PyLith Developer Guide

Brad Aagaard, Matthew Knepley, Charles Williams

October 23, 2018

Contents

Chapter 1

Multiphysics Finite-Element Formulation

This chapter will become part of the governing equations chapter in the PyLith Manual.

1.1 General Finite-Element Formulation

Within the PETSc solver framework, we want to solve a system of partial differential equations in which the strong form can be expressed as $F(t,s,\dot{s}) = G(t,s)$, $s(t_0) = s_0$, where F and G are vector functions, t is time, and s is the solution vector.

Using the finite-element method we manipulate the weak form of the system of equations involving a vector field $\vec{u}$ into integrals over the domain Ω with the form,

$$\int_\Omega \vec{\psi}^u_{trial} \cdot \vec{f}_0(t,s,\dot{s}) + \nabla\vec{\psi}^u_{trial} : \boldsymbol{f}_1(t,s,\dot{s})\, d\Omega = \int_\Omega \vec{\psi}^u_{trial} \cdot \vec{g}_0(t,s) + \nabla\vec{\psi}^u_{trial} : \boldsymbol{g}_1(t,s)\, d\Omega, \tag{1.1}$$

where $\vec{\psi}^u_{trial}$ is the trial function for field $\vec{u}$, $\vec{f}_0$ and $\vec{g}_0$ are vectors, and $\boldsymbol{f}_1$ and $\boldsymbol{g}_1$ are tensors. With multiple partial differential equations we will have multiple equations of this form, and the solution vector s, which we usually write as $\vec{s}$, will be composed of several different fields, such as displacement $\vec{u}$, velocity $\vec{v}$, pressure p, and temperature T.

For consistency with the PETSc time stepping formulation, we call $G(t,s)$ the RHS function and call $F(t,s,\dot{s})$ the LHS (or I) function. Likewise, the Jacobian of $G(t,s)$ is the RHS Jacobian and the Jacobian of $F(t,s,\dot{s})$ is the LHS Jacobian. In most cases, we can take $F(t,s,\dot{s}) = \dot{s}$, or as close to this as possible. This results in miminal changes to the formulation in order to accommodate both implicit and explicit time stepping algorithms.

Using a finite-element discretization we break up the domain and boundary integrals into sums over the cells and boundary faces, respectively. Using numerical quadrature those sums in turn involve sums over the values at the quadrature points with appropriate weights. Thus, computation of the RHS function boils down to point-wise evaluation of $\vec{g}_0(t,s)$ and $\boldsymbol{g}_1(t,s)$, and computation of the LHS function boils down to point-wise evaluation of $\vec{f}_0(t,s,\dot{s})$ and $\boldsymbol{f}_1(t,s,\dot{s})$.

1.1.1 Jacobian

The LHS Jacobian J_F is $\frac{\partial F}{\partial s} + t_{shift}\frac{\partial F}{\partial \dot{s}}$ and the RHS Jacobian J_G is $\frac{\partial G}{\partial s}$, where t_{shift} arises from the temporal discretization . We put the Jacobians for each equation in the form:

$$J_F = \int_\Omega \vec{\psi}_{trial} \cdot \boldsymbol{J}_{f0}(t,s,\dot{s}) \cdot \vec{\psi}_{basis} + \vec{\psi}_{trial} \cdot \boldsymbol{J}_{f1}(t,s,\dot{s}) : \nabla\vec{\psi}_{basis} + \nabla\vec{\psi}_{trial} : \boldsymbol{J}_{f2}(t,s,\dot{s}) \cdot \vec{\psi}_{basis} + \nabla\vec{\psi}_{trial} : \boldsymbol{J}_{f3}(t,s,\dot{s}) : \nabla\vec{\psi}_{basis}\, d\Omega \tag{1.2}$$

$$J_G = \int_\Omega \vec{\psi}_{trial} \cdot \boldsymbol{J}_{g0}(t,s) \cdot \vec{\psi}_{basis} + \vec{\psi}_{trial} \cdot \boldsymbol{J}_{g1}(t,s) : \nabla\vec{\psi}_{basis} + \nabla\vec{\psi}_{trial} : \boldsymbol{J}_{g2}(t,s) \cdot \vec{\psi}_{basis} + \nabla\vec{\psi}_{trial} : \boldsymbol{J}_{g3}(t,s) : \nabla\vec{\psi}_{basis}\, d\Omega, \tag{1.3}$$

where $\vec{\psi}_{basis}$ is a basis function. Expressed in index notation the Jacobian coupling solution field components s_i and s_j is

$$J^{s_i s_j} = \int_\Omega \psi_{trial\,i} J_0^{s_i s_j} \psi_{basis\,j} + \psi_{trial\,i} J_1^{s_j s_j l} \psi_{basis\,j,l} + \psi_{trial\,i,k} J_2^{s_i s_j k} \psi_{basis\,j} + \psi_{trial\,i,k} J_3^{s_i s_j kl} \psi_{basis\,j,l}\, d\Omega, \tag{1.4}$$

It is clear that the tensors J_0, J_1, J_2, and J_3 have various sizes: $J_0(n_i, n_j)$, $J_1(n_i, n_j, d)$, $J_2(n_i, n_j, d)$, $J_3(n_i, n_j, d, d)$, where n_i is the number of components in solution field s_i, n_j is the number of components in solution field s_j, and d is the spatial dimension. Alternatively, expressed in discrete form, the Jacobian for the coupling between solution fields s_i and s_j is

$$J^{s_i s_j} = J_0^{s_i s_j} + J_1^{s_i s_j} B + B^T J_2^{s_i s_j} + B^T J_3^{s_i s_j} B, \tag{1.5}$$

where B is a matrix of the derivatives of the basis functions and B^T is a matrix of the derivatives of the trial functions.

MATT: (Comment from Brad) Additionally, I think it is very important that we have a way to control allocation of the sparse matrix. We do not want to allocate portions that are not coupled, because it is way too much memory. A simple way to do this would be to create an array that is #fields x #fields and have the materials populate it with values to indicate whether they couple those fields or not. We could use a value to indicate if the Jacobian was diagonal or not as well.

1.1.2 PETSc TS Notes

- If no LHS (or I) function is given, then the PETSc TS assumes $F(t, s, \dot{s}) = \dot{s}$.
- If no RHS function is given, then the PETSc TS assumes $G(t, s) = 0$.
- Explicit time stepping with the PETSc TS requires $F(t, s, \dot{s}) = \dot{s}$.
  - Because $F(t, s, \dot{s}) = \dot{s}$, we do not specify the functions $\vec{f}_0(t, s, \dot{s})$ and $\boldsymbol{f}_1(t, s, \dot{s})$ because the PETSc TS will assume this is the case if no LHS (or I) function is given.
  - We also do not specify J_F or J_G.
  - This leaves us with only needing to specify $\vec{g}_0(t, s)$ and $\boldsymbol{g}_1(t, s)$.
  - The PETSc TS will verify that the LHS (or I) function is null.

1.1.3 Explicit Time Stepping

For explicit time stepping with the PETSc TS, we need $F(t, s, \dot{s}) = \dot{s}$. Using a finite-element formulation, $F(t, s, \dot{s})$ generally involves integrals of the inertia over the domain. It is tempting to simply move these terms to the RHS, but that introduces inertial terms into the boundary conditions, which makes them less intuitive. Instead, we transform our equation into the form $F^*(t, s, \dot{s}) = \dot{s} = G^*(t, s)$ by writing $F(t, s, \dot{s}) = M\dot{s}$, so that $\dot{s} = M^{-1}G(t, s) = G^*(t, s)$. We take M to be a lumped (diagonal) matrix, so that M^{-1} is trivial to compute. In computing the RHS function, $G^*(t, s)$, we compute $G(t, s)$, then compute M and M^{-1}, and then $M^{-1}G(t, s)$. For the elasticity equation with inertial terms, M contains the mass matrix.

1.2 Elasticity With Infinitesimal Strain and No Faults

We begin with the elasticity equation including the inertial term,

$$\rho \frac{\partial^2 \vec{u}}{\partial t^2} - \vec{f}(\vec{x}, t) - \boldsymbol{\nabla} \cdot \boldsymbol{\sigma}(\vec{u}) = \vec{0} \text{ in } \Omega, \tag{1.6}$$

$$\boldsymbol{\sigma} \cdot \vec{n} = \vec{\tau}(\vec{x}, t) \text{ on } \Gamma_\tau, \tag{1.7}$$

$$\vec{u} = \vec{u}_0(\vec{x}, t) \text{ on } \Gamma_u, \tag{1.8}$$

where $\vec{u}$ is the displacement vector, ρ is the mass density, $\vec{f}$ is the body force vector, $\boldsymbol{\sigma}$ is the Cauchy stress tensor, $\vec{x}$ is the spatial coordinate, and t is time. We specify tractions $\vec{\tau}$ on boundary Γ_τ, and displacements $\vec{u}_0$ on boundary Γ_u. Because both $\vec{\tau}$ and $\vec{u}$ are vector quantities, there can be some spatial overlap of boundaries Γ_τ and Γ_u; however, a degree of freedom at any location cannot be associated with both prescribed displacements (Dirichlet) and traction (Neumann) boundary conditions simultaneously.

1.2.1 Notation

- Unknowns
  - $\vec{u}$ Displacement field
  - $\vec{v}$ Velocity field (if including inertial term)
- Derived quantities
  - $\boldsymbol{\sigma}$ Stress tensor
  - $\boldsymbol{\epsilon}$ Strain tensor
- Constitutive parameters
  - μ Shear modulus
  - K Bulk modulus
  - ρ Density
- Source terms
  - $\vec{f}$ Body force per unit volume, for example $\rho\vec{g}$

1.2.2 Neglecting Inertia

If we neglect the intertial term, then time dependence only arises from history-dependent constitutive equations and boundary conditions. Considering the displacement $\vec{u}$ as the unknown, we have

$$\vec{s}^T = (\vec{u})^T, \tag{1.9}$$

$$\vec{0} = \vec{f}(\vec{x}, t) + \boldsymbol{\nabla}\cdot\boldsymbol{\sigma}(\vec{u}) \text{ in } \Omega, \tag{1.10}$$

$$\boldsymbol{\sigma}\cdot\vec{n} = \vec{\tau}(\vec{x}, t) \text{ on } \Gamma_\tau, \tag{1.11}$$

$$\vec{u} = \vec{u}_0(\vec{x}, t) \text{ on } \Gamma_u. \tag{1.12}$$

We create the weak form by taking the dot product with the trial function $\vec{\psi}^u_{trial}$ and integrating over the domain:

$$0 = \int_\Omega \vec{\psi}^u_{trial}\cdot\left(\vec{f}(t) + \boldsymbol{\nabla}\cdot\boldsymbol{\sigma}(\vec{u})\right) d\Omega. \tag{1.13}$$

Using the divergence theorem and incorporating the Neumann bounday condition yields

$$0 = \int_\Omega \vec{\psi}^u_{trial}\cdot\vec{f}(t) + \nabla\vec{\psi}^u_{trial} : -\boldsymbol{\sigma}(\vec{u})\, d\Omega + \int_{\Gamma_\tau} \vec{\psi}^u_{trial}\cdot\vec{\tau}(\vec{x}, t)\, d\Gamma. \tag{1.14}$$

Identifying $F(t, s, \dot{s})$ and $G(t, s)$, we have

$$F^u(t, s, \dot{s}) = \vec{0}, \qquad G^u(t, s) = \int_\Omega \vec{\psi}^u_{trial}\cdot\underbrace{\vec{f}(\vec{x}, t)}_{g^u_0} + \nabla\vec{\psi}^u_{trial} : \underbrace{-\boldsymbol{\sigma}(\vec{u})}_{g^u_1} d\Omega + \int_{\Gamma_\tau} \vec{\psi}^u_{trial}\cdot\underbrace{\vec{\tau}(\vec{x}, t)}_{g^u_0} d\Gamma. \tag{1.15}$$

1.2.2.1 Jacobians

With the solution composed of the displacement field and no LHS function, we only have Jacobians for the RHS,

$$J^{uu}_G = \frac{\partial G^u}{\partial u} = \int_\Omega \nabla\vec{\psi}^u_{trial} : \frac{\partial}{\partial u}(-\boldsymbol{\sigma})\, d\Omega = \int_\Omega \nabla\vec{\psi}^u_{trial} : -\boldsymbol{C} : \frac{1}{2}(\nabla + \nabla^T)\vec{\psi}^u_{basis}\, d\Omega = \int_\Omega \psi^v_{trial\, i,k} \underbrace{\left(-C_{ikjl}\right)}_{J^{uu}_{g3}} \psi^u_{basis\, j,l}\, d\Omega \tag{1.16}$$

1.2.3 Including Inertia

For convenience we cast the elasticity equation in the form of a first order equation by considering both the displacement $\vec{u}$ and velocity $\vec{v}$ as unknowns,

$$\vec{s}^T = (\vec{u} \quad \vec{v})^T, \tag{1.17}$$

$$\frac{\partial \vec{u}}{\partial t} = \vec{v}, \tag{1.18}$$

$$\rho \frac{\partial \vec{v}}{\partial t} = \vec{f}(\vec{x}, t) + \boldsymbol{\nabla} \cdot \boldsymbol{\sigma}(\vec{u}) \text{ in } \Omega, \tag{1.19}$$

$$\boldsymbol{\sigma} \cdot \vec{n} = \vec{\tau}(\vec{x}, t) \text{ on } \Gamma_\tau, \tag{1.20}$$

$$\vec{u} = \vec{u}_0(\vec{x}, t) \text{ on } \Gamma_u. \tag{1.21}$$

For trial functions $\vec{\psi}^u_{trial}$ and $\vec{\psi}^v_{trial}$ we write the weak form as

$$\int_\Omega \vec{\psi}^u_{trial} \cdot \left(\frac{\partial \vec{u}}{\partial t} \right) d\Omega = \int_\Omega \vec{\psi}^u_{trial} \cdot \vec{v} \, d\Omega, \tag{1.22}$$

$$\int_\Omega \vec{\psi}^v_{trial} \cdot \left(\rho \frac{\partial \vec{v}}{\partial t} \right) d\Omega = \int_\Omega \vec{\psi}^v_{trial} \cdot \left(\vec{f}(\vec{x}, t) + \boldsymbol{\nabla} \cdot \boldsymbol{\sigma}(\vec{u}) \right) d\Omega. \tag{1.23}$$

Using the divergence theorem and incorporating the Neumann boundary conditions, we can rewrite the second equation as

$$\int_\Omega \vec{\psi}^v_{trial} \cdot \left(\rho \frac{\partial \vec{v}}{\partial t} \right) d\Omega = \int_\Omega \vec{\psi}^v_{trial} \cdot \vec{f}(\vec{x}, t) + \nabla \vec{\psi}^v_{trial} : -\boldsymbol{\sigma}(\vec{u}) \, d\Omega + \int_{\Gamma_\tau} \vec{\psi}^v_{trial} \cdot \vec{\tau}(\vec{x}, t) \, d\Gamma. \tag{1.24}$$

1.2.3.1 Implicit Time Stepping

The resulting system of equations to solve is

$$\int_\Omega \vec{\psi}^u_{trial} \cdot \left(\frac{\partial \vec{u}}{\partial t} \right) d\Omega = \int_\Omega \vec{\psi}^u_{trial} \cdot \vec{v} \, d\Omega, \tag{1.25}$$

$$\int_\Omega \vec{\psi}^v_{trial} \cdot \left(\rho \frac{\partial \vec{v}}{\partial t} \right) d\Omega = \int_\Omega \vec{\psi}^v_{trial} \cdot \vec{f}(\vec{x}, t) + \nabla \vec{\psi}^v_{trial} : -\boldsymbol{\sigma}(\vec{u}) \, d\Omega + \int_{\Gamma_\tau} \vec{\psi}^u_{trial} \cdot \vec{\tau}(\vec{x}, t) \, d\Gamma. \tag{1.26}$$

Identifying $F(t, s, \dot{s})$ and $G(t, s)$, we have

$$F^u(t, s, \dot{s}) = \int_\Omega \vec{\psi}^u_{trial} \cdot \underbrace{\left(\frac{\partial \vec{u}}{\partial t} \right)}_{f_0^u} d\Omega, \qquad G^u(t, s) = \int_\Omega \vec{\psi}^u_{trial} \cdot \underbrace{\vec{v}}_{g_0^u} d\Omega, \tag{1.27}$$

$$F^v(t, s, \dot{s}) = \int_\Omega \vec{\psi}^v_{trial} \cdot \underbrace{\left(\rho \frac{\partial \vec{v}}{\partial t} \right)}_{f_0^v} d\Omega, \qquad G^v(t, s) = \int_\Omega \vec{\psi}^v_{trial} \cdot \underbrace{\vec{f}(\vec{x}, t)}_{g_0^v} + \nabla \vec{\psi}^v_{trial} : \underbrace{-\boldsymbol{\sigma}(\vec{u})}_{g_1^v} d\Omega + \int_{\Gamma_\tau} \vec{\psi}^u_{trial} \cdot \underbrace{\vec{\tau}(\vec{x}, t)}_{g_0^v} d\Gamma. \tag{1.28}$$

1.2.3.2 Jacobians

With two fields we have four Jacobians for each side of the equation associated with the coupling of the two fields,

$$J_F^{uu} = \frac{\partial F^u}{\partial u} + t_{shift}\frac{\partial F^u}{\partial \dot{u}} = \int_\Omega \vec{\psi}^u_{trial}\cdot t_{shift}\,\vec{\psi}^u_{basis}\,d\Omega = \int_\Omega \psi^u_{trial\,i}\underbrace{t_{shift}\delta_{ij}}_{J^{uu}_{f0}}\psi^u_{basis\,j}\,d\Omega \tag{1.29}$$

$$J_F^{uv} = \frac{\partial F^u}{\partial v} + t_{shift}\frac{\partial F^u}{\partial \dot{v}} = \mathbf{0} \tag{1.30}$$

$$J_F^{vu} = \frac{\partial F^v}{\partial u} + t_{shift}\frac{\partial F^v}{\partial \dot{u}} = \mathbf{0} \tag{1.31}$$

$$J_F^{vv} = \frac{\partial F^v}{\partial v} + t_{shift}\frac{\partial F^v}{\partial \dot{v}} = \int_\Omega \vec{\psi}^v_{trial}\cdot t_{shift}\,\rho\,\vec{\psi}^v_{basis}\,d\Omega = \int_\Omega \psi^v_{trial\,i}\underbrace{t_{shift}\,\rho\,\delta_{ij}}_{J^{vv}_{f0}}\psi^v_{basis\,j}\,d\Omega \tag{1.32}$$

$$J_G^{uu} = \frac{\partial G^u}{\partial u} = \mathbf{0} \tag{1.33}$$

$$J_G^{uv} = \frac{\partial G^u}{\partial v} = \int_\Omega \vec{\psi}^u_{trial}\cdot\vec{\psi}^v_{basis}\,d\Omega = \int_\Omega \psi^u_{trial\,i}\underbrace{\delta_{ij}}_{J^{uv}_{g0}}\psi^v_{basis\,j}\,d\Omega \tag{1.34}$$

$$J_G^{vu} = \frac{\partial G^v}{\partial u} = \int_\Omega \nabla\vec{\psi}^v_{trial} : \frac{\partial}{\partial u}(-\boldsymbol{\sigma})\,d\Omega = \int_\Omega \nabla\vec{\psi}^v_{trial} : -\boldsymbol{C} : \frac{1}{2}(\nabla+\nabla^T)\vec{\psi}^u_{basis}\,d\Omega = \int_\Omega \psi^v_{trial\,i,k}\underbrace{\left(-C_{ikjl}\right)}_{J^{vu}_{g3}}\psi^u_{basis\,j,l}\,d\Omega \tag{1.35}$$

$$J_G^{vv} = \frac{\partial G^v}{\partial v} = \mathbf{0} \tag{1.36}$$

1.2.4 Explicit Time Stepping

Recall that explicit time stepping requires $F(t,s,\dot{s}) = \dot{s}$. We write $F^*(t,s,\dot{s}) = \dot{s}$ and $G^*(t,s) = J_F^{-1}G(t,s)$ and we do not provide functions for f_0 and f_1. Thus, our system of equations to solve is

$$\int_\Omega \vec{\psi}^u_{trial}\cdot\frac{\partial\vec{u}}{\partial t}\,d\Omega = \int_\Omega \vec{\psi}^u_{trial}\cdot\vec{v}\,d\Omega, \tag{1.37}$$

$$\int_\Omega \vec{\psi}^v_{trial}\cdot\frac{\partial\vec{v}}{\partial t}\,d\Omega = \frac{1}{\int_\Omega \vec{\psi}^v_{trial}\cdot\rho\,\vec{\psi}^v_{basis}\,d\Omega}\left(\int_\Omega \vec{\psi}^v_{trial}\cdot\vec{f}(\vec{x},t) + \nabla\vec{\psi}^v_{trial} : -\boldsymbol{\sigma}(\vec{u})\,d\Omega + \int_{\Gamma_\tau}\vec{\psi}^u_{trial}\cdot\vec{\tau}(\vec{x},t)\,d\Gamma\right). \tag{1.38}$$

Identifying $F(t,s,\dot{s})$ and $G(t,s)$, we have

$$F^u(t,s,\dot{s}) = \int_\Omega \vec{\psi}^u_{trial}\cdot\frac{\partial\vec{u}}{\partial t}\,d\Omega, \tag{1.39}$$

$$G^u(t,s) = \int_\Omega \vec{\psi}^u_{trial}\cdot\underbrace{\vec{v}}_{g^u_0}\,d\Omega, \tag{1.40}$$

$$F^v(t,s,\dot{s}) = \int_\Omega \vec{\psi}^v_{trial}\cdot\frac{\partial\vec{v}}{\partial t}\,d\Omega, \tag{1.41}$$

$$G^v(t,s) = \frac{1}{\int_\Omega \vec{\psi}^v_{trial}\cdot\underbrace{\rho}_{J^{vv}_{f0}}\vec{\psi}^v_{basis}\,d\Omega}\left(\int_\Omega \vec{\psi}^v_{trial}\cdot\underbrace{\vec{f}(t)}_{g^v_0} + \nabla\vec{\psi}^v_{trial} : \underbrace{-\boldsymbol{\sigma}(\vec{u})}_{g^v_1}\,d\Omega + \int_{\Gamma_\tau}\vec{\psi}^v_{trial}\cdot\underbrace{\vec{\tau}(\vec{x},t)}_{g^v_0}\,d\Gamma\right). \tag{1.42}$$

where $J^{uu}_{f0} = \boldsymbol{I}$, and we refer to J_F as the LHS (or I) Jacobian for explicit time stepping.

