip_time.iohandler.filename = spatialdb/sliptime.spatialdb
```

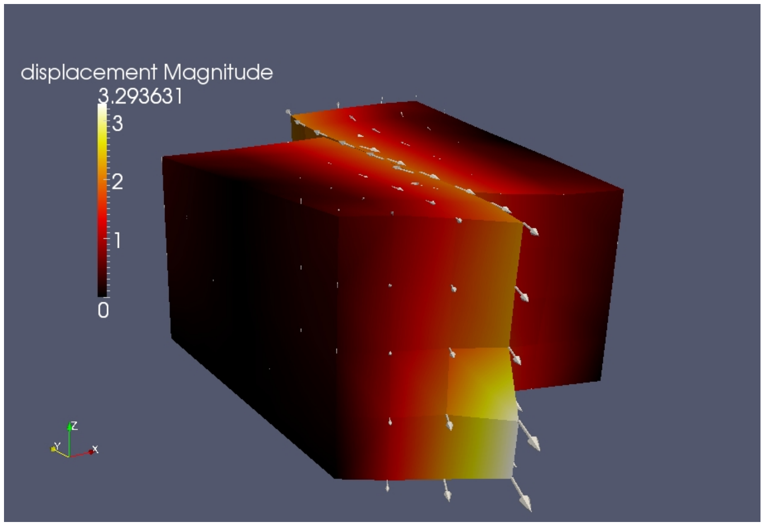



Figure 7.28: Displacement field for example step06 at t = 300 years visualized using ParaView. The mesh has been distorted by the computed displacements (magnified by 500), and the vectors show the computed displacements.

For all earthquake sources we provide both an **origin_time** and a **slip_function.slip_time**. The first provides the starting time for the entire earthquake source, while the second provides any spatial variation in the slip time with respect to the **origin_time** (if any). Since there are multiple earthquake sources of different types, there are a number of additional fault information fields available for output. We add these additional fields' output to the fault information file:

Excerpt from `step06.cfg`

```
[pylithapp.timedependent.interfaces.fault]
output.vertex_info_fields = [normal_dir, strike_dir, dip_dir, final_slip_creep, \
  final_slip_one, final_slip_two, final_slip_three, slip_time_creep, slip_time_one, \
  slip_time_two, slip_time_three]
```

This additional information will be contained in file `step06-fault_info.h5`. It will contain final slip information for each earthquake source along with slip time information. When we have run the simulation, the output HDF5 and Xdmf files will be contained in `examples/3d/hex8/output` (all with a prefix of `step06`). To open the files in ParaView, the Xdmf (`xmf`) files should be opened, as these files describe the HDF5 data structure. Results using ParaView are shown in Figure 7.28.

#### 7.9.6.5 Step07 - Dirichlet Velocity Boundary Conditions with Time-Dependent Kinematic Fault Slip

In step07 we add velocity boundary conditions in the positive and negative y-directions on the positive and negative x-faces, so that the external boundaries keep pace with the average fault slip. This problem is nearly identical to the strike-slip fault model of Savage and Prescott [Savage and Prescott, 1978], except that the fault creep extends through the viscoelastic portion of the domain.

We use the default ZeroDispBC for the initial displacements on the positive and negative x-faces, as well as the negative z-face. For the velocities on the positive and negative x-faces, we use a UniformDB:

```
# Boundary condition on +x face
[pylithapp.timedependent.bc.x_pos]
bc_dof = [0, 1]
label = face_xpos
db_initial.label = Dirichlet BC on +x
db_rate = spatialdata.spatialdb.UniformDB
db_rate.label = Dirichlet rate BC on +x
db_rate.values = [displacement-rate-x, displacement-rate-y, rate-start-time]
db_rate.data = [0.0*cm/year, 1.0*cm/year, 0.0*year]
```

```
# Boundary condition on -x face
[pylithapp.timedependent.bc.x_neg]
bc_dof = [0, 1]
label = face_xneg
db_initial.label = Dirichlet BC on -x
db_rate = spatialdata.spatialdb.UniformDB
db_rate.label = Dirichlet rate BC on +x
db_rate.values = [displacement-rate-x, displacement-rate-y, rate-start-time]
db_rate.data = [0.0*cm/year, -1.0*cm/year, 0.0*year]
```

The fault definition information is identical to example `step06`. In previous examples, we have just used the default output for the domain and subdomain (ground surface), which includes the displacements. In many cases, it is also useful to include the velocities. PyLith provides this information, computing the velocities for the current time step as the difference between the current displacements and the displacements from the previous time step, divided by the time-step size. This is more accurate than computing the velocities from the displacement field output that has been decimated in time. We can obtain this information by explicitly requesting it in **vertex_data_fields**:

Excerpt from `step07.cfg`

```
# Give basename for output of solution over domain.
[pylithapp.problem.formulation.output.domain]
# We specify that output occurs in terms of a given time frequency, and
# ask for output every 50 years.
# We also request velocity output in addition to displacements.
vertex_data_fields = [displacement, velocity]
output_freq = time_step
time_step = 50.0*year

# We are using HDF5 output so we must change the default writer.
writer = pylith.meshio.DataWriterHDF5
writer.filename = output/step07.h5

# Give basename for output of solution over ground surface.
[pylithapp.problem.formulation.output.subdomain]
# Name of nodeset for ground surface.
label = face_zpos

# We also request velocity output in addition to displacements.
vertex_data_fields = [displacement, velocity]
# We keep the default output frequency behavior (skip every n steps), and
# ask to skip 0 steps between output, so that we get output every time step.
skip = 0

# We again switch the writer to produce HDF5 output.
writer = pylith.meshio.DataWriterHDF5
writer.filename = output/step07-groundsurf.h5
```

When we have run the simulation, the output HDF5 and Xdmf files will be contained in `examples/3d/hex8/output` (all with a prefix of `step07`). As for example step06, make sure to open the `xmf` files rather than the `h5` files. Results using ParaView are shown in Figure 7.29 on the next page.

#### 7.9.6.6 Step08 - Dirichlet Velocity Boundary Conditions with Time-Dependent Kinematic Fault Slip and Power-Law Rheology

The `step08.cfg` file defines a problem that is identical to example step07, except the the lower crust is composed of a power-law viscoelastic material. Since the material behavior is now nonlinear, we must use the nonlinear solver:

Excerpt from `step08.cfg`

```
[pylithapp.timedependent]
# For this problem we must switch to a nonlinear solver.
```

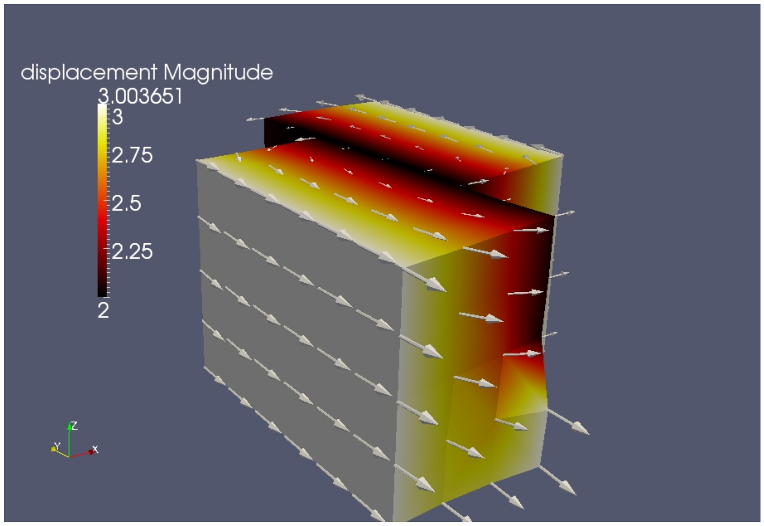



Figure 7.29: Displacement field (color contours) and velocity field (vectors) for example step07 at t = 300 years visualized using ParaView. The mesh has been distorted by the computed displacements (magnified by 500), and the vectors show the computed velocities.

```
implicit.solver = pylith.problems.SolverNonlinear
```

Although we have not discussed the PyLith PETSc settings previously, note that the use of the nonlinear solver may require additional options if we wish to override the defaults. These settings are contained in `pylithapp.cfg`:

Excerpt from `step08.cfg`

```
[pylithapp.petsc]
# Nonlinear solver monitoring options.
snes_rtol = 1.0e-8
snes_atol = 1.0e-12
snes_max_it = 100
snes_monitor = true
snes_view = true
snes_converged_reason = true
```

These settings are ignored unless we are using the nonlinear solver.

When setting the physical properties for the power-law material in PyLith, the parameters (see Section 5.3.4.1 on page 75) do not generally correspond to the values provided in laboratory results. PyLith includes a utility code, `powerlaw_gendb.py`, to simplify the process of using laboratory results with PyLith. This utility code is installed in the same location as PyLith. An example of how to use it is in `examples/3d/hex8/spatialdb/powerlaw`. The user must provide a spatial database defining the spatial distribution of laboratory-derived parameters (contained in `powerlaw_params.spatialdb`), another spatial database defining the temperature field in degrees K (contained in `temperature.spatialdb`), and a set of points for which values are desired (`powerlaw_points.txt`). The parameters for the code are defined in `powerlaw_gendb.cfg`. The properties expected by PyLith are `reference_strain_rate`, `reference_stress`, and `power_law_exponent`. The user must specify either `reference_strain_rate` or `reference_stress` so that `powerlaw_gendb.py` can compute the other property. Default values of 1.0e-6 1/s and 1 MPa are provided. In this example, the same database was used for all parameters, and a separate database was used to define the temperature distribution. In practice, the user can provide any desired thermal model to provide the spatial database for the temperature. In this example, a simple 1D (vertically-varying) distribution was used. The utility code can be used by simply executing it from the `examples/3d/hex8/spatialdb/powerlaw` directory:

```
$ powerlaw_gendb.py
```

This code will automatically read the parameters in `powerlaw_gendb.cfg` in creating the file `examples/3d/hex8/spatialdb/ma`

We first change the material type of the lower crust to PowerLaw3D:

Excerpt from `step08.cfg`

```
# Change material type of lower crust to power-law viscoelastic.
[pylithapp.timedependent]
materials.lower_crust = pylith.materials.PowerLaw3D
```

In many cases, it is useful to obtain the material properties from two different sources. For example, the elastic properties may come from a seismic velocity model while the viscous properties may be derived from a thermal model. In such a case we can use a CompositeDB, which allows a different spatial database to be used for a subset of the properties. We do this as follows:

Excerpt from `step08.cfg`

```
# Provide a spatial database from which to obtain property values.
# In this case, we prefer to obtain the power-law properties from one
# database and the elastic properties from another database, so we use
# a CompositeDB. Each part of the CompositeDB is a SimpleDB.
[pylithapp.timedependent.materials.lower_crust]
db_properties = spatialdata.spatialdb.CompositeDB
db_properties.db_A = spatialdata.spatialdb.SimpleDB
db_properties.db_B = spatialdata.spatialdb.SimpleDB
```

We must define the properties that come from each spatial database and then provide the database parameters:

Excerpt from `step08.cfg`

```
# Provide the values to be obtained from each database and the database
# name.
[pylithapp.timedependent.materials.lower_crust.db_properties]
values_A = [density, vs, vp]   ; Elastic properties.
db_A.label = Elastic properties
db_A.iohandler.filename = spatialdb/mat_elastic.spatialdb
values_B = [reference-stress, reference-strain-rate, power-law-exponent] ; Power-law properties.
db_B.label = Power-law properties
db_B.iohandler.filename = spatialdb/mat_powerlaw.spatialdb
```

The PowerLaw3D material has additional properties and state variables with respect to the default ElasticIsotropic3D material, so we request that these properties be written to the **`lower_crust`** material files:

Excerpt from `step08.cfg`

```
# Since there are additional properties and state variables for the
# power-law model, we explicitly request that they be output. Properties are
# named in cell_info_fields and state variables are named in
# cell_data_fields.
[pylithapp.timedependent.materials.lower_crust]
output.cell_info_fields = [density, mu, lambda, reference_strain_rate, reference_stress, power_law_expon
output.cell_data_fields = [total_strain, stress, viscous_strain]
```

When we have run the simulation, the output HDF5 and Xdmf files will be contained in `examples/3d/hex8/output` (all with a prefix of `step08`). Results using ParaView are shown in Figure 7.30 on the next page.

#### 7.9.6.7 Step09 - Dirichlet Velocity Boundary Conditions with Time-Dependent Kinematic Fault Slip and Drucker-Prager Elastoplastic Rheology

In this example we use a Drucker-Prager elastoplastic rheology in the lower crust. As in example step08, the material behavior is nonlinear so we again use the nonlinear solver. The material is elastoplastic, there is no inherent time-dependent response and the stable time-step size for the material depends on the loading conditions. To avoid this, we set the maximum time-step size to 5 years rather than the value of 10 years used in example `step08`:

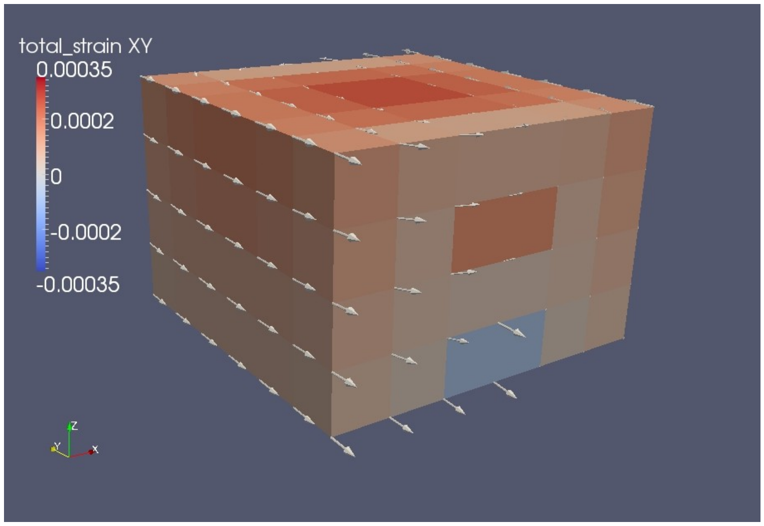



Figure 7.30: The XY-component of strain (color contours) and displacement field (vectors) for example step08 at t = 150 years visualized using ParaView. For this visualization, we loaded both the `step08-lower_crust.xmf` and `step08-upper_crust.xmf` files to contour the strain field, and superimposed on it the displacement field vectors from `step08.xmf`.

Excerpt from `step09.cfg`

```
# Change the total simulation time to 700 years, and set the maximum time
# step size to 5 years.
[pylithapp.timedependent.implicit.time_step]
total_time = 700.0*year
max_dt = 5.0*year
stability_factor = 1.0 ; use time step equal to stable value from materials

# For this problem we set adapt\_skip to zero so that the time step size is
# readjusted every time step.
adapt_skip = 0

# Change material type of lower crust to Drucker-Prager.
[pylithapp.timedependent]
materials.lower_crust = pylith.materials.DruckerPrager3D

# Provide a spatial database from which to obtain property values.
# In this case, we prefer to obtain the Drucker-Prager properties from one
# database and the elastic properties from another database, so we use
# a CompositeDB. Each part of the CompositeDB is a SimpleDB.
[pylithapp.timedependent.materials.lower_crust]
db_properties = spatialdata.spatialdb.CompositeDB
db_properties.db_A = spatialdata.spatialdb.SimpleDB
db_properties.db_B = spatialdata.spatialdb.SimpleDB
```

As for the step08 example, we first define the properties that come from each spatial database and then provide the database filename:

Excerpt from `step09.cfg`

```
# Provide the values to be obtained from each database and the database
# name.
[pylithapp.timedependent.materials.lower_crust.db_properties]
values_A = [density,vs,vp] ; Elastic properties.
db_A.label = Elastic properties
db_A.iohandler.filename = spatialdb/mat_elastic.spatialdb

values_B = [friction-angle, cohesion, dilatation-angle] ; Drucker-Prager properties.
```

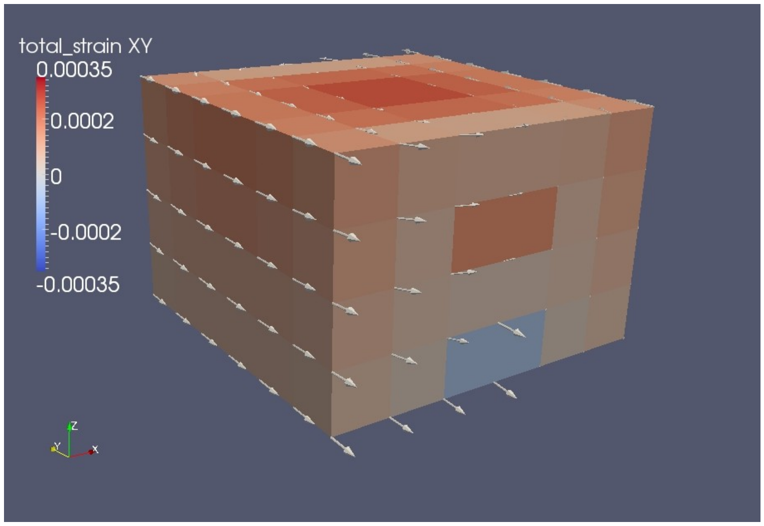



Figure 7.31: The XY-component of strain (color contours) and displacement field (vectors) for example step09 at t = 150 years visualized using ParaView. For this visualization, we loaded both the `step09-lower_crust.xmf` and `step09-upper_crust.xmf` files to contour the strain field, and superimposed on it the displacement field vectors from `step09.xmf`.

```
db_B.label = Drucker-Prager properties
db_B.iohandler.filename = spatialdb/mat\_druckerprager.spatialdb
```

We also request output of the properties and state variables that are unique to the DruckerPrager3D material:

Excerpt from `step09.cfg`

```
# Since there are additional properties and state variables for the
# Drucker-Prager model, we explicitly request that they be output.
# Properties are named in cell\_info\_fields and state variables are named in
# cell_data_fields.
[pylithapp.timedependent.materials.lower_crust]
output.cell_info_fields = [density, mu, lambda, alpha_yield, beta, alpha_flow]
output.cell_data_fields = [total_strain, stress, plastic_strain]
```

When we have run the simulation, the output HDF5 and Xdmf files will be contained in `examples/3d/hex8/output` (all with a prefix of `step09`). Results using ParaView are shown in Figure 7.31.

### 7.9.7 Fault Friction Examples

PyLith features discussed in this example:

- Static fault friction
- Slip-weakening fault friction
- Rate-and-state fault friction
- Nonlinear solver

#### 7.9.7.1 Overview

This set of examples provides an introduction to using fault friction in static and quasi-static problems with PyLith. Dynamic problems with fault friction are discussed in Section 7.13 on page 187. The boundary conditions are all either static or quasi-static Dirichlet conditions, and only elastic materials are used. In all the fault friction examples we apply axial (x) displacements on both the positive and negative x-faces to maintain a compressive normal tractions on the fault. Otherwise, there would be no frictional resistance. Fault friction generates nonlinear behavior, so we use the nonlinear solver. All of the examples are contained in the directory `examples/3d/hex8`, and the corresponding `cfg` files are `step10.cfg`, `step11.cfg`, `step12.cfg`, `step13.cfg`, and `step14.cfg`. Run the examples as follows:

```
# Step10
$ pylith step10.cfg

# Step11
$ pylith step11.cfg

# Step12
$ pylith step12.cfg

# Step13
$ pylith step13.cfg

# Step14
$ pylith step14.cfg
```

This will cause PyLith to read the default parameters in `pylithapp.cfg`, and then override or augment them with the additional parameters in the `stepXX.cfg` file. Each `cfg` file is extensively documented, to provide detailed information on the various parameters.

#### 7.9.7.2 Step10 - Static Friction (Stick) with Static Dirichlet Boundary Conditions

The `step10.cfg` file defines a problem that is identical to example step01, except for the presence of a vertical fault with static friction. In this case, the applied displacements are insufficient to cause the fault to slip, so the solution is identical to that in example step01. As in previous examples involving faults, we must first provide an array defining the fault interfaces:

Excerpt from `step10.cfg`

```
[pylithapp.timedependent]
# Set interfaces to an array of 1 fault: 'fault'.
interfaces = [fault]

# Fault friction models are nonlinear, so use nonlinear solver.
[pylithapp.timedependent.implicit]
solver = pylith.problems.SolverNonlinear
```

We need to change the fault interface from the default (FaultCohesiveKin) to FaultCohesiveDyn and we set the friction model to use:

Excerpt from `step10.cfg`

```
[pylithapp.timedependent.interfaces]
fault = pylith.faults.FaultCohesiveDyn ; Change to dynamic fault interface.

[pylithapp.timedependent.interfaces.fault]
friction = pylith.friction.StaticFriction ; Use static friction model.
```

The StaticFriction model requires values for the coefficient of friction and the cohesion (see Section 6.4.5.3 on page 105). We provide both of these using a UniformDB:

Excerpt from `step10.cfg`

```
[pylithapp.timedependent.interfaces.fault]
# Set static friction model parameters using a uniform DB. Set the
# static coefficient of friction to 0.6 and cohesion to 0.0 Pa.
friction.db_properties = spatialdata.spatialdb.UniformDB
friction.db_properties.label = Static friction
```

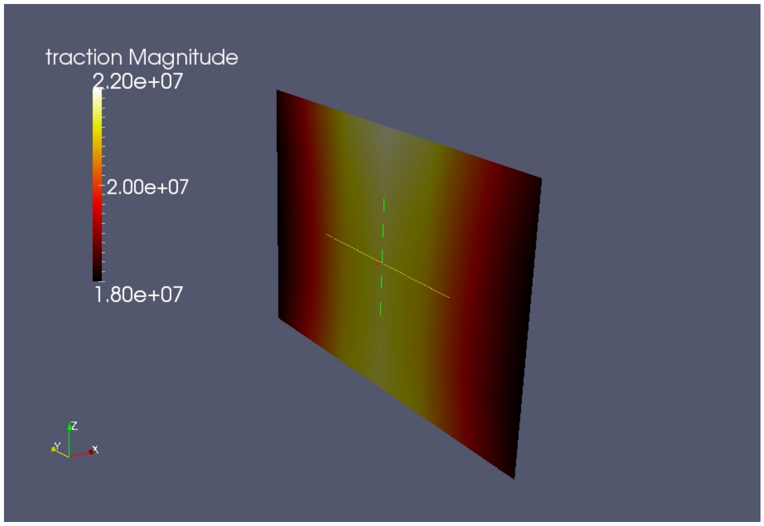



Figure 7.32: Magnitude of tractions on the fault for example step10 visualized using ParaView.

```
friction.db_properties.values = [friction-coefficient, cohesion]
friction.db_properties.data = [0.6, 0.0*Pa]

# Fault friction models require additional PETSc settings:
[pylithapp.petsc]
# Friction sensitivity solve used to compute the increment in slip
# associated with changes in the Lagrange multiplier imposed by the
# fault constitutive model.
friction_pc_type = asm
friction_sub_pc_factor_shift_type = nonzero
friction_ksp_max_it = 25
friction_ksp_gmres_restart = 30

# Uncomment to view details of friction sensitivity solve.
#friction_ksp_monitor = true
#friction_ksp_view = true
friction_ksp_converged_reason = true
```

When we have run the simulation, the output VTK files will be contained in `examples/3d/hex8/output` (all with a prefix of `step10`). Results using ParaView are shown in Figure 7.32.

#### 7.9.7.3 Step11 - Static Friction (Slip) with Static Dirichlet Boundary Conditions

In step11 we apply twice as much shear displacement as in step10, which is sufficient to induce slip on the fault. All other settings are identical. To change the amount of shear displacement, we change the spatial database for the positive and negative x-faces to a UniformDB, and apply the altered values within the `cfg` file:

Excerpt from `step11.cfg`

```
# Boundary condition on +x face
[pylithapp.timedependent.bc.x_pos]
bc_dof = [0, 1]
label = face_xpos
db_initial = spatialdata.spatialdb.UniformDB
db_initial.label = Dirichlet BC on +x
db_initial.values = [displacement-x, displacement-y]
db_initial.data = [-1.0*m, 2.0*m]

# Boundary condition on -x face
```

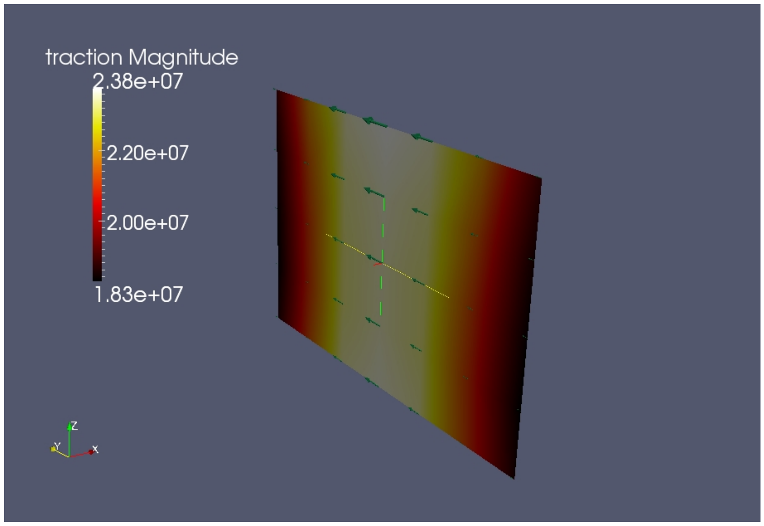



Figure 7.33: Magnitude of tractions on the fault for example step10 visualized using ParaView. Vectors of fault slip are also plotted. Note that PyLith outputs slip in the fault coordinate system, so we transform them to the global coordinate system using the Calculator in ParaView. A more general approach involves outputing the fault coordinate system information and using these fields in the Calculator.

```
[pylithapp.timedependent.bc.x_neg]
bc_dof = [0, 1]
label = face_xneg
db_initial = spatialdata.spatialdb.UniformDB
db_initial.label = Dirichlet BC on -x
db_initial.values = [displacement-x, displacement-y]
db_initial.data = [1.0*m, -2.0*m]
```

When we have run the simulation, the output VTK files will be contained in `examples/3d/hex8/output` (all with a prefix of `step11`). Results using ParaView are shown in Figure 7.33.

#### 7.9.7.4 Step12 - Static Friction with Quasi-Static Dirichlet Boundary Conditions

The `step12.cfg` file describes a problem that is similar to examples step10 and step11, except that we apply velocity boundary conditions and run the simulation for 200 years. Once fault friction is overcome, the fault slips at a steady rate. To prevent convergence problems we set the time step size to a constant value of 5 years:

Excerpt from `step12.cfg`

```
# Change the total simulation time to 200 years, and use a constant time
# step size of 5 years.
[pylithapp.timedependent.implicit.time_step]
total_time = 200.0*year
dt = 5.0*year
```

As in the other fault friction examples, we apply initial displacements along the x-axis (to maintain a compressive stress on the fault), and we apply velocity boundary conditions that yield a left-lateral sense of motion:

Excerpt from `step12.cfg`

```
# Boundary condition on +x face -- Dirichlet
[pylithapp.timedependent.bc.x_pos]
bc_dof = [0,1]
label = face_xpos
db_initial = spatialdata.spatialdb.UniformDB
```

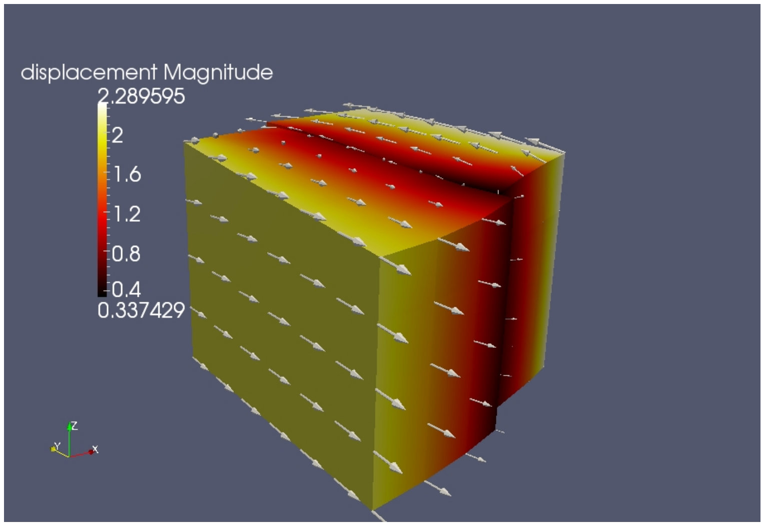



Figure 7.34: Displacement field for example step12 at t = 200 years visualized using ParaView. The mesh has been distorted by the computed displacements (magnified by 500), and the vectors show the computed displacements.

```
db_initial.label = Dirichlet BC on +x
db_initial.values = [displacement-x, displacement-y]
db_initial.data = [-1.0*m, 0.0*m]

db_rate = spatialdata.spatialdb.UniformDB
db_rate.label = Dirichlet rate BC on +x
db_rate.values = [displacement-rate-x, displacement-rate-y, rate-start-time]
db_rate.data = [0.0*cm/year, 1.0*cm/year, 0.0*year] \\

# Boundary condition on -x face
[pylithapp.timedependent.bc.x_neg]
bc_dof = [0, 1]
label = face_xneg
db_initial.label = Dirichlet BC on -x

db_rate = spatialdata.spatialdb.UniformDB
db_rate.label = Dirichlet rate BC on -x
db_rate.values = [displacement-rate-x, displacement-rate-y, rate-start-time]
db_rate.data = [0.0*cm/year, -1.0*cm/year, 0.0*year]
```

For this example, we keep the same coefficient of friction as examples step10 and step11, but we include a cohesion of 2 MPa:

Excerpt from `step12.cfg`

```
[pylithapp.timedependent.interfaces.fault]
# Set static friction model parameters using a uniform DB. Set the
# static coefficient of friction to 0.6 and cohesion to 2.0 MPa.
friction.db_properties = spatialdata.spatialdb.UniformDB
friction.db_properties.label = Static friction
friction.db_properties.values = [friction-coefficient, cohesion]
friction.db_properties.data = [0.6, 2.0*MPa]
```

When we have run the simulation, the output VTK files will be contained in `examples/3d/hex8/output` (all with a prefix of `step12`). Results using ParaView are shown in Figure 7.34.

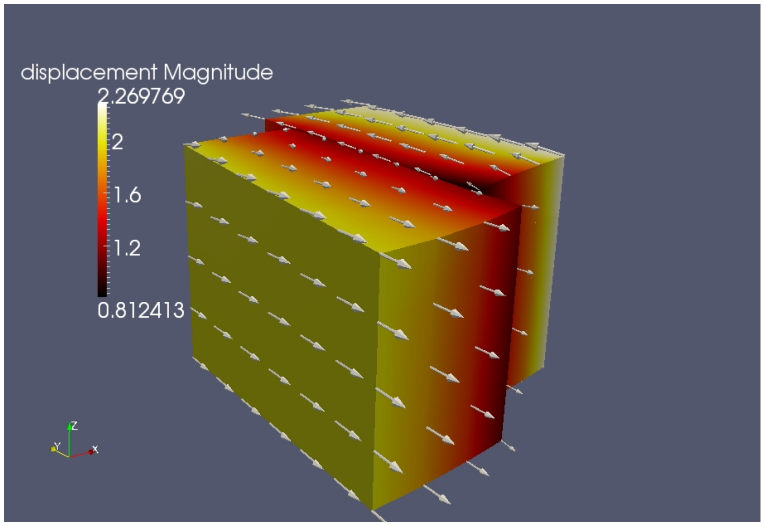



Figure 7.35: Displacement field for example step13 at t = 200 years visualized using ParaView. The mesh has been distorted by the computed displacements (magnified by 500), and the vectors show the computed displacements.

#### 7.9.7.5 Step13 - Slip-Weakening Friction with Quasi-Static Dirichlet Boundary Conditions

In this example we replace the static friction fault constitutive model in step12 with a slip-weakening friction fault constitutive model. Fault friction is overcome at about t = 80 years, the fault slips in each subsequent time step. We again use a constant time step size of 5 years and apply the same intial displacement and velocity boundary conditions.

We first define the friction model for the simulation:

Excerpt from `step13.cfg`

```
[pylithapp.timedependent.interfaces.fault]
# Use the slip-weakening friction model.
friction = pylith.friction.SlipWeakening

[pylithapp.timedependent.interfaces.fault]
# Set slip-weakening friction model parameters using a uniform DB. Set the
# parameters as follows:
# static coefficient of friction: 0.6
# dynamic coefficient of friction: 0.5
# slip-weakening parameter: 0.2 m
# cohesion: 0 Pa
friction.db_properties = spatialdata.spatialdb.UniformDB
friction.db_properties.label = Slip weakening
friction.db_properties.values = [static-coefficient,dynamic-coefficient, \
   slip-weakening-parameter,cohesion]
friction.db_properties.data = [0.6,0.5,0.2{*}m,0.0{*}Pa]
```

When we have run the simulation, the output VTK files will be contained in `examples/3d/hex8/output` (all with a prefix of `step13`). Results using ParaView are shown in Figure 7.35.

#### 7.9.7.6 Step14 - Rate-and-State Friction with Quasi-Static Dirichlet Boundary Conditions

In step14 we use a rate-and-state friction model with an ageing law instead of a slip-weakening friction model. Slip begins to occur at about t = 45 years, and continues in each subsequent time step. We again use a constant time step size of 5 years and apply the same intial displacement and velocity boundary conditions.

We first define the friction model for the simulation:

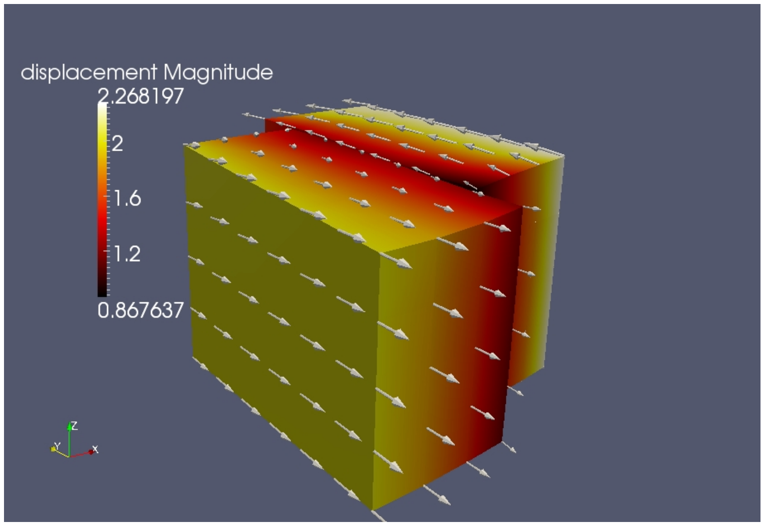



Figure 7.36: Displacement field for example step14 at t = 200 years visualized using ParaView. The mesh has been distorted by the computed displacements (magnified by 500), and the vectors show the computed displacements.

Excerpt from `step14.cfg`

```
[pylithapp.timedependent.interfaces.fault]
# Use the rate-and-state aging friction model.
friction = pylith.friction.RateStateAgeing

[pylithapp.timedependent.interfaces.fault]
# Set rate-and-state parameters using a UniformDB. Set the parameters as
# follows:
# reference coefficient of friction: 0.6
# reference slip rate: 1.0e-06 m/s
# slip-weakening parameter: 0.037 m
# a: 0.0125
# b: 0.0172
# cohesion: 0 Pa
friction.db_properties = spatialdata.spatialdb.UniformDB
friction.db_properties.label = Rate State Ageing
friction.db_properties.values = [reference-friction-coefficient, reference-slip-rate, \
  characteristic-slip-distance, constitutive-parameter-a, constitutive-parameter-b, cohesion]
friction.db_properties.data = [0.6, 1.0e-6*m/s, 0.0370*m, 0.0125, 0.0172, 0.0*Pa]
```

For this model, we also want to set the initial value of the state variable:

Excerpt from `step14.cfg`

```
[pylithapp.timedependent.interfaces.fault]
# Set spatial database for the initial value of the state variable.
friction.db_initial_state = spatialdata.spatialdb.UniformDB
friction.db_initial_state.label = Rate State Ageing State
friction.db_initial_state.values = [state-variable]
friction.db_initial_state.data = [92.7*s]
```

When we have run the simulation, the output VTK files will be contained in `examples/3d/hex8/output` (all with a prefix of `step14`). Results using ParaView are shown in Figure 7.36.

### 7.9.8 Gravitational Body Force Examples

PyLith features discussed in this example:

- Gravitational body forces
- Initial stresses
- Finite strain
- Generalized Maxwell linear viscoelastic material

#### 7.9.8.1 Overview

This set of examples describes a set of problems for PyLith involving gravitational body forces. All of the examples are quasi-static and run for a time period of 200 years. These examples also demonstrate the use of a generalized Maxwell viscoelastic material, which is used for the lower crust in all examples. The final example (step17) demonstrates the usage of a finite strain formulation, which automatically invokes the nonlinear solver. All of the examples are contained in the directory `examples/3d/hex8`, and the corresponding `cfg` files are `step15.cfg`, `step16.cfg`, and `step17.cfg`. Run the examples as follows:

```
# Step15
$ pylith step15.cfg

# Step16
$ pylith step16.cfg

# Step17
$ pylith step17.cfg
```

This will cause PyLith to read the default parameters in `pylithapp.cfg`, and then override or augment them with the additional parameters in the `stepXX.cfg` file. Each `cfg` file is extensively documented, to provide detailed information on the various parameters.

#### 7.9.8.2 Step15 - Gravitational Body Forces

The `step15.cfg` file defines a problem with extremely simple Dirichlet boundary conditions. On the positive and negative x-faces, the positive and negative y-faces, and the negative z-face, the displacements normal to the face are set to zero. Because all of the materials in the example have the same density, the elastic solution for loading via gravitational body forces is

$$\sigma_{zz} = \rho g h; \sigma_{xx} = \sigma_{yy} = \frac{\nu \rho g h}{1-\nu}. \tag{7.1}$$

We set the gravity field, which by default has values of 9.80655 $m/s^2$ for acceleration and $[0,0,-1]$ for direction and time stepping implementation:

Excerpt from `Step15.cfg`

```
[pylithapp.timedependent]
gravity_field = spatialdata.spatialdb.GravityField ; Set gravity field

[pylithapp.timedependent.implicit]
# Change time stepping algorithm from uniform time step, to adaptive
# time stepping.
time_step = pylith.problems.TimeStepAdapt

# Change the total simulation time to 200 years, and set the maximum time
# step size to 10 years.
[pylithapp.timedependent.implicit.time_step]
total_time = 200.0*year
max_dt = 10.0*year
stability_factor = 1.0 ; use time step equal to stable value from materials
```

We use a generalized Maxwell model for the lower crust (see Section 5.3.3 on page 70), and use a SimpleDB to provide the properties. We also request the relevant properties and state variables for output:

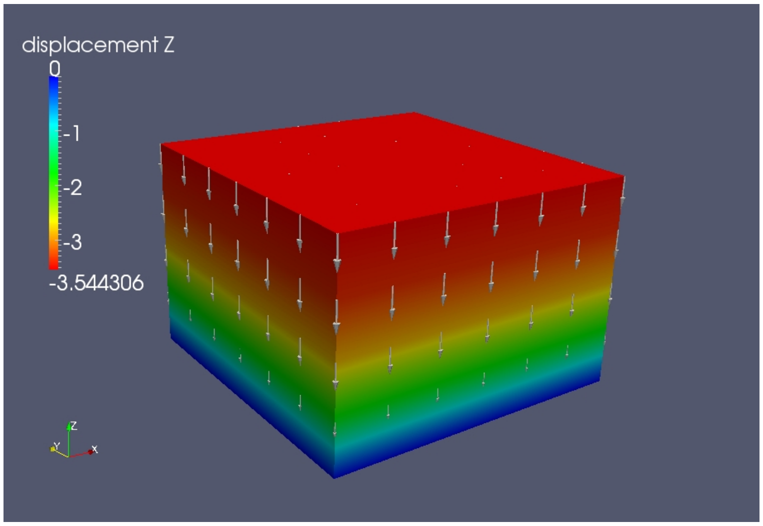



Figure 7.37: Displacement field for example step15 at t = 200 years visualized using ParaView. The z-component of the displacement field is shown with the color contours, and the vectors show the computed displacements.

Excerpt from `Step15.cfg`

```
# Change material type of lower crust to generalized Maxwell viscoelastic.
[pylithapp.timedependent]
materials.lower_crust = pylith.materials.GenMaxwellIsotropic3D
# Provide a spatial database from which to obtain property values.
# Since there are additional properties and state variables for the
# generalized Maxwell model, we explicitly request that they be output.
# Properties are named in cell\_info\_fields and state variables are named in
# cell\_data\_fields.
[pylithapp.timedependent.materials.lower_crust]
db_properties.iohandler.filename = spatialdb/mat\_genmaxwell.spatialdb
output.cell_info_fields = [density, mu, lambda, shear_ratio, maxwell_time]
output.cell_data_fields = [total_strain, stress, viscous_strain_1, viscous_strain_2, \\
  viscous_strain_3]
```

The boundary conditions for this example are trivial, so we are able to use the default ZeroDispDB for all faces. When we have run the simulation, the output VTK files will be contained in `examples/3d/hex8/output` (all with a prefix of `step15`). Results using ParaView are shown in Figure 7.37.

#### 7.9.8.3 Step16 - Gravitational Body Forces with Initial Stresses

The `step16.cfg` file defines a problem that is identical to example step15, except that initial stresses are used to prevent the initial large displacements due to 'turning on' gravity. Since all normal stress components are given an initial stress of $\rho g h$, the initial stress state is lithostatic, which is an appropriate condition for many tectonic problems in the absence of tectonic stresses (e.g., McGarr [McGarr, 1988]). When compared to example step15, this example should maintain a lithostatic state of stress for the entire simulation, and displacements should remain essentially zero.

We set the gravity field, as in example step15, and we again use adaptive time stepping with a generalized Maxwell rheology for the lower crust. We provide values for the initial stress for both the upper and lower crust. Since the materials have the same density, we are able to use the same SimpleDB with a linear variation for both (see file `examples/3d/hex8/spatialdb/initial_st`

Excerpt from `Step16.cfg`

```
# We must specify initial stresses for each material.
# We provide a filename for the spatial database that gives the stresses,
# and we change the query_type from the default 'nearest' to 'linear'.
```

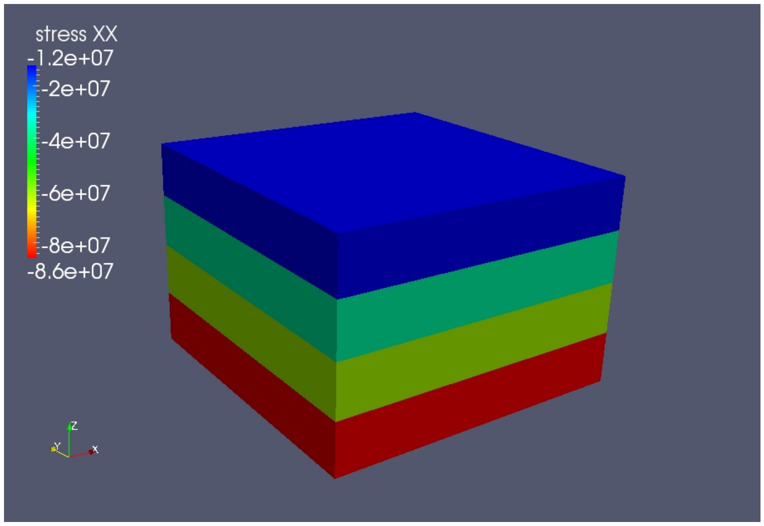



Figure 7.38: Stress field (xx-component) for example step16 at t = 200 years visualized using ParaView. Note that for this example, Stress_xx = Stress_yy = Stress_zz, and there is no vertical displacement throughout the simulation. Also note that the stresses appear as four layers since we have used CellFilterAvg for material output.

