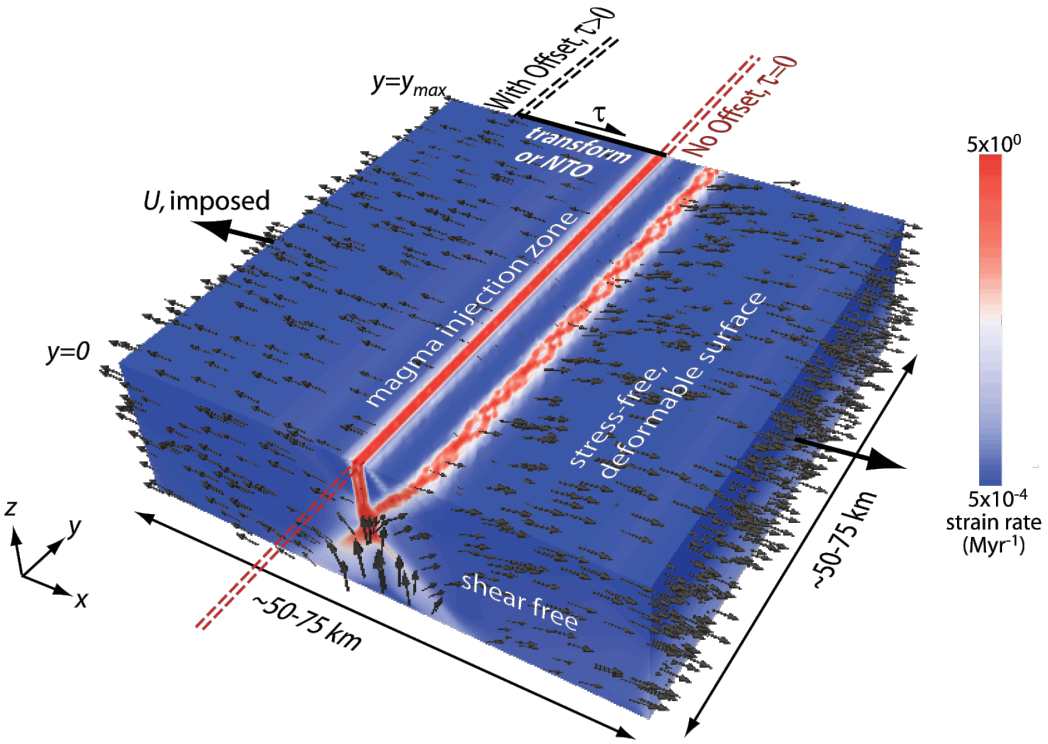
COMPUTATIONAL INFRASTRUCTURE FOR GEODYNAMICS (CIG)
VICTORIA PARTNERSHIP FOR ADVANCED COMPUTING (VPAC)
MONASH UNIVERSITY

Gale

User Manual
Version 1.6.1



Walter Landry
Luke Hodkinson
Susan Kientz

www.geodynamics.org

Gale

Walter Landry and Luke Hodkinson
Version 1.6.1

November 10, 2010

About the cover: A 3D simulation of a mid-ocean ridge courtesy of Garrett Ito.

Contents

List of Figures

Chapter 1

Preface

1.1 Who Will Use Gale?

The main audience for Gale is research scientists interested in modeling tectonic processes on long geological time scales. Examples of problems that can be solved are the development of tectonic structures associated with extension and compression, especially where localization is important. You do not have to be an expert in finite element analysis or scientific computing to use this software.

1.2 Citation

Computational Infrastructure for Geodynamics (CIG) is making this source code available to you in the hope that the software will enhance your research in geophysics. The underlying C code for the finite element package was donated to CIG in July of 2005. A number of individuals have contributed a significant portion of their careers toward the development of Gale. It is essential that you recognize these individuals in the normal scientific practice by making appropriate acknowledgments.

The code is based on the method described in

- Moresi, L.N., F. Dufour, and H.-B. Mühlhaus (2003), A Lagrangian integration point finite element method for large deformation modeling of viscoelastic geomaterials, *J. Comp. Phys., 184*, 476-497.

The code was originally developed by the Victorian Partnership for Advanced Computing (VPAC) and Louis Moresi's group at Monash University. Walter Landry of CIG and Luke Hodkinson of VPAC have enhanced the code to satisfy the requirements of the long-term tectonics community. Roger Buck, Gus Correa, Robert Bialas, Guillaume Duclaux, John Sheehan, Garrett Ito, Noah Fay, Neil de Laplante, Matthieu Quinquis, Patrice Rey, Lara O'Dwyer, Louise Kellogg, Laetitia Le Pourhiet, Leonardo Da Cruz, Jolante Van Wijk, Tristan Salles, Mark Fleharty, Taichi Sato, and Lester Anderson provided valuable user testing. The Gale team requests that in your oral presentations and in your papers that you indicate your use of this code and acknowledge the authors of the code, CIG (`www.geodynamics.org`), Victoria Partnership for Advanced Computing (`www.vpac.org`), and Monash University (`www.monash.edu`).

1.3 Support

Gale development is supported by a grant from the National Science Foundation to CIG, managed by the California Institute of Technology, under Grant No. EAR-0406751. However, most of the software components below Gale have been developed by the Victoria Partnership for Advanced Computing (VPAC) and Monash University. Some of the support for VPAC has come from subawards from CIG.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

1.4 Gale History

Gale arose from discussions at an NSF-sponsored workshop on Tectonic Modeling held in Breckenridge, Colorado, in June 2005; see Geodynamic Modeling of Tectonics Processes 2005 workshop report (`www.geodynamics.org/cig/workinggroups/long/workshops/2005/issues`). At that workshop, members of the tectonics community advocated that CIG develop a new open-source software package for solving tectonic problems. Existing private codes have seen a great deal of use in crustal and lithospheric deformation problems such as orogenesis, rifting, and subduction. They have also been coupled with surface erosion models, as well as being employed for deeper mantle dynamics problems. Gale is an open-source code that is intended to cover these research areas, with the addition of 3D capability.

The development of Gale was jump-started by building on top of Underworld [3], a mantle convection code developed by Louis Moresi's group at Monash and the Victorian Partnership for Advanced Computing (VPAC). Underworld was created as a parallel version of Ellipsis3D [6], a mantle convection code which grew out of CitCom [7]. Walter Landry of CIG and Luke Hodkinson of VPAC are the primary developers of the Gale-specific components.

Chapter 2

Introduction

2.1 About Gale

Gale is a parallel, two- or three-dimensional code that solves problems related to orogenesis, rifting, and subduction. Gale starts with a collection of particles to track material properties such as density and, for strain-softening materials, strain. At each point in time, a finite element mesh is superimposed over the particles. This allows Gale to simulate problems with large deformations and irregular boundaries.

CIG developed Gale in response to community demand by building on existing work by VPAC and Louis Moresi's group at Monash University. The code is being released under the GNU General Public License.

2.2 Gale Computational Approach and Governing Equations

2.2.1 Infrastructure

Particles are the fundamental object in Gale. Particles store all of the material properties, including density, strain, and thermal diffusivity. A logically regular finite element mesh is created at each time step. Material properties are interpolated from the particles to the mesh, and the Stokes equations are then solved on that mesh. This mesh can become quite distorted, as the boundaries of the mesh will be stretched to cover the particles wherever they go. Once the Stokes equations are solved, the mesh is retained only to provide a good initial guess for the next time step.

2.2.2 Units

Gale has no internal knowledge of units. So if you tell Gale that a box is 10 units across, it does not know or care whether it is 10 cm or 10 km. You only have to make sure that you are consistent. For example, if you give velocities in cm/year, make sure that your viscosities and lengths also use cm and years. However, you may have to scale your units to make the solver work (see Section 2.2.8.2).

2.2.3 Basic Equations

We start by decomposing the stress tensor σ into pressure p and deviatoric stress τ

$$\sigma_{ij} = \tau_{ij} - p\delta_{ij}, \tag{2.1}$$

where δ is the Kronecker delta. In its simplest form, Gale solves a conservation equation for momentum

$$\tau_{ij,j} - p_{,i} = 0, \tag{2.2}$$

subject to the (incompressible) continuity equation

$$v_{i,i} = 0, \tag{2.3}$$

where v is the velocity. We use the convention that repeated indices (e.g., $v_{i,i}$) imply a sum over all dimensions. So in three dimensions

$$v_{i,i} \equiv v_{x,x} + v_{y,y} + v_{z,z}. \tag{2.4}$$

Note that there is no explicit time dependency in the momentum Equation 2.2. Gale simulates creeping flows, so acceleration terms are neglected and material motion evolves through a series of equilibria. If your boundary condition has a time dependent component, then you may infer a time. For example, if the boundaries move inwards at 1 mm/sec, then the solution when the boundaries have moved 5 mm would correspond to 5 seconds.

Assuming a simple Newtonian fluid, we can write τ in terms of the rate of strain tensor $\dot{\varepsilon}$

$$\tau_{ij} = 2\eta\dot{\varepsilon}_{ij} \equiv \eta\left(v_{i,j} + v_{j,i}\right), \tag{2.5}$$

where η is the viscosity.

Note that equation 2.2 has no dependence on the magnitude of the velocity. Rather, only the derivative of the velocity comes into play. This means that, in the absence of boundary conditions, you can take a valid solution, add 10^{40} to all of the velocity components, and you will still have a valid solution. In practice, if you do not specify the velocity somewhere, the code will have problems finding a solution.

This means that, in 2D, you must specify v_x and v_y for at least in one point in your simulation (it does not have to be the same point).

2.2.4 Gravity

Equations 2.2 and 2.3 do not include the effect of gravity. Gravity is accounted for by adding a body force term to Equation 2.2

$$\tau_{ij,j} - p_{,i} = f_i, \tag{2.6}$$

where

$$\begin{array}{rcl} f_x &=& 0 \\ f_y &=& -g\rho \\ f_z &=& 0 \end{array}. \tag{2.7}$$

Note that the vertical direction is in the y direction, not the z direction. This makes it easy to switch between 2D and 3D models without rewriting the entire input file.

2.2.5 Divergence Forces

It can sometimes be convenient to create a model where material is created within the simulation. For example, magma chambers can be fed through small channels that emanate from far away, outside the simulation. Simulating these small channels would be too computationally expensive. Instead, we can model the magma as just being created in the chamber.

You do this by adding a divergence term to the continuity Equation (Eq. 2.3),

$$v_{i,i} = d, \tag{2.8}$$

where d is a scalar that can depend on anything: time, space, strain, etc. In this form, the divergence modifies the velocity. However, since the velocity and pressure are not really independent, you can also think of it as setting a condition on the pressure. For example, consider a one dimensional isoviscous case with no gravity. You can write the momentum Equation (Eq. 2.2) as

$$\eta\left(v_{i,jj} + v_{j,ij}\right) + p_{,i} = 0. \tag{2.9}$$

In one dimension, Equation 2.8 becomes

$$v_{x,x} = d, \tag{2.10}$$

which implies

$$2\eta d_{,x} + p_{,x} = 0. \tag{2.11}$$

So if you specify the divergence as a constant in one region and zero outside, that is equivalent to specifying a pressure drop across the boundary of the region. This result also holds in general for spherical and ellipsoidal regions, although not if the viscosity varies across the boundary of the region.

2.2.6 Rheology

Gale incorporates a number of different rheologies and allows you to create your own. For more complicated, non-linear rheologies, Gale still solves Equation 2.5 for the velocity. However, because the viscosity may depend on the velocity and its derivatives, Gale now has to iterate until it reaches a self consistent solution for the viscosity and velocity. See Section 2.2.8.4 for more details.

For details on the existing rheologies, see Section A.5. To create your own rheology, see Chapter 7 for guidance.

2.2.7 Temperature

Equation 2.6 does not explicitly include the effect of temperature and heat transfer. Temperature can be implicitly included by using a temperature dependent viscosity and/or modifying the gravitational force to have a thermal buoyancy term. To make the simulation completely self consistent, we solve the energy equation

$$\frac{\partial T}{\partial t} + v \cdot \nabla T = \kappa \nabla^2 T + Q, \tag{2.12}$$

where T is the temperature, κ is the thermal diffusivity, and Q is the rate of energy production (e.g., from radiogenic sources). Note that Equation 2.12 introduces time into the equation.

2.2.8 Numerical Solution

2.2.8.1 Artificial Compressibility

In Gale, the velocity and pressure are represented by linear (Q1) finite elements (formally, this is called a Q1-Q1 scheme). Normally, this formulation is unstable. To stabilize it, we follow Dohrmann & Bochev [14] [15] and add a compressibility term to the divergence equation

$$v_{i,i} + Cp = d, \tag{2.13}$$

where

$$\begin{array}{rcl} C & \equiv & \frac{1}{\nu}\left(M - D\right) \\ M & \equiv & \int_{\Omega_e} \psi(x)\psi^T(x) d\Omega_e \\ D & \equiv & \int_{\Omega_e} d\Omega_e \end{array},$$

Ω_e is the finite element, and $\psi(x)$ is its basis function. That is, we add a compressibility that is scaled by the viscosity and proportional to the mass matrix and the area. So as we increase resolution, both of these terms scale as the area of the element and go to zero.

Prior versions of Gale used constant elements for the pressure (a Q1-P0 scheme). That scheme admits solutions that have an arbitrarily sized checkerboard pressure term. For simple viscous problems, the pressure is not used any further than getting the velocity solution, which the checkerboard term does not affect. However, with a yielding rheology, pressure plays a critical role in determining when materials yield. We can moderate this checkerboard instability by averaging the pressure at the nodes of the elements. In practice, this does not solve all of the problems. Figure 2.1 shows a simple subduction model with the old Q1-P0 scheme and averaged pressures. Figure 2.2 shows the new Q1-Q1 scheme.

These instabilities in the old Q1-P0 scheme also slowed down convergence, while the new Q1-Q1 scheme is dramatically faster.

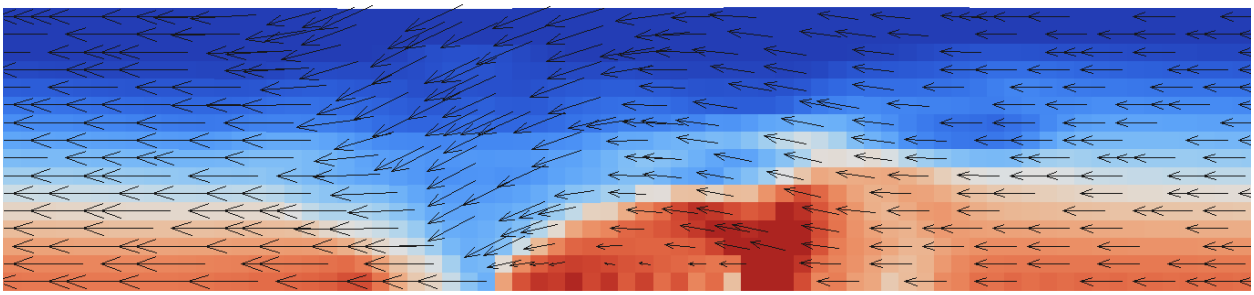


Figure 2.1: Pressure for a subduction model using Q1-P0 elements and averaged pressures. Notice that, even when averaging the pressure, there is an even-odd artifact centered on the bottom.

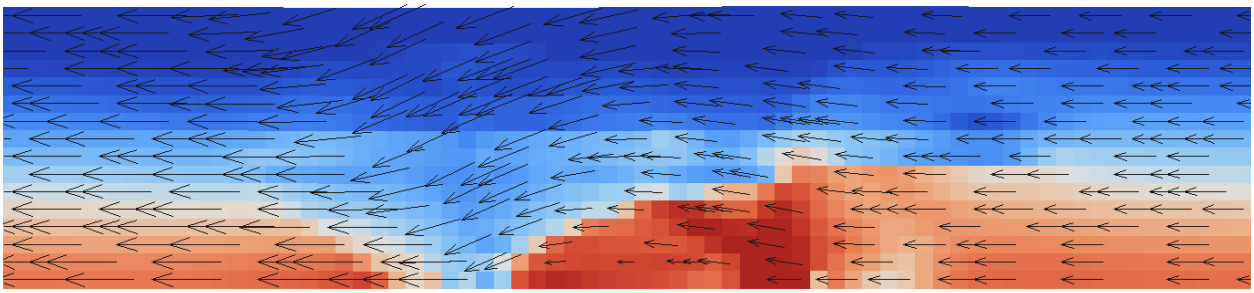


Figure 2.2: Strain rate invariant for a subduction model using Q1-Q1 elements

2.2.8.2 Scaling

The new scheme is not without its downsides. First of all, the new code does not scale the physical quantities as well as the old code. In particular, note that the units for Equations 2.6 and 2.13 are different. Equation 2.13 has been divided by viscosity and multiplied by length. So, if you have a viscosity of $10^{25} Pa \cdot s$ and you express your viscosities in $Pa \cdot s$, the compressibility pressure term will become too small and the code will be unable to solve. One workaround is to scale the units of time and mass (e.g., seconds and kg) so that the viscosities are around 1. So if the viscosities are around 10^{25}, then scale time and mass as

$$\begin{aligned} s &\rightarrow 10^{25} s, \\ kg &\rightarrow 10^{50} kg. \end{aligned}$$

This implies that a viscosity of $10^{25} Pa \cdot s$ becomes 1, and a velocity of $10^{-11} m/s$ becomes 10^{14}. Viscosities become small and velocities become large.

Scaling it this way means that you do not have to scale the length or stresses. You also do not have to scale the density or gravity, since they only appear when multiplied by each other. The main things you have to change are the viscosities and velocities. For thermal simulations, you also have to scale the thermal diffusivity and heat production rate. If you are using the `NonNewtonian` Rheology (see Section A.5.2.4), you have to scale `A`, `refStrainRate`, `minViscosity`, and `maxViscosity`. Note that the scaling for `A` is non-trivial. It has units of $s^{-1/n} Pa^{-1}$, so in this case $A_{new} = A_{old} \left(10^{25}\right)^{1/n}$.