1.2.5 Elasticity Constitutive Models

The Jacobian for the elasticity equation is

$$J_G^{vu} = \frac{\partial G^{v_i}}{\partial u_j}. \tag{1.43}$$

In computing the derivative, we consider the linearized form:

$$\sigma_{ik} = C_{ikjl}\epsilon_{jl} \tag{1.44}$$

$$\sigma_{ik} = C_{ikjl}\frac{1}{2}(u_{j,l} + u_{l,j}) \tag{1.45}$$

$$\sigma_{ik} = \frac{1}{2}(C_{ikjl} + C_{iklj})u_{j,l} \tag{1.46}$$

$$\sigma_{ik} = C_{ikjl}u_{j,l} \tag{1.47}$$

$$\tag{1.48}$$

In computing the Jacobian, we take the derivative of the stress tensor with respect to the displacement field,

$$\frac{\partial}{\partial u_j}\sigma_{ik} = C_{ikjl}\psi^u_{basis\,j,l}, \tag{1.49}$$

so we have

$$\boxed{J^{vu}_{g3}(i,j,k,l) = -C_{ikjl}} \tag{1.50}$$

For many elasticity constitutive models we prefer to separate the stress into the mean stress and deviatoric stress:

$$\boldsymbol{\sigma} = \sigma^{mean}\boldsymbol{I} + \boldsymbol{\sigma}^{dev}, \text{ where} \tag{1.51}$$

$$\sigma^{mean} = \frac{1}{3}\mathrm{Tr}(\boldsymbol{\sigma}) = \frac{1}{3}(\sigma_{11} + \sigma_{22} + \sigma_{33}). \tag{1.52}$$

Sometimes it is convenient to use pressure (positive pressure corresponds to compression) instead of the mean stress:

$$\boldsymbol{\sigma} = -p\boldsymbol{I} + \boldsymbol{\sigma}^{dev}, \text{ where} \tag{1.53}$$

$$p = -\frac{1}{3}\mathrm{Tr}(\boldsymbol{\sigma}). \tag{1.54}$$

The Jacobian with respect to the deviatoric stress is

$$\frac{\partial \sigma^{dev}_{ik}}{\partial u_j} = \frac{\partial}{\partial u_j}\left(\sigma_{ik} - \frac{1}{3}\sigma_{mm}\delta_{ik}\right) \tag{1.55}$$

$$\frac{\partial \sigma^{dev}_{ik}}{\partial u_j} = C_{ikjl}\psi^u_{basis\,j,l} - \frac{1}{3}C_{mmjl}\delta_{ik}\psi^u_{basis\,j,l}. \tag{1.56}$$

We call these modified elastic constantas C^{dev}_{ikjl}, so that we have

$$\boxed{C^{dev}_{ikjl} = C_{ikjl} - \frac{1}{3}C_{mmjl}\delta_{ik}.} \tag{1.57}$$

.

1.2.5.1 Isotropic Linear Elasticity

We implement isotropic linear elasticity both with and without a reference stress-strain state. With a linear elastic material it is often convenient to compute the deformation relative to an unknown initial stress-strain state. Furthermore, when we use an initial undeformed configuration with zero stress and strain, the reference stress and strain are zero, so this presents a simplifcation of the more general case of the stress-strain state relative to the reference stress-strain state.

Without a reference stress-strain state, we have

$$\sigma_{ij} = \lambda \epsilon_{kk} \delta_{ij} + 2\mu \epsilon_{ij}, \tag{1.58}$$

and with a reference stress-strain state, we have

$$\sigma_{ij} = \sigma_{ij}^{ref} + \lambda \left(\epsilon_{kk} - \epsilon_{kk}^{ref} \right) \delta_{ij} + 2\mu \left(\epsilon_{ij} - \epsilon_{ij}^{ref} \right). \tag{1.59}$$

The mean stress is

$$\sigma^{mean} = \frac{1}{3} \sigma_{kk}, \tag{1.60}$$

$$\sigma^{mean} = \frac{1}{3} \sigma_{kk}^{ref} + \left(\lambda + \frac{2}{3} \mu \right) \left(\epsilon_{kk} - \epsilon_{kk}^{ref} \right), \tag{1.61}$$

$$\boxed{\sigma^{mean} = \frac{1}{3} \sigma_{kk}^{ref} + K \left(\epsilon_{kk} - \epsilon_{kk}^{ref} \right),} \tag{1.62}$$

where $K = \lambda + 2\mu/3$ is the bulk modulus. If the reference stress and reference strain are both zero, then this reduces to

$$\boxed{\sigma^{mean} = K \epsilon_{kk}.} \tag{1.63}$$

The deviatoric stress is

$$\sigma_{ij}^{dev} = \sigma_{ij} - \sigma^{mean} \delta_{ij}, \tag{1.64}$$

$$\sigma_{ij}^{dev} = \sigma_{ij}^{ref} + \lambda \left(\epsilon_{kk} - \epsilon_{kk}^{ref} \right) \delta_{ij} + 2\mu \left(\epsilon_{ij} - \epsilon_{ij}^{ref} \right) - \left(\frac{1}{3} \sigma_{kk}^{ref} + \left(\lambda + \frac{2}{3} \mu \right) \left(\epsilon_{kk} - \epsilon_{kk}^{ref} \right) \right) \delta_{ij}, \tag{1.65}$$

$$\sigma_{ij}^{dev} = \sigma_{ij}^{ref} - \frac{1}{3} \sigma_{kk}^{ref} \delta_{ij} + 2\mu \left(\epsilon_{ij} - \epsilon_{ij}^{ref} \right) - \frac{2}{3} \mu \left(\epsilon_{kk} - \epsilon_{kk}^{ref} \right) \delta_{ij}, \tag{1.66}$$

$$\boxed{\sigma_{ij}^{dev} = \begin{cases} \sigma_{ii}^{ref} - \frac{1}{3} \sigma_{kk}^{ref} + 2\mu \left(\epsilon_{ii} - \epsilon_{ii}^{ref} \right) - \frac{2}{3} \mu \left(\epsilon_{kk} - \epsilon_{kk}^{ref} \right) & \text{if} \quad i = j, \\ \sigma_{ij}^{ref} + 2\mu \left(\epsilon_{ij} - \epsilon_{ij}^{ref} \right) & \text{if} \quad i \neq j. \end{cases}} \tag{1.67}$$

If the reference stress and reference strain are both zero, then this reduces to

$$\boxed{\sigma_{ij}^{dev} = \begin{cases} 2\mu \epsilon_{ii} - \frac{2}{3} \mu \epsilon_{kk} & \text{if} \quad i = j, \\ 2\mu \epsilon_{ij} & \text{if} \quad i \neq j. \end{cases}} \tag{1.68}$$

For isotropic linear elasticity

$$C_{1112} = C_{1113} = C_{1113} = C_{1121} = C_{1123} = C_{1131} = C_{1132} = 0 \tag{1.69}$$

$$C_{1211} = C_{1213} = C_{1222} = C_{1223} = C_{1231} = C_{1232} = C_{1233} = 0, \tag{1.70}$$

and

$$C_{1111} = C_{2222} = C_{3333} = \lambda + 2\mu, \tag{1.71}$$

$$C_{1122} = C_{1133} = C_{2233} = \lambda, \tag{1.72}$$

$$C_{1212} = C_{2323} = C_{1313} = \mu. \tag{1.73}$$

The deviatoric elastic constants are:

$$C_{1111}^{dev} = C_{2222}^{dev} = C_{3333}^{dev} = \frac{4}{3} \mu, \tag{1.74}$$

$$C_{1122}^{dev} = C_{1133}^{dev} = C_{2233}^{dev} = -\frac{2}{3} \mu, \tag{1.75}$$

$$C_{1212}^{dev} = C_{2323}^{dev} = C_{1313}^{dev} = \mu. \tag{1.76}$$

1.2.5.2 Isotropic Generalized Maxwell Viscoelasticity

We use the same general formulation for both the simple Maxwell viscoelastic model and the generalized Maxwell model (several Maxwell models in parallel). We implement the Maxwell models both with and without a reference stress-strain state. Note that it is also possible to specify an initial state variable value (viscous strain). Viscous flow is completely deviatoric, so we split the stress into volumetric and deviatoric parts, as described above. The volumetric part is identical to that of an isotropic elastic material. The deviatoric part is given by:

$$\sigma_{ij}^{dev}(t) = 2\mu_{tot}\left(\mu_0\epsilon_{ij}^{dev}(t) + \sum_{m=1}^{N}\mu_m h_{ij}^m(t) - \epsilon_{ij}^{refdev}\right) + \sigma_{ij}^{refdev}, \tag{1.77}$$

where μ_{tot} is the total shear modulus of the model, μ_0 is the fraction of the shear modulus accommodated by the elastic spring in parallel with the Maxwell models, the μ_m are the fraction of the shear modulus accommodated by each Maxwell model spring, and ϵ_{ij}^{refdev} and σ_{ij}^{refdev} are the reference deviatioric strain and stress, respectively. The viscous strain is:

$$h_{ij}^m(t) = \exp\frac{-\Delta t}{\tau_m}h_{ij}^m(t_n) + \Delta h_{ij}^m, \tag{1.78}$$

where t_n is a time between $t = 0$ and $t = t$, Δh_{ij}^m is the viscous strain between $t = t_n$ and $t = t$, and τ_m is the Maxwell time:

$$\tau_m = \frac{\eta_m}{\mu_{tot}\mu_m}. \tag{1.79}$$

Approximating the strain rate as constant over each time step, this is given as:

$$\Delta h_{ij}^m = \frac{\tau_m}{\Delta t}\left(1 - \exp\frac{-\Delta t}{\tau_m}\right)\left(\epsilon_{ij}^{dev}(t) - \epsilon_{ij}^{dev}(t_n)\right) = \Delta h^m\left(\epsilon_{ij}^{dev}(t) - \epsilon_{ij}^{dev}(t_n)\right). \tag{1.80}$$

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

$$\Delta h^m \approx 1 - \frac{1}{2}\left(\frac{\Delta t}{\tau_m}\right) + \frac{1}{3!}\left(\frac{\Delta t}{\tau_m}\right)^2 - \frac{1}{4!}\left(\frac{\Delta t}{\tau_m}\right)^3 + \cdots. \tag{1.81}$$

This converges with only a few terms.

1.3 Elasticity With Infinitesimal Strain and Faults With Prescribed Slip

For each fault, which is an internal interface, we add a boundary condition prescribing the jump in the displacement field across the fault,

$$\rho\frac{\partial^2\vec{u}}{\partial t^2} - \vec{f}(\vec{x},t) - \boldsymbol{\nabla}\cdot\boldsymbol{\sigma}(\vec{u}) = \vec{0} \text{ in } \Omega, \tag{1.82}$$

$$\boldsymbol{\sigma}\cdot\vec{n} = \vec{\tau}(\vec{x},t) \text{ on } \Gamma_\tau, \tag{1.83}$$

$$\vec{u} = \vec{u}_0(\vec{x},t) \text{ on } \Gamma_u, \tag{1.84}$$

$$\vec{0} = \vec{d}(\vec{x},t) - \vec{u}^+(\vec{x},t) + \vec{u}^-(\vec{x},t) \text{ on } \Gamma_f, \tag{1.85}$$

where $\vec{u}^+$ is the displacement vector on the "positive" side of the fault, $\vec{u}^-$ is the displacement vector on the "negative side of the fault, $\vec{d}$ is the slip vector on the fault, and $\vec{n}$ is the fault normal which points from the negative side of the fault to the positive side of the fault. Using a domain decomposition approach for constraining the fault slip, yields additional Neumann-like boundary conditions on the fault surface,

$$\boldsymbol{\sigma}\cdot\vec{n} = -\vec{\lambda}(\vec{x},t) \text{ on } \Gamma_{f^+}, \tag{1.86}$$

$$\boldsymbol{\sigma}\cdot\vec{n} = +\vec{\lambda}(\vec{x},t) \text{ on } \Gamma_{f^-}, \tag{1.87}$$

where $\vec{\lambda}$ is the vector of Lagrange multipliers corresponding to the tractions applied to the fault surface to generate the prescribed slip.

1.3.1 Notation

- Unknowns
  - $\vec{u}$ Displacement field
  - $\vec{v}$ Velocity field (if including inertial term)
  - $\vec{\lambda}$ Lagrange multiplier field
- Derived quantities
  - $\boldsymbol{\sigma}$ Stress tensor
  - $\boldsymbol{\epsilon}$ Strain tensor
- Constitutive parameters
  - μ Shear modulus
  - K Bulk modulus
  - ρ Density
- Source terms
  - $\vec{f}$ Body force per unit volume, for example $\rho\vec{g}$
  - $\vec{d}$ Slip vector field on the fault corresponding to a jump in the displacement field across the fault

1.3.2 Neglecting Inertia

If we neglect the intertial term, then time dependence only arises from history-dependent constitutive equations and boundary conditions. Considering the displacement $\vec{u}$ and Lagrange multiplier $\vec{\lambda}$ fields as the unknowns, we have

$$\vec{s}^T = (\vec{u} \quad \vec{\lambda})^T, \tag{1.88}$$

$$\vec{0} = \vec{f}(\vec{x}, t) + \boldsymbol{\nabla}\cdot\boldsymbol{\sigma}(\vec{u}) \text{ in } \Omega, \tag{1.89}$$

$$\boldsymbol{\sigma}\cdot\vec{n} = \vec{\tau}(\vec{x}, t) \text{ on } \Gamma_\tau, \tag{1.90}$$

$$\vec{u} = \vec{u}_0(\vec{x}, t) \text{ on } \Gamma_u, \tag{1.91}$$

$$\vec{0} = \vec{d}(\vec{x}, t) - \vec{u}^+(\vec{x}, t) + \vec{u}^-(\vec{x}, t) \text{ on } \Gamma_f, \tag{1.92}$$

$$\boldsymbol{\sigma}\cdot\vec{n} = -\vec{\lambda}(\vec{x}, t) \text{ on } \Gamma_{f^+}, \tag{1.93}$$

$$\boldsymbol{\sigma}\cdot\vec{n} = +\vec{\lambda}(\vec{x}, t) \text{ on } \Gamma_{f^-}. \tag{1.94}$$

We create the weak form by taking the dot product with the trial function $\vec{\psi}^u_{trial}$ or $\vec{\psi}^\lambda_{trial}$ and integrating over the domain:

$$0 = \int_\Omega \vec{\psi}^u_{trial}\cdot\left(\vec{f}(t) + \boldsymbol{\nabla}\cdot\boldsymbol{\sigma}(\vec{u})\right) d\Omega, \tag{1.95}$$

$$0 = \int_{\Gamma_f} \vec{\psi}^\lambda_{trial}\cdot\left(\vec{d}(\vec{x}, t) - \vec{u}^+(\vec{x}, t) + \vec{u}^-(\vec{x}, t)\right) d\Gamma. \tag{1.96}$$

Using the divergence theorem and incorporating the Neumann boundary and fault interface conditions, we can rewrite the first equation as

$$0 = \int_\Omega \vec{\psi}^u_{trial}\cdot\vec{f}(t) + \nabla\vec{\psi}^u_{trial} : -\boldsymbol{\sigma}(\vec{u})\, d\Omega + \int_{\Gamma_\tau} \vec{\psi}^u_{trial}\cdot\vec{\tau}(\vec{x}, t)\, d\Gamma + \int_{\Gamma_{f^+}} \vec{\psi}^u_{trial}\cdot -\vec{\lambda}(\vec{x}, t)\, d\Gamma + \int_{\Gamma_{f^-}} \vec{\psi}^u_{trial}\cdot +\vec{\lambda}(\vec{x}, t)\, d\Gamma. \tag{1.97}$$

In practice we integrate over the fault surface by integrating over the faces of the cohesive cells on the positive and negative sides of the fault. Breaking up the integral over the fault surface in the second equation into integrals over the positive and negative sides of the fault, we have

$$0 = \int_{\Gamma_{f^+}} \vec{\psi}^\lambda_{trial}\cdot\left(\frac{1}{2}\vec{d}(\vec{x}, t) - \vec{u}(\vec{x}, t)\right) d\Gamma + \int_{\Gamma_{f^-}} \vec{\psi}^\lambda_{trial}\cdot\left(\frac{1}{2}\vec{d}(\vec{x}, t) + \vec{u}(\vec{x}, t)\right) d\Gamma. \tag{1.98}$$

Identifying $F(t,s,\dot{s})$ and $G(t,s)$, we have

$$F^u(t,s,\dot{s}) = 0, \tag{1.99}$$

$$F^\lambda(t,s,\dot{s}) = 0, \tag{1.100}$$

$$\begin{aligned} G^u(t,s) = &\int_\Omega \vec{\psi}^u_{trial} \cdot \underbrace{\vec{f}(\vec{x},t)}_{g^u_0} + \nabla\vec{\psi}^u_{trial} : \underbrace{-\boldsymbol{\sigma}(\vec{u})}_{g^u_1} \, d\Omega \\ &+ \int_{\Gamma_\tau} \vec{\psi}^u_{trial} \cdot \underbrace{\vec{\tau}(\vec{x},t)}_{g^u_0} \, d\Gamma \\ &+ \int_{\Gamma_{f^+}} \vec{\psi}^u_{trial} \cdot \underbrace{-\vec{\lambda}(\vec{x},t)}_{g^u_0} \, d\Gamma + \int_{\Gamma_{f^-}} \vec{\psi}^u_{trial} \cdot \underbrace{+\vec{\lambda}(\vec{x},t)}_{g^u_0} \, d\Gamma, \end{aligned} \tag{1.101}$$

$$G^\lambda(t,s) = \int_{\Gamma_{f^+}} \vec{\psi}^\lambda_{trial} \cdot \underbrace{\left(\frac{1}{2}\vec{d}(\vec{x},t) - \vec{u}(\vec{x},t)\right)}_{g^\lambda_0} d\Gamma + \int_{\Gamma_{f^-}} \vec{\psi}^\lambda_{trial} \cdot \underbrace{\left(\frac{1}{2}\vec{d}(\vec{x},t) + \vec{u}(\vec{x},t)\right)}_{g^\lambda_0} d\Gamma. \tag{1.102}$$

Note that we have multiple g^u_0 functions, each associated with an integral over a different domain or boundary. The integral over the domain Ω is implemented by the material, the integral over the boundary Γ_τ is implemented by the Neumann boundary condition, and the integrals over the interfaces Γ_{f^+} and Γ_{f^-} are implemented by the fault (cohesive cells).

1.3.2.1 Jacobians

With the solution composed of the displacement and Lagrange multiplier fields, the Jacobians are:

$$J^{uu}_F = \mathbf{0} \tag{1.103}$$

$$J^{\lambda\lambda}_F = \mathbf{0} \tag{1.104}$$

$$J^{uu}_G = \frac{\partial G^u}{\partial u} = \int_\Omega \nabla\vec{\psi}^u_{trial} : \frac{\partial}{\partial u}(-\boldsymbol{\sigma}) \, d\Omega = \int_\Omega \nabla\vec{\psi}^u_{trial} : -\boldsymbol{C} : \frac{1}{2}(\nabla + \nabla^T)\vec{\psi}^u_{basis} \, d\Omega = \int_\Omega \psi^v_{trial\,i,k} \underbrace{\left(-C_{ikjl}\right)}_{J^{uu}_{g3}} \psi^u_{basis\,j,l} \, d\Omega \tag{1.105}$$

$$\begin{aligned} J^{u\lambda}_G = \frac{\partial G^u}{\partial \lambda} &= \int_{\Gamma_{f^+}} \vec{\psi}^u_{trial} \cdot \frac{\partial}{\partial \lambda}(-\vec{\lambda}) \, d\Gamma + \int_{\Gamma_{f^-}} \vec{\psi}^u_{trial} \cdot \frac{\partial}{\partial \lambda}(+\vec{\lambda}) \, d\Gamma \\ &= \int_{\Gamma_{f^+}} \psi^u_{trial\,i} \underbrace{-1}_{J^{u\lambda}_{g0}} \psi^\lambda_{basis\,j} \, d\Gamma + \int_{\Gamma_{f^-}} \psi^u_{trial\,i} \underbrace{+1}_{J^{u\lambda}_{g0}} \psi^\lambda_{basis\,j} \, d\Gamma \end{aligned} \tag{1.106}$$

$$\begin{aligned} J^{\lambda u}_G = \frac{\partial G^\lambda}{\partial u} &= \int_{\Gamma_{f^+}} \vec{\psi}^\lambda_{trial} \cdot \frac{\partial}{\partial u}\left(\frac{1}{2}\vec{d}(\vec{x},t) - \vec{u}(\vec{x},t)\right) d\Gamma + \int_{\Gamma_{f^-}} \vec{\psi}^\lambda_{trial} \cdot \frac{\partial}{\partial u}\left(\frac{1}{2}\vec{d}(\vec{x},t) + \vec{u}(\vec{x},t)\right) d\Gamma \\ &= \int_{\Gamma_{f^+}} \psi^\lambda_{trial\,i} \underbrace{(-1)}_{J^{\lambda u}_{g0}} \psi^u_{basis\,j} \, d\Gamma + \int_{\Gamma_{f^-}} \psi^\lambda_{trial\,i} \underbrace{(+1)}_{J^{\lambda u}_{g0}} \psi^u_{basis\,j} \, d\Gamma \end{aligned} \tag{1.107}$$

$$J^{\lambda\lambda}_G = 0 \tag{1.108}$$

1.3.3 Including Inertia

For convenience we cast the elasticity equation in the form of a first order equation by considering the displacement $\vec{u}$, velocity $\vec{v}$, and Lagrange multipliers $\vec{\lambda}$ as unknowns,

$$\vec{s}^T = (\vec{u} \quad \vec{v} \quad \vec{\lambda})^T, \tag{1.109}$$

$$\frac{\partial \vec{u}}{\partial t} = \vec{v}, \tag{1.110}$$

$$\rho \frac{\partial \vec{v}}{\partial t} = \vec{f}(\vec{x}, t) + \boldsymbol{\nabla} \cdot \boldsymbol{\sigma}(\vec{u}) \text{ in } \Omega, \tag{1.111}$$

$$\boldsymbol{\sigma} \cdot \vec{n} = \vec{\tau}(\vec{x}, t) \text{ on } \Gamma_\tau, \tag{1.112}$$

$$\vec{u} = \vec{u}_0(\vec{x}, t) \text{ on } \Gamma_u, \tag{1.113}$$

$$\vec{0} = \vec{d}(\vec{x}, t) - \vec{u}^+(\vec{x}, t) + \vec{u}^-(\vec{x}, t) \text{ on } \Gamma_f, \tag{1.114}$$

$$\boldsymbol{\sigma} \cdot \vec{n} = -\vec{\lambda}(\vec{x}, t) \text{ on } \Gamma_{f^+}, \tag{1.115}$$

$$\boldsymbol{\sigma} \cdot \vec{n} = +\vec{\lambda}(\vec{x}, t) \text{ on } \Gamma_{f^-}. \tag{1.116}$$