```
[pylithapp.timedependent.materials.upper_crust]
db_initial_stress = spatialdata.spatialdb.SimpleDB
db_initial_stress.iohandler.filename = spatialdb/initial_stress.spatialdb
db_initial_stress.query_type = linear

[pylithapp.timedependent.materials.lower_crust]
db_initial_stress = spatialdata.spatialdb.SimpleDB
db_initial_stress.iohandler.filename = spatialdb/initial_stress.spatialdb
db_initial_stress.query_type = linear
```

Note that we use a `linear` **query_type** rather than the default type of `nearest`, so that a linear interpolation is performed along the z-direction. When we have run the simulation, the output VTK files will be contained in `examples/3d/hex8/output` (all with a prefix of `step16`). Results using ParaView are shown in Figure 7.38.

#### 7.9.8.4 Step17 - Gravitational Body Forces with Small Strain

The `step17.cfg` file defines a problem that is identical to example step15, except that we now use a small strain formulation (see Section 2.5 on page 14). All of the problems up to this point have assumed infinitesimal strain, meaning that the change in shape of the domain during deformation is not taken into account. In many problems it is important to consider the change in shape of the domain. This is particularly important in many problems involving gravitational body forces, since a change in shape of the domain results in a different stress field. By examining the stress and deformation fields for this example in comparison with those of example step15, we can see what effect the infinitesimal strain approximation has on our solution.

We set the gravity field, as in example step15 and again use adaptive time stepping withs a generalized Maxwell rheology for the lower crust. The only change is that we change the problem formulation from the default Implicit to ImplicitLgDeform. Since the large deformation formulation is nonlinear, PyLith automatically switches the solver from the default SolverLinear to SolverNonlinear. It is thus only necessary to change the formulation:

Excerpt from `Step17.cfg`

```
[pylithapp.timedependent]
# Set the formulation for finite strain. The default solver will
# automatically be switched to the nonlinear solver.
formulation = pylith.problems.ImplicitLgDeform
```

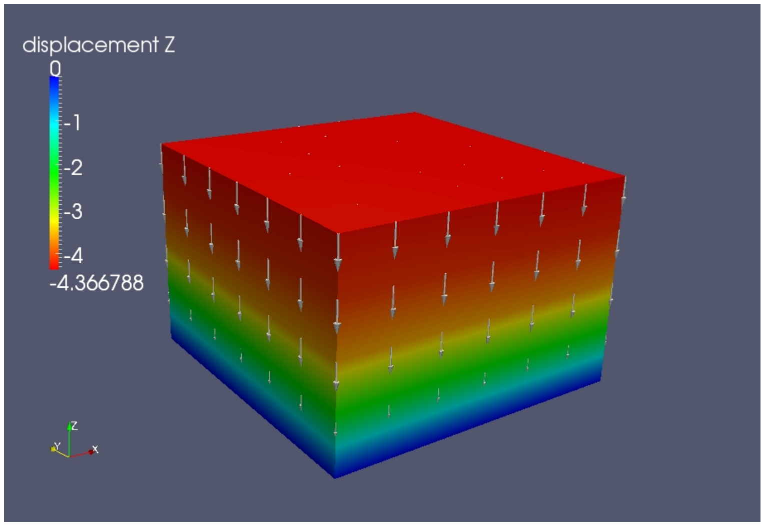



Figure 7.39: Displacement field for example step17 at t = 200 years visualized using ParaView. The z-component of the displacement field is shown with the color contours, and the vectors show the computed displacements. Note the larger displacements compared with example step15.

When we have run the simulation, the output VTK files will be contained in `examples/3d/hex8/output` (all with a prefix of `step17`). Results using ParaView are shown in Figure 7.39.

### 7.9.9 Surface Load Traction Examples

PyLith features discussed in this example:

- Time-dependent Neumann (traction) boundary conditions
- Dirichlet boundary conditions
- Elastic material
- Output of solution at user-defined locations

#### 7.9.9.1 Overview

This set of examples describes a set of problems for PyLith involving surface loading with a Neumann (traction) applied to the ground surface. The first example demonstrates the use of a surface load in a static problem, and the second example demonstates how to apply a cyclic load in a quasi-static problem. The second problem also includes output of the solution at user-defined locations. All of the examples are contained in the directory `examples/3d/hex8`, and the corresponding `cfg` files are `step18.cfg` and `step19.cfg`. Run the examples as follows:

```
# Step18
$ pylith step18.cfg

# Step19
$ pylith step19.cfg
```

This will cause PyLith to read the default parameters in `pylithapp.cfg`, and then override or augment them with the additional parameters in the `stepXX.cfg` file. Each `cfg` file is extensively documented, to provide detailed information on the various parameters.

#### 7.9.9.2 Step18 - Static Surface Load

The `step18.cfg` file defines a problem with a spatially varying axial surface load applied to the top surface with Dirichlet (roller) boundary conditions on the lateral and bottom surfaces. We first set the array of boundary conditions with one for each surface of the domain. As in the other examples, we also setup output for the ground surface.

For the Dirichlet boundary conditions we fix the degree of freedom associated with motion normal to the boundary while leaving the other degrees of freedom free. We do not explicitly specify the use of a Dirichlet boundary condition because it is the default. Similarly, the ZeroDispDB is the default spatial database for the displacements in a Dirichlet boundary condition, so all we need to specify is the degree of freedom that is constrained, the name of the nodeset from CUBIT, and a label used in diagnostic output. For the Dirichlet boundary condition on the +x surface we have:

Excerpt from `Step18.cfg`

```
[pylithapp.timedependent.bc.x_pos]
label = face_xpos
bc_dof = [0]

db_initial.label = Dirichlet BC on +x
```

On the top surface we apply a Neumann boundary condition for the surface load, so we first set the boundary condition type and then specify the nodeset in CUBIT associated with this surface. For the static surface load, we use a spatial database for the initial value and linear interpolation. We integrate the surface tractions over the boundary, so we also specify the numerical integration scheme to use. Finally, we specify a vector for the up direction because the tractions are applied to a horizontal surface, resulting in ambiguous shear directions for our default orientation convention.

Excerpt from `Step18.cfg`

```
[pylithapp.timedependent.bc]
z_pos = pylith.bc.Neumann

[pylithapp.timedependent.bc.z_pos]
label = face_zpos

db_initial = spatialdata.spatialdb.SimpleDB
db_initial.label = Neumann BC on +z
db_initial.iohandler.filename = spatialdb/tractions\_axial\_pressure.spatialdb
db_initial.query_type = linear ; Use linear interpolation.

# Diagnostic output
output.cell_info_fields = [initial-value]
output.writer.filename = output/step18-traction.vtk
output.cell_filter = pylith.meshio.CellFilterAvg

# We must specify quadrature information for the cell faces.
quadrature.cell = pylith.feassemble.FIATLagrange
quadrature.cell.dimension = 2
quadrature.cell.quad_order = 2 \\

# Because normal for +z surface is {[}0,0,1{]}, the horizontal and
# vertical shear directions are ambiguous. We provide a ``fake'' up
# direction of [0,1,0] so that the horizontal shear direction (cross
# product of ``up'' and normal is [1,0,0] and the vertical shear
# direction (cross product of normal and horizontal) is [0,1,0].
up_dir = [0,1,0]
```

When we have run the simulation, the output VTK files will be contained in `examples/3d/hex8/output` (all with a prefix of `step18`). Results using ParaView are shown in Figure 7.40 on the next page.

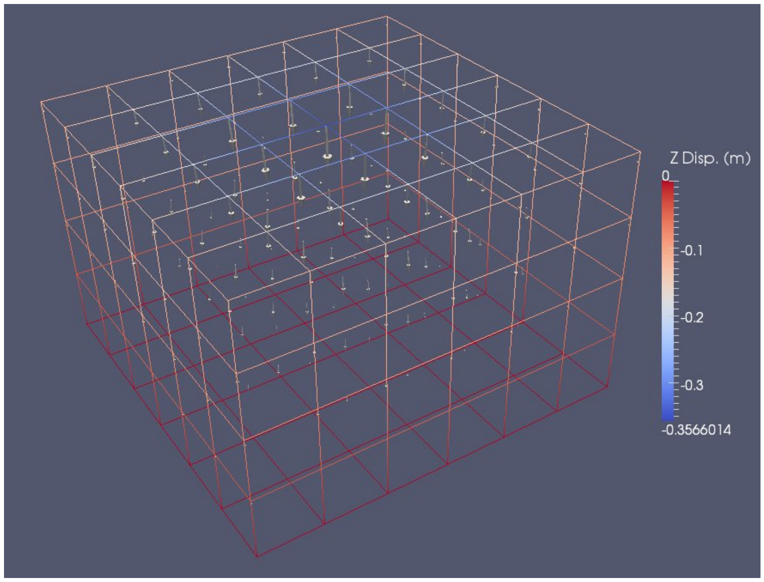



Figure 7.40: Displacement field for example step18 visualized using ParaView. The vectors show the displacement field while the colors in the wireframe correspond to the z-component of the displacement field.

#### 7.9.9.3 Step19 - Time-Dependent Surface Load

The `step19.cfg` file defines a problem that is identical to example step18, except that we vary the amplitude of the surface load as a function of time. We use a temporal database (analogous to our spatial databases for specifying spatial variations) to prescribe a piecewise linear variation of the amplitude with time as given in the file `spatialdb/loadcycle.timedb`. The amplitude begins at zero, progresses to 1.0, then 1.5, before decreasing in a symmetric fashion. The temporal database can use variable time steps to prescribe arbitrary time histories.

Rather than specify a spatial database for the initial value of the Neumann boundary condition corresponding to the surface load, we specify a spatial database for the change in value and the temporal database:

Excerpt from `Step19.cfg`

```
[pylithapp.timedependent.bc.z_pos]
label = face_zpos

db_change = spatialdata.spatialdb.SimpleDB
db_change.label = Amplitude of Neumann BC on +z
db_change.iohandler.filename = spatialdb/tractions_axial_pressure.spatialdb
db_change.query_type = linear ; Use linear interpolation

th_change = spatialdata.spatialdb.TimeHistory
th_change.label = Time history for Neumann BC on +z
th_change.filename = spatialdb/loadcycle.timedb
```

When we have run the simulation, the output VTK files will be contained in `examples/3d/hex8/output` (all with a prefix of `step19`). Results using ParaView are shown in Figure 7.41 on the facing page. We also output the solution at user-defined locations, which are given in the file `output_points.txt`. See Section 4.7.2 on page 48 for a discussion of the output parameters. This type of output is designed for comparison against observations and inversions and output via HDF5 files (see Section 4.7.5 on page 49).

### 7.9.10 Dike Intrusion Example

PyLith features discussed in this example:

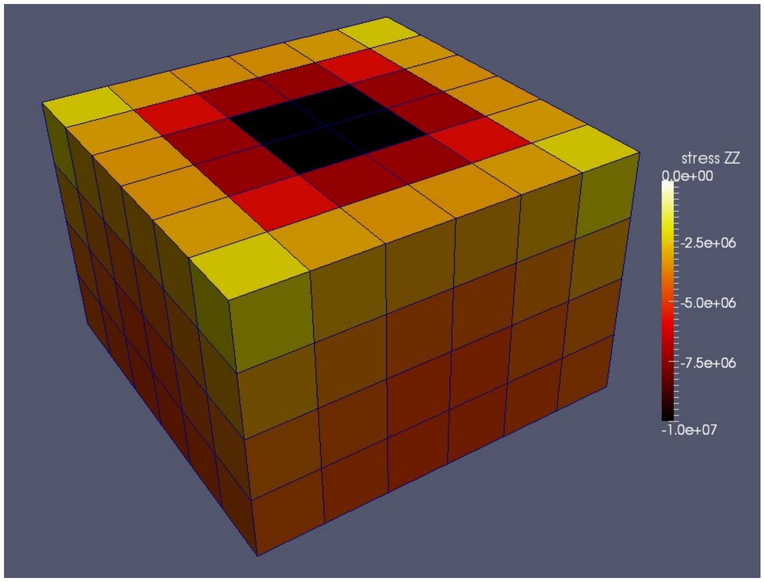



Figure 7.41: Stress field (zz-component) for example step19 at t = 200 years visualized using ParaView. The stresses appear as four layers since we have used CellFilterAvg for material output.

- Fault opening via prescribed tractions to mimic a dike instrusion
- Dirichlet boundary conditions
- Elastic material
- VTK output

#### 7.9.10.1 Overview

This set of examples describes a problem where prescribed tensile tractions are imposed on a fault to mimic a dike intrusion. The example is contained in the directory `examples/3d/hex8`, and the corresponding `cfg` file is `step20.cfg`. The example may be run as follows:

```
$ pylith step20.cfg
```

This will cause PyLith to read the default parameters in `pylithapp.cfg`, and then override or augment them with the additional parameters in the `step20.cfg` file. The `cfg` file is extensively documented, to provide detailed information on the various parameters.

#### 7.9.10.2 Step20 - Static Dike Intrusion

The `step20.cfg` file defines a problem with spatially varying tensile normal tractions on the fault surface associated with a fluid intrusion. The lateral sides and bottom of the domain are fixed using Dirichlet (roller) boundary conditions. As in the other examples, we also setup output for the ground surface.

We use the FaultCohesiveDyn object to impose tractions on the fault surface. We must include a fault constitutive model so we choose static friction with a coefficient of friction of 0.1. The coefficient of friction is irrelevant for the center of the fault where we impose uniform tensile tractions (10 MPa) and the fault opens, but it facilitates clamping the edges of the fault via compressive normal tractions (-100 MPa). Note that we must set the property **`open_free_surface`** to False in order for the tractions to be imposed when the fault is open; the default behavior for fault opening is a free surface (the two sides of the fault are completely uncoupled). The most important fault parameters for prescribing the tensile fault tractions are

Excerpt from `Step20.cfg`

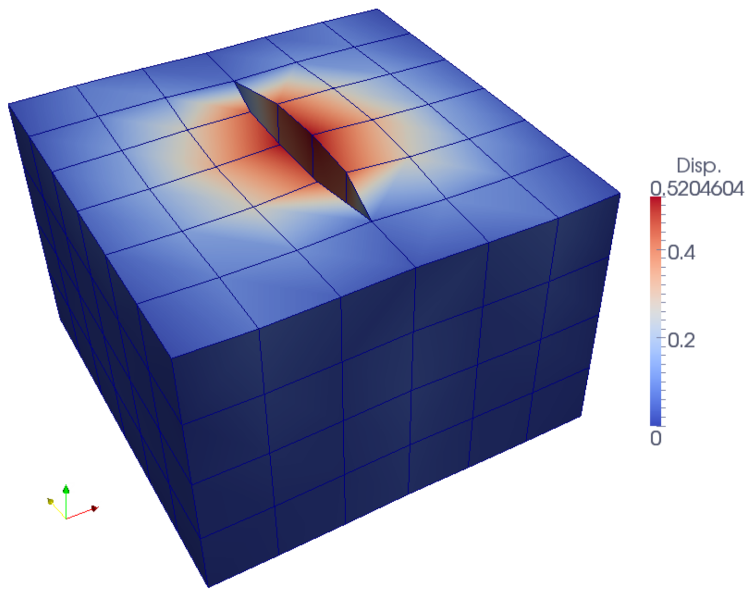



Figure 7.42: Displacement magnitude for example step20 visualized using ParaView.

```
[pylithapp.timedependent.interfaces.fault]
open_free_surface = False
traction_perturbation = pylith.faults.TractPerturbation

[pylithapp.timedependent.interfaces.fault.traction_perturbation]
db_initial = spatialdata.spatialdb.SimpleDB
db_initial.label = Initial fault tractions
db_initial.iohandler.filename = spatialdb/tractions_opening.spatialdb
db_initial.query_type = nearest
```

When we have run the simulation, the output VTK files will be contained in `examples/3d/hex8/output` (all with a prefix of `step20`). Results using ParaView are shown in Figure 7.42.

### 7.9.11 Green's Functions Generation Example

PyLith features discussed in this example:

- Generation of Green's functions from a fault
- Kinematic fault impulses
- Running a different problem type
- Dirichlet boundary conditions
- Elastic material
- HDF5 output
- Interpolated point output

#### 7.9.11.1 Overview

This example describes a problem where we generate a set of Green's functions that could be used in an inversion. The example is contained in the directory `examples/3d/hex8`, and the corresponding `cfg` file is `step21.cfg`. The example may be run as follows:

```
$ pylith step21.cfg --problem=pylith.problems.GreensFns
```

This will cause PyLith to read the default parameters in `pylithapp.cfg` and `greensfns.cfg`, and then override or augment them with the additional parameters in the `step21.cfg` file. The `cfg` files are extensively documented, to provide detailed information on the various parameters.

#### 7.9.11.2 Step21 - Green's Function Generation

This problem makes use of two `cfg` files that are read by default – `pylithapp.cfg` and `greensfns.cfg`. The `greensfns.cfg` file is read automatically because we have changed the problem type to GreensFns (as opposed to the default TimeDependent problem type). The facility name then becomes **greensfns**, and PyLith will therefore search for a `cfg` file matching the name of the facility. The `greensfns.cfg` file contains settings that are specific to the GreensFns problem type:

Excerpt from `Step21.cfg`

```
[greensfns]
fault_id = 10

[greensfns.interfaces]
fault = pylith.faults.FaultCohesiveImpulses

[greensfns.interfaces.fault]
impulse_dof = [0, 1]

db_impulse_amplitude.label = Amplitude of slip impulses
db_impulse_amplitude.iohandler.filename = spatialdb/impulse_amplitude.spatialdb
db_impulse_amplitude.query_type = nearest
```

We specify the **fault_id**, which is required by the GreensFns problem type (it is the same as the ID used when generating the mesh). We also change the fault type to FaultCohesiveImpulses, which allows us to apply a single impulse of slip for each impulse with a nonzero slip value in the corresponding spatial database file (`spatialdb/impulse_amplitude.spatialdb`). We indicate that we would like to apply slip impulses in both the left-lateral (**impulse_dof** = 0) and updip (**impulse_dof** = 1) directions, and we use nearest-neighbor interpolation to determine the amount of applied slip. Note that in the `spatialdb/impulse_am` file we specify negative slip, thus reversing the sense of applied slip for both slip directions. Note that we also put a margin of zeros around the edge of the fault, which prevents impulses from being applied along this boundary.

The `step21.cfg` file defines the remainder of the parameters for this problem. The boundary conditions and fault information are provided as for previous examples. Rather than computing the solution over the ground surface, we choose to provide output at a set of points. PyLith provides the ability to interpolate displacements to a specified set of points, which would generally be necessary when generating Green's functions:

Excerpt from `Step21.cfg`

```
[pylithapp.problem.formulation]
output = [domain, points]
output.points = pylith.meshio.OutputSolnPoints

[pylithapp.problem.formulation.output.points]
writer = pylith.meshio.DataWriterHDF5
writer.filename = output/step21-points.h5
reader.filename = greensfns_points.txt
coordsys.space_dim = 3
coordsys.units = m
```

We first define OutputSolnPoints as the output manager for points output. We use HDF5 output for all of the Green's function output, as it will generally be more efficient (faster I/O, smaller file sizes). We must provide a set of points for point output. The file `greensfns_points.txt` contains a set of (x,y,z) coordinates. We must also provide the spatial dimension of the coordinates as well as the units used. Note that we do not output any info or data fields for state variable output, as this would otherwise create a large amount of output for each applied slip impulse. When we have run the simulation, the output HDF5 files will be contained in `examples/3d/hex8/output` (all with a prefix of `step21`). In Figure 7.43 on the following

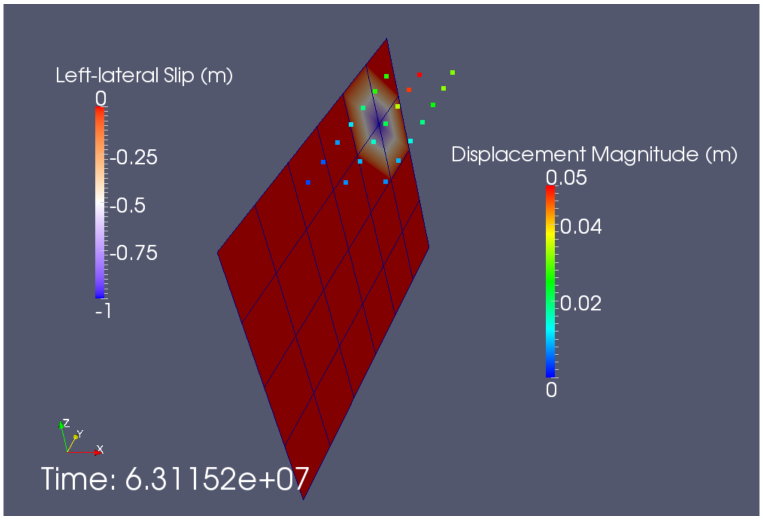



Figure 7.43: A slip impulse and the resulting point displacement responses visualized using ParaView.

page we show an impulse of left-lateral slip applied on the fault and the resulting response at the specified set of points. The time corresponds to the impulse number in multiples of the specified time step size.

## 7.10 Example for Slip on a 2D Subduction Zone

PyLith features discussed in this example:

- Static solution
- Quasi-static solution
- CUBIT/Trelis mesh generation w/APREPRO
- Nonplanar geometry
- Variable mesh resolution
- Linear triangular cells
- HDF5 output
- Dirichlet displacement and velocity boundary conditions
- ZeroDispDB spatial database
- UniformDB spatial database
- SimpleDB spatial database
- SimpleGridDB
- Multiple materials
- Nonlinear solver
- Plane strain linearly elastic material
- Plane strain linear Maxwell viscoelastic material
- Prescribed slip
- Spontaneous rupture
- Multiple faults
- Spatially variable coseismic slip
- Spatially variable aseismic creep
- Afterslip via fault friction
- Static friction
- Slip-weakening friction
- Rate-state friction

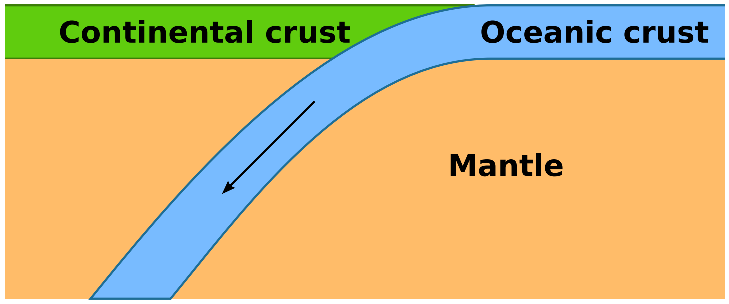



Figure 7.44: Cartoon of subduction zone example.

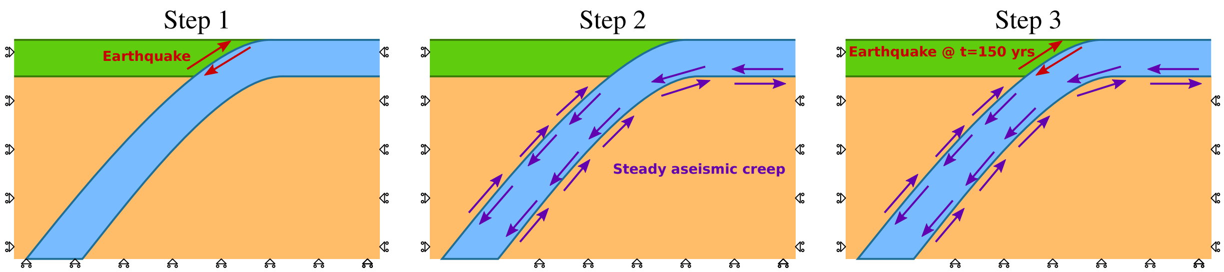



Figure 7.45: Diagram of fault slip and boundary conditions for each step in the subduction zone example.

All of the files necessary to run the examples are contained in the directory `examples/2d/subduction`.

### 7.10.1 Overview

This example examines quasi-static interseismic and coseismic deformation in 2D for a subduction zone (see Figure 7.44). It is based on the 2011 M9.0 Tohoku earthquake off the east coast of Japan. Figure 7.45 shows the three steps of increasing complexity. Step 1 focuses on the coseismic slip, Step 2 focuses on interseismic deformation, and Step 3 combines the two into a pseudo-earthquake cycle deformation simulation. Step 4 focuses on using the change in tractions from Step 1 to construct a simulation with afterslip controlled by frictional sliding. Steps 5 and 6 replace the prescribed aseismic slip on the subducting slab in Step 2 with a frictional interface, producing spontaneous earthquake ruptures and creep.

### 7.10.2 Mesh Description

We construct the mesh in CUBIT by constructing the geometry, prescribing the discretization, running the mesher, and then grouping cells and vertices for boundary conditions and materials. We use the APREPRO programming language within the journal files to enable use of units and to set variables for values used many times. An appendix in the CUBIT documentation discusses the features available with APREPRO in CUBIT. The CUBIT commands are in three separate journal files. The main driver is in the journal file `mesh_tri3.jou`. It calls the journal file `geometry.jou` to construct the geometry and `createbc.jou` to set up the groups associated with boundary conditions and materials. The journal files are documented and describe the various steps outlined below.

1. Create the geometry defining the domain.

   (a) Create points.

   (b) Connect points into spline curves.

   (c) Split curves to separate them into sections bounding surfaces.

   (d) Connect curves into surfaces.

   (e) Stitch surfaces together.

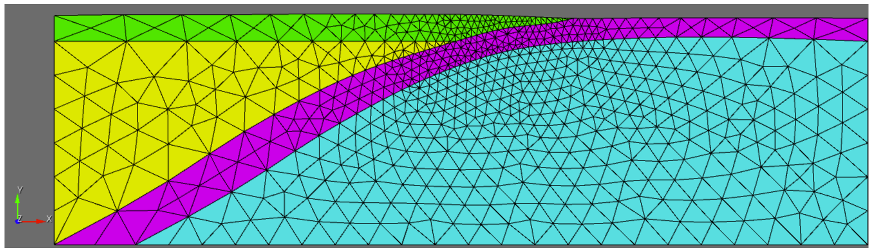


Figure 7.46: Variable resolution finite-element mesh with triangular cells. The nominal cell size increases at a geometric rate of 1.2 away from the region of coseismic slip.

2. Define meshing scheme and cell size variation.
   (a) Define cell size along curves near fault.
   (b) Increase cell size away from fault at a geometric rate (bias).
3. Generate mesh.
4. Create blocks for materials and nodesets for boundary conditions.
5. Export mesh.

### 7.10.3 Common Information

As in the examples discussed in previous sections of these examples, we place parameters common to the three steps in the `pylithapp.cfg` file so that we do not have to duplicate them for each step. The settings contained in `pylithapp.cfg` for this problem consist of:

**pylithapp.journal.info** Settings that control the verbosity of the output written to stdout for the different components.

**pylithapp.mesh_generator** Settings that control mesh importing, such as the importer type, the filename, and the spatial dimension of the mesh.

**pylithapp.timedependent** Settings that control the problem, such as the total time, time-step size, and spatial dimension.

**pylithapp.timedependent.materials** Settings that control the material type, specify which material IDs are to be associated with a particular material type, and give the name of the spatial database containing the physical properties for the material. The quadrature information is also given.

**pylithapp.problem.formulation.output** Settings related output of the solution over the domain and subdomain (ground surface).

**pylithapp.timedependent.materials.*MATERIAL*.output** Settings related to output of the state variables for material *MATERIAL*.

**pylithapp.petsc** PETSc settings to use for the problem, such as the preconditioner type.

The physical properties for each material are specified in spatial database files. For example, the elastic properties for the continental crust are in `mat_concrust.spatialdb`. The provided spatial database files all use just a single point to specify uniform physical properties within each material. A good exercise is to alter the spatial database files with the physical properties to match PREM.

### 7.10.4 Step 1: Coseismic Slip Simulation

The first example problem is earthquake rupture involving coseismic slip along the interface between the subducting slab and the continental crust and uppermost portion of the mantle below the continental crust. The spatial variation of slip comes from a cross-section of Gavin Hayes' finite-source model earthquake.usgs.gov/earthquakes/eqinthenews/

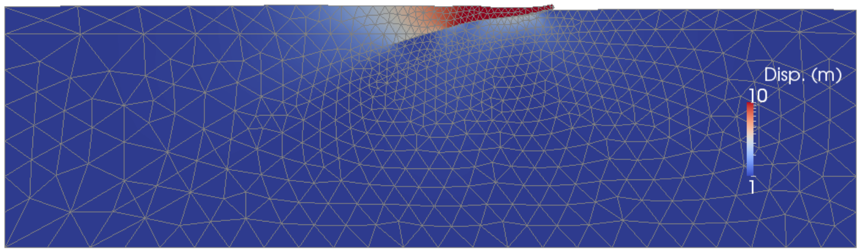



Figure 7.47: Solution for Step 1. The colors indicate the magnitude of the displacement, and the deformation is exaggerated by a factor of 1000.

2011/usc0001xgp/finite_fault.php. On the lateral and bottom boundaries of the domain, we fix the degrees of freedom perpendicular to the boundary as shown in Figure 7.45 on page 175. Parameter settings that augment those in pylithapp.cfg are contained in the file step01.cfg. These settings are:

**pylithapp.timedependent.formulation.time_step** Adjust the total simulation time to 0 years (static simulation).

**pylithapp.timedependent** Specifies the array of boundary conditions.

**pylithapp.timedependent.bc.*BOUNDARY*** Defines the settings for boundary *BOUNDARY*, including which degrees of freedom are being constrained (x or y), the label (defined in mesh_tri3.exo) corresponding to the nodeset in CUBIT, and a label to the boundary condition used in any error messages.

**pylithapp.timedependent.interfaces.fault** Specify the coseismic slip along the interface between the oceanic crust and continental crust with a small amount of slip penetrating into the upper mantle.

**pylithapp.problem.formulation.output.domain** Gives the base filenames for HDF5 output (for example, step01.h5).

Run Step 1 simulation

```
$ pylith step01.cfg
```

The problem will produce twelve pairs of HDF5/Xdmf files. The HDF5 files contain the data and the Xdmf files contain the metadata required by ParaView and Visit (and possibly other visualization tools that use Xdmf files) to access the mesh and data sets in the HDF5 files. The files include the solution over the domain and ground surface (two pairs of files), physical properties, stress, and strain within each material (eight pairs of files), and fault parameters, slip, and traction (two pairs of files).

Figure 7.47, which was created using ParaView, displays the magnitude of the displacement field with the deformation exaggerated by a factor of 1000.

### 7.10.5 Step 2: Interseismic Deformation Simulation

In this example we simulate the interseismic deformation associated with the oceanic crust subducting beneath the continental crust and into the mantle. We prescribe steady aseismic slip of 8 cm/yr along the interfaces between the oceanic crust and mantle with the interface between the oceanic crust and continental crust locked as shown in Figure 7.45 on page 175. We adjust the Dirichlet boundary conditions on the lateral edges and bottom of the domain by pinning only the portions of the boundaries in the mantle and continental crust (i.e., not part of the oceanic crust). Parameter settings that augment those in pylithapp.cfg are contained in the file step02.cfg. These settings include:

**pylithapp.timedependent.formulation.time_step** Adjust the total simulation time to 100 years.

**pylithapp.timedependent** Specifies the array of boundary conditions.

**pylithapp.timedependent.bc.*BOUNDARY*** Defines the settings for boundary *BOUNDARY*, including which degrees of freedom are being constrained (x or y), the label (defined in mesh_tri3.exo) corresponding to the nodeset in CUBIT, and a label to the boundary condition used in any error messages.

**pylithapp.timedependent.interfaces** Specify the steady aseismic slip as a constant slip rate on the fault surfaces.

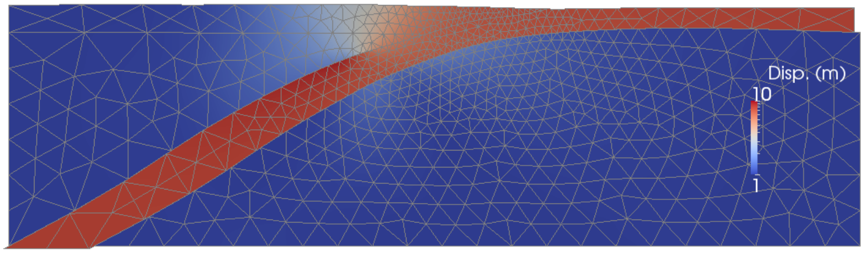



Figure 7.48: Solution for Step 2 at 100 years. The colors indicate the magnitude of the displacement, and the deformation is exaggerated by a factor of 1000.

**pylithapp.problem.formulation.output.domain** Gives the base filename for HDF5 output (for example, `step02.h5`).

Run Step 2 simulation

```
$ pylith step02.cfg
```

The simulation will produce pairs of HDF5/Xdmf files with separate files for each material and fault interface. Figure 7.48, which was created using ParaView, displays the magnitude of the displacement field with the deformation exaggerated by a factor of 1000. Using the animation features within ParaView or Visit you can illustrate how the continental crust near the trench subsides during the interseismic deformation.

### 7.10.6 Step 3: Pseudo-Earthquake Cycle Model

This simulation combines 300 years of interseismic deformation from Step 2 with the coseismic deformation from Step 1 applied at 150 years to create a simple model of the earthquake cycle. Parameter settings that augment those in `pylithapp.cfg` are contained in the file `step03.cfg`. These settings include:

**pylithapp.timedependent.formulation.time_step** Adjust the total simulation time to 300 years.

**pylithapp.timedependent** Specifies the array of boundary conditions.

**pylithapp.timedependent.bc.*BOUNDARY*** The Dirichlet boundary conditions match those in Step 2.

**pylithapp.timedependent.interfaces** On the interface between the subducting oceanic crust and the mantle, we prescribe the same steady, aseismic slip as that in Step 2. On the interface along the top of the subducting oceanic crust and the continental crust and mantle we create two earthquake ruptures, The first rupture applies the coseismic slip form Step 1 at 150 years, while the second rupture prescribes the same steady, aseismic slip as in Step 2.

**pylithapp.problem.formulation.output.domain** Gives the base filename for HDF5 output (for example, `step03.h5`).

We run this example by typing

```
$ pylith step03.cfg
```

The simulation will produce pairs of HDF5/Xdmf files with separate files for each material and fault interface. Figure 7.49 on the next page, which was created using ParaView, displays the magnitude of the displacement field with the deformation exaggerated by a factor of 1000. Using the animation features within ParaView or Visit you can illustrate how the continental crust near the trench rebounds during the earthquake after subsiding during the interseismic deformation.

### 7.10.7 Step 4: Frictional Afterslip Simulation

This simulation demonstrates how to combine the change in tractions associated with coseismic slip with a background stress field to compute afterslip controlled by static friction. The Python script `afterslip_tractions.py` will create a spatial database file with initial tractions based on the change in tractions from Step 1 and a background stress field. The background stress field is simply normal tractions consistent with the overburden (lithostatic load) for a uniform half-space and shear

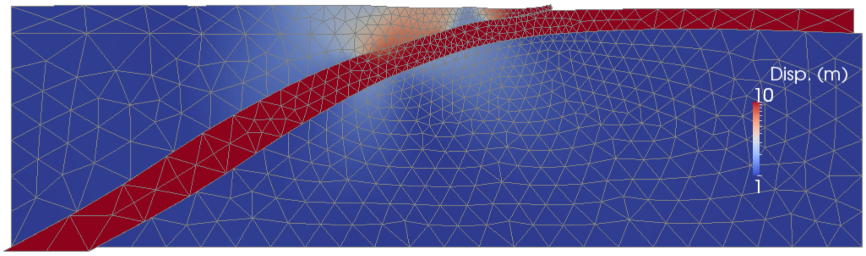



Figure 7.49: Solution for Step 3 at 150 years (immediately following the earthquake rupture). The colors indicate the magnitude of the displacement, and the deformation is exaggerated by a factor of 1000.

tractions consistent with a coefficient of friction of 0.6. The `afterslip_tractions.spatialdb` file is provided, so you do not need to run the Python script `afterslip_tractions.py`; however, you can do so by typing

Optional: Generate `afterslip_tractions.spatialdb`

```
$ python afterslip_tractions.py
```

We provide 2.0 MPa of strength excess associated with the background stress field by using a cohesion of 2.0 MPa in the static friction model. Slip will occur in regions where the coseismic slip increased the shear tractions by more than 2.0 MPa. On the lateral and bottom boundaries of the domain, we fix the degrees of freedom perpendicular to the boundary as shown in Figure 7.45 on page 175. Parameter settings that augment those in `pylithapp.cfg` are contained in the file `step04.cfg`. These settings are:

**pylithapp.timedependent.formulation.time_step** Adjust the total simulation time to 0 years (static simulation).

**pylithapp.timedependent** Selects the nonlinear solver and specifies the array of boundary conditions.

**pylithapp.timedependent.bc.*BOUNDARY*** Defines the settings for boundary *BOUNDARY*, including which degrees of freedom are being constrained (x or y), the label (defined in `mesh_tri3.exo`) corresponding to the nodeset in CUBIT, and a label to the boundary condition used in any error messages.

**pylithapp.timedependent.interfaces.fault** Specify a fault with a fault constitutive model (static friction) and initial fault tractions.

**pylithapp.problem.formulation.output.domain** Gives the base filenames for HDF5 output (for example, `step04.h5`).

Run Step 4 simulation

```
$ pylith step04.cfg
```

The problem will produce twelve pairs of HDF5/Xdmf files. The HDF5 files contain the data and the Xdmf files contain the metadata required by ParaView and Visit (and possibly other visualization tools that use Xdmf files) to access the mesh and data sets in the HDF5 files. The files include the solution over the domain and ground surface (two pairs of files), physical properties, stress, and strain within each material (eight pairs of files), and fault parameters, slip, and traction (two pairs of files).

Figure 7.50 on the next page, which was created using ParaView, displays the magnitude of the displacement field with the original configuration. Slip occurs down-dip from the coseismic slip as well as in three areas with sharp gradients in slip, including the trench. The location of the afterslip can be shifted by changing the spatial variation of the coseismic slip and background stress field.

### 7.10.8 Step 5: Spontaneous Earthquakes With Slip-Weakening Friction

We simulate earthquake cycles over 100 years with spontaneous rupture using slip-weakening friction. As in Step 4 including fault friction requires the nonlinear solver. Through trial and error we choose a time step of 2.5 years that permits reasonable convergence of the nonlinear solver and runtime.

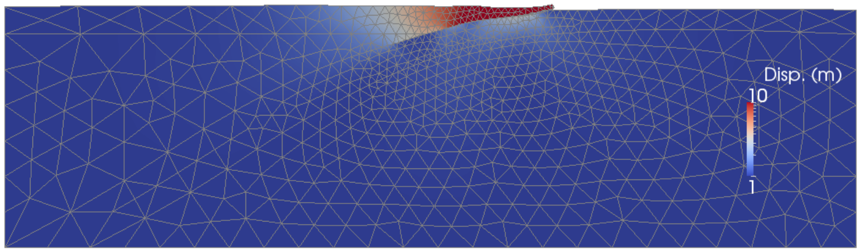



Figure 7.50: Solution for Step 4. The colors indicate the magnitude of the displacement.

Excerpt from step05.cfg

```
[pylithapp.problem.formulation]
# Fault friction is a nonlinear problem so we need to use the
# nonlinear solver.
solver = pylith.problems.SolverNonlinear

[pylithapp.timedependent.formulation.time_step]
total_time = 100.0*year
dt = 2.5*year
```

In simulations for research purposes, we would use a higher resolution mesh and smaller time steps and investigate the robustness of the solution to these parameters.

We constrain the displacement normal to the lateral and bottom boundaries without restraining the subducting slab. We also constrain the vertical deformation of the west boundary to facilitate the downward motion of the subducting slab.

Excerpt from step05.cfg

```
[pylithapp.timedependent.bc.boundary_west]
bc_dof = [0, 1]
label = bndry_west
db_initial.label = Dirichlet BC on west boundary
```

We replace the prescribed aseismic slip on the subduction interface that we used in Step 2 with a friction interface with the slip-weakening fault constitutive model.

Excerpt from step05.cfg

```
[pylithapp.timedependent]
interfaces = [fault_slabtop, fault_slabbot]

# Set the type of fault interface conditions.
[pylithapp.timedependent.interfaces]
fault_slabtop = pylith.faults.FaultCohesiveDyn
fault_slabbot = pylith.faults.FaultCohesiveKin
```

In order to generate stick-slip events, we need the coefficient of friction to decrease with slip. We choose a slip-weakening friction model with a dynamic coefficient of friction that is less than the static coefficient of friction to provide this behavior. In quasistatic modeling we use time steps much longer than the slip rise time in an earthquake, so we want the slip confined to one time step or just a few time steps. This means the drop in the coefficient of friction should be independent in each time step; that is, we want the fault to fully heal between time steps. This corresponds to setting the **force_healing** property of the SlipWeakening object.

A common feature in numerical modeling of subduction zones is stable sliding near the trench and below the seismogenic zone. We implement stable sliding with the slip-weakening friction via a constant coefficient of friction (equal values for the static and dynamic coefficients of friction). We create a lower dynamic coefficient of friction in the seismogenic zone, by introducing depth-dependent variations in the dynamic coefficient of friction. using a SimpleGridDB spatial database. As discussed in Section 4.5 on page 43 This provides more efficient interpolation compared to the SimpleDB implementation. We impose initial tractions on the fault in a similar fashion as we did in Step 4. We reduce the initial shear tractions slightly in the seismogenic zone, consistent with a stress drop in the penultimate earthquake followed by loading during the interseismic period.

Excerpt from `step05.cfg`

```
[pylithapp.timedependent.interfaces.fault_slabtop]
# --- Skipping general information discussed previously ---
# Friction
friction = pylith.friction.SlipWeakening
friction.label = Slip weakening
# Force healing after each time step, so weakening is confined to each
# time step and is not carried over into subsequent time steps.
friction.force_healing = True

friction.db_properties = spatialdata.spatialdb.SimpleGridDB
friction.db_properties.label = Slip weakening
friction.db_properties.filename = fault_slabtop_slipweakening.spatialdb

# Initial fault tractions
traction_perturbation = pylith.faults.TractPerturbation
traction_perturbation.db_initial = spatialdata.spatialdb.SimpleGridDB
traction_perturbation.db_initial.label = Initial fault tractions
traction_perturbation.db_initial.filename = fault_slabtop_tractions.spatialdb
```

We adjust several of the solver tolerances. In general, we impose larger tolerances to reduce runtime at the expense of a less accurate solution. We set the zero tolerances for detecting slip and suppressing fault opening to $1.0 \times 10^{-8}$. We want tolerances for the linear solve to be smaller than these values, so we use an absolute tolerance of $1.0 \times 10^{-9}$ and a very small relative tolerance to force the residual below the absolute tolerance. We impose an absolute tolerance for the nonlinear solver to be greater than our zero tolerances and also force the residual to match the absolute tolerance level by using a very small relative tolerances. Finally, we set the parameters for the solver used to calculate consistent values for the change in slip for a given change in the Lagrange multipliers (which we sometimes call the friction sensitivity solve).

Excerpt from `step05.cfg`

```
[pylithapp.timedependent.interfaces.fault_slabtop]
zero_tolerance = 1.0e-8
zero_tolerance_normal = 1.0e-8

# Convergence parameters.
ksp_rtol = 1.0e-20
ksp_atol = 1.0e-9
ksp_max_it = 1000

snes_rtol = 1.0e-20
snes_atol = 1.0e-7
snes_max_it = 1000

# Friction sensitivity solve used to compute the increment in slip
# associated with changes in the Lagrange multiplier imposed by the
# fault constitutive model.
friction_pc_type = asm
friction_sub_pc_factor_shift_type = nonzero
friction_ksp_max_it = 25
friction_ksp_gmres_restart = 30
friction_ksp_error_if_not_converged = true
```

Run Step 5 simulation

```
$ pylith step05.cfg
```

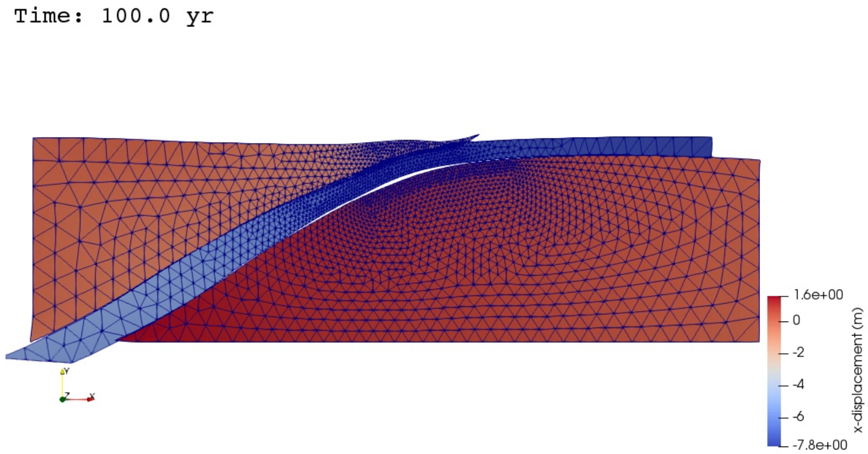



Figure 7.51: Solution for Step 5 at the end of the simulation. The colors indicate the magnitude of the x-displacement component and the deformation has been exaggerated by a factor of 10,000.

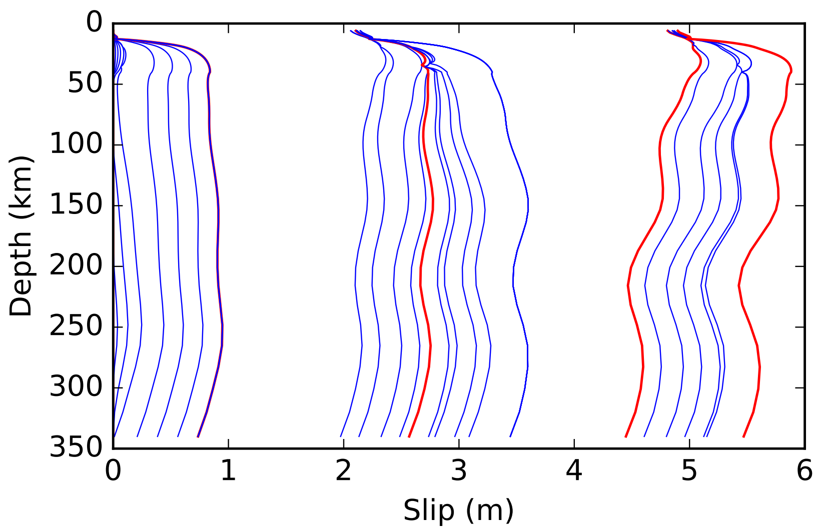



Figure 7.52: Cumulative slip as a function of time and depth in Step 5. The red lines indicate slip every 10 time steps.

The problem will produce fourteen pairs of HDF5/Xdmf files. Figure 7.51, which was created using the ParaView Python script `viz/plot_dispwarp.py` (see Section 7.2 on page 112 for a discussion of how to run ParaView Python scripts), displays the magnitude of the velocity field with the original configuration exaggerated by a factor of 4000. Steady slip is largely confined to the stable sliding regions with a sequence of ruptures in the seismogenic zone; most have a duration of a few time steps, although most of the slip occurs in a single time step. Figure 7.52 shows the cumulative slip as a function of time and distance down dip from the trench.

### 7.10.9 Step 6: Spontaneous Earthquakes With Rate-State Friction

In this example we replace the slip-weakening in Step 5 with rate- and state-friction using the ageing law. We also lengthen the duration of the simulation to 200 years and reduce the time step to 1.0 years, which were determined through trial and error to get a couple earthquake cycles with reasonable convergence for this relatively coarse resolution mesh.

Excerpt from `step06.cfg`