2.2.8.3 Hydrostatic Pressure

Ideally, the compressibility term should be applied to the dynamic pressure, not the hydrostatic pressure. This is not a problem in simple sandbox simulations, where the dynamic pressure is about the same size as the hydrostatic pressure. However, in geological models, the hydrostatic pressure is much, much larger than the dynamic pressure. This means that the compressibility term is far too large, leading to excessive compressibility. As Figure 2.3 shows, this can completely alter the dynamics.

To fix this, you can subtract out the hydrostatic term. You can write Equation 2.6 as

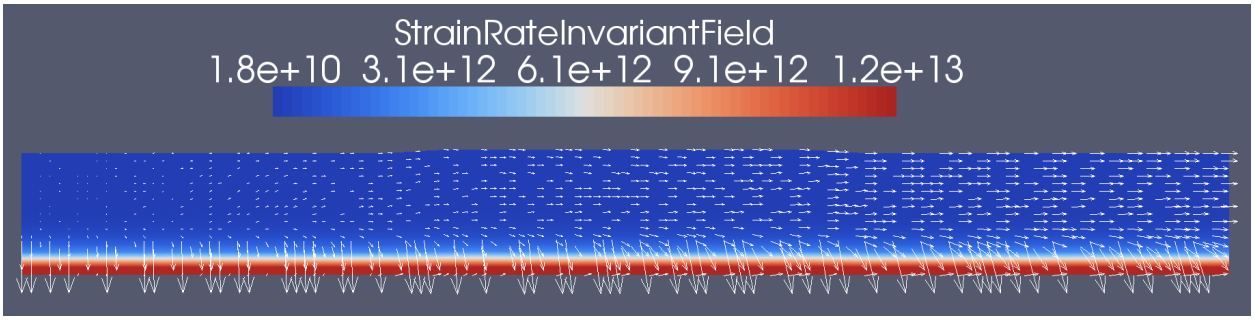


Figure 2.3: Strain rate invariant and velocity arrows for a geologic model without hydrostatic correction. The compressibility is so large on the bottom that the whole region falls down, dominating the dynamics.

$$\tau_{ij,j} - p_{0,i} - \delta p_{,i} = f_{0i} + \delta f_i, \tag{2.14}$$

where p_0 is the hydrostatic pressure, and f_0 is the background buoyancy force that leads to a hydrostatic pressure. Since p_0 and f_0 are constructed to be hydrostatic, they cancel each other. You can then rewrite Equations 2.6 and 2.8 as

$$\begin{array}{rcl} \tau_{ij,j} - \delta p_{,i} & = & \delta f_i \\ v_{i,i} + C\delta p & = & d \end{array} . \tag{2.15}$$

Note that if you are simulating a simple viscous problem in a box, you never even use the complete pressure. In the general case such as with a yielding rheology, the viscosity may depend on the overall pressure.

2.2.8.4 Uzawa Algorithm

Using standard finite-element techniques, you can collect all of the terms together and represent them in matrix form

$$\begin{pmatrix} K & G \\ G^T & C \end{pmatrix} \begin{pmatrix} v \\ p \end{pmatrix} = \begin{pmatrix} f \\ d \end{pmatrix}, \tag{2.16}$$

where K is a complicated submatrix depending on material properties, G is the simple gradient operator, C is the artificial compressibility term, f is the body force (e.g., gravity), and d is the divergence term. This implies the separate relations

$$\begin{array}{rcl} Kv + Gp & = & f \\ G^T v + Cp & = & d \end{array} . \tag{2.17}$$

In order to solve this, it turns out to be useful to solve a simplified form of

$$\left(G^T K^{-1} G\right) z = r,$$

where r is given and z is unknown. Starting from an approximate solution to this equation makes it easier to find a solution to the complete equation. The choice used in Gale is to approximate $G^T K^{-1} G$ with

$$Q \equiv G^T \left[diag\,(K)\right]^{-1} G.$$

Q is known as a preconditioner. To actually solve Equation 2.17, we use the Uzawa algorithm [5]. In particular, the steps are

1. Start with an initial guess of q_0 of the pressure-like variable.
2. Solve $K\,u_0 = f - Gq_0$ for u_0.

3. Calculate the residual $r_0 = G^T u_0 + C q_0 - d$.
4. do
5. k=1
6. Solve $Q z_{k-1} = r_{k-1}$ for z_{k-1}.
7. if k==1
8. $s_1 = z_0$
9. else
10. $\beta = \frac{z_{k-1}^T r_{k-1}}{z_{k-2}^T r_{k-2}}$
11. $s_k = r_{k-1} + \beta s_{k-1}$
12. end if
13. Solve $K u^* = G s_k$ for u^*.
14. $\alpha = \frac{z_{k-1}^T r_{k-1}}{s_k^T (G^T u^* - M s_k)}$
15. $q_{k+1} = q_k + \alpha s_k$
16. $u_{k+1} = u_k - \alpha u^*$
17. $r_k = r_{k-1} - \alpha \left(G^T u^* - M s_k \right)$
18. k=k+1
19. while $\left(u_{k+1} - u_k\right) / u_{k+1} > linear\, tolerance$

That will give us a single solution to Equation 2.17 with a certain viscosity. However, because of yielding or strain-rate dependent rheologies, the viscosity will change and the solution will not be consistent. To make it consistent, we need to recompute the viscosities with the new solution for the pressure and velocity. Then we solve Equation 2.17 again using our previous solution for the pressure as a starting point. We continue this process until the change in the velocity is less than the non-linear tolerance.

Chapter 3

Installation and Getting Help

3.1 Introduction

Installation of Gale on a desktop or laptop machine is, in most cases, very easy. Binary packages have been created for the most common platforms, i.e., Linux, Mac OS X, and Windows. Installation on other architectures or on parallel machines requires building the software from the source code, which can be difficult for inexperienced users.

3.2 Binaries

If you do not need to run on parallel machines, the easiest way to install Gale is to download binaries for your platform from the Gale website (`geodynamics.org/cig/software/packages/long/gale/`). Then you can run Gale from the command line or DOS prompt. CIG provides binaries for Linux, Mac OS X (10.4 or greater), and Windows (2000 and XP).

3.3 Building from Source

Read this only if the binaries are not sufficient for you.

3.3.1 System Requirements

Gale works on a variety of computational platforms and has been tested on workstations running

- Mac OS X 10.4.6 (Intel)
- Windows Vista
- Debian stable (x86 and AMD64), testing (x86 and AMD64), and unstable (x86)

Gale has also been tested on clusters running RedHat Enterprise Linux 3 (EM64T).

3.3.2 Dependencies

In order to build Gale, you must have the headers and development libraries for

- MPI
- PETSc 3.0 (not 3.1!)
- libxml2

You must also have python 2.2.1 or greater installed. If you do not already have MPI, then in many cases PETSc can install a version for you. Installing PETSc also requires a Blas/Lapack implementation, which, again, PETSc can install for you.

3.3.3 Downloading the Code

You can get the source for the latest release from the Gale website (`geodynamics.org/cig/software/packages/long/gale/`). In that tarball is the file INSTALL. For some platforms, there are platform-specific instructions. Generally, the hardest part is not installing Gale itself, but PETSc.

3.3.3.1 Source Code Repository (Experts Only)

Advanced users and software developers may be interested in downloading the latest Gale source code directly from the CIG source code repository, instead of using the prepared source package. To check whether you have a Mercurial client installed on your machine, type:

```
hg
```

You should get a help message that starts with:

```
Mercurial Distributed SCM
...
```

Otherwise, you will need to download and install a Mercurial client, available at the Mercurial Website (`mercurial.selenic.com`). Then the code can be checked out with the following commands:

```
hg clone http://geodynamics.org/hg/long/3D/gale gale
hg clone http://geodynamics.org/hg/long/3D/gale/PICellerator gale/PICellerator
hg clone http://geodynamics.org/hg/long/3D/gale/StGermain gale/StGermain
hg clone http://geodynamics.org/hg/long/3D/gale/StgDomain gale/StgDomain
hg clone http://geodynamics.org/hg/long/3D/gale/StgFEM gale/StgFEM
hg clone http://geodynamics.org/hg/long/3D/gale/Underworld gale/Underworld
hg clone http://geodynamics.org/hg/long/3D/gale/config gale/config
hg clone http://geodynamics.org/hg/long/3D/gale/gLucifer gale/gLucifer
```

You can then update your checkout with the commands

```
cd gale
hg pull
hg up
cd PICellerator
hg pull
hg up
cd ../StGermain
hg pull
hg up
cd ../StgDomain
hg pull
hg up
cd ../StgFEM
hg pull
hg up
cd ../Underworld
hg pull
hg up
cd ../config
hg pull
hg up
cd ../gLucifer
hg pull
hg up
```

3.4 Support

The primary point of support for Gale is the CIG Long-Term Crustal Dynamics Mailing List (`cig-long@geodynamics.org`). Feel free to send questions, comments, feature requests, and bugs to the list. The mailing list is archived at

`(geodynamics.org/pipermail/cig-long/)`

You may also use the bug tracker

`(geodynamics.org/roundup)`

to submit bugs and requests for new features.

Chapter 4

Running Gale

4.1 Basic Usage

If you downloaded binaries for your platform, you can run the Gale executable directly. For example,

```
./Gale-1_6_1 input/cookbook/yielding.xml
```

will output

```
TimeStep = 0, Time = 0
TimeStep = 1, Time = 0.010764
TimeStep = 2, Time = 0.0214745
TimeStep = 3, Time = 0.0321333
TimeStep = 4, Time = 0.0427393
TimeStep = 5, Time = 0.0532923
TimeStep = 6, Time = 0.0637925
TimeStep = 7, Time = 0.0742399
TimeStep = 8, Time = 0.0846353
TimeStep = 9, Time = 0.0949793
```

It will also create a great deal of output in the directory `output`/.

If you do not specify an input file, you will get no output. If Gale cannot find the file, you will get an error:

```
Error: File input/cookbook/foo.xml doesn't exist, not readable, or not valid.
```

Due to quirks in some implementations of MPI, you may have to specify the complete path to the input file (e.g., `./Gale-1_6_1 /home/juser/gale/input/cookbook/yielding.xml`).

In general, Gale does not have many defaults, so almost everything must be specified in the input file. For examples of how to create your own input files, see Chapter 5. For a complete description of the input file format, see Appendix A.

If you compile Gale yourself, you can run it from where you installed it. If running in parallel on your own machine, prepend `mpirun` or `mpiexec` (depending on your local implementation of MPI). For example, if your computer has two cores, then

```
mpirun -np 2 bin/Gale /home/juser/gale/input/cookbook/yielding.xml
```

will use both cores.

4.2 Advanced Usage

4.2.1 Drucker-Prager Rheology

The Drucker-Prager rheology models a material that is rigid until the shear stress reaches a breaking, or yield, stress. Once the material yields, Gale reduces the viscosity of the material such that, given the strains applied to the material, the induced stress will now equal the yield stress. Unfortunately, there are two problems with this.

1. This is a numerical process, so the viscosity may be set too low. If the viscosity is too low, then the material will slip too easily, and there may be problems with convergence.

2. There is no length scale inherent in this method. So as you increase resolution, you will get finer and finer faults. This would not be too much of a problem if you just got the same faults, but more finely resolved. But what happens is that you tend to get more and more faults everywhere. The algorithm never converges to a single answer, and so it is difficult to say whether any results you get are reasonable. Moreover, if the size of your faults is always only a few points, you may get a systematic error in the fault angles [18].

Gale has two ways of solving this problem. One is to just set the minimum viscosity. This robustly solves the first problem. It also, in a sense, solves the second problem. Consider a model problem where two blocks are sliding against each other as in Figure 4.1. If the yielding stress only depends on cohesion, then a length scale naturally comes out

$$L_{\eta_{min}} = \frac{\eta_{min} v}{C},$$

where η_{min} is the minimum viscosity, v is the velocity of the sliding blocks, and C is the cohesion.

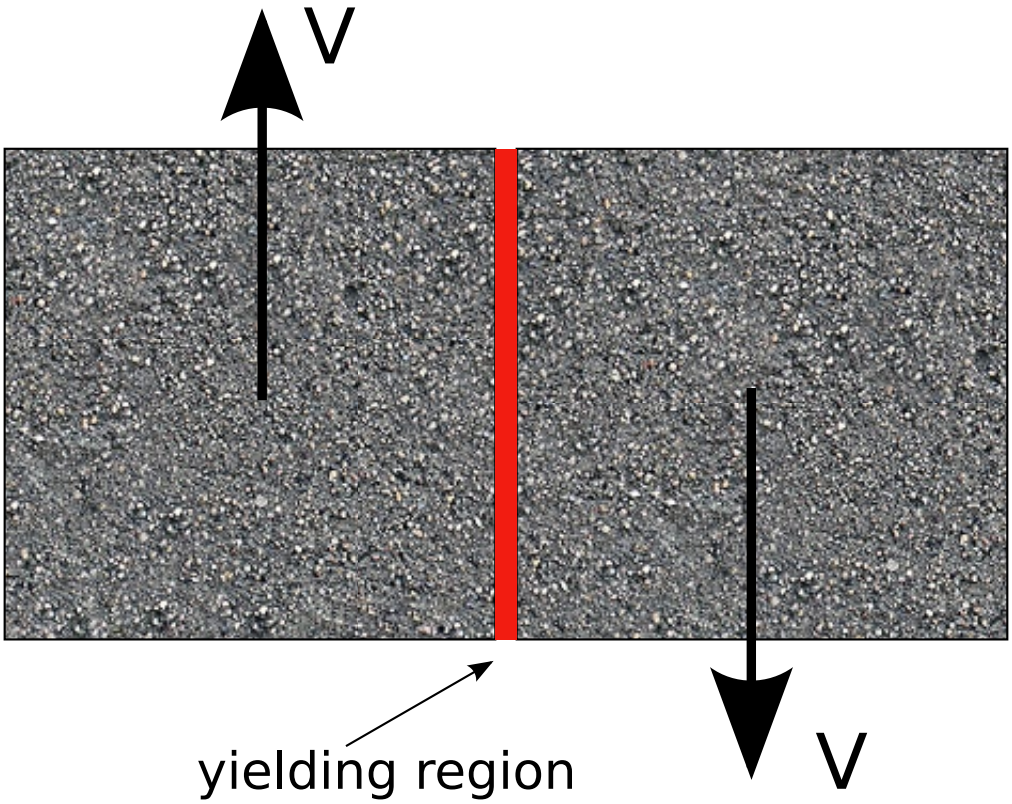


Figure 4.1: Two blocks sliding past each other with a yielding region between them.

For a general Drucker-Prager rheology, though, the yield stress depends on the pressure as well. In that case, as you look at material deeper and deeper in the earth, where the pressure, and hence yield stress, is higher, then the length scale will get shorter and shorter. If you set η_{min} such that, at the surface, you get a reasonable length scale for your resolution, then the length scale will be much smaller and unresolved in the mantle.

So the other solution Gale provides is to set a maximum strain rate. It does this by looking at what the strain rate is, and making sure that the viscosity is not set so low such that the strain rate will exceed the maximum strain rate. This provides a length scale even more simply

$$L_{\dot{\epsilon}_{max}} = \frac{v}{\dot{\epsilon}_{max}}.$$

In practice, both of these quantities may need to be set. A minimum viscosity may assist in taming irregularities arising from activities on the surface, such as landslides. A maximum strain rate, in the mean time, will assist in ensuring that the code is convergent. Both of these parameters are used for the Drucker-Prager benchmark (Section C.5), and the Geomod benchmarks (Sections C.6 and C.7).

4.2.2 Direct Solvers

If you have a problem with strong viscosity gradients, the default solver (GMRES) may converge very slowly. Strong viscosity gradients occur when you start with materials with different viscosities (e.g., Appendix C.2 and C.1), or when materials yield (e.g., Appendix C.6).

One solution is to use a direct solver instead of GMRES. PETSc has a facility where you can use command-line arguments to change the solver. For example, on a single machine, to use a direct LU solve, you only need to append arguments to the command line

```
./Gale-1_6_1  input/cookbook/yielding.xml -pc_type lu -ksp_type preonly
```

In parallel, the analogous approach would be to use Mumps, a parallel direct solver. You first need to make sure that your version of PETSc was installed with Mumps. If you built PETSc yourself, you need to add the option "`--download-mumps=yes`" when configuring.

Once that is done, enabling it is again just appending a few arguments to the command line

```
./Gale-1_6_1  input/cookbook/yielding.xml -pc_factor_mat_solver_package mumps \
-ksp_type preonly -pc_type lu -mat_mumps_icntl_14 100
```

Note that this is different from previous versions of Gale. Petsc changed the syntax for calling Mumps solvers. Also, Mumps changed the default amount of memory it allocates. This is not an issue for small simulations, but larger simulations can easily run out of memory. The option "`-mat_mumps_icntl_14 100`" tells Mumps to allocate more memory.

4.2.3 Multigrid

For very large problems and 3D problems in general, the direct solver may not work well. In that case, you can try multigrid. To do that, add the plugin

```
<struct>
  <param name="Type">StgFEM_Multigrid</param>
  <param name="Context">context</param>
</struct>
```

to the beginning of the file, and add the components

```
<struct name="mgSolver">
  <param name="Type"> PETScMGSolver </param>
  <param name="levels"> mgLevels </param>
  <param name="opGenerator"> mgOpGenerator </param>
```

```
</struct>
<struct name="mgOpGenerator">
  <param name="Type"> SROpGenerator </param>
  <param name="fineVariable"> VelocityField </param>
</struct>
```

You may have to modify `mgLevels` for your problem.

4.2.4 Command-Line Parameters

You can also change the default values of the input file without modifying that file by appending arguments. For example, to change only the number of time steps from the default value of 10 to 20, use the following command

```
./Gale-1_6_1  input/cookbook/yielding.xml --maxTimeSteps=20
```

You can append any number of modified parameters in one unbroken line (here shown wrapped around)

```
./Gale-1_6_1  input/cookbook/yielding.xml --maxTimeSteps=20 --dim=3 --elementResI=64
--elementResJ=64 --elementResK=64 --particlesPerCell=60 --dumpEvery=10
```

4.2.5 Checkpointing

Gale can save the state of the simulation so that it can be restarted from that point. To save the state for every time step, add the line

```
<param name="checkpointEvery">1</param>
```

to the variables at the end of the input file or add

```
--checkpointEvery=1
```

to the command line. To restart from step 5, add

```
--restartTimestep=5
```

to the command line.