For trial functions $\vec{\psi}^u_{trial}$, $\vec{\psi}^v_{trial}$, and $\vec{\psi}^\lambda_{trial}$ we write the weak form as

$$\int_\Omega \vec{\psi}^u_{trial} \cdot \left(\frac{\partial \vec{u}}{\partial t} \right) d\Omega = \int_\Omega \vec{\psi}^u_{trial} \cdot \vec{v} \, d\Omega, \tag{1.117}$$

$$\int_\Omega \vec{\psi}^v_{trial} \cdot \left(\rho \frac{\partial \vec{v}}{\partial t} \right) d\Omega = \int_\Omega \vec{\psi}^v_{trial} \cdot \left(\vec{f}(\vec{x}, t) + \boldsymbol{\nabla} \cdot \boldsymbol{\sigma}(\vec{u}) \right) d\Omega, \tag{1.118}$$

$$0 = \int_{\Gamma_{f^+}} \vec{\psi}^\lambda_{trial} \cdot \left(\frac{1}{2} \vec{d}(\vec{x}, t) - \vec{u}(\vec{x}, t) \right) d\Gamma + \int_{\Gamma_{f^-}} \vec{\psi}^\lambda_{trial} \cdot \left(\frac{1}{2} \vec{d}(\vec{x}, t) + \vec{u}(\vec{x}, t) \right) d\Gamma. \tag{1.119}$$

Using the divergence theorem and incorporating the Neumann boundary and fault interface conditions, we can rewrite the second equation as

$$\begin{aligned} \int_\Omega \vec{\psi}^v_{trial} \cdot \left(\rho \frac{\partial \vec{v}}{\partial t} \right) d\Omega = \int_\Omega \vec{\psi}^v_{trial} \cdot \vec{f}(\vec{x}, t) + \nabla \vec{\psi}^v_{trial} : -\boldsymbol{\sigma}(\vec{u}) \, d\Omega + \int_{\Gamma_\tau} \vec{\psi}^v_{trial} \cdot \vec{\tau}(\vec{x}, t) \, d\Gamma \\ + \int_{\Gamma_{f^+}} \vec{\psi}^v_{trial} \cdot -\vec{\lambda}(\vec{x}, t) \, d\Gamma + \int_{\Gamma_{f^-}} \vec{\psi}^v_{trial} \cdot +\vec{\lambda}(\vec{x}, t) \, d\Gamma. \end{aligned} \tag{1.120}$$

1.3.4 Explicit Time Stepping

Recall that for explicit time stepping we want $F(t, s, \dot{s}) = \dot{s}$. However, our fault interface constraint equation cannot be put into this form. Nevertheless, we put the first two equations in this form. The resulting equation will be a differential algebraic equation (DAE). Our system of equations to solve is

$$\int_\Omega \vec{\psi}^u_{trial} \cdot \frac{\partial \vec{u}}{\partial t} \, d\Omega = \int_\Omega \vec{\psi}^u_{trial} \cdot \vec{v} \, d\Omega, \tag{1.121}$$

$$\begin{aligned} \int_\Omega \vec{\psi}^v_{trial} \cdot \frac{\partial \vec{v}}{\partial t} \, d\Omega = \frac{1}{\int_\Omega \vec{\psi}^v_{trial} \cdot \rho \vec{\psi}^v_{basis} \, d\Omega} \left(\int_\Omega \vec{\psi}^v_{trial} \cdot \vec{f}(\vec{x}, t) + \nabla \vec{\psi}^u_{trial} : -\boldsymbol{\sigma}(\vec{u}) \, d\Omega + \int_{\Gamma_\tau} \vec{\psi}^v_{trial} \cdot \vec{\tau}(\vec{x}, t) \, d\Gamma \right. \\ \left. + \int_{\Gamma_{f^+}} \vec{\psi}^v_{trial} \cdot -\vec{\lambda}(\vec{x}, t) \, d\Gamma + \int_{\Gamma_{f^-}} \vec{\psi}^v_{trial} \cdot +\vec{\lambda}(\vec{x}, t) \, d\Gamma \right), \end{aligned} \tag{1.122}$$

$$0 = \int_{\Gamma_{f^+}} \vec{\psi}^\lambda_{trial} \cdot \left(\frac{1}{2} \vec{d}(\vec{x}, t) - \vec{u}(\vec{x}, t) \right) d\Gamma + \int_{\Gamma_{f^-}} \vec{\psi}^\lambda_{trial} \cdot \left(\frac{1}{2} \vec{d}(\vec{x}, t) + \vec{u}(\vec{x}, t) \right) d\Gamma. \tag{1.123}$$

Identifying $F(t,s,\dot{s})$ and $G(t,s)$, we have

$$F^u(t,s,\dot{s}) = \int_\Omega \vec{\psi}^u_{trial} \cdot \underbrace{\frac{\partial \vec{u}}{\partial t}}_{f_o^u} \, d\Omega, \tag{1.124}$$

$$F^v(t,s,\dot{s}) = \int_\Omega \vec{\psi}^v_{trial} \cdot \underbrace{\frac{\partial \vec{v}}{\partial t}}_{f_0^v} \, d\Omega, \tag{1.125}$$

$$F^\lambda(t,s,\dot{s}) = 0, \tag{1.126}$$

$$G^u(t,s) = \int_\Omega \vec{\psi}^u_{trial} \cdot \underbrace{\vec{v}}_{g_0^u} \, d\Omega, \tag{1.127}$$

$$\begin{aligned} G^v(t,s) = \frac{1}{\int_\Omega \vec{\psi}^v_{trial} \cdot \underbrace{\rho}_{J^{*vv}_{f0}} \vec{\psi}^v_{basis} \, d\Omega} \Bigg(& \int_\Omega \vec{\psi}^v_{trial} \cdot \underbrace{\vec{f}(t)}_{g_0^v} + \nabla \vec{\psi}^v_{trial} : \underbrace{-\boldsymbol{\sigma}(\vec{u})}_{g_1^v} \, d\Omega + \int_{\Gamma_\tau} \vec{\psi}^v_{trial} \cdot \underbrace{\vec{\tau}(\vec{x},t)}_{g_0^v} \, d\Gamma \\ & + \int_{\Gamma_{f^+}} \vec{\psi}^v_{trial} \cdot \underbrace{-\vec{\lambda}(\vec{x},t)}_{g_0^v} \, d\Gamma + \int_{\Gamma_{f^-}} \vec{\psi}^v_{trial} \cdot \underbrace{+\vec{\lambda}(\vec{x},t)}_{g_0^v} \, d\Gamma \Bigg), \end{aligned} \tag{1.128}$$

$$G^\lambda(t,s) = \int_{\Gamma_{f^+}} \vec{\psi}^\lambda_{trial} \cdot \underbrace{\left(\frac{1}{2}\vec{d}(\vec{x},t) - \vec{u}(\vec{x},t) \right)}_{g_0^\lambda} d\Gamma + \int_{\Gamma_{f^-}} \vec{\psi}^\lambda_{trial} \cdot \underbrace{\left(\frac{1}{2}\vec{d}(\vec{x},t) + \vec{u}(\vec{x},t) \right)}_{g_0^\lambda} d\Gamma. \tag{1.129}$$

Note that we have multiple g_0^u functions, each associated with an integral over a different domain or boundary. The integral over the domain Ω is implemented by the material, the integral over the boundary Γ_τ is implemented by the Neumann boundary condition, and the integrals over the interfaces Γ_{f^+} and Γ_{f^-} are implemented by the fault (cohesive cells).

1.3.4.1 Jacobians

With a differential algebraic equation, we only need to specify the Jacobians for the LHS.

$$J_F^{uu} = \frac{\partial F^u}{\partial u} + t_{shift} \frac{\partial F^u}{\partial \dot{u}} = \int_\Omega \psi^u_{trial\,i} \underbrace{t_{shift}}_{J^{uu}_{f0}} \psi^u_{basis\,j} \, d\Omega, \tag{1.130}$$

$$J_F^{vv} = \frac{\partial F^v}{\partial v} + t_{shift} \frac{\partial F^v}{\partial \dot{v}} = \int_\Omega \psi^v_{trial\,i} \underbrace{t_{shift}}_{J^{vv}_{f0}} \psi^v_{basis\,j} \, d\Omega, \tag{1.131}$$

$$J_F^{\lambda\lambda} = \frac{\partial F^\lambda}{\partial \lambda} + t_{shift} \frac{\partial F^\lambda}{\partial \dot{\lambda}} = \underbrace{\mathbf{0}}_{J^{\lambda\lambda}_{f0}} \tag{1.132}$$

1.4 Incompressible Isotropic Elasticity with Infinitesimal Strain (Bathe) and No Faults

TODO @charles

This section needs to be updated. Break up formulation into cases without inertial terms (solution includes displacement and pressure fields) and with inertial terms (solution includes displacement, velocity, and pressure fields).

Building from the elasticity equation (equations 1.18 and 1.19), we consider an incompressible material. As the bulk modulus (K) approaches infinity, the volumetric strain ($\text{Tr}(\epsilon)$) approaches zero and the pressure remains finite, $p = -K\,\text{Tr}(\epsilon)$. We consider pressure p as an independent variable and decompose the stress into the pressure and deviatoric components. This leads to the following set of governing equations:

$$\vec{s}^T = \left(\vec{u}\ \vec{v}\ p\right)^T, \tag{1.133}$$

$$\frac{\partial \vec{u}(t)}{\partial t} = \vec{v}(t), \tag{1.134}$$

$$\rho \frac{\partial \vec{v}(t)}{\partial t} = \vec{f}(t) + \boldsymbol{\nabla} \cdot \boldsymbol{\sigma}(\vec{u}) \text{ in } \Omega, \tag{1.135}$$

$$0 = \vec{\nabla} \cdot \vec{u} + \frac{p}{K}, \tag{1.136}$$

$$\boldsymbol{\sigma} \cdot \vec{n} = \vec{\tau} \text{ on } \Gamma_\tau, \tag{1.137}$$

$$\vec{u} = \vec{u}_0 \text{ on } \Gamma_u. \tag{1.138}$$

Using trial functions $\vec{\psi}^u_{trial}$, $\vec{\psi}^v_{trial}$, and ψ^p_{trial} and incorporating the Neumann boundary conditions, we write the weak form as

$$\int_\Omega \vec{\psi}^u_{trial} \cdot \left(\frac{\partial \vec{u}(t)}{\partial t}\right) d\Omega = \int_\Omega \vec{\psi}^u_{trial} \cdot \vec{v}(t)\, d\Omega, \tag{1.139}$$

$$\int_\Omega \vec{\psi}^v_{trial} \cdot \left(\rho \frac{\partial \vec{v}(t)}{\partial t}\right) d\Omega = \int_\Omega \vec{\psi}^v_{trial} \cdot \vec{f}(t) + \nabla \vec{\psi}^v_{trial} : \left(p\boldsymbol{I} - \boldsymbol{\sigma}^{dev}(\vec{u})\right) d\Omega + \int_{\Gamma_\tau} \vec{\psi}^v_{trial} \cdot \vec{\tau}(t)\, d\Gamma, \quad 0 = \int_\Omega \psi^p_{trial} \cdot \left(\vec{\nabla} \cdot \vec{u} + \frac{p}{K}\right) d\Omega. \tag{1.140}$$

1.4.1 Implicit Time Stepping

The resulting system of equations to solve is

$$\int_\Omega \vec{\psi}^u_{trial} \cdot \underbrace{\left(\frac{\partial \vec{u}(t)}{\partial t}\right)}_{f^u_0} d\Omega = \int_\Omega \vec{\psi}^u_{trial} \cdot \underbrace{\vec{v}(t)}_{g^u_0} d\Omega, \tag{1.141}$$

$$\int_\Omega \vec{\psi}^v_{trial} \cdot \underbrace{\left(\rho \frac{\partial \vec{v}(t)}{\partial t}\right)}_{f^v_0} d\Omega = \int_\Omega \vec{\psi}^v_{trial} \cdot \underbrace{\vec{f}(t)}_{g^v_0} + \nabla \vec{\psi}^v_{trial} : \underbrace{p\boldsymbol{I} - \boldsymbol{\sigma}^{dev}(\vec{u})}_{g^v_1} d\Omega + \int_{\Gamma_\tau} \vec{\psi}^v_{trial} \cdot \underbrace{\vec{\tau}(t)}_{g^v_0} d\Gamma, \tag{1.142}$$

$$0 = \int_\Omega \psi^p_{trial} \cdot \underbrace{\left(\vec{\nabla} \cdot \vec{u} + \frac{p}{K}\right)}_{g^p_0} d\Omega. \tag{1.143}$$

1.4.1.1 Jacobians

With three fields we have nine Jacobians for the LHS and RHS associated with the coupling of the three fields.

$$J_F^{uu} = \frac{\partial F^u}{\partial u} + t_{shift}\frac{\partial F^u}{\partial \dot{u}} = \int_\Omega \vec{\psi}^u_{trial}\cdot t_{shift}\vec{\psi}^u_{basis}\,d\Omega = \int_\Omega \psi^u_{trial\,i}\underbrace{t_{shift}\delta_{ij}}_{J^{uu}_{f0}}\psi^u_{basis\,j}\,d\Omega \tag{1.144}$$

$$J_F^{uv} = \frac{\partial F^u}{\partial v} + t_{shift}\frac{\partial F^u}{\partial \dot{v}} = \mathbf{0} \tag{1.145}$$

$$J_F^{up} = \frac{\partial F^u}{\partial p} + t_{shift}\frac{\partial F^u}{\partial \dot{p}} = \mathbf{0} \tag{1.146}$$

$$J_F^{vu} = \frac{\partial F^v}{\partial u} + t_{shift}\frac{\partial F^v}{\partial \dot{u}} = \mathbf{0} \tag{1.147}$$

$$J_F^{vv} = \frac{\partial F^v}{\partial v} + t_{shift}\frac{\partial F^v}{\partial \dot{v}} = \int_\Omega \vec{\psi}^v_{trial}\cdot t_{shift}\,\rho\,\vec{\psi}^v_{basis}\,d\Omega = \int_\Omega \psi^v_{trial\,i}\underbrace{t_{shift}\,\rho\,\delta_{ij}}_{J^{vv}_{f0}}\psi^v_{basis\,j}\,d\Omega \tag{1.148}$$

$$J_F^{vp} = \frac{\partial F^v}{\partial p} + t_{shift}\frac{\partial F^v}{\partial \dot{p}} = \mathbf{0} \tag{1.149}$$

$$J_F^{pu} = \frac{\partial F^p}{\partial u} + t_{shift}\frac{\partial F^p}{\partial \dot{u}} = \mathbf{0} \tag{1.150}$$

$$J_F^{pv} = \frac{\partial F^p}{\partial v} + t_{shift}\frac{\partial F^p}{\partial \dot{v}} = \mathbf{0} \tag{1.151}$$

$$J_F^{pp} = \frac{\partial F^p}{\partial p} + t_{shift}\frac{\partial F^p}{\partial \dot{p}} = \mathbf{0} \tag{1.152}$$

$$J_G^{uu} = \frac{\partial G^u}{\partial u} = \mathbf{0} \tag{1.153}$$

$$J_G^{uv} = \frac{\partial G^u}{\partial v} = \int_\Omega \vec{\psi}^u_{trial}\cdot\vec{\psi}^v_{basis}\,d\Omega = \int_\Omega \psi^u_{trial\,i}\underbrace{\delta_{ij}}_{J^{uv}_{g0}}\psi^v_{basis\,j}\,d\Omega \tag{1.154}$$

$$J_G^{up} = \frac{\partial G^u}{\partial p} = \mathbf{0} \tag{1.155}$$

$$J_G^{vu} = \frac{\partial G^v}{\partial u} = \int_\Omega \nabla\vec{\psi}^u_{trial} : \frac{\partial}{\partial u}(-\boldsymbol{\sigma}^{dev})\,d\Omega = \int_\Omega \nabla\vec{\psi}^u_{trial} : -\boldsymbol{C} : \frac{1}{2}(\nabla+\nabla^T)\vec{\psi}^u_{basis}\,d\Omega = \int_\Omega \psi^v_{trial\,i,k}\underbrace{\left(-C^{dev}_{ikjl}\right)}_{J^{vu}_{g3}}\psi^u_{basis\,j,l}\,d\Omega \tag{1.156}$$

$$J_G^{vv} = \frac{\partial G^v}{\partial v} = \mathbf{0} \tag{1.157}$$

$$J_G^{vp} = \frac{\partial G^v}{\partial p} = \int_\Omega \nabla\vec{\psi}^v_{trial} : \boldsymbol{I}\psi^p_{basis}\,d\Omega = \int_\Omega \psi^v_{trial\,i,j} : \underbrace{\delta_{ij}}_{J^{vp}_{g2}}\psi^p_{basis}\,d\Omega \tag{1.158}$$

$$J_G^{pu} = \frac{\partial G^p}{\partial u} = \int_\Omega \psi^p_{trial}(\vec{\nabla}\cdot\vec{\psi}^u_{basis})\,d\Omega = \int_\Omega \psi^p_{trial}\underbrace{\delta_{ij}}_{J^{pu}_{g1}}\psi^u_{basis\,i,j})\,d\Omega \tag{1.159}$$

$$J_G^{pv} = \frac{\partial G^p}{\partial v} = \mathbf{0} \tag{1.160}$$

$$J_G^{pp} = \frac{\partial G^p}{\partial p} = \int_v \psi^p_{trial}\underbrace{\frac{1}{K}}_{J^{pu}_{g1}}\psi^p_{basis}\,d\Omega \tag{1.161}$$

1.4.2 Explicit Time Stepping

Recall that explicit time stepping requires $F(t,s,\dot{s}) = \dot{s}$. As a result, we use the time derivative of the pressure equation. We write $F^*(t,s,\dot{s}) = \dot{s}$ and $G^*(t,s) = M^{-1}G(t,s)$ and we do not provide functions for f_0 and f_1. Thus, we our system of equations to solve is

$$\int_\Omega \vec{\psi}^u_{trial} \cdot \left(\frac{\partial \vec{u}(t)}{\partial t} \right) d\Omega = (M^{uu})^{-1} \left(\int_\Omega \vec{\psi}^u_{trial} \cdot \underbrace{\vec{v}(t)}_{g_0^u} d\Omega \right), \tag{1.162}$$

$$\int_\Omega \vec{\psi}^v_{trial} \cdot \left(\frac{\partial \vec{v}(t)}{\partial t} \right) d\Omega = (M^{vv})^{-1} \left(\int_\Omega \vec{\psi}^v_{trial} \cdot \underbrace{\vec{f}(t)}_{g_0^v} + \nabla \vec{\psi}^u_{trial} : \underbrace{p\boldsymbol{I} - \boldsymbol{\sigma}^{dev}(\vec{u})}_{g_1^v} d\Omega + \int_{\Gamma_\tau} \vec{\psi}^u_{trial} \cdot \underbrace{\vec{\tau}(t)}_{g_0^v} d\Gamma \right), \tag{1.163}$$

$$\int_\Omega \psi^p_{trial} \cdot \frac{\partial p}{\partial t} d\Omega = (M^{pp})^{-1} \left(\int_\Omega \psi^p_{trial} \cdot \underbrace{\vec{\nabla} \cdot -\vec{v}}_{g_0^p} d\Omega \right), \tag{1.164}$$

where

$$M^{uu} = I, \tag{1.165}$$

$$M^{vv} = Lump(\int_\Omega \vec{\psi}^v_{trial} \cdot J^{vv}_{f0} \cdot \vec{\psi}^v_{basis} d\Omega), \quad J^{vv}_{f0} = \rho, \tag{1.166}$$

$$M^{pp} = \int_\Omega \psi^p_{trial} \cdot J^{pp}_{f0} \cdot \psi^p_{basis} d\Omega, \quad J^{pp}_{f0} = \frac{1}{K}, \tag{1.167}$$

$$\tag{1.168}$$

and we refer to M as the LHS (or I) Jacobian for explicit time stepping.

1.5 Poroelasticity with Infinitesimal Strain and No Faults or Inertia

> **TODO @Hackathon**
>
> Before the hackathon, please finish filling in the details of this formulation. Write out the point-wise functions for the residual. Derive the point-wise functions for the Jacobian. If you want to use different equations and/or assumptions, please complete this formulation first, and then derive the other formulation with alternative assumptions and/or different physics.

Formulation based on Zheng et al. and Detournay and Cheng (1993).

In this poroelasticity formulation we assume a compressible fluid completely saturates a porous solid undergoing infinitesimal strain. We neglect the inertial effects and do not consider faults.

We begin with the elasticity equilibrium equation neglecting the inertial term,

$$-\vec{f}(\vec{x},t) - \boldsymbol{\nabla} \cdot \boldsymbol{\sigma}(\vec{u}, p_f) = \vec{0} \text{ in } \Omega, \tag{1.169}$$

$$\boldsymbol{\sigma} \cdot \vec{n} = \vec{\tau}(\vec{x},t) \text{ on } \Gamma_\tau, \tag{1.170}$$

$$\vec{u} = \vec{u}_0(\vec{x},t) \text{ on } \Gamma_u, \tag{1.171}$$

where $\vec{u}$ is the displacement vector, $\vec{f}$ is the body force vector, $\boldsymbol{\sigma}$ is the Cauchy stress tensor, and t is time. We specify tractions $\vec{\tau}$ on boundary Γ_τ, and displacements $\vec{u}_0$ on boundary Γ_u. If gravity is included in the problem, then usually $\vec{f} = \rho\vec{g}$, where ρ

is the average density $\rho = (1-\phi)\rho_s + \phi\rho_f$, ϕ is the porosity of the solid, ρ_s is the density of the solid, and ρ_f is the density of the fluid.

Enforcing mass balance of the fluid gives

$$\frac{\partial \zeta(\vec{u}, p_f)}{\partial t} + \nabla \cdot \vec{q}(p_f) = \gamma(\vec{x}, t) \text{ in } \Omega, \tag{1.172}$$

$$\vec{q} \cdot \vec{n} = q_0(\vec{x}, t) \text{ on } \Gamma_q, \tag{1.173}$$

$$p_f = p_0(\vec{x}, t) \text{ on } \Gamma_p, \tag{1.174}$$

where ζ is the variation in fluid content, $\vec{q}$ is the rate of fluid volume crossing a unit area of the porous solid, γ is the rate of injected fluid per unit volume of the porous solid, q_0 is the outward fluid velocity normal to the boundary Γ_q, and p_0 is the fluid pressure on boundary Γ_p.