```
[pylithapp.timedependent.formulation.time_step]
total_time = 200.0*year
dt = 1.0*year
```

The specification of the parameters for the rate- and state-friction model follow a similar pattern to the ones for the slip-weakening friction in Step 5. Our regularization of the coefficient of friction for near zero slip rate values involves a transition to a linear dependence on slip rate; in this example we specify that this transition should occur at a nondimensional slip rate of $1.0 \times 10^{-6}$. We impose depth variation of the friction model parameters via a SimpleGridDB spatial database in order to generate earthquake-like ruptures in the seismogenic zone with stable sliding above and below. For the initial tractions, we impose uniform values using a SimpleDB spatial database. We set the initial state for the friction model to be roughly consistent with steady state sliding at the reference coefficient of friction at the reference slip rate, and include it in the state variable in the output as a check.

Excerpt from `step06.cfg`

```
[pylithapp.timedependent.interfaces.fault_slabtop]
# --- Skipping parameters discussed in previous examples. ---
# Friction
friction = pylith.friction.RateStateAgeing
friction.label = Rate-state friction
# Nondimensional slip rate below which friction depends linearly on slip rate.
friction.linear_slip_rate = 1.0e-6

# Set spatial database for distribution of friction parameters
friction.db_properties = spatialdata.spatialdb.SimpleGridDB
friction.db_properties.label = Slip weakening
friction.db_properties.filename = fault_slabtop_ratestate.spatialdb

# Set spatial database for the initial value of the state variable.
friction.db_initial_state = spatialdata.spatialdb.UniformDB
friction.db_initial_state.label = Rate State Ageing State
friction.db_initial_state.values = [state-variable]
# theta_ss = characteristic_slip_dist / reference_slip_rate
friction.db_initial_state.data = [20.0*year]

# Initial fault tractions
traction_perturbation = pylith.faults.TractPerturbation
traction_perturbation.db_initial = spatialdata.spatialdb.UniformDB
traction_perturbation.db_initial.label = Initial fault tractions
traction_perturbation.db_initial.values = [traction-shear, traction-normal]
traction_perturbation.db_initial.data = [-12.0*MPa, -20.0*MPa]

[pylithapp.problem.interfaces.fault_slabtop.output]
writer = pylith.meshio.DataWriterHDF5
writer.filename = output/step06-fault-slabtop.h5
vertex_info_fields = [normal_dir, strike_dir]
vertex_data_fields = [slip, slip_rate, traction, state_variable]
```

Run Step 6 simulation

```
$ pylith step06.cfg
```

The problem will produce fourteen pairs of HDF5/Xdmf files. Figure 7.53 on the next page, which was created using the ParaView Python script `viz/plot_dispwarp.py`, displays the magnitude of the velocity field with the original configuration exaggerated by a factor of 4000. Steady slip is largely confined to the stable sliding regions with a sequence of ruptures in the seismogenic zone; note how the rate-state friction allows a more natural nucleation of the ruptures compared to the slip-weakening friction. Figure 7.52 shows the cumulative slip as a function of time and distance down dip from the trench.

### 7.10.10 Exercises

The list below includes some suggested modifications to these examples that will allow you to become more familiar with PyLith while examining some interesting physics.

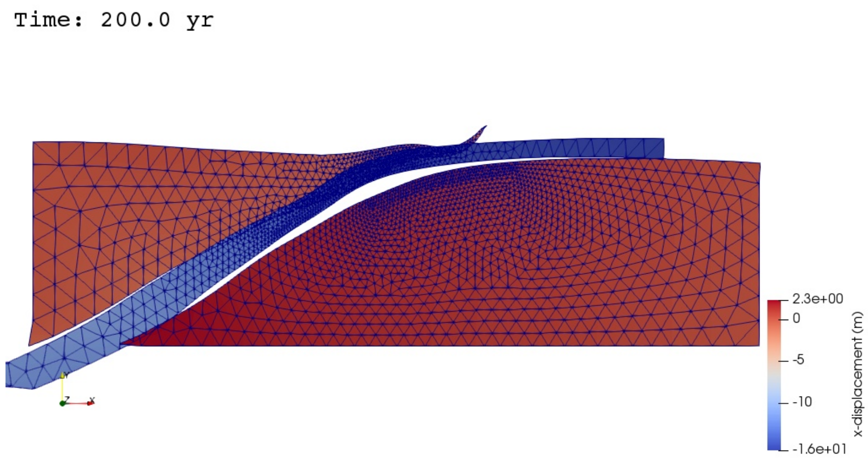



Figure 7.53: Solution for Step 6 at the end of the simulation. The colors indicate the magnitude of the x-displacement component and the deformation has been exaggerated by a factor of 10,000.

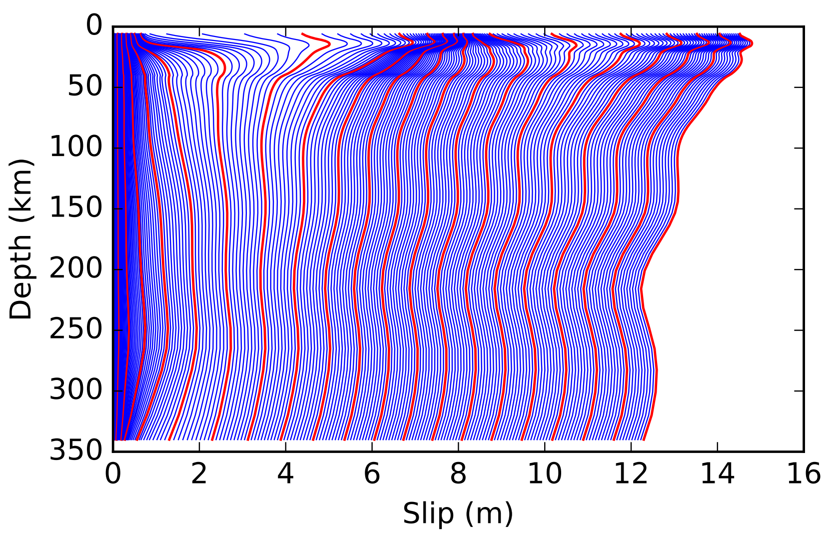



Figure 7.54: Cumulative slip as a function of time and depth in Step 6. The red lines indicate slip every 10 time steps.

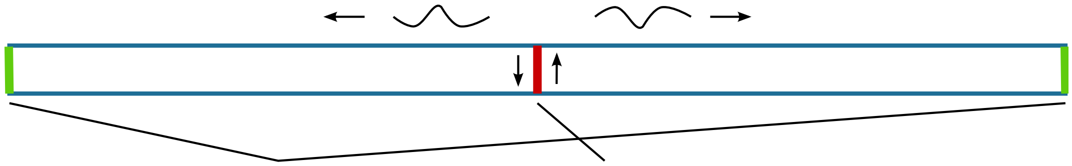



Figure 7.55: Domain for shear wave propagation in a 8.0 km bar with 400 m cross-section. We generate a shear wave via slip on a fault located in the middle of the bar while limiting deformation to the transverse direction.

- Change the resolution of the mesh by editing the `mesh_tri3.jou` journal file. Change the resolution and bias factor.
- Add depth dependent viscosity to the mantle and crust. This requires using the linear Maxwell plane strain bulk constitutive model in the crust as well and creating spatial databases that include viscosity for the crust. Specifying a depth dependent variation in the parameters will require adding points, updating num-locs accordingly, and changing data-dim to 1.
- Modify the spatial database files for the material properties to use depth-dependent elastic properties based on PREM (Dziewonski and Anderson, 1981, 10.1016/0031-9201(81)90046-7). See `geophysics.ou.edu/solid_earth/prem.html` for a simple table of values. Add points, update num-locs accordingly, and change data-dim to 1.
- Modify the CUBIT journal files to use quad4 cells rather than tri3 cells. This requires using the pave mesh scheme.
- Modify Steps 5 and 6 to use a user-defined variable time step. Experiment with longer time steps between earthquake ruptures and smaller time steps around the time of the earthquake ruptures. Can you develop a simple algorithm for choosing the time step?
- Adjust the parameters of the friction models and examine the effects on the deformation and the convergence of the nonlinear solve. In which cases do you need to adjust the time step to retain reasonable convergence?

## 7.11 Shear Wave in a Bar

This suite of examples focuses on the dynamics of a shear wave propagating down an 8 km-long bar with a 400 m-wide cross-section. Motion is limited to shear deformation by fixing the longitudinal degree of freedom. For each cell type (tri3, quad4, tet4, and hex8) we generate a shear wave using a kinematic fault rupture with simultaneous slip over the fault surface, which we place at the center of the bar. The discretization size is 200 m in all cases. The slip-time histories follow the integral of Brune's far-field time function with slip initiating at 0.1 s, a left-lateral final slip of 1.0 m, and a rise time of 2.0 s. The shear wave speed in the bar is 1.0 km/s, so the shear wave reaches each end of the bar at 4.1 s. Absorbing boundaries on the ends of the bar prevent significant reflections. The bar comes to a rest with a static offset.

For the bar discretized with quad4 cells we also consider the fault subjected to frictional sliding controlled by static friction, linear slip-weakening friction, and rate- and state-friction. We use initial tractions applied to the fault to drive the dislocation and generate the shear wave. Because the fault tractions are constant in time, they continue to drive the motion even after the shear wave reaches the absorbing boundary, leading to a steady state solution with uniform shear deformation in the bar and a constant slip rate on the fault.

## 7.12 2D Bar Discretized with Triangles

PyLith features discussed in this example:

- Dynamic solution
- CUBIT format
- Absorbing dampers boundary conditions
- Kinematic fault interface conditions
- Plane strain linearly elastic material

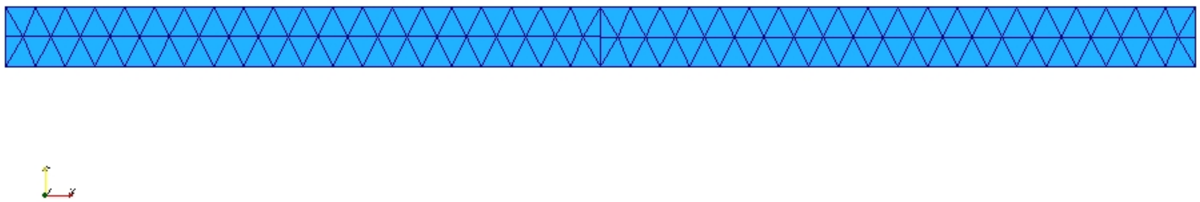


Figure 7.56: Mesh composed of triangular cells generated by CUBIT used for the example problem.

- VTK output
- Linear triangular cells
- SimpleDB spatial database
- ZeroDispDB spatial database

All of the files necessary to run the examples are contained in the directory `examples/bar_shearwave/tri3`.

### 7.12.1 Mesh Generation

The mesh is a simple rectangle 8 km by 400 m (Figure 7.63 on page 192). This mesh could be generated via a simple script, but it is even easier to generate this mesh using CUBIT. We provide documented journal files in `examples/bar_shearwave/tri3`. We first create the geometry, mesh the domain using triangular cells, and then create blocks and nodesets to associate the cells and vertices with materials and boundary conditions. See Section 7.9 on page 139 for more information on using CUBIT to generate meshes.

### 7.12.2 Simulation Parameters

All of the parameters are set in the `pylithapp.cfg` file. The structure of the file follows the same pattern as in all of the other examples. We set the parameters for the journal information followed by the mesh reader, problem, materials, boundary conditions, fault, and output. We change the time-stepping formulation from the default value of implicit time stepping to explicit time stepping with a lumped Jacobian matrix by setting the formulation object via

```
formulation = pylith.problems.Explicit
```

Using the Explicit object automatically triggers lumping of the Jacobian cell matrices and assembly into a vector rather than a sparse matrix. Lumping the Jacobian decouples the equations, so we can use a very simple direct solver. Use of this simple solver is also triggered by the selection of any of the Explicit formulation objects.

For dynamic problems we use the NondimElasticDynamic object to nondimensionalize the equations. This object provides scales associated with wave propagation for nondimensionalization, including the minimum wave period, the shear wave speed, and mass density. In this example we use the default values of a minimum wave period of 1.0 s, a shear wave speed of 3 km/s, and a mass density of 3000 kg/m$^3$. We simulate 12.0 s of motion with a time step of 1/30 s. This time step must follow the Courant-Friedrichs-Lewy condition; that is, the time step must be smaller than the time it takes the P wave to propagate across the shortest edge of a cell.

The boundary conditions include the absorbing dampers at the ends of the bar and a Dirichlet boundary condition to prevent longitudinal motion. Because we cannot overlap the Dirichlet BC with the fault, we use the nodeset associated with all vertices except the fault. For the output over the entire domain, we request both displacement and velocity fields:

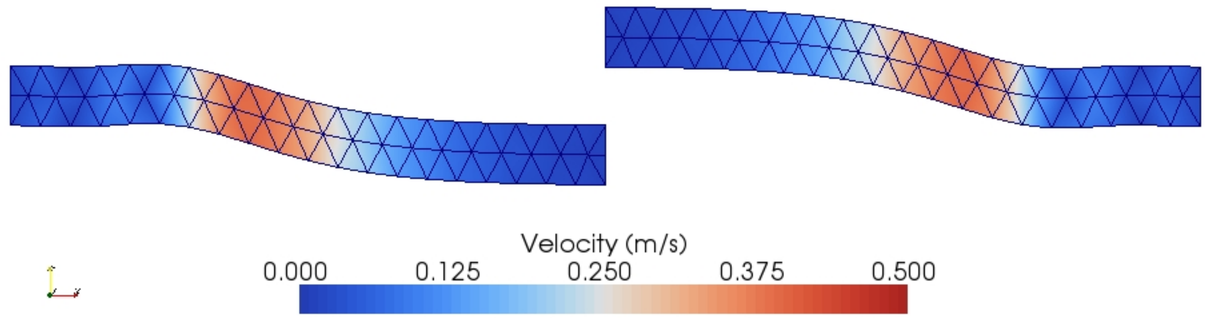



Figure 7.57: Displacement field in the bar at 3.0 s. Deformation has been exaggerated by a factor of 800.

```
[pylithapp.timedependent.output]
vertex_data_fields = [displacement, velocity]
```

To run the problem, simply run PyLith without any command line arguments:

```
$ pylith
```

The VTK files will be written to the `output` directory. The output includes the displacement and velocity fields over the entire domain at every 3rd time step (0.10 s), the slip and change in traction vectors on the fault surface in along-strike and normal directions at every 3rd time step (0.10 s), and the strain and stress tensors for each cell at every 30th time step (1.0 s). If the problem ran correctly, you should be able to generate a figure such as Figure 7.57, which was generated using ParaView.

## 7.13 3D Bar Discretized with Quadrilaterals

PyLith features discussed in this example:

- Dynamic solution
- CUBIT mesh format
- Absorbing dampers boundary conditions
- Kinematic fault interface conditions
- Dynamic fault interface conditions
- Plane strain linearly elastic material
- VTK output
- Linear quadrilateral cells
- SimpleDB spatial database
- ZeroDispDB spatial database
- UniformDB spatial database

All of the files necessary to run the examples are contained in the directory `examples/bar_shearwave/quad4.`

### 7.13.1 Mesh Generation

The mesh is a simple rectangular prism 8 km by 400 m by 400 m (Figure 7.58 on the next page). We provide documented CUBIT journal files in `examples/bar_shearwave/quad4.` We first create the geometry, mesh the domain using quadrilateral cells, and then create blocks and nodesets associated with the materials and boundary conditions.

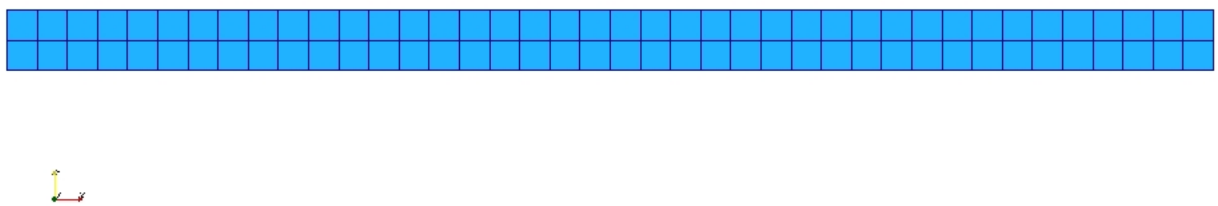


Figure 7.58: Mesh composed of hexahedral cells generated by CUBIT used for the example problem.

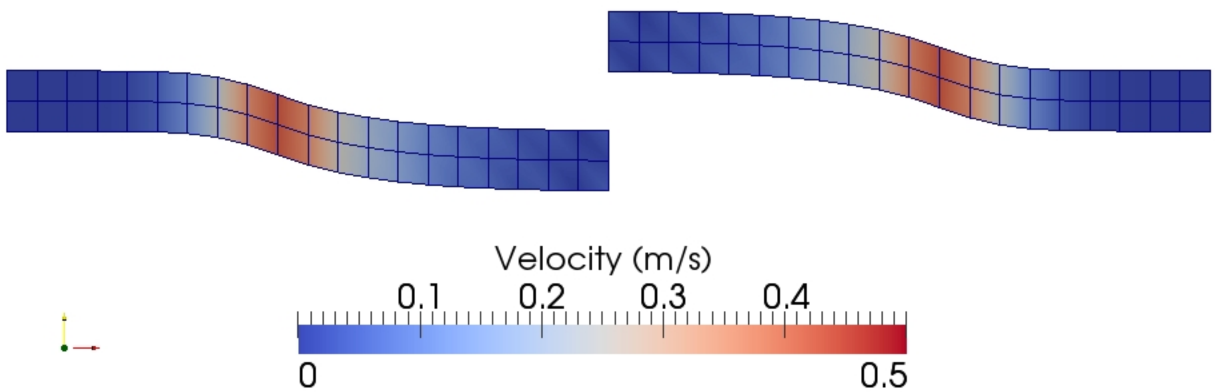



Figure 7.59: Displacement field in the bar at 3.0 s. Deformation has been exaggerated by a factor of 800.

### 7.13.2 Kinematic Fault (Prescribed Slip)

The simulation parameters match those in the tri3, tet4, and hex8 examples. Using four-point quadrature permits use of a time step of 1/20 s, which is slightly larger than the time step of 1/30 s used in the tri3 and tet4 simulations. In contrast to the tri3, tet4, and hex8 shear wave examples which only contained a single simulation in a directory, in this example we consider several different simulations. Consequently, we separate the parameters into multiple `cfg` files. The common parameters are placed in `pylithapp.cfg` with the parameters specific to the kinematic fault (prescribed rupture) example in `prescribedrup.cfg`. To run the problem, simply run PyLith via:

```
$ pylith prescribedrup.cfg
```

The VTK files will be written to the `output` directory with the prefix `prescribedrup`. The output includes the displacement field over the entire domain at every other time step (0.10 s), the slip and traction vectors on the fault surface in along-strike and normal directions at every other time step (0.10 s), and the strain and stress tensors for each cell at every 20th time step (1.0 s). If the problem ran correctly, you should be able to generate a figure such as Figure 7.59, which was generated using ParaView.

### 7.13.3 Dynamic Fault (Spontaneous Rupture)

In this set of examples we replace the kinematic fault interface with the dynamic fault interface, resulting in fault slip controlled by a fault-constitutive model. See Section 6.4.5.3 on page 105 for detailed information about the fault constitutive models available in PyLith. Because this is a dynamic simulation we want the generated shear wave to continue to be absorbed at the ends of the bar, so we drive the fault by imposing initial tractions directly on the fault surface rather than through deformation within the bar. We impose initial tractions (75 MPa of right-lateral shear and 120 MPa of compression) plus a temporal variation (smoothly increasing from 0 to 25 MPa of right-lateral shear) similar to what would be used in a 2-D or 3-D version. While the magnitude of these stresses is reasonable for tectonic problems, they give rise to very large slip rates in this 1-D bar. The temporal variation, as specified via the `traction_change.timedb` file, has the functional form:

$$f(t) = \begin{cases} \exp\left(\frac{(t-t_n)^2}{t(t-2t_n)}\right), & 0 < t \le t_n \\ 1, & t > t_n \end{cases} \tag{7.2}$$

where $t_n = 1.0$ s. We request that the fault output include the initial traction value and the slip, slip rate, and traction fields:

```
[pylithapp.timedependent.interfaces.fault.output]
vertex_info_fields = [traction_initial_value]
vertex_data_fields = [slip, slip_rate, traction]
```

The steady-state solution for this problem is constant velocity and slip rate with uniform strain within the bar. A Python script, `analytical_soln.py`, is included for computing values related to the steady-state solution.

#### 7.13.3.1 Dynamic Fault with Static Friction

The parameters specific to this example involve the static friction fault constitutive model. We set the fault constitutive model via

```
[pylithapp.timedependent.interfaces.fault]
friction = pylith.friction.StaticFriction
```

and use a UniformDB to set the static friction parameters. We use a coefficient of friction of 0.6 and no cohesion (0 MPa). The parameters specific to this example are in `spontaneousrup_staticfriction.cfg`, so we run the problem via:

```
$ pylith spontaneousrup.cfg spontaneousrup_staticfriction.cfg
```

The VTK files will be written to the `output` directory with the prefix `staticfriction`. The output includes the displacement and velocity fields over the entire domain at every other time step (0.10 s), the slip, slip rate, and traction vectors on the fault surface in along-strike and normal directions at every other time step (0.10 s), and the strain and stress tensors for each cell at every 20th time step (1.0 s). If the problem ran correctly, you should be able to generate a figure such as Figure 7.60 on the following page, which was generated using ParaView. The steady-state solution is a constant slip rate of 22.4 m/s, a shear traction of 72.0 MPa on the fault surface, a uniform shear strain of 5.6e-3 in the bar with uniform, and constant velocities in the y-direction of +11.2 m/s and -11.2 m/s on the -x and +x sides of the fault, respectively.

#### 7.13.3.2 Dynamic Fault with Slip-Weakening Friction

The parameters specific to this example are related to the use of the slip-weakening friction fault constitutive model (see Section 6.4.5.3 on page 105). We set the fault constitutive model via

```
[pylithapp.timedependent.interfaces.fault]
friction = pylith.friction.SlipWeakening
```

and use a UniformDB to set the slip-weakening friction parameters. We use a static coefficient of friction of 0.6, a dynamic coefficient of friction of 0.5, a slip-weakening parameter of 0.2 m, and no cohesion (0 MPa). The fault constitutive model is associated with the fault, so we can append the fault constitutive model parameters to the vertex information fields:

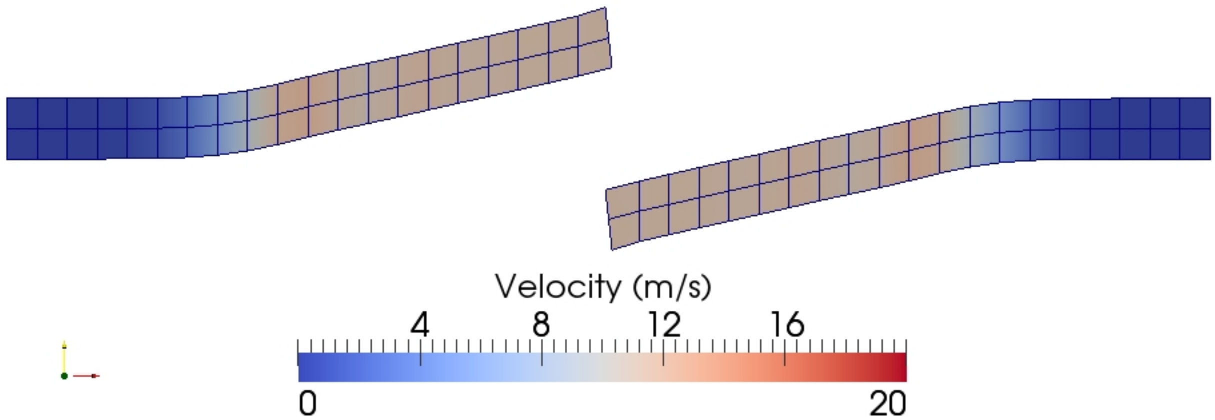



Figure 7.60: Velocity field in the bar at 3.0 s for the static friction fault constitutive model. Deformation has been exaggerated by a factor of 20.

```
[pylithapp.timedependent.interfaces.fault.output]
vertex_info_fields = [strike_dir, normal_dir, initial_traction, static_coefficient, dynamic_coefficient,
```

The parameters specific to this example are in `spontaneousrup_slipweakening.cfg`, so we run the problem via:

```
$ pylith spontaneousrup.cfg spontaneousrup_slipweakening.cfg
```

The VTK files will be written to the `output` directory with the prefix `slipweakening`. If the problem ran correctly, you should be able to generate a figure such as Figure 7.61 on the facing page, which was generated using ParaView. The steady-state solution is a constant slip rate of 32.0 m/s and shear traction of 60.0 MPa on the fault surface, a uniform shear strain of 8.0e-3 in the bar with uniform, constant velocities in the y-direction of +16.0 m/s and -46.0 m/s on the -x and +x sides of the fault, respectively.

#### 7.13.3.3 Dynamic Fault with Rate-State Friction

The parameters specific to this example are related to the use of the rate- and state-friction fault constitutive model (see Section 6.4.5.3 on page 105). The evolution of the state variable uses the ageing law. We set the fault constitutive model and add the state variable to the output via

```
[pylithapp.timedependent.interfaces.fault]
friction = pylith.friction.RateStateAgeing

[pylithapp.timedependent.interfaces.fault.output]
vertex_data_fields = [slip, slip_rate, traction, state_variable]
```

and use a UniformDB to set the rate-state friction parameters. We use a reference coefficient of friction of 0.6, reference slip rate of 1.0e-6 m/s, characteristic slip distance of 0.02 m, coefficients a and b of 0.008 and 0.012, and no cohesion (0 MPa). We set the initial value of the state variable so that the fault is in equilibrium for the initial tractions. The parameters specific to this example are in `spontaneousrup_ratestateageing.cfg`, so we run the problem via:

```
$ pylith spontaneousrup.cfg spontaneousrup_ratestateageing.cfg
```

The VTK files will be written to the `output` directory with the prefix `ratestateageing`. If the problem ran correctly, you should be able to generate a figure such as Figure 7.62 on page 192, which was generated using ParaView. The steady-state solution is a constant slip rate of 30.0 m/s and shear traction of 63.7 MPa on the fault surface, a uniform shear strain of 7.25e-3

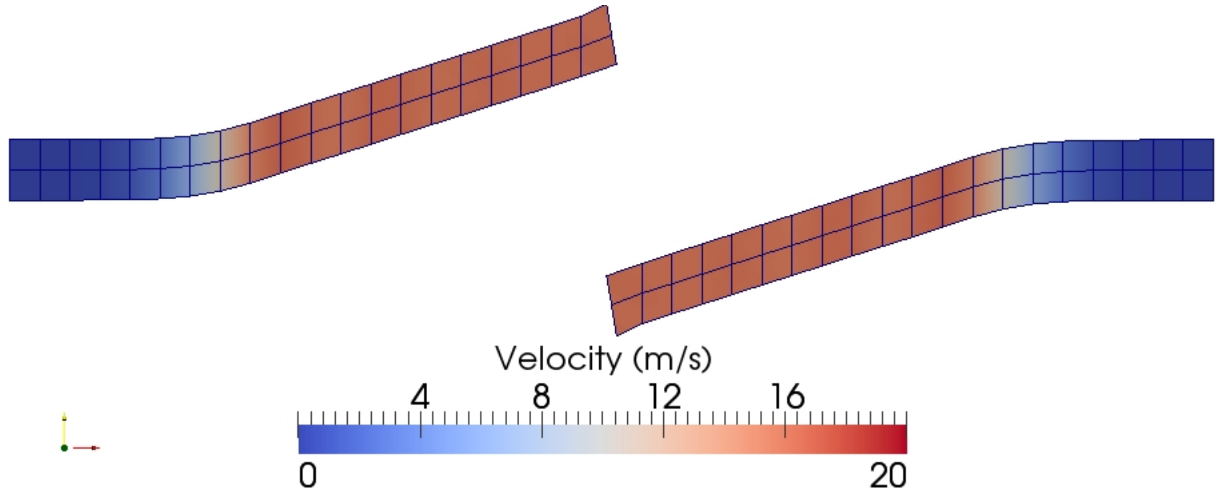



Figure 7.61: Velocity field in the bar at 3.0 s for the slip-weakening friction fault constitutive model. Deformation has been exaggerated by a factor of 20.

in the bar with uniform, constant velocities in the y-direction of +15.0 m/s and -15.0 m/s on the -x and +x sides of the fault, respectively.

## 7.14 3D Bar Discretized with Tetrahedra

PyLith features discussed in this example:

- Dynamic solution
- LaGriT mesh format
- Absorbing dampers boundary conditions
- Kinematic fault interface conditions
- Elastic isotropic linearly elastic material
- VTK output
- Linear tetrahedral cells
- SimpleDB spatial database
- ZeroDispDB spatial database

All of the files necessary to run the examples are contained in the directory `examples/bar_shearwave/tet4.`

### 7.14.1 Mesh Generation

The mesh is a simple rectangular prism 8 km by 400 m by 400m (Figure 7.63 on the next page). This mesh could be generated via a simple script, but it is even easier to generate this mesh using LaGriT. We provide documented LaGriT files in `examples/bar_shearwave/tet4.` We first create the geometry and regions, mesh the domain using tetrahedral cells, and then create point sets associated with boundary conditions.

### 7.14.2 Simulation Parameters

The simulation parameters match those in the tri3 example with the exception of using the LaGriT mesh reader and switching from a two-dimensional problem to a three-dimensional problem. In addition to fixing the longitudinal degree of freedom, we

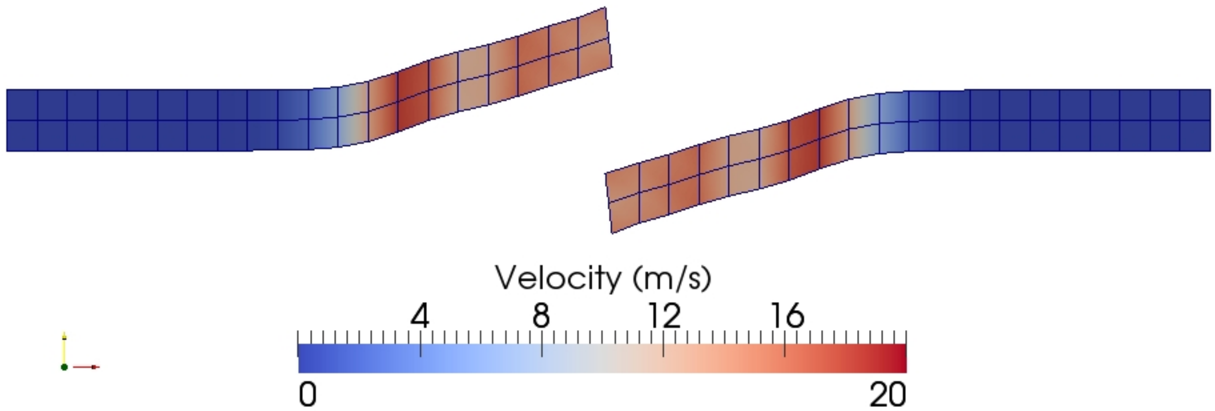



Figure 7.62: Velocity field in the bar at 3.0 s for the rate- and state-friction fault constitutive model. Deformation has been exaggerated by a factor of 20.

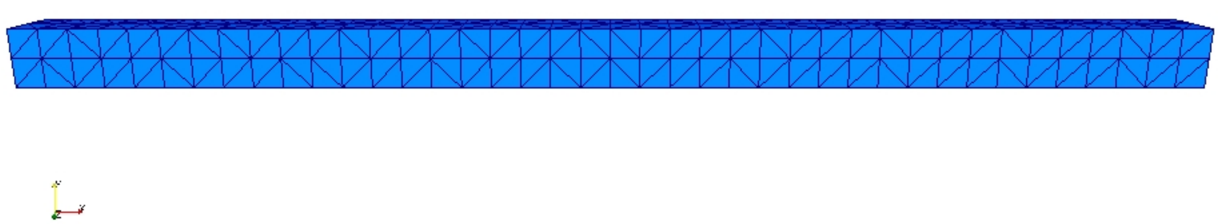


Figure 7.63: Mesh composed of tetrahedral cells generated by LaGriT used for the example problem.

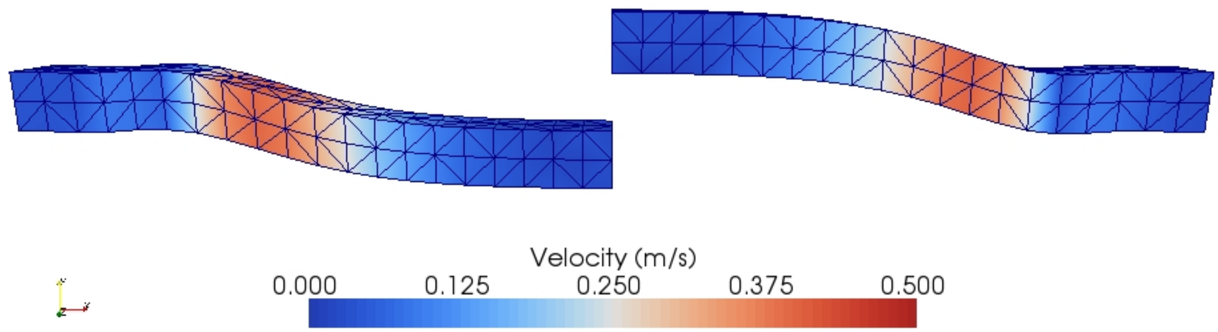



Figure 7.64: Displacement field in the bar at 3.0 s. Deformation has been exaggerated by a factor of 800.

also fix the out-of-plane transverse degree of freedom. Because the fault separates two material regions in LaGriT, we use two materials in PyLith. All of the parameters are set in the `pylithapp.cfg` file. To run the problem, simply run PyLith without any command line arguments:

```
$ pylith
```

The VTK files will be written to the `output` directory. The output includes the displacement and velocity fields over the entire domain at every 3rd time step (0.10 s), the slip and change in traction vectors on the fault surface in along-strike and normal directions at every 3rd time step (0.10 s), and the strain and stress tensors for each cell at every 30th time step (1.0 s). If the problem ran correctly, you should be able to generate a figure such as Figure 7.64, which was generated using ParaView.

## 7.15 3D Bar Discretized with Hexahedra

PyLith features discussed in this example:

- Dynamic solution
- CUBIT mesh format
- Absorbing dampers boundary conditions
- Kinematic fault interface conditions
- Elastic isotropic linearly elastic material
- VTK output
- Linear hexahedral cells
- SimpleDB spatial database
- ZeroDispDB spatial database

All of the files necessary to run the examples are contained in the directory `examples/bar_shearwave/hex8.`

### 7.15.1 Mesh Generation

The mesh is a simple rectangular prism 8 km by 400 m by 400 m (Figure 7.65 on the next page). This mesh could be generated via a simple script, but it is even easier to generate this mesh using CUBIT. We provide documented CUBIT journal files in `examples/bar_shearwave/hex8.` We first create the geometry, mesh the domain using hexahedral cells, and then create blocks and nodesets associated with the materials and boundary conditions.

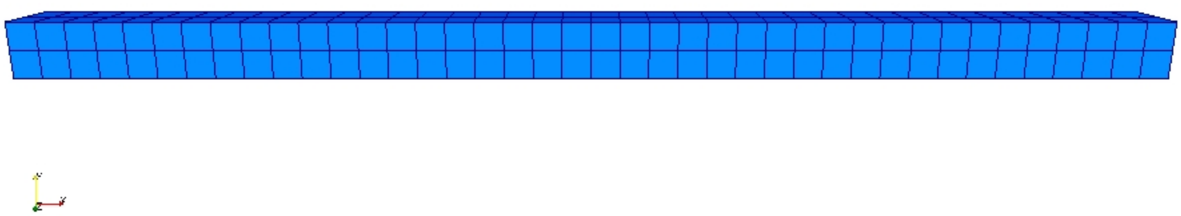


Figure 7.65: Mesh composed of hexahedral cells generated by CUBIT used for the example problem.

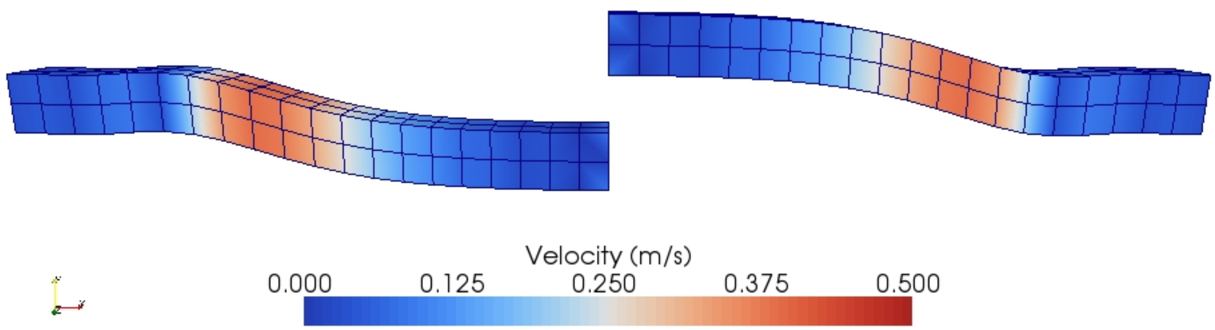



Figure 7.66: Displacement field in the bar at 3.0 s. Deformation has been exaggerated by a factor of 800.

### 7.15.2 Simulation Parameters

The simulation parameters match those in the tri3 and tet4 examples. As in the tet4 example, we fix both the longitudinal degree of freedom and the out-of-plane transverse degree of freedom. Using eight-point quadrature permits use of a time step of 1/20 s, which is slightly larger than the time step of 1/30 s used in the tri3 and tet4 simulations. All of the parameters are set in the `pylithapp.cfg` file. To run the problem, simply run PyLith without any command line arguments:

```
$ pylith
```

The VTK files will be written to the `output` directory. The output includes the displacement and velocity fields over the entire domain at every other time step (0.10 s), the slip and change in traction vectors on the fault surface in along-strike and normal directions at every other time step (0.10 s), and the strain and stress tensors for each cell at every 20th time step (1.0 s). If the problem ran correctly, you should be able to generate a figure such as Figure 7.66, which was generated using ParaView.

## 7.16 Example Generating and Using Green's Functions in Two Dimensions

PyLith features discussed in this example:

- Green's functions
- HDF5 output
- HDF5 point output

- Reading HDF5 output using h5py
- Simple inversion procedure
- Plotting results using matplotlib
- Cubit mesh generation
- Variable mesh resolution
- APREPRO programming language
- Static solution
- Linear triangular cells
- Kinematic fault interface conditions
- Plane strain linearly elastic material
- SimpleDB spatial database
- ZeroDispDB spatial database
- UniformDB spatial database

All of the files necessary to run the examples are contained under the directory `examples/2d/greensfns.`

### 7.16.1 Overview

This example examines the steps necessary to generate Green's functions using PyLith and how they may be used in a linear inversion. For simplicity we discuss strike-slip and reverse faulting examples in the context of 2D simulations. In each example, we first compute surface displacement at a set of points, and these computed displacements provide the "data" for our inversion. Second, we compute a set of Green's functions using the same fault geometries, and output the results at the same set of points. Third, we perform a simple linear inversion. An important aspect for both the forward problem and the Green's function problem is that the computed solution is output at a set of user-specified points (not necessarily coincident with mesh vertices), rather than over a mesh or sub-mesh as for other types of output. To do this, PyLith internally performs the necessary interpolation. There is a README file in the top-level directory that explains how to perform each step in the two problems.

### 7.16.2 Mesh Description

We use linear triangular cells for the meshes in each of the two problems. We construct the mesh in CUBIT following the same techniques used in the 2D subduction zone example. The main driver is in the journal file `mesh_tri3.jou`. It calls the journal file `geometry.jou` to construct the geometry. It then calls the journal file `gradient.jou` to set the variable discretization sizes used in this mesh. Finally, the `createbc.jou` file is called to set up the groups associated with boundary conditions and materials. The mesh used for the strike-slip example is shown in Figure 7.67 on the next page The journal files are documented and describe the various steps outlined below.

1. Create the geometry defining the domain.
2. Create fault surface by splitting domain across the given locations.
3. Define meshing scheme and cell size variation.
4. Define cell size along curves near fault.
5. Increase cell size away from fault at a geometric rate (bias).
6. Generate mesh.
7. Create blocks for materials and nodesets for boundary conditions.
8. Export mesh.

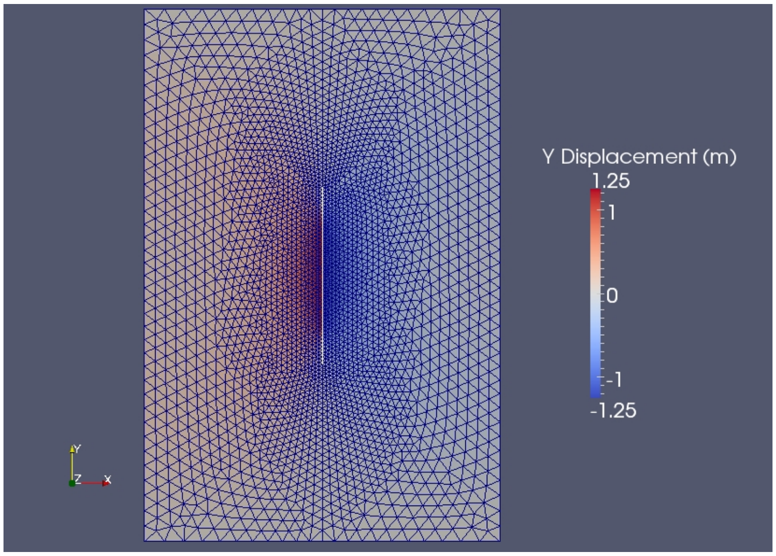



Figure 7.67: Mesh used for both forward and Green's function computations for the strike-slip problem. Computed y-displacements for the forward problem are shown with the color scale.

### 7.16.3 Additional Common Information

As in the examples discussed in previous sections of these examples, we place parameters common to the forward model and Green's function computation in the `pylithapp.cfg` file so that we do not have to duplicate them for the two procedures. The settings contained in `pylithapp.cfg` for this problem consist of:

**pylithapp.journal.info** Settings that control the verbosity of the output written to stdout for the different components.

**pylithapp.mesh_generator** Settings that control mesh importing, such as the importer type, the filename, and the spatial dimension of the mesh.

**pylithapp.problem** Settings that control the problem, such as the total time, time-step size, and spatial dimension.

**pylithapp.problem.materials** Settings that control the material type, specify which material IDs are to be associated with a particular material type, and give the name of the spatial database containing the physical properties for the material. The quadrature information is also given.

**pylithapp.problem.bc** Settings that control the applied boundary conditions.

**pylithapp.problem.interfaces** Settings that control the specification of faults, including quadrature information.

**pylithapp.problem.formulation.output** Settings related to output of the solution over the domain and points (surface observation locations).

**pylithapp.petsc** PETSc settings to use for the problem, such as the preconditioner type.

One aspect that has not been covered previously is the specification of output at discrete points, rather than over a mesh or sub-mesh. We do this using the OutputSolnPoints output type:

Excerpt from `pylithapp.cfg`

```
[pylithapp.problem.formulation]
output = [domain, points]
output.points = pylith.meshio.OutputSolnPoints

[pylithapp.problem.formulation.output.points]
coordsys.space_dim = 2
coordsys.units = km
writer = pylith.meshio.DataWriterHDF5
reader.filename = output_points.txt
```

We provide the number of spatial dimensions and the units of the point coordinates, and then the coordinates are given in a

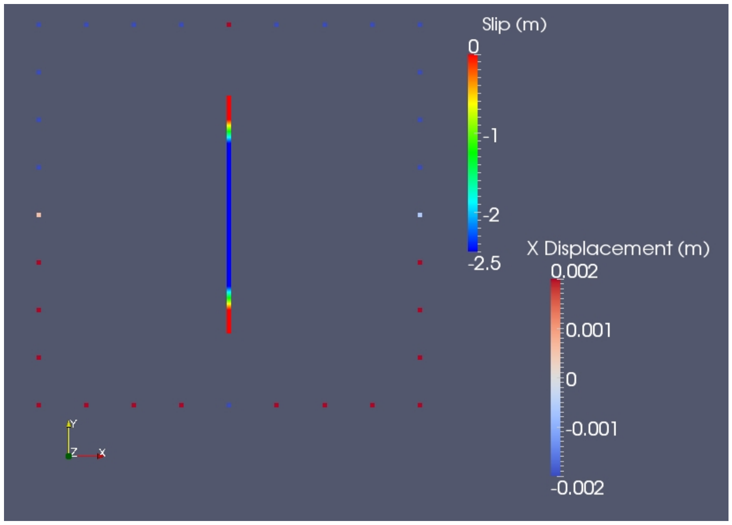



Figure 7.68: Applied fault slip for the strike-slip forward problem as well as computed x-displacements at a set of points.

simple ASCII file (`output_points.txt`). These same points are used for both the forward model computation and the generation of the Green's functions.

### 7.16.4 Step 1: Solution of the Forward Problem

For both the strike-slip problem and the reverse fault problem, we first run a static simulation to generate our synthetic data. Parameter settings that augment those in `pylithapp.cfg` are contained in the file `eqsim.cfg`. These settings are:

**`pylithapp.problem.interfaces`** Give the type of fault interface condition and provide the slip distribution to use. Linear interpolation is used for the slip distribution.

**`pylithapp.problem.formulation.output`** Gives the output filenames for domain output, fault output, point output, and material output. All output uses HDF5 format.

The applied fault slip is given in the file `eqslip.spatialdb`. For both the strike-slip and reverse problems, no fault opening is given, so only the left-lateral component is nonzero. We run the forward models by typing (in the appropriate directory)