Note that the example input files do not, by default, save and restore the temperature. To enable that, add the line

```
  <param>TemperatureField</param>
```

after the lines

```
<list name="FieldVariablesToCheckpoint">
  <param>VelocityField</param>
  <param>PressureField</param>
```

4.2.6 Debugging Input Files

It can often happen that you set up an input file incorrectly and try to run it, but Gale never gets far enough to tell you what you did wrong. The first thing you should do is to turn on verbose output as in Section A.14. That way, you can look at the residual for the linear and non-linear solvers. If the residuals go up and down, even after a number of iterations, then the solver will probably not converge. On the other hand, if the residuals go steadily down, you can determine whether you should try different input parameters or just wait longer.

Even with that, you may not know what to fix. For example, you may have unwittingly set the minimum viscosity for a yielding material to be too low. If the non-linear solver never converges, then you will not be able to see that you specified too low of a minimum viscosity. One way to get around this is to temporarily set the tolerance for the non-linear solver (`nonLinearTolerance`) to be very large. Another way is to set the maximum number of non-linear iterations (`nonLinearMaxIterations`) to be relatively small. Then Gale will produce output that, while it may not be a good solution to the Stokes equations, nevertheless gives you clues on how to fix the input file.

4.3 Output and Visualization

The sample input files will produce a directory in which you will find a number of files. The formats of these files are described fully in Appendix B.

The default setting is to create VTK files every 10 time steps. To change the frequency for creating VTK files, you need to change the parameter `dumpEvery`. For example, if you modify the line with `dumpEvery` to

```
<param name="dumpEvery">25</param>
```

then the VTK files will only be created every 25 time steps. Similarly, to change the frequency of checkpoints, you need to change the parameter `checkpointEvery`. You can change the value to any number you choose.

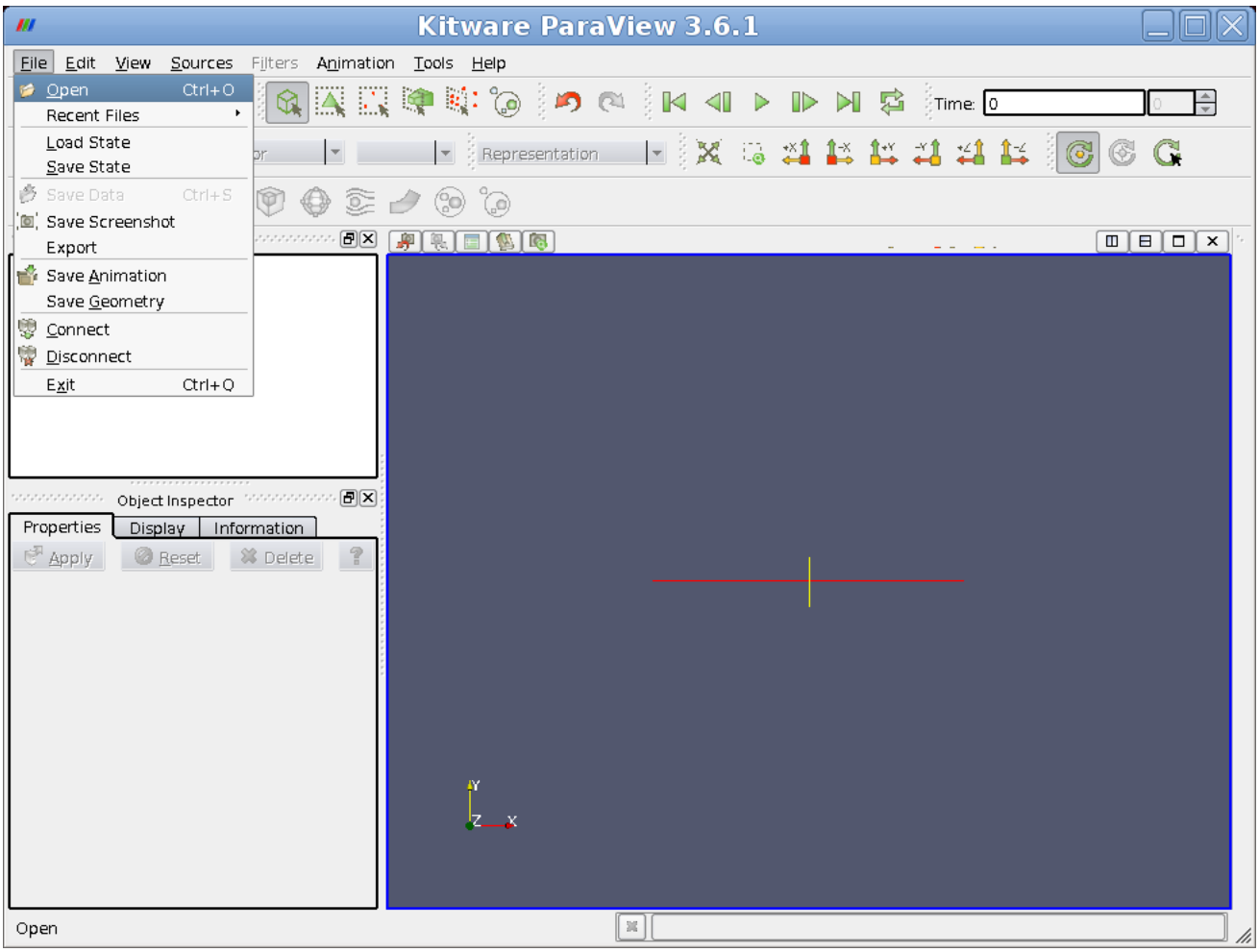
There are a number of different options for visualizing the data. The easiest way is to use the VTK files. These files are in a standard, self-describing file format that can be easily visualized with several different visualization programs, e.g., ParaView (`paraview.org`), MayaVI (`mayavi.sf.net`), and Visit (`www.llnl.gov/visit`). ParaView is recommended as it is easy to get working, easy to use, and scales to large data sets.

Another option is to use the scripts in the `tools/` directory to convert the VTK files into CSV format. Then the data can be easily manipulated with standard tools like Python or Matlab.

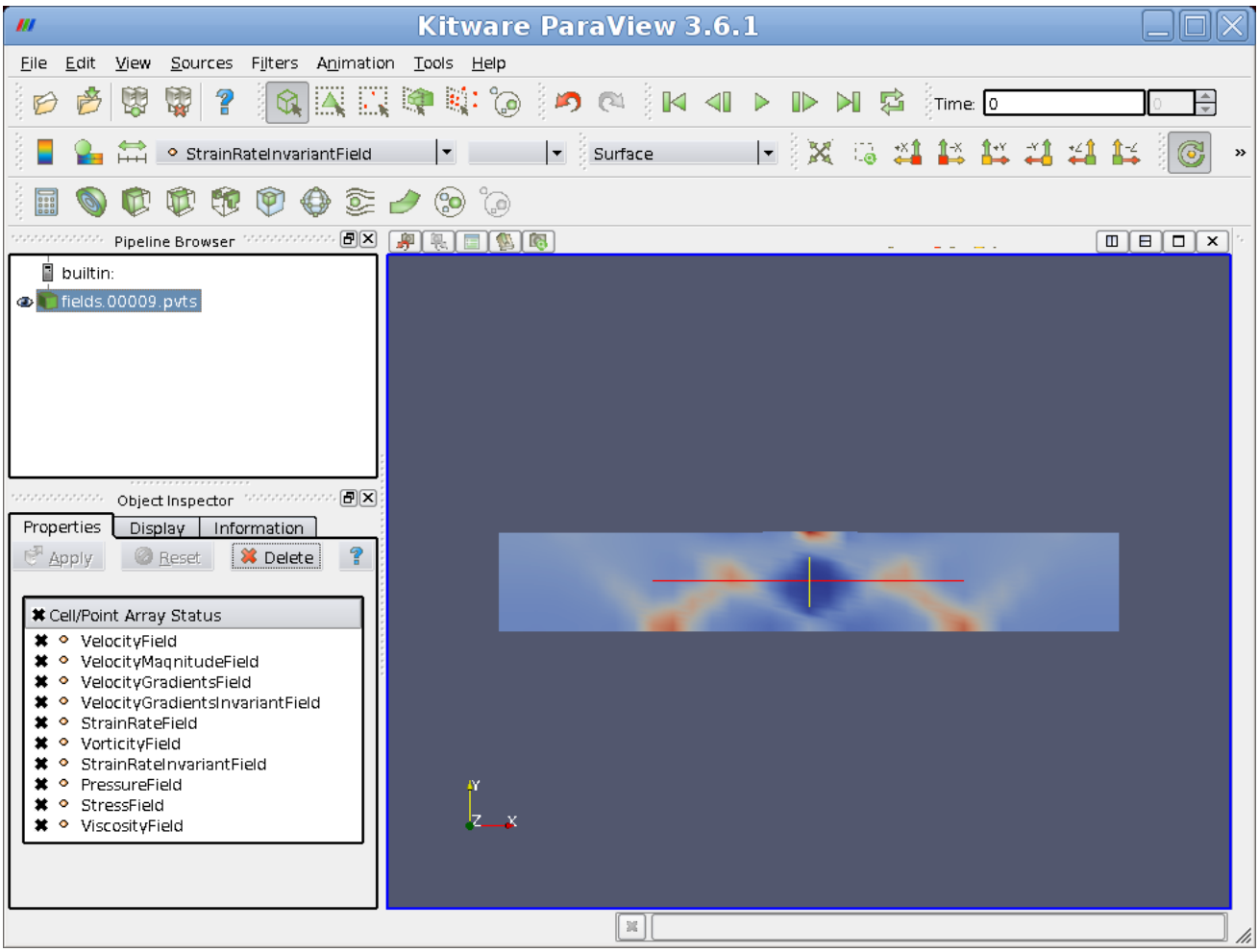
4.3.1 Basic Visualization with ParaView

These instructions are for Paraview version 3.6.1. To visualize step 10 of `input/cookbook/yielding.xml`,

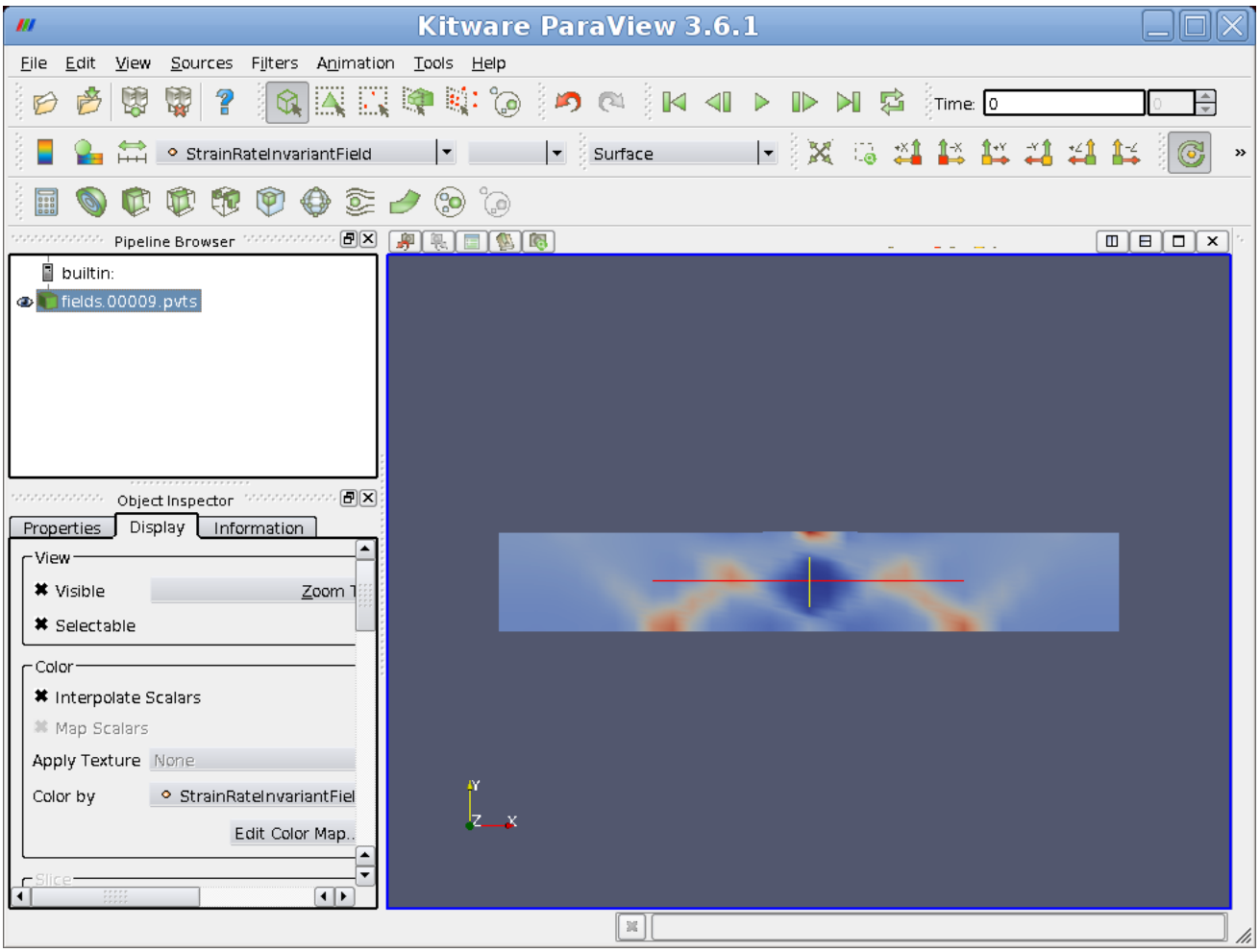
1. Start Paraview and open a new data file: File ▷ Open



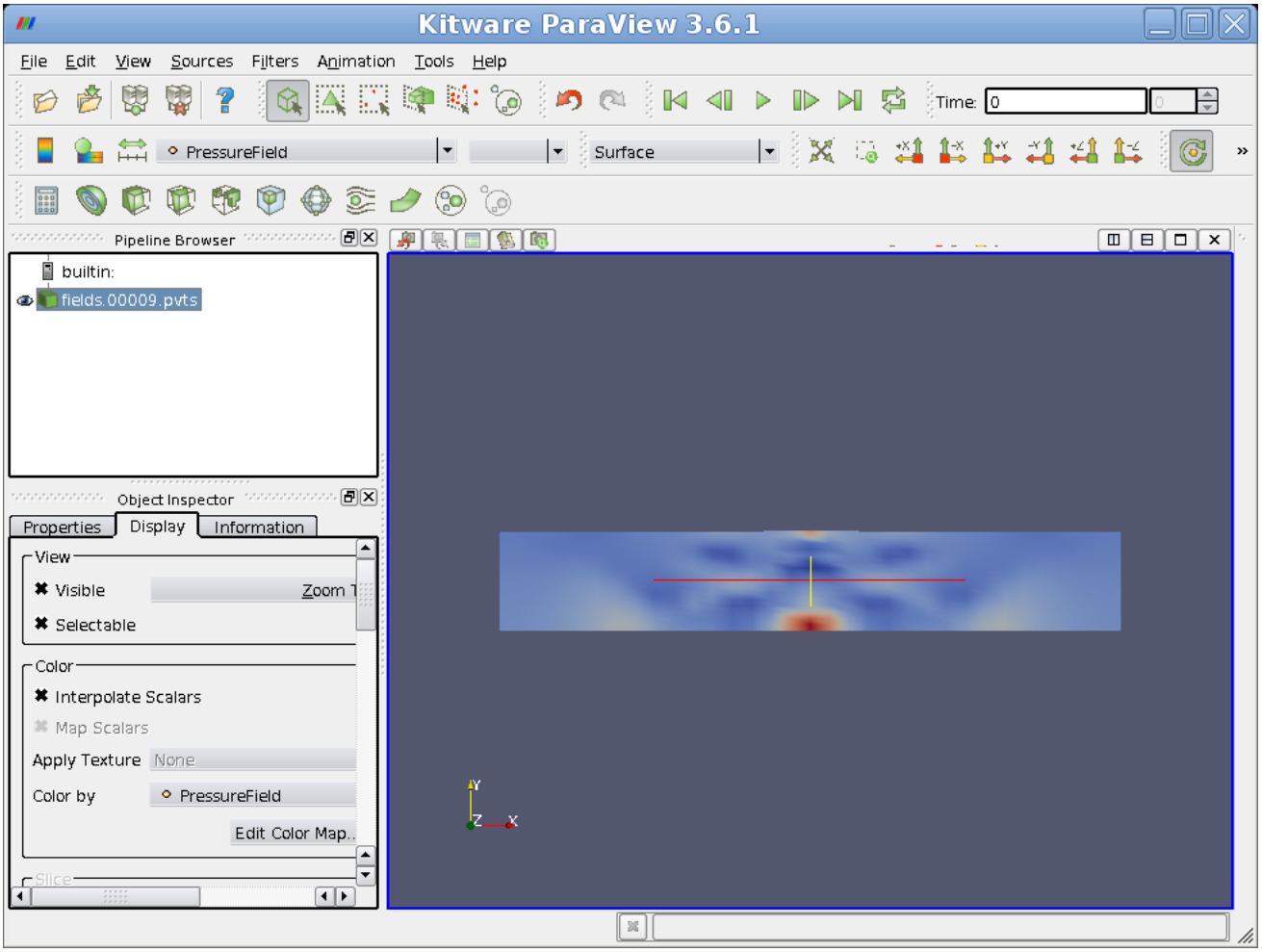
2. A file dialog screen will appear. Navigate to the output directory and select `fields.00009.pvts`. Then click the green "Apply" button. Paraview will display a pseudocolor plot of the strain rate invariant.