We require the fluid flow to follow Darcy's law (Navier-Stokes equation neglecting inertial effects),

$$\vec{q}(p_f) = -\kappa(\nabla p_f - \vec{f}_f), \tag{1.175}$$

$$\kappa = \frac{k}{\eta_f} \tag{1.176}$$

where κ is the permeability coefficient (Darcy conductivity), k is the intrinsic permeability, η_f is the viscosity of the fluid, p_f is the fluid pressure, and $\vec{f}_f$ is the body force in the fluid. If gravity is included in a problem, then usually $\vec{f}_f = \rho_f \vec{g}$, where ρ_f is the density of the fluid and $\vec{g}$ is the gravitational acceleration vector.

We assume linear elasticity for the solid phase, so the constitutive behavior can be expressed as

$$\boldsymbol{\sigma}(\vec{u}, p_f) = \boldsymbol{C} : \boldsymbol{\epsilon} + \alpha p_f \boldsymbol{I}, \tag{1.177}$$

where $\boldsymbol{\sigma}$ is the stress tensor, $\boldsymbol{C}$ is the tensor of elasticity constants, α is the Biot coefficient (effective stress coefficient), $\boldsymbol{\epsilon}$ is the strain tensor, and $\boldsymbol{I}$ is the identity tensor.

For the constitutive behavior of the fluid, we use the volumetric strain to couple the fluid- solid behavior,

$$\zeta(\vec{u}, p_f) = \alpha \operatorname{Tr}(\boldsymbol{\epsilon}) + \frac{p_f}{M}, \tag{1.178}$$

$$\frac{1}{M} = \frac{\alpha - \phi}{K_s} + \frac{\phi}{K_f}, \tag{1.179}$$

where $1/M$ is the specific storage coefficient at constant strain, K_s is the bulk modulus of the solid, and K_f is the bulk modulus of the fluid. We can write the trace of the strain tensor as the dot product of the gradient and displacement field, so we have

$$\zeta(\vec{u}, p_f) = \alpha(\nabla \cdot \vec{u}) + \frac{p_f}{M}. \tag{1.180}$$

1.5.1 Notation

- Unknowns
  - $\vec{u}$ Displacement field
  - p_f Fluid pressure
- Derived quantities
  - $\boldsymbol{\sigma}$ Stress tensor
  - $\boldsymbol{\epsilon}$ Strain tensor
  - ζ Variation of fluid content; variation of fluid volumer per unit volume of porous material
  - q rate of fluid volume crossing a unit area of the porous solid; fluid flux
  - $1/M$ Specific storage coefficient at constant strain

κ permability coefficient; Darcy conductivity; $\kappa = k/\eta_f$

ρ Average density; $\rho = (1-\phi)\rho_s + \phi\rho_f$

- Constitutive parameters

  μ Shear modulus of solid

  K_s Bulk modulus of solid

  K_f Bulk modulus of fluid

  α Biot coefficient; effective stress coefficient

  k Intrinsic permeability

  η_f Fluid viscosity

  ρ_s Density of solid

  ρ_f Density of fluid

  ϕ Porosity; $\frac{V_p}{V}$ (V_p is the volume of the pore space)

- Source terms

  $\vec{f}$ Body force, for example $\rho\vec{g} = (1-\phi)\rho_s + \phi\rho_f$

  $\vec{f}_f$ Body force in fluid, for example $\rho_f\vec{g}$

  γ Source density; rate of injected fluid per unit volume of the porous solid

1.5.2 Neglecting Inertia

We consider the displacement $\vec{u}$, volumetric strain e, and fluid pressure p_f as unknowns,

$$\vec{s}^T = (\vec{u}e \quad p_f)^T, \tag{1.181}$$

$$\vec{0} = \vec{f}(\vec{x},t) + \boldsymbol{\nabla}\cdot\boldsymbol{\sigma}(\vec{u},p_f) \text{ in } \Omega, \tag{1.182}$$

$$\frac{\partial\zeta(e,p_f)}{\partial t} = \gamma(\vec{x},t) - \nabla\cdot\vec{q}(p_f) \text{ in } \Omega, \tag{1.183}$$

$$\bigtriangledown.\vec{u} = e \text{ in } \Omega, \tag{1.184}$$

$$\vec{q}(p_f) = -\kappa(\nabla p_f - \vec{f}_f), \tag{1.185}$$

$$\boldsymbol{\sigma}\cdot\vec{n} = \vec{\tau}(\vec{x},t) \text{ on } \Gamma_\tau, \tag{1.186}$$

$$\zeta(e,p_f) = \alpha e + \frac{p_f}{M}, \tag{1.187}$$

$$\vec{u} = \vec{u}_0(\vec{x},t) \text{ on } \Gamma_u, \tag{1.188}$$

$$\vec{q}\cdot\vec{n} = q_0(\vec{x},t) \text{ on } \Gamma_q, \tag{1.189}$$

$$p_f = p_0(\vec{x},t) \text{ on } \Gamma_p. \tag{1.190}$$

Using trial functions $\vec{\psi^u}_{trial}$, $\vec{\psi^e}_{trial}$, and $\vec{\psi^p}_{trial}$ and incorporating the Neumann boundary conditions, we write the weak form as

$$\vec{0} = \int_\Omega \vec{\psi}^u_{trial}.\vec{f}(\vec{x},t) + \bigtriangledown\vec{\psi}^u_{trial} : -\sigma(\vec{u},p_f)\ d\Omega + \int_{\Gamma_\tau} \vec{\psi}^u_{trial}.\vec{\tau}(\vec{x},t)\ d\Gamma, \tag{1.191}$$

$$\int_\Omega \psi^p_{trial}\frac{\partial\zeta(\vec{u},p_f)}{\partial t}\ d\Omega = \int_\Omega \psi^p_{trial}\gamma(\vec{x},t) + \bigtriangledown\psi^p_{trial}.\vec{q}(p_f)\ d\Omega + \int_{\Gamma_q} \psi^p_{trial}(-q_0(\vec{x},t))\ d\Gamma. \tag{1.192}$$

$$\vec{0} = \int_\Omega \psi^e_{trial}.(\bigtriangledown.\vec{u} - e)\ d\Omega \tag{1.193}$$

Identifying $F(t,s,\dot{s})$ and $G(t,s)$ we have

$$F^u(t,s,\dot{s}) = \vec{0} \tag{1.194}$$

$$F^e(t,s,\dot{s}) = \vec{0} \tag{1.195}$$

$$F^p(t,s,\dot{s}) = \int_\Omega \psi^p_{trial}(\frac{\partial \zeta(\vec{u},p_f)}{\partial t})\ d\Omega = \int_\Omega \psi^p_{trial}(\frac{\partial}{\partial t}(\alpha e + \frac{p_f}{M}))\ d\Omega \tag{1.196}$$

$$= \int_\Omega \psi^p_{trial}(\alpha \frac{\partial e}{\partial t} + \frac{1}{M}\frac{\partial p_f}{\partial t})\ d\Omega = \int_\Omega \psi^p_{trial}(f^p_0)\ d\Omega \tag{1.197}$$

$$G^u(t,s) = \int_\Omega \vec{\psi}^u_{trial}.(\vec{f}(\vec{x},t)) + \nabla\vec{\psi}^u_{trial} : (-\sigma(\vec{u},p_f))\ d\Omega + \int_{\Gamma_\tau} \vec{\psi}^u_{trial}.(\vec{\tau}(\vec{x},t))\ d\Gamma \tag{1.198}$$

$$= \int_\Omega \vec{\psi}^u_{trial}.(\vec{f}(\vec{x},t)) + \nabla\vec{\psi}^u_{trial} : (-(C:\varepsilon + \alpha p_f I))\ d\Omega + \int_{\Gamma_\tau} \vec{\psi}^u_{trial}.(\vec{\tau}(\vec{x},t))\ d\Gamma \tag{1.199}$$

$$= \int_\Omega \vec{\psi}^u_{trial}.(g^u_0) + \nabla\vec{\psi}^u_{trial} : (g^u_1)\ d\Omega + \int_{\Gamma_\tau} \vec{\psi}^u_{trial}.(g^u_0)\ d\Gamma \tag{1.200}$$

$$G^e(t,s) = \int_\Omega \psi^e_{trial}.(\nabla.\vec{u} - e)\ d\Omega = \int_\Omega \psi^e_{trial}.(g^e_0)\ d\Omega \tag{1.201}$$

$$G^p(t,s) = \int_\Omega \psi^p_{trial}(\gamma(\vec{x},t)) + \nabla\psi^p_{trial}.(\vec{q}(p_f))\ d\Omega + \int_{\Gamma_q} \psi^p_{trial}(-q_0(\vec{x},t))\ d\Gamma \tag{1.202}$$

$$= \int_\Omega \psi^p_{trial}(\gamma(\vec{x},t)) + \nabla\psi^p_{trial}.(-\kappa(\nabla p_f - \vec{f}_f))\ d\Omega + \int_{\Gamma_q} \psi^p_{trial}(-q_0(\vec{x},t))\ d\Gamma \tag{1.203}$$

$$= \int_\Omega \psi^p_{trial}(g^p_0) + \nabla\psi^p_{trial}.(g^p_1)\ d\Omega + \int_{\Gamma_q} \psi^p_{trial}(g^p_0)\ d\Gamma \tag{1.204}$$

1.5.2.1 Jacobians

With three fields we have nine Jacobians for each side of the equation associated with the coupling of the three fields,

$$J^{uu}_F = \frac{\partial F^u}{\partial u} + t_{shift}\frac{\partial F^u}{\partial \dot{u}} = 0 \tag{1.205}$$

$$J^{ue}_F = \frac{\partial F^u}{\partial e} + t_{shift}\frac{\partial F^u}{\partial \dot{e}} = 0 \tag{1.206}$$

$$J^{up}_F = \frac{\partial F^u}{\partial p} + t_{shift}\frac{\partial F^u}{\partial \dot{p}} = 0 \tag{1.207}$$

$$J^{eu}_F = \frac{\partial F^e}{\partial u} + t_{shift}\frac{\partial F^e}{\partial \dot{u}} = 0 \tag{1.208}$$

$$J^{ee}_F = \frac{\partial F^e}{\partial e} + t_{shift}\frac{\partial F^e}{\partial \dot{e}} = 0 \tag{1.209}$$

$$J^{ep}_F = \frac{\partial F^e}{\partial p} + t_{shift}\frac{\partial F^e}{\partial \dot{p}} = 0 \tag{1.210}$$

$$J^{pu}_F = \frac{\partial F^p}{\partial u} + t_{shift}\frac{\partial F^p}{\partial \dot{u}} = 0 \tag{1.211}$$

$$J^{pe}_F = \frac{\partial F^p}{\partial e} + t_{shift}\frac{\partial F^p}{\partial \dot{e}} = \int_\Omega \psi^p_{trial} t_{shift}\alpha\psi^e_{basis}\ d\Omega = \int_\Omega \psi^p_{trial}(t_{shift}\alpha)\psi^e_{basis}\ d\Omega = \int_\Omega \psi^p_{trial}(J^{pe}_{f0})\psi^e_{basis}\ d\Omega \tag{1.212}$$

$$J^{pp}_F = \frac{\partial F^p}{\partial p} + t_{shift}\frac{\partial F^p}{\partial \dot{p}} = \int_\Omega \psi^p_{trial}(t_{shift}\frac{1}{M})\psi^p_{basis}\ d\Omega = \int_\Omega \psi^p_{trial}(J^{pp}_{f0})\psi^p_{basis}\ d\Omega \tag{1.213}$$

$$J_G^{uu} = \frac{\partial G^u}{\partial u} = \int_\Omega \nabla \vec{\psi}^u_{trial} : \frac{\partial}{\partial u}(-\sigma(\vec{u}, p))\ d\Omega = \int_\Omega \nabla \vec{\psi}^u_{trial} : \frac{\partial}{\partial u}(-(C : \varepsilon + \alpha p I))\ d\Omega \tag{1.214}$$

$$= \int_\Omega \nabla \vec{\psi}^u_{trial} : -C : \frac{1}{2}(\nabla + \nabla^T)\vec{\psi}^u_{basis}\ d\Omega = \int_\Omega \psi^u_{triali,k}(-C_{ikjl})\psi^u_{basisj,l}\ d\Omega = \int_\Omega \psi^u_{triali,k}(J^{uu}_{g3})\psi^u_{basisj,l}\ d\Omega \tag{1.215}$$

$$J_G^{ue} = \frac{\partial G^u}{\partial e} = 0 \tag{1.216}$$

$$J_G^{up} = \frac{\partial G^u}{\partial p} = \int_\Omega \nabla \vec{\psi}^u_{trial} : \frac{\partial}{\partial p}(-\alpha p_f I)\ d\Omega = \int_\Omega \psi^u_{triali,j}(-\alpha\delta_{ij})\psi^p_{basis}\ d\Omega = \int_\Omega \psi^u_{triali,j}(J^{up}_{g2})\psi^p_{basis}\ d\Omega \tag{1.217}$$

$$J_G^{eu} = \frac{\partial G^e}{\partial u} = \int_\Omega \psi^e_{trial} \nabla . \vec{\psi}^u_{basis}\ d\Omega = \int_\Omega \psi^e_{trial}(\delta_{ij})\psi^u_{basisi,j}\ d\Omega = \int_\Omega \psi^e_{trial}(J^{eu}_{g1})\psi^u_{basisi,j}\ d\Omega \tag{1.218}$$

$$J_G^{ee} = \frac{\partial G^e}{\partial e} = \int_\Omega \psi^e_{trial}(-1)\psi^e_{basis}\ d\Omega = \int_\Omega \psi^e_{trial}(J^{ee}_{g0})\psi^e_{basis}\ d\Omega \tag{1.219}$$

$$J_G^{ep} = \frac{\partial G^e}{\partial p} = 0 \tag{1.220}$$

$$J_G^{pu} = \frac{\partial G^p}{\partial u} = 0 \tag{1.221}$$

$$J_G^{pe} = \frac{\partial G^p}{\partial e} = 0 \tag{1.222}$$

$$J_G^{pp} = \frac{\partial G^p}{\partial p} = \int_\Omega \nabla \psi^p_{trial} \cdot \frac{\partial}{\partial p}(-\kappa(\nabla p_f - \vec{f})\ d\Omega = \int_\Omega \nabla \psi^p_{trial}(-\kappa \nabla . \psi^p_{basis})\ d\Omega = \int_\Omega \psi^p_{trial,k}(-\kappa\delta_{kl})\psi^p_{basis,l}\ d\Omega \tag{1.223}$$

$$= \int_\Omega \psi^p_{trial,k}(J^{pp}_{g3})\psi^p_{basis,l}\ d\Omega \tag{1.224}$$

Chapter 2

Analytical Benchmark Solutions

2.1 Poroelastic Problems

2.1.1 Terzaghi's Consolidation Problem

2.1.1.1 Description

A one dimensional poroelastic analytical problem may be arrived at by reviewing the consolidation problem of Terzaghi[**?**] in light of Biot theory. An initially undisturbed soil layer of thickness L, resting upon a rigid, impermeable base is considered, and a constant load is applied to the top layer. All sides are held to be no flux boundaries, with the exception of the top, which is taken to represent drained conditions. At $t = 0^+$, the sample compacts, and the sudden applied load causes the pore pressure to spike to the undrained value, also known as the Skempton effect.

For an homogeneous, porous medium, one dimensional deformation may be expressed using the storage equation[**?**]:

$$\alpha \frac{\partial \epsilon}{\partial t} + S \frac{\partial p}{\partial t} = \frac{\kappa}{\gamma_f} \frac{\partial^2 p}{\partial z^2} \tag{2.1}$$

$$\sigma_{zz} = qH(t), \quad p = 0; \qquad \text{at} \quad z = 0; \quad t \geq 0 \tag{2.2}$$

$$u_z = 0, \quad \frac{\partial p}{\partial z} = 0; \qquad \text{at} \quad z = L; \quad t \geq 0 \tag{2.3}$$

$$p = 0; \qquad \text{at} \quad t = 0^-; \quad 0 \leq z \leq L \tag{2.4}$$

For one dimensional deformation, volume change is equivalent to vertical strain, and with the assumption of linear elastic behavior, the volumetric strain rate may be expressed as:

$$\frac{\partial \epsilon}{\partial t} = -m_v \frac{\partial \sigma^{'}_{zz}}{\partial t} = -m_v \left[\frac{\partial \sigma_{zz}}{\partial t} - \alpha \frac{\partial p}{\partial t} \right] \tag{2.5}$$

Combining the prior expression for volumetric strain rate and the storage equation, and after some algebra, the general differential equation for one dimensional consolidation may be written:

$$\frac{\partial p}{\partial t} = \frac{\alpha m_v}{S + \alpha^2 m_v} \frac{\partial \sigma_{zz}}{\partial t} + \frac{\kappa}{\gamma_f \left(S + \alpha^2 m_v \right)} \frac{\partial^2 p}{\partial z^2} \tag{2.6}$$

The problem dictates that at a time $t = 0$ a vertical loading of magnitude q is to be applied, and to continue for $t > 0$. Thus, for $t > 0$, the above reduces to:

$$\frac{\partial p}{\partial t} = \cancel{\frac{\alpha m_v}{S + \alpha^2 m_v} \frac{\partial \sigma_{zz}}{\partial t}} + \overbrace{\frac{\partial \sigma_{zz}}{\partial t} = 0} \underbrace{\frac{\kappa}{\gamma_f \left(S + \alpha^2 m_v\right)}}_{c_v} \frac{\partial^2 p}{\partial z^2} \tag{2.7}$$

where c_v may be defined as the consolidation coefficient.

The initial condition, as defined at the initial moment of loading, $t = 0^+$, is determined under the consideration that insufficient time has elapsed to allow for any fluid outflow, and thus is defined as:

$$p_0 = \frac{\alpha m_v}{S + \alpha^2 m_v} q \tag{2.8}$$

2.1.1.2 Solution

The pressure distribution over time may be represented with a closed form, analytical solution:

$$p(z,t) = \frac{4p_0}{\pi} \sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{2n-1} \cos\left[(2n-1)\frac{\pi z}{2h}\right] e^{-(2n-1)^2 \frac{\pi^2 c_v t}{4h^2}} \tag{2.9}$$

2.1.2 Mandel's Problem

The analytical solution for pore pressure, generalized for the case of compressible constituents may be expressed as[**?**]:

$$p = \frac{1}{A_2 - A_1} \sum_{i=1}^{\infty} D_i \left[\cos\left(\frac{\alpha_i x}{a}\right) - \cos\alpha_i\right] e^{\frac{-\alpha_i^2 c_x t}{a^2}} \tag{2.10}$$

2.1.3 Cryer's Problem

2.1.4 Flow to Wells

Chapter 3

Extending PyLith

This chapter will replace the current Extending PyLith chapter in the PyLith Manual.

3.1 Code Layout

The PyLith software suite is composed of a C++ library, Python modules, a Python application, and a few Python preprocessing and post-processing utilities.

3.1.1 Directory Structure

The C++, Python, and SWIG Python/C++ interface files all sit in different directories. Similarly, the unit tests, full-scale tests, examples and documentation are also in their own directories.

`applications` Top-level Python application and utility drivers.

`libsrc` C++ source for PyLith library.

`modulesrc` SWIG interface files for C++ Python bindings.

`pylith` Python source code for PyLith Python modules.

`doc` Documentation.

`examples` PyLith example suite.

`share` Useful settings and configuration files.

`unittests` Python and C++ unit tests source files. Run using `make check`.

`tests_auto` Full-scale tests. Run using `make check`.

`tests` Full-scale tests that require manual checking of results.

`travis` Helper scripts for building PyLith in Travis-CI.

`m4` Autoconf macros (link to geodynamics/autoconfig Git repository).

`playpen` Scratch area (obsolete). Use branches for scratch work instead.

We use the Pyre Python framework to collect all user parameters and to launch the MPI application. As a result, the top-level code is written in Python. In most cases there is a low-level C++ object of the same name with the low-level implementation of the object. We limit the Python code to collection of the user parameters, some simple checking of the parameters, and passing the parameters to the corresponding C++ objects.

The C++ library and Python modules are organized into several subpackages.

bc Boundary conditions.

faults Faults.

feassemble General finite-element formulation.

fekernels Finite-element point-wise functions (kernels).

friction Fault constitutive models.

materials Material behavior, including bulk constitutive models.

meshio Input and output.

problems General problem formulation.

topology Finite-element mesh topology.

utils General utilities.

3.1.2 PyLith Application Flow

The PyLith application driver performs two main functions. First, it collects all user parameters from input files (e.g., `.cfg` files) and the command line, and then it performs from simple checks on the parameters. Second, it launches the MPI job.

Once the MPI job launches, the application flow is:

1. Read the finite-element mesh; pylith.meshio.MeshImporter.
   (a) Read the mesh (serial); pylith::meshio::MeshIO.
   (b) Reorder the mesh; pylith::topology::ReverseCuthillMcKee.
   (c) Insert cohesive cells as necessary (serial); pylith::faults::FaultCohesive.
   (d) Distribute the mesh across processes (parallel); pylith::topology::Distributor.
   (e) Refine the mesh if desired (parallel); pylith::topology::RefineUniform.
2. Setup the problem.
   (a) Preinitialize the problem by passing information from Python to C++ and doing minimal setup; pylith.Problem.preinitialize().
   (b) Perform consistency checks and additional checks of user parameters; pylith.Problem.verifyConfiguration().
   (c) Complete initialization of the problem; pylith::problems::Problem::initialize().
3. Run the problem; pylith.problems.Problem.run().
4. Cleanup; pylith.problems.Problem.finalize().
   (a) Close output files.
   (b) Deallocate memory.
   (c) Output PETSc log summary, if desired.

In the first step, we list the object performing the work, whereas in subsequent steps we list the top-level object method responsible for the work. Python objects are listed using the path.class syntax while C++ objects are listed using namespace::class syntax. Note that a child class may redefine or perform additional work compared to what is listed in the parent class method.

Reading the mesh and the first two steps of the problem setup are controlled from Python. That is, at each step Python calls the corresponding C++ methods using SWIG. Starting with the complete initialization of the problem, the flow is controlled at the C++ level.

3.1.2.1 Time-Dependent Problem

In a time-dependent problem, the PETSc TS object (relabeled PetscTS within PyLith) controls the time stepping. At the beginning of each time step, the PetscTS object calls problems::TimeDependent::prestep(), and at the end of each time step, it calls problems::TimeDependent::poststep(). Within each time step, the PetscTS object calls the PETSc linear and nonlinear solvers as needed, which call the following methods of the C++ pylith::problems::TimeDependent object as needed computeRHSResidual(), computeRHSJacobian(), computeLHSResidual(), and computeLHSJacobian(). The pylith::problems::TimeDependent object calls the corresponding methods in the boundary conditions, constraints, and materials objects.

3.1.2.2 Boundary between Python and C++

The Python code is limited to collecting user input. Everything else is done in C++. This facilitates debugging (it is easier to track symbols in the C/C++ debugger) and unit testing, and reduces the amount of information that needs to be passed from Python to C++. The source code that follows shows the essential ingredients for Python and C++ objects, using the concrete example of the Material objects.