```
$ pylith eqsim.cfg
```

Once the problem has run, four HDF5 files will be produced. The file named `eqsim.h5` (and the associated XDMF file) contains the solution for the entire domain. This corresponds to the solution shown in Figure 7.67 on the facing page. The `eqsim-fault.h5` file contains the applied fault slip and the change in fault tractions, while the `eqsim-fault_info.h5` file contains the final slip, the fault normal, and the slip time. The final file (`eqsim-points.h5`) contains the solution computed at the point locations provided in the `output_points.txt` file. These are the results that will be used as synthetic data for our inversion. One the problem has run, the results may be viewed with a visualization package such as ParaView. In Figure 7.68 we show the applied fault slip (from `eqsim-fault.h5`) and the resulting x-displacements (from `eqsim-points.h5`) for our strike-slip forward problem.

### 7.16.5 Step 2: Generation of Green's Functions

The next step is to generate Green's functions that may be used in an inversion. The procedure is similar to that for running the forward problem; however, it is necessary to change the problem type from the default **`timedependent`** to **`greensfns`**. This is accomplished by simply typing

```
pylith --problem=pylith.problems.GreensFns
```

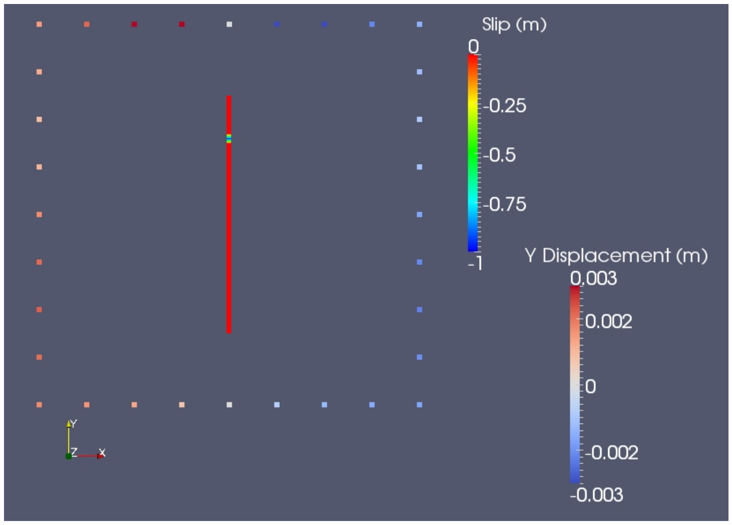



Figure 7.69: Applied fault slip and computed responses (at points) for the seventh Green's function generated for the strike-slip fault example.

This changes the problem type and it also causes PyLith to read the file `greensfns.cfg` by default, in addition to `pylithapp.cfg`. These additional parameter settings provide the information necessary to generate the Green's functions:

Excerpt from `greensfns.cfg`

```
[greensfns]
fault_id = 100

# Set the type of fault interface condition.
[greensfns.interfaces]
fault = pylith.faults.FaultCohesiveImpulses

# Set the parameters for the fault interface condition.
[greensfns.interfaces.fault]
# Generate impulses for lateral slip only, no fault opening.
# Fault DOF 0 corresponds to left-lateral slip.
impulse_dof = [0]

# Set the amplitude of the slip impulses (amplitude is nonzero on only
# a subset of the fault)
db_impulse_amplitude.label = Amplitude of slip impulses
db_impulse_amplitude.iohandler.filename = impulse_amplitude.spatialdb
db_impulse_amplitude.query_type = nearest
```

Note that the top-level identifier is now **greensfns** rather than **pylithapp**. We first set the fault interface condition type to FaultCohesiveImpulses, and then specify the slip component to use. The amplitude of the fault slip and the fault vertices to use are provided in the `impulse_amplitude.spatialdb` file. Fault vertices for which zero slip is specified will not have associated Green's functions generated. The remainder of the `greensfns.cfg` file provides output information, which is exactly analogous to the settings in `eqsim.cfg`.

The generation of Green's functions is somewhat similar to the solution of a time-dependent problem with multiple time steps. In this case, each 'time step' corresponds to the solution computed for a slip impulse at a particular fault vertex. The output files contain the solution for each separate impulse (slip on a single fault vertex). The `greensfns-fault_info.h5` file simply contains the slip amplitude and fault normal. In Figure 7.69 we show the applied impulse (from file `greensfns-fault.h5`) and associated point responses (from file `greensfns-points.h5`) for the seventh generated Green's function in the strike-slip example. In the next section we will show how to read these Green's functions and use them to perform a simple linear inversion.

### 7.16.6 Step 3: Simple Inversion Using PyLith-generated Green's Functions

In the previous two steps we generated a set of synthetic data as well as a set of Green's functions. Both are stored in HDF5 files. To make use of them, we provide a simple Python script that reads the HDF5 results using the h5py Python package. Once we have read the necessary information, we will perform a simple least-squares inversion using the penalty method. We will be solving the equation:

$$G_a m = d_a\,, \tag{7.3}$$

where $m$ are the model parameters (slip), $G_a$ is the augmented set of Green's functions, and $d_a$ is the augmented data vector. The Green's functions are augmented by the addition of a penalty function:

$$G_a = \begin{bmatrix} G \\ \lambda D \end{bmatrix}, \tag{7.4}$$

and the data vector is augmented by the addition of the *a priori* model parameter values:

$$d_a = \begin{bmatrix} d \\ m_{ap} \end{bmatrix}. \tag{7.5}$$

The matrix $D$ is the penalty function, and $\lambda$ is the penalty parameter. The solution is obtained using the generalized inverse (e.g., [Menke, 1984]):

$$G^{-g} = \left(G_a^T G_a\right)^{-1} G_a^T\,, \tag{7.6}$$

and the estimated solution is then:

$$m_{est} = G^{-g} d_a\,. \tag{7.7}$$

The code to read the synthetic data and Green's functions and to perform the inversion is contained in the file `invert_slip.py`, which is contained in the top-level directory. For this simple example, we have simply used a diagonal matrix as the penalty funtion, and the *a priori* parameter values are assumed to be zero. The solution is performed for a range of values of the penalty parameter, which are contained in the file `penalty_params.txt` within each subdirectory. The inversion is performed by running the script in the top-level directory from each subdirectory.

Run the inversion

```
$ ../invert_slip.py --impulses=output/greensfns-fault.h5 \
  --responses=output/greensfns-points.h5 --data=output/eqsim-points.h5 \
  --penalty=penalty_params.txt --output=output/slip_inverted.txt \
```

This will produce an ASCII file (`slip_inverted.txt`), which will contain the estimated solution.

### 7.16.7 Step 4: Visualization of Estimated and True Solutions

Once we have computed the solution, we would then like to visualize the results. We do this using another Python script that requires the matplotlib plotting package (this package is not included in the PyLith binary). We also use the h5py package again to read the applied slip for the forward problem. The Python code to plot the results is contained in the `plot_invresults.py` file contained within each subdirectory.

Plot the results (requires Matplotlib)

```
$ plot_invresults.py --solution=output/eqsim-fault.h5 --predicted=output/slip_inverted.txt
```

The script will produce an interactive matplotlib window that shows the estimated solution compared to the true solution (Figure 7.70 on the next page). As the penalty parameter is increased, the solution is progressively damped. In a real inversion we would also include the effects of data uncertainties, and the penalty parameter would represent a factor controlling the tradeoff between solution simplicity and fitting the noise in the data.

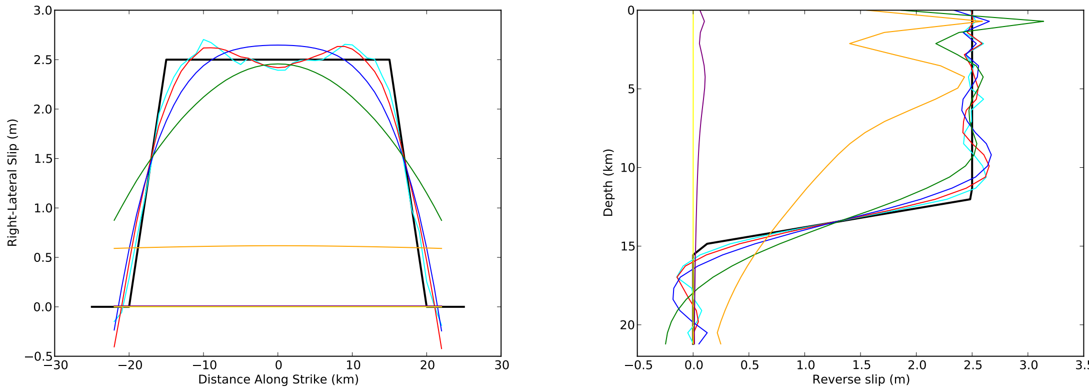



Figure 7.70: Inversion results from running Python plotting script.

## 7.17 Example Using Gravity and Finite Strain in Two Dimensions

PyLith features discussed in this example:

- Gravitational body forces (GravityField)
- Initial stresses
- Finite (or small) strain (ImplicitLgDeform)
- Direct solver in simulations without a fault
- Iterative solver with custom fault preconditioner for a fault
- Generating a spatial database using h5py from state variables output in HDF5 files
- Cubit mesh generation
- Quasi-static solution
- Linear quadrilateral cells
- Plane strain linearly elastic material
- Plane strain Maxwell viscoelastic material
- SimpleDB spatial database
- ZeroDispDB spatial database
- UniformDB spatial database

All of the files necessary to run the examples are contained under the directory `examples/2d/gravity`. The directory also contains a `README` file that describes the simulations and how to run them.

### 7.17.1 Overview

This example illustrates concepts related to using gravitational body forces and finite (or small) strain. We focus on setting up initial conditions consistent with gravitational body forces and using them in a simulation of postseismic deformation with the small strain formulation. We examine the differences between simulations with and without gravitational body forces and the infinitesimal versus small strain formulation.

Steps 1-3 illustrate issues that arise when using gravitational body forces and how to achieve realistic stress states. Steps 4-8 illustrate the differences between infinitesimal and finite strain with and without gravitational body forces for postseismic relaxation following an earthquake with reverse slip.

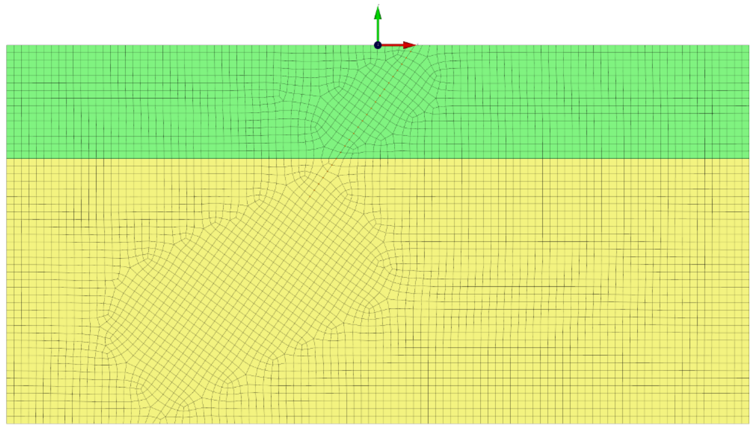


Figure 7.71: Mesh used for 2d gravity simulations with a 30 km thick elastic crust over a 70 km thick linear Maxwell viscoelastic layer.

### 7.17.2 Problem Description

The geometry of the problem is a 200km-wide by 100km-deep domain with a flat ground surface. We use a 30km-thick elastic layer over a linear Maxwell viscoelastic half-space to approximate the crust and mantle. A reverse fault dipping 45 degrees cuts through the elastic layer and extends into the top portion of the viscoelastic layer. Gravitational body forces act in the vertical direction. We apply roller Dirichlet boundary conditions to constrain the displacement normal to the boundary.

We discretize the domain using quadrilateral cells with a nominal cell size of 2.0 km. We construct the mesh in CUBIT following the same techniques used in the 2D subduction zone example, except that this mesh is simpler. The main driver is in the journal file `mesh.jou`. It calls the journal file `geometry.jou` to construct the geometry. The mesh shown in Figure 7.71 The journal files are documented and describe the various steps outlined below.

1. Create the geometry defining the domain.
2. Set the meshing scheme and cell size.
3. Generate the mesh.
4. Create blocks for materials and nodesets for boundary conditions.
5. Export the mesh.

### 7.17.3 Additional Common Information

As in the examples discussed in previous sections of these examples, we place parameters common to all of the simulations in the `pylithapp.cfg` file so that we do not have to duplicate them in each simulation parameter file. In some cases we do override the values of parameters in simulation specific parameter files. The settings contained in `pylithapp.cfg` for this problem consist of:

**pylithapp.journal.info** Settings that control the verbosity of the output written to stdout for the different components.

**pylithapp.mesh_generator** Settings that control mesh importing, such as the importer type, the filename, and the spatial dimension of the mesh.

**pylithapp.problem** Settings that control the problem, such as the total time, time-step size, and spatial dimension. Note that we turn off the elastic prestep here, since it is only used in the first simulation. We also turn on gravity for the problem. The `total_time` of `2000.0*year` is used for most of the simulations.

**pylithapp.problem.materials** Settings that control the material type, specify which material IDs are to be associated with a particular material type, and give the name of the spatial database containing the physical properties for the material. The quadrature information is also given.

**pylithapp.problem.bc** We apply Dirichlet roller boundary conditions (pin displacement perpendicular to the boundary) on the lateral sides and bottom of the domain.

**pylithapp.problem.formulation.output** Settings related to output of the solution over the domain and subdomain. We specify both displacements and velocities for the output.

**pylithapp.petsc** PETSc settings to use for the problem, such as the preconditioner type.

Since we do not desire an initial elastic solution prior to beginning our time stepping for the simulations, we turn off the elastic prestep:

Excerpt from `pylithapp.cfg`

```
[pylithapp.timedependent]
elastic_prestep = False
```

For two-dimensional problems involving gravity, we also need to change the default **gravity_dir**:

Excerpt from `pylithapp.cfg`

```
[pylithapp.timedependent]
gravity_field = spatialdata.spatialdb.GravityField
gravity_field.gravity_dir = [0.0, -1.0, 0.0]
```

### 7.17.4 Step 1: Gravitational Body Forces and Infinitesimal Strain

This simulation applies gravitational body forces to a domain without any initial conditions, so the gravitational body forces cause the domain to compress in the vertical direction. The shear stresses in the mantle relax, so that the solution in the mantle trends towards $\sigma_{xx} = \sigma_{yy}$. The crust is elastic and compressible, so $\sigma_{xx} \neq \sigma_{yy}$. In the earth's crust we generally observe $\sigma_{xx} \approx \sigma_{yy}$, so this simulation does not yield a stress state consistent with that observed in nature. The file `gravity_infstrain.cfg` contains the simulation specific parameter settings that augment those in `pylithapp.cfg`. In addition to the filenames for the HDF5 ouput we also set the filename for the progress monitor. You can look at this file during the simulation to monitor the progress and get an estimate of when the simulation will finish.

Run Step 1 simulation

```
$ pylith gravity_infstrain.cfg
```

The simulation produces HDF5 (and corresponding XDMF) files with the output of the displacements on the ground surface and the entire domain, and the state variables for the crust and mantle. Note that the output files contain both cauchy_stress and stress fields. For problems using the infinitesimal strain formulation, these are identical. For the small strain formulation, the stress field corresponds to the second Piola-Kirchoff stress tensor, which does not have the physical meaning associated with the Cauchy stress. Loading the axial stress fields for the crust and mantle into ParaView via the XDMF files (`output/gravity_infstrain-crust.xmf` and `output/gravity_infstrain-mantle.xmf`) illustrates how the axial stresses are not equal. We would like gravitational body forces to yield nearly axial stresses consistent with the overburden pressure observed in nature.

### 7.17.5 Step 2: Gravitational Body Forces, Infinitesimal Strain, and Initial Stresses

This simulation uses depth-dependent initial stresses that satisfy the governing equations. As a result, there is zero deformation. In practice, there would be no need to run such a simulation, because the initial stresses give us the stress state produced in the simulation. In Steps 3-7, we use these initial stresses as initial conditions for postseismic deformation simulations. Because we reuse the initial stress parameter settings in multiple simulations, we place them in their own parameter file, `gravity_initstress.cfg`. As in Step 1, the simulation specific parameter file contains the filenames for the output.

Run Step 2 simulation

```
$ pylith gravity_initstress.cfg gravity_isostatic.cfg
```

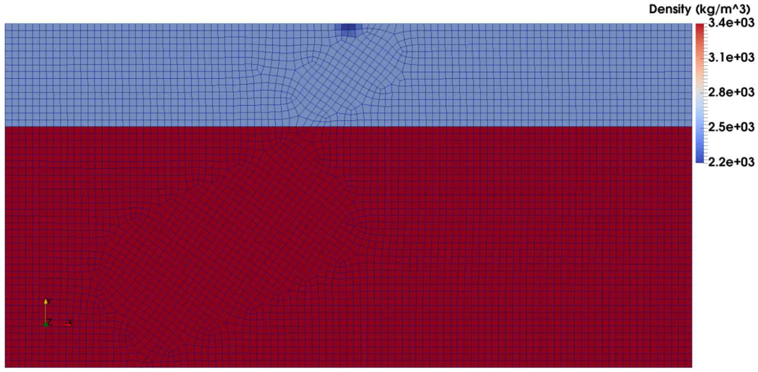



Figure 7.72: Spatial variation in density in the finite element mesh. The mantle has a uniform density of 3400 kg/m$^3$ and the crust has a uniform density of 2500 kg/m$^3$ except near the origin where we impose a low density semi-circular region.

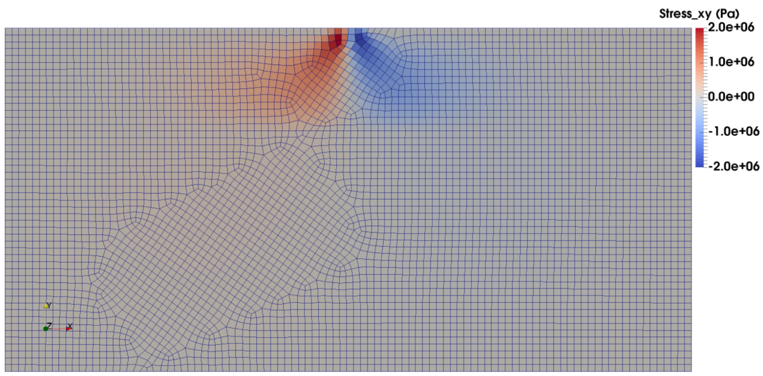



Figure 7.73: Shear stress in the crust (linearly elastic) and mantle (linear Maxwell viscoelastic) associated gravitational body forces and a low density region forces.

### 7.17.6 Step 3: Infinitesimal Strain Simulation with Initial Stresses and a Local Density Variation

This simulation adds a local density variation in the elastic layer to the problem considered in Step 2. Near the origin, the density is reduced in a semi-circular region with a radius of 5.0 km, roughly approximating a sedimentary basin. In this example, we focus on the workflow and use a coarse mesh so we are not concerned with the fact that our mesh with a discretization size of 2.0 km does a poor job of resolving this density variation; in a real research problem we would use a finer mesh in the low density region. Figure 7.72shows the spatial variation in density, including the contrast in density between the mantle and crust and the circular low density region around the origin.

We use the same initial stress state as for the previous two examples. The initial stress state is a close approximation to the equilibrium state, so there is little deformation. The mantle relaxes corresponding to the viscous strains and shear stresses approaching zero; shear stress associated with the lateral density variation becomes confined to the crust. In the region with the lower density, the initial stresses do not satisfy the governing equation and the solution slowly evolves towards a steady state. This slow asymptotic evolution presents some difficulties with using the output of this simulation (which has not reached the equilibrium state) as a starting point in other simulations, as we will see in Step 8. Nevertheless, this simulation serves as an example of how to use initial stresses from vertically layered material properties in computing an equilibrium or steady state stress state associated with gravitational body forces and lateral density variations or topography.

Run Step 3 simulation

```
$ pylith gravity_initstress.cfg gravity_vardensity.cfg
```

Figure 7.73 shows the shear stress field at the end of the simulation.

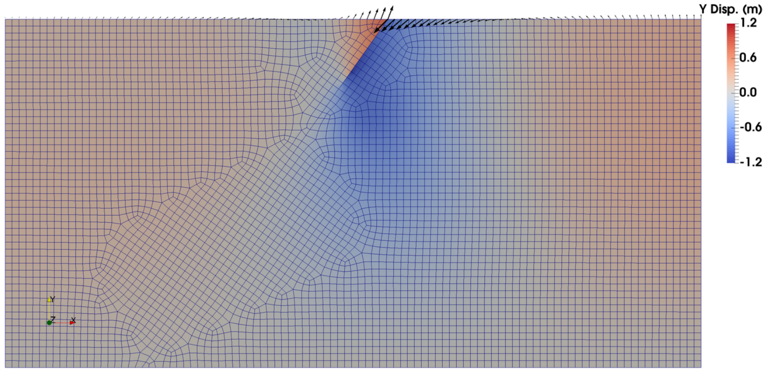



Figure 7.74: Vertical displacement at the end of the postseismic deformation simulation (t=4000 years).

### 7.17.7 Step 4: Postseismic Relaxation with Infinitesimal Strain

We impose slip on the reverse fault within the elastic layer and compute the postseismic deformation associated with relaxation in the mantle. We do not include gravitational body forces. The earthquake slip is 2.0 m above a depth of 15 km and tapers linearly to zero at a depth of 20 km. We impose the earthquake at time 0.0 years, but use a start time of -100 years so that any pre-earthquake deformation trends are clear. We use one parameter file (`nogravity.cfg`) to turn off gravity (by setting the gravitional acceleration to zero) and another parameter file (`postseismic.cfg`) for the earthquake related parameters. Note that we change the preconditioner to the algebraic multigrid preconditioner for the elastic block and the custom fault preconditioner for the Lagrange multipliers.

Run Step 4 simulation

```
$ pylith postseismic.cfg nogravity.cfg postseismic_infstrain_nograv.cfg
```

Figure 7.74 shows the vertical displacement field at the end of the simulation.

### 7.17.8 Step 5: Postseismic Relaxation with Finite Strain

This simulation is the same as Step 4, but we use the finite strain formulation.:

Excerpt from `postseismic_finstrain.cfg`

```
[pylithapp.timedependent]
formulation = pylith.problems.ImplicitLgDeform
```

When we use the finite strain formulation, the solver is automatically switched to the nonlinear solver.

Run Step 5 simulation

```
$ pylith postseismic.cfg nogravity.cfg postseismic_finstrain_nograv.cfg
```

The results are nearly identical to those with infinitesimal strain.

### 7.17.9 Step 6: Postseismic Relaxation with Infinitesimal Strain and Gravitational Body Forces

This simulation is the same as Step 4, but we include gravitational body forces. We use initial stresses that satisfy the governing equations, so our initial conditions are axial stresses equal to the overburden pressure.

Run Step 6 simulation

```
$ pylith postseismic.cfg postseismic_infstrain.cfg
```

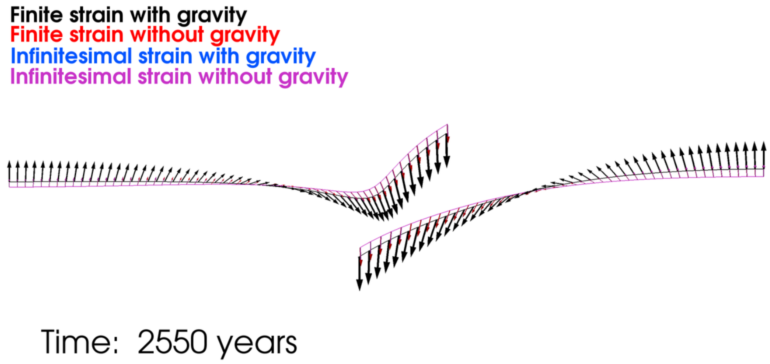



Figure 7.75: Displacement field on the ground surface after 2550 years of postseismic deformation in Step 4 (Infinitesimal strain without gravity), Step 5 (Finite strain without gravity), Step 6 (Infinitesimal strain with gravity), and 7 (Finite strain with gravity). The displacement fields for Steps 4-6 are essentially identical.

With the infinitesimal strain formulation and linearly material behavior, the initial stress state of equal axial stresses does not alter the response. We obtain a displacement field and shear stresses identical to that in Step 4. The axial stresses are the superposition of the initial stresses and those from the postseismic deformation.

### 7.17.10 Step 7: Postseismic Relaxation with Finite Strain and Gravitational Body Forces

This simulation is the same as Step 5, but we include gravitational body forces; this is also the same as Step 6, but with finite strain.

Run Step 7 simulation

```
$ pylith postseismic.cfg postseismic_finstrain.cfg
```

The finite strain formulation accounts for the redistribution of gravitational body forces as the domain deforms during the postseismic response. As a result, the displacement field differs from that in Steps 4-6. To see this difference, we have created a ParaView state file to view the ground surface deformation from the output of Steps 4-7. After running all four simulations, open ParaView and load the state file `postseismic.pvsm`. If you start ParaView from the `examples/2d/gravity` directory (`PATH_TO_PARAVIEW/bin/paraview`, `File→Load State→postseismic.pvsm`), you should not need to update the locations of the filenames. If you start ParaView from a dock or other directory, you will need to set the relative or absolute paths to the output files. Figure 7.75 shows the ground deformation 2550 years after the earthquake using the state file.

### 7.17.11 Step 8: Postseismic Relaxation with Finite Strain, Gravitational Body Forces, and Variable Density

We use the output of Step 3 to create realistic initial stresses for this simulation of postseismic deformation with variable density. In Step 3 we average the stresses over the quadrature points within a cell using CellFilterAvg. For initial stresses consistent with the state of the simulation at the end of Step 3, we want the stresses at each of the quadrature points. Note that Step 3 uses the infintesimal strain formulation and for Step 8 we will use a finite strain formulation; any inconsistencies in using the output from a simulation with one strain formulation as the input in a simulation for another strain formulation are very small given that we start Step 8 from an undeformed state so that the Cauchy stresses are very close to the second Pioloa-Kirchoff stresses. Our first step is to modify the `pylithapp.cfg` file by commenting out the lines with the CellFilterAvg settings:

Modify `pylithapp.cfg` when rerunning Step 3

```
#cell_filter = pylith.meshio.CellFilterAvg
```

for both the crust and mantle.

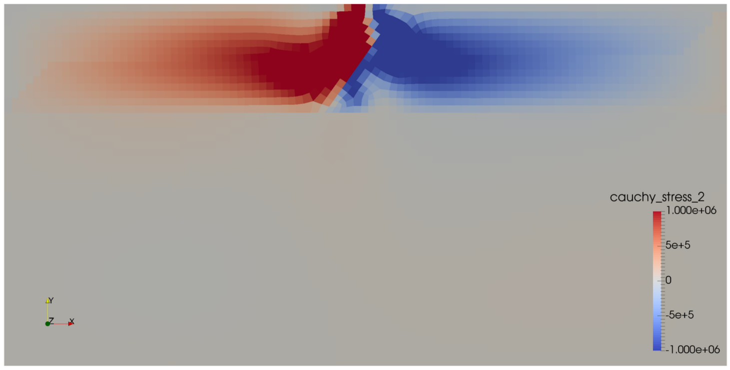



Figure 7.76: Cauchy shear stress at the end of the simulation of postseismic deformation with variable density in the crust. We saturate the color scale at ±1 MPa to show the evidence of viscoelastic relaxation (near zero shear stress) in the mantle.

Rerun Step 3 after modifying `pylithapp.cfg`

```
$ pylith gravity_initstress.cfg gravity_vardensity.cfg
```

This will change how the values appear in ParaView output. Because the output data fields contain the values at multiple points within a cell, PyLith does not label them as tensor components; instead, it simply numbers the values 0..N. For the stress tensor components, values 0, 1, and 2 are the $\sigma_{xx}$, $\sigma_{yy}$, and $\sigma_{xy}$ values at the first quadrature point; values 3, 4, and 5 correspond to the values at the second quadrature point, etc. We use the Python script `generate_statedb.py` to generate the spatial databases with the initial stresses from the output of Step 3.

Generate intial stresses using output from Step 3

```
$ ./generate_statedb.py
```

After generating the initial state variables, we uncomment the `cell_filter` lines in `pylithapp.cfg` to allow easier visualization of Step 8 results.

Run Step 8 simulation

```
$ pylith postseismic.cfg gravity_initstress.cfg postseismic_vardensity.cfg
```

In the 100 years before the earthquake, it is clear that there is some ongoing deformation associated with the relaxation of the mantle. Immediately following the earthquake the postseismic deformation signal is stronger at most locations, but as it decays the ongoing deformation associated with the gravitational body forces and variable density become evident again. This ongoing deformation is most obvious in the displacement and velocity fields. The postseismic deformation is much more dominant for the stress field. This contamination by the initial conditions can be avoided with initial stress conditions at equilibrium as we did in Step 7. However, this is much more difficult to obtain for complex lateral variations in density or topography. Figure 7.76 shows the ground deformation at time 2000 years into the simulation using the state file.

### 7.17.12 Exercises

The `README` file in `examples/2d/gravity` includes some suggetions of additional simulations to run to further explore some of the issues discussed in this suite of examples.

## 7.18 Examples for a 3D Subduction Zone

### 7.18.1 Overview

This suite of examples demonstrates use of a wide variety of features and the general workflow often used in research simulations. We base the model on the Cascadia subduction zone (Figure 7.77). These examples will focus on modeling the deformation associated with the the subducting slab, including interseismic deformation with aseismic slip (creep) and viscoelastic relaxation, coseismic slip on the slab interface and a splay fault, and slow slip events on the subduction interface. We want to account for the 3-D material properties associated with different elastic properties for the subducting slab, mantle, continental crust, and an accretionary wedge. To keep the computation time in these examples short, we limit our model to an 800 km × 800 km × 400 km domain and we will use a relatively coarse discretization. For simplicity and to reduce complexity in constructing the mesh, we use a flat top surface (elevation of 0 with respect to mean sea level).

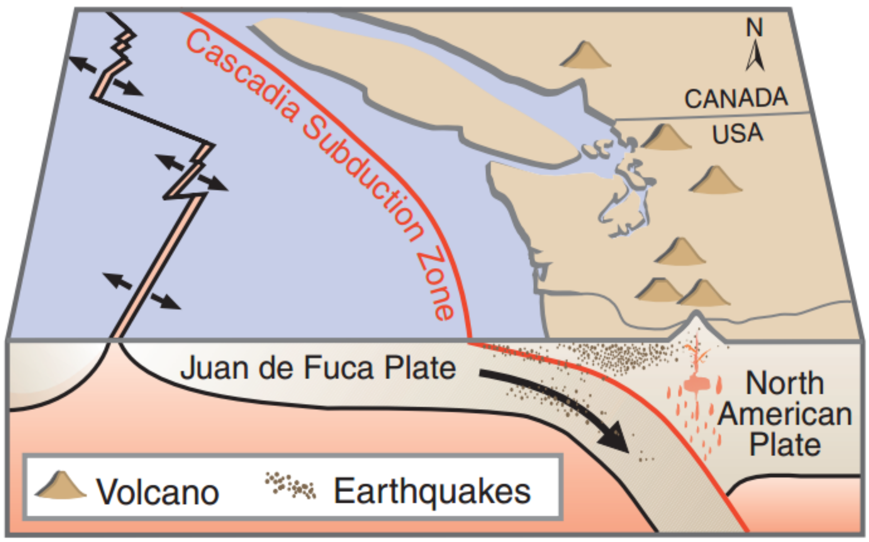



Figure 7.77: Cartoon of the Cascadia Subduction Zone showing the subduction of the Juan de Fuca Plate under the North American Plate. Source: U.S. Geological Survey Fact Sheet 060-00

Figure 7.78 shows our conceptual model with a slab, mantle, continental crust, and accretionary wedge. We cut off the slab at a depth of 100 km. We use a transverse geographic projection coordinate system with Portland, Oregon, as the origin in order to georeference our model. In order to model the motion of the slab, we include a fault for the subduction interface (the interface between the top of the slab and the mantle, crust, and wedge), as well as a fault between the bottom of the slab and the mantle.

The files associated with this suite of examples are contained in the directory `examples/3d/subduction`. This directory contains several subdirectories:

**`mesh`** Files used to construct the finite-element mesh using CUBIT/Trelis.

**`spatialdb`** Files associated with the spatial and temporal databases.

**`viz`** ParaView Python scripts and other files for visualizing results.

**`output`** Directory containing simulation output. It is created automatically when running the simulations.

### 7.18.2 Features Illustrated

Table 7.3 lists the features discussed in each of these 3-D subduction zone examples. With the intent of illustrating features used in research simulations, we use HDF5 output and, we make extensive use the most efficient implementations of spatial

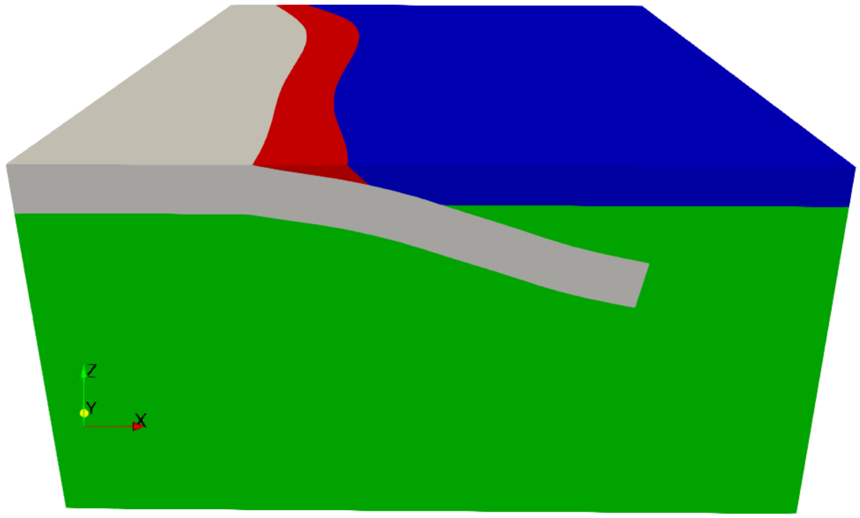



Figure 7.78: Conceptual model based on the Cascadia Subduction Zone. The model includes the subduction slab (white), the mantle (green), continental crust (blue), and an accretionary wedge (red).

databases (UniformDB and SimpleGridDB). We also use ParaView Python scripts for visualizing the output. These scripts can be run within the ParaView GUI or outside the ParaView GUI, although the interaction is limited to rotating, translating, and zooming when run outside the ParaView GUI.

### 7.18.3 Generating the Finite-Element Mesh

We use CUBIT/Trelis to generate the finite-element mesh. Due to its size, we do not include the finite-element mesh in the PyLith source or binary distributions. If you do not have CUBIT/Trelis, you can download the mesh from `https://wiki.geodynamics.org/software:pylith:examples:files` and skip generating the mesh.

We use contours of the Cascadia Subduction Zone from Slab v1.0 [Hayes et al., 2012] for the geometry of the subduction interface. In order to make use of these contours from within CUBIT/Trelis, we use a Python script (`generate_surfjou.py`) to read the contours file and create a CUBIT/Trelis journal file (`generate_surfs.jou`) that adds additional contours west of the trench and then constructs the top and bottom surfaces of the slab. The Python script also constructs a splay fault by copying a contour to a depth below the slab and above the ground surface.

> ★ **Tip**
>
> We define the coordinate systems we use in the simulations in the the Python script `coordsys.py` to make it easier to convert to/from various georeference coordinate systems in the pre- and post-processing. PyLith will automatically convert among compatible coordinate systems during the simulation.

Generate `generate_surfs.jou`

```
# Make sure you are in the 'mesh' directory and then run the Python
# script to generate the journal file 'generate_surfs.jou'.
$ ./generate_surfjou.py
```

The next step is to use CUBIT/Trelis to run the `generate_surfs.jou` journal file to generate the spline surfaces for the slab and splay fault and save them as ACIS surfaces.

Table 7.3: PyLith features covered in the suite of 3-D subduction zone examples.

| Example | General | | | | | | | | Solver | | | Spatial Database | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Dimension | Coordinate system | Mesh generator | Cells | Refinement | Reordering | Problem type | Time dependence | Solver | Preconditioner | Time stepping | Uniform | Simple | Simple grid | Composite | Time history |
| 3d/subduction/step01 | 3 | Proj | CUBIT | Tet | | ✔ | TD | S | L | ILU | | x2 | x4 | | | |
| 3d/subduction/step02 | 3 | Proj | CUBIT | Tet | | ✔ | TD | QS | L | ML+Cust | BE | x2 | x3 | x2 | x2 | |
| 3d/subduction/step03 | 3 | Proj | CUBIT | Tet | | ✔ | TD | QS | L | ML+Cust | BE | x4 | x3 | x2 | x2 | |
| 3d/subduction/step04 | 3 | Proj | CUBIT | Tet | | ✔ | TD | QS | L | ML+Cust | BE | x7 | x3 | x5 | x2 | |
| 3d/subduction/step05 | 3 | Proj | CUBIT | Tet | | ✔ | TD | QS | NL | ML+Cust | BE | x7 | x3 | x5 | x2 | |
| 3d/subduction/step06 | 3 | Proj | CUBIT | Tet | | ✔ | TD | QS | L | ML+Cust | BE | x1 | x4 | x1 | | x1 |
| 3d/subduction/step07a,b | 3 | Proj | CUBIT | Tet | | ✔ | GF | S | L | ML+Cust | BE | x1 | x4 | | | |
| 3d/subduction/step08a | 3 | Proj | CUBIT | Tet | | ✔ | TD | S | L | ML+Cust | | | x4 | | | |
| 3d/subduction/step08b | 3 | Proj | CUBIT | Tet | | ✔ | TD | S | L | ML+Cust | | | x4 | | | |
| 3d/subduction/step08c | 3 | Proj | CUBIT | Tet | | ✔ | TD | QS | NL | ML+Cust | BE | | x4 | | | |

**Coordinate system** – Cart: Cartesian, Proj: geographic projection. **Mesh generator** – ASCII: ASCII, CUBIT: CUBIT/Trelis, LaGriT: LaGriT. **Problem type** – TD: time dependent, GF: Green's functions. **Time dependence** – S: static, QS: quasi-static, D: dynamic. **Solver** – L: linear, NL: nonlinear. **Preconditioner** – ILU: ILU, ASM: Additive Schwarz, SCHUR: Schur complement, Cust: custom, ML: ML algebraic multigrid, GAMG: geometric algebraic multigrid. **Time stepping** – BE: Backward Euler, FE: Forward Euler.

| Example | Boundary Condition | | | | Fault | | | | | | | | Bulk Rheology | | | | | | | | | Output | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Dirichlet | Neumann | Absorbing | Point force | Prescribed slip | Slip time function | Constitutive model | Static friction | Slip-weakening friction | Time-weakening friction | Rate-state friction w/ageing | Traction perturbation | Linear elastic | Linear Maxwell viscoelastic | Generalized Maxwell viscoelastic | Powerlaw viscoelastic | Drucker-Prager elastoplastic | Stress/strain formulation | Inertia | Reference state | Gravity | Format | Domain output | Surface output | Point output | State variable output | ParaView | Matplotlib |
| 3d/subduction/step01 | x5 | | | | | | | | | | | | x4 | | | | | Inf | | | | H5 | x1 | x1 | | x4 | ✔ | |
| 3d/subduction/step02 | x5 | | | | x1 | Step | | | | | | | x2 | x2 | | | | Inf | | | | H5 | x1 | x1 | | x4 | ✔ | |
| 3d/subduction/step03 | x5 | | | | x2 | Rate | | | | | | | x2 | x2 | | | | Inf | | | | H5 | x1 | x1 | | x4 | ✔ | |
| 3d/subduction/step04 | x5 | | | | x3 | Step | | | | | | | x2 | x2 | | | | Inf | | | | H5 | x1 | x1 | | x4 | ✔ | |
| 3d/subduction/step05 | x5 | | | | x1 | Rate | x1 | | ✔ | | | ✔ | x2 | x2 | | | | Inf | | | | H5 | x1 | x1 | | x4 | ✔ | |
| 3d/subduction/step06 | x5 | | | | x1 | User | | | | | | | x4 | | | | | Inf | | | | H5 | x1 | x1 | x1 | x4 | ✔ | |
| 3d/subduction/step07a,b | x5 | | | | x1 | Step | | | | | | | x4 | | | | | Inf | | | | H5 | | x1 | x1 | x4 | ✔ | |
| 3d/subduction/step08a | x5 | | | | | | | | | | | | x4 | | | | | Inf | | ✔ | ✔ | H5 | x1 | x1 | | x4 | ✔ | |
| 3d/subduction/step08b | x5 | | | | | | | | | | | | x4 | | | | | Inf | | ✔ | ✔ | H5 | x1 | x1 | | x4 | ✔ | |
| 3d/subduction/step08c | x5 | | | | | | | | | | | | x2 | x2 | | | | Fin | | ✔ | ✔ | H5 | x1 | x1 | | x4 | ✔ | |

**Stress/strain formulation** – Inf: infinitesimal, Fin: small, finite strain. **Format** – VTK: VTK, H5: HDF5, H5Ext: HDF5 w/external datasets.

**Important**

The CUBIT/Trelis journal files name objects and then later reference them by name. When objects are cut, a suffix of @LETTER is appended to the original name (for example, domain becomes domain and domain@A). However, which one retains the original name and which ones gets the suffix is ambiguous. In general, the names are consistent across versions of CUBIT/Trelis with the same version of the underlying ACIS library. **As a result, you may need to update the ids in the references to previously named objects that have been split (for example domain@A may need to be changed to domain@B, etc) in order to account for differences in how your version of CUBIT/Trelis has named split objects.**

Currently we discretize the domain using a uniform, coarse resolution of 25 km. This allows the simulations to run relatively quickly and fit on a laptop. In a real research problem, we would tailor the resolution to match the length scales we want to capture and use a finer resolution. We provide journal files for both a mesh with tetrahedral cells (`mesh_tet.jou`) and a mesh with hexahedral cells (`mesh_tet.jou`). In the following examples, we will focus exclusively on the mesh with tetrahedral cells because the mesh with hexahedral cells contains cells that are significantly distorted; this illustrates how it is often difficult to generate high quality meshes with hexahedral cells for domains with complex 3-D geometry.

After you generate the ACIS surface files, run the `mesh_tet.jou` journal file to construct the geometry, and generate the mesh. In the end you will have an Exodus-II file `mesh_tet.exo`, which is a NetCDF file, in the `mesh` directory. You can load this file into ParaView.

**Tip**

We recommend carefully examining the `geometry.jou` journal file to understand how we assemble the 3-D slab and cut the rectangular domain into pieces.

#### 7.18.3.1 Visualizing the Mesh

The Exodus-II file `mesh_tet.exo` can be viewed with ParaView. We provide the Python script `viz/pot_mesh.py` to visualize the nodesets and the mesh quality using the condition number metric. As in our other Python scripts for ParaView (see Section 7.2 on page 112 for a discussion of how to use Python ParaView scripts), you can override the default parameters by setting appropriate values in the Python shell (if running within the ParaView GUI) or from the command line (if running the script directly outside the GUI). When viewing the nodesets, the animation controls allow stepping through the nodesets. When viewing the mesh quality, only the cells with the given quality metric above some threshold (poorer quality) are shown. The default quality metric is condition number and the default threshold is 2.0.

To visualize the mesh, start ParaView. Within the ParaView GUI Python shell (Tools→Python Shell), we override the `EXODUS_FILE` and `SHOW_QUALITY` parameters.

ParaView Python shell

```
# Import the os module so we can get access to the HOME environment variable.
>>> import os
>>> HOME = os.environ["HOME"]
# You may need to adjust the next line, depending on where you installed PyLith.
>>> EXODUS_FILE = os.path.join(HOME,"pylith","examples","3d","subduction","mesh","mesh_tet.exo")
# Turn off display of the mesh quality (show only the nodesets).
>>> SHOW_QUALITY = False
```

We then click on the Run Script button and navigate to the `examples/3d/subduction/viz` directory and select `plot_mesh.py`.

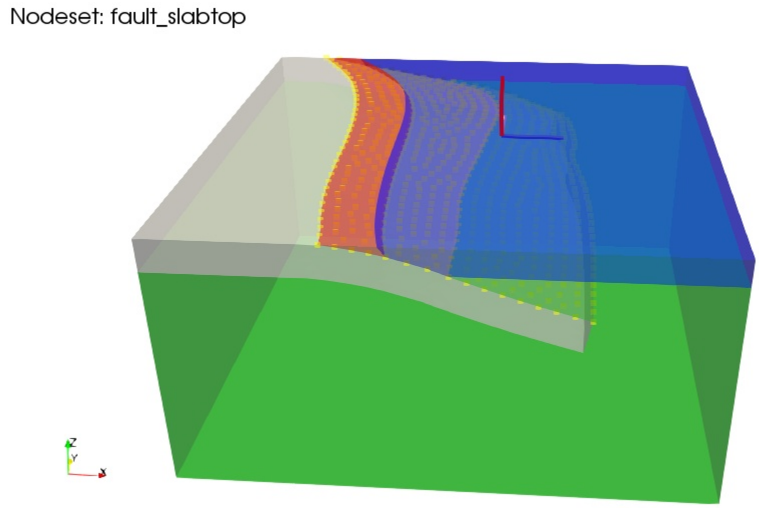



Figure 7.79: Visualization of the fault_slabtop nodeset (yellow dots) for the Exodus-II file `mesh/mesh_tet.exo` using the `viz/plot_mesh.py` ParaView Python script. One can step through the different nodesets using the animation controls. This script can also be use to show the mesh quality.

### 7.18.4 Organization of Simulation Parameters

PyLith automatically reads in `pylithapp.cfg` from the current directory, if it exists. As a result, we generally put all parameters common to a set of examples in this file to avoid duplicating parameters across multiple files. Because we often use a single mesh for multiple simulations in a directory, we place all parameters related to our mesh and identifying the materials in our mesh in `pylithapp.cfg`. We assign the bulk constitutive model and its parameters to each material in other files, because we vary those across the simulations. In general, we place roller boundary conditions (Dirichlet boundary conditions constraining the degrees of freedom perpendicular to the boundary) on the lateral and bottom boundaries, so we include those in `pylithapp.cfg`. In some simulations we will overwrite the values for parameters will values specific to a given example. This file is also a convenient place to put basic solver parameters and to turn on Pyre journals for displaying informational messages during a run.journalling debugging flags.

Hence the settings contained in `pylithapp.cfg` include:

**`pylithapp.journal.info`** Settings that control the verbosity of the output written to stdout for the different components.

**`pylithapp.mesh_generator`** Parameters for the type of mesh importer (generator), reordering of the mesh, and the mesh coordinate system.

**`pylithapp.problem.materials`** Basic parameters for each of the four materials, including the label, block id in the mesh file, discretization, and output writer.

**`pylithapp.problem.bc`** Parameters for Dirichlet boundary conditions on the lateral and bottom boundaries of the domain.

**`pylithapp.problem.formulation.output`** Settings related output of the solution over the domain and subdomain (ground surface).

**`pylithapp.petsc`** PETSc solver and logging settings.

#### 7.18.4.1 Coordinate system

We generated the mesh in a Cartesian coordinate system corresponding to a transverse Mercator projection. We specify this geographic projection coordinate system in the `pylithapp.cfg` file, so that we can use other convenient georeferenced coordinate systems in the spatial databases. PyLith will automatically transform points between compatible coordinate systems. Our spatialdata library uses Proj4 for geographic projections, so we specify the projection using Proj4 syntax in the **`proj_options`** property:

Excerpt from pylithapp.cfg

```
[pylithapp.mesh_generator.reader]
coordsys = spatialdata.geocoords.CSGeoProj
coordsys.space_dim = 3
coordsys.datum_horiz = WGS84
coordsys.datum_vert = mean sea level
coordsys.projector.projection = tmerc
coordsys.projector.proj_options = +lon_0=-122.6765 +lat_0=45.5231 +k=0.9996
```

#### 7.18.4.2 Materials

The finite-element mesh marks cells for each material and the type of cell determines the type of basis functions we use in the discretization. This means we can specify this information in the pylithapp.cfg file and avoid duplicating it in each simulation parameter file. To set up the materials, we first create an array of materials that defines the name for each material component. For example, we create the array of four materials and then set the parameters for the slab:

Excerpt from pylithapp.cfg

```
[pylithapp.problem]
materials = [slab, wedge, crust, mantle]

[pylithapp.problem.materials.slab]
label = Subducting slab ; Label for informative error messages
id = 1 ; Block id in ExodusII file from CUBIT/Trelis
quadrature.cell = pylith.feassemble.FIATSimplex ; Tetrahedral cells
quadrature.cell.dimension = 3

# Average cell output over quadrature points, yielding one point per cell
output.cell_filter = pylith.meshio.CellFilterAvg
output.writer = pylith.meshio.DataWriterHDF5 ; Output using HDF5
```

In this set of examples, we will consider cases in which all materials are linear, isotropic elastic and cases where the crust and wedge are linear, isotropic elastic but the slab and mantle are linear Maxwell viscoelastic. As a result, we put the parameters for these two cases in separate cfg files with mat_elastic.cfg for the case with purely elastic models and mat_viscoelastic.cfg for the case with a mix of elastic and viscoelastic models. Each of these files specifies the bulk constitutive model and spatial database to use for the properties for each material. The values for the material properties are loosely based on a 3-D seismic velocity model for the Pacific Northwest [Stephenson, 2007].

#### 7.18.4.3 Boundary Conditions

For the Dirichlet boundary conditions, we specify the degree of freedom constrained, the name of the nodeset in the ExodusII file from CUBIT/Trelis that defines the boundary, and a label for the spatial database (required for informative error messages). These settings constrain the y-displacement on the north (+y) boundary:

Excerpt from pylithapp.cfg

```
[pylithapp.problem.bc.y_pos]
bc_dof = [1] ; Degree of freedoms are: x=0, y=1, and z=2
label = boundary_ypos ; nodeset in ExodusII file form CUBIT/Trelis
db_initial.label = Dirichlet BC on +y ; label for informative error messages
```

#### 7.18.4.4 Solver Parameters

We group solver parameters into a few different files to handle different cases. The pylithapp.cfg contains tolerance values for the linear and nonlinear solvers and turns on some simple diagnostic information. The file also directs PyLith to use a direct solver, which is suitable for debugging and test problems that do not include a fault; a direct solver is not well-suited for production runs because it does not scale well and uses a lot of memory.

Excerpt from `pylithapp.cfg`

```
[pylithapp.petsc]
malloc_dump = ; Dump information about PETSc memory not deallocated.