3. You can instead plot the pressure by first clicking on the “Display” tab.

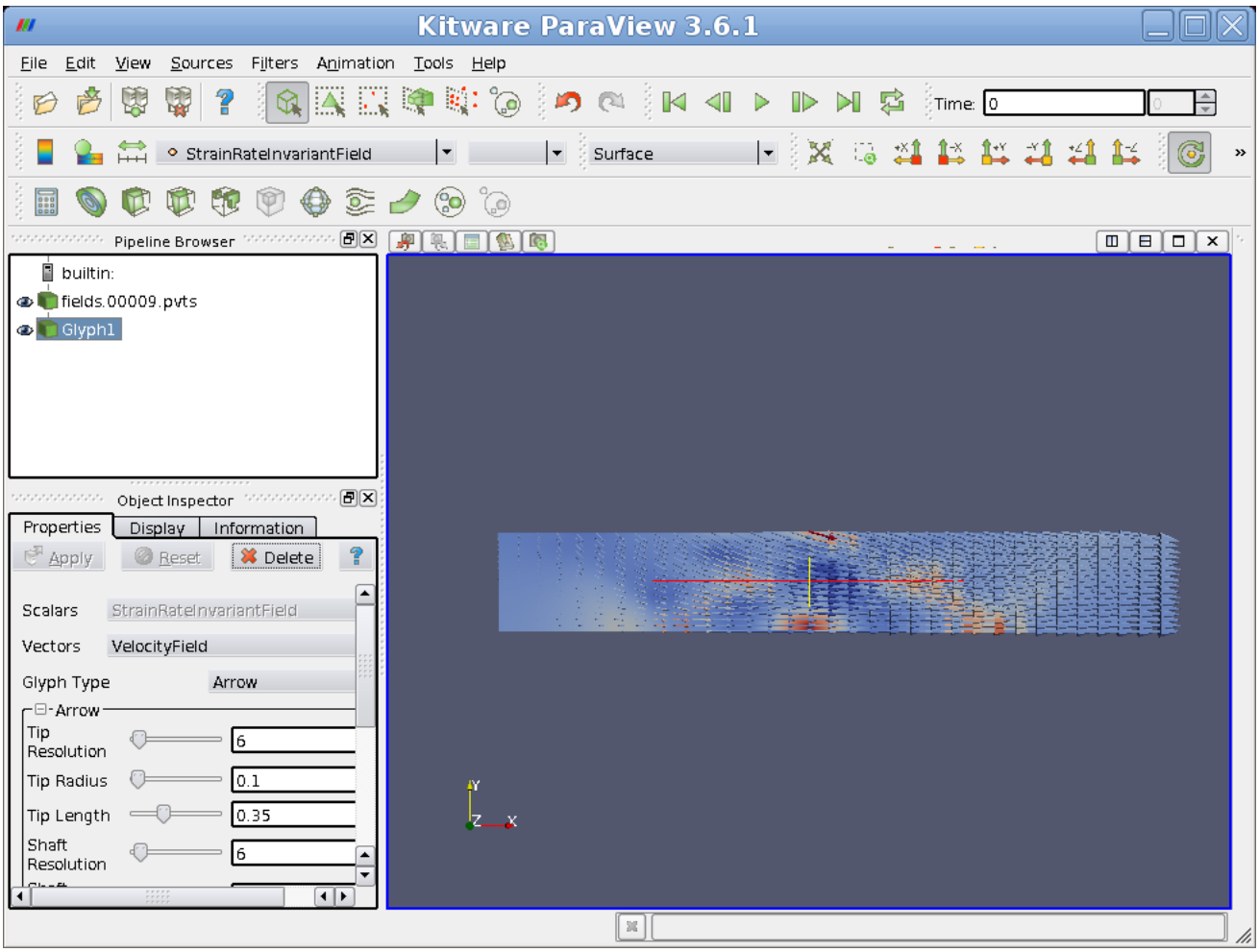


4. Click on the drop-down list to the right of "Color by" and select "PressureField."

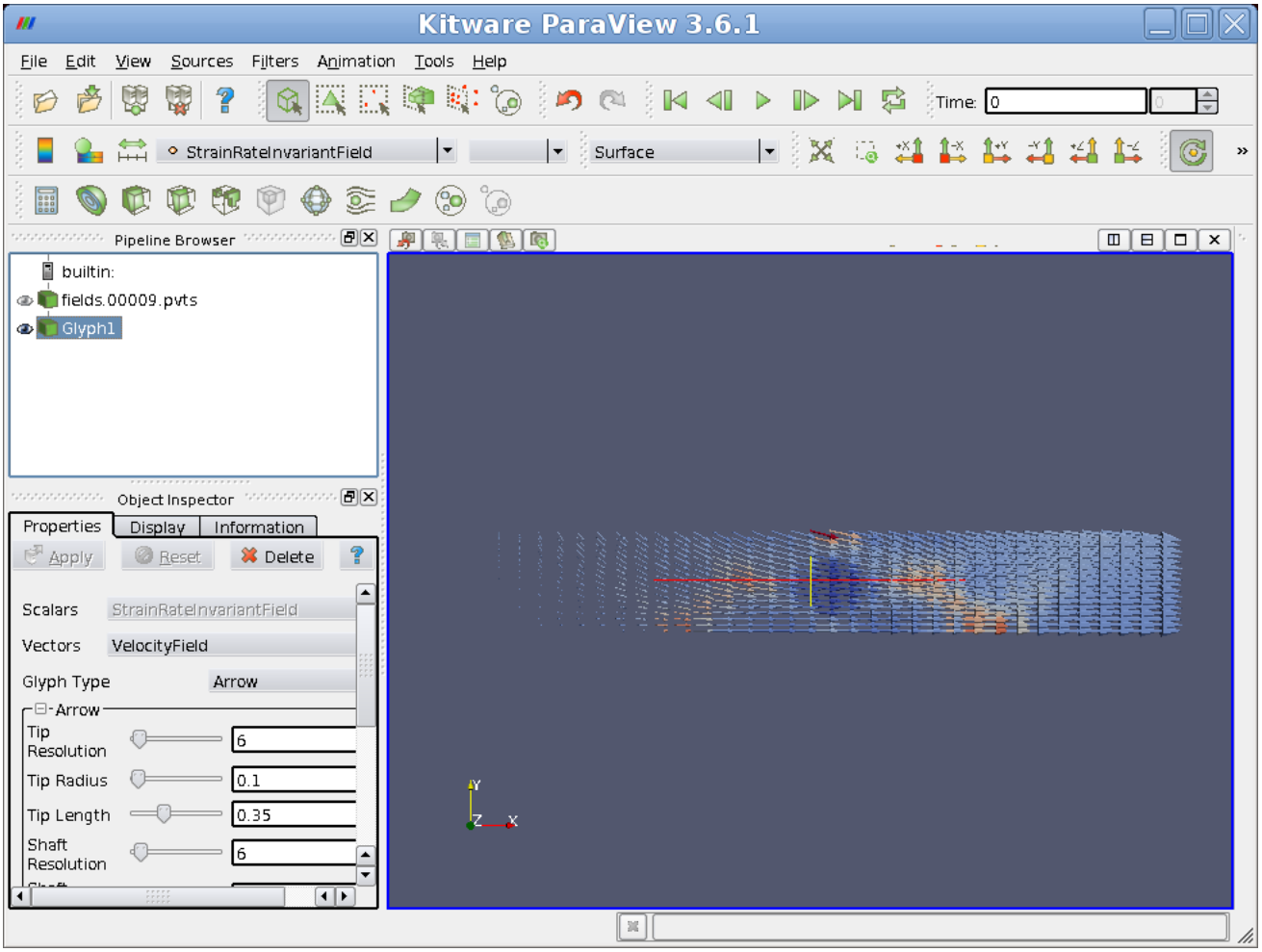


5. Now you can plot the velocity as arrows on top of the pressure: Click on the “Glyph” symbol

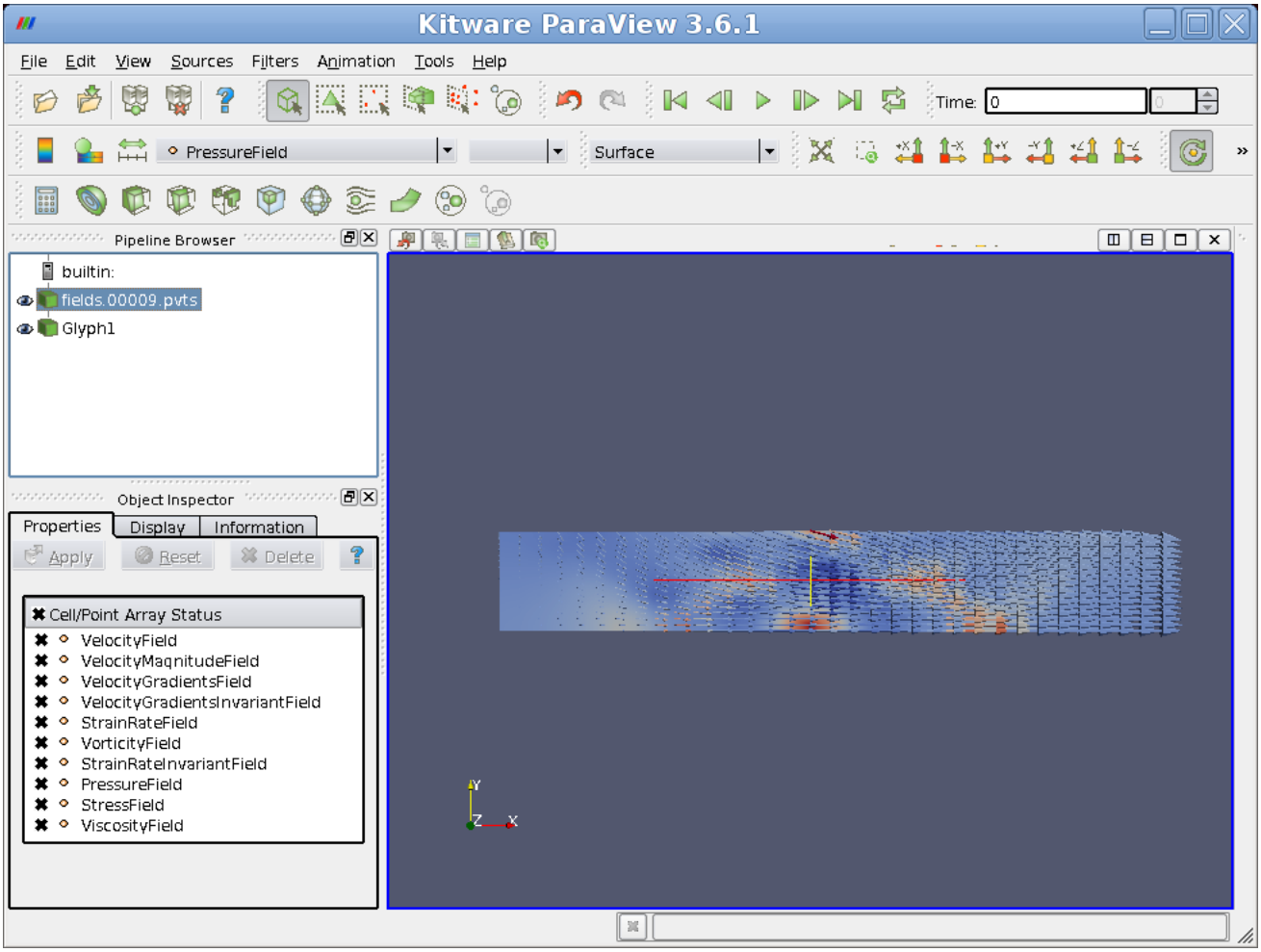
and then press the green “Apply” button.



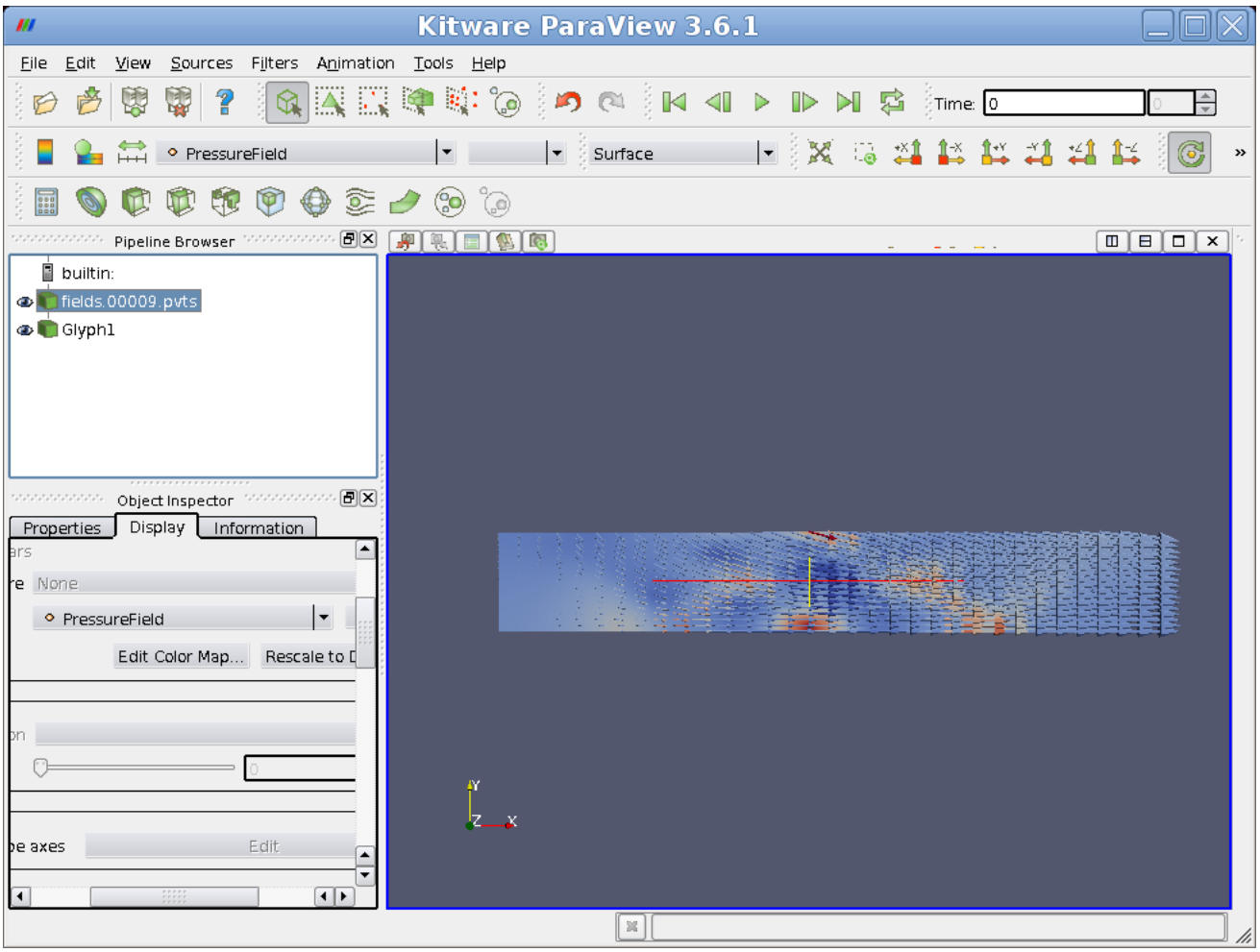
6. You can temporarily remove the pressure to see the velocities better by clicking on the "eye" graphic  next to `fields.00009.pvts`.



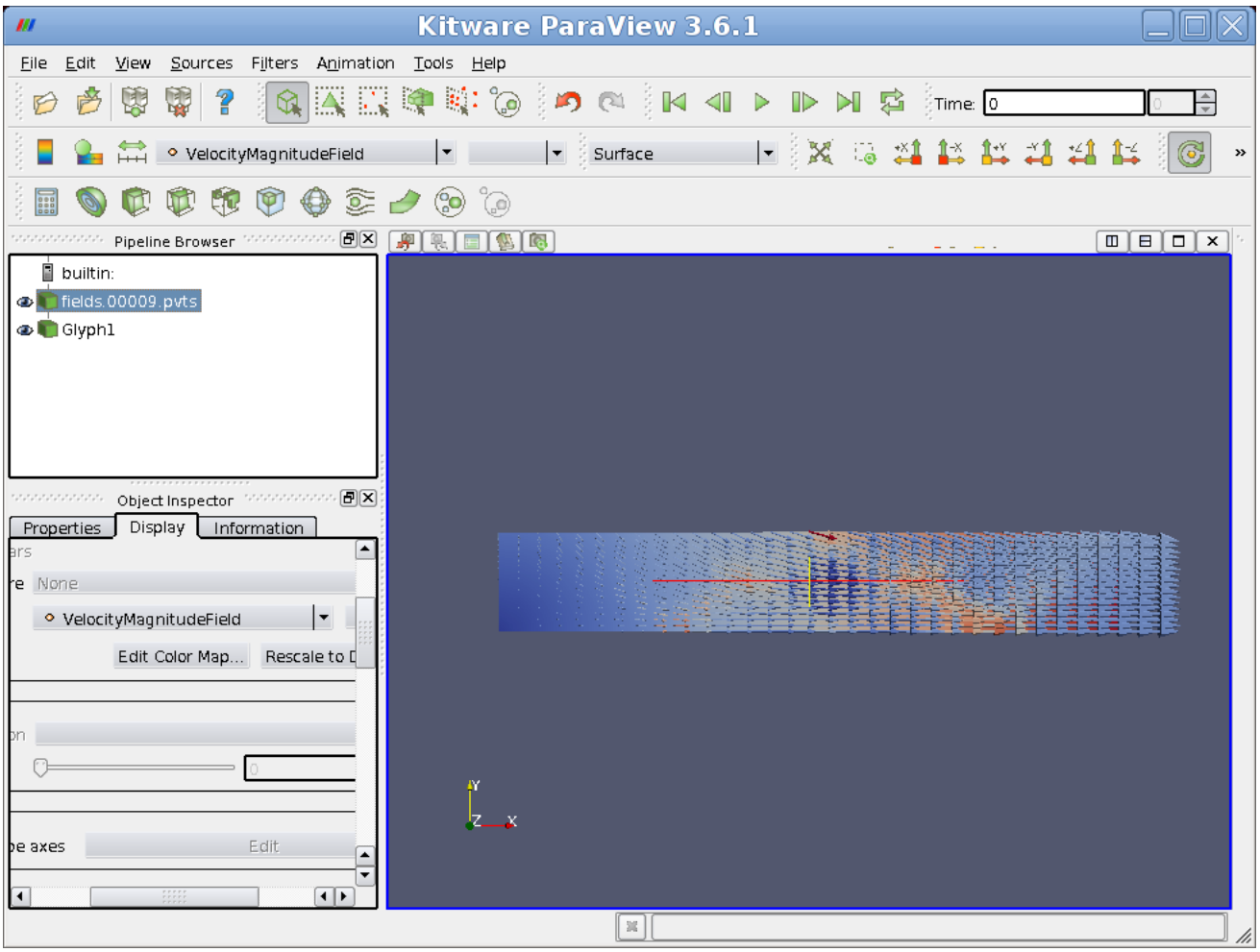
7. You can also visualize the magnitude of the velocity: First click the "eye" again, then click on the text `fields.00009.pvts`.