Warning

The examples below show skeleton Python and C++ objects to illustrate the essential ingredients. We have omitted documentation and comments that we would normally include and simplified the object hierarchy. See Section 4 on page 53 for details about the coding style we use in PyLith.

Important

Consistent inheritance between C++ and Python is important in order for SWIG to generate a Python interface that is consistent with the C++ interface.

Skeleton Python object in PyLith

```
from pylith.utils.PetscComponent import PetscComponent
from .materials import Material as ModuleMaterial

# Python objects should inherit the corresponding SWIG interface object (ModuleMaterial).
# Python object inheritance should match C++ object inheritance.
class Material(PetscComponent, ModuleMaterial):

    # Pyre inventory: properties and facilities
    import pyre.inventory

    materialId = pyre.inventory.int("id", default=0)
```

```
    materialId.meta['tip'] = "Material identifier (from mesh generator)."

    label = pyre.inventory.str("label", default="", validator=validateLabel)
    label.meta['tip'] = "Descriptive label for material."


    # Public methods

    def __init__(self, name="material"):
        IntegratorPointwise.__init__(self, name)
        return

    def preinitialize(self, mesh):

        # Create the C++ object. This method must be called exactly once.
        self._createModuleObj()

        # Pass name of Pyre component to C++
        ModuleMaterial.identifier(self, self.aliases[-1])

        # Pass Pyre inventory to C++
        ModuleMaterial.id(self, self.materialId)
        ModuleMaterial.label(self, self.label)
        return

    def verifyConfiguration(self, solution):
        # Avoid implementing this method at the Python level if at all possible.
        return


    # Private methods

    def _createModuleObj(self):
        ModuleMaterial.__init__(self)
        return
```

Skeleton C++ header file in PyLith

```
#if !defined(pylith_materials_material_hh) // Include guard
#define pylith_materials_material_hh

#include "materialsfwd.hh" // forward declaration of Material object

#include "pylith/utils/PyreComponent.hh" // ISA PyreComponent

class pylith::materials::Material : public pylith::utils::PyreComponent {
    friend class TestMaterial // unit testing

public: // public methods

    // Constructor and desctructor

    Material(const int dimension);
    virtual ~Material(void);

    // Method to deallocate PETSc data structures before calling PetscFinalize().
    virtual void deallocate(void);

    // Accessors

    int dimension(void) const;
    void id(const int value);
    int id(void) const;
    void label(const char* value);
    const char* label(void) const;

    // Initialization
```

```
    void verifyConfiguration(const pylith::topology::Field& solution);
    void initialize(const pylith::topology::Field& solution);

    // Finite-element integration

    void computeRHSResidual(pylith::topology::Field* residual,
                            const PylithReal t,
                            const PylithReal dt,
                            const pylith::topology::Field& solution);
    void computeRHSJacobian(PetscMat jacobianMat,
                            PetscMat preconMat,
                            const PylithReal t,
                            const PylithReal dt,
                            const pylith::topology::Field& solution);


private: // Data members

    const int _dimension;
    int _id;
    std::string _label;

    static const char* _pyreComponent; // Name of Pyre component.


private: // Methods not implemented by design to avoid expensive copies.

    Material(const Material&);
    const Material& operator=(const Material&);

};

#endif // pylith_materials_material_hh
```

Skeleton C++ definition file in PyLith

```
// Information about local configuration generated while running configure script.
#include <portinfo>

#include "Material.hh" // implementation of object methods

const char* pylith::materials::Material::_pyreComponent = "material";

pylith::materials::Material::Material(const int dimension) :
     _dimension(dimension),
     _id(0),
     _label("") {
    pylith::utils::PyreComponent::name(_pyreComponent);
} // constructor

pylith::materials::Material::~Material(void) {
    deallocate();
} // destructor

void
pylith::materials::Material::deallocate(void) {
    PYLITH_METHOD_BEGIN;

    // Deallocate PETSc and other data structures

    PYLITH_METHOD_END;
} // deallocate

int
pylith::materials::Material::dimension(void) const {
    return _dimension;
```

```
}

void
pylith::materials::Material::id(const int value) {
    PYLITH_COMPONENT_DEBUG("id(value="<<value<<")");

    _id = value;
} // id

int
pylith::materials::Material::id(void) const {
    return _id;
} // id

void
pylith::materials::Material::label(const char* value) {
    PYLITH_COMPONENT_DEBUG("label(value="<<value<<")");

    _label = value;
} // label

const char*
pylith::materials::Material::label(void) const {
    return _label.c_str();
} // label

void
pylith::materials::Material::initialize(const pylith::topology::Field& solution) {
    PYLITH_METHOD_BEGIN;
    PYLITH_COMPONENT_DEBUG("intialize(solution="<<solution.label()<<")");

    // Implementation not shown.

    PYLITH_METHOD_END;
} // initialize

// Implementation of other methods not shown.
```

3.1.2.3 SWIG Interface Files

SWIG interface files are essentially stripped down versions of C++ header files. Because SWIG only implements the public interface, we omit all data members and all protected and private data methods that are not abstract methods or implement abstract methods.

SWIG interface file

```
// The class declaration must appear within the appropriate namespace blocks.

namespace pylith {
    namespace materials {

        class Material : public pylith::utils::PyreComponent {
            public: // public methods

            // Constructor and desctructor

            Material(const int dimension);
            virtual ~Material(void);

            // Method to deallocate PETSc data structures before calling PetscFinalize().
            virtual void deallocate(void);

            // Accessors

            int dimension(void) const;
```

```
            void id(const int value);
            int id(void) const;
            void label(const char* value);
            const char* label(void) const;

            // Initialization

            void verifyConfiguration(const pylith::topology::Field& solution);
            void initialize(const pylith::topology::Field& solution);

            // Finite-element integration

            void computeRHSResidual(pylith::topology::Field* residual,
                                    const PylithReal t,
                                    const PylithReal dt,
                                    const pylith::topology::Field& solution);
            void computeRHSJacobian(PetscMat jacobianMat,
                                    PetscMat preconMat,
                                    const PylithReal t,
                                    const PylithReal dt,
                                    const pylith::topology::Field& solution);

        };
    }
}
```

3.2 Building for Development

For those less familiar with building software from source and using Git for version control, we recommend using the PyLith developer binary. This is a special binary that contains binary versions of all of the PyLith dependencies except for spatialdata and PETSc. ose two dependencies and PyLith are downloaded from the Git repositories, configured, and built from source using a utility script. See Section **??** on page ?? for detailed instructions.

For those comfortable building from source and using Git for version control, we recommend using the PyLith Installer utility (see Section **??** on page ??) to build PyLith and its dependencies from source.

3.2.1 Developer Workflow

We use the Git version control system `https://git-scm.com` with a central GitHub repository `https://github.com/geodynamics/pylith` for development. We will refer to this central repository as the geodynamics/pylith repository. Only the PyLith maintainers have write access to the geodynamics/pylith repository; everyone else is limited to read access. Contributions are incorporated into the geodynamics/pylith repository via pull requests.

Currently, the PyLith maintainers use the Gitflow Workflow, which is a variation of the Feature Branch Workflow. New features are added on individual branches and merged to specific shared branches.

At any given time there are three main shared Git branches:

`master` The `master` branch is the stable development branch. We generally start new development by creating branches from this branch. Releases are created when the `master` branch has accumulated a desired set of features.

`maint` The `maint` branch is the maintenance branch for the current release. Bug fixes are pushed to this branch and merged to the `master` as appropriate. Upon a new release, the `master` branch is merged into the `maint` branch.

`next` The `next` branch is used to test integration of development branches. The PyLith maintainers test integration of new features they are implementing by merging to this branch, before merging them to the `master` branch.

If you wish to contribute to PyLith development, then you should follow the Forking Workflow illustrated in Figures 3.1 on the next page and 3.2 on page 31. You fork the geodynamics/pylith repository so that you have a copy of the repository for

your changes. Each new, independent feature you develop should be done in a separate branch. Large feature contributions should be broken up into multiple phases, where each phase provides a working version that passes all of its tests. Each phase corresponds to a separate branch that is merged back into the geodynamics/pylith repository (usually the `master` branch) via a pull request when it is completed.

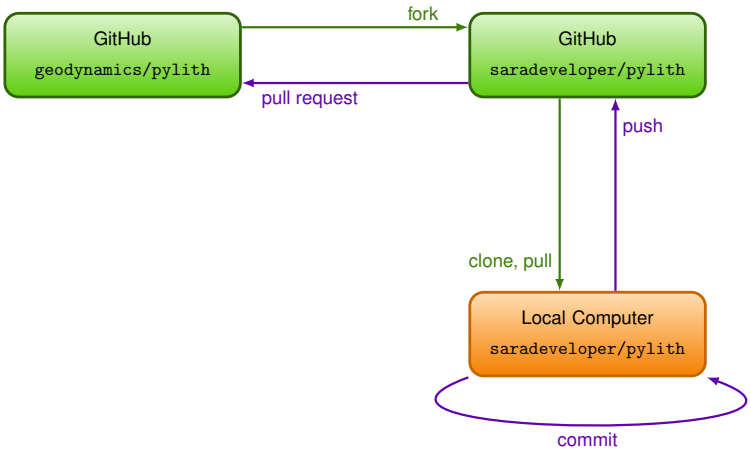

Figure 3.1: Overview of repositories for the Git forking workflow used in PyLith development. The main repository is the geodynamics/pylith at GitHub. Developers create a fork of that repository under their own GitHub account (e.g., saradeveloper), which is cloned to their local computer. Changes and additions to the code are committed to the repoisitory on the local computer, which are pushed to the developer's GitHub account. Once a development task is completed, a developer is encouraged to contribute the changes and additions to the main repository.

The PyLith maintainers would prefer to use the Forking Workflow for all development, but this requires multiple dedicated maintainers to review each other's code.

There are two steps to setting up a copy of the PyLith repository that you can use for development under the Forking Workflow:

1. Create a GitHub account.
2. Fork the geodynamics/pylith repository.

   This will create a copy of the PyLith repository in your GitHub account. You can make changes to this copy and when you are ready to contribute changes back to the geodynamics/pylith repository you will create a pull request.

★ Tip

We recommend adding your SSH public key to your GitHub account. See GitHub Help: Connecting To GitHub with SSH for more information. This provides additional security and eliminates the need for you to enter your username and password whenenver you push to your repository. For additional security consider setting up your GitHub account to use two-factor authentication. See GitHub Help: Securing your account with two-factor authentication.

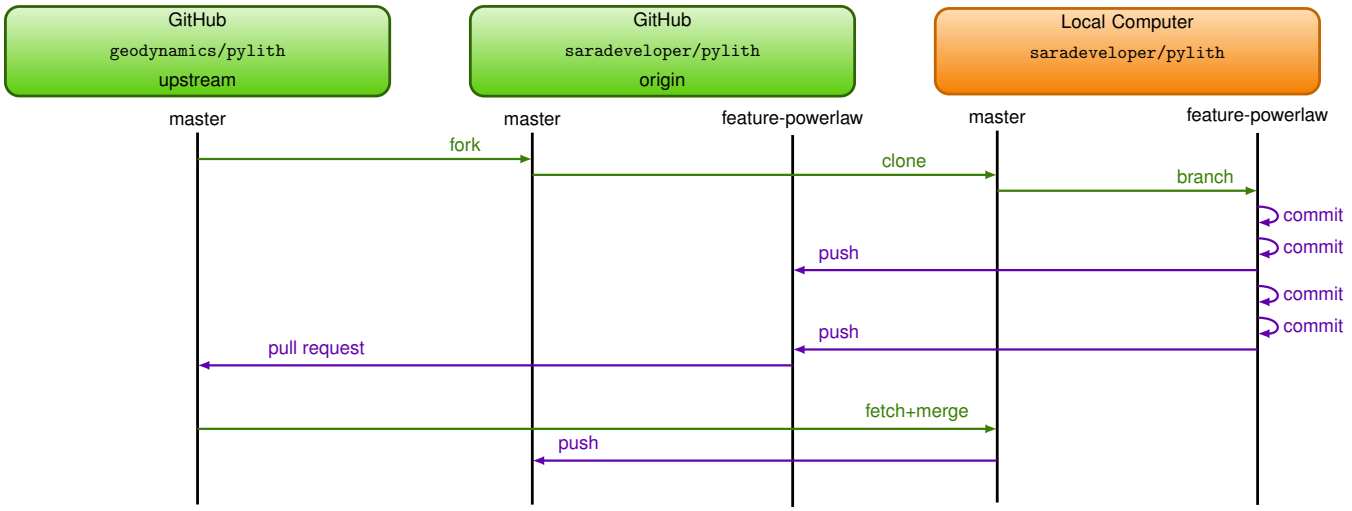

Figure 3.2: Overview of repositories and branches for the Git forking workflow used in PyLith development. From the `master` branch on your local machine, you create a feature branch, e.g., `feature-powerlaw`, to complete a single task such as adding a feature, fixing a bug, or making an improvement to the documentation. You should break up the changes into several steps, which are saved locally as commits. The commits may be pushed to the your repository on GitHub for backup or syncing across multiple computers. Once the task is completed, you submit a pull request to have the changes merged to the master branch on the community repository. Once the pull request is merged, you update your own master branch on your local computer from the community repository and then push the changes to your GitHub repository.

3.2.2 Developer Binary

TODO @brad

Add this section when the developer binary is created.

3.2.3 PyLith Installer for Development

The PyLith Installer utility can be used to build PyLith and all of its dependencies from source. There are several configure arguments relavant to using the installer for development:

`--with-pylith-repo=URL` Clone the PyLith source code from this URL. The URL corresponds to your fork of the PyLith repository. If you are accessing GitHub via SSH, the URL usually has the form `git@github.com:YOUR_GITHUB_USERNAME/pylith.git`. If you are accessing GitHub via HTTPS, the URL usually has the form `https://github.com/YOUR_GITHUB_USERNAME/pylith.git`. The default URL is the geodynamics/pylith repository, `https://github.com/geodynamics/pylith.git`.

`--with-pylith-git=BRANCH` Build the `BRANCH` of PyLith. If you are just starting development and have not created any feature branches, then use the `knepley/feature-petsc-fe` branch.

`--with-spatialdata-repo=URL` Clone the spatialdata source code from this URL. The default is the geodynamics/spatialdata repository, `https://github.com/geodynamics/spatialdata.git`. You do not need to change this unless you want to contribute to development of the spatialdata library.

TODO @brad

Update the name of the branch in the description of `--with-pylith-git` to `master` when `knepley/feature-petsc-fe` has been merged.

Other common configure arguments used when configuring for development include:

`--enable-swig` Build the SWIG library for building the Python/C++ interface.

`--enable-pcre` Build the PCRE library for use in SWIG.

`--enable-debugging` Use debugging (generate debugging symbols and use low-level optimization) when building spatialdata, PETSc, and PyLith.

Tip

For all development, we build with debugging turned on. By building in directories separate from the source code, we can build an optimized version for production runs in a different directory and use environment variables to select the desired build.

Example PyLith Installer setup for development

```
# Prerequisites:
#  * C/C++ compiler
#  * autotools
#  * fork of PyLith repository

# Set some variables to hold information about our installer setup.
# The PYLITH_REPO should be
# https://github.com/YOUR_GITHUB_USERNAME/pylith.git or
# git@github.com:YOUR_GITHUB_USERNAME/pylith.git
$ PYLITH_DIR=$HOME/pylith-developer
$ PYLITH_REPO=git@github.com:YOUR_GITHUB_USERNAME/pylith.git
$ PYLITH_BRANCH=knepley/feature-petsc-fe

# Create a top-level directory for PyLith.
$ mkdir -p $PYLITH_DIR
$ cd $PYLITH_DIR

# Clone the installer.
$ git clone --recursive https://github.com/geodynamics/pylith_installer.git

# Generate the configure script (must have autotools installed).
$ cd pylith_installer && autoreconf -if

# Create a build directory for the installer and configure the installer
$ cd $PYLITH_DIR && mkdir build && cd build
$ ../pylith_installer/configure \
    --with-pylith-git=$PYLITH_BRANCH \
    --with-pylith-repo=$PYLITH_REPO \
    --enable-debugging \
    --enable-mpi=mpich \
    --enable-pcre \
    --enable-swig \
    --with-fetch=curl \
    --with-make-threads=2 \
    --prefix=$PYLITH_DIR/dist
```

```
# Check the configure output to make sure it ran without errors and
# the configuration matches what you want.

# Setup your environment
$ source setup.sh

# Build the dependencies and PyLith
$ make
```

3.2.4 Keeping Your Fork in Sync with `geodyamics/pylith`

See Figure 3.2 on page 31 for the diagram of the workflow associated with these steps.

3.2.4.1 Set the upstream repository (done once per computer)

On each computer where you have the clone of your fork, you need to create a link to the "upstream" geodynamics/pylith repository. This allows you to keep your repository in sync with the central repository.

Setting upstream repository

```
# List the current remotes for your fork.
$ git remote -v
origin git@github.com/YOUR_GITHUB_USERNAME/pylith.git (fetch)
origin git@github.com/YOUR_GITHUB_USERNAME/pylith.git (push)

# Set the link to the remote upstream repository
$ git remote add upstream https://github.com/geodynamics/pylith.git

# Verify the upstream repository has been added.
$ git remote -v
origin git@github.com/YOUR_GITHUB_USERNAME/pylith.git (fetch)
origin git@github.com/YOUR_GITHUB_USERNAME/pylith.git (push)
upstream https://github.com/geodynamics/pylith.git (fetch)
upstream https://github.com/geodynamics/pylith.git (push)
```

3.2.4.2 Merging Updates from the Upstream Repository

Make sure all of your local changes have been committed or stashed.

Syncing your fork

```
# Update your local version of the upstream repository
$ get fetch upstream

# Check out the 'knepley/feature-petsc-fe' branch
$ git checkout knepley/feature-petsc-fe
Switched to branch 'knepley/feature-petsc-fe'

# Merge 'knepley/feature-petsc-fe' from upstream to your local clone.
$ git merge upstream/knepley/feature-petsc-fe

# If there are no conflicts, push the changes to your fork on GitHub.
$ git push

# If you need to update a local branch with changes from
# 'knepley/feature-petsc-fe', then checkout the branch and
# merge the local 'knepley/feature-petsc-fe' branch.
$ git checkout MY_USERNAME/feature-something-great
$ git merge knepley/feature-petsc-fe
```

3.2.5 Creating a New Feature Branch

Before creating a new feature branch, you should merge updates from the upstream repository as described in Section 3.2.4.2 on the preceding page.

Creating a feature branch

```
# Start from the current development branch (usually 'master')
$ git checkout master

# Make sure it is up to date.
$ git pull

# Create branch from 'master', substituting appropriate names for
# USERNAME and BRANCH.
$ git checkout -b USERNAME/BRANCH
```

3.2.6 Staging, Committing, and Pushing Changes

The Git `add` and `commit` commands are used to commit changes to a branch. A commit changes only the current branch in the repository clone on your local machine. In order to update your GitHub fork you need to `push` your changes. See the Git documentation for details about these commands. There are Git interfaces built into a number of editors and integrated development environments (IDEs), and there are standalong graphical user interfaces to Git.

★ Tip

If you have multiple branches, make sure you are on the correct branch before making commits.

3.2.7 Making Pull Requests

Once you have completed implementing and testing a new feature on a branch in your fork and wish to contribute it back to the geodynamics/pylith repository, you open a pull request. See GitHub Help: About pull requests for more information.

3.2.8 Rebuilding PETSc

Updating and rebuilding PETSc is quite simple once it has been configured and built once before.

⚠ Important

After rebuilding PETSc, you should rebuild PyLith. If there are incompatibilities between the two, then you will normally get compilation errors building PyLith. If there are incompatibilities and you do not rebuild PyLith, then you will usually get a segmentation fault when running PyLith.

Updating and rebuilding PETSc

```
# Change to PETSc source Directory
$ cd PETSC_SOURCE_DIRECTORY
```

```
# Get updates from PETSc repository.
$ git pull

# Reconfigure
$ arch-pylith/lib/petsc/conf/reconfigure-arch-pylith.py

# Rebuild
$ make

# Install
$ make install
```

3.2.9 Rebuilding PyLith

3.2.9.1 Overview

Pylith uses the GNU Build System (autotools: autoconf, automake, libtool) to configure and build. The configure settings are defined in `configure.ac` with additional macros in the `m4` directory. Note that the `m4` directory is a Git submodule corresponding to the geodynamics/autoconf Git repository.

3.2.9.2 Makefiles

The PyLith `configure` script uses automake to convert each `Makefile.am` file into a `Makefile`. The organization and content of the `Makefile.am` file depends on whether it is related to the C++ library, SWIG interface files, Python modules, C++ unit tests, Python unit tests, or examples.

For the C++ library files within the `libsrc` directory, the `libsrc/Makefile.am` contains the implementation files while the header files are listed in the `Makefile.am` file within the underlying directories.

For the SWIG interface files within the `modulesrc` directory, the `Makefile.am` file contains information on how to build the SWIG Python module.

The Python modules use a single `Makefile.am` file in the `pylith` directory.

Each test suite (C++ unit tests for each subpackage, Python unit tests for each subpackage, and each suite of full-scale tests) use a single `Makefile.am`. These files define how the tests are built, additional input files that should be included in the source distribution, and temporary files that should be deleted.

Each suite of examples contains a `Makefile.am` that defines the files that are to be included in the source distribution. It also defines which files are created and should be deleted upon `make clean`.

3.2.9.3 Build Targets

Several build targets are defined to make it easy to rebuild/reinstall PyLith rerun tests whenever the source code changes. Each target can be run using `make TARGET` where `TARGET` is one of the following:

all Build.

install Build and install.

check Run all unit tests and full-scale automated tests.

★ Tip

On a machine with multiple cores, faster builds of the C++ code (library and C++ unit tests) are available using the `-jNTHREADS` command line argument, where `NTHREADS` is the number of cores to use. For example, `make -j16 all`.

After modifying code, the C++ library, SWIG modules, and Python code need to be rebuilt and reinstalled before running a PyLith simulation.

Rebuilding Pylith C++ library, SWIG modules, and Python modules

```
# Change to top-level PyLith build directory.
$ cd PATH_TO_PYLITH_BUILD_DIRECTORY

# Rebuild and reinstall library using 8 threads.
$ make install -j8 -C libsrc

# Rebuild and reinstall modules
$ make install -C modulesrc

# Reinstall Python code
$ make install -C PyLith

# Reinstall everything using 8 threads to build library.
$ make install -j8
```

When fixing a bug exposed by a C++ unit test, after changing C++ code, only the C++ library needs to be rebuilt before running a C++ unit test.