# Use LU preconditioner (helpful for learning and debugging, not production simulations)
pc_type = lu

# Convergence parameters.
ksp_rtol = 1.0e-10 ; Converge if residual norm decreases by this amount
ksp_atol = 1.0e-11 ; Converge if residual norm drops below this value
ksp_max_it = 500 ; Maximum number of iterations in linear solve
ksp_gmres_restart = 50 ; Restart orthogonalization in GMRES after this number of iterations

# Linear solver monitoring options.
ksp_monitor = true ; Show residual norm at each iteration
#ksp_view = true ; Show solver parameters (commented out)
ksp_converged_reason = true ; Show reason linear solve converged
ksp_error_if_not_converged = true ; Generate an error if linear solve fails to converge

# Nonlinear solver monitoring options.
snes_rtol = 1.0e-10 ; Converge if nonlinear residual norm decreases by this amount
snes_atol = 1.0e-9 ; Converge if nonlinear residual norm drops below this value
snes_max_it = 100 ; Maximum number of iterations in nonlinear solve
snes_monitor = true ; Show nonlinear residual norm at each iteration
snes_linesearch_monitor = true ; Show nonlinear solver line search information
#snes_view = true ; Show nonlinear solver parameters (commented out)
snes_converged_reason = true ; Show reason nonlinear solve converged
snes_error_if_not_converged = true ; Generate an error if nonlinear solve fails to converge

# PETSc summary -- useful for performance information.
log_view = true
```

The `solver_algebraicmultigrid.cfg` provides more optimal settings for simulations without a fault by using an algebraic multigrid preconditioner. Similarly, for simulations with a fault `solver_fieldsplit.cfg` provides settings for applying the algebraic multigrid preconditioner to the elasticity portion of the system Jacobian matrix and our custom fault preconditioner to the Lagrange multiplier portion.

### 7.18.5 Step 1: Axial Compression

We start with a very simple example of axial compression in the east-west direction with purely elastic material properties, and no faults (Figure 7.80). We impose axial compression using Dirichlet boundary conditions on the east (+x) and west (-x) boundaries and confine the domain in the north-south direction via zero displacement Dirichlet boundary conditions on the north (+y) and south (-y) boundaries. We constrain the vertical displacement by imposing zero displacement boundary conditions on the bottom (-z) boundary.

The `pylithapp.cfg` file creates an array of five boundary conditions, which impose zero displacements by default. We overwrite this behavior in the `step01.cfg` file for the -x and +x boundaries using spatial databases with a single uniform displacement value to create the axial compression:

Excerpt from `step01.cfg`

```
# -x face
[pylithapp.problem.bc.x_neg]
db_initial = spatialdata.spatialdb.UniformDB
db_initial.label = Dirichlet BC on -x
db_initial.values = [displacement-x]
db_initial.data = [+2.0*m]
```

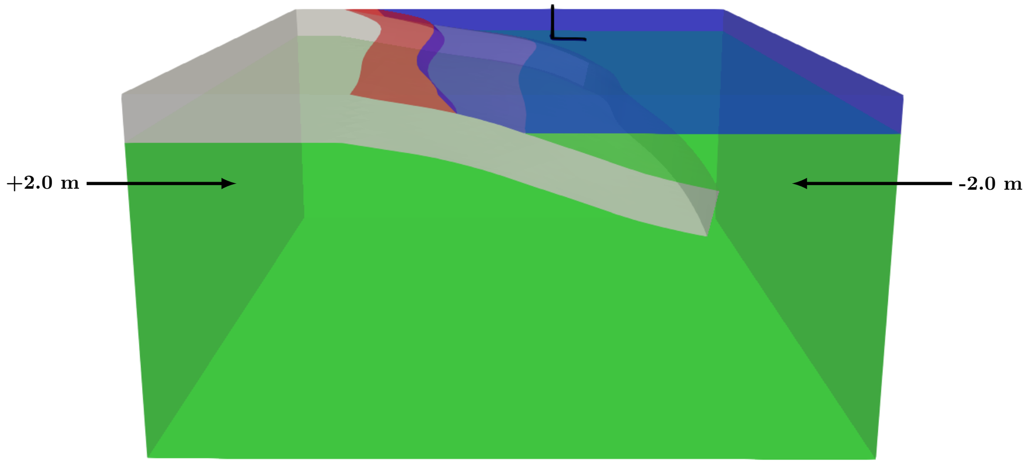



Figure 7.80: Diagram of Step 1: Axial compression. This static simulation uses Dirichlet boundary conditions with axial compression in the east-west (x-direction), roller boundary conditions on the north, south, and bottom boundaries, and purely elastic properties.

```
# +x face
[pylithapp.problem.bc.x_pos]
db_initial = spatialdata.spatialdb.UniformDB
db_initial.label = Dirichlet BC on +x
db_initial.values = [displacement-x]
db_initial.data = [-2.0*m]
```

As discussed in Section 7.18.4 on page 211, we use `mat_elastic.cfg` to specify the parameters associated with linear, isotropic elastic bulk constitutive models for all of the materials for convenient reuse across several different simulations.

Excerpt from `mat_elastic.cfg`

```
[pylithapp.problem.materials]
slab = pylith.materials.ElasticIsotropic3D
wedge = pylith.materials.ElasticIsotropic3D
crust = pylith.materials.ElasticIsotropic3D
mantle = pylith.materials.ElasticIsotropic3D

# Slab
[pylithapp.problem.materials.slab]
db_properties = spatialdata.spatialdb.SimpleDB
db_properties.label = Properties for subducting slab
db_properties.iohandler.filename = spatialdb/mat_slab_elastic.spatialdb

# Wedge
[pylithapp.problem.materials.wedge]
db_properties = spatialdata.spatialdb.SimpleDB
db_properties.label = Properties for accretionary wedge
db_properties.iohandler.filename = spatialdb/mat_wedge_elastic.spatialdb

# Mantle
[pylithapp.problem.materials.mantle]
db_properties = spatialdata.spatialdb.SimpleDB
db_properties.label = Properties for mantle
db_properties.iohandler.filename = spatialdb/mat_mantle_elastic.spatialdb

# Crust
[pylithapp.problem.materials.crust]
db_properties = spatialdata.spatialdb.SimpleDB
db_properties.label = Properties for continental crust
```

```
db_properties.iohandler.filename = spatialdb/mat_crust_elastic.spatialdb
```

We specify different elastic properties for each material (slab, wedge, mantle, and crust) using SimpleDB spatial databases with a single point to specify uniform properties within a material. We choose SimpleDB rather than UniformDB, because we will reuse some of these spatial databases for the elastic properties when we use linear Maxwell viscoelastic constitutive model.

The remaining parameters in the `step01.cfg` file are mostly associated with setting filenames for all of the various output, including all of the parameters used and version information in a JSON file (`output/step01-parameters.json`), a file reporting the progress of the simulation and estimated time of completion (`output/step01-progress.txt`), and the filenames for the HDF5 files (the corresponding Xdmf files will use the same filename with the `xmf` suffix).

Run Step 1 simulation

```
$ pylith step01.cfg mat_elastic.cfg
```

The simulation will produce ten pairs of HDF5/Xdmf files in the `output` directory:

**step01-domain.h5[.xmf]** Time series of the solution field over the domain.

**step01-groundsurf.h5[.xmf]** Time series of the solution field over the ground surface.

**step01-slab_info.h5[.xmf]** Properties for the slab material.

**step01-slab.h5[.xmf]** Time series of the state variables (stress and strain) for the slab material.

**step01-wedge_info.h5[.xmf]** Properties for the wedge material.

**step01-wedge.h5[.xmf]** Time series of the state variables (stress and strain) for the wedge material.

**step01-crust_info.h5[.xmf]** Properties for the crust material.

**step01-crust.h5[.xmf]** Time series of the tate variables (stress and strain) for the crust material.

**step01-mantle_info.h5[.xmf]** Properties for the mantle material.

**step01-mantle.h5[.xmf]** Time series of the state variables (stress and strain) for the mantle material.

The HDF5 files contain the data and the Xdmf files contain the metadata required by ParaView and Visit (and other visualization tools that use Xdmf files) to access the mesh and data sets in the HDF5 files.

Figure 7.81, which was created using the ParaView Python script `plot_dispvec.py` (see Section 7.2 on page 112 for how to run ParaView Python scripts), displays the magnitude of the displacement field arrows showing the direction and magnitude of the deformation. Material properties with a positive Poisson's ratio result in vertical deformation along with the axial compression. The variations in material properties among the properties result in local spatial variations that are most evident in the horizontal displacement components.

#### 7.18.5.1 Exercises

- Run PyLith again and add `solver_algebraicmultigrid.cfg` as an argument on the command line to switch to the algebraic multigrid preconditioner.
  - Using the PETSc log summary to compare the runtime and memory use between the original LU preconditioner and the ML algebraic multigrid preconditioner. Hint: The algebraic multigrid preconditioner is faster.
  - Run the simulation again with the algebraic multigrid preconditioner using multiple cores via the `--nodes=NCORES` argument, replacing `NCORES` with 2 or up to the number of cores on your machine. Examine the PETSc log summary for the various runs to see how the time spent at varies stages changes with the number of cores. Make a plot of runtime versus the number of cores.

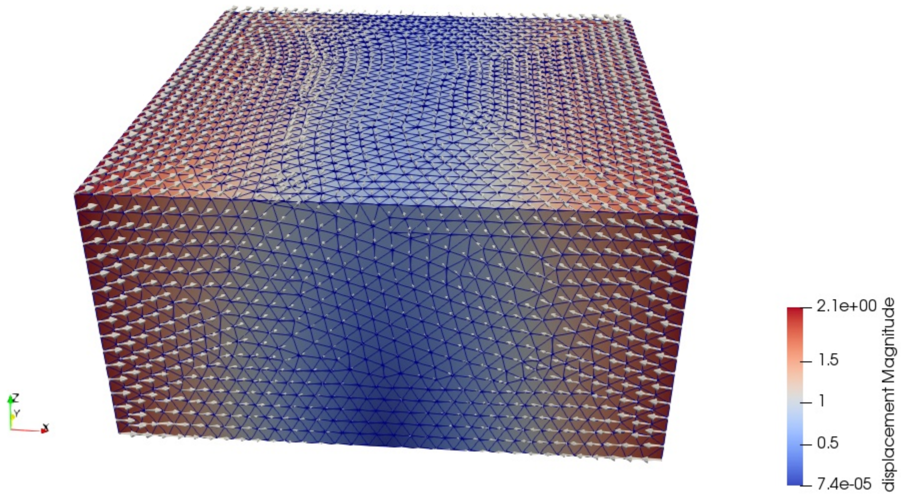



Figure 7.81: Solution over the domain for Step 1. The colors indicate the magnitude of the displacement and the arrows indicate the direction with the length of each arrow equal to 10,000 times the magnitude of the displacement.

- Adjust the material properties in the spatial databases so that the slab is stiffer and the wedge is more compliant. What happens to the solution if you make the materials nearly incompressible? Does this also affect the rate of convergence of the linear solve?
- Change the Dirichlet boundary conditions to impose pure shear instead of axial compression. Hint: You will need to change the boundary conditions on the east, west, north, and south boundaries.

### 7.18.6 Step 2: Prescribed Coseismic Slip and Postseismic Relaxation

In this example we model the postseismic relaxation of the deep slab and mantle resulting from coseismic slip on a fault patch in the central portion of the subduction (top of the slab) interface. For simplicity we will prescribed uniform slip on the fault patch and use a linear Maxwell viscoelastic constitutive models for the slab and mantle. As the lateral and bottom boundaries are far from the earthquake source, we use roller boundary conditions on these boundaries. We do not expect significant relaxation of stresses on the shallow part of the slab, so we impose a depth-dependent viscosity. Figure 7.82 summarizes the problem description.

The `pylithapp.cfg` completely specifies the Dirichlet roller boundary conditions on the five boundaries, so we do not include any boundary condition information in `step02.cfg`. As discussed in Section 7.18.4 on page 211, we bundle the parameters for specification of an elastic crust and wedge and viscoelastic slab and mantle in `mat_viscoelastic.cfg`.

We describe the properties of the linear, isotropic Maxwell viscoelastic constitutive model using viscosity in addition to the Vp, Vs, and density used to describe purely linear, isotropic elastic models. Rather than create a database with all four of these parameters, we leverage the SimpleDB spatial databases used by `mat_elastic.cfg` for the elastic properties and simply create a single new spatial database with the depth-dependent viscosity for the slab and mantle. We use the CompositeDB spatial database to combine these two spatial databases into a single spatial database with the material properties. Rather than using a SimpleDB for the depth-dependent viscosity, we use a SimpleGridDB spatial database (`spatialdb/mat_viscosity`), which provides faster interpolation using a bilinear search algorithm along each coordinate direction. We use a very large viscosity at depths above 20 km to give behavior that is essentially elastic and decrease it so the Maxwell relaxation time (viscosity divided by the shear modulus) is approximately 200 years at a depth of 30 km, 100 years at a depth of 100 km, and 50 years at a depth of 400 km. Using linear interpolation results in a piecewise linear variation in the viscosity with depth.

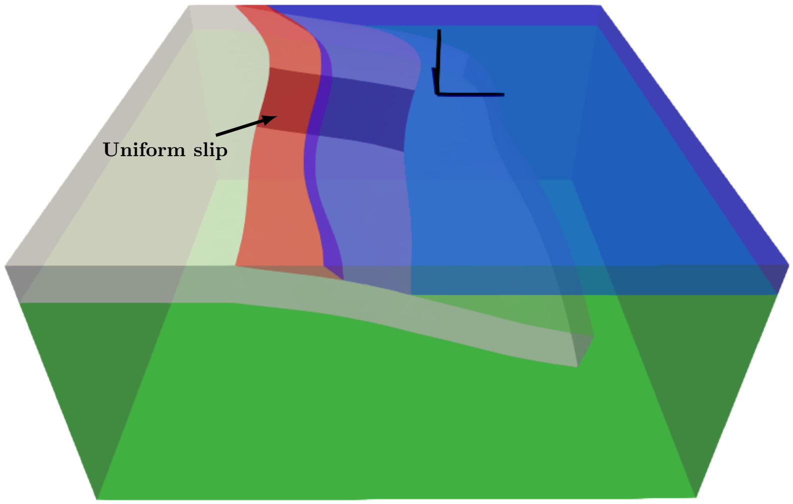



Figure 7.82: Diagram of Step 2: Prescribed coseismic slip and postseismic relaxation. This quasistatic simulation prescribes uniform slip on the central rupture patch on the subduction interface, depth-dependent viscoelastic relaxation in the slab and mantle, and roller boundary conditions on the lateral (north, south, east, and west) and bottom boundaries.

> ★ **Tip**
>
> The SimpleGridDB should be used whenever the points in a spatial database can be described with a logically rectangular grid. The grid points along each direction do not need to be uniformly spaced.

In setting the parameters for the CompositeDB in `mat_viscoelastic.cfg`, we specify which properties are contained in each of the two spatial databases in the composite database and the type and parameters for each of those spatial databases. For the slab we have:

Excerpt from `mat_viscoelastic.cfg`

```
[pylithapp.problem.materials.slab]
db_properties = spatialdata.spatialdb.CompositeDB
db_properties.label = Composite spatial database for slab material properties

[pylithapp.timedependent.materials.slab.db_properties]
# Elastic properties
values_A = [density, vs, vp]
db_A = spatialdata.spatialdb.SimpleDB
db_A.label = Elastic properties
db_A.iohandler.filename = spatialdb/mat_slab_elastic.spatialdb

# Viscoelastic properties
values_B = [viscosity]
db_B = spatialdata.spatialdb.SimpleGridDB
db_B.label = Linear Maxwell viscoelastic properties
db_B.filename = spatialdb/mat_viscosity.spatialdb
db_B.query_type = linear
```

In the simulation specific parameter file `step02.cfg`, we specify the parameters for the quasistatic time stepping, the coseismic rupture, and the filenames for output. By default, PyLith will use implicit time stepping with uniform time steps, so we need only specify the duration and time step size.

Excerpt from `step02.cfg`

```
[pylithapp.problem.formulation.time_step]
# Define the total time for the simulation and the time step size.
total_time = 200.0*year
dt = 10.0*year
```

In prescribing coseismic slip on the single fault patch, we create an array with one fault interface and then set its parameters. Because the edges of the central fault patch are buried within the domain, we need to specify the nodeset that corresponds to the buried edges as well as the nodeset for the entire fault surface. This ensures that PyLith inserts the cohesive cells and properly terminates the fault surface at the edges. Just as we do for the boundary conditions and materials, we create an array of components (in this case an array with one fault interface, **slab**, and then refer to those components by name, **pylithapp.problem.interfaces.slab**. We must also set the discretization information for the fault.

Excerpt from `step02.cfg`

```
[pylithapp.problem]
# We prescribe slip on the slab fault patch.
interfaces = [slab]

[pylithapp.problem.interfaces]
slab = pylith.faults.FaultCohesiveKin ; Default

[pylithapp.problem.interfaces.slab]
label = fault_slabtop_patch ; Nodeset for entire fault surface
edge = fault_slabtop_patch_edge ; Nodeset for buried edges

# We must define the quadrature information for fault cells.
# The fault cells are 2D (surface).
quadrature.cell = pylith.feassemble.FIATSimplex
quadrature.cell.dimension = 2
```

Prescribing the coseismic slip distribution on the fault involves specifying an origin time for the rupture (default is 0.0), and a slip time function along with its parameters. In this case, we treat the earthquake rupture as just the coseismic slip happening in one time step, so we use a step function for the slip time function (which is the default). The parameters include the final slip and slip initiation time. This slip initiation time is relative to the earthquake source origin time, which is 0 by default. Thus, to specify the time of the slip for a step function, we can either specify the origin time or the slip initiation time; in this case, we use the slip initiation time. In Step 4 we will use the origin time. Because we want uniform slip and a uniform rise time, we use UniformDB spatial databases for both of these. Note that we specify oblique slip with 1.0 m of right-lateral motion and 4.0 m of reverse motion.

Excerpt from `step02.cfg`

```
[pylithapp.problem.interfaces.slab.eq_srcs.rupture.slip_function]
slip = spatialdata.spatialdb.UniformDB
slip.label = Final slip
slip.values = [left-lateral-slip, reverse-slip, fault-opening]
slip.data = [-1.0*m, 4.0*m, 0.0*m]

slip_time = spatialdata.spatialdb.UniformDB
slip_time.label  = Slip initiation time
slip_time.values = [slip-time]
slip_time.data = [9.999*year] ; Use 10*year-small value to account for roundoff errors

[pylithapp.problem.interfaces.slab.output]
writer = pylith.meshio.DataWriterHDF5
writer.filename = output/step02-fault-slab.h5
vertex_info_fields = [normal_dir, strike_dir, dip_dir, final_slip_rupture, slip_time_rupture]
```

Run Step 2 simulation

```
$ pylith step02.cfg mat_viscoelastic.cfg solver_fieldsplit.cfg
```

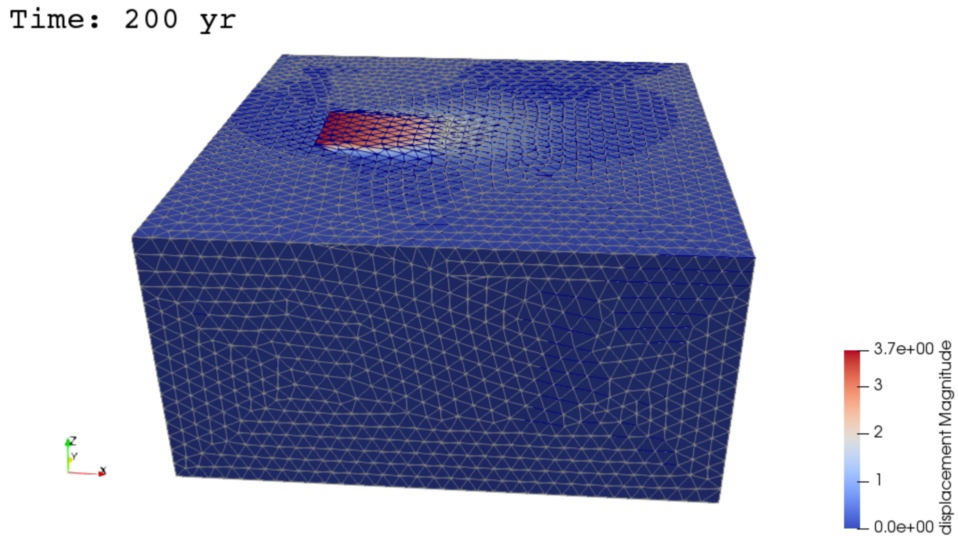



Figure 7.83: Solution over the domain for Step 2 at $t = 200$yr. The colors indicate the magnitude of the displacement and we have exaggerated the deformation by a factor of 10,000.

In addition to the ten pairs of HDF5/Xdmf files analogous to those produced in Step 1, we also have two pairs of HDF5/Xdmf files associated with the fault:

**`step02-domain.h5[.xmf]`** Time series of the solution field over the domain.

**`step02-groundsurf.h5[.xmf]`** Time series of the solution field over the ground surface.

**`step02-slab_info.h5[.xmf]`** Properties for the slab material.

**`step02-slab.h5[.xmf]`** Time series of the state variables (stress and strain) for the slab material.

**`step02-wedge_info.h5[.xmf]`** Properties for the wedge material.

**`step02-wedge.h5[.xmf]`** Time series of the state variables (stress and strain) for the wedge material.

**`step02-crust_info.h5[.xmf]`** Properties for the crust material.

**`step02-crust.h5[.xmf]`** Time series of the tate variables (stress and strain) for the crust material.

**`step02-mantle_info.h5[.xmf]`** Properties for the mantle material.

**`step02-mantle.h5[.xmf]`** Time series of the state variables (stress and strain) for the mantle material.

**`step02-fault-slab_info.h5[.xmf]`** Fault orientation and rupture information.

**`step02-fault-slab.h5[.xmf]`** Time series of slip and traction changes.

Figure 7.83, which was created using the ParaView Python script `plot_dispwarp.py`, displays the magnitude of the displacement field exaggerated by a factor of 10,000 at the final time step (200 yr). The shallow fault results in deformation that is localized over a small region.

#### 7.18.6.1 Exercises

- Change the slip from the subduction interface rupture patch to the splay fault rupture patch. Hint: Identify the nodesets for the splay fault patch.
- Create simultaneous rupture on the subduction interface rupture patch and the splay fault rupture patch.

- Prescribe coseismic slip on the central patch for splay fault and the subduction interface below the intersection with the splay fault.
  - Implement this without changing any of the nodesets in CUBIT/Trelis. Hint: you will need to create two fault interfaces. What do you notice about the slip at the intersection between the splay fault and slab?
  - Add nodesets in CUBIT/Trelis to create a uniform coseismic slip distribution across the splay fault and on the subduction interface below the splay fault.

### 7.18.7 Step 3: Prescribed Aseismic Creep and Interseismic Deformation

We now increase the complexity of our fault model by simulating the interseismic deformation associated with the subducting slab. We approximate the motion of the Juan de Fuca Plate subducting under the North American Plate by introducing aseismic slip (creep) on the bottom of the slab and the deeper portion of the subduction interface; we keep the interface between the subduction interface and the accretionary wedge and shallow crust locked. As in Step 2, we will use the linear Maxwell viscoelastic constitutive model for the slab and mantle. Figure 7.84 summarizes the problem description.

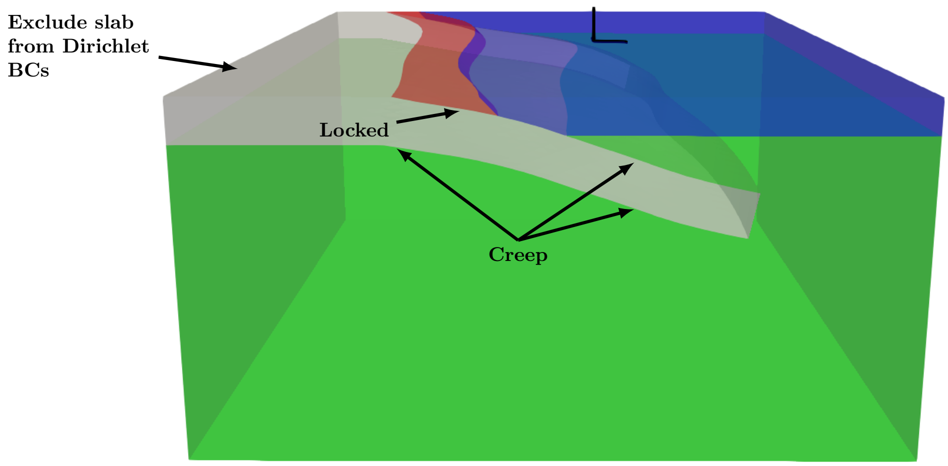



Figure 7.84: Diagram of Step 3: Prescribed aseismic slip (creep) and interseismic deformation for the subducting slab. We prescribe steady, uniform creep on the bottom of the slab and deeper portion of the subduction interface. We impose roller Dirichlet boundary conditions on the lateral and bottom boundaries, except where they overlap with the slab and splay fault.

With slip on the top and bottom of the slab, our fault interfaces array contains two components, one for the top of the slab (subduction interface), **slab_top**, and one for the bottom of the slab, **slab_bottom**. We use the FaultCohesiveKin object for each of these interfaces since we want to prescribe the slip.

Excerpt from step03.cfg

```
[pylithapp.problem]
interfaces = [slab_bottom, slab_top]

[pylithapp.problem.interfaces]
slab_bottom = pylith.faults.FaultCohesiveKin
slab_top = pylith.faults.FaultCohesiveKin
```

We specify the **id** used to identify the cohesive cells for this fault so that it is unique among all materials and faults. We also specify the appropriate nodesets identifying the entire fault surface and the buried edges. Some portions of the bottom of the slab are perfectly horizontal, so our procedure that uses the vertical direction and the fault normal to set the along-strike and up-dip shear components breaks down. We remedy this by tweaking the **up_dir** direction from being completely vertical (0,0,1) to tilting slightly to the west. This results in consistent along-strike and up-dip directions across the fault surface. For the aseismic slip we use a constant slip rate time function (ConstRateSlipFn) with UniformDB spatial databases to specify the constant, uniform oblique slip rate of 2.0 cm/yr of left-lateral motion and 4.0 cm/yr of normal motion. Note that slip on the bottom of the subducting slab has the opposite sense of motion as that on the top of the slab.

Excerpt from `step03.cfg`

```
[pylithapp.problem.interfaces.slab_bottom]
id = 100 ; Must be different from ids used for materials
label = fault_slabbot ; Nodeset for the entire fault surface
edge = fault_slabbot_edge ; Nodeset for the buried edges
# Give slight westward tilt to the up_dir to avoid ambiguous
# directions for the shear components on the horizontal portions of the
# fault.
up_dir = [-0.1,0,0.9]

# We must define the quadrature information for fault cells.
# The fault cells are 2D (surface).
quadrature.cell = pylith.feassemble.FIATSimplex
quadrature.cell.dimension = 2

# Use the constant slip rate time function.
eq_srcs.rupture.slip_function = pylith.faults.ConstRateSlipFn

# The slip time and final slip are defined in spatial databases.
[pylithapp.problem.interfaces.slab_bottom.eq_srcs.rupture.slip_function]
slip_rate = spatialdata.spatialdb.UniformDB
slip_rate.label = Slab bottom slip rate.
slip_rate.values = [left-lateral-slip, reverse-slip, fault-opening]
slip_rate.data = [+2.0*cm/year, -4.0*cm/year, 0.0*cm/year]

slip_time = spatialdata.spatialdb.UniformDB
slip_time.label  = Slip initiation time
slip_time.values = [slip-time]
slip_time.data = [0.0*year]

[pylithapp.problem.interfaces.slab_bottom.output]
writer = pylith.meshio.DataWriterHDF5
writer.filename = output/step03-fault-slabbot.h5
vertex_info_fields = [normal_dir, strike_dir, dip_dir]
```

The parameters for the top of the slab (subduction interface) closely resemble those for the bottom of the slab. The main difference is that we use a SimpleGridDB to define a depth variation in the slip rate. The fault is locked at depths above 45 km and increases linearly to the same slip rate as the bottom of the slab at a depth of 60 km.

Excerpt from `step03.cfg`

```
[pylithapp.problem.interfaces.slab_top]
id = 101 ; Must be different from ids used for materials
label = fault_slabtop ; Nodeset for the entire fault surface
edge = fault_slabtop_edge ; Nodeset for the buried edges

# We must define the quadrature information for fault cells.
# The fault cells are 2D (surface).
quadrature.cell = pylith.feassemble.FIATSimplex
quadrature.cell.dimension = 2

# Use the constant slip rate time function.
eq_srcs.rupture.slip_function = pylith.faults.ConstRateSlipFn

# The slip time and final slip are defined in spatial databases.
[pylithapp.problem.interfaces.slab_top.eq_srcs.rupture.slip_function]
slip_rate = spatialdata.spatialdb.SimpleGridDB
slip_rate.label = Slab top slip rate.
slip_rate.filename = spatialdb/fault_slabtop_creep.spatialdb
slip_rate.query_type = linear
```

```
slip_time = spatialdata.spatialdb.UniformDB
slip_time.label  = Slip initiation time
slip_time.values = [slip-time]
slip_time.data = [0.0*year]

[pylithapp.problem.interfaces.slab_top.output]
writer = pylith.meshio.DataWriterHDF5
writer.filename = output/step03-fault-slabtop.h5
vertex_info_fields = [normal_dir, strike_dir, dip_dir]
```

We do not want the boundaries to constrain the motion of the subducting slab, so we use the nodesets that exclude vertices on the subducting slab. Furthermore, PyLith does not permit overlap between the fault interfaces and Dirichlet boundary conditions. This is why we exclude vertices on the splay fault in these nodesets as well. We only update the name of the nodeset for the -x, -y, and +y boundaries.

Excerpt from `step03.cfg`

```
# -x face
[pylithapp.problem.bc.x_neg]
label = boundary_xneg_noslab

# -y face
[pylithapp.problem.bc.y_neg]
label = boundary_yneg_noslab

# +y face
[pylithapp.problem.bc.y_pos]
label = boundary_ypos_noslab
```

Run Step 3 simulation

```
$ pylith step03.cfg mat_viscoelastic.cfg solver_fieldsplit.cfg
```

The simulation will produce fourteen pairs of HDF5/Xdmf files, beginning with `step03`, in the `output` directory:

**`step03-domain.h5[.xmf]`** Time series of the solution field over the domain.

**`step03-groundsurf.h5[.xmf]`** Time series of the solution field over the ground surface.

**`step03-slab_info.h5[.xmf]`** Properties for the slab material.

**`step03-slab.h5[.xmf]`** Time series of the state variables (stress and strain) for the slab material.

**`step03-wedge_info.h5[.xmf]`** Properties for the wedge material.

**`step03-wedge.h5[.xmf]`** Time series of the state variables (stress and strain) for the wedge material.

**`step03-crust_info.h5[.xmf]`** Properties for the crust material.

**`step03-crust.h5[.xmf]`** Time series of the tate variables (stress and strain) for the crust material.

**`step03-mantle_info.h5[.xmf]`** Properties for the mantle material.

**`step03-mantle.h5[.xmf]`** Time series of the state variables (stress and strain) for the mantle material.

**`step03-fault-slabbot_info.h5[.xmf]`** Fault orientation and rupture information for the bottom of the slab.

**`step03-fault-slabbot.h5[.xmf]`** Time series of slip and traction changes for the bottom of the slab.

**`step03-fault-slabtop_info.h5[.xmf]`** Fault orientation and rupture information for the top of the slab.

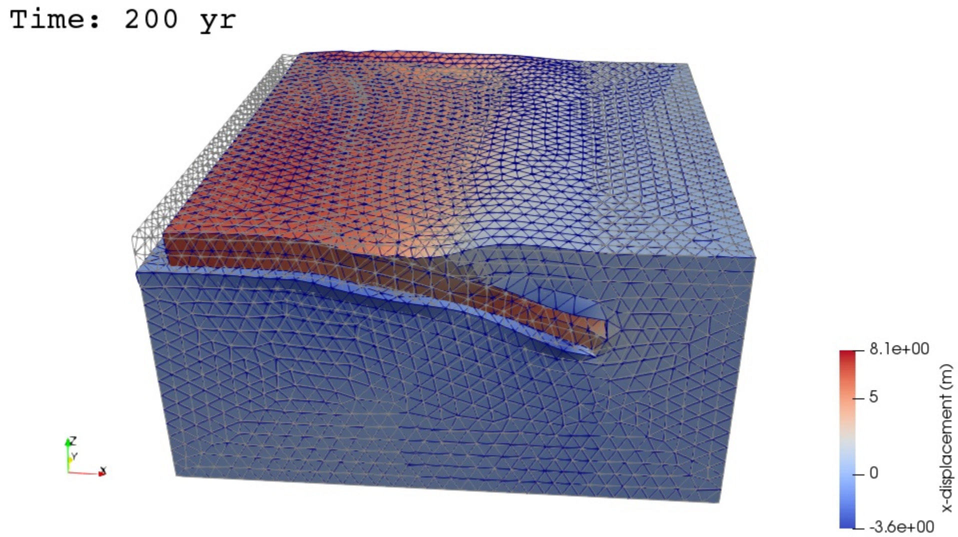



Figure 7.85: Solution over the domain for Step 2 at $t = 200$yr. The colors indicate the x-displacement and we have exaggerated the deformation by a factor of 5,000.

**step03-fault-slabtop.h5[.xmf]** Time series of slip and traction changes for the top of the slab.

As in Step 2, there are two pairs of HDF5/Xdmf files for each fault; one set for the fault orientation and rupture information and one set for the time series of slip and change in tractions.

Figure 7.85, which was created using the ParaView Python script `plot_dispwarp.py`, shows the deformation exaggerated by a factor of 5,000 at the final time step of t=200*yr. Notice that there are some local edge effects associated with the unconstrained degrees of freedom at the intersection of the boundaries and fault surfaces.

#### 7.18.7.1 Exercises

- Adjust the locking depth for the subduction interface. How does this affect the spatial distribution of the change in tractions on the fault interfaces?
- Increase the rigidity of the slab and decrease the rigidity of the wedge and/or crust. How do these affect the change in tractions on the fault interfaces?

### 7.18.8 Step 4: Prescribed Earthquake Cycle

In Step 4, We combine the interseismic deformation in Step 3 with the coseismic slip in Step 2 to simulate two earthquake cycles. We also include an earthquake on the splay fault. This illustrates how to include multiple earthquake sources on a single fault. We use the same roller Dirichlet boundary conditions and combination of elastic and viscoelastic materials as we did in Step 3.

We create an array of three fault interfaces, one for the top of the slab (subduction interface), one for the bottom of the slab, and one for the splay fault. The splay fault terminates into the fault on the top of the slab, so we must list the through-going fault on the top of the slab first.

Excerpt from `step04.cfg`

```
# We prescribe slip on the top and bottom of the slab and on the splay fault.
[pylithapp.problem]
interfaces = [slab_top, slab_bottom, splay]

[pylithapp.problem.interfaces]
slab_top = pylith.faults.FaultCohesiveKin
```

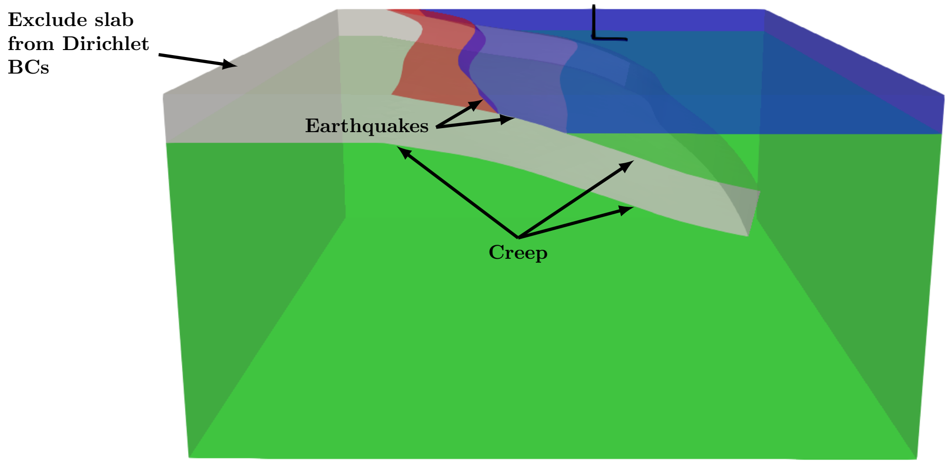



Figure 7.86: Diagram of Step 4: A simple earthquake cycle combining the prescribed aseismic slip (creep) from Step 3 with prescribed coseismic slip for two earthquakes on the shallow portion of the subduction interface and one earthquake on the play fault. We impose roller Dirichlet boundary conditions on the lateral and bottom boundaries, except where they overlap with the slab and splay fault.

```
slab_bottom = pylith.faults.FaultCohesiveKin
splay = pylith.faults.FaultCohesiveKin
```

> ⚠ **Important**
>
> When including intersecting faults, the through-going fault must be listed first in the array of fault interfaces. This ensures its cohesive cells are created before the adjacent fault that terminates into the through-going fault. For nonintersecting faults, the order in the list of fault interfaces does not matter.

The settings for the fault interface on the bottom of the slab match those used in Step 3. For the subduction interface, we want to impose creep on the deeper portion and earthquakes (coseismic slip) at specific times on the upper portion. We create an array of earthquake sources, one for the creep and one for each of the earthquakes. We want the earthquake to be imposed at specific times, so we set their origin time equal to the desire rupture time (100 years and 200 years) minus a value much smaller than the time step, so that roundoff errors do not result in the ruptures occurring one time step later than intended. We use the same settings as we did in Step 3 for the creep earthquake source. For the coseismic slip, we use a SimpleGridDB to impose a depth-dependent slip distribution that exactly complements the depth-dependent slip distribution of the creep. Note that the slip time within an earthquake rupture is relative to the origin time, so we set the slip time to zero to coincide with the specified origin time.

Excerpt from `step04.cfg`

```
[pylithapp.problem.interfaces.slab_top]
# --- Skipping lines already discussed in Step 3 ---
eq_srcs = [creep, eq1, eq2]
eq_srcs.creep.origin_time = 0.0*year
eq_srcs.eq1.origin_time = 99.999*year ; 100*yr - small value
eq_srcs.eq2.origin_time = 199.999*year l 200*yr - small value

# Use the constant slip rate time function for the creep earthquake source.
eq_srcs.creep.slip_function = pylith.faults.ConstRateSlipFn

# Creep
[pylithapp.problem.interfaces.slab_top.eq_srcs.creep.slip_function]
```

```
slip_rate = spatialdata.spatialdb.SimpleGridDB
slip_rate.label = Slab top slip rate.
slip_rate.filename = spatialdb/fault_slabtop_creep.spatialdb
slip_rate.query_type = linear

slip_time = spatialdata.spatialdb.UniformDB
slip_time.label  = Slip initiation time
slip_time.values = [slip-time]
slip_time.data = [0.0*year] ; Slip time is relative to origin time

# Earthquake 1
[pylithapp.problem.interfaces.slab_top.eq_srcs.eq1.slip_function]
slip = spatialdata.spatialdb.SimpleGridDB
slip.label = Slab top slip rate.
slip.filename = spatialdb/fault_slabtop_coseismic.spatialdb
slip.query_type = linear

slip_time = spatialdata.spatialdb.UniformDB
slip_time.label  = Slip initiation time
slip_time.values = [slip-time]
slip_time.data = [0.0*year] ; Slip time is relative to origin time.