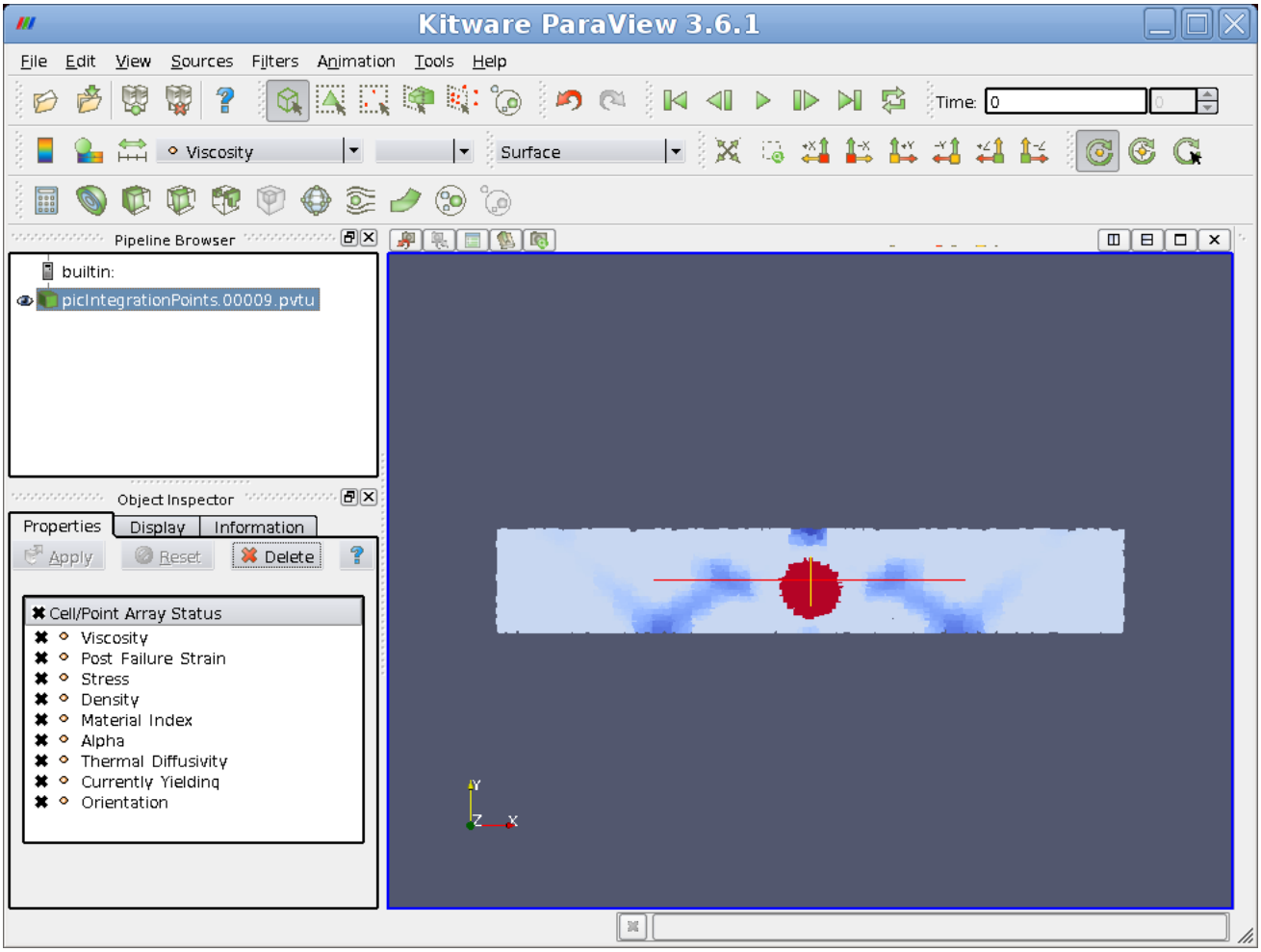
8. Next click on the “Display” tab.



9. Finish by clicking on the drop-down list to the right of "Color by" and selecting "VelocityMagnitude-Field."

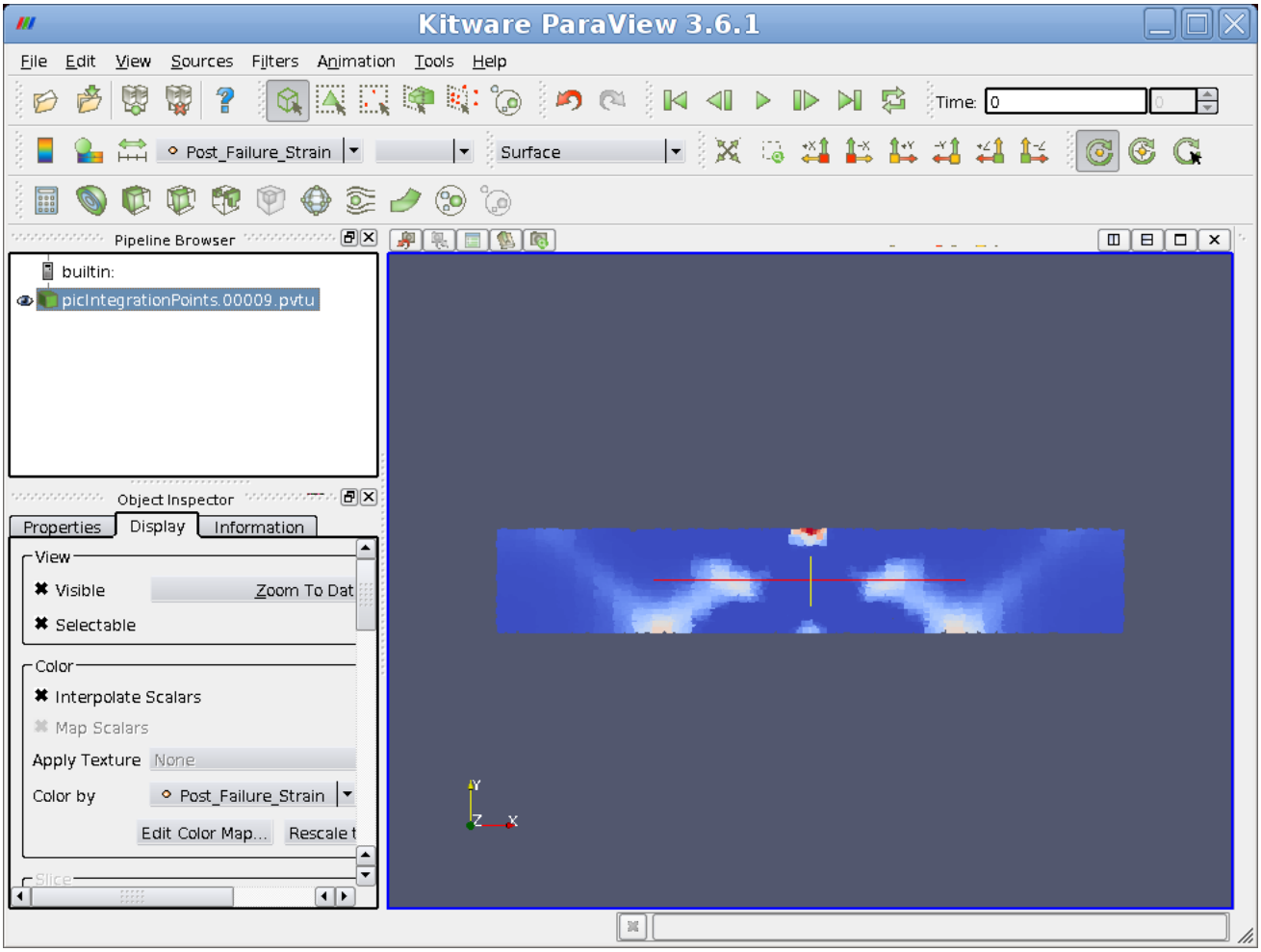


10. Now you can look at the particles. Starting over, open `picIntegrationPoints.00009.pvtu` and click on the green "Apply" button.

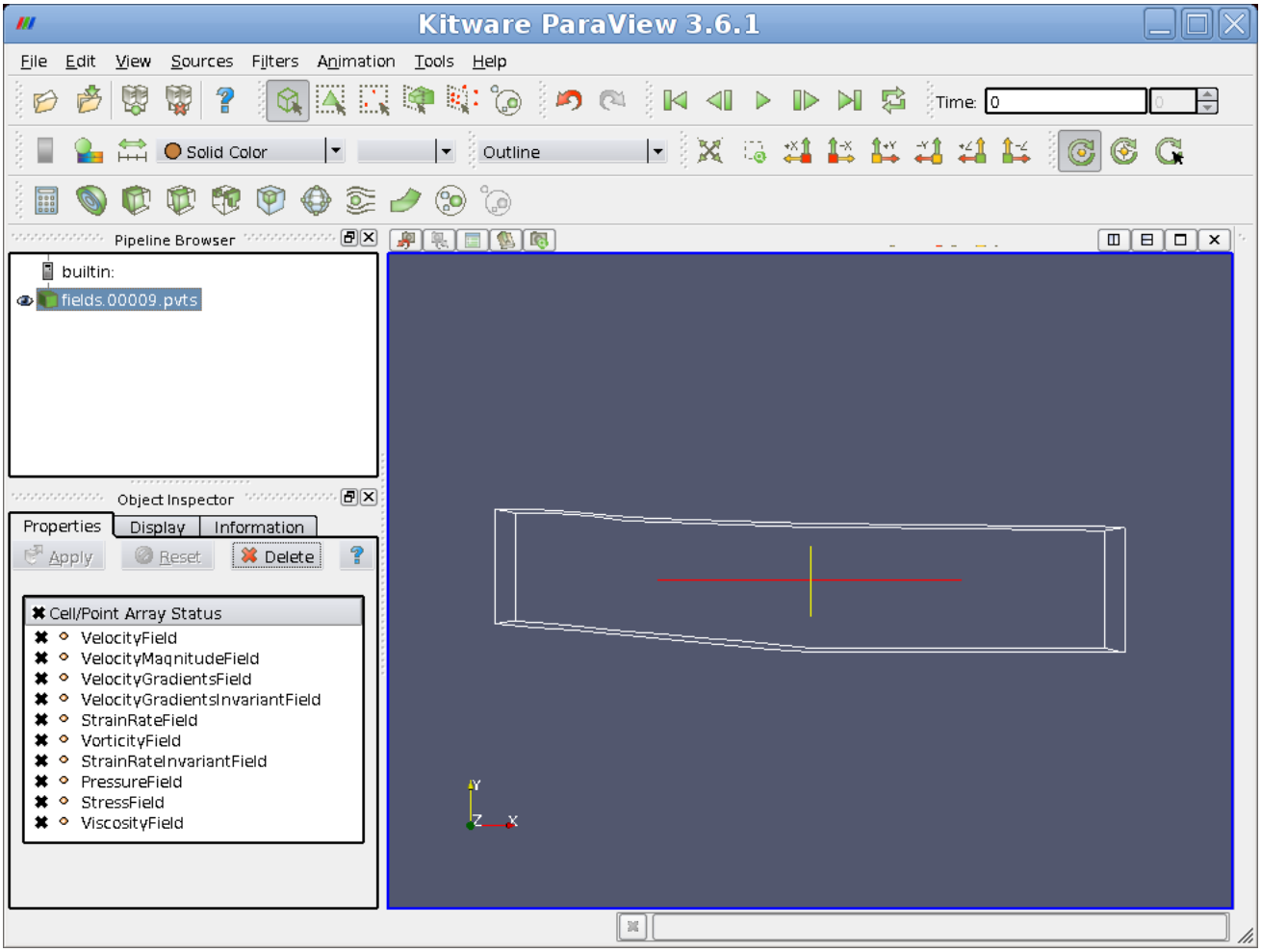


This displays the viscosity of the particles. The red points represent the high viscosity ball, while the blue points represent material that has yielded.

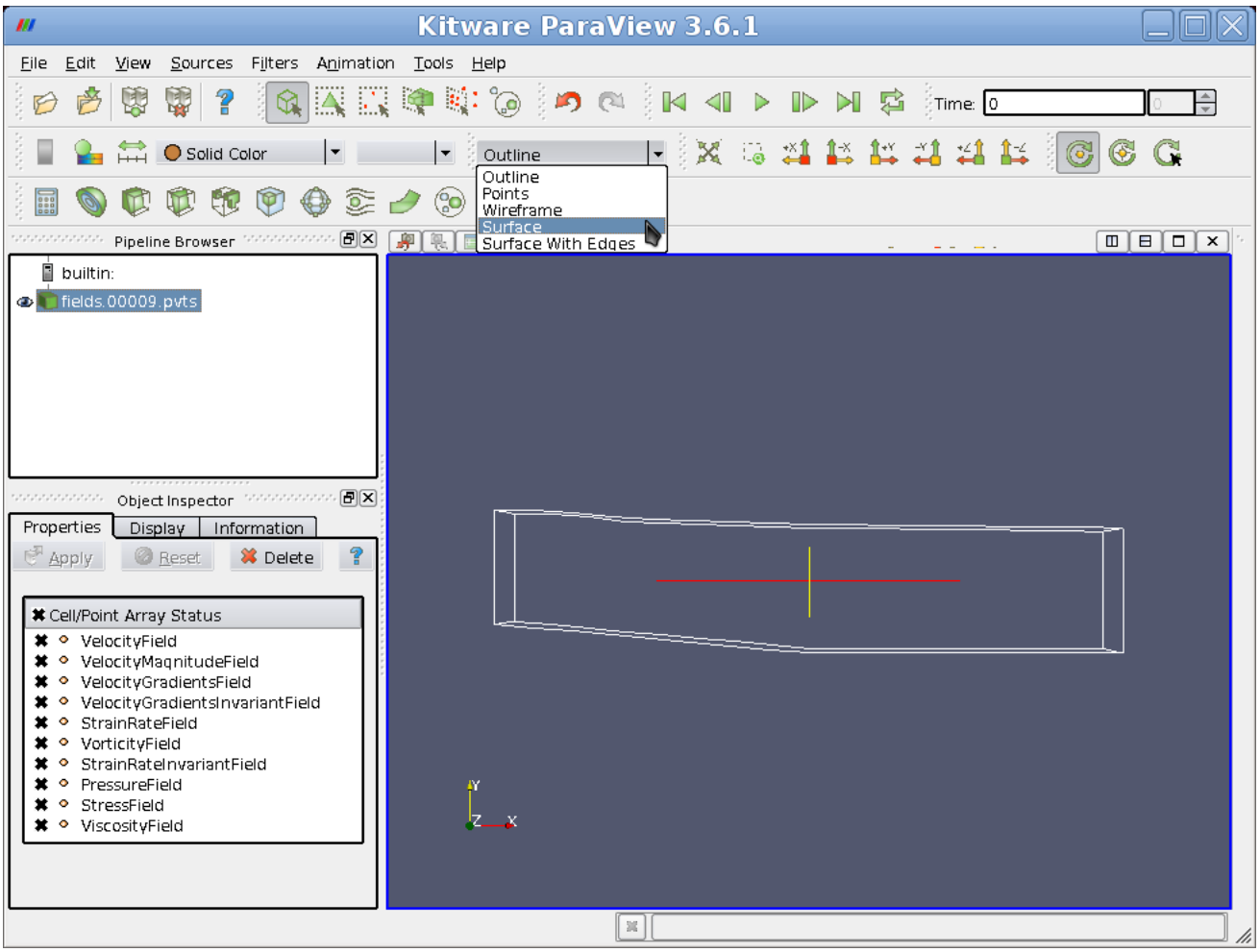
11. To see explicitly which material has failed plastically, click on the "Display" tab and change the "Color by" box to "Post_Failure_Strain."