Rebuilding Pylith C++ library and rerunning a C++ unit test

```
# Change to top-level PyLith build directory.
$ cd PATH_TO_PYLITH_BUILD_DIRECTORY

# Rebuild library using 8 threads.
$ make -j8 -C libsrc

# Rerun boundary condition C++ unit tests.
$ make check -C unittests/libtests/bc
```

Similarly, when fixing a bug exposed by a Python unit test, after changing Python code, only the Python modules need to be reinstalled before running a Python unit test. If changes are also made to the C++ code, then the C++ library and modules must be rebuilt and reinstalled before running a Python unit test.

Rebuilding Pylith Python modules and rerunning a Python unit test

```
# Change to top-level PyLith build directory.
$ cd PATH_TO_PYLITH_BUILD_DIRECTORY

# Reinstall PyLith Python modules.
$ make install -C pylith

# Rerun boundary condition Python unit tests.
$ make check -C unittests/pytests/bc
```

3.3 PETSc Finite-Element Implementation

Formulating the weak form of the governing equation in terms of point-wise functions allows the PyLith implementation of the equations to be done at a rather high level. Most of the finite-element details are encapsulated in PETSc routines that compute

the integrals and solve the system of equations. In fact, adding materials and boundary conditions requires calling only a few PETSc finite-element functions to register the point-wise functions. A new material may need to add to the library of solution subfields and auxiliary subfields, but adding these fields is also done at a high-level.

The remainder of this section discusses three aspects of the finite-element implementation handled by PyLith to give you a peek of what is doing on under the hood.

3.3.1 DMPlex

The finite-element mesh is stored as a DMPlex object. This is a particular implementation of the PETSc Data Management (DM, renamed PetscDM within PyLith) object. Within a DMPlex object vertices, edges, faces, and cells are all called points. The points are numbered sequentially, beginning with cells, followed by vertices, edges, and then faces. Treating all topological pieces of the mesh the same way, as points in an abstract graph, allows us to write algorithms which apply to very different meshes without change. For example, we can write a finite element assembly loop that applies to meshes of any dimension, with any cell shape, with (almost) any finite element.

3.3.1.1 Point Depth and Height

In general, vertices are at a *depth* of 0 and cells are at the maximum depth. Similarly, cells are at a *height* of 0 and vertices are at the maximum height. Notice that depth and height correspond to the usual dimension and codimension. Table 3.1 shows the heights and depths of the vertices edges, faces, and cells in a 3-D mesh.

Table 3.1: Depth and height of various topological pieces.

Point type	Depth	Height
Vertices	0	3
Edges	1	2
Faces	2	1
Cells	3	0

For a boundary mesh, we currently store the full set (vertices, edges, faces, and cells). Obviously for 2-D meshes, the boundary mesh doesn't contain "volume" cells, but just vertices, edges, and faces. This means the "boundary" cells are at a height of 1 and a depth equal to the maximum depth $-$ 1 .

3.3.2 PetscSection and PetscVec

We store a field over the mesh in a vector (PETSc Vec object, renamed PetscVec in PyLith). The PetscSection object describes the layout of the vector over the DMPlex object. The vector may hold multiple subfields, each with its own discretization. The chart of the PetscSection defines the range of points (minimum and maximum) over which the section is defined. For each point in the chart, the section holds the number of degrees of freedom a point has and the offset in the PetscVec for its first degree of freedom. The section also holds the number of degrees of freedom and offset for each individual subfield within the section for each point in the chart.

Because the PetscSection knows only about points, and not about topology or geometry, it allows us to express the mathematical concepts without getting lost in the details. For example, a P_1 3D vector field assigns 3 dofs to each vertex. This same section could be used to layout a field over a circle, surface of a cylinder, Möbius strip, sphere, cube, ball, or Pylith mesh. It separates the layout of data over a mesh from the actual storage of values, which is done by the PetscVec. The section tells you how to look up the values, which are associated with a piece of the mesh, inside the vector. For example, you can ask for the offset into the vector for the values associated with an edge in the mesh, and how many dofs there are on the edge.

The PetscSection also includes the information about the constrained degrees of freedom. We refer to a PetscVec that includes the values for the constrained degrees of freedom as a *local* vector and a PetscVec with the values for the constrained

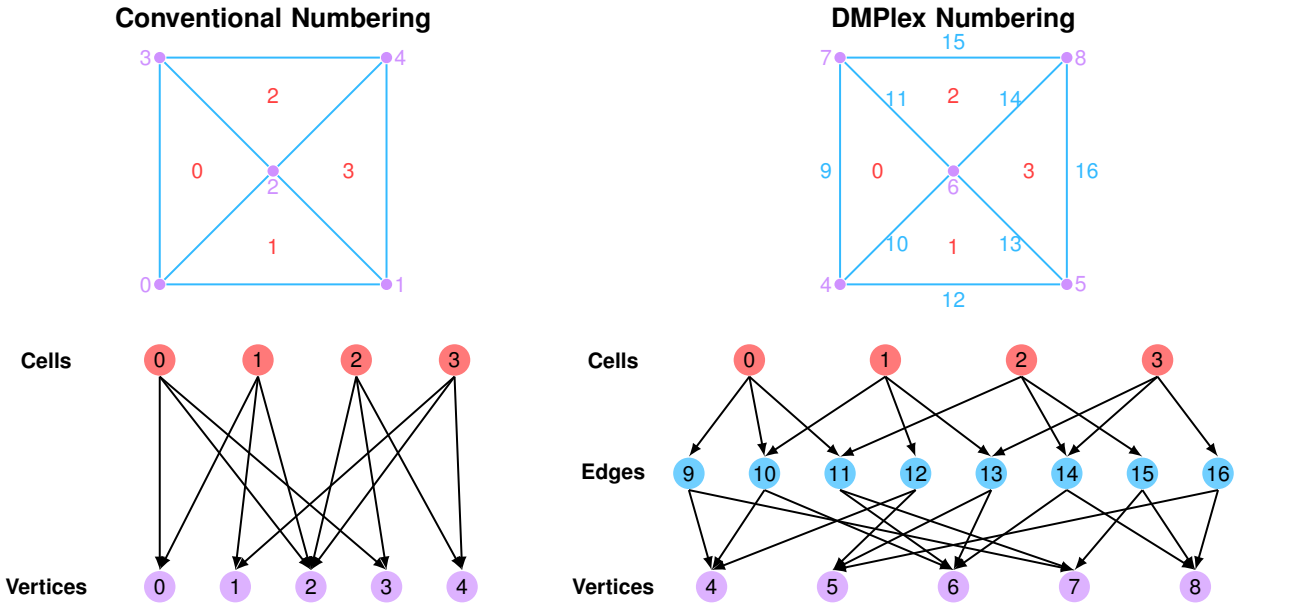

Figure 3.3: Conventional numbering (left) with vertices and cells numbered independently and DMPlex numbering (right) with cells, vertices, edges, (and faces), numbered sequentially.

degrees of freedom removed as a *global* vector. Constraints often arise from Dirichlet boundary conditions, which change the basis functions of the approximating space, but can also arise from algebraic relations between unknowns. A local vector is used for assembly of the residual vector and Jacobian matrix, because we need the boundary values in order to compute those integrals. Global vectors are used for the algebraic solver because we do not want solution values fixed by constraints to participate in the solve.

PetscSection information for a solution field with three subfields

```
# This example solution field has three subfields:
#   * displacement (vector field, 2 components, basis order 1)
#   * velocity (vector field, 2 components, basis order 1)
#   * pressure (scalar field, 1 component, basis order 0)
#
# The displacement and velocity subfields have degrees of freedom on the vertices.
# The pressure subfield has degrees of freedom on the cells.
#
# The order of the values (offsets in the PetscVec) follows the
# ordering of the points (cells, vertices, edges, and faces).
# In this example, the pressure subfield appears first (offsets 0-3),
# followed by the two components of the displacement subfield and
# velocity subfield for each point.
PetscSection Object: 1 MPI processes
  type not yet set
3 fields
  field 0 with 2 components # displacement field
Process 0:
# (POINT) dim SUBFIELD_NUM_COMPONENTS offset OFFSET
(   0) dim  0 offset   0 # Cells
(   1) dim  0 offset   0
(   2) dim  0 offset   0
(   3) dim  0 offset   0
(   4) dim  2 offset   4 # Vertices
(   5) dim  2 offset   8
(   6) dim  2 offset  12
(   7) dim  2 offset  16
(   8) dim  2 offset  20
(   9) dim  0 offset  24 # Edges
(  10) dim  0 offset  24
(  11) dim  0 offset  24
```

```
(  12) dim  0 offset  24
(  13) dim  0 offset  24
(  14) dim  0 offset  24
(  15) dim  0 offset  24
(  16) dim  0 offset  24
field 1 with 2 components # velocity field
Process 0:
(   0) dim  0 offset   0 # Cells
(   1) dim  0 offset   0
(   2) dim  0 offset   0
(   3) dim  0 offset   0
(   4) dim  2 offset   6 # Vertices
(   5) dim  2 offset  10
(   6) dim  2 offset  14
(   7) dim  2 offset  18
(   8) dim  2 offset  22
(   9) dim  0 offset  24 # Edges
(  10) dim  0 offset  24
(  11) dim  0 offset  24
(  12) dim  0 offset  24
(  13) dim  0 offset  24
(  14) dim  0 offset  24
(  15) dim  0 offset  24
(  16) dim  0 offset  24
field 2 with 1 components # pressure field
Process 0:
(   0) dim  1 offset   0 # Cells
(   1) dim  1 offset   1
(   2) dim  1 offset   2
(   3) dim  1 offset   3
(   4) dim  0 offset   4 # Vertices
(   5) dim  0 offset   4
(   6) dim  0 offset   4
(   7) dim  0 offset   4
(   8) dim  0 offset   4
(   9) dim  0 offset   4 # Edges
(  10) dim  0 offset   4
(  11) dim  0 offset   4
(  12) dim  0 offset   4
(  13) dim  0 offset   4
(  14) dim  0 offset   4
(  15) dim  0 offset   4
(  16) dim  0 offset   4
```

3.3.3 Integration

Integration involves integrals over the domain or the boundary of the domain. These two operations are done by different PETSc functions.

DMPlexComputeResidual_Internal Compute the contribution to the LHS or RHS residual for a single material. This function and the functions it calls handle looping over the cells in the material, integrating the weak form for each of the fields, and adding them to the residual.

A more appropriate name would be DMPlexComputeResidualSingle although that may change once we have a separate PetscDM for each material.

DMPlexComputeBdResidualSingle Compute the contribution to the LHS or RHS residual for a single boundary condition. This function and the functions it calls handle looping over the boundary, integrating the weak form for each of the fields, and adding them to the residual.

3.3.4 Projection

Input and output often involve projecting fields to/from the finite-element space. PETSc provides a family of functions for this. We generally use two of these, one for analytical functions and one for discretized fields. Projection may be a misleading term here, since we are not refering to the common L_2 projection, but rather interpolation of the function by functions in our finite-element space.

Let's start with the simple example of Fourier analysis, which most people have experience with. If we want the Fourier interpolant $\tilde{f}$ for a given function f, then we need to determine its Fourier coefficients, f_k, where

$$\tilde{f} = \sum_k f_k e^{ikx}. \tag{3.1}$$

This is straightforward because the basis functions in the Fourier representation are orthogonal,

$$\int_0^{2\pi} e^{-imx} e^{ikx}\, dx = 2\pi\delta_{km}. \tag{3.2}$$

To find the coefficient f_m, we just multiply by the conjugate of the basis function and integrate,

$$\int_0^{2\pi} e^{-imx} \tilde{f}\, dx = \int_0^{2\pi} e^{-imx} \sum_k f_k e^{ikx}\, dx, \tag{3.3}$$

$$= \sum_k f_k \int_0^{2\pi} e^{-imx} e^{ikx}\, dx, \tag{3.4}$$

$$= \sum_k f_k 2\pi\delta_{km}, \tag{3.5}$$

$$= 2\pi f_m, \tag{3.6}$$

and we have our coefficient f_m,

$$f_m = \frac{1}{2\pi} \int_0^{2\pi} e^{-imx} \tilde{f}\, dx. \tag{3.7}$$

The finite element basis $\{\phi_i\}$ is not orthogonal, so we have an extra step. We could take the inner product of f with all the basis functions, and then sort out the dependecies by solving a linear system (the mass matrix), which is what happens in L_2 projection. However, suppose we have another basis $\{\psi_i\}$ of linear functionals which is *biorthogonal* to $\{\phi_i\}$, meaning

$$\psi_i(\phi_j) = \delta_{ij}. \tag{3.8}$$

We can easily pick out the coefficient of $\tilde{f}$ by acting with the corresponding basis functional. Our interpolant is

$$\tilde{f} = \sum_k f_k \phi_k(x). \tag{3.9}$$

Acting on the interpolant with the biorthogonal basis results in

$$\psi_i(\tilde{f}) = \psi_i\left(\sum_k f_k \phi_k(x)\right). \tag{3.10}$$

Making use of the fact that the finite-element basis $\{\phi_i\}$ is linear, yields

$$\psi_i(\tilde{f}) = \sum_k f_k \psi_i\left(\phi_k(x)\right), \tag{3.11}$$

$$\psi_i(\tilde{f}) = \sum_k f_k \delta_{ik}, \tag{3.12}$$

$$\psi_i(\tilde{f}) = f_i. \tag{3.13}$$

We call $\{\phi_i\}$ the *primal* basis, and $\{\psi_i\}$ the *dual* basis. We note that if f does not lie in our approximation space spanned by $\{\phi_i\}$, then interpolation is not equivalent to L_2 projection. This will not usually be important for our purposes.

DMProjectFunctionLocal Project an analytical function into the given finite-element space.

DMProjectFieldLocal Project a discretized field in one finite-element space into another finite-element space.

3.3.5 Point-wise Functions

The following code excerpts show the interface for point-wise functions for the residual and the Jacobian.

PetscPointFunc Interface

```
/** Interface for PETSc point-wise functions.
 *
 * @param[in] dim Spatial dimension of problem.
 * @param[in] Nf Number of subfields in the field.
 * @param[in] NfAux Number of subfields in the auxiliary field.
 * @param[in] uOff Offset into u and u_t[] for each subfield.
 * @param[in] uOff_x Offset into u_x for each subfield.
 * @param[in] u Values of each subfield at the current point.
 * @param[in] u_t Time derivative of each subfield at the current point.
 * @param[in] u_x Gradient of each subfield at the current point.
 * @param[in] aOff Offset into u and u_t[] for each auxiliary subfield.
 * @param[in] aOff_x Offset into u_x for each auxiliary subfield.
 * @param[in] a Values of each auxiiliary subfield at the current point.
 * @param[in] a_t Time derivative of each auxiliary subfield at the current point.
 * @param[in] a_x Gradient of each auxiliary subfield at the current point.
 * @param[in t Current time.
 * @param[in] x Coordinates of current point.
 * @param[in] numConstants Number of constant parameters.
 * @param[in] constants Constant parameters.
 * @param[out] f Output values at current point.
 */
void
func(PetscInt dim,
     PetscInt Nf,
     PetscInt NfAux,
     const PetscInt uOff[],
     const PetscInt uOff_x[],
     const PetscScalar u[],
     const PetscScalar u_t[],
     const PetscScalar u_x[],
     const PetscInt aOff[],
     const PetscInt aOff_x[],
     const PetscScalar a[],
     const PetscScalar a_t[],
     const PetscScalar a_x[],
     PetscReal t,
     const PetscReal x[],
     PetscScalar f0[]);
```

PetscPointJac Interface

```
/** Interface for PETSc point-wise Jacobians.
 *
 * This is identical to the PetscPointFunc with the addition of the
 * u_tShift argument.
 *
 * @param[in] u_tShift The multiplier for dF/dU_t.
 */
void
func(PetscInt dim,
     PetscInt Nf,
     PetscInt NfAux,
     const PetscInt uOff[],
     const PetscInt uOff_x[],
     const PetscScalar u[],
     const PetscScalar u_t[],
     const PetscScalar u_x[],
     const PetscInt aOff[],
     const PetscInt aOff_x[],
     const PetscScalar a[],
     const PetscScalar a_t[],
```

```
      const PetscScalar a_x[],
      PetscReal t,
      PetscReal u_tShift,
      const PetscReal x[],
      PetscScalar Jf0[]);
```

The two functions in the following code excerpts show the interface for setting the point-wise functions.

PetscDSSetResidual Function

```
/** Set point-wise functions for LHS or RHS residual.
 *
 * @param[in] prob PetscDS associated with solution field.
 * @param[in] f Index of solution subfield for residual term.
 * @param[in] f0 Point-wise function for f0/g0 term in weak form.
 * @param[in] f1 Point-wise function for f1/g1 term in weak form.
 *
 * @returns PETSc error code (0 indicates no errors).
 */
PetscErrorCode
PetscDSSetResidual(PetscDS prob,
                   PetscInt f,
                   PetscPointFunc f0,
                   PetscPointFunc f1);
```

PetscDSSetJacobian Function

```
/** Set point-wise functions for LHS or RHS Jacobian.
 *
 * @param[in] prob PetscDS associated with solution field.
 * @param[in] f Index of trial subfield for Jacobian term.
 * @param[in] g Index of field subfield for Jacobian term.
 * @param[in] Jf0 Point-wise function for Jf0/Jg0 term in weak form.
 * @param[in] Jf1 Point-wise function for Jf1/Jg1 term in weak form.
 * @param[in] Jf2 Point-wise function for Jf2/Jg2 term in weak form.
 * @param[in] Jf3 Point-wise function for Jf3/Jg3 term in weak form.
 *
 * @returns PETSc error code (0 indicates no errors).
 */
PetscErrorCode
PetscDSSetResidual(PetscDS prob,
                   PetscInt f,
                   PetscInt g,
                   PetscPointJac Jf0,
                   PetscPointJac Jf1,
                   PetscPointJac Jf2,
                   PetscPointJac Jf3);
```

3.4 Adding New Governing Equations and/or Materials

There are four basic pieces to adding new physics in the form of a governing equation or material:

1. Select the fields for the solution. This will control the form of the partial differential equation and the terms in the residuals and Jacobians.
2. Derive the point-wise functions for the residuals and Jacobians. Determine flags that will be used to indicate which terms to include.
3. Determine which parameters in the point-wise functions could vary in space as well as any state variables. We bundle all state variables and spatially varying parameters into a field called the auxiliary field. Each material has a separate auxiliary field.
4. Parameters that are spatially uniform are treated separately from the parameters in the auxliary field.

3.4.1 Python

- Define solution subfields.
  - All subfields in the solution field are SolutionSubfield objects (see 3.4 on the following page). PyLith already includes several solution subfields:
    SubfieldDisplacement Displacement vector field.
    SubfieldVelocity Velocity vector field.
    SubfieldLagrangeFault Lagrange multiplier field for fault constraints.
    SubfieldPressure Fluid pressure or mean stress scalar field.
    SubfieldTemperature Temperature scalar field.
  - PyLith includes solution field containers with predefined subfields:
    SolnDisp Solution composed of a displacement field.
    SolnDispVel Solution composed of displacement and velocity fields.
    SolnDispPres Solution composed of displacement and mean stress (pressure) fields.
    SolnDispLagrange Solution composed of displacement and Lagrange multiplier fields.
    SolnDispPresLagrange Solution composed of displacement, mean stress (pressure), and Lagrange multiplier subfields.
    SolnDispVelLagrange Solution composed of displacement, velocity, and Lagrange multiplier subfields.
- Define auxiliary subfields.
  The auxiliary subfields for a material are defined as facilities in a Pyre Component. For example, the ones for IsotropicLinearElasticityPlaneStrain are in AuxFieldIsotropicLinearElasticity. The order of the subfields is defined *not* by the order they are listed in the Pyre component, but by the order they are added to the auxiliary field in the C++ object.
- Flags to turn on/off terms in governing equation.
  For the elasticity equation, we sometimes do not include body forces or inertial terms in our simulations. Rather than implement these cases as separate materials, we simply include flags in the material to turn these terms on/off. The flags are implemented as Pyre properties in our material component.

3.4.2 C++

> **Warning**
>
> We will likely be refactoring the Material and IntegratorPointwise, so how the pointwise functions for the residual, Jacobians, state variable update, and computation of derived fields are set will likely be significantly different and easier in the future. For now, you should use IsotropicLinearElasticityPlaneStrain as a model for how to implement a material.

- Define auxiliary subfields.
  We build the auxiliary field using classes derived from pylith::feassemble::AuxiliaryFactory. The method corresponding to each subfield specifies the name of the subfield, its components, and scale for nondimensionalizing. We generally create a single auxiliary factory object for each governing equation but not each bulk constitutive model, because constitutive models for the same governing equation often have many of the same subfields. For example, most of our bulk constitutive models for the elasticity contain density, bulk modulus, and shear modulus auxiliary subfields.

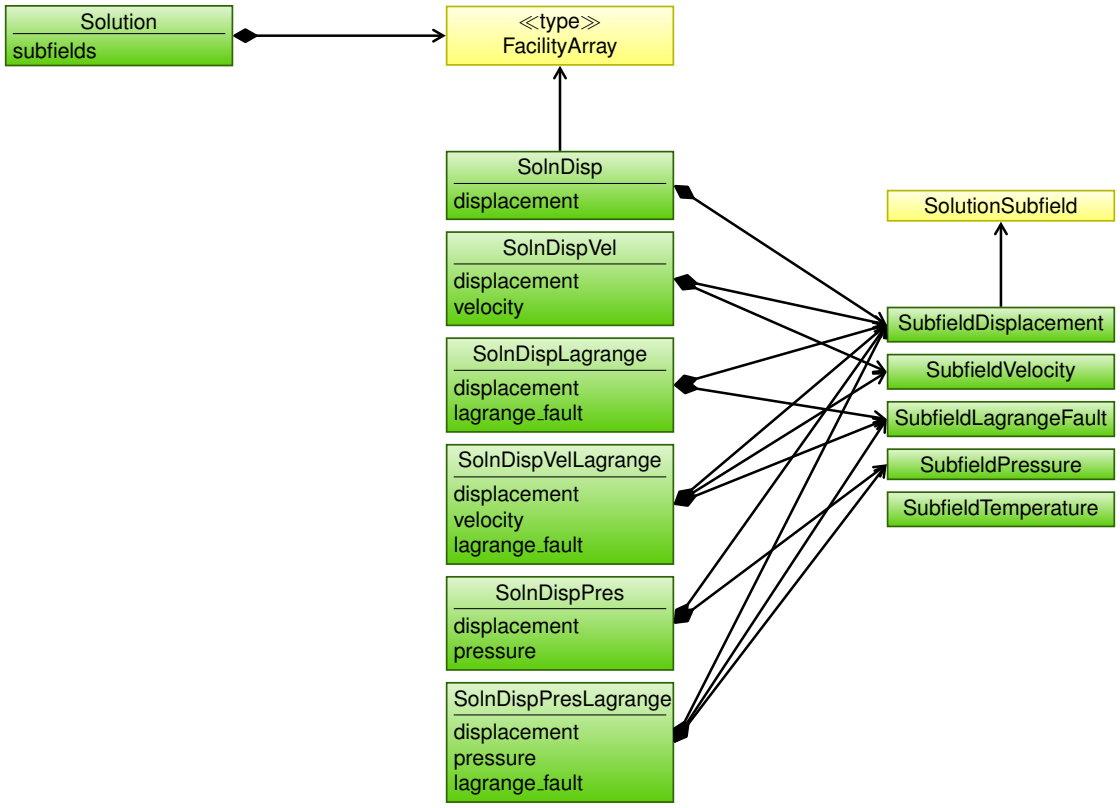

Figure 3.4: Class diagram for the solution field, solution subfields, and pre-defined containers of solution subfields.