# Earthquake 2 (same as earthquake 1)
[pylithapp.problem.interfaces.slab_top.eq_srcs.eq2.slip_function]
slip = spatialdata.spatialdb.SimpleGridDB
slip.label = Slab top slip rate.
slip.filename = spatialdb/fault_slabtop_coseismic.spatialdb
slip.query_type = linear

slip_time = spatialdata.spatialdb.UniformDB
slip_time.label  = Slip initiation time
slip_time.values = [slip-time]
slip_time.data = [0.0*year] ; Slip time is relative to origin time.
# --- Omitting output settings already discussed ---
```

The settings for the splay fault look very similar to those for the coseismic slip on the slab rupture patch in Step 2. The primary difference is that we specify an origin time of 250 years.

Excerpt from `step04.cfg`

```
[pylithapp.problem.interfaces.splay]
id = 102 ; id must be unique across all materials and faults
label = fault_splay ; Nodeset for the entire fault surface
edge = fault_splay_edge ; Nodeset for the buried edges

# We must define the quadrature information for fault cells.
# The fault cells are 2D (surface).
quadrature.cell = pylith.feassemble.FIATSimplex
quadrature.cell.dimension = 2

# Origin time for splay fault earthquake.
eq_srcs.rupture.origin_time = 249.999*year

# The slip time and final slip are defined in spatial databases.
[pylithapp.problem.interfaces.splay.eq_srcs.rupture.slip_function]
slip = spatialdata.spatialdb.UniformDB
slip.label = Splay fault slip.
slip.values = [left-lateral-slip, reverse-slip, fault-opening]
slip.data = [-1.0*m, 2.0*m, 0.0*m]

slip_time = spatialdata.spatialdb.UniformDB
slip_time.label  = Slip initiation time
slip_time.values = [slip-time]
slip_time.data = [0.0*year] ; Relative to the origin time
```

```
# --- Omitting output settings already discussed ---
```

Run Step 4 simulation

```
$ pylith step04.cfg mat_viscoelastic.cfg solver_fieldsplit.cfg
```

The simulation will produce sixteen pairs of HDF5/Xdmf files, beginning with `step04`, in the `output` directory:

**`step04-domain.h5[.xmf]`** Time series of the solution field over the domain.

**`step04-groundsurf.h5[.xmf]`** Time series of the solution field over the ground surface.

**`step04-slab_info.h5[.xmf]`** Properties for the slab material.

**`step04-slab.h5[.xmf]`** Time series of the state variables (stress and strain) for the slab material.

**`step04-wedge_info.h5[.xmf]`** Properties for the wedge material.

**`step04-wedge.h5[.xmf]`** Time series of the state variables (stress and strain) for the wedge material.

**`step04-crust_info.h5[.xmf]`** Properties for the crust material.

**`step04-crust.h5[.xmf]`** Time series of the tate variables (stress and strain) for the crust material.

**`step04-mantle_info.h5[.xmf]`** Properties for the mantle material.

**`step04-mantle.h5[.xmf]`** Time series of the state variables (stress and strain) for the mantle material.

**`step04-fault-slabbot_info.h5[.xmf]`** Fault orientation and rupture information for the bottom of the slab.

**`step04-fault-slabbot.h5[.xmf]`** Time series of slip and traction changes for the bottom of the slab.

**`step04-fault-slabtop_info.h5[.xmf]`** Fault orientation and rupture information for the top of the slab.

**`step04-fault-slabtop.h5[.xmf]`** Time series of slip and traction changes for the top of the slab.

**`step04-fault-splay_info.h5[.xmf]`** Fault orientation and rupture information for the splay fault.

**`step04-fault-splay.h5[.xmf]`** Time series of slip and traction changes for the splay fault.

Figure 7.87, which was created using the ParaView Python script `plot_dispwarp.py`, shows the deformation exaggerated by a factor of 5,000 at the final time step of t=300*yr. Compared to the solution in Step 3, we see the earthquakes have reduced the deformation in the crust and accretionary wedge.

#### 7.18.8.1 Exercises

- Adjust the timing of the earthquake rupture sequence. How does this affect the deformation?
- Add additional earthquakes with different depth variations in slip, keeping the total equal to the overall slip rate.
- Adjust the nodesets in CUBIT/Trelis so that the splay fault and the deeper portion of the subduction interface form the through-going fault and the upper portion of the subduction interface is the secondary fault. How does this affect the stress accumulation in the crust and upper mantle?

### 7.18.9 Step 5: Spontaneous Rupture Driven by Subducting Slab

This example is not yet complete. The parameters need to be fine tuned to produce the desired behavior and improve the convergence for the nonlinear solve. See Steps 5 and 6 in Section 7.10 on page 174 for a 2-D example.

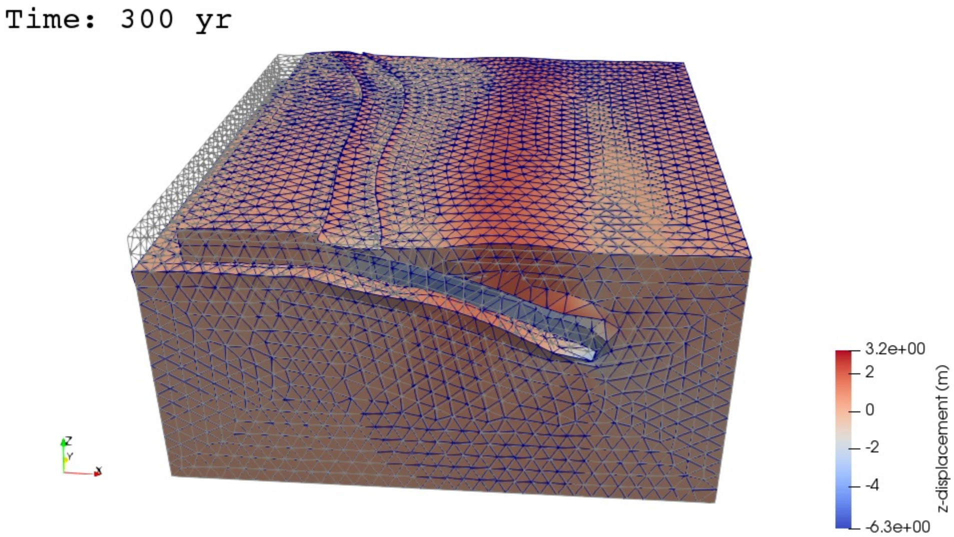



Figure 7.87: Solution over the domain for Step 4 at $t = 300$yr. The colors indicate the z-displacement and we have exaggerated the deformation by a factor of 5,000.

### 7.18.10 Step 6: Prescribed Slow-Slip Event

This example simulates a simple slow slip event (SSE) on the subduction interface, in which the entire patch slips simultaneously with an amplitude that grows with time. We impose a constant rake angle of 110 degrees, and a time duration of 30 days. The time duration is much shorter than the Maxwell time for our viscoelastic materials, so we use elastic material properties (as we did in Step 1).

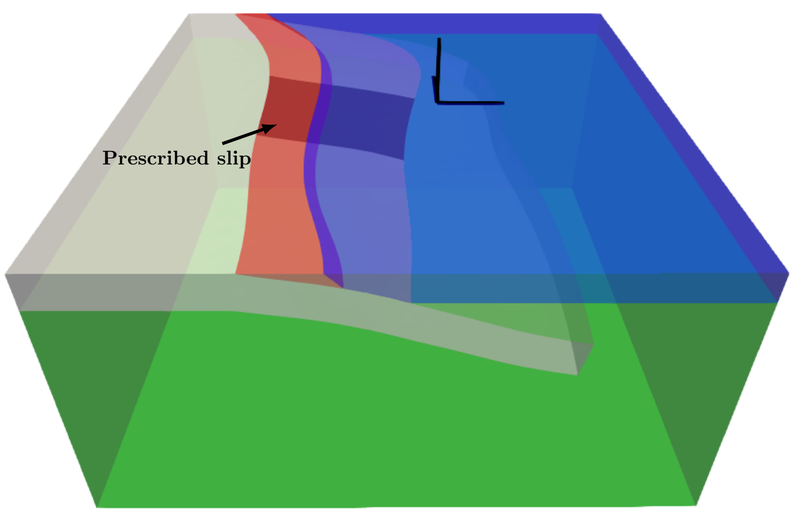



Figure 7.88: Diagram of Step 6: Prescribed slow-slip event on the subduction interface. This quasistatic simulation prescribes a Gaussian slip distribution on the central rupture patch of the subduction interface, purely elastic material properties, and roller boundary conditions on the lateral (north, south, east, and west) and bottom boundaries.

The only time dependence in this problem is the time evolution of slip, so we set the duration of the simulation to match the duration of the slow slip event. We use a time step of 2.0 days to insure that we resolve the temporal evolution of the slip.

Excerpt from `step06.cfg`

```
[pylithapp.problem.formulation.time_step]
total_time = 30.0*day
```

```
dt = 2.0*day
```

The results in this example will be used to simulate output at fake continuous GPS (cGPS) stations in Step 7, so we add an output manager for saving the solution at specific points (OutputSolnPoints) in addition to our output managers over the domain and top surface:

Excerpt from `step06.cfg`

```
[pylithapp.problem.implicit]
output = [domain, subdomain, cgps_sites]

# Default output is for the entire domain.
# We need to set the type of output for the subdomain and points.
output.subdomain = pylith.meshio.OutputSolnSubset
output.cgps_sites = pylith.meshio.OutputSolnPoints
```

For the point output we specify the output data writer, the file containing the list of cGPS stations and the coordinate system associated with the station locations. The format of the station file is whitespace separated columns of station name and then the coordinates of the station. See Section C.6 on page 273 for more information.

Excerpt from `step06.cfg`

```
[pylithapp.problem.formulation.output.cgps_sites]
writer = pylith.meshio.DataWriterHDF5
writer.filename = output/step06-cgps_sites.h5

# File with coordinates of cGPS stations.
reader.filename = cgps_sites.txt

# Specify coordinate system used in cGPS station file.
coordsys = spatialdata.geocoords.CSGeo
coordsys.space_dim = 3
coordsys.datum_horiz = WGS84
coordsys.datum_vert = mean sea level
```

The fault parameters are very similar to those in Step 2, in which we also prescribed slip on the subduction interface patch. The primary difference is that we use a user-defined slip time history function (TimeHistorySlipFn). This slip time function requires spatial databases for the amplitude of the final slip and slip initiation time, and a time history file specifying the normalized amplitude as a function of time. Additionally, to illustrate PyLith's ability to use spatial databases with points in other, but compatible, georeferenced coordinate systems, we specify the slip distribution using geographic (longitude and latitude) coordinates.

Excerpt from `step06.cfg`

```
[pylithapp.problem]
# We prescribe slip on the slab fault patch.
interfaces = [slab]

[pylithapp.problem.interfaces]
slab = pylith.faults.FaultCohesiveKin

[pylithapp.problem.interfaces.slab]
# Nodeset corresponding to the fault patch and buried edge.
label = fault_slabtop_patch
edge = fault_slabtop_patch_edge

# We must define the quadrature information for fault cells.
# The fault cells are 2D (surface).
quadrature.cell = pylith.feassemble.FIATSimplex
quadrature.cell.dimension = 2

# We use a time history slip function.
```

```
[pylithapp.problem.interfaces.slab.eq_srcs.rupture]
slip_function = pylith.faults.TimeHistorySlipFn

# The slip is defined in a SimpleGridDB spatial database with linear interpolation.
[pylithapp.problem.interfaces.slab.eq_srcs.rupture.slip_function]
slip = spatialdata.spatialdb.SimpleGridDB
slip.label = Gaussian slip distribution for SSE
slip.filename = spatialdb/fault_slabtop_slowslip.spatialdb
slip.query_type = linear

# We use a UniformDB to specify the slip initiation time.
slip_time = spatialdata.spatialdb.UniformDB
slip_time.label = Slip initiation time
slip_time.values = [slip-time]
slip_time.data = [0.0*year]

# We use a temporal database to provide the slip time history.
time_history.label = Time history of slip
time_history.filename = spatialdb/fault_slabtop_slowslip.timedb
```

You will notice that the `spatialdb` directory does not contain the `fault_slabtop_slowslip.spatialdb` and `fault_slabtop_slowslip.timedb` files. We use the `generate_slowslip.py` Python script to generate these files as an illustration of how to use Python to generate more simple spatial variations and the SimpleGridAscii object to write spatial database files. This script reads parameters from `generate_slowslip.cfg` to generate a Gaussian slip distribution in geographic coordinates, along with a temporal database providing the slip amplitudes at different times.

★ **Tip**

The `generate_slowslip.py` script is one of several examples where we make use of the Python interface to the spatialdata package. This provides useful methods for handling coordinate systems and spatial databases.

To run the simulation, first run the Python script to generate the spatial database files, and then run PyLith.

Run Step 6 simulation

```
# Generate the spatial database files
$ cd spatialdb && ./generate_slowslip.py
$ ls fault_slabtop_slowslip.*
# You should see
fault_slabtop_slowslip.spatialdb  fault_slabtop_slowslip.timedb
# Change back to the subduction directory and run PyLith
$ cd ..
$ pylith step06.cfg mat_elastic.cfg solver_fieldsplit.cfg
```

The problem will produce thirteen pairs of HDF5/Xdmf files:

**step06-domain.h5[.xmf]** Time series of the solution field over the domain.

**step06-groundsurf.h5[.xmf]** Time series of the solution field over the ground surface.

**step06-cgps_sites.h5[.xmf]** Time series of the solution field at the cGPS sites.

**step06-slab_info.h5[.xmf]** Properties for the slab material.

**step06-slab.h5[.xmf]** Time series of the state variables (stress and strain) for the slab material.

**step06-wedge_info.h5[.xmf]** Properties for the wedge material.

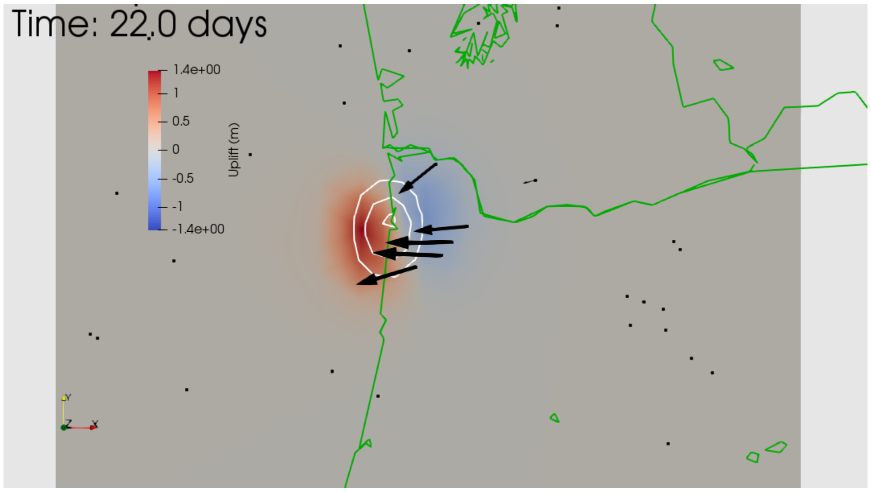



Figure 7.89: Solution for Step 6. The colors indicate the vertical displacement, the vectors represent the horizontal displacements at fake cGPS sites, and the contours represent the applied slip at t = 24 days.

**`step06-wedge.h5[.xmf]`** Time series of the state variables (stress and strain) for the wedge material.

**`step06-crust_info.h5[.xmf]`** Properties for the crust material.

**`step06-crust.h5[.xmf]`** Time series of the tate variables (stress and strain) for the crust material.

**`step06-mantle_info.h5[.xmf]`** Properties for the mantle material.

**`step06-mantle.h5[.xmf]`** Time series of the state variables (stress and strain) for the mantle material.

**`step06-fault-slab_info.h5[.xmf]`** Fault orientation and rupture information for the top of the slab.

**`step06-fault-slab.h5[.xmf]`** Time series of slip and traction changes for the top of the slab.

The additional HDF5 file that was not present in previous examples is `step06-cgps_sites.h5`, which contains the displacements at the fake cGPS sites.

Figure 7.89, which was created using ParaView, shows the surface vertical displacement along with horizontal displacement vectors at the cGPS sites, superimposed on contours of the applied slip at $t = 24$days.

#### 7.18.10.1 Exercises

- Change spatial distribution and time history of slip.
  - Edit `generate_slowslip.cfg` to change spatial and temporal distributions, and edit `step06.cfg` to change the time duration and/or time step size.
- Add propagation of the slow slip (spatial variation of slip initiation time).
  - Either alter Python script to produce a spatial database of slip initiation times, or write a new script. Can you produce a more realistic-looking slow slip event?

### 7.18.11 Step 7: Inversion of Slow-Slip Event using 3-D Green's Functions

This example is a three-dimensional analog of 7.16 on page 194 and is a more realistic example of how PyLith can be used to perform geodetic inversions. We divide generating Green's functions for slip impulses on the central rupture patch of the subduction interface two sub-problems:

**Step 7a** Left-lateral slip component.

**Step 7b** Reverse slip component.

Although PyLith can generate the two components in one simulation, we often prefer to speed up the process by running simulations for each of the components at the same time using multiple processes on a cluster.

To generate the Green's functions we change the problem from the default TimeDependent to GreensFns. We do this on the command line (as illustrated below). PyLith automatically reads the `greensfns.cfg` parameter file. This file contains settings that are common to both sub-problems. Note that the settings in the `greensfns.cfg` only apply to parameters associated with the GreensFns and its sub-components. For the Green's function problem, we must specify the fault interface and the id for the fault. We specify the amplitude of the impulses via a UniformDB spatial database, because we want impulses over the entire fault patch. We also request the amplitude of the impulses to be included in the fault info file.

Excerpt from `greensfns.cfg`

```
# Define the interfaces (slab) and provide a fault_id.
[greensfns]
interfaces = [slab]
fault_id = 100

# Switch fault to FaultCohesiveImpulses for generation of Green's functions.
[greensfns.interfaces]
slab = pylith.faults.FaultCohesiveImpulses

[greensfns.interfaces.slab]
# Nodesets corresponding to the fault and its buried edges.
label = fault_slabtop_patch
edge = fault_slabtop_patch_edge

# We must define the quadrature information for fault cells.
# The fault cells are 2D (surface).
quadrature.cell = pylith.feassemble.FIATSimplex
quadrature.cell.dimension = 2

# Spatial database for slip impulse amplitude.
db_impulse_amplitude = spatialdata.spatialdb.UniformDB
db_impulse_amplitude.label = Amplitude of fault slip impulses
db_impulse_amplitude.values = [slip]
db_impulse_amplitude.data = [1.0]

# Add impulse amplitude to fault info output.
output.vertex_info_fields = [normal_dir, strike_dir, dip_dir, impulse_amplitude]
output.writer = pylith.meshio.DataWriterHDF5
```

We do not make use of the state variable output for the impulse responses, so we turn off the data fields for all of the materials to eliminate these large data files.

Excerpt from `greensfns.cfg`

```
# Turn off output of state variables for materials.
[greensfns.materials.slab.output]
cell_data_fields = []

[greensfns.materials.wedge.output]
cell_data_fields = []

[greensfns.materials.crust.output]
cell_data_fields = []

[greensfns.materials.mantle.output]
cell_data_fields = []
```

The `step07a.cfg` and `step07b.cfg` files are identical, except for the impulse type specification and file names.

Excerpt from step07a.cfg

```
[pylithapp.problem.interfaces.slab]
# If we wanted to generate impulses for both the left-lateral and
# reverse components in the same simulation, we would use:
# impulse_dof = [0,1]
#
# Impulses for left-lateral slip.
impulse_dof = [0]
```

In the output settings, we turn off writing the solution field for the domain:

Excerpt from step07a.cfg

```
[pylithapp.problem.formulation.output.domain]
writer.filename = output/step07a-domain.h5
# Turn off data fields.
vertex_data_fields = []
```

Run Step 7 simulations

```
$ pylith --problem=pylith.problems.GreensFns step07a.cfg mat_elastic.cfg solver_fieldsplit.cfg
$ pylith --problem=pylith.problems.GreensFns step07b.cfg mat_elastic.cfg solver_fieldsplit.cfg
```

Each simulation will produce four pairs of HDF5/Xdmf files. For Step 7a these will be:

**step07a-groundsurf.h5[.xmf]** Solution field over the ground surface for each slip impulse.

**step07a-cgps_sites.h5[.xmf]** Solution field at continuous GPS sites for each slip impulse.

**step07a-fault-slab_info.h5[.xmf]** Fault orientation and impulse information.

**step07a-fault-slab.h5[.xmf]** Fault slip for each slip impulse.

> ★ **Tip**
>
> To save time, run the two sub-problems simultaneously in separate shells (terminal windows or tabs). For a problem this size, this should work fine on a laptop. For larger problems, we would run the simulations via separate jobs on a cluster with each job running on multiple processes.

Before we can run the inversion, we post-process the output from Step 6 to create synthetic data. We use the same generalized inverse approach described in 7.16.6 on page 199. The Python script make_synthetic_gpsdisp.py reads the parameters in make_synthetic_gpsdisp.cfg and generates synthetic data from the selected time step with a specified amount of noise.

Generate synthetic GPS data

```
$ ./make_synthetic_gpsdisp.py
```

This will create the following files:

**cgps_synthetic_displacement.txt** read by the inversion script.

**cgps_synthetic_displacement.vtk** for visualization.

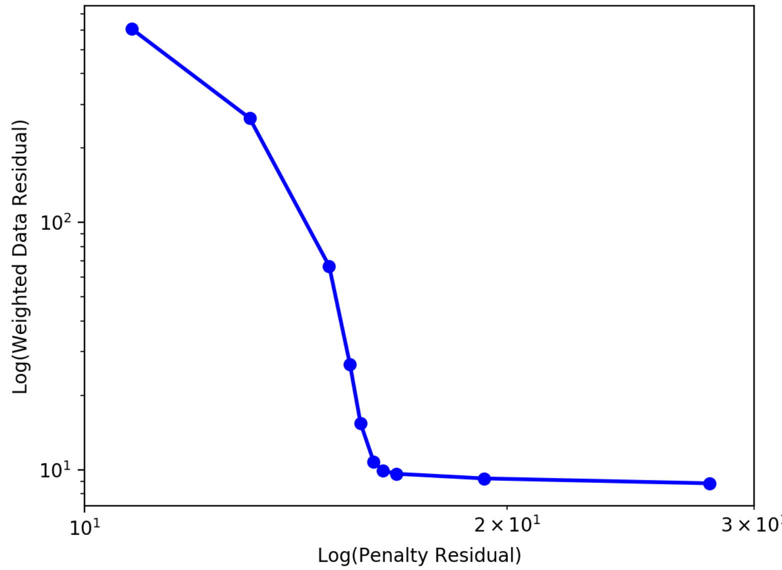



Figure 7.90: Plot of the 'L-curve' for inversion in Step 7. The 'corner' of the L-curve would be about the third or fourth point from the right of the plot, representing a penalty weight of 0.5 or 1.0 in our example.

We perform a simple inversion using the `slip_invert.py` script, with parameters defined in `slip_invert.cfg`. This script performs a set of linear inversions, in a manner similar to the inversion in 7.16.6 on page 199.

Run the inversion

```
$ ./slip_invert.py
```

This will create a number of files in the output directory.

**step07-inversion-slip.h5** This HDF5/Xdmf pair of files may be used to visualize the predicted slip distributions for different values of the penalty weight.

**step07-inversion-displacement.h5** This HDF5/Xdmf pair of files may be used to visualize the predicted cGPS displacements for each solution.

**step07-inversion-summary.txt** This file provides a summary of the inversion results for each value of the penalty weight.

One approach to finding the optimal penalty weight is to find the corner of the 'L-curve' for the log of the weighted data residual versus the log of the penalty residual. This is viewed as the point of diminishing returns for reducing the penalty weight. Further reductions provide little improvement to the weighted data residual, while providing a solution with less regularization. Figure 7.90 shows that this procedure suggests an optimal penalty weight of 0.1 for our inversion.

Figure 7.91 shows the predicted slip, the observed and predicted displacement vectors, and the slip applied from example step06 for a penalty weight of 1.0. The data fit is very good, and the predicted slip distribution is very close to the applied slip, although the magnitude is slightly underestimated.

#### 7.18.11.1 Exercises

- Investigate the effects of data noise.
  - How do the noisy data vectors compare to the raw data vectors from example step06?

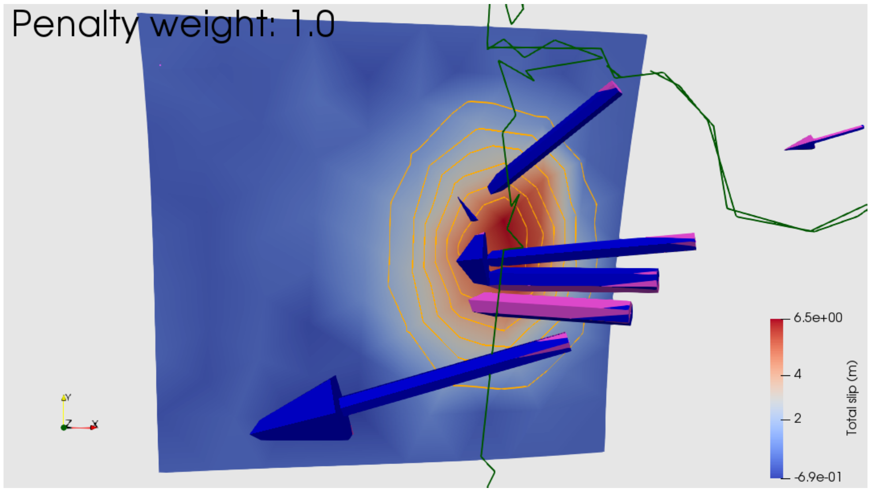



Figure 7.91: ParaView image of the inversion solution for a penalty weight of 1.0. 'Data' is shown with blue arrows and predicted displacements are shown with magenta arrows. Color contours represent the predicted slip distribution and orange line contours show the applied slip from the forward problem.

  - Create a new simulated dataset with more noise and see how well the solution matches the applied slip.
- Different initial slip distribution.
  - Move the slip distribution to a different location, vary the amplitude, etc. This will involve running another instance of example step06 to create a new dataset. How is the solution affected?
  - Move the slip onto the splay fault. This will involve creating a new forward model as well as generating Green's functions for the splay fault.
- What happens if your material properties are incorrect?
  - Try creating your forward model with heterogeneous properties and your Green's functions with homogeneous properties (or vice-versa). What happens to your solution?
- Try inverting for slip at various time steps.
- Try a different inversion method.
  - If you analyze the predicted slip distribution you will find some negative slip, which is unrealistic. To overcome this problem you could try NNLS (non-negative least squares). If you have the Python scipy package installed on your computer, you could replace the generalized inverse solution with the NNLS package included in `scipy.optimize.nnls`.

### 7.18.12 Step 8: Stress Field Due to Gravitational Body Forces

This example demonstrates the use of gravitational body forces as well as the use of initial stresses to balance the body forces. This involves enabling gravity within our domain with Dirichlet roller boundary conditions on the lateral and bottom boundaries; we do not include faults in this example. We also demonstrate what happens when the initial stresses are not in balance with the gravitational stresses, and show how viscoelastic problems with gravitational stresses will in general not reach a steady-state solution. The example is divided into three sub-problems:

**Step 8a** Gravitational body forces with 3-D density variations in elastic materials and initial stresses for a uniform density.

**Step 8b** Gravitational body forces with 3-D density variations in elastic materials and initial stresses from Step 8a (initial stresses satisfy equilibrium, so there is almost no deformation).

**Step 8c** Gravitational body forces with 3-D density variations in elastic and viscoelastic materials and initial stresses from Step 8a plus finite strain formulation (does not reach a steady-state solution).

#### 7.18.12.1 Step 08a

For Step 8a we apply gravitational stresses and attempt to balance these with analytically computed stresses consistent with the density of the mantle. Since the actual density is not uniform, the stresses are out of balance and we end up with some deformation. In `step08a.cfg` we turn on gravity and set the total time to zero (there is no time dependence in this model).

Excerpt from `step08a.cfg`

```
[pylithapp.problem]
# Set gravity field (default is None).
gravity_field = spatialdata.spatialdb.GravityField

[pylithapp.problem.formulation.time_step]
# Define the total time for the simulation.
total_time = 0.0*year
```

Our initial stress field corresponds to $\sigma_{xx} = \sigma_{yy} = \sigma_{zz} = \rho_{mantle} g z$ for all four materials, where $\rho_{mantle}$ is the density of the mantle, $g$ is the acceleration due to gravity, and $z$ is elevation. With only two control points necessary to describe this linear variation, we use the same SimpleDB spatial database for all four materials.

Excerpt from `step08a.cfg`

```
# We specify initial stresses for each material via a SimpleDB and linear interpolation.
[pylithapp.problem.materials.slab]
db_initial_stress = spatialdata.spatialdb.SimpleDB
db_initial_stress.label = Initial stress in the slab
db_initial_stress.iohandler.filename = spatialdb/mat_initial_stress_grav.spatialdb
db_initial_stress.query_type = linear

[pylithapp.problem.materials.wedge]
db_initial_stress = spatialdata.spatialdb.SimpleDB
db_initial_stress.label = Initial stress in the wedge
db_initial_stress.iohandler.filename = spatialdb/mat_initial_stress_grav.spatialdb
db_initial_stress.query_type = linear

[pylithapp.problem.materials.mantle]
db_initial_stress = spatialdata.spatialdb.SimpleDB
db_initial_stress.label = Initial stress in the mantle
db_initial_stress.iohandler.filename = spatialdb/mat_initial_stress_grav.spatialdb
db_initial_stress.query_type = linear

[pylithapp.problem.materials.crust]
db_initial_stress = spatialdata.spatialdb.SimpleDB
db_initial_stress.label = Initial stress in the crust
db_initial_stress.iohandler.filename = spatialdb/mat_initial_stress_grav.spatialdb
db_initial_stress.query_type = linear
```

Run Step 8a simulation

```
$ pylith step08a.cfg mat_elastic.cfg solver_algebraicmultigrid.cfg
```

The simulation will generate ten pairs of HDF5/Xdmf files beginning with `step08a`:

**step08a-domain.h5[.xmf]** Time series of the solution field over the domain.

**step08a-groundsurf.h5[.xmf]** Time series of the solution field over the ground surface.

**step08a-slab_info.h5[.xmf]** Properties for the slab material.

**step08a-slab.h5[.xmf]** Time series of the state variables (stress and strain) for the slab material.

**step08a-wedge_info.h5[.xmf]** Properties for the wedge material.

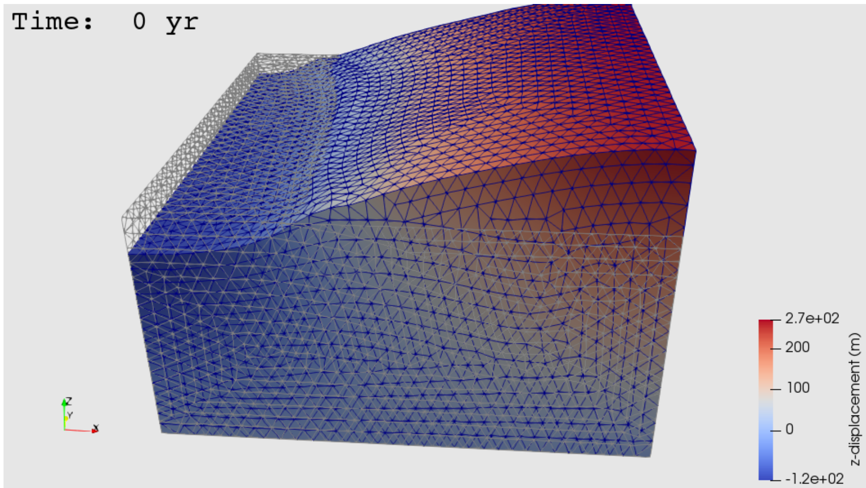



Figure 7.92: Solution for Step 8a. The deformation has been exaggerated by a factor of 500 and the colors highlight the vertical displacement component. The crustal material in the east is less dense than the assumed mantle material for initial stresses, while the slab material in the west is more dense. The result is uplift in the east and subsidence in the west.

**step08a-wedge.h5[.xmf]** Time series of the state variables (stress and strain) for the wedge material.

**step08a-crust_info.h5[.xmf]** Properties for the crust material.

**step08a-crust.h5[.xmf]** Time series of the tate variables (stress and strain) for the crust material.

**step08a-mantle_info.h5[.xmf]** Properties for the mantle material.

**step08a-mantle.h5[.xmf]** Time series of the state variables (stress and strain) for the mantle material.

When the problem has run, we see deformation that is consistent with the mismatched densities. The slab subsides while the crust undergoes uplift due to the differences in density relative to the mantle. Figure 7.92 shows the deformed mesh visualized with the `plot_dispwarp.py` ParaView Python script.

#### 7.18.12.2 Step 8b

Step 8b is similar to Step 8a, but we use the stresses output from Step 8a as the initial stress rather than analytically computing initial stresses. Because the initial stresses are consistent with the variations in density across the materials, the initial stresses will satisfy equilibrium and there will be essentially no deformation; the initial stresses do not perfectly balance because in Step 8a we average the values over the quadrature points for the output. We use the Python script `generate_initial_stress.py`, located in the `spatialdb` directory, to postprocess the output from Step 8a and generate the initial stress spatial database. Note that this script uses the Python interface to the spatialdata package to write the spatial database; this is much easier than writing a script to format the data to conform to the format of the spatial database. The spatial database will contain the stresses at each cell of our unstructured mesh, so the points are not on a logical grid, and we must use a SimpleDB.

Generate the initial stresses for Step 8b

```
# From the examples/3d/subduction directory, change to the spatialdb subdirectory.
$ cd spatialdb
$ ./generate_initial_stress.py
```

This will create spatial databases containing initial stresses for each of the four materials.

In the `step08b.cfg` file we specify the SimpleDB spatial database for each material (they are now material specific). With points at each cell centroid, we use nearest interpolation (default) rather than linear interpolation; this is a small approximation but it is much faster than using linear interpolation in this unstructured set of points.

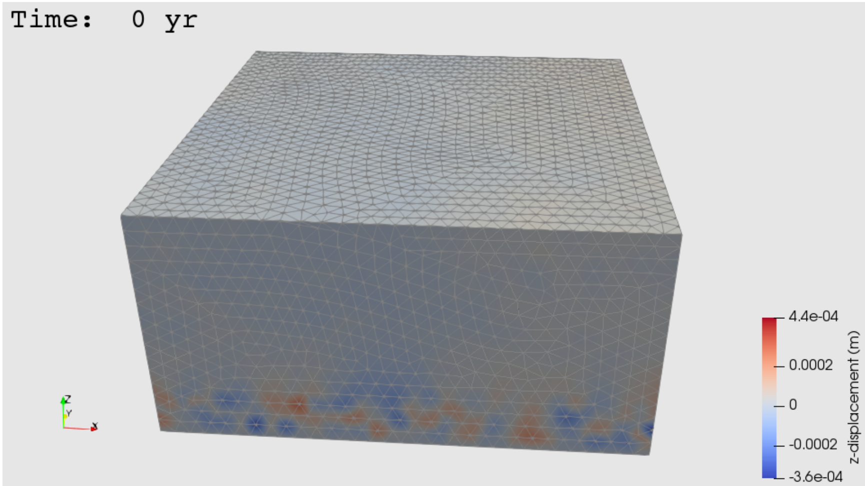



Figure 7.93: Solution for Step 8b. In this case the initial stresses satisfy the governing equation, so there is no deformation.

Excerpt from `step08b.cfg`

```
[pylithapp.problem.materials.slab]
db_initial_stress = spatialdata.spatialdb.SimpleDB
db_initial_stress.label = Initial stress in the slab
db_initial_stress.iohandler.filename = spatialdb/mat_initial_stress_grav-slab.spatialdb

[pylithapp.problem.materials.wedge]
db_initial_stress = spatialdata.spatialdb.SimpleDB
db_initial_stress.label = Initial stress in the wedge
db_initial_stress.iohandler.filename = spatialdb/mat_initial_stress_grav-wedge.spatialdb

[pylithapp.problem.materials.mantle]
db_initial_stress = spatialdata.spatialdb.SimpleDB
db_initial_stress.label = Initial stress in the mantle
db_initial_stress.iohandler.filename = spatialdb/mat_initial_stress_grav-mantle.spatialdb

[pylithapp.problem.materials.crust]
db_initial_stress = spatialdata.spatialdb.SimpleDB
db_initial_stress.label = Initial stress in the crust
db_initial_stress.iohandler.filename =
spatialdb/mat_initial_stress_grav-crust.spatialdb
```

Run Step 8b simulation

```
$ pylith step08b.cfg mat_elastic.cfg solver_algebraicmultigrid.cfg
```

This simulation will produce files in the `output` directory analogous to Step 8a.

When we compare the resulting elastic displacements with those of Step 8a, we find that there is essentially no displacement, as seen in Figure 7.93.

#### 7.18.12.3 Step 8c

In this example we use linear Maxwell viscoelastic models in place of the elastic models for the slab and mantle. We also use the small strain formulation (ImplicitLgDeform) so that the deformed configuration is taken into account; Steps 8a and 8b use the default Implicit infinitesimal strain formulation. The small strain formulation should generally be used for viscoelastic problems with gravity where you need accurate estimates of the vertical deformation.

**Warning**

The shear stress variations in the initial stresses will cause the viscoelastic materials to drive viscous flow, resulting in time-dependent deformation. As long as the elastic materials impose deviatoric stresses in the viscoelastic materials through continuity of strain, the viscoelastic materials will continue to flow. **As a result, in this case and many other simulations with viscoelastic materials and gravitational body forces, it is difficult to find a steady state solution.**

The only difference between the parameters in `step08b.cfg` and `step08c.cfg` is in the formulation setting and the simulation time:

Excerpt from `step08c.cfg`

```
[pylithapp.timedependent]
# Turn on the small strain formulation, which automatically runs the
# simulation as a nonlinear problem.
formulation = pylith.problems.ImplicitLgDeform

# Set gravity field (default is None).
gravity_field = spatialdata.spatialdb.GravityField

[pylithapp.problem.formulation.time_step]
# Define the total time for the simulation and the time step size.
total_time = 100.0*year
dt = 10.0*year
```

We use the material settings in `mat_viscoelastic.cfg`.

Run Step 8c simulation

```
$ pylith step08c.cfg mat_viscoelastic.cfg solver_algebraicmultigrid.cfg
```

This simulation will produce files in the `output` directory analogous to Steps 8a and 8b.

The resulting deformation is shown in Figure 7.94. As a result of viscous flow, the vertical deformation is even larger than that for Step 8a. If we were to run the simulation for a longer time period, the amount of vertical deformation would continue to increase.

#### 7.18.12.4 Exercises

- What happens in sub-problem step08a if we use a different reference density to compute our initial stresses?
- For sub-problem step08b, what happens if, for one of the materials you use the initial stresses from sub-problem step08a?
- For sub-problem step08c, what happens if you:
  - Run the simulation for a longer period of time?
  - Change the viscoelastic properties? For example, reduce the viscosity, make all materials viscoelastic, switch to a power-law rheology, etc.
- Is it possible to find a better initial stress state for sub-problem step08c?
  - What if the initial stresses were computed with nearly incompressible materials, and all materials in the model are viscoelastic?

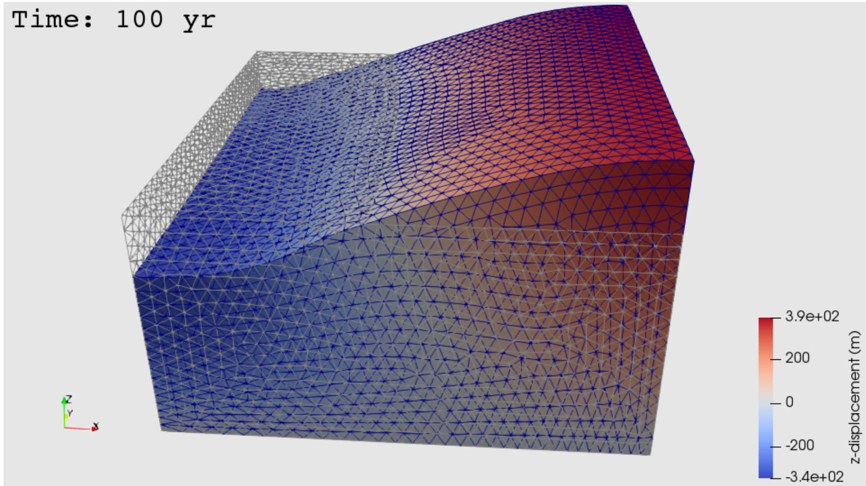



Figure 7.94: Image generated by running the `plot_dispwarp.py` script for sub-problem step08c. Although the stresses balance in the elastic solution, viscous flow in subsequent time steps results in large vertical deformation.

## 7.19 Additional Examples

### 7.19.1 CUBIT Meshing Examples

The directory `examples/meshing` contains several examples of using CUBIT to construct finite-element meshes for complex geometry. This includes features such as constructing nonplanar fault geometry from contours, constructing topography from a DEM, and merging sheet bodies (surfaces). A separate examples discusses defining the discretization size using a vertex field in an Exodus-II file. See the `README` files in the subdirectories for more detailed descriptions of these examples.

### 7.19.2 Debugging Examples

The directory `examples/debugging` contains a few examples to practice debugging a variety of user errors in parameters files and problem setup. The files with the errors corrected are in `examples/debugging/correct`. Step-by-step corrections are discussed in the Debugging PyLith Simulations sessions of the 2014 and 2015 PyLith tutorials (`wiki.geodynamics.org/software:pylith:start`).

### 7.19.3 Code Verification Benchmarks

The CIG GitHub software repository `https://github.com/geodynamics/pylith_benchmarks` contains input files for a number of community benchmarks. The benchmarks do not include the mesh files because they are so large; instead they include the CUBIT journal files that can be used to generate the meshes. Most, but not all, of the input files in the repository are updated for PyLith v2.0.0, so you will need to modify them if you use another version of PyLith.

# Chapter 8

# Benchmarks

## 8.1 Overview

The Crustal Deformation Modeling and Earthquake Source Physics Focus Groups within the Southern California Earthquake Center and the Short-Term Tectonics Working Group within CIG have developed a suite of benchmarks to test the accuracy and performance of 3D numerical codes for quasi-static crustal deformation and earthquake rupture dynamics. The benchmark definitions for the quasi-static crustal deformation benchmarks are posted on the CIG website at Short-Term Tectonics Benchmarks geodynamics.org/cig/workinggroups/short/workarea/benchmarks/ and the definitions for the earthquake rupture benchmarks are posted on the SCEC website scecdata.usc.edu/cvws/cgi-bin/cvws.cgi. This suite of benchmarks permits evaluating the relative performance of different types of basis functions, quadrature schemes, and discretizations for geophysical applications. The files needed to run the 3D benchmarks are in the CIG GitHub Repository https://github.com/geodynamics/pylith_benchmarks. In addition to evaluating the efficiency and accuracy of numerical codes, the benchmarks also make good test problems, where users can perform simulations based on actual geophysical problems. The benchmarks are performed at various resolutions and using different element types. By comparing the runtime and accuracy for different resolutions and element types, users can evaluate which combination will be best for their problems of interest.

## 8.2 Strike-Slip Benchmark

This benchmark problem computes the viscoelastic (Maxwell) relaxation of stresses from a single, finite, strike-slip earthquake in 3D without gravity. Dirichlet boundary conditions equal to the analytical elastic solution are imposed on the sides of a cube with sides of length 24 km. Anti-plane strain boundary conditions are imposed at y = 0, so the solution is equivalent to that for a domain with a 48 km length in the y direction. We can use the analytical solution of [Okada, 1992] both to apply the boundary conditions and to compare against the numerically-computed elastic solution.

### 8.2.1 Problem Description

Figure 8.1 on the following page shows the geometry of the strike-slip fault (red surface) embedded in the cube consisting of an elastic material (yellow block) over a Maxwell viscoelastic material (blue block).

**Domain** The domain spans the region

$$0 \leq x \leq 24\ km,$$
$$0 \leq y \leq 24\ km,$$
$$-24\ km \leq z \leq 0.$$

The top (elastic) layer occupies the region $-12\ km \leq z \leq 0$ and the bottom (viscoelastic) layer occupies the region $-24\ km \leq z \leq -12\ km$.

**Material properties** The material is a Poisson solid with a shear modulus of 30 GPa. The domain is modeled using an elastic isotropic material for the top layer and a Maxwell viscoelastic material for the bottom layer. The bottom layer has a viscosity of 1.0e+18 Pa-s.

**Fault** The fault is a vertical, right-lateral strike-slip fault. The strike is parallel to the y-direction at the center of the model:

$$x = 12\ km,$$
$$0 \leq y \leq 16\ km,$$
$$-16\ km \leq z \leq 0.$$

Uniform slip of 1 m is applied over the region $0 \leq y \leq 12\ km$ and $-12\ km \leq z \leq 0$ with a linear taper to 0 at y = 16 km and z = -16 km. The tapered region is the light red portion of the fault surface in Figure 8.1. In the region where the two tapers overlap, each slip value is the minimum of the two tapers (so that the taper remains linear).

**Boundary conditions** Bottom and side displacements are set to the elastic analytical solution, and the top of the model is a free surface. There are two exceptions to these applied boundary conditions. The first is on the y=0 plane, where y-displacements are left free to preserve symmetry, and the x- and z-displacements are set to zero. The second is along the line segment between (12, 0, -24) and (12, 24, -24), where the analytical solution blows up in some cases. Along this line segment, all three displacement components are left free.

**Discretization** The model is discretized with nominal spatial resolutions of 1000 m, 500 m, and 250 m.

**Basis functions** We use trilinear hexahedral cells and linear tetrahedral cells.

**Solution** We compute the error in the elastic solution and compare the solution over the domain after 0, 1, 5, and 10 years.

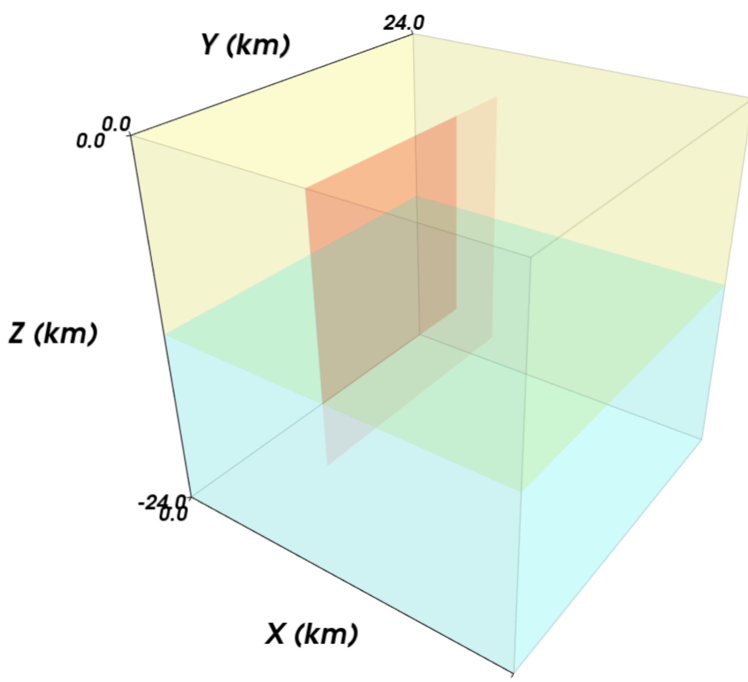



Figure 8.1: Geometry of strike-slip benchmark problem.

### 8.2.2 Running the Benchmark

Each benchmark uses three `.cfg` files in the parameters directory: `pylithapp.cfg`, a mesher related file (`strikeslip_cubit.cfg` or `strikeslip_lagrit.cfg`), and a resolution and cell related file (e.g., `strikeslip_hex8_1000m.cfg`).