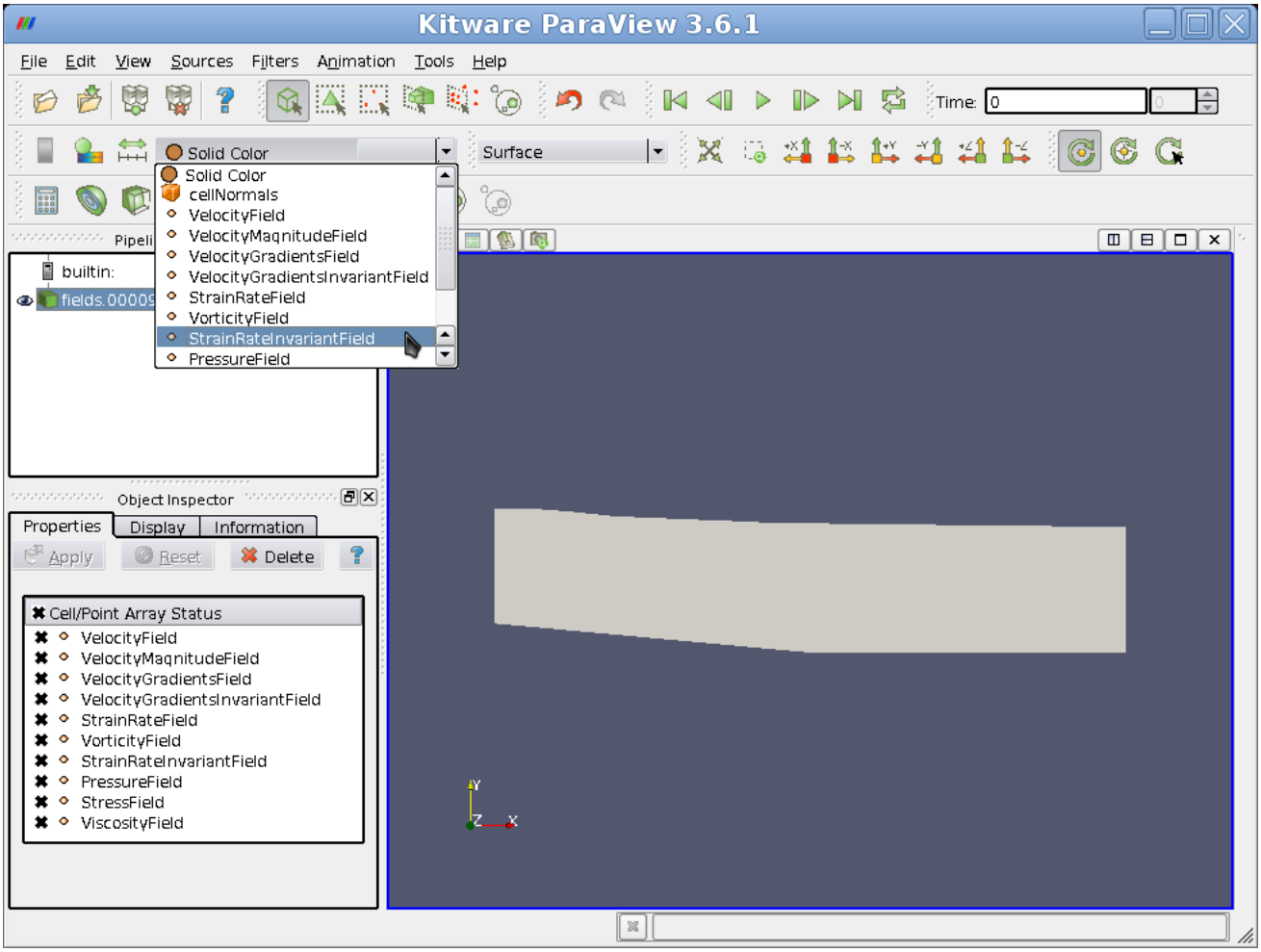
12. Now we will visualize a 3D simulation. Run Gale with the input file `input/cookbook/viscous_extension3D.xml` and open `fields.00009.ptvs`. Paraview will display an outline of the 3D volume.



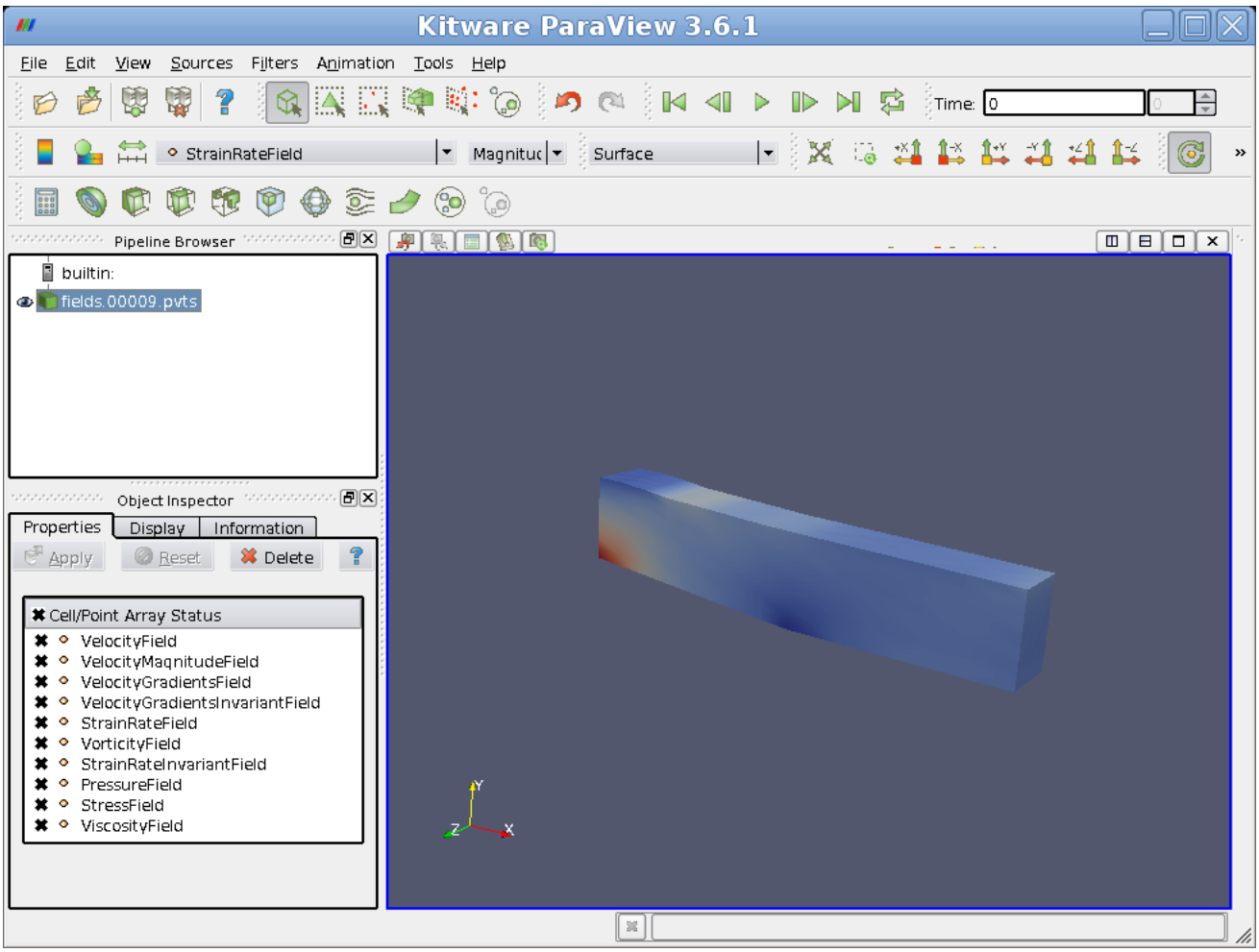
13. Click on the button near the top center labeled ”Outline” and change it to ”Surface”.



14. Next click on the button near the top left labeled ”Solid Color” and change it to ”StrainRateInvariant-Field”



15. You can now rotate it by clicking and holding with button 1 on your mouse. You can zoom in and out by clicking and holding button 2, and translate by clicking and holding button 3.



4.3.2 Visualizing Movies with Paraview

To create a movie, with Paraview, you need to create a special `pvd` file that stitches all of the individual frames together. You can do this with the utility `tools/generate_pvd`. You generally invoke it as

```
generate_pvd NAME TYPE START END STEP
```

where `NAME` is usually either "fields" or "picIntegrationPoints", depending on whether you want to animate the fields or the particles. If you create your own swarm of particles, then they will have their own VTK files, and you would use that name with `generate_pvd`.

`TYPE` is either '`u`' for unstructured (particles), or '`s`' for unstructured (fields). START is the time of the first time step, END is the last step, and STEP is the step size between successive frames. So the command

```
generate_pvd picIntegrationPoints u 0 100 10
```

will generate `picIntegrationPoints.pvd`. That file starts at t=0 and includes every 10'th step up to and including 100. So to animate the post yielding strain in `input/viscous_extension3D.xml`, you would run, from the top directory,

```
generate_pvd fields s 0 9
```

Loading up `fields.pvd` in Paraview gives you access to the movie buttons

4.3.3 Generating CSV files

The script `tools/vtk2csv.py` converts VTK files to a simple comma separated format. To run it, you must have Paraview installed. Included with the Paraview installation is a utility `pvpython`. This is a version of Python set up to automatically have access to all of the Paraview libraries. It takes any number of VTK files on input and outputs corresponding CSV files. So to convert the file `fields.00009.pvts` and all associated `vts` files, you run it as

```
pvpython vtk2csk.py fields.00009.pvts
```

and it will output `fields.00009.csv`.

4.4 Gauging Accuracy

Gale makes a number of approximations. Before trusting any results you get from Gale, you must vary a number of parameters to insure that the results are not an artifact of Gale's approximations.

The most obvious parameter to vary is the mesh resolution. The grid is where the Stokes equations are solved, and defines the resolution of everything in the "fields" output files (e.g., velocity, pressure, strain rate, etc.). The resolution of the grid is determined by `elementResI`, `elementResJ`, and `elementResK`.

But sometimes the mesh resolution is not the principal source of error. For example, for the 2D Divergence benchmark (Section C.4), the principal source of error is the tolerance in the linear solver. This is because the solution can be represented exactly on even a tiny grid, so the determining factor is just how well the equations are solved on the mesh. To vary the tolerance for the linear solve, change the parameter `linearTolerance`.

Similarly, the tolerance for the non-linear solve may determine the overall error. You can set that tolerance with the parameter `nonLinearTolerance`. You still have to be careful, though, because the solver can still settle on a wrong solution as in the Drucker-Prager benchmark (Section C.5). The initial solution always gives a yielding angle of 45°. Only after a number of iterations does the solution finally move to the correct angle. To fix this, you can set `nonLinearMinIterations`.

The number of particles can also determine the error, as in the 3D Divergence benchmark (Section C.4). There is a more or less constant number of particles per mesh element. If you have a smooth velocity field, but a complex particle properties field, you may need more particles for each element. To set the particle resolution, change the parameter `particlesPerCell`.

When using a yielding rheology, you should vary `minimumViscosity` and `maxStrainRate` (see Section 4.2.1).

For some problems where you are comparing against a solution over an infinite domain (e.g. Sections C.1, C.2, C.3), then you may need to vary the size of the box (`minX`, `minY`, `minZ`, `maxX`, `maxY`, `maxZ`). Finally, you may need to vary the scaling factor for time steps (`dtFactor`) (see Section A.1.4).

How much to vary the various parameters depends upon each parameter. For some parameters, such as the resolution, changing it by a factor of two is often good enough to tell whether your error depends on resolution. For others, such as the tolerance for the solver, you may want to vary it by a factor of ten (e.g. Figure C.11).

Chapter 5

Cookbooks

5.1 Introduction

In this chapter, you will edit a template file (`input/cookbook/template.xml`) to create customized input files. You should be able to use the template file as a basis for most of your own input files. There are two things in the template file, however, that might need modification: the force of gravity, which by default is set to 1 (if you are using cgs, for example, the force of gravity must be changed to 980), and the normal velocity on all boundaries except the top, which are set to zero. Beyond that, you only need to specify where to place materials.

5.1.1 Adding Lines to the Template File

Unless otherwise specified, when you are instructed to add sections to the input file[1], that text should be added after the line

```
<struct name="components">
```

at the beginning of the file, and before the lines

```
</struct>
<list name="FieldVariablesToCheckpoint">
```

The template file is indented to make it easier to notice if you add a component in the middle of another component. This is solely for the benefit of humans. Gale does not pay attention to indentation when reading the files.

5.1.2 Adding Variables to the Template File

When you are instructed to add a variable, you must add it at the end of the file before the closing line

```
</StGermainData>
```

Finished versions of all of these examples are found in `input/cookbook/`.

5.2 Viscous Material

This example simply fills up the computational domain with a single viscous material. It is a valid input file, but will not run as nothing is moving. This file mainly serves as the basis for subsequent examples.

[1] To copy and paste from this PDF with Adobe Acrobat, right click to get the context menu and select "Allow Hand Tool to Select Text."

1. First, copy `template.xml` to `myviscous.xml` to edit as follows.

2. Add in a material. The simplest variety is a purely viscous material, so add a box covering the whole domain:

```
<struct name="boxShape">
  <param name="Type">Box</param>
  <param name="startX">minX</param>
  <param name="endX">maxX</param>
  <param name="startY">minY</param>
  <param name="endY">maxY</param>
  <param name="startZ">minZ</param>
  <param name="endZ">maxZ</param>
</struct>
```

   Note: Default parameters for the box (e.g., `minX`, `maxX`, `minY`, etc.) are already defined in `template.xml`.

3. Then set the material's viscosity

```
<struct name="backgroundViscosity">
  <param name="Type">MaterialViscosity</param>
  <param name="eta0">1.0</param>
</struct>
```

   Remember that Gale has no internal knowledge of units, so if you think of everything in cgs, then this implies a viscosity of $1 \frac{g}{cm^2\, s}$.

4. Finally, you create the material, using the components just created.

```
<struct name="viscous">
  <param name="Type">RheologyMaterial</param>
  <param name="Shape">boxShape</param>
  <param name="density">1.0</param>
  <list name="Rheology">
    <param>backgroundViscosity</param>
    <param>storeViscosity</param>
    <param>storeStress</param>
  </list>
</struct>
```

   The `storeViscosity` and `storeStress` parameters are standard components that enable you to get the viscosity and stress information on each particle.

5. Save this file, as it will be the basis for other examples that follow.

6. You can now run this example, and the output will go into the directory `output`. If you want to run in 3D, you only need to change the line after

```
<param name="outputPath">./output.template</param>
```

   from

```
<param name="dim">2</param>
```

   to

```
<param name="dim">3</param>
```

You can compare your result with the worked example in the file `input/cookbook/viscous.xml`.

5.3 Viscous Material in Simple Extension

The input file you created in Section 5.2 is valid, but nothing moves, so Gale will output errors if you try to run it. In this next example, you will make the material extend by having the right boundary move.

1. Copy `myviscous.xml` to `myviscous_extension.xml`.
2. Make the right boundary move by changing the line after this section

```
<param name="type">WallVC</param>
<param name="wall">right</param>
<list name="variables">
  <struct>
    <param name="name">vx</param>
    <param name="type">double</param>
```

from

```
<param name="value">0.0</param>
```

to

```
<param name="value">1.0</param>
```

Warning: There are several `WallVC` structs: `front`, `back`, `left`, `right`, `top` and `bottom`. Here we have only modified the `right` side.

A worked example is at `input/cookbook/viscous_extension.xml`. Figure 5.1 shows the strain rate invariant and velocity (see Section 4.3.1).

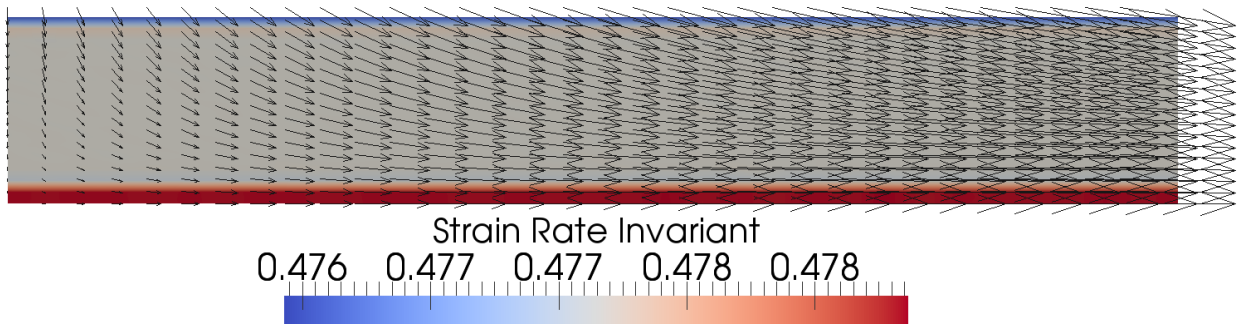


Figure 5.1: Strain rate invariant and velocity of viscous material in extension

The structure in the strain rate invariant arises from the artificial compressibility (see Section 2.2.8.1). The magnitude of this structure is small, so you can ignore it for now. Later, in Section 5.11, you will remove it by adding a `HydrostaticTerm`.

5.4 Viscous Material with Complex Boundaries

Another exercise is to make the bottom boundary move differently, and not just have the material slide along. In particular, this example will simulate a box like in Figure 5.2, where the bottom right side of the box moves, but the viscous material sticks to the bottom left.

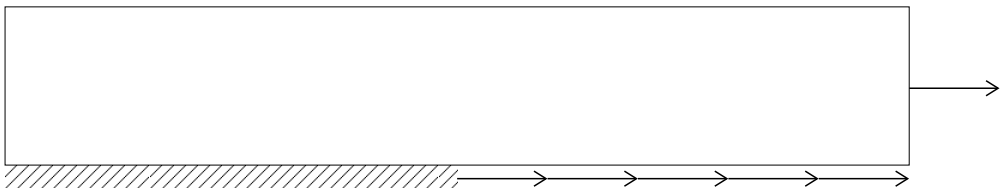


Figure 5.2: Split Boundary

1. First, copy `myviscous_extension.xml` to `myviscous_split.xml`
2. Modify the bottom boundary condition of WallVC to

```
<struct>
  <param name="type">WallVC</param>
  <param name="wall">bottom</param>
  <list name="variables">
    <struct>
      <param name="name">vy</param>
      <param name="type">double</param>
      <param name="value">0.0</param>
    </struct>
    <struct>
      <param name="name">vx</param>
      <param name="type">func</param>
      <param name="value">StepFunction</param>
    </struct>
  </list>
</struct>
```

   This makes the velocity of the bottom boundary a step function.

3. You must also specify the parameters of the step function by adding the variables

```
<param name="StepFunctionLowerOffset">1.0</param>
<param name="StepFunctionUpperOffset">1.0</param>
<param name="StepFunctionValue">1.0</param>
<param name="StepFunctionDim">0</param>
<param name="StepFunctionLessThan">False</param>
```

   to the end of the file (just before `</StGermainData>`).

   Warning: Do not add them in the list named "`variables`."

A worked example is in the file `input/cookbook/viscous_split.xml`. Figure 5.3 shows the strain rate invariant and velocity (see Section 4.3.1). The strain rate is concentrated around the step function in the bottom velocity boundary. Notice the development of a basin above the discontinuity. The ability to track the development of topography on the free surfaces is one of the strengths of Gale.