> ⚠ **Important**
>
> Within the concrete implementation of the material object, we add the subfields to the auxiliary field. The order in which they are added determines the order they will be in the auxiliary field. You will need to use know this order when you implement the point-wise functions.

- Implement the point-wise functions.
  The point-wise functions for the residuals, Jacobians, and projections follow nearly identical interfaces. Note that within PyLith, we use PylithInt, PylithReal, and PylithScalar instead of PetscInt, PetscReal, and PetscScalar.
- Set the point-wise functions.
  We set the point-wise functions for the residual using PetscDSSetResidual and for the Jacobian using PetscDSSetJacobian.

3.4.3 C++ Unit Tests

The C++ unit tests focus on testing all methods of a C++ object. In most cases, this includes the accessors, layout and values of the auxiliary field, the residual, and Jacobian. For the residual and Jacobian, we use the method of manufactured solutions. In testing the residual, we manufacture a solution that solves the governing equation. We setup the object so that the auxiliary field and solution passed to the residual computation should produce a zero residual vector. The UserFunctionDB spatial database allows us to populate the auxiliary field and solution using anlytical functions. To test the computation of the RHS Jacobian, we compute

$$J_g(s)(p-s) = G(p) - G(s), \tag{3.14}$$

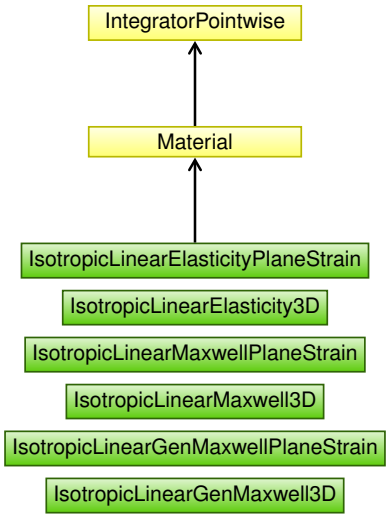

Figure 3.5: Class diagram for the bulk materials. Each material implementation inherits from the abstract Material class which inherits from the abstract IntegratorPointwise class.

where $J_g(s)$ is the RHS Jacobian, s is a solution to the governing equation, p is s plus a small perturbation, $G(p)$ is the RHS residual for p, and $G(s)$ is the RHS residual for s. We test the computation of the LHS Jacobian, using the corresponding LHS Jacobian J_f and LHS residual F.

> ⚠ **Important**
>
> In setting up a test using the Method of Manufactured Solutions it is important to make sure the discretization of the used for each of the subfields in the auxiliary field and solution field are sufficient to represent the analytical function.

3.4.4 Python Unit Tests

With the Python implementation focused on gathering the user configuration of a simulation, there is minimal functionality we can test at the Python level. Consequently, the Python unit tests are generally limited to testing the SWIG interface, which involves creating the C++ object and insuring the Pyre properties and facilities and any other information is passed to the C++ object.

3.5 Debugging

Debugging PyLith can be challenging due its many dependencies and complex interaction with PETSc. The most efficient strategy for debugging is to first try to expose the bug in a serial unit test, followed by a serial full-scale test, and then a parallel full-scale test. This may require creating new unit tests or full-scale tests if the bug is not exposed by current tests. The PyLith developers make extensive use debuggers, such as `gdb` and `lldb`, and memory management analysis tools, such as `valgrind`, to detect and squash bugs.

3.5.1 Runing PyLith in a Debugger

Running PyLith in a debugger is often an efficient way to quickly zero in on the origin of an error. To start the `gdb` debugger when running the PyLith application, simply add the command line argument `--petsc.start_in_debugger`. To use an

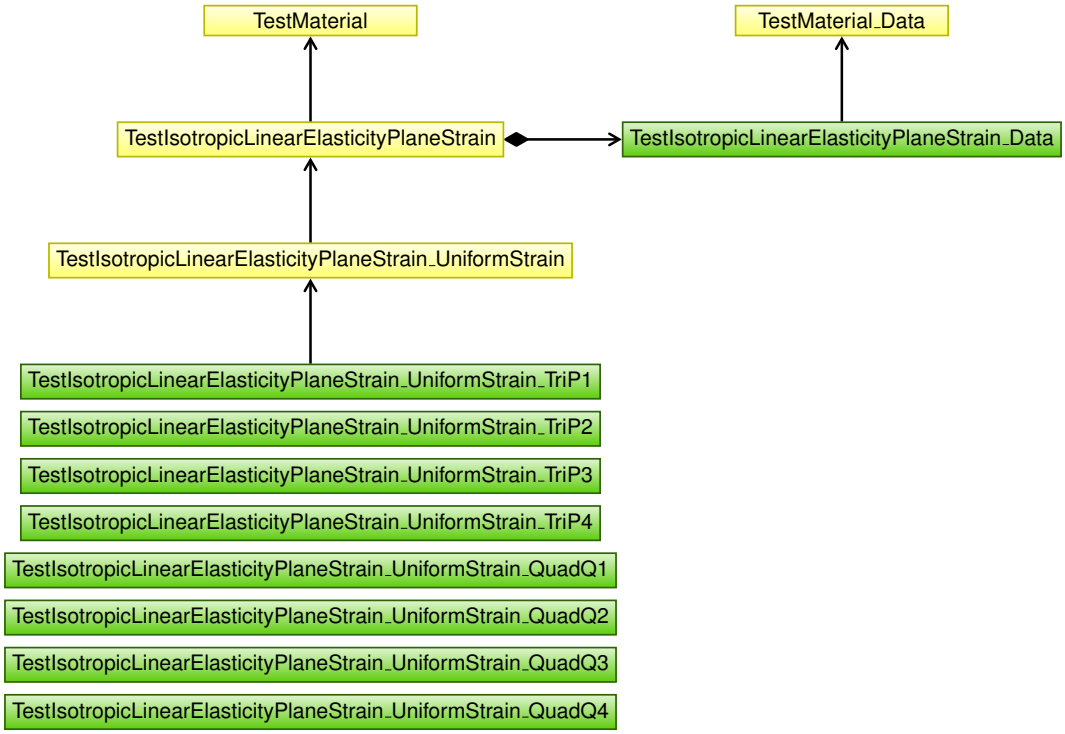

Figure 3.6: Class diagram for testing the IsotropicLinearElasticityPlaneStrain material with a solution generated using the Method of Manufactured Solutions. TestIsotropicLinearElasticityPlaneStrain_UniformStrain contains the parameters and generated solution and the subclasses include various discretizations. All of the test data is held in the TestIsotropicLinearElasticityPlaneStrain_Data object.

alternative debugger, such as `lldb`, append the name of the debugger executable, for example `--petsc.start_in_debugger=lldb`. By default, PETSc will try to start the debugger in an xterm. To use an alternative terminal program, use the command line argument `--petsc.debug_terminal=TERMINAL`. For example for the GNOME terminal, use `--petsc.debug_terminal="gnome-terminal -x"`.

Running PyLith in a debugger

```
# Start the gdb debugger when running PyLith
$ pylith --petsc.start_in_debugger

# Start the lldb debugger when running PyLith
$ pylith --petsc.start_in_debugger=lldb

# Start the gdb debugger in a GNOME terminal.
$ pylith --petsc.start_in_debugger --petsc.debug_terminal="gnome-terminal␣-x"
```

Debugging with gdb

```
# Set breakpoint at line: b FILE:LINE
# Set breakpoint at line 150 of Material.cc
(gdb) b Material.cc:150

# Set breakpoint at exception throw
(gdb) catch throw

# Show arguments for the current frame
(gdb) info args

# Show local variables for the current frame
(gdb) info locals

# Show the contents of a local variable: p VARIABLE
(gdb) p numFields

# Show the contents of local variable: p POINTER[0]@SIZE
# Print array of 4 values pointed to by variable values
(gdb) p values[0]@4

# Print stack trace
(gdb) backtrace
```

Debugging with lldb

```
# Set breakpoint at line: b FILE:LINE
# Set breakpoint at line 150 of Material.cc
(lldb) breakpoint set -f Material.cc -l 150

# Set breakpoint at exception throw
(lldb) break set -E C++

# Show local variables
(lldb) frame variable

# Show the contents of a local variable: frame variable VARIABLE
(lldb) frame variable numFields
# Alternatively
(lldb) p numFields

# Show the contents of an array of values: p *(TYPE(*)[SIZE])POINTER
# Show the contents of an int array of 10 values pointed to by the variable values (int*).
(lldb) p *(int(*)[10])values
```

It is also helpful to run the debugger when finding errors in unit tests. Note that the executable in the unit tests build directory is a wrapper around the actual executable in the `.libs` directory. Consequently, you must run the debugger on the executable in the `.libs` directory.

> **Important**
>
> The executables for unit tests in the build directory are wrappers that insure the current version of the PyLith library is used with the actual executable. Thus, when running the debugger on the actual executable, be sure you have run `make install` for the library so that the current version will be used.

Running the debugger for a unit test

```
# Run the debugger on the testbc executable
$ gdb .libs/testbc
```

3.5.2 Runing Valgrind on PyLith

Valgrind is a useful tool for finding memory leaks, use of uninitialized variables, and invalid reads and writes to memory. When running valgrind there are three useful command line arguments:

`--log-file=FILENAME` Send output to FILENAME. This does not work when running the PyLith application because each new process wipes out the log file.

`--suppressions=FILE` Omit errors matching given patterns when reporting errors. Valgrind often reports lots of errors arising from the way OpenMPI and Python handle memory allocation and deallocation. We usually use the Python suppression file `share/valgrind-python.supp` when running valgrind.

`--trace-children=yes` Continue tracing errors in subprocesses. This is important when running valgrind on the PyLith executable, as the actual computation is done in a forked process.

Running Valgrind

```
# Run valgrind on the testbc executable
$ valgrind --log-file=valgrind_bc.log \
  --suppressions=$HOME/src/cig/pylith/share/valgrind-python.supp .libs/testbc

# Run valgrind on the PyLith executable
$ valgrind --trace-children=yes \
  --suppressions=$HOME/src/cig/pylith/share/valgrind-python.supp pylith
```

3.5.3 Debugging Output

In addition to using the debugger to inspect code and variables, it is often helpful to print fields to stdout or inspect where a computed field does not match the expected field. Turning on this type of output is usually done by temporarily inserting calls to a few viewing functions within the test code.

Viewing a field will print the subfield metadata, the layout of the field, and the field values. See Section 3.3.2 on page 37 for how to interpret the layout of a field as given by PetscSection.

Viewing a field

```
// Call Field::view(const char* description) method.
solution.view("Solution Field");
```

Output from Field::view()

```
Viewing field 'solution' Solution Field.
  Subfields: # Order of subfields is given by the index, not the order listed.
    Subfield displacement, index: 0, components: displacement_x displacement_y, scale: 1000, basisOrder: 1, quadOrder
    Subfield fluid_pressure, index: 2, components: fluid_pressure, scale: 0.1, basisOrder: 1, quadOrder: 1
    Subfield velocity, index: 1, components: velocity_x velocity_y, scale: 100, basisOrder: 1, quadOrder: 1
  dimensionalize flag: 0
DM Object: 1 MPI processes
  type: plex
DM_0xe6a550_38 in 2 dimensions:
  0-cells: 5
  1-cells: 8
  2-cells: 4
Labels:
  boundary_bottom: 1 strata with value/size (1 (3))
  boundary: 1 strata with value/size (1 (8))
  material-id: 1 strata with value/size (24 (4))
  depth: 3 strata with value/size (0 (5), 1 (8), 2 (4))
PetscSection Object: 1 MPI processes
  type not yet set
3 fields
  field 0 with 2 components # displacement vector field
Process 0:
  (   0) dim  0 offset   0
  (   1) dim  0 offset   0
  (   2) dim  0 offset   0
  (   3) dim  0 offset   0
  (   4) dim  2 offset   0 constrained 1 # y degree of freedom is constrained
  (   5) dim  2 offset   5 constrained 1 # y degree of freedom is constrained
  (   6) dim  2 offset  10
  (   7) dim  2 offset  15
  (   8) dim  2 offset  20
  (   9) dim  0 offset  25
  (  10) dim  0 offset  25
  (  11) dim  0 offset  25
  (  12) dim  0 offset  25
  (  13) dim  0 offset  25
  (  14) dim  0 offset  25
  (  15) dim  0 offset  25
  (  16) dim  0 offset  25
  field 1 with 2 components # velocity vector field
Process 0:
  (   0) dim  0 offset   0
  (   1) dim  0 offset   0
  (   2) dim  0 offset   0
  (   3) dim  0 offset   0
  (   4) dim  2 offset   2
  (   5) dim  2 offset   7
  (   6) dim  2 offset  12
  (   7) dim  2 offset  17
  (   8) dim  2 offset  22
  (   9) dim  0 offset  25
  (  10) dim  0 offset  25
  (  11) dim  0 offset  25
  (  12) dim  0 offset  25
  (  13) dim  0 offset  25
  (  14) dim  0 offset  25
  (  15) dim  0 offset  25
  (  16) dim  0 offset  25
  field 2 with 1 components # pressure scalar field
Process 0:
  (   0) dim  0 offset   0
  (   1) dim  0 offset   0
  (   2) dim  0 offset   0
  (   3) dim  0 offset   0
  (   4) dim  1 offset   4
  (   5) dim  1 offset   9
```

```
  (   6) dim  1 offset  14
  (   7) dim  1 offset  19
  (   8) dim  1 offset  24
  (   9) dim  0 offset  25
  (  10) dim  0 offset  25
  (  11) dim  0 offset  25
  (  12) dim  0 offset  25
  (  13) dim  0 offset  25
  (  14) dim  0 offset  25
  (  15) dim  0 offset  25
  (  16) dim  0 offset  25
Proc 0 local vector # 5 nondimensionalized values per point: displacement (2), velocity (2), pressure (1)
Vec Object: unknown 1 MPI processes
  type: seq
-0.999 # offset  0, point 4, x-displacement
-4.2 # offset 1, point 4, y-displacement
-9.99 # offset 2, point 4, x-velocity
-9.99 # offset 3, point 4, y-velocity
-9990. # offset 4, point 4, pressure
-0.999 # offset 5, point 5, x-displacement
0.6
-9.99
-9.99
-9990.
-0.999 # offset 10, point 6, x-displacement
0.
-9.99
-9.99
-9990.
-0.999 # offset 15, point 7, x-displacement
-0.6
-9.99
-9.99
-9990.
-0.999 # offset 20, point 8, x-displacement
4.2
-9.99
-9.99
-9990.
```

In tests in which we compare a computed field against one from an analytical solution using DMPlexComputeL2DiffLocal() and the fields do not agree, it is generally helpful to determine which pieces do not agree. The DMPlex object contains an internal switch to print the point-by-point differences while computing the norm.

Turn on debugging within DMPlexComputeL2DiffLocal()

```
PetscOptionsSetValue(NULL, "-dm_plex_print_l2", "1");

// Pass DMPlex object of computed field used in DMPlexComputeL2DiffLocal.
DMSetFromOptions(solution.dmMesh());

DMPlexComputeL2DiffLocal(dm, t, query.functions(), (void**)query.contextPtrs(), solution.localVector(), &norm);
```

Debugging output from DMPlexComputeL2DiffLocal()

```
Cell 0 Element Solution for Field 0 # displacement vector field
  | -0.999 | # Values of solution field variable at vertices of cell 0
  | -4.2 |
  | -0.999 |
  | 0. |
  | -0.999 |
  | -0.6 |
    elem 0 field 0 diff 0.  # Differences at quadrature points with respect to field given by analytical function
    elem 0 field 0 diff 6.27226e-32
    elem 0 field 0 diff 2.24281e-33
    elem 0 field 0 diff 8.97125e-33
```

```
    elem 0 field 0 diff 0.
    elem 0 field 0 diff 0.
    elem 0 field 0 diff 2.24281e-33
    elem 0 field 0 diff 0.
Cell 0 Element Solution for Field 1 # velocity field vector field
  | -9.99 |
  | -9.99 |
  | -9.99 |
  | -9.99 |
  | -9.99 |
  | -9.99 |
    elem 0 field 1 diff 0.
    elem 0 field 1 diff 0.
    elem 0 field 1 diff 0.
    elem 0 field 1 diff 0.
    elem 0 field 1 diff 0.
    elem 0 field 1 diff 0.
    elem 0 field 1 diff 0.
    elem 0 field 1 diff 0.
Cell 0 Element Solution for Field 2 # pressure scalar field
  | -9990. |
  | -9990. |
  | -9990. |
    elem 0 field 2 diff 0.
    elem 0 field 2 diff 0.
    elem 0 field 2 diff 0.
    elem 0 field 2 diff 0.
  elem 0 diff 7.61795e-32
Cell 1 Element Solution for Field 0
  | -0.999 | # Values of solution field variable at vertices of cell 1
  | 0.6 |
  | -0.999 |
  | 0. |
  | -0.999 |
  | -4.2 |
    elem 1 field 0 diff 0.
    elem 1 field 0 diff 3.92016e-33
    elem 1 field 0 diff 2.24281e-33
    elem 1 field 0 diff 1.4354e-31
    elem 1 field 0 diff 0.
    elem 1 field 0 diff 0.
    elem 1 field 0 diff 2.24281e-33
    elem 1 field 0 diff 0.
Cell 1 Element Solution for Field 1
  | -9.99 |
  | -9.99 |
  | -9.99 |
  | -9.99 |
  | -9.99 |
  | -9.99 |
    elem 1 field 1 diff 0.
    elem 1 field 1 diff 0.
    elem 1 field 1 diff 0.
    elem 1 field 1 diff 0.
    elem 1 field 1 diff 0.
    elem 1 field 1 diff 0.
    elem 1 field 1 diff 0.
    elem 1 field 1 diff 0.
Cell 1 Element Solution for Field 2
  | -9990. |
  | -9990. |
  | -9990. |
    elem 1 field 2 diff 0.
    elem 1 field 2 diff 0.
    elem 1 field 2 diff 0.
    elem 1 field 2 diff 0.
  elem 1 diff 1.51946e-31
... # Output continues for values in other cells
```

Chapter 4

Coding Style

This chapter will be an appendix in the PyLith Manual.

There are a number of standard coding styles for programming languages, notably PEP8 for Python. For PyLith we try to be consistent in naming conventions across Python and C++ while following a subset of the used in PETSc and PEP8 with documentation styles consistent with Doxygen.

> **Important**
>
> We use 4 spaces for indentation. Configure your editor to use spaces instead of tabs.

The principal guidelines include:

- Naming conventions
  - Use self-documenting names.
  - Avoid single letter variables. Choose names that are readily found via searches across single or multiple files (e.g., grep).
  - Class names are generally nouns and methods are verbs.
  - Class names use upper camel case, e.g., TimeDependent.
  - Public method names use camel case, e.g., computeRHSResidual().
  - Protected and private method names use camel case preceded by an underscore, e.g., _setFEKernelsRHSResidual().
  - In C++ data members are private and use camel case preceded by an underscore, e.g., _gravityField.
  - In Python data members are public and use camel case, e.g., self.gravityField.
  - Local variables use camel case, e.g., numIntegrators.
- Comments
  - List authors, copyright, and license info at the very beginning of every file.
  - For every class method, describe its function and include a description for each argument. For Python this is done in the docstring of the method, and for C++ this is done in a doxygen style comment immediately before the method declaration in the header file.
  - Document nontrivial algorithms and any assumptions.
- Error checking
  - PyLith should never crash.
  - All user errors should be trapped as early as possible and reported with an informative error message. If possible, suggest ways to correct the error.

  - Messages for internal errors should indicate the location in the code where the error was trapped.
  - All pointers should be checked for null values before use.
  - Check the return values for all calls to functions in external libraries.
- Testing
  - All methods should be covered by unit tests.
  - All functionality should be covered by full-scale tests.

4.1 Error Checking

Our philosophy is that PyLith should never crash. If it encounters a fatal error, then it should generate an appropriate error message and abort. In C++ we throw std::runtime_error exceptions for errors resulting from user input and std::logic_error exceptions for internal inconsistencies or logic errors. In Python we use standard exception objects.

Additional protections against crashing include: using asserts to verify pointers are non-null before using them and using the PYLITH_CHECK_ERROR macro to check the return value after *every* call to a PETSc function.

Example of using assert()

```
assert(_solution); // Verify _solution is not NULL.

// Initialize integrators.
const size_t numIntegrators = _integrators.size();
for (size_t i = 0; i < numIntegrators; ++i) {
    assert(_integrators[i]); // Verify _integrators[i] is not NULL.
    _integrators[i]->initialize(*_solution);
} // for
```

> ★ **Tip**
>
> For production runs only, we often build with -DNDEBUG to remove assert() calls.

Example of using PYLITH_CHECK_ERROR macro

```
PetscErrorCode err = TSGetTimeStep(ts, &dt); PYLITH_CHECK_ERROR(err);
```

In combination with the above procedures, we also make use of the Pyre journals to display warnings and errors to facilitate debugging. The journals provide the file name and line number along with the message. By default, Pyre journals for errors are turned on and those for warnings and debugging are turned off.

Example of using Pyre journals and standard exceptions

```
switch (bitUse) {
case 0x1:
    _bcKernel = pylith::fekernels::TimeDependentFn::initial_scalar;
    break;
case 0x2:
    _bcKernel = pylith::fekernels::TimeDependentFn::rate_scalar;
    break;
case 0x0:
    PYLITH_COMPONENT_WARNING("Dirichlet␣BC␣provides␣no␣constraints.");
    break;
default:
    PYLITH_COMPONENT_ERROR("Unknown␣combination␣of␣flags␣for␣Dirichlet␣BC␣terms␣"
        << "(useInitial="<<_useInitial<<",␣useRate="<<_useRate<<").");
    throw std::logic_error("Unknown␣combination␣of␣flags␣for␣Dirichlet␣BC␣terms.");
} // switch
```

4.2 C/C++

4.2.1 Object Definition Files

Object definition (header) files use the `.hh` suffix. C header files use the `.h` suffix. The following code excerpt demonstrates the conventions we use in formatting header files and including comments.