```
# Checkout the benchmark files from the CIG Git repository.
$ git clone https://github.com/geodynamics/pylith_benchmarks.git
# Change to the quasistatic/sceccrustdeform/strikeslip directory.
$ cd quasistatic/sceccrustdeform/strikeslip
# Decompress the gzipped files in the meshes and parameters
directories.
$ gunzip meshes/*.gz parameters/*.gz
# Change to the parameters directory.
$ cd parameters
# Examples of running static (elastic solution only) cases.
$ pylith strikeslip_cubit.cfg strikeslip_hex8_1000m.cfg
$ pylith strikeslip_cubit.cfg strikeslip_hex8_0500m.cfg
$ pylith strikeslip_cubit.cfg strikeslip_tet4_1000m.cfg
# Append timedep.cfg to run the time-dependent (viscoelastic cases).
$ pylith strikeslip_cubit.cfg strikeslip_hex8_1000m.cfg timedep.cfg
$ pylith strikeslip_cubit.cfg strikeslip_hex8_0500m.cfg timedep.cfg
$ pylith strikeslip_cubit.cfg strikeslip_tet4_1000m.cfg timedep.cfg
```

This will run the problem for 10 years, using a time-step size of 0.1 years, and results will be output for each year. The benchmarks at resolutions of 1000 m, 500 m, and 250 m require approximately 150 MB, 960 MB, and 8 GB, respectively.

### 8.2.3 Benchmark Results

Figure 8.2 shows the displacement field from the simulation with hexahedral cells using trilinear basis functions at a resolution of 1000 m. For each resolution and set of basis functions, we measure the accuracy by comparing the numerical solution against the semi-analytical Okada solution [Okada, 1992]. We also compare the accuracy and runtime across resolutions and different cell types. This provides practical information about what cell types and resolutions are required to achieve a given level of accuracy with the shortest runtime.

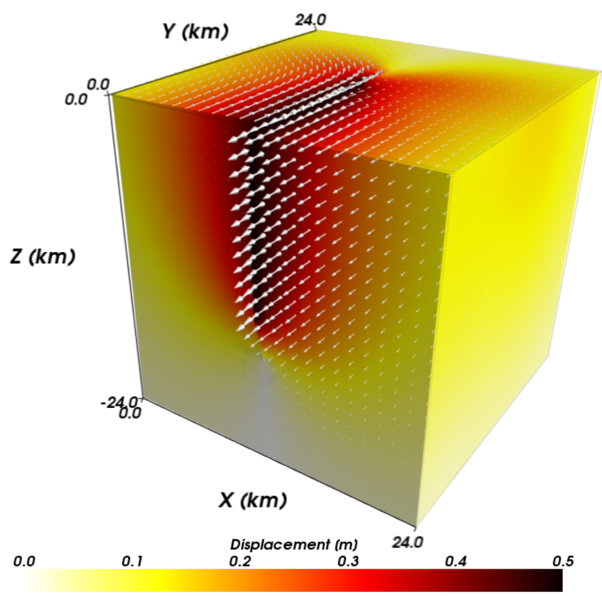



Figure 8.2: Displacement field for strike-slip benchmark problem.

#### 8.2.3.1 Solution Accuracy

We quantify the error in the finite-element solution by integrating the L2 norm of the difference between the finite-element solution and the semi-analytical solution evaluated at the quadrature points. We define the local error (error for each finite-element cell) to be

$$\epsilon_{local} = \frac{1}{V_{cell}} \sqrt{\int_{cell} \left(u_i^t - u_i^{fem}\right)^2 dV}, \tag{8.1}$$

where $u_i^t$ is the ith component of the displacement field for the semi-analytical solution, and $u_i^{fem}$ is the ith component of the displacement field for the finite-element solution. Taking the square root of the L2 norm and normalizing by the volume of the cell results in an error metric with dimensions of length. This roughly corresponds to the error in the magnitude of the displacement field in the finite element solution. We define the global error in a similar fashion,

$$\epsilon_{global} = \frac{1}{V_{domain}} \sqrt{\int_{domain} \left(u_i^t - u_i^{fem}\right)^2 dV}, \tag{8.2}$$

where we sum the L2 norm computed for the local error over all of the cells before taking the square root and dividing by the volume of the domain. CIG has developed a package called Cigma geodynamics.org/cig/software/packages/cs/cigma that computes these local and global error metrics.

Figures 8.3 through 8.8 on page 247 show the local error for each of the three resolutions and two cell types. The error decreases with decreasing cell size as expected for a converging solution. The largest errors, which approach 1 mm for 1 m of slip for a discretization size of 250 m, occur where the gradient in slip is discontinuous at the boundary between the region of uniform slip and linear taper in slip. The linear basis functions cannot match this higher order variation. The trilinear basis functions in the hexahedral element provide more terms in the polynomial defining the variation in the displacement field within each cell compared to the linear basis functions for the tetrahedral cell. Consequently, for this problem the error for the hexahedral cells at a given resolution is smaller than that for the tetrahedral cells. Both sets of cell types and basis functions provide the same rate of convergence as shown in Figure 8.9 on page 247.

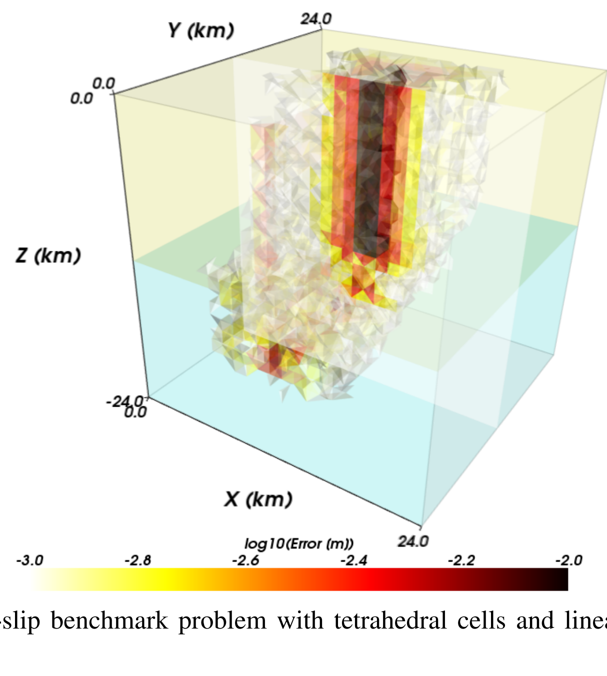



Figure 8.3: Local error for strike-slip benchmark problem with tetrahedral cells and linear basis functions with a uniform discretization size of 1000 m.

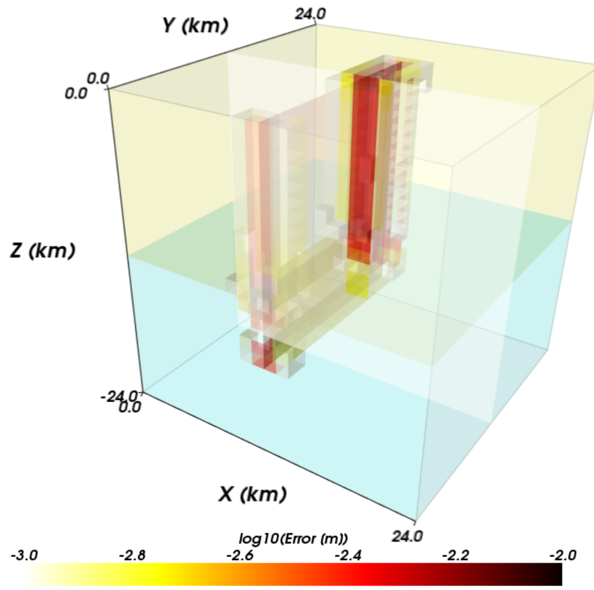



Figure 8.4: Local error for strike-slip benchmark problem with hexahedral cells and trilinear basis functions with a uniform discretization size of 1000 m.

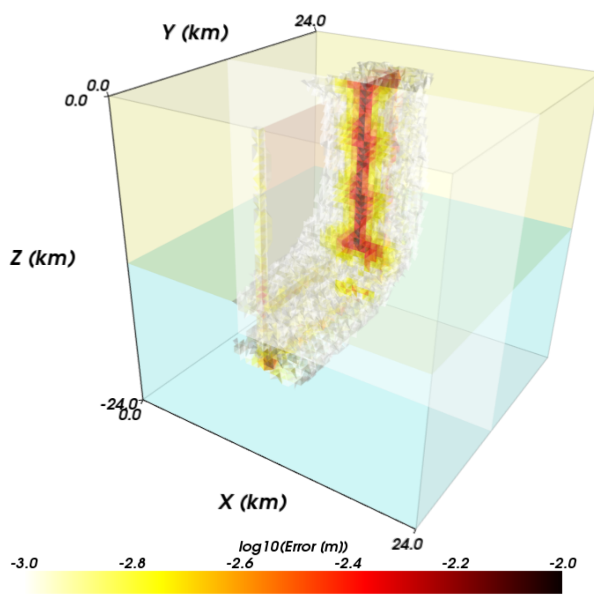



Figure 8.5: Local error for strike-slip benchmark problem with tetrahedral cells and linear basis functions with a uniform discretization size of 500 m.

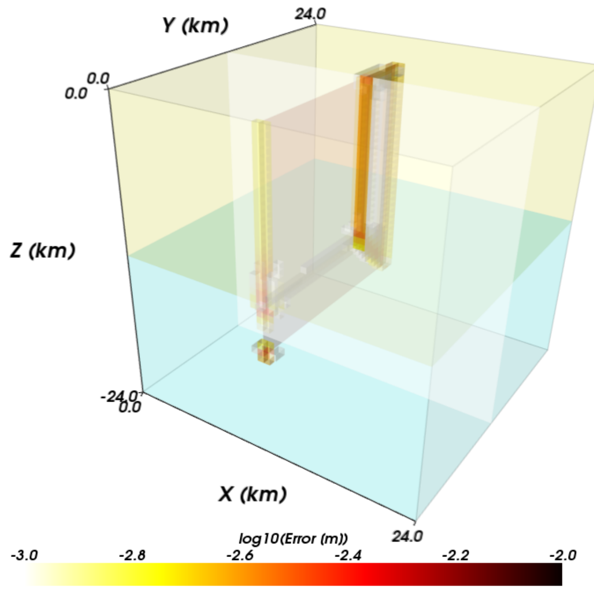



Figure 8.6: Local error for strike-slip benchmark problem with hexahedral cells and trilinear basis functions with a uniform discretization size of 500 m.

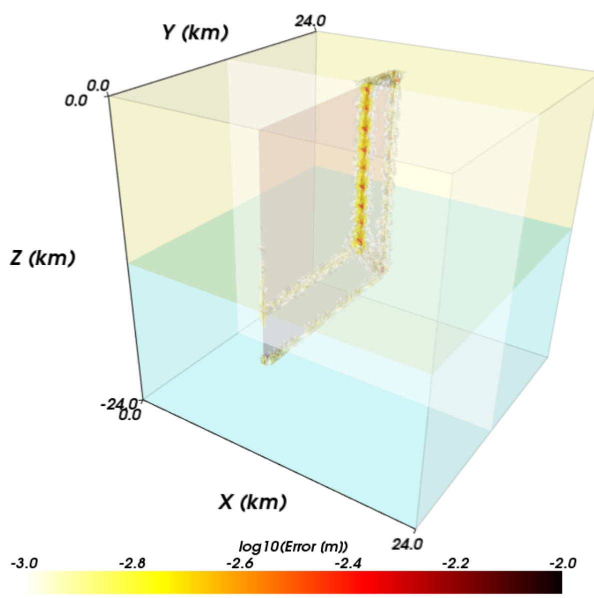



Figure 8.7: Local error for strike-slip benchmark problem with tetrahedral cells and linear basis functions with a uniform discretization size of 250 m.

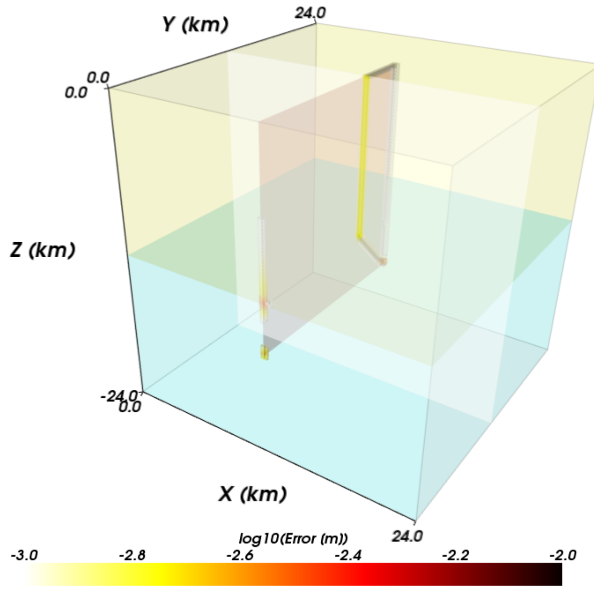



Figure 8.8: Local error for strike-slip benchmark problem with hexahedral cells and trilinear basis functions with a uniform discretization size of 250 m.

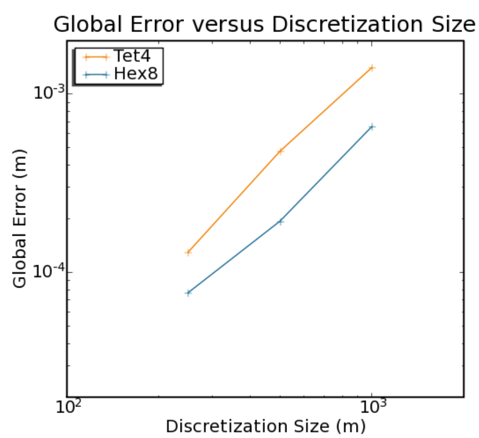



Figure 8.9: Convergence rate for the strike-slip benchmark problem with tetrahedral cells and linear basis functions and with hexahedral cells with trilinear basis functions.

#### 8.2.3.2 Performance

Figure 8.10 summarizes the overall performance of each of the six simulations. Although at a given resolution, the number of degrees of freedom in the hexahedral and tetrahedral meshes are the same, the number of cells in the tetrahedral mesh is about six times greater. However, we use only one integration point per tetrahedral cell compared to eight for the hexahedral cell. This leads to approximately the same number of integration points for the two meshes, but the time required to unpack/pack information for each cell from the finite-element data structure is greater than the time required to do the calculation for each quadrature point (which can take advantage of the very fast, small memory cache in the processor). As a result, the runtime for the simulations with hexahedral cells is significantly less than that for the tetrahedral cells at the same resolution.

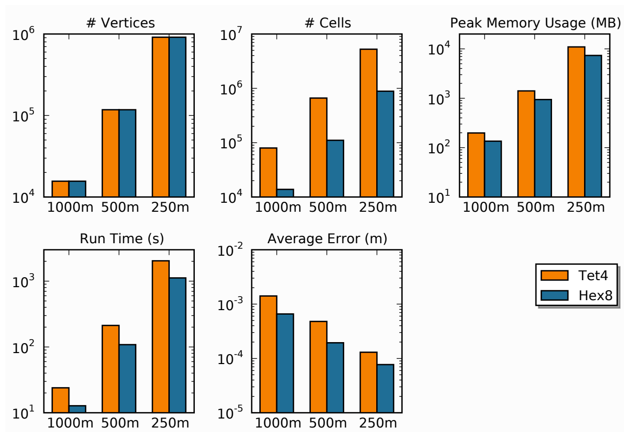



Figure 8.10: Summary of performance of PyLith for the six simulations of the strike-slip benchmark. For a given discretization size, hexahedral cells with trilinear basis functions provide greater accuracy with a shorter runtime compared with tetrahedral cells and linear basis functions.

Figure 8.11 on the next page compares the runtime for the benchmark (elastic solution only) at 500 m resolution for 1 to 16 processors. The total runtime is the time required for the entire simulation, including initialization, distributing the mesh over the processors, solving the problem in parallel, and writing the output to VTK files. Some initialization steps, writing the output to VTK files, and distributing the mesh are essentially serial processes. For simulations with many time steps these steps will generally occupy only a fraction of the runtime, and the runtime will be dominated by the solution of the equations. Figure 8.11 on the facing page also shows the total time required to form the Jacobian of the system, form the residual, and solve the system. These steps provide a more accurate representation of the parallel-performance of the computational portion of the code and show excellent performance as evident in the approximately linear slope of 0.7. S linear decrease with a slope of 1 would indicate strong scaling, which is rarely achieved in real applications.

## 8.3 Savage and Prescott Benchmark

This benchmark problem computes the viscoelastic (Maxwell) relaxation of stresses from repeated infinite, strike-slip earthquakes in 3D without gravity. The files needed to run the benchmark may be found at `https://github.com/geodynamics/pylith_benchmarks/tree/master/quasistatic/sceccrustdeform/savageprescott`. An analytical solution to this problem is described by Savage and Prescott [Savage and Prescott, 1978], which provides a simple way to check our numerical solution. A python utility code is provided in the utils directory to compute the analytical solution. Although this problem is actually 2.5D (infinite along-strike), we solve it using a 3D finite element model.

### 8.3.1 Problem Description

Figure 8.12 on page 250 shows the geometry of the problem, as described by [Savage and Prescott, 1978]. The analytical solution describes the surface deformation due to repeated earthquakes on an infinite strike-slip fault embedded in an elastic layer

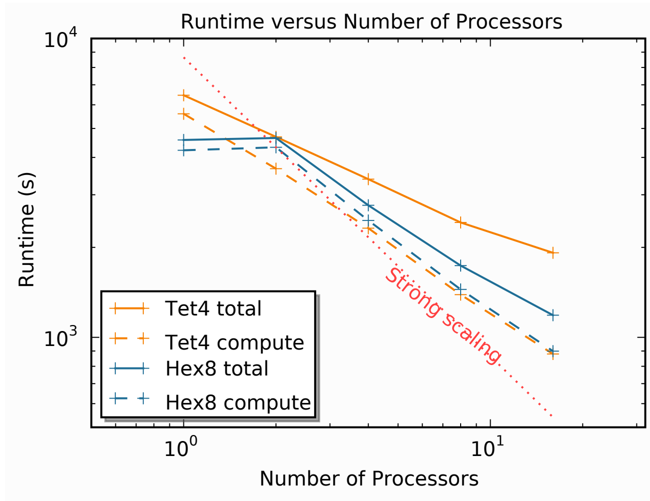



Figure 8.11: Parallel performance of PyLith for the strike-slip benchmark problem with tetrahedral cells and linear basis functions with a uniform discretization size of 500 m. The total runtime (total) and the runtime to compute the Jacobian and residual and solve the system (compute) are shown. The compute runtime decreases with a slope of about 0.7; a linear decrease with a slope of 1 would indicate strong scaling, which is rarely achieved in any real application.

overlying a Maxwell viscoelastic half-space. The upper portion of the fault (red in the figure) is locked between earthquakes, while the lower portion (blue in the figure) creeps at plate velocity. At regular recurrence intervals, the upper portion of the fault abruptly slips by an amount equal to the plate velocity multiplied by the recurrence interval, thus 'catching up' with the lower part of the fault.

There are some differences between the analytical solution and our numerical representation. First, the analytical solution represents the earthquake cycle as the superposition of uniform fault creep and an elementary earthquake cycle. Uniform fault creep is simply the uniform movement of the two plates past each other at plate velocity. For the elementary earthquake cycle, no slip occurs below the locked portion of the fault (blue portion in the figure). On the locked (red) portion of the fault, backslip equal to plate velocity occurs until the earthquake recurrence interval, at which point abrupt forward slip occurs. In the finite element solution, we perform the simulation as described in the figure. Velocity boundary conditions are applied at the extreme edges of the model to simulate block motion, steady creep is applied along the blue portion of the fault, and regular earthquakes are applied along the upper portion of the fault. It takes several earthquake cycles for the velocity boundary conditions to approximate the steady flow due to steady block motion, so we would not expect the analytical and numerical solutions to match until several earthquakes have occurred. Another difference lies in the dimensions of the domain. The analytical solution assumes an infinite strike-slip fault in an elastic layer overlying a Maxwell viscoelastic half-space. In our finite element model we are restricted to finite dimensions. We therefore extend the outer boundaries far enough from the region of interest to approximate boundaries at infinity.

Due to the difficulties in representing solutions in an infinite domain, there are several meshes that have been tested for this problem. The simplest meshes have uniform resolution (all cells have equal dimensions); however, such meshes typically do not provide accurate solutions since the resolution is too coarse in the region of interest. For that reason, we also tested meshes where the mesh resolution decreases away from the center. In the problem description that follows, we will focus on the hexahedral mesh with finer discretization near the fault (`meshes/hex8_6.7km.exo.gz`), which provides a good match with the analytical solution. It will first be necessary to gunzip this mesh so that it may be used by PyLith.

**Domain** The domain for this mesh spans the region

$$-1000 \leq x \leq 1000\ km,$$
$$-500 \leq y \leq 500\ km,$$
$$-400\ km \leq z \leq 0.$$

The top (elastic) layer occupies the region $-40\ km\ \leq z \leq 0$ and the bottom (viscoelastic) layer occupies the region $-400\ km\ \leq z \leq -40\ km$.

**Material properties** The material is a Poisson solid with a shear modulus ($\mu$) of 30 GPa. The domain is modeled using an elastic isotropic material for the top layer and a Maxwell viscoelastic material for the bottom layer. The bottom layer has a viscosity ($\eta$) of 2.36682e+19 Pa-s, yielding a relaxation time ($2\eta/\mu$) of 50 years.

**Fault** The fault is a vertical, left-lateral strike-slip fault. The strike is parallel to the y-direction at the center of the model:

$$x = 0 \ km,$$
$$-500 \leq y \leq 500 \ km,$$
$$-40 \ km \leq z \leq 0.$$

The locked portion of the fault (red section in Figure 8.12) extends from $-20 \, km \leq z \leq 0$, while the creeping section (blue) extends from $-40 \, km \leq z \leq 0$. Along the line where the two sections coincide ($z = -20 \, km$), half of the coseismic displacement and half of the steady creep is applied (see `finalslip.spatialdb` and `creeprate.spatialdb`).

**Boundary conditions** On the bottom boundary, vertical displacements are set to zero, while on the y-boundaries the x-displacements are set to zero. On the x-boundaries, the x-displacements are set to zero, while constant velocities of +/- 1 cm/yr are applied in the y-direction, giving a relative plate motion of 2 cm/year.

**Discretization** For the nonuniform hexahedral mesh, the resolution at the outer boundaries is 20 km. An inner region is then put through one level of refinement, so that near the center of the mesh the resolution is 6.7 km. All meshes were generated with CUBIT.

**Basis functions** We use trilinear hexahedral cells.

**Solution** We compute the surface displacements and compare these to the analytical solution in Figure 8.13 on the next page.

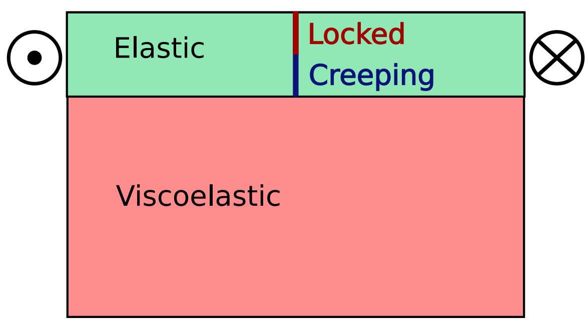



Figure 8.12: Problem description for the Savage and Prescott strike-slip benchmark problem.

### 8.3.2 Running the Benchmark

There are a number of `.cfg` files corresponding to the different meshes, as well as a `pylithapp.cfg` file defining parameters common to all problems. Each problem uses four `.cfg` files: `pylithapp.cfg`, `fieldsplit.cfg` (algebraic multigrid preconditioner), a cell-specific file (e.g., `hex8.cfg`), and a resolution specific file (e.g., hex8_6.7km.cfg).

```
# If you have not do so already, checkout the benchmarks from the CIG Git repository.
$ git clone https://github.com/geodynamics/pylith_benchmarks.git
# Change to the quasistatic/sceccrustdeform/savageprescott directory.
$ cd quasistatic/sceccrustdeform/savageprescott
# Decompress the gzipped files in the meshes directory.
$ gunzip meshes/*.gz
# Run one of the simulations.
$ pylith hex8.cfg hex8_6.7km.cfg fieldsplit.cfg
```

Each simulation uses 10 earthquake cycles of 200 years each, using a time-step size of 10 years, for a total simulation time of 2000 years. Ground surface output occurs every 10 years, while all other outputs occur every 50 years.

Once the problem has run, results will be placed in the `output` directory. These results may be viewed directly using 3-D visualization software such as ParaView; however, to compare results to the analytical solution, some postprocessing is required. First, generate the analytical results by running the `calc_analytic.py` script. This will produce files with displacements and velocities (`analytic_disp.txt` and `analytic_vel.txt`) in the `output` directory that are easy to use with a plotting package, such as matplotlib or Matlab.

### 8.3.3 Benchmark Results

Figure 8.13 shows the computed surface displacements for the 10th earthquake cycle compared with the analytical solution. The profile results were obtained as described above, and then all results (analytical and numerical) were referenced to the displacements immediately following the last earthquake. We find very good agreement between the analytical and numerical solutions, even for meshes with uniform refinement. We have not yet explored quantitative fits as a function of mesh resolution. For this benchmark, it is also important to consider the distance of the boundary from the region of interest. Also note that the agreement between analytical and numerical solutions is poor for early earthquake cycles, due to the differences in simulating the problem, as noted above.

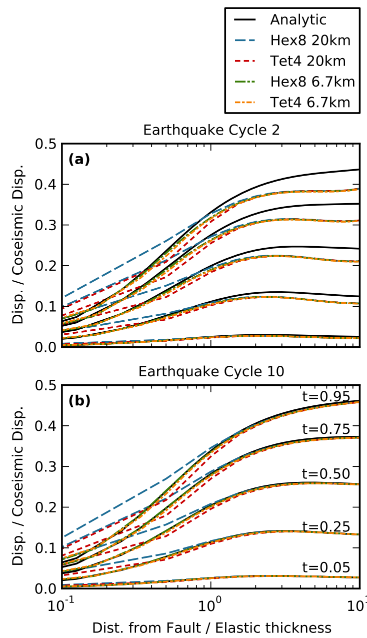



Figure 8.13: Displacement profiles perpendicular to the fault for a PyLith simulation with hex8 cells and the analytical solution for earthquake cycle 10.

## 8.4 SCEC Dynamic Rupture Benchmarks

The SCEC website `scecdata.usc.edu/cvws/cgi-bin/cvws.cgi` includes a graphical user interface for examining the benchmark results. Benchmark results for PyLith are available for TPV205-2D (horizontal slice through a vertical strike-slip fault), TPV205 (vertical strike-slip fault with high and low stress asperities), TPV210-2D (vertical slice through a 60-degree dipping normal fault), TPV210 (60-degree dipping normal fault), TPV11, TPV12, TPV13, TPV14-2D and TPV15-2D (horizontal slice through a verticel strike-slip fault with a branch), TPV14, TPV15, TPV 24, TPV25 (vertical strike-slip fault with a branch), TPV 16 and 17 (vertical strike-slip fault with spatially heterogeneous initial tractions), TPV 22 and 23 (vertical strike-slip fault with a stepover), TPV102 (vertical strike-slip fault with rate-state friction).

The benchmark results indicate that triangular and tetrahedral cells generate less numerical noise than quadrilateral or hexahedral cells. The input files in the repository are updated for PyLith v2.0.0, so you will need to modify them if you use another version of PyLith.

# Chapter 9

# Extending PyLith

One of the powerful features of using the Pyre framework in PyLith is the ability to extend the functionality of the software without altering any of the PyLith code. Any of the components can be replaced with other compatible components. You are already familiar with this feature from running the examples; when you set the spatial database to UniformDB, SimpleDB, or SCECCVMH you are switching between different compatible components for a spatial database facility. Modifying the governing equations to include other physical processes requires changing the data structures associated with the solution and altering the PyLith code.

In this section we provide examples of how to extend PyLith for components that users will most likely want to replace with their own custom versions. You will need a familiarity with Python, Makefiles, and C++ to write your own components. The primary steps in constructing a component to extend PyLith's functionality include:

1. Setting up the source files for the component or set of components based on the templates.
2. Edit the Python source file (`py`) for the component.
   (a) Define the user-specified properties and facilities.
   (b) Transfer the user-specified data from the Python object to the corresponding C++ object via calls to the SWIG interface object.
3. Edit the C++ header (`hh`) and implementation files (`cc`) for the component.
   (a) Implement the methods required to satisfy the interface definition of the component.
   (b) Implement the desired functionality of the component in C++.
4. Edit the SWIG interface files (`.i`) that provide the glue between Python and C++.
5. Edit the Python source file that tests the functionality of the component.
6. Run configure, build, install, and run the tests of the component.

## 9.1 Spatial Databases

PyLith provides several types of spatial databases that can be used for specification of parameters associated with boundary conditions, earthquake ruptures, and physical properties. In this example we demonstrate how to provide a spatial database, UniformVelModel, for specifying elastic properties. The source files are included in the source for the spatialdata package in the `templates/spatialdb` directory. The `README` file in `templates/spatialdb` provides detailed instructions for the various steps involved, and the source files contain numerous comments to help guide you through the customization process.

The UniformVelModel component provides uniform physical properties: P-wave speed, S-wave speed, and density. Although this is a rather trivial specification of physical properties that could be easily done using a UniformDB, this example demonstrates how to create a user-defined component that matches the requirements of a spatial database for elastic physical properties. Adding additional physical properties is simply a matter of including some additional values in the spatial database. Furthermore, in cases where we are constructing a spatial database for a seismic velocity model, the data points are georeferenced. With our uniform physical properties we do not need to worry about any differences in coordinate systems between our seismic velocity model and points at which the model is queried. However, in many cases we do, so we illustrate this functionality by using a geographic projection as the coordinate system in our example.

Using a top-down approach, the first step is to determine what information the user will need to supply to the component. Is the data for the spatial database in a file or a series of files? If so, file names and possible paths to a directory containing files with known names might be necessary. Are there other parameters that control the behavior of the component, such as a minimum shear wave speed? In our example the user supplies values for the P-wave speed, S-wave speed, and density. The user-supplied parameters become Pyre properties and facilities in the Python source file. Because our user supplied parameters are floating point values with dimensions, we create dimensional properties vs, vp, and density. In addition to defining the properties of the component, we also need to transfer these properties to the C++ object that does the real work. This is done by calling the C++ `vs()`, `vp()`, and `density()` accessor functions that are accessible via the Python module created by SWIG.

In the C++ object we must implement the functions that are required by the spatial database interface. These functions are listed near the beginning of the UniformVelModel class definition at the top of the C++ header file, `UniformVelModel.hh`. The C++ object also includes the accessor functions that allow us to set the P-wave speed, S-wave speed, and density values to the user-specified values in the Python object. Additional information, such as a file name, parameters defined as data structures, etc., would be set via similar accessor functions. You can also add additional functions and data structures to the C++ class to provide the necessary operations and functionality of the spatial database.

In SimpleDB we use a separate class to read in the spatial database and yet another class to perform the actual query. In our example, the C++ object also creates and stores the UTM zone 10 geographic projection for the seismic velocity model. When the spatial database gets a query for physical properties, we transform the coordinates of the query point from its coordinate system to the coordinate system of our seismic velocity model.

In order to use SWIG to create the Python module that allows us to call C++ from Python, we use a “main” SWIG interface file (`spatialdbcontrib.i` in this case) and then one for each object (`UniformVelModel.i` in this case). This greatly simplifies keeping the Python module synchronized with the C++ and Python code. The `UniformVelModel.i` SWIG file is nearly identical to the corresponding C++ header file. There are a few differences, as noted in the comments within the file. Copying and pasting the C++ header file and then doing a little cleanup is a very quick and easy way to construct a SWIG interface file for a C++ object. Because very little changes from SWIG module to SWIG module, it is usually easiest to construct the “main” SWIG interface by copying and customizing an existing one.

Once the Python, C++, and SWIG interface files are complete, we are ready to build the module. The `Makefile.am` file defines how to compile and link the C++ library and generate the Python module via SWIG. The `configure.ac` file contains the information used to build a configure script. The configure script checks to make sure it can find all of the tools needed to build the component (C++ compiler, Python, installed spatial database package, etc.). See the README file for detailed instructions on how to generate the configure script, and build and install the component.

We recommend constructing tests of the component to insure that it is functioning properly before attempting to use it in an application. The `tests` directory within `templates/spatialdb` contains a Python script, `testcontrib.py`, that runs the tests of the UniformVelModel component defined in `TestUniformVelModel.py`. Normally, one would want to test each function individually to isolate errors and create C++ tests as well as the Python tests included here. In our rather simple example, we simply test the overall functionality of the component. For examples of thorough testing, see the spatialdata and PyLith source code.

Once you have built, installed, and tested the UniformVelModel, it is time to use it in a simple example. Because the seismic velocity model uses georeferenced coordinates, our example must also use georeferenced coordinates. The dislocation example in the PyLith `examples/twocells/twotet4-geoproj` directory uses UTM zone 11 coordinates. The spatial database package will transform the coordinates between the two projections as defined in the UniformVelModel `query()` function. The dislocation example uses the SCEC CVM-H seismic velocity model. In order to replace the SCEC CVM-H seismic velocity with our uniform velocity model, in `pylithapp.cfg`

```
# Replace these two lines
db_properties = spatialdata.spatialdb.SCECCVMH
db_properties.data_dir = /home/john/data/sceccvm-h/vx53/bin
# with
db_properties = spatialdata.spatialdb.contrib.UniformVelModel
```

When you run the dislocation example, the `dislocation-statevars_info.vtk` file should reflect the use of physical properties from the uniform seismic velocity with $\mu = 1.69 \times 10^{10}$Pa, $\lambda = 1.6825 \times 10^{10}$Pa, and $\rho = 2500$kg/m$^3$.

## 9.2 Bulk Constitutive Models

PyLith includes several linearly elastic and inelastic bulk constitutive models for 2D and 3D problems. In this example, we demonstrate how to extend PyLith by adding your own bulk constitutive model. We reimplement the 2D plane strain constitutive model while adding the current strain and stress tensors as state variables. This constitutive model, PlaneStrainState, is not particularly useful, but it illustrates the basic steps involved in creating a bulk constitutive model with state variables. The source files are included with the main PyLith source code in the `templates/materials` directory. The `README` file in `templates/materials` provides detailed instructions for the various steps, and the source files contain numerous comments to guide you through the customization process.

In contrast to our previous example of creating a customized spatial database, which involved gathering user-specified parameters via the Pyre framework, there are no user-defined parameters for bulk constitutive models. The specification of the physical properties and state variables associated with the constitutive model is handled directly in the C++ code. As a result, the Python object for the constitutive model component is very simple, and customization is limited to simply changing the names of objects and labels.

The properties and state variables used in the bulk constitutive model are set using arguments to the constructor of the C++ ElasticMaterial object. We define a number of constants at the top of the C++ file and use the Metadata object to define the properties and state variables. The C++ object for the bulk constitutive component includes a number of functions that implement elasticity behavior of a bulk material as well as several utility routines:

**`_dbToProperties()`** Computes the physical properties used in the constitutive model equations from the physical properties supplied in spatial databases.

**`_nondimProperties()`** Nondimensionalizes the physical properties used in the constitutive model equations.

**`_dimProperties()`** Dimensionalizes the physical properties used in the constitutive model equations.

**`_stableTimeStepImplicit()`** Computes the stable time step for implicit time stepping in quasi-static simulations given the current state (strain, stress, and state variables).

**`_calcDensity()`** Computes the density given the physical properties and state variables. In most cases density is a physical property used in the constitutive equations, so this is a trivial function in those cases.

**`_calcStress()`** Computes the stress tensor given the physical properties, state variables, total strain, initial stress, and initial strain.

**`_calcElasticConsts()`** Computes the elastic constants given the physical properties, state variables, total strain, initial stress, and initial strain.

**`_updateStateVars()`** Updates the state variables given the physical properties, total strain, initial stress, and initial strain. If a constitutive model does not use state variables, then this routine is omitted.

Because it is sometimes convenient to supply physical properties for a bulk constitutive model that are equivalent but different from the ones that appear in the constitutive equations (e.g., P-wave speed, S-wave speed, and density rather then Lame's constants $\mu$, $\lambda$, and density), each bulk constitutive model component has routines to convert the physical property parameters and state variables a user specifies via spatial databases to the physical property properties and state variables used in the constitutive model equations. In quasi-static problems, PyLith computes an elastic solution for the first time step ($-\Delta t$ to $t$). This means that for inelastic materials, we supply two sets of functions for the constitutive behavior: one set for the initial elastic step and one set for the remainder of the simulation. See the source code for the inelastic materials in PyLith for an illustration of an efficient mechanism for doing this.

The SWIG interface files for a bulk constitutive component are set up in the same manner as in the previous example of creating a customized spatial database component. The "main" SWIG interface file (`materialscontrib.i` in this case) sets up the Python module, and the SWIG interface file for the component (`PlaneStrainState.i` in this case) defines the functions that should be included in the Python module. Note that because the C++ ElasticMaterial object defines a number of pure virtual methods (which merely specify the interface for the functions and do not implement default behavior), we must include many protected functions in the SWIG interface file. If these are omitted, SWIG will give a warning indicating that some of the functions remain abstract (i.e., some pure virtual functions defined in the parent class ElasticMaterial were not implemented in the child class PlaneStrainState), and no constructor is created. When this happens, you cannot create a PlaneStrainState Python object.

Once the Python, C++, and SWIG interface files are complete, you are ready to configure and build the C++ library and Python module for the component. Edit the `Makefile.am` file as necessary, then generate the configure script, run `configure`, and then build and install the library and module (see the `README` file for detailed instructions).

Because most functionality of the bulk constitutive model component is at the C++ level, properly constructed unit tests for the component should include tests for both the C++ code and Python code. The C++ unit tests can be quite complex, and it is best to examine those used for the bulk constitutive models included with PyLith. In this example we create the Python unit tests to verify that we can create a PlaneStrainState Python object and call some of the simple underlying C++ functions. The source files are in the `templates/materials/tests` directory. The `testcontrib.py` Python script runs the tests defined in `TestPlaneStrainState.py`.

Once you have built, installed, and tested the PlaneStrainState component, it is time to use it in a simple example. You can try using it in any of the 2D examples. For the examples in `examples/twocells/twoquad4`, in `pylithapp.cfg`

```
# Replace
material = pylith.materials.ElasticPlaneStrain
# with
material = pylith.materials.contrib.PlaneStrainState
```

or simply add the command line argument `--timedependent.homogeneous.material=pylith.materials.contrib.Plan` when running any of the 2D examples. The output should remain identical, but you should see the PlaneStrainState object listed in the information written to stdout.

## 9.3 Fault Constitutive Models

PyLith includes two of the most widely used fault constitutive models, but there are a wide range of models that have been proposed to explain earthquake source processes. In this example, we demonstrate how to extend PyLith by adding your own fault constitutive model. We implement a linear viscous fault constitutive model wherein the perturbation in the coefficient of friction is linearly proportional to the slip rate. This constitutive model, ViscousFriction, is not particularly useful, but it illustrates the basic steps involved in creating a fault constitutive model. The source files are included with the main PyLith source code in the `templates/friction` directory. The `README` file in `templates/friction` provides detailed instructions for the various steps, and the source files contain numerous comments to guide you through the customization process.

Similar to our previous example of creating a customized bulk constitutive model, the parameters are defined in the C++ code, not in the Pyre framework. As a result, the Python object for the fault constitutive model component is very simple and customization is limited to simply changing the names of objects and labels.

The properties and state variables used in the fault constitutive model are set using arguments to the constructor of the C++ FrictionModel object, analogous to the ElasticMaterial object for bulk constitutive models. In fact, both types of constitutive models used the same underlying C++ object (Metadata::ParamDescription) to store the description of the parameters and state variables. We define a number of constants at the top of the C++ file and use the Metadata object to define the properties and state variables. The C++ object for the fault constitutive component includes a number of functions that implement friction as well as several utility routines:

**`_dbToProperties()`** Computes the physical properties used in the constitutive model equations from the physical properties supplied in spatial databases.

**`_nondimProperties()`** Nondimensionalizes the physical properties used in the constitutive model equations.

**`_dimProperties()`** Dimensionalizes the physical properties used in the constitutive model equations.

**`_dbToStateVars()`** Computes the initial state variables used in the constitutive model equations from the initial values supplied in spatial databases.

**`_nondimStateVars()`** Nondimensionalizes the state variables used in the constitutive model equations.

**`_dimStateVars()`** Dimensionalizes the state variables used in the constitutive model equations.

**`_calcFriction()`** Computes the friction stress given the physical properties, state variables, slip, slip rate, and normal traction.

**`_updateStateVars()`** Updates the state variables given the physical properties, slip, slip rate, and normal traction.

If a constitutive model does not use state variables, then the state variable routines are omitted.

Because it is sometimes convenient to supply physical properties for a fault constitutive model that are equivalent but different from the ones that appear in the constitutive equations, each fault constitutive model component has routines to convert the physical property parameters and state variables a user specifies via spatial databases to the physical property properties and state variables used in the constitutive model equations.

The SWIG interface files for a fault constitutive component are set up in the same manner as in the previous examples of creating a bulk constitutive model or a customized spatial database component. The "main" SWIG interface file (`frictioncontrib.i` in this case) sets up the Python module, and the SWIG interface file for the component (`ViscousFriction.i` in this case) defines the functions that should be included in the Python module. Note that because the C++ FrictionModel object defines a number of pure virtual methods (which merely specify the interface for the functions and do not implement default behavior), we must include many protected functions in the SWIG interface file. If these are omitted, SWIG will give a warning indicating that some of the functions remain abstract (i.e., some pure virtual functions defined in the parent class FrictionModel were not implemented in the child class ViscousFriction), and no constructor is created. When this happens, you cannot create a ViscousFriction Python object.

Once the Python, C++, and SWIG interface files are complete, you are ready to configure and build the C++ library and Python module for the component. Edit the `Makefile.am` file as necessary, then generate the configure script, run `configure`, and then build and install the library and module (see the `README` file for detailed instructions).

Because most functionality of the fault constitutive model component is at the C++ level, properly constructed unit tests for the component should include tests for both the C++ code and Python code. The C++ unit tests can be quite complex, and it is best to examine those used for the fault constitutive models included with PyLith. In this example we create the Python unit tests to verify that we can create a ViscousFriction Python object and call some of the simple underlying C++ functions. The source files are in the `templates/friction/tests` directory. The `testcontrib.py` Python script runs the tests defined in `TestViscousFriction.py`.

Once you have built, installed, and tested the ViscousFriction component, it is time to use it in a simple example. You can try using it in any of the 2D or 3D examples. For the examples in `examples/bar_shearwave/quad4`, in `shearwave_staticfriction.cfg`