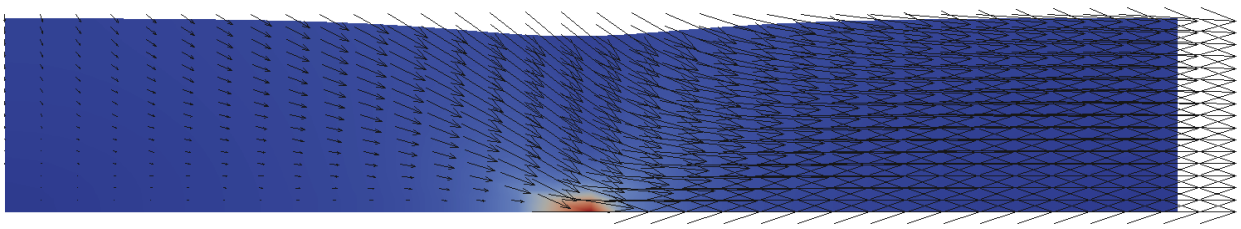


Figure 5.3: Strain rate invariant and velocity with complex boundaries

5.5 Viscous Material with Boundary Conditions Read From a File

You may want to specify custom boundary conditions that are not already implemented. For this, you can set boundary conditions using date from a file. For this example, we will replace the sharp step function with an smoother approximation. The data is in the file `input/cookbook/velocities`. To get Gale to use it:

1. Copy `myviscous_extension.xml` to `myviscous_file.xml`
2. Modify the bottom boundary condition of WallVC to

```
<struct>
  <param name="type">WallVC</param>
  <param name="wall">bottom</param>
  <list name="variables">
    <struct>
      <param name="name">vy</param>
      <param name="type">double</param>
      <param name="value">0.0</param>
    </struct>
    <struct>
      <param name="name">vx</param>
      <param name="type">func</param>
      <param name="value">File1</param>
    </struct>
  </list>
</struct>
```

3. Specify the particulars of the file by adding the variables

```
<param name="File1_Name">velocities</param>
<param name="File1_Dim">0</param>
<param name="File1_N">102</param>
```

   to the end of the file (just before `</StGermainData>`).

There is a fully worked out example in `input/cookbook/viscous_file.xml`.

5.6 Viscous Material with Inflow/Outflow Boundaries

This example implements a different kind of boundary condition, where material flows in one side and out another as in Figure 5.4. The current example is not intended to be geologically realistic in any sense, but is meant to illustrate the flexibility we have in the development of complex boundary conditions.

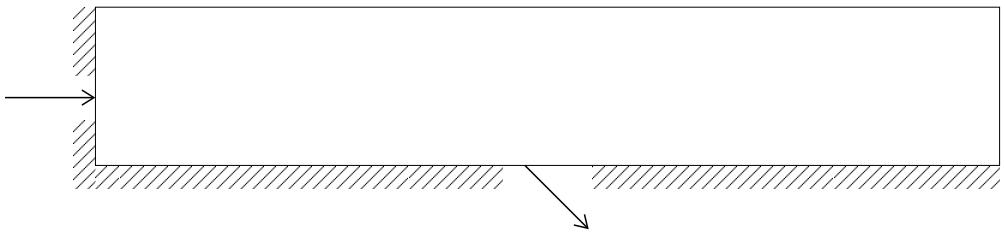


Figure 5.4: Inflow/Outflow Boundary

1. Copy the file `myviscous.xml` that you created in Section 5.2 to `myviscous_inflow.xml`.

2. Then, add the following lines after the `wrapTop` line so that Gale keeps the left and bottom sides fixed:

```
<param name="staticLeft">True</param>
<param name="staticBottom">True</param>
```

3. Now specify the velocity on the boundaries using the `StepFunctionProduct` functions. For the left boundary, modify the left `WallVC` to

```
<param name="type">WallVC</param>
<param name="wall">left</param>
<list name="variables">
  <struct>
    <param name="name">vx</param>
    <param name="type">func</param>
    <param name="value">StepFunctionProduct3</param>
  </struct>
</list>
```

   and add the variables at the end of the file (just before `</StGermainData>`)

```
<param name="StepFunctionProduct3Start">0.1</param>
<param name="StepFunctionProduct3End">0.2</param>
<param name="StepFunctionProduct3Value">1</param>
```

4. For the bottom boundary, modify the bottom `WallVC` to

```
<param name="type">WallVC</param>
<param name="wall">bottom</param>
<list name="variables">
  <struct>
    <param name="name">vy</param>
    <param name="type">func</param>
    <param name="value">StepFunctionProduct2</param>
  </struct>
  <struct>
    <param name="name">vx</param>
    <param name="type">func</param>
    <param name="value">StepFunctionProduct1</param>
  </struct>
</list>
```

   and add the variables to the end of the file (just before `</StGermainData>`)

```
<param name="StepFunctionProduct1Start">0.9</param>
<param name="StepFunctionProduct1End">1.1</param>
<param name="StepFunctionProduct1Value">1.0</param>
<param name="StepFunctionProduct2Start">0.9</param>
<param name="StepFunctionProduct2End">1.1</param>
<param name="StepFunctionProduct2Value">-1.0</param>
```

5. Finally, when Gale knows that there is an inflow condition, it is more careful when creating particles. This slows down the code a little, so it is not enabled by default. To enable this, modify the `PCDVC` struct by adding the line

   ```
   <param name="Inflow">True</param>
   ```

   after the lines

   ```
   <struct name="weights">
     <param name="Type">PCDVC</param>
     <param name="resolutionX">10</param>
     <param name="resolutionY">10</param>
     <param name="resolutionZ">10</param>
     <param name="lowerT">0.6</param>
     <param name="upperT">25</param>
     <param name="maxDeletions">3</param>
     <param name="maxSplits">3</param>
     <param name="MaterialPointsSwarm">materialSwarm</param>
   ```

A worked example is in the file `input/cookbook/viscous_inflow.xml`. Figure 5.5 shows the strain rate invariant and velocity (see Section 4.3.1).

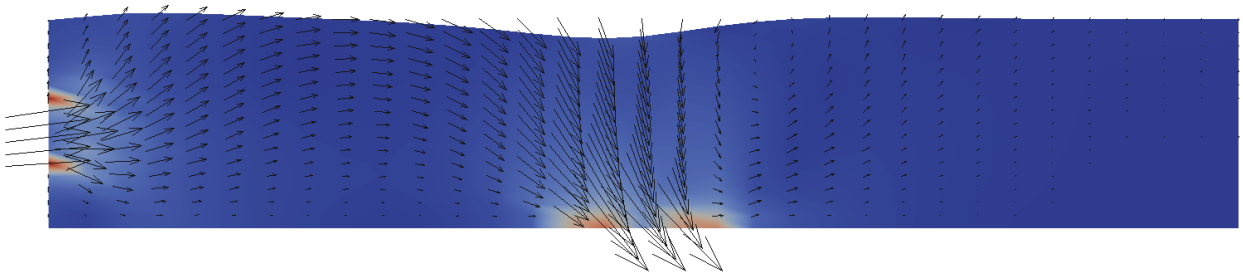


Figure 5.5: Strain rate invariant and velocity with inflow/outflow boundaries

5.7 Viscous Material in Extension with Normal Stress Boundaries

This example modifies the extension example in Section 5.3 to use a stress boundary normal to the bottom surface, instead of specifying the velocity. A normal stress boundary condition simulates the effect of material below the material pushing up, supporting the material in the box. Then, when material piles up, gravity forces will overcome the stress boundary and flow out of the simulation. Conversely, if material is thinned out, the stress boundary will push new material into the simulation. This kind of boundary is often more relevant for geological simulations.

1. Copy `my_viscous_extension.xml` to `my_viscous_normal_stress.xml`

2. Remove the current bottom boundary condition by removing the lines

```
<struct>
  <param name="type">WallVC</param>
  <param name="wall">bottom</param>
  <list name="variables">
    <struct>
      <param name="name">vy</param>
      <param name="type">double</param>
      <param name="value">0.0</param>
    </struct>
  </list>
</struct>
```

3. Add in a StressBC component

```
<struct name="stressBC">
  <param name="Type">StressBC</param>
  <param name="ForceVector">mom_force</param>
  <param name="Swarm">picIntegrationPoints</param>
  <param name="wall">bottom</param>
  <param name="y_type">double</param>
  <param name="y_value">0.35</param>
</struct>
```

   The density of the material is 1, and the height is 0.35, so the stress needed to counteract gravity is 0.35.

4. The bottom essentially becomes an inflow/outflow boundary, so you need to prevent the bottom from moving by adding after

```
<struct>
  <param name="mesh">mesh-linear</param>
  <param name="remesher">velocityRemesher</param>
  <param name="velocityField">VelocityField</param>
  <param name="wrapTop">True</param>
```

   the line

```
<param name="staticBottom">True</param>
```

5. As in Section 5.6, for an inflow condition, you need to modify the `PCDVC` struct by adding the line

```
<param name="Inflow">True</param>
```

   after the lines

```
<struct name="weights">
  <param name="Type">PCDVC</param>
  <param name="resolutionX">10</param>
  <param name="resolutionY">10</param>
  <param name="resolutionZ">10</param>
  <param name="lowerT">0.6</param>
  <param name="upperT">25</param>
  <param name="maxDeletions">3</param>
  <param name="maxSplits">3</param>
  <param name="MaterialPointsSwarm">materialSwarm</param>
```

6. When you deleted the bottom boundary condition, the vertical velocity became unspecified. Recall that the momentum equation (Equation 2.2) only depends on the derivative of the velocity. So stress boundary conditions cannot set the overall magnitude of the velocity. To fix this, you can fix the material to the sides of the simulation. You do this by adding

```
<struct>
  <param name="name">vy</param>
  <param name="type">double</param>
  <param name="value">0.0</param>
</struct>
```

in two places: after

```
<param name="type">WallVC</param>
<param name="wall">left</param>
<list name="variables">
  <struct>
    <param name="name">vx</param>
    <param name="type">double</param>
    <param name="value">0.0</param>
  </struct>
```

and after

```
<param name="type">WallVC</param>
<param name="wall">right</param>
<list name="variables">
  <struct>
    <param name="name">vx</param>
    <param name="type">double</param>
    <param name="value">1.0</param>
  </struct>
```

A worked example is at `input/cookbook/viscous_normal_stress.xml`. Figure 5.6 shows the strain rate invariant and velocity (see Section 4.3.1). Notice that material is now flowing in from the bottom.

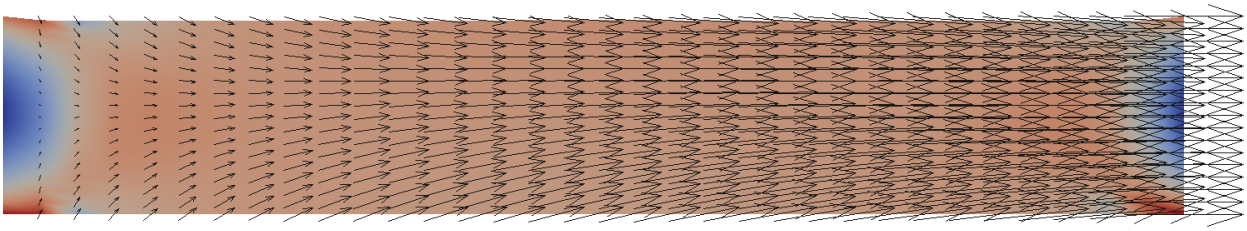


Figure 5.6: Strain rate invariant and velocity of viscous material in extension with a normal stress boundary

5.8 Viscous Material with Deformable Bottom Boundary

The previous example can be modified so that, instead of having material flow through the bottom boundary, the boundary itself deforms. You can do this by changing the one line

```
<param name="staticBottom">True</param>
```

to

```
<param name="wrapBottom">True</param>
```

A worked example is in `input/cookbook/viscous_bottom.xml`. Figure 5.7 shows the strain rate invariant and velocity.

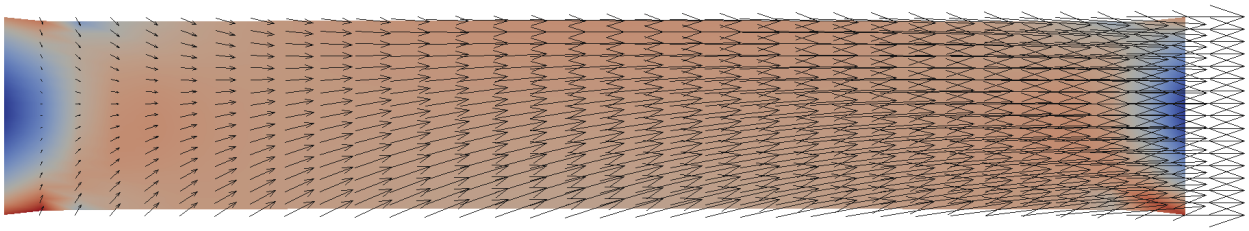


Figure 5.7: Strain rate invariant and velocity of viscous material with a deformable bottom boundary

5.9 Viscous Material with Initially Deformed Upper Boundary

All of the previous examples are set up as a regular rectangular box. However, Gale can also start with the top initially deformed, such as if we had a mountain range with substantial topography. This example will make it sinusoidal as in Figure 5.8. This example has no moving boundaries, so the material will simply relax.

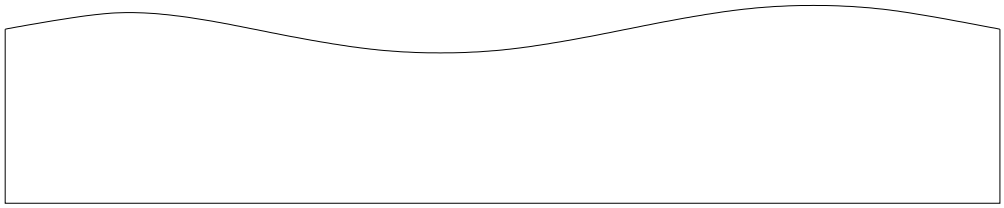


Figure 5.8: Sinusoidal Top

1. Copy `myviscous.xml` to `myviscous_sinusoid.xml`.
2. Add a `SurfaceAdaptor` component:

```
<struct name="surfaceAdaptor">
  <param name="Type">SurfaceAdaptor</param>
  <param name="mesh">mesh-linear</param>
  <param name="sourceGenerator">linearMesh-generator</param>
  <param name="topSurfaceType">sine</param>
  <list name="topOrigin">
    <param>0.0</param>
  </list>
  <param name="topAmplitude">0.1</param>
  <param name="topFrequency">6.28318530718</param>
</struct>
```

A worked example is in `input/cookbook/viscous_sinusoid.xml`. Figure 5.9 shows the strain rate invariant and velocity (see Section 4.3.1).

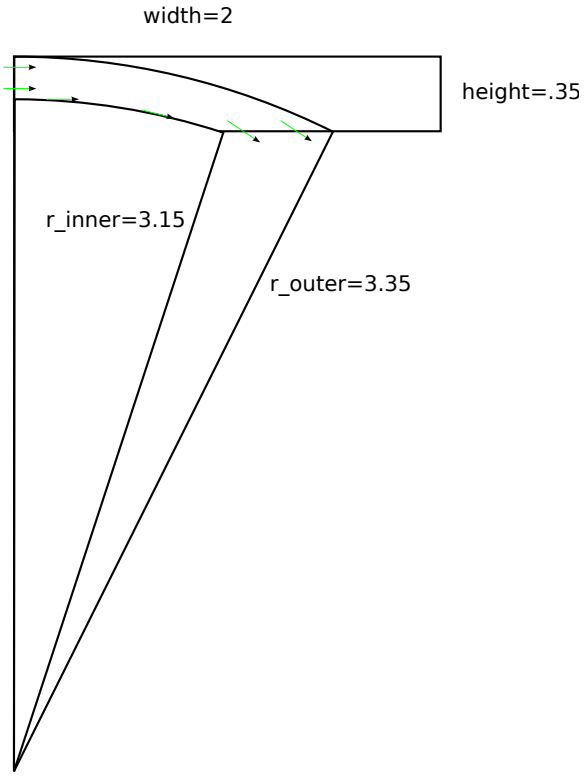


Figure 5.10: Geometry and boundary conditions for the fixed, deformed bottom boundary

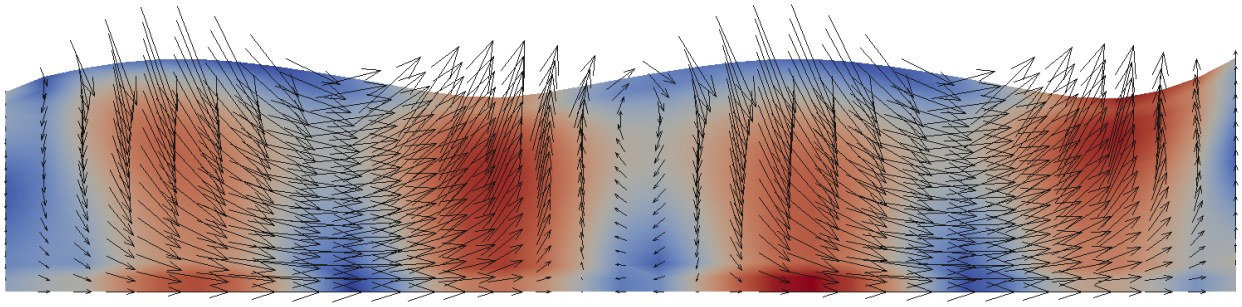


Figure 5.9: Strain rate invariant and velocity with initially deformed upper boundary

5.10 Viscous Material with Fixed, Deformed Bottom Boundary

This example deforms the bottom boundary and keeps it fixed. We will set the left half of the boundary to follow a circle, while the right half will still be flat. Then, the boundary condition for the velocity is set to move the material in from the left and out through the bottom as in Figure 5.10. This is meant to approximate one slab subducting under another.