> **⚠ Important**
>
> *All* declarations of class methods should include a description of what the method does and a description of each argument and the return value if it is not void.

Sample C++ declaration (header) file

```
// ======================================================================
//
// Brad T. Aagaard, U.S. Geological Survey
// Charles A. Williams, GNS Science
// Matthew G. Knepley, The State University of New York at Buffalo
//
// This code was developed as part of the Computational Infrastructure
// for Geodynamics (http://geodynamics.org).
//
// Copyright (c) 2010-2018 University of California, Davis
//
// See COPYING for license information.
//
// ======================================================================
//

/* Next list the file name with the relative path of the file along with a
 * brief description.
 */

/**
 * @file libsrc/problems/Problem.hh
 *
 * @brief C++ object that manages formulating the equations.
 */

// Use full namespace in header guard.
#if !defined(pylith_problems_problem_hh)
#define pylith_problems_problem_hh

// Include directives ---------------------------------------------------
/* Order is local forward declarations, then local header files,
 * before other header files, and then system header files.
 */
#include "problemsfwd.hh" // forward declarations

#include "pylith/utils/PyreComponent.hh" // ISA PyreComponent

#include "pylith/feassemble/feassemblefwd.hh" // HASA IntegratorPointwise
#include "pylith/topology/topologyfwd.hh" // USES Mesh, Field
#include "pylith/meshio/meshiofwd.hh" // HASA OutputManager
#include "spatialdata/units/unitsfwd.hh" // HASA Nondimensional
#include "spatialdata/spatialdb/spatialdbfwd.hh" // HASA GravityField

#include "pylith/utils/petscfwd.h" // USES PetscVec, PetscMat

#include "pylith/utils/array.hh" // HASA std::vector
```

```
// Problem -------------------------------------------------------
/* Provide description of class.
 */

/** Reform the Jacobian and residual for the problem.
 *
 * We cast the problem in terms of F(t,s,\dot{s}) = G(t,s), s(t0) = s0.
 *
 * In PETSc time stepping (TS) notation, G is the RHS, and F is the I
 * function (which we call the LHS).
 *
 */
class pylith::problems::Problem : public pylith::utils::PyreComponent {
  /* Order of declarations is:
   *
   * 1. Friend classes.
   * 2. Public enums
   * 3. Public structs
   * 4. Public methods.
   * 5. Protected methods.
   * 6. Private methods
   * 7. Protected members.
   * 8. Private members.
   * 9. Methods not implemented.
   *
   * Within each group, we generally order the methods by:
   * 1. Constructor/destructors.
   * 2. Accessors.
   * 3. Other methods.
   *
   * Use the full namespace when declaring data members and method
   * arguments to avoid ambiguity.
   *
   * Method arguments are listed one per line.
   *
   * Include opening braces at the end of a line. Use a comment to
   * document all closing braces.
   *
   * Before every member method, describe what the method does and
   * include a description for every argument. We use Doxygen
   * syntax.
   */

  friend class TestProblem;   // unit testing

    // PUBLIC ENUM ////////////////////////////////////////////////////
public:

    enum SolverTypeEnum {
        LINEAR, // Linear solver.
        NONLINEAR, // Nonlinear solver.
    };   // SolverType


    // PUBLIC MEMBERS /////////////////////////////////////////////////
public:

    // Constructor
    Problem(void);

    /// Destructor
    virtual ~Problem(void);

    /* We call the deallocate method before calling PetscFinalize() to
     * deallocate any memory allocated using PETSc. In general, the
     * destructor will simply call deallocate().
     */
```

```
/// Deallocate PETSc and local data structures.
void deallocate(void);

/** Set solver type.
 *
 * @param[in] value Solver type.
 */
void solverType(const SolverTypeEnum value);

/** Get solver type.
 *
 * @returns Solver type.
 */
SolverTypeEnum solverType(void) const;

/** Set manager of scales used to nondimensionalize problem.
 *
 * @param[in] dim Nondimensionalizer.
 */
void normalizer(const spatialdata::units::Nondimensional& dim);

/** Set gravity field.
 *
 * @param[in] g Gravity field.
 */
void gravityField(spatialdata::spatialdb::GravityField* const g);

/** Set solution field.
 *
 * @param[in] field Solution field.
 */
void solution(pylith::topology::Field* field);

/** Set handles to integrators.
 *
 * @param[in] integratorArray Array of integrators.
 * @param[in] numIntegrators Number of integrators.
 */
void integrators(pylith::feassemble::IntegratorPointwise* integratorArray[],
                 const int numIntegrators);

/** Do minimal initialization.
 *
 * @param mesh Finite-element mesh.
 */
virtual
void preinitialize(const pylith::topology::Mesh& mesh);

/** Verify configuration.
 *
 * @param[in] materialIds Array of material ids.
 * @param[in] numMaterials Size of array (number of materials).
 *
 */
virtual
void verifyConfiguration(int* const materialIds,
                         const int numMaterials) const;

/** Set solution values according to constraints (Dirichlet BC).
 *
 * @param[in] t Current time.
 * @param[in] solutionVec PETSc Vec with current global view of solution.
 * @param[in] solutionDotVec PETSc Vec with current global view of time derivative of solution.
 */
void setSolutionLocal(const PylithReal t,
                      PetscVec solutionVec,
                      PetscVec solutionDotVec);
```

```
    /** Compute RHS residual, G(t,s) and assemble into global vector.
     *
     * @param[out] residualVec PETSc Vec for residual.
     * @param[in] t Current time.
     * @param[in] dt Current time step.
     * @param[in] solutionVec PETSc Vec with current trial solution.
     */
    void computeRHSResidual(PetscVec residualVec,
                            const PetscReal t,
                            const PetscReal dt,
                            PetscVec solutionVec);

    // PROTECTED MEMBERS //////////////////////////////////////////////////
protected:

    /* Use pointers to hide implementation details and speed up compilation.
     */

    pylith::topology::Field* _solution;   ///< Handle to solution field.
    pylith::topology::Field* _residual; ///< Handle to residual field.

    spatialdata::units::Nondimensional* _normalizer; ///< Nondimensionalization of scales.
    spatialdata::spatialdb::GravityField* _gravityField; ///< Gravity field.
    std::vector<pylith::feassemble::IntegratorPointwise*> _integrators;   ///< Array of integrators.
    SolverTypeEnum _solverType;   ///< Problem (solver) type.

    // NOT IMPLEMENTED ////////////////////////////////////////////////////
private:

    /* Declare expensive/fragile copy methods private, so using them
     * fails in an error at compile time.
     */

    Problem(const Problem&);   ///< Not implemented
    const Problem& operator=(const Problem&);   ///< Not implemented

}; // Problem

#endif // pylith_problems_problem_hh


// End of file
```

4.2.2 Object Implementation Files

Object implementation files use the `.cc` suffix. Inline implementation files use the `.icc` suffix and are included from the definition files. C implementation files use the `.c` suffix.

To facilitate debugging and error messages, we use the following macros:

PYLITH_METHOD_BEGIN This macro allows line numbers of source files to be included in PETSc error messages. Use this macro at the beginning of all methods using any PETSc routines as well as most other methods. We don't use this macro in destructors because many of them are called *after* PetscFinalize. We also do not use this macro in trivial or inline methods that do not call any PETSc routines.

PYLITH_METHOD_END Use the macro at the end of all methods that begin with PYLITH_METHOD_BEGIN and return void.

PYLITH_RETURN_END Use this macro at the end of all methods that begin with PYLITH_METHOD_BEGIN and return non-void values.

PYLITH_CHECK_ERROR after *every* call to a PETSc function to check the return value.

PYLITH_JOURNAL_DEBUG Use this macro immediately after PYLITH_METHOD_BEGIN in methods of all objects that inherit from GenericComponent.

PYLITH_COMPONENT_DEBUG Use this macro immediately after PYLITH_METHOD_BEGIN in methods of all objects that inherit from PyreComponent. Non-abstract classes should call PyreComponent::name() in the constructor. We recommend using a static data member for the name with the lowercase name matching the Pyre component, e.g., timedependent.

Sample C++ definition (implementation) file

```
// ======================================================================
//
// Brad T. Aagaard, U.S. Geological Survey
// Charles A. Williams, GNS Science
// Matthew G. Knepley, The State University of New York at Buffalo
//
// This code was developed as part of the Computational Infrastructure
// for Geodynamics (http://geodynamics.org).
//
// Copyright (c) 2010-2015 University of California, Davis
//
// See COPYING for license information.
//
// ======================================================================
//

/* Order of including header files is:
 * 1. portinfo: (stuff from configure)
 * 2. Header file for this class.
 * 3. Header files for local classes.
 * 4. Header files for classes in other libraries.
 * 5. Standard header files.
 *
 * List why each header file is included (USES/HASA/HOLDSA).
 *
 * Order of method implementations should match header file.
 *
 * For each method:
 *   1. Return values should go on previous line.
 *   2. Put each method argument on a separate line.
 */
#include <portinfo>

#include "Problem.hh" // implementation of class methods

#include "pylith/topology/Mesh.hh" // USES Mesh
#include "pylith/topology/Field.hh" // USES Field

#include "pylith/feassemble/IntegratorPointwise.hh" // USES IntegratorPointwise

#include "pylith/utils/error.hh" // USES PYLITH_CHECK_ERROR
#include "pylith/utils/journals.hh" // USES PYLITH_COMPONENT_*

#include <cassert> // USES assert()
#include <typeinfo> // USES typeid()

// ----------------------------------------------------------------------
// Constructor
pylith::problems::Problem::Problem() :
    _solution(NULL),
    _solutionDot(NULL),
    _residual(NULL),
    _jacobianLHSLumpedInv(NULL),
    _normalizer(NULL),
    _gravityField(NULL),
    _integrators(0),
    _constraints(0),
```

```
    _outputs(0),
    _solverType(LINEAR)
{ // constructor
} // constructor

// ----------------------------------------------------------------------
// Destructor
pylith::problems::Problem::~Problem(void)
{ // destructor
    deallocate();
} // destructor

// ----------------------------------------------------------------------
// Deallocate PETSc and local data structures.
void
pylith::problems::Problem::deallocate(void) {
    PYLITH_METHOD_BEGIN;

    _solution = NULL; // Held by Python. :KLUDGE: :TODO: Use shared pointer.
    delete _solutionDot; _solutionDot = NULL;
    delete _residual; _residual = NULL;
    delete _jacobianLHSLumpedInv; _jacobianLHSLumpedInv = NULL;
    delete _normalizer; _normalizer = NULL;
    _gravityField = NULL; // Held by Python. :KLUDGE: :TODO: Use shared pointer.

    PYLITH_METHOD_END;
} // deallocate

// ----------------------------------------------------------------------
// Set problem type.
void
pylith::problems::Problem::solverType(const SolverTypeEnum value) {
    PYLITH_COMPONENT_DEBUG("Problem::solverType(value="<<value<<")");

    _solverType = value;
} // solverType

// ----------------------------------------------------------------------
// Get problem type.
pylith::problems::Problem::SolverTypeEnum
pylith::problems::Problem::solverType(void) const {
    return _solverType;
} // solverType

// ----------------------------------------------------------------------
// Set manager of scales used to nondimensionalize problem.
void
pylith::problems::Problem::normalizer(const spatialdata::units::Nondimensional& dim) {
    PYLITH_COMPONENT_DEBUG("Problem::normalizer(dim="<<typeid(dim).name()<<")");

    if (!_normalizer) {
        _normalizer = new spatialdata::units::Nondimensional(dim);
    } else {
        *_normalizer = dim;
    } // if/else
} // normalizer

// ----------------------------------------------------------------------
// Set gravity field.
void
pylith::problems::Problem::gravityField(spatialdata::spatialdb::GravityField* const g) {
    PYLITH_COMPONENT_DEBUG("Problem::gravityField(g="<<typeid(*g).name()<<")");

    _gravityField = g;
} // gravityField

// ----------------------------------------------------------------------
// Set solution field.
```

```
void
pylith::problems::Problem::solution(pylith::topology::Field* field)
{ // solution
    PYLITH_COMPONENT_DEBUG("Problem::solution(field="<<typeid(*field).name()<<")");

    _solution = field;
} // solution

// ----------------------------------------------------------------------
// Set integrators over the mesh.
void
pylith::problems::Problem::integrators(pylith::feassemble::IntegratorPointwise* integratorArray[],
                                       const int numIntegrators) {
    PYLITH_METHOD_BEGIN;
    PYLITH_COMPONENT_DEBUG("Problem::integrators("<<integratorArray<<",␣numIntegrators="<<numIntegrators<<")");

    assert( (!integratorArray && 0 == numIntegrators) || (integratorArray && 0 < numIntegrators) );

    _integrators.resize(numIntegrators);
    /* Declare loop variables inline. Always use braces at begin/end
     * of if and for statements.
     */
    for (int i = 0; i < numIntegrators; ++i) {
        _integrators[i] = integratorArray[i];
    } // for

    PYLITH_METHOD_END;
} // integrators

// ----------------------------------------------------------------------
// Do minimal initialization.
void
pylith::problems::Problem::preinitialize(const pylith::topology::Mesh& mesh) {
    PYLITH_METHOD_BEGIN;
    PYLITH_COMPONENT_DEBUG("Problem::preinitialzie(mesh="<<typeid(mesh).name()<<")");

    assert(_normalizer);

    const size_t numIntegrators = _integrators.size();
    for (size_t i = 0; i < numIntegrators; ++i) {
        assert(_integrators[i]);
        _integrators[i]->normalizer(*_normalizer);
        _integrators[i]->gravityField(_gravityField);
    } // for

    PYLITH_METHOD_END;
} // preinitialize

// ----------------------------------------------------------------------
// Verify configuration.
void
pylith::problems::Problem::verifyConfiguration(int* const materialIds,
                                               const int numMaterials) const {
    PYLITH_METHOD_BEGIN;
    PYLITH_COMPONENT_DEBUG("Problem::verifyConfiguration(materialIds="<<materialIds<<",␣numMaterials="<<numMaterials<

    assert(_solution);

    // Check to make sure material-id for each cell matches the id of a material.
    pylith::topology::MeshOps::checkMaterialIds(_solution->mesh(), materialIds, numMaterials);

    // Check to make sure integrators are compatible with the solution.
    const size_t numIntegrators = _integrators.size();
    for (size_t i = 0; i < numIntegrators; ++i) {
        assert(_integrators[i]);
        _integrators[i]->verifyConfiguration(*_solution);
    } // for
```

```
    PYLITH_METHOD_END;
}  // verifyConfiguration

// ----------------------------------------------------------------------
// Set solution values according to constraints (Dirichlet BC).
void
pylith::problems::Problem::setSolutionLocal(const PylithReal t,
                                            PetscVec solutionVec,
                                            PetscVec solutionDotVec) {
    PYLITH_METHOD_BEGIN;
    PYLITH_COMPONENT_DEBUG("setSolutionLocal(t="<<t<<",␣solutionVec="<<solutionVec<<",␣solutionDotVec="<<solutionDotV

    // Update PyLith view of the solution.
    assert(_solution);
    _solution->scatterVectorToLocal(solutionVec);

    if (solutionDotVec) {
        if (!_solutionDot) {
            _solutionDot = new pylith::topology::Field(_solution->mesh());
            _solutionDot->cloneSection(*_solution);
            _solutionDot->label("solutionDot");
        } // if
        _solutionDot->scatterVectorToLocal(solutionDotVec);
    } // if

    PYLITH_METHOD_END;
} // setSolutionLocal

// ----------------------------------------------------------------------
// Compute RHS residual for G(t,s).
void
pylith::problems::Problem::computeRHSResidual(PetscVec residualVec,
                                              const PylithReal t,
                                              const PylithReal dt,
                                              PetscVec solutionVec) {
    PYLITH_METHOD_BEGIN;
    PYLITH_COMPONENT_DEBUG("Problem::computeRHSResidual(t="<<t<<",␣dt="<<dt<<",␣solutionVec="<<solutionVec<<",␣residu

    assert(residualVec);
    assert(solutionVec);
    assert(_solution);

    // Update PyLith view of the solution.
    PetscVec solutionDotVec = NULL;
    setSolutionLocal(t, solutionVec, solutionDotVec);

    // Sum residual contributions across integrators.
    _residual->zeroLocal();
    const size_t numIntegrators = _integrators.size();
    assert(numIntegrators > 0); // must have at least 1 integrator
    for (size_t i = 0; i < numIntegrators; ++i) {
        _integrators[i]->computeRHSResidual(_residual, t, dt, *_solution);
    } // for

    // Assemble residual values across processes.
    PetscErrorCode err = VecSet(residualVec, 0.0); PYLITH_CHECK_ERROR(err); // Move to TSComputeIFunction()?
    _residual->scatterLocalToVector(residualVec, ADD_VALUES);

    PYLITH_METHOD_END;
} // computeRHSResidual

// End of file
```

4.3 Python

Example of Python source code using our preferred coding style:

Sample Python source code

```
# ----------------------------------------------------------------------
#
# Brad T. Aagaard, U.S. Geological Survey
# Charles A. Williams, GNS Science
# Matthew G. Knepley, The State University of New York at Buffalo
#
# This code was developed as part of the Computational Infrastructure
# for Geodynamics (http://geodynamics.org).
#
# Copyright (c) 2010-2015 University of California, Davis
#
# See COPYING for license information.
#
# ----------------------------------------------------------------------
#
# Next list the filename with the relative path of the file along with a
# brief description.
#
# @file pylith/problems/Problem.py
#
# @brief Python abstract base class for crustal dynamics problems.
#
# Factory: problem.

# Order of imports should be
# 1. standard modules
# 2. other modules
# 3. PyLith modules (parent classes first).
#
# Prefix imports from this directory with . for clarity.
#
# The SWIG interface to the C++ object is imported as ModuleProblem in
# order to avoid clashing with the Python Problem class.

from pylith.utils.PetscComponent import PetscComponent
from .problems import Problem as ModuleProblem

# ITEM FACTORIES ////////////////////////////////////////////////////

# Define any factory methods needed in the Inventory class.
def materialFactory(name):
    """
    Factory for material items.
    """
    from pyre.inventory import facility
    from pylith.materials.ElasticityPlaneStrain import ElasticityPlaneStrain
    return facility(name, family="material", factory=IsotropicLinearElasticityPlaneStrain)

class Problem(PetscComponent, ModuleProblem):
    """Python abstract base class for crustal dynamics problems.

    INVENTORY

    Properties
      - *solver_type* Type of solver to user.

    Facilities
      - *solution* Solution field for problem.
      - *materials* Materials in problem.

    FACTORY: problem. List the factory if one exists.
```

```
    """

    import pyre.inventory

    # Usually, we put all arguments on a single line. If the
    # function call is really long, break the arguments into
    # logical pieces, usually one argument per line.
    solverTypeStr = pyre.inventory.str(
        "solver",
        default="linear",
        validator=pyre.inventory.choice(["linear", "nonlinear"])
    )
    solverTypeStr.meta['tip'] = "Type of solver to use ['linear', 'nonlinear']."

    from Solution import Solution
    solution = pyre.inventory.facility("solution", family="solution", factory=Solution)
    solution.meta['tip'] = "Solution field for problem."

    from pylith.materials.Homogeneous import Homogeneous
    materials = pyre.inventory.facilityArray(
        "materials",
        itemFactory=materialFactory,
        factory=Homogeneous
    )
    materials.meta['tip'] = "Materials in problem."


    # PUBLIC METHODS /////////////////////////////////////////////////////

    def __init__(self, name="problem"):
        """Constructor.
        """
        PetscComponent.__init__(self, name, facility="problem")

        # Initialize all data members not in the inventory.
        self.mesh = None
        return

    def preinitialize(self, mesh):
        """Do minimal initialization.
        """
        # On process 0 only, print progress information to info journal.
        from pylith.mpi.Communicator import mpi_comm_world
        comm = mpi_comm_world()
        if 0 == comm.rank:
            self._info.log("Performing minimal initialization before verifying configuration.")

        # Pass information to corresponding C++ object.
        #
        # For all calls to the C++ interaface, call using the
        # class name and pass self as an argument to make it clear
        # that this is calling a C++ method and not a Python method.
        ModuleProblem.identifier(self, self.aliases[-1])

        if self.solverTypeStr == "linear":
            ModuleProblem.solverType(self, ModuleProblem.LINEAR)
        elif self.solverTypeStr == "nonlinear":
            ModuleProblem.solverType(self, ModuleProblem.NONLINEAR)
        else:
            raise ValueError("Unknown solver type '%s'." % self.solverTypeStr)

        # Do minimal setup of solution.
        self.solution.preinitialize(mesh, self.normalizer)
        ModuleProblem.solution(self, self.solution.field)

        # Preinitialize materials
        for material in self.materials.components():
            material.preinitialize(mesh)
```

```
        ModuleProblem.preinitialize(self, mesh)
        return

    def initialize(self):
        """Initialize integrators and constraints.
        """
        # On process 0 only, print progress information to info journal.
        from pylith.mpi.Communicator import mpi_comm_world
        comm = mpi_comm_world()
        if 0 == comm.rank:
            self._info.log("Initializing problem.")

        ModuleProblem.initialize(self)
        return

    def run(self, app):
        """Solve the problem.
        """
        # Generate error if method is not implemented in child class.
        raise NotImplementedError("run() not implemented.")
        return

    # PRIVATE METHODS /////////////////////////////////////////////////

    def _configure(self):
        """Set data members based using inventory.
        """
        PetscComponent._configure(self)

        return

# FACTORIES ////////////////////////////////////////////////////////////

# Define any facility factories.
def problem():
    """
    Factory associated with Problem.
    """
    return Problem()


# End of file
```

4.4 Formatting C++ and Python Source Files

We use `autopep8` and `uncrustify` to format Python and C++ source files, respectively. The corresponding configuration files are `autopep8.cfg` and `uncrustify.cfg` in the `doc/developer` directory. The Python script `doc/developer/format_source.py` is a handy utility for calling `autopep8` and `uncrustify` with the appropriate arguments and formatting multiple files.

Formatting Python and C++ source code using `format_source.py`

```
# Prerequisites:
#  * autopep8
#  * uncrustify
#
# Run from the top-level source directory.

# Format a C++ file
$ doc/developer/format_source.py --cplusplus=libsrc/pylith/materials/Material.cc

# Format C++ files in a directory
```

```
$ doc/developer/format_source.py --cplusplus=libsrc/pylith/materials/*.cc

# Format a Python file
$ doc/developer/format_source.py --python=pylith/materials/Material.py

# Format Python files in a directory
$ doc/developer/format_source.py --python=pylith/materials/*.py
```