```
# Replace
friction = pylith.friction.StaticFriction
# with
friction = pylith.friction.contrib.ViscousFriction
```

or simply add the command line argument `--timedependent.interfaces.fault.friction=pylith.friction.contrib.` when running any of the friction examples. You will also need to supply a corresponding spatial database with the physical properties for the viscous friction constitutive model.

# Appendix A

# Glossary

## A.1 Pyre

**component** Basic building block of a Pyre application. A component may be built-up from smaller building blocks, where simple data types are called properties and data structures and objects are called facilities. In general a component is a specific implementation of the functionality of a facility. For example, SimpleDB is a specific implementation of the spatial database facility. A component is generally composed of a Python object and a C++ object, although either one may be missing. We nearly always use the naming convention such that for an object called Foo the Python object is in file Foo.py, the C++ class definition is in Foo.hh, inline C++ functions are in foo.icc, the C++ class implementation is in Foo.cc, and the SWIG interface file that glues the C++ and Python code together is in Foo.i.

**facility** Complex data type (object or data structure) building block of a component. See component.

**property** Simple data type (string, integer, real number, or boolean value) parameter for a component.

## A.2 DMPlex

The plex construction is a representation of the topology of the finite element mesh based upon a covering relation. For example, segments are covered by their endpoints, faces by their bounding edges, etc. Geometry is absent from the plex, and is represented instead by a field with the coordinates of the vertices. Meshes can also be understood as directed acyclic graphs, where we call the constituents points and arrows.

**mesh** A finite element mesh, used to partition space and provide support for the basis functions.

**cell** The highest dimensional elements of a mesh, or mesh entities of codimension zero.

**vertex** The zero dimensional mesh elements.

**face** Mesh elements that separate cells, or mesh entities of codimension one.

**field** A parallel section which can be completed, or made consistent, across process boundaries. These are used to represent continuum fields.

**section** These objects associate values in vectors to points (vertices, edges, faces, and cells) in a mesh. The section describes the offset into the vector along with the number of values associated with each point.

**dimension** The topological dimension of the mesh, meaning the cell dimension. It can also mean the dimension of the space in which the mesh is embedded, but this is properly the embedding dimension.

**fiber dimension** Dimension of the space associated with the field. For example, the scalar field has a fiber dimension of 1 and a vector field has a fiber displacement equal to the dimension of the mesh.

**cohesive cell** A zero volume cell inserted between any two cells which shared a fault face. They are prisms with a fault face as the base.

**cone** The set of entities which cover any entity in a mesh. For example, the cone of a triangle is its three edges.

**support** The set of mesh entities which are covered by any entity in a mesh. For example, the support of a triangle is the two tetrahedra it separates.

# Appendix B

# PyLith and Spatialdata Components

The name of the component is followed by the full path name and description. The full path name is used when setting a component to a facility in a `.cfg` file or with command line arguments.

## B.1 Application components

**PyLithApp** `pylith.apps.PyLithApp`
PyLith application.

### B.1.1 Problem Components

**TimeDependent** `pylith.problems.TimeDependent`
Time-dependent problem.

**GreensFns** `pylith.problems.GreensFns`
Static Green's function problem with slip impulses.

**Implicit** `pylith.problems.Implicit`
Implicit time stepping for static and quasi-static simulations with infinitesimal strains.

**ImplicitLgDeform** `pylith.problems.ImplicitLgDeform`
Implicit time stepping for static and quasi-static simulations including the effects of rigid body motion and small strains.

**Explicit** `pylith.problems.Explicit`
Explicit time stepping for dynamic simulations with a lumped system Jacobian matrix.

**ExplicitLgDeform** `pylith.problems.ExplicitLgDeform`
Explicit time stepping for dynamic simulations including the effects of rigid body motion and small strains with a lumped system Jacobian matrix.

**ExplicitTri3** `pylith.problems.ExplicitTri3`
Optimized elasticity formulation for linear triangular cells and one quadrature point for explicit time stepping in dynamic simulations.

**ExplicitTet4** `pylith.problems.ExplicitTet4`
Optimized elasticity formulation for linear tetrahedral cells and one quadrature point for explicit time stepping in dynamic simulations.

**SolverLinear** `pylith.problems.SolverLinear`
Linear PETSc solver (KSP).

**SolverNonlinear** `pylith.problems.SolverNonlinear`
Nonlinear PETSc solver (SNES).

**SolverLumped** `pylith.problems.SolverLumped`
Built-in simple, optimized solver for solving systems with a lumped Jacobian.

**TimeStepUniform** `pylith.problems.TimeStepUniform`
Uniform time stepping.

**TimeStepAdapt** `pylith.problems.TimeStepAdapt`
Adaptive time stepping (time step selected based on estimated stable time step).

**TimeStepUser** `pylith.problems.TimeStepUser`
User defined time stepping (variable time step set by user).

### B.1.2 Utility Components

**NullComponent** `pylith.utils.NullComponent`
Null component used to set a facility to an empty value.

**EventLogger** `pylith.utils.EventLogger`
PETSc event logger.

**VTKDataReader** `pylith.utils.VTKDataReader`
Data reader for VTK files, requires TVTK Enthought package available from `https://github.com/enthought/mayavi`.

### B.1.3 Topology Components

**Distributor** `pylith.topology.Distributor`
Distributor of mesh among processors in parallel simulations.

**JacobianViewer** `pylith.topology.JacobianViewer`
Viewer for writing Jacobian sparse matrix to file.

**MeshGenerator** `pylith.topology.MeshGenerator`
Mesh generator/importer.

**MeshImporter** `pylith.topology.MeshImporter`
Mesh importer/reader.

**MeshRefiner** `pylith.topology.MeshRefiner`
Default (null) mesh refinement object that does not refine the mesh.

**RefineUniform** `pylith.topology.RefineUniform`
Uniform global mesh refinement.

**ReverseCuthillMcKee** `pylith.topology.ReverseCuthillMcKee`
Object used to manage reordering cells and vertices using the reverse Cuthill-McKee algorithm.

### B.1.4 Material Components

**ElasticPlaneStrain** `pylith.materials.ElasticPlaneStrain`
Linearly elastic 2D bulk constitutive model with plane strain ($\epsilon_{zz} = 0$).

**ElasticPlaneStress** `pylith.materials.ElasticPlaneStress`
Linearly elastic 2D bulk constitutive model with plane stress ($\sigma_{zz} = 0$).

**ElasticIsotropic3D** `pylith.materials.ElasticIsotropic3D`
Linearly elastic 3D bulk constitutive model.

**MaxwellIsotropic3D** `pylith.materials.MaxwellIsotropic3D`
Linear Maxwell viscoelastic bulk constitutive model.

**MaxwellPlaneStrain** `pylith.materials.MaxwellPlaneStrain`
Linear Maxwell viscoelastic bulk constitutive model for plane strain problems.

**GenMaxwellIsotropic3D** `pylith.materials.GenMaxwellIsotropic3D`
Generalized Maxwell viscoelastic bulk constitutive model.

**GenMaxwellPlaneStrain** `pylith.materials.GenMaxwellPlaneStrain`
Generalized Maxwell viscoelastic bulk constitutive model for plane strain problems.

**PowerLaw3D** `pylith.materials.PowerLaw3D`
Power-law viscoelastic bulk constitutive model.

**PowerLawPlaneStrain** `pylith.materials.PowerLawPlaneStrain`
Power-law viscoelastic bulk constitutive model for plane strain problems.

**DruckerPrage3D** `pylith.materials.DruckerPrager3D`
Drucker-Prager elastoplastic bulk constitutive model.

**DruckerPragePlaneStrain** `pylith.materials.DruckerPragerPlaneStrain`
Drucker-Prager elastoplastic bulk constitutive model for plane strain problems.

**Homogeneous** `pylith.materials.Homogeneous`
Container with a single bulk material.

### B.1.5 Boundary Condition Components

**AbsorbingDampers** `pylith.bc.AbsorbingDampers`
Absorbing boundary condition using simple dashpots.

**DirichletBC** `pylith.bc.DirichletBC`
Dirichlet (prescribed displacements) boundary condition for a set of points.

**DirichletBoundary** `pylith.bc.DirichletBoundary`
Dirichlet (prescribed displacements) boundary condition for a set of points associated with a boundary surface.

**Neumann** `pylith.bc.Neumann`
Neumann (traction) boundary conditions applied to a boundary surface.

**PointForce** `pylith.bc.PointForce`
Point forces applied to a set of vertices.

**ZeroDispDB** `pylith.bc.ZeroDispDB`
Specialized UniformDB with uniform zero displacements at all degrees of freedom.

### B.1.6 Fault Components

**FaultCohesiveKin** `pylith.faults.FaultCohesiveKin`
Fault surface with kinematic (prescribed) slip implemented using cohesive elements.

**FaultCohesiveDyn** `pylith.faults.FaultCohesiveDyn`
Fault surface with dynamic (friction) slip implemented using cohesive elements.

**FaultCohesiveImpulses** `pylith.faults.FaultCohesiveImpulses`
Fault surface with Green's functions slip impulses implemented using cohesive elements.

**EqKinSrc** `pylith.faults.EqKinSrc`
Kinematic (prescribed) slip earthquake rupture.

**SingleRupture** `pylith.faults.SingleRupture`
Container with one kinematic earthquake rupture.

**StepSlipFn** `pylith.faults.StepSlipFn`
Step function slip-time function.

**ConstRateSlipFn** `pylith.faults.ConstRateSlipFn`
Constant slip rate slip-time function.

**BruneSlipFn** `pylith.faults.BruneSlipFn`
Slip-time function where slip rate is equal to Brune's far-field slip function.

**LiuCosSlipFn** `pylith.faults.LiuCosSlipFn`
Slip-time function composed of three sine/cosine functions. Similar to Brune's far-field time function but with more abrupt termination of slip.

**TimeHistorySlipFn** `pylith.faults.TimeHistorySlipFn`
Slip-time function with a user-defined slip time function.

**TractPerturbation** `pylith.faults.TractPerturbation`
Prescribed traction perturbation applied to fault with constitituve model in additional to tractions from deformation (generally used to nucleate a rupture).

### B.1.7 Friction Components

**StaticFriction** `pylith.friction.StaticFriction`
Static friction fault constitutive model.

**SlipWeakening** `pylith.friction.SlipWeakening`
Linear slip-weakening friction fault constitutive model.

**RateStateAgeing** `pylith.friction.RateStateAgeing`
Dieterich-Ruina rate and state friction with ageing law state variable evolution.

**TimeWeakening** `pylith.friction.TimeWeakening`
Linear time-weakening friction fault constitutive model.

### B.1.8 Discretization Components

**FIATLagrange** `pylith.feassemble.FIATLagrange`
Finite-element basis functions and quadrature rules for a Lagrange reference finite-element cell (point, line, quadrilateral, or hexahedron) using FIAT. The basis functions are constructed from the tensor produce of 1D Lagrange reference cells.

**FIATSimplex** `pylith.feassemble.FIATSimplex`
Finite-element basis functions and quadrature rules for a simplex finite-element cell (point, line, triangle, or tetrahedron) using FIAT.

### B.1.9 Output Components

**MeshIOAscii** `pylith.meshio.MeshIOAscii`
Reader for simple mesh ASCII files.

**MeshIOCubit** `pylith.meshio.MeshIOCubit`
Reader for CUBIT Exodus files.

**MeshIOLagrit** `pylith.meshio.MeshIOLagrit`
Reader for LaGriT GMV/Pset files.

**OutputManager** `pylith.meshio.OutputManager`
General output manager for mesh information and data.

**OutputSoln** `pylith.meshio.OutputSoln`
Output manager for solution data.

**OutputSolnSubset** `pylith.meshio.OutputSolnSubset`
Output manager for solution data over a submesh.

**OutputSolnPoints** `pylith.meshio.OutputSolnPoints`
Output manager for solution data at arbitrary points in the domain.

**OutputDirichlet** `pylith.meshio.OutputDirichlet`
Output manager for Dirichlet boundary condition information over a submesh.

**OutputNeumann** `pylith.meshio.OutputNeumann`
Output manager for Neumann boundary condition information over a submesh.

**OutputFaultKin** `pylith.meshio.OutputFaultKin`
Output manager for fault with kinematic (prescribed) earthquake ruptures.

**OutputFaultDyn** `pylith.meshio.OutputFaultDyn`
Output manager for fault with dynamic (friction) earthquake ruptures.

**OutputFaultImpulses** `pylith.meshio.OutputFaultImpulses`
Output manager for fault with static slip impulses.

**OutputMatElastic** `pylith.meshio.OutputMatElastic`
Output manager for bulk constitutive models for elasticity.

**SingleOutput** `pylith.meshio.SingleOutput`
Container with single output manger.

**PointsList** `pylith.meshio.PointsList`
Manager for text file container points for `OutputSolnPoints`.

**DataWriterVTK** `pylith.meshio.DataWriterVTK`
Writer for output to VTK files.

**DataWriterHDF5** `pylith.meshio.DataWriterHDF5`
Writer for output to HDF5 files.

**DataWriterHDF5Ext** `pylith.meshio.DataWriterHDF5Ext`
Writer for output to HDF5 files with datasets written to external raw binary files.

**CellFilterAvg** `pylith.meshio.CellFilterAvg`
Filter that averages information over quadrature points of cells.

**VertexFilterVecNorm** `pylith.meshio.VertexFilterVecNorm`
Filter that computes magnitude of vectors for vertex fields.

## B.2 Spatialdata Components

### B.2.1 Coordinate System Components

**CSCart** `spatialdata.geocoords.CSCart`
Cartesian coordinate system (1D, 2D, or 3D).

**CSGeo** `spatialdata.geocoords.CSGeo`
Geographic coordinate system.

**CSGeoProj** `spatialdata.geocoords.CSGeoProj`
Coordinate system associated with a geographic projection.

**CSGeoLocalCart** `spatialdata.geocoords.CSGeoLocalCart`
Local, georeferenced Cartesian coordinate system.

**Projector** `spatialdata.geocoords.Projector`
Geographic projection.

**Converter** `spatialdata.geocoords.Converter`
Converter for transforming coordinates of points from one coordinate system to another.

### B.2.2 Spatial database Components

**UniformDB** `spatialdata.spatialdb.UniformDB`
Spatial database with uniform values.

**SimpleDB** `spatialdata.spatialdb.SimpleDB`
Simple spatial database that defines fields using a point cloud. Values are determined using a nearest neighbor search or linear interpolation in 0D, 1D, 2D, or 3D.

**SimpleIOAscii** `spatialdata.spatialdb.SimpleIOAscii`
Reader/writer for simple spatial database files.

**SCECCVMH** `spatialdata.spatialdb.SCECCVMH`
Spatial database interface to the SCEC CVM-H (seismic velocity model).

**CompositeDB** `spatialdata.spatialdb.CompositeDB`
Spatial database composed from multiple other spatial databases.

**TimeHistory** `spatialdata.spatialdb.TimeHistory`
Time history for temporal variations of a parameter.

**GravityField** `spatialdata.spatialdb.GravityField`
Spatial database providing vector for body forces associated with gravity.

### B.2.3 Nondimensionalization components

**Nondimensional** `spatialdata.units.Nondimensional`
Nondimensionalization of length, time, and pressure scales.

**NondimensionalElasticDynamic** `spatialdata.units.NondimensionalElasticDynamic`
Nondimensionalization of scales for dynamic problems.

**NondimensionalElasticQuasistatic** `spatialdata.units.NondimensionalElasticQuasistatic`
Nondimensionalization of scales for quasi-static problems.

# Appendix C

# File Formats

## C.1 PyLith Mesh ASCII Files

PyLith mesh ASCII files allow quick specification of the mesh information for very small, simple meshes that are most easily written by hand. We do not recommend using this format for anything other than these very small, simple meshes.

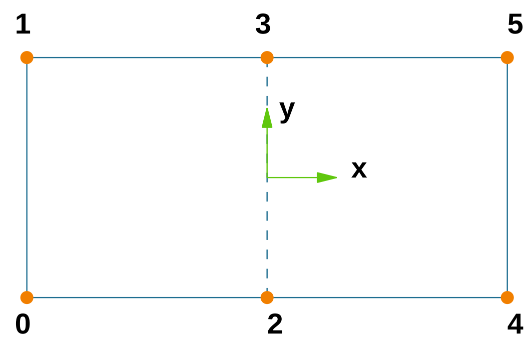



Figure C.1: Diagram of mesh specified in the MeshIOAscii file.

```
// This mesh file defines a finite-element mesh composed of two
// square cells of edge length 2.
//
// Comments can appear almost anywhere in these files and are
// delimited with two slashes (//) just like in C++. All text and
// whitespace after the delimiter on a given line is ignored.

mesh = { // begin specification of the mesh
  dimension = 2 // spatial dimension of the mesh

  // Begin vertex and cell labels with 0. This is the default so
  // this next line is optional
  use-index-zero = true

  vertices = { // vertices or nodes of the finite-element cells
    dimension = 2 // spatial dimension of the vertex coordinates
    count = 6 // number of vertices in the mesh
    coordinates = { // list of vertex index and coordinates
      // the coordinates must coincide with the coordinate
      // system specified in the Mesh object
      // exactly one vertex must appear on each line
      // (excluding whitespace)
      0    -2.0  -1.0
```

```
      1    -2.0  +1.0
      2     0.0  -1.0
      3     0.0  +1.0
      4    +2.0  -1.0
      5    +2.0  +1.0
    } // end of coordinates list
  } // end of vertices

  cells = { // finite-element cells
    count = 2 // number of cells in the mesh
    num-corners = 4 // number of vertices defining the cell
    simplices = { // list of vertices in each cell
      // see Section 4.2 for diagrams giving the order for each
      // type of cell supported in PyLith
      // index of cell precedes the list of vertices for the cell
      // exactly one cell must appear on each line
      // (excluding whitespace)
      0    0  2  3  1
      1    4  5  3  2
    } // end of simplices list

    material-ids = { // associated each cell with a material model
      // the material id is specified using the index of the cell
      // and then the corresponding material id
      0   0 // cell 0 has a material id of 0
      1   2 // cell 1 has a material id of 2
    } // end of material-ids list
  } // end of cells

  // This next section lists groups of vertices that can be used
  // in applying boundary conditions to portions of the domain
  group = { // start of a group
    // the name can have whitespace, so no comments are allowed
    // after the name
    name = face +y

    // Either groups of vertices or groups of cells can be
    // specified, but currently PyLith only makes use of groups
    // of vertices
    type = vertices // 'vertices' or 'cells'
    count = 2 // number of vertices in the group
    indices = { // list of vertex indices in the group
      // multiple vertices may appear on a line
      0  4 // this group contains vertices 0 and 4
    } // end of list of vertices
  } // end of group

  // additional groups can be listed here
```

## C.2 SimpleDB Spatial Database Files

SimpleDB spatial database files contain a header describing the set of points and then the data with each line listing the coordinates of a point followed by the values of the fields for that point.

```
// This spatial database specifies the distribution of slip on the
// fault surface. In this case we prescribe a piecewise linear,
// depth dependent distribution of slip. The slip is 2.0 m
// right-lateral with 0.25 m of reverse slip at the surface with
// a linear taper from 2.0 m to 0.0 m from -2 km to -4 km.
//
// Comments can appear almost anywhere in these files and are
```

```
// delimited with two slashes (///) just like in C++. All text and
// whitespace after the delimiter on a given line is ignored.
//
// The next line is the magic header for spatial database files
// in ASCII format.
#SPATIAL.ascii 1
SimpleDB { // start specifying the database parameters
  num-values = 3 // number of values in the database

  // Specify the names and the order of the values as they appear
  // in the data. The names of the values must correspond to the
  // names PyLith requests in querying the database.
  value-names =  left-lateral-slip  reverse-slip  fault-opening

  // Specify the units of the values in Python syntax (e.g., kg/m**3).
  value-units =  m  m  m
  num-locs = 3 // Number of locations where values are given
  data-dim = 1 // Locations of data points form a line
  space-dim = 3 // Spatial dimension in which data resides

  // Specify the coordinate system associated with the
  // coordinates of the locations where data is given
  cs-data = cartesian { // Use a Cartesian coordinate system
    to-meters = 1.0e+3 // Coordinates are in km

    // Specify the spatial dimension of the coordinate system
    // This value must match the one associated with the database
    space-dim = 3

  } // cs-data // end of coordinate system specification

} // end of SimpleDB parameters
// The locations and values are listed after the parameters.
// Columns are coordinates of the points (1 column for each
// spatial dimension) followed by the data values in the order
// specified by the value-names field.
0.0  0.0  0.0    -2.00  0.25  0.00
0.0  0.0 -2.0    -2.00  0.00  0.00
0.0  0.0 -4.0     0.00  0.00  0.00
```

### C.2.1 Spatial Database Coordinate Systems

The spatial database files support four different types of coordinate systems. Conversions among the three geographic coordinate systems are supported in 3D. Conversions among the geographic and geographic projected coordinate systems are supported in 2D. In using the coordinate systems, we assume that you have installed the `proj` program in addition to the Proj.4 libraries, so that you can obtain a list of support projections, datums, and ellipsoids. Alternatively, refer to the documentation for the Proj.4 Cartographic Projections library `trac.osgeo.org/proj`.

#### C.2.1.1 Cartesian

This is a conventional Cartesian coordinate system. Conversions to other Cartesian coordinate systems are possible.

```
cs-data = cartesian {
  to-meters = 1.0e+3 // Locations in km
  space-dim = 2 // 1, 2, or 3 dimensions
}
```

#### C.2.1.2 Geographic

This coordinate system is for geographic coordinates, such as longitude and latitude. Specification of the location in three-dimensions is supported. The vertical datum can be either the reference ellipsoid or mean sea level. The vertical coordinate is positive upwards.

```
cs-data = geographic {
  // Conversion factor to get to meters (only applies to vertical
  // coordinate unless you are using a geocentric coordinate system).
  to-meters = 1.0e+3
  space-dim = 2 // 2 or 3 dimensions

  // Run ''proj -le'' to see a list of available reference ellipsoids.
  // Comments are not allowed at the end of the next line.
  ellipsoid = WGS84

  // Run ''proj -ld'' to see a list of available datums.
  // Comments are not allowed at the end of the next line.
  datum-horiz = WGS84

  // ''ellipsoid'' or ''mean sea level''
  // Comments are not allowed at the end of the next line.
  datum-vert = ellipsoid

  // Use a geocentric coordinate system?
  is-geocentric = false // true or false
}
```

#### C.2.1.3 Geographic Projection

This coordinate system applies to geographic projections. As in the geographic coordinate system, the vertical coordinate (if used) can be specified with respect to either the reference ellipsoid or mean sea level. The coordinate system can use a local origin and be rotated about the vertical direction.

```
cs-data = geo-projected {
  to-meters = 1.0e+3 // Conversion factor to get to meters.
  space-dim = 2 // 2 or 3 dimensions

  // Run ''proj -le'' to see a list of available reference ellipsoids.
  // Comments are not allowed at the end of the next line.
  ellipsoid = WGS84

  // Run ''proj -ld'' to see a list of available datums.
  // Comments are not allowed at the end of the next line.
  datum-horiz = WGS84

  // ''ellipsoid'' or ''mean sea level''
  // Comments are not allowed at the end of the next line.
  datum-vert = ellipsoid

  // Longitude of local origin in WGS84.
  origin-lon = -120.0

  // Latitude of local origin in WGS84.
  origin-lat = 37.0

  // Rotation angle in degrees (CCW) or local x-axis from east.
  rotation-angle = 23.0

  // Run ''proj -lp'' to see a list of available geographic
  // projections.
    projector = projection {
```

```
    // Name of the projection. run ``proj -lp'' to see a list of
    // supported projections. Use the Universal Transverse Mercator
    // projection.
    projection = utm
    units = m // Units in the projection.

    // Provide a list of projection options; these are the command
    // line arguments you would use with the proj program. Refer to
    // the Proj.4 library documentation for complete details.
    // Comments are not allowed at the end of the next line.
    proj-options = +zone=10
}
```

#### C.2.1.4 Geographic Local Cartesian

This coordinate system is a geographically referenced, local 3D Cartesian coordinate system. This allows use of a conventional Cartesian coordinate system with accurate georeferencing. For example, one can construct a finite-element model in this coordinate system and use spatial databases in geographic coordinates. This coordinate system provides an alternative to using a geographic projection as the Cartesian grip. The advantage of this coordinate system is that it retains the curvature of the Earth, while a geographic projection does not.

```
cs-data = geo-local-cartesian {
  // Conversion factor to get to meters (only applies to vertical
  // coordinate unless you are using a geocentric coordinate system).
  to-meters = 1.0 // use meters
  space-dim = 2 // 2 or 3 dimensions

  // Run ``proj -le'' to see a list of available reference ellipsoids.
  // Comments are not allowed at the end of the next line.
  ellipsoid = WGS84

  // Run ``proj -ld'' to see a list of available datums.
  // Comments are not allowed at the end of the next line.
  datum-horiz = WGS84

  // ``ellipsoid'' or ``mean sea level''
  // Comments are not allowed at the end of the next line.
  datum-vert = ellipsoid

  // Origin of the local Cartesian coordinate system. To avoid
  // round-off errors it is best to pick a location near the center of
  // the region of interest. An elevation on the surface of the Earth
  // in the middle of the region also works well (and makes the
  // vertical coordinate easy to interpret).
  origin-lon = -116.7094 // Longitude of the origin in decimal degrees
                         // (west is negative).

  origin-lat = 36.3874 // Latitude of the origin in decimal degrees
                       // (north is positive).

  // Elevation with respect to the vertical datum. Units are the
  // same as the Cartesian coordinate system (in this case meters).
  origin-elev = 3.5
}
```

## C.3 SimpleGridDB Spatial Database Files

SimpleGridDB spatial database files contain a header describing the grid of points and then the data with each line listing the coordinates of a point followed by the values of the fields for that point. The coordinates for each dimension of the grid do not need to be uniformly spaced. The coordinate systems are specified the same way as they are in SimpleDB spatial database files as described in Section C.2 on page 268.

```
// This spatial database specifies the elastic properties on a
// 2-D grid in 3-D space.
//
// Comments can appear almost anywhere in these files and are
// delimited with two slashes (//) just like in C++. All text and
// whitespace after the delimiter on a given line is ignored.

// The next line is the magic header for spatial database files
// in ASCII format.
#SPATIAL_GRID.ascii 1
SimpleGridDB { // start specifying the database parameters
  num-values = 3 // number of values in the database

  // Specify the names and the order of the values as they appear
  // in the data. The names of the values must correspond to the
  // names PyLith requests in querying the database.
  value-names =  Vp  Vs  Density

  // Specify the units of the values in Python syntax.
  value-units =  km/s  km/s  kg/m{*}{*}3
  num-x = 3 // Number of locations along x coordinate direction
  num-y = 1 // Number of locations along y coordinate direction
  num-z = 2 // Number of locations along z coordinate direction
  space-dim = 3 // Spatial dimension in which data resides

  // Specify the coordinate system associated with the
  // coordinates of the locations where data is given
  cs-data = cartesian \{ // Use a Cartesian coordinate system
    to-meters = 1.0e+3 // Coordinates are in km

    // Specify the spatial dimension of the coordinate system
    // This value must match the one associated with the database
    space-dim = 3
  } // cs-data // end of coordinate system specification
} // end of SimpleGridDB specification
// x coordinates
-3.0  1.0  2.0
// y coordinates
8.0
// z coordinates
2.0  4.0

// The locations and values are listed after the parameters.
// Columns are coordinates of the points (1 column for each
// spatial dimension) followed by the data values in the order
// specified by the value-names field. The points can be in any order.
-3.0  8.0  2.0    6.0  4.0  2500.0
 1.0  8.0  2.0    6.2  4.1  2600.0
 2.0  8.0  2.0    5.8  3.9  2400.0
-3.0  8.0  4.0    6.1  4.1  2500.0
 1.0  8.0  4.0    5.9  3.8  2450.0
 2.0  8.0  4.0    5.7  3.7  2400.0
```

## C.4 TimeHistory Database Files

TimeHistory database files contain a header describing the number of points in the time history and the units for the time stamps followed by a list with pairs of time stamps and amplitude values. The amplitude at an arbitrary point in time is computed via interpolation of the values in the database. This means that the time history database must span the range of time values of interest. The points in the time history must also be ordered in time.

```
// This time history database specifies temporal variation in
// amplitude. In this case we prescribe a triangular slip time
// history.
//
// Comments can appear almost anywhere in these files and are
// delimited with two slashes (//) just like in C++. All text and
// whitespace after the delimiter on a given line is ignored.
//
// The next line is the magic header for spatial database files
// in ASCII format.
#TIME HISTORY ascii
TimeHistory { // start specifying the database parameters
  num-points = 5 // number of points in time history

  // Specify the units used in the time stamps.
  time-units = year
} // end of TimeHistory header
// The time history values are listed after the parameters.
// Columns time and amplitude where the amplitude values are unitless.
 0.0     0.00
 2.0     1.00
 6.0     4.00
10.0     2.00
11.0     0.00
```

## C.5 User-Specified Time-Step File

This file lists the time-step sizes for nonuniform, user-specified time steps associated with the TimeStepUser object. The file's format is an ASCII file that includes the units for the time-step sizes and then a list of the time steps.

```
// This time step file specifies five time steps with the units in years.
// Comments can appear almost anywhere in these files and are
// delimited with two slashes (//) just like in C++. All text and
// whitespace after the delimiter on a given line is ignored.
//
// Units for the time steps
units = year
1.0 // Comment
2.0
3.0
2.5
3.0
```

## C.6 PointsList File

This file lists the coordinates of the locations where output is requested for the `OutputSolnPoints` component. The coordinate system is specified in the `OutputSolnPoints` component.

```
# Comments are limited to complete lines. The default delimiter for comments
# is '#', which can be changed via parameters. Additionally, the delimiter
# separating values can also be customized (default is whitespace).
#
# The first column is the station name. The coordinates of the points are given
# in the subsequent columns.
P0  1.0  -2.0   0.0
```

```
P1  2.0  -4.0  -0.1
P2  0.0  +2.0   0.0
P3  2.5  -0.2  -0.2
P4  0.0   2.0  +0.2
```

# Appendix D

# Alternative Material Model Formulations

## D.1 Viscoelastic Formulations

The viscoelastic formulations presently used in PyLith are described in Section 5.3 on page 68. In some cases there are alternative formulations that may be used in future versions of PyLith, and those are described here.

### D.1.1 Effective Stress Formulation for a Linear Maxwell Viscoelastic Material

An alternative technique for solving the equations for a Maxwell viscoelastic material is based on the effective stress formulation described in Section 5.3.4 on page 74. A linear Maxwell viscoelastic material may be characterized by the same elastic parameters as an isotropic elastic material ($E$ and $\nu$), as well as the viscosity, $\eta$. The creep strain increment is

$$\underline{\Delta e}^{C} = \frac{\Delta t \, {}^{\tau}\underline{S}}{2\eta} . \tag{D.1}$$

Therefore,

$$\Delta \overline{e}^{C} = \frac{\Delta t \sqrt{{}^{\tau}J_2'}}{\sqrt{3}\eta} = \frac{\Delta t \, {}^{\tau}\overline{\sigma}}{3\eta}, \text{ and } {}^{\tau}\gamma = \frac{1}{2\eta} . \tag{D.2}$$

Substituting Equations 5.48 on page 74, D.1, and D.2 into 5.45 on page 74, we obtain

$${}^{t+\Delta t}\underline{S} = \frac{1}{a_E}\left\{ {}^{t+\Delta t}\underline{e}' - \frac{\Delta t}{2\eta}\left[ (1-\alpha)\, {}^{t}\underline{S} + \alpha \, {}^{t+\Delta t}\underline{S} \right] \right\} + \underline{S}^{I} . \tag{D.3}$$

Solving for ${}^{t+\Delta t}\underline{S}$,

$${}^{t+\Delta t}\underline{S} = \frac{1}{a_E + \frac{\alpha \Delta t}{2\eta}} \left[ {}^{t+\Delta t}\underline{e}' - \frac{\Delta t}{2\eta}(1-\alpha)\, {}^{t}\underline{S} + \frac{1+\nu}{E}\underline{S}^{I} \right] . \tag{D.4}$$

In this case it is possible to solve directly for the deviatoric stresses, and the effective stress function approach is not needed. To obtain the total stress, we simply use

$${}^{t+\Delta t}\sigma_{ij} = {}^{t+\Delta t}S_{ij} + \frac{1}{a_m}\left( {}^{t+\Delta t}\theta - \theta^{I} \right)\delta_{ij} + P^{I}\delta_{ij} . \tag{D.5}$$

To compute the viscoelastic tangent material matrix relating stress and strain, we need to compute the first term in Equation 5.55 on page 75. From Equation D.4, we have

$$\frac{\partial \, {}^{t+\Delta t}S_i}{\partial \, {}^{t+\Delta t}e_k'} = \frac{\delta_{ik}}{a_E + \frac{\alpha \Delta t}{2\eta}} . \tag{D.6}$$

Using this, along with Equations 5.55 on page 75, 5.56 on page 75, and 5.57 on page 75, the final material matrix relating stress and tensor strain is

$$C_{ij}^{VE} = \frac{1}{3a_m}\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} + \frac{1}{3\left(a_E + \frac{\alpha\Delta t}{2\eta}\right)}\begin{bmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ -1 & -1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 \end{bmatrix}. \tag{D.7}$$

Note that the coefficient of the second matrix approaches $E/3(1+\nu) = 1/3a_E$ as $\eta$ goes to infinity. To check the results we make sure that the regular elastic constitutive matrix is obtained for selected terms in the case where $\eta$ goes to infinity.

$$\begin{aligned} C_{11}^{E} &= \frac{E(1-\nu)}{(1+\nu)(1-2\nu)} \\ C_{12}^{E} &= \frac{E\nu}{(1+\nu)(1-2\nu)}\,. \\ C_{44}^{E} &= \frac{E}{1+\nu} \end{aligned} \tag{D.8}$$

This is consistent with the regular elasticity matrix, and Equation D.7 should thus be used when forming the stiffness matrix. We do not presently use this formulation, but it may be included in future versions.

# Appendix E

# Analytical Solutions

## E.1 Traction Problems

Computation of analytical solutions for elastostatic problems over regular domains is a relatively straightforward procedure. These problems are typically formulated in terms of a combination of displacement and traction boundary conditions, and such problems provide a good test of the code accuracy, as well as specifically testing the implementation of traction boundary conditions. We present here two simple problems for this purpose.

### E.1.1 Solutions Using Polynomial Stress Functions

Our derivation follows the procedures outlined in Timoshenko and Goodier [Timoshenko and Goodier, 1987], and we restrict ourselves to two-dimensional problems. Any problem in elastostatics must satisfy the equilibrium equations

$$\begin{aligned} \frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{xy}}{\partial y} + X = 0 \\ \frac{\partial \sigma_{yy}}{\partial y} + \frac{\partial \sigma_{xy}}{\partial x} + Y = 0, \end{aligned} \tag{E.1}$$

where $X$ and $Y$ are the body force components in the $x$ and $y$ directions, respectively, and the stress components are given by $\sigma$. In the problems considered here, we neglect body forces, so $X$ and $Y$ disappear from the equilibrium equations. The solution must also satisfy the boundary conditions, given as surface tractions over the surface of the body. Finally, the solution must satisfy the conditions of compatibility, which may be expressed as:

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)\left(\sigma_{xx} + \sigma_{yy}\right) = 0. \tag{E.2}$$

To compute the displacement field, it is also necessary to specify displacement boundary conditions.

Equations E.1 may be satisfied by taking any function $\phi$ of $x$ and $y$, and letting the stress components be given by the following expressions:

$$\sigma_{xx} = \frac{\partial^2 \phi}{\partial y^2}, \sigma_{yy} = \frac{\partial^2 \phi}{\partial x^2}, \sigma_{xy} = -\frac{\partial^2 \phi}{\partial x \partial y}. \tag{E.3}$$

The solution must also satisfy the compatibility equations. Substituting Equations E.3 into Equation E.2, we find that the stress function $\phi$ must satisfy

$$\frac{\partial^4 \phi}{\partial x^4} + 2\frac{\partial^4 \phi}{\partial x^2 \partial y^2} + \frac{\partial^4 \phi}{\partial y^4} = 0. \tag{E.4}$$

A relatively easy way to solve a number of problems is to select expressions for $\phi$ consisting of polynomial functions of $x$ and $y$ of second degree or higher. Any polynomial of second or third degree will satisfy Equation E.4. For polynomials of higher degree, they must be substituted into Equation E.4 on the preceding page to determine what restrictions must be placed on the solution.

### E.1.2 Constant Traction Applied to a Rectangular Region

For this problem, a constant normal traction, $\overrightarrow{N}$, is applied along the positive $x$-edge of the rectangular region ($x = x_0$), and roller boundary conditions are applied along the left and bottom boundaries ( E.1). Since the tractions are constant, we assume a second degree polynomial for the stress function:

$$\phi_2 = \frac{a_2}{2}x^2 + b_2 xy + \frac{c_2}{2}y^2. \tag{E.5}$$

This yields the following expressions for the stresses:

$$\sigma_{xx} = \frac{\partial^2 \phi_2}{\partial y^2} = c_2 = N,\ \sigma_{yy} = a_2 = 0,\ \sigma_{xy} = -b_2 = 0. \tag{E.6}$$

From Hooke's law, we have:

$$\epsilon_{xx} = \frac{\partial u}{\partial x} = \frac{\left(1-\nu^2\right)N}{E},\ \epsilon_{yy} = \frac{\partial v}{\partial y} = \frac{-\nu\left(1+\nu\right)N}{E},\ \epsilon_{xy} = \frac{1}{2}\left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right) = 0. \tag{E.7}$$

The strain components are thus easily obtained from the stress components.

To obtain the displacements, we must integrate Equation E.7 and make use of the displacement boundary conditions. Integrating the first two of these, we obtain

$$u = \frac{\left(1-\nu^2\right)Nx}{E} + f\left(y\right),\ v = \frac{-\nu\left(1+\nu\right)Ny}{E} + f\left(x\right). \tag{E.8}$$

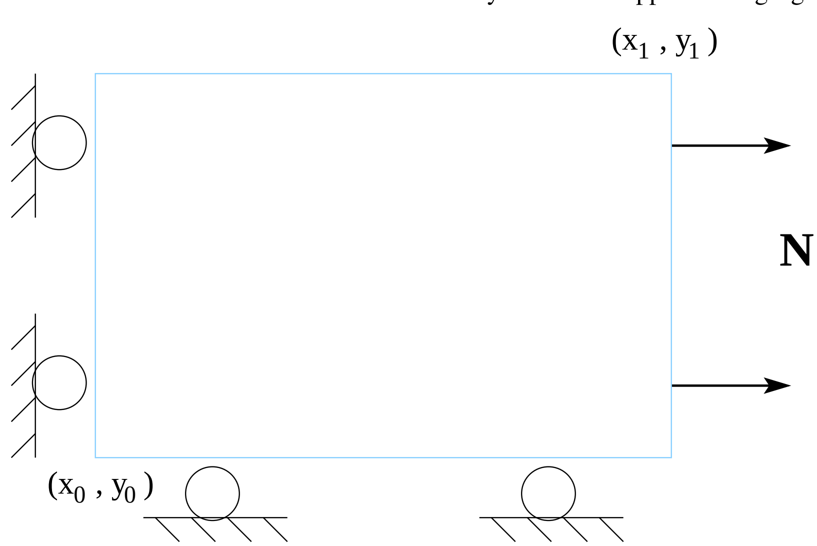
Substituting these into the third expression, we obtain

$$\frac{\partial f\left(x\right)}{\partial x} = -\frac{\partial f\left(y\right)}{\partial y}, \tag{E.9}$$

which means that both $f\left(x\right)$ and $f\left(y\right)$ must be constant. We solve for these constants using the displacement boundary conditions along $x = x_0$ and $y = y_0$. Doing this, we obtain the following expressions for the displacement components:

$$u = \frac{\left(1-\nu^2\right)N}{E}\left(x - x_0\right),\ v = \frac{\nu\left(1+\nu\right)N}{E}\left(y_0 - y\right). \tag{E.10}$$

Figure E.1: Problem with constant traction boundary conditions applied along right edge.



# Appendix F

# PyLith Software License

# Bibliography


[Aagaard et al., 2001a] Aagaard, B.T., J.F. Hall, and T.H. Heaton (2001), Characterization of near-source ground motions with earthquake simulations, *Earthquake Spectra, 17*(2), 177-207.

[Aagaard et al., 2001b] Aagaard, B.T., T.H. Heaton, and J.F. Hall (2001), Dynamic earthquake ruptures in the presence of lithostatic normal stresses: Implications for friction models and heat production, *Bulletin of the Seismological Society of America, 91*(6), 1765-1796.

[Aagaard et al., 2007] Aagaard, B., C. Williams, and M. Knepley (2007), PyLith: A finite-element code for modeling quasi-static and dynamic crustal deformation, *Eos Trans. AGU, 88*(52), Fall Meet. Suppl., Abstract T21B-0592.

[Aagaard et al., 2008] Aagaard, B., C. Williams, and M. Knepley (2008), PyLith: A finite-element code for modeling quasi-static and dynamic crustal deformation, *Eos Trans. AGU*, *89(*53), Fall Meet. Suppl., Abstract T41A-1925.

[Bathe, 1995] Bathe, K.-J. (1995), *Finite-Element Procedures*, Prentice Hall, Upper Saddle River, New Jersey, 1037 pp.

[Ben-Zion and Rice, 1997] Ben-Zion, Y. and J.R. Rice (1997), Dynamic simulations of slip on a smooth fault in an elastic solid, *Journal of Geophysical Research, 102*, 17,771–17,784.

[Brune, 1970] Brune, J.N. (1970), Tectonic stress and spectra of seismic shear waves from earthquakes, *Journal of Geophysical Research, 75*, 4997-5009.

[Courant et al., 1967] Courant, R., K. Friedrichs and H. Lewy (1967), On the Partial Differential Equations of Mathematical Physics, *IBM Journal of Research and Development*, 11(2), 215–234.

[Day and Ely, 2002] Day, S.M. and G.P. Ely (2002), Effct of a shallow weak zone on fault rupture: Numerical simulation of scale-model experiments, *Bull. Seismol. Soc. Am.*, 92(8), 3022-3041, doi: 10.1785/0120010273.

[Drucker and Prager, 1952] Drucker, D. C. and Prager, W. (1952). Soil mechanics and plastic analysis for limit design, *Quarterly of Applied Mathematics*, *10*, 157–165.

[Hayes et al., 2012] Hayes, G. P., D. J. Wald, and R. L. Johnson (2012), Slab1.0: A three-dimensional model of global subduction zone geometries, J. Geophys. Res., 117, B01302, doi:10.1029/2011JB008524.

[Liu et al., 2006] Liu, P., R.J. Archuleta, S.H. Hartzell (2006), Prediction of broadband ground-motion time histories: Hybrid low/high-frequency method with correlated random source parameters, *Bull. Seismol. Soc. Am., 96*, 2118-2130.

[Kaneko et al., 2008] Kaneko, Y., N. Lapusta, and J.-P. Ampuero (2008), Spectral element modeling of spontaneous earthquake rupture on rate and state faults: Effect of velocity-strengthening friction at shallow depths, *Journal of Geophysical Research*, *113*, B09317, doi:10.1029/2007JB005553.

[Kaus et al., 2010] Kaus, B. J. P., H. Muhlhaus, and D. A. May (2010), A stabilization algorithm for geodynamic numerical simulations with a free surface, *Physics of the Earth and Planetary Interiors*, *181*, 12-20, doi:10.1016/j.pepi.2010.04.007.

[Kirby and Kronenberg, 1987] Kirby, S. H. and A. K. Kronenberg (1987), Rheology of the lithosphere: Selected topics, *Reviews of Geophysics, 25*, 1219-1244.

[Knopoff and Ni, 2001] Knopoff, L. and X.X. Ni (2001), Numerical instability at the edge of a dynamic fracture, *Geophysical Journal International, 147*(3), 1-6, doi: 10.1046/j.1365-246x.2001.01567.x.

[Kojic and Bathe, 1987] Kojic, M. and K.-J. Bathe (1987), The 'Effective Stress-Function' Algorithm for Thermo-Elasto-Plasticity and Creep, *Int. J. Num. Meth. Eng., 24*, 1509-1532.

[McGarr, 1988] McGarr, A. (1988), On the state of lithospheric stress in the absence of applied tectonic forces, *Journal of Geophysical Research*, *93*, 13,609-13,617.

[Menke, 1984] Menke, W. (1984), *Geophysical Data Analysis: Discrete Inverse Theory*, Academic Press, Inc., Orlando, 260 pp.

[Okada, 1992] Okada, Y., Internal deformation due to shear and tensile faults in a half-space (1992), *Bull. Seismol. Soc. Am.*, *83*, 1018-1040.

[Paterson, 1994] Paterson, W. S. B. (1994), *The Physics of Glaciers, Third Edition*, Elsevier Science Ltd., Oxford, 480 pp.

[Prentice, 1968] Prentice, J. H, (1968), Dimensional problem of the power law in rheology, *Nature*, *217*, 157.

[Savage and Prescott, 1978] Savage, J. C. and W. H. Prescott (1978), Asthenosphere readjustment and the earthquake cycle, *Journal of Geophysical Research, 83*, 3369-3376.

[Stephenson, 2007] Stephenson, W.J. (2007), Velocity and density models incorporating the Cascadia subduction zone for 3D earthquake ground motion simulations, Version 1.3: U.S. Geological Survey, Earthquake Hazards Ground Motion Investigations, Open-File Report 2007âĂŞ1348, 24 pages, https://pubs.usgs.gov/of/2007/1348/.

[Taylor, 2003] Taylor, R.L. (2003), 'FEAP–A Finite Element Analysis Program', *Version 7.5 Theory Manual*, 154 pp.

[Timoshenko and Goodier, 1987] Timoshenko, S.P. and J.N. Goodier (1987), *Theory of Elasticity, Third Edition*, McGraw-Hill, New York, 567 pp.

[Williams et al., 2005] Williams, C.A., B. Aagaard, and M.G. Knepley (2005), Development of software for studying earthquakes across multiple spatial and temporal scales by coupling quasi-static and dynamic simulations, *Eos Trans. AGU, 86*(52), Fall Meet. Suppl., Abstract S53A-1072.

[Williams, 2006] Williams, C.A. (2006), Development of a package for modeling stress in the lithosphere, *Eos Trans. AGU, 87*(36), Jt. Assem. Suppl., Abstract T24A-01 Invited.

[Zienkiewicz and Taylor, 2000] Zienkiewicz, O.C. and R.L. Taylor (2000), *The Finite Element Method, Fifth Edition, Volume 2: Solid Mechanics*, Butterworth-Heinemann, Oxford, 459 pp.125