1. Copy `myviscous_inflow.xml` to `myviscous_deformed_bottom.xml`
2. Add a `SurfaceAdaptor` component for the bottom boundary:

```
<struct name="surfaceAdaptor">
```

```
  <param name="Type">SurfaceAdaptor</param>
  <param name="mesh">mesh-linear</param>
  <param name="sourceGenerator">linearMesh-generator</param>
  <param name="bottomSurfaceType">cylinder</param>
  <param name="bottomX0">SolidBodyRotationCentreX</param>
  <param name="bottomY0">SolidBodyRotationCentreY</param>
  <param name="bottomRadius">InnerRadiusCylinder</param>
  <param name="bottomMinX">CylinderMinX</param>
  <param name="bottomMaxX">CylinderMaxX</param>
</struct>
```

3. Replace the left boundary condition for the velocity

```
<struct>
  <param name="type">WallVC</param>
  <param name="wall">left</param>
  <list name="variables">
    <struct>
      <param name="name">vx</param>
      <param name="type">func</param>
      <param name="value">StepFunctionProduct3</param>
    </struct>
  </list>
</struct>
```

   with

```
<struct>
  <param name="type">WallVC</param>
  <param name="wall">left</param>
  <list name="variables">
    <struct>
      <param name="name">vx</param>
      <param name="type">func</param>
      <param name="value">Velocity_PartialRotationX</param>
    </struct>
    <struct>
      <param name="name">vy</param>
      <param name="type">func</param>
      <param name="value">Velocity_PartialRotationY</param>
    </struct>
  </list>
</struct>
```

4. Replace the bottom boundary condition

```
<struct>
  <param name="type">WallVC</param>
  <param name="wall">bottom</param>
  <list name="variables">
    <struct>
      <param name="name">vy</param>
      <param name="type">func</param>
      <param name="value">StepFunctionProduct2</param>
    </struct>
```

```
        <struct>
          <param name="name">vx</param>
          <param name="type">func</param>
          <param name="value">StepFunctionProduct1</param>
        </struct>
      </list>
    </struct>
```

with

```
    <struct>
      <param name="type">WallVC</param>
      <param name="wall">bottom</param>
      <list name="variables">
        <struct>
          <param name="name">vx</param>
          <param name="type">func</param>
          <param name="value">Velocity_PartialRotationX</param>
        </struct>
        <struct>
          <param name="name">vy</param>
          <param name="type">func</param>
          <param name="value">Velocity_PartialRotationY</param>
        </struct>
      </list>
    </struct>
```

5. Finally, replace the `StepFunction` variables

```
      <param name="StepFunctionProduct3Start">0.1</param>
      <param name="StepFunctionProduct3End">0.2</param>
      <param name="StepFunctionProduct3Value">1</param>
      <param name="StepFunctionProduct1Start">0.9</param>
      <param name="StepFunctionProduct1End">1.1</param>
      <param name="StepFunctionProduct1Value">1.0</param>
      <param name="StepFunctionProduct2Start">0.9</param>
      <param name="StepFunctionProduct2End">1.1</param>
      <param name="StepFunctionProduct2Value">-1.0</param>
```

with rotation variables

```
      <param name="SolidBodyRotationOmega">-1</param>
      <param name="SolidBodyRotationCentreX">0</param>
      <param name="SolidBodyRotationCentreY">-3</param>
      <param name="InnerRadiusCylinder">3.15</param>
      <param name="CylinderMinX">0</param>
      <param name="CylinderMaxX">0.960468635615</param>
      <param name="RadiusCylinder">3.35</param>
```

A worked example is in input/cookbook/viscous_deformed_bottom.xml. Figure shows the strain rate invariant and velocity.

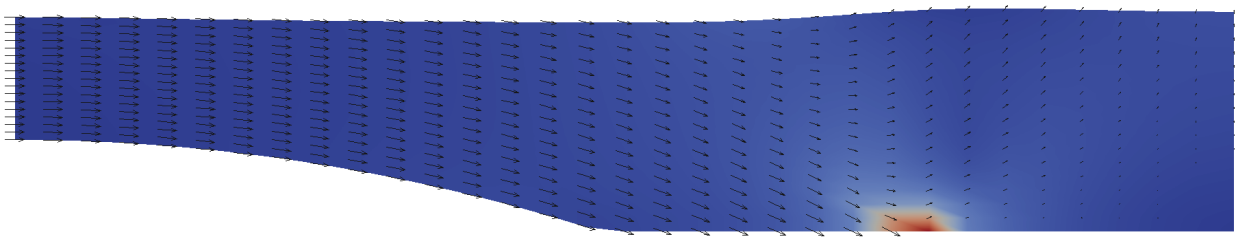


Figure 5.11: Strain rate invariant and velocity for a deformed bottom boundary

5.11 Hydrostatic Term

This example will add a HydrostaticTerm to the simple extension model (Section 5.3) to improve its accuracy. So first copy `myviscous_extension.xml` to `myhydrostatic.xml`. Then add a `HydrostaticTerm` and `StressBC` just before the `BuoyancyForceTerm`

```
<struct name="hydrostaticTerm">
  <param name="Type">HydrostaticTerm</param>
  <param name="upperDensity">1</param>
  <param name="height">maxY</param>
  <param name="gravity">gravity</param>
</struct>
<struct name="stressBC">
  <param name="Type">StressBC</param>
  <param name="ForceVector">mom_force</param>
  <param name="Swarm">picIntegrationPoints</param>
  <param name="wall">top</param>
  <param name="y_type">HydrostaticTerm</param>
  <param name="y_value">hydrostaticTerm</param>
</struct>
```

Then, in the `BuoyancyForceTerm`, add an entry for `HydrostaticTerm` so that the `BuoyancyForceTerm` now looks like

```
<struct name="buoyancyForceTerm">
  <param name="Type">BuoyancyForceTerm</param>
  <param name="ForceVector">mom_force</param>
  <param name="Swarm">picIntegrationPoints</param>
  <param name="gravity">gravity</param>
  <param name="HydrostaticTerm">hydrostaticTerm</param>
</struct>
```

Figure 5.12 shows the strain rate invariant with the same color scale as Figure 5.1. With the `HydrostaticTerm`, the artifacts seen at the top and bottom in Figure 5.1 disappear.

A worked example is in `input/cookbook/hydrostatic.xml`.

5.12 Multiple Viscous Materials

All of the previous examples have only one type of viscous material. This example will create a simulation where there are multiple viscous materials such as in Figure 5.13.

1. Copy `myviscous_extension.xml` (see Section 5.3) to `mymulti_material.xml`.

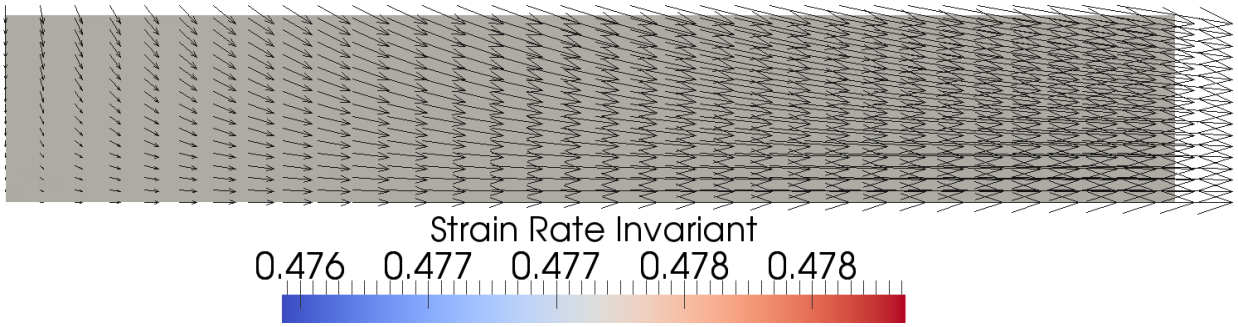


Figure 5.12: Strain rate invariant for an extension model with the hydrostatic pressure subtracted out.

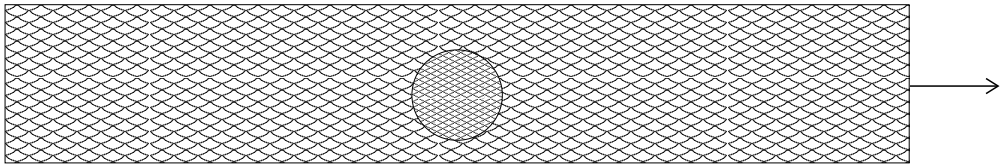


Figure 5.13: Multiple Viscous Materials

2. Add the sphere.

```
<struct name="sphereShape">
  <param name="Type">Sphere</param>
  <param name="CentreX">1.0</param>
  <param name="CentreY">0.15</param>
  <param name="radius">0.10</param>
</struct>
```

   Note: If run in 3D, this is a sphere. The example shown here is run in 2D, so the result is a circle.

3. Then add the new material.

```
<struct name="sphereViscosity">
  <param name="Type">MaterialViscosity</param>
  <param name="eta0">10.0</param>
</struct>
<struct name="sphereViscous">
  <param name="Type">RheologyMaterial</param>
  <param name="Shape">sphereShape</param>
  <param name="density">1.0</param>
  <list name="Rheology">
    <param>sphereViscosity</param>
    <param>storeViscosity</param>
    <param>storeStress</param>
  </list>
</struct>
```

4. Change the shape of the original material so it is not inside the sphere. To do this, create a new shape which is the old shape minus the sphere:

```
<struct name="nonSphereShape">
  <param name="Type">Intersection</param>
  <list name="shapes">
```

```
      <param>boxShape</param>
      <param>!sphereShape</param>
    </list>
  </struct>
```

5. Finally, modify the original viscous material to use this new `nonSphereShape` by changing the line after

```
<struct name="viscous">
  <param name="Type">RheologyMaterial</param>
```

from

```
<param name="Shape">boxShape</param>
```

to

```
<param name="Shape">nonSphereShape</param>
```

A worked example is in `input/cookbook/multi_material.xml`. Figure 5.14 shows the strain rate invariant and velocity (see Section 4.3.1),

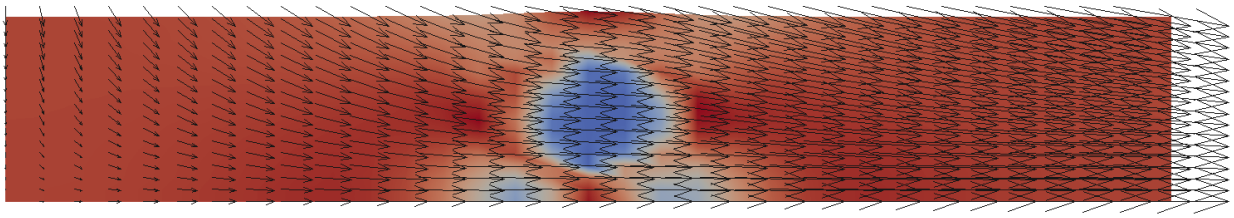


Figure 5.14: Strain rate invariant and velocity with multiple viscous materials

and Figure 5.15 shows the viscosity of the particles.

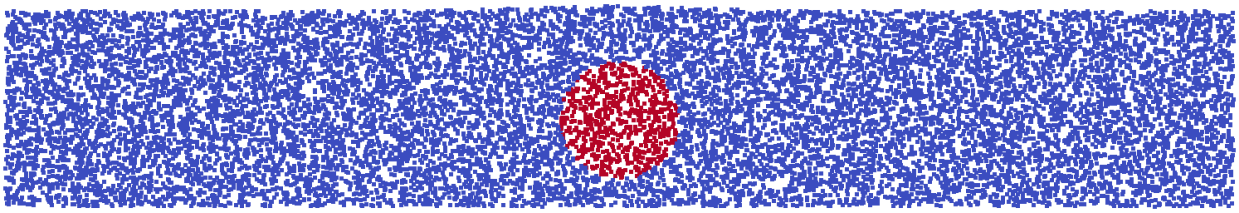


Figure 5.15: Viscosities with multiple viscous materials

5.13 Yielding Material in Simple Extension

This example replaces the background viscous material with a yielding material. This will produce localizations as some material fails.

1. Copy `mymulti_material.xml` to `myyielding.xml`
2. Add a `StrainWeakening` component and a `DruckerPrager` component

```
<struct name="strainWeakening">
  <param name="Type">StrainWeakening</param>
  <param name="TimeIntegrator">timeIntegrator</param>
  <param name="MaterialPointsSwarm">materialSwarm</param>
  <param name="softeningStrain">0.1</param>
  <param name="initialDamageFraction">0.0</param>
  <param name="initialDamageWavenumber">0.5</param>
  <param name="initialDamageFactor">0.5</param>
  <param name="healingRate">0.0</param>
</struct>
<struct name="yielding">
  <param name="Type">DruckerPrager</param>
  <param name="PressureField">PressureField</param>
  <param name="VelocityGradientsField">VelocityGradientsField</param>
  <param name="MaterialPointsSwarm">materialSwarm</param>
  <param name="Context">context</param>
  <param name="StrainWeakening">strainWeakening</param>
  <param name="cohesion">1.0</param>
  <param name="cohesionAfterSoftening">0.0001</param>
  <param name="frictionCoefficient">0.0</param>
  <param name="frictionCoefficientAfterSoftening">0.0</param>
</struct>
```

3. Finally, remove the existing viscous `RheologyMaterial`

```
<struct name="viscous">
  <param name="Type">RheologyMaterial</param>
  <param name="Shape">nonSphereShape</param>
  <param name="density">1.0</param>
  <list name="Rheology">
    <param>backgroundViscosity</param>
    <param>storeViscosity</param>
    <param>storeStress</param>
  </list>
</struct>
```

   and replace it with a yielding `RheologyMaterial`

```
<struct name="crust">
  <param name="Type">RheologyMaterial</param>
  <param name="Shape">nonSphereShape</param>
  <list name="Rheology">
    <param>backgroundViscosity</param>
    <param>yielding</param>
    <param>storeViscosity</param>
    <param>storeStress</param>
  </list>
</struct>
```

A worked example is in `input/cookbook/yielding.xml`. Figure 5.16 shows the strain rate invariant and velocity (see Section 4.3.1).

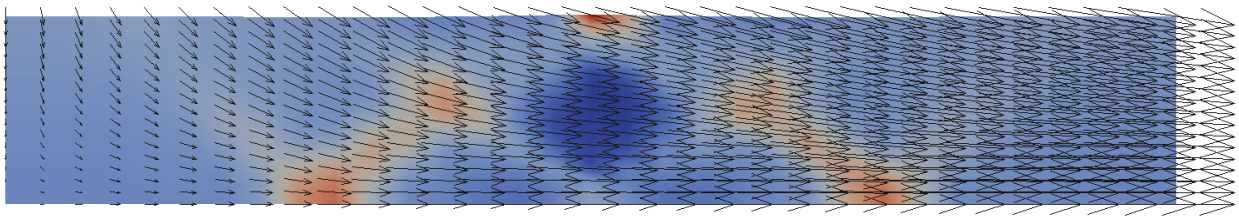


Figure 5.16: Strain rate invariant and velocity of yielding material in extension

Figure 5.17 shows the viscosity of the particles,

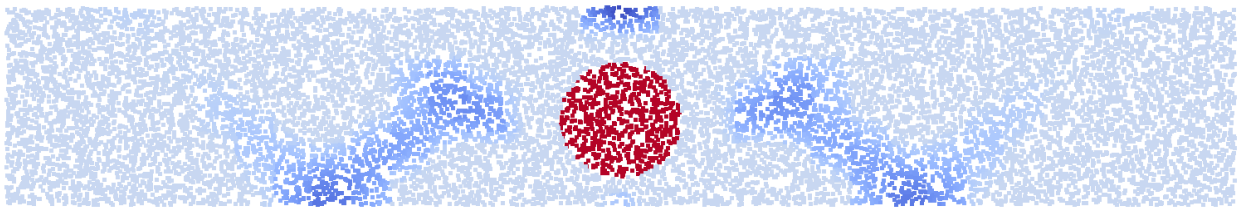


Figure 5.17: Viscosity of yielding material in extension

and Figure 5.18 shows the accumulated post-yielding strain of the particles.

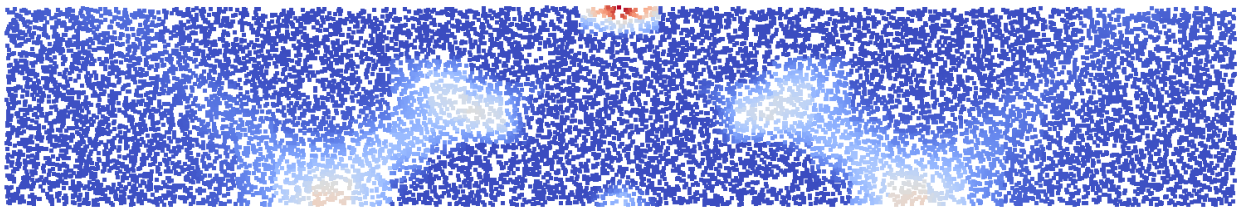


Figure 5.18: Accumulated post-yielding strain of yielding material in extension

5.14 Thermal Convection

Temperature can play a decisive role in geophysical processes. This example takes the multiple viscous material example from Section 5.12, heats it on the bottom, and adds in radiogenic heating throughout.

1. Copy `mymulti_material.xml` to `mythermal.xml`
2. Add in the thermal components from Section A.3.
3. Add in temperature boundary conditions after the velocity boundary conditions

```
<struct name="temperatureBCs">
  <param name="type">CompositeVC</param>
  <list name="vcList">
    <struct>
      <param name="type">WallVC</param>
      <param name="wall">bottom</param>
      <list name="variables">
        <struct>
          <param name="name">temperature</param>
          <param name="type">double</param>
          <param name="value">2.0</param>
        </struct>
      </list>
    </struct>
    <struct>
      <param name="type">WallVC</param>
      <param name="wall">left</param>
      <list name="variables">
        <struct>
          <param name="name">temperature</param>
          <param name="type">double</param>
          <param name="value">1.0</param>
        </struct>
      </list>
    </struct>
    <struct>
      <param name="type">WallVC</param>
      <param name="wall">right</param>
      <list name="variables">
        <struct>
          <param name="name">temperature</param>
          <param name="type">double</param>
          <param name="value">1.0</param>
        </struct>
      </list>
    </struct>
    <struct>
      <param name="type">WallVC</param>
      <param name="wall">top</param>
      <list name="variables">
        <struct>
          <param name="name">temperature</param>
          <param name="type">double</param>
          <param name="value">1.0</param>
        </struct>
      </list>
    </struct>
    <struct>
      <param name="type">WallVC</param>
      <param name="wall">front</param>
      <list name="variables">
        <struct>
          <param name="name">temperature</param>
          <param name="type">double</param>
```

```
            <param name="value">1.0</param>
          </struct>
        </list>
      </struct>
      <struct>
        <param name="type">WallVC</param>
        <param name="wall">back</param>
        <list name="variables">
          <struct>
            <param name="name">temperature</param>
            <param name="type">double</param>
            <param name="value">1.0</param>
          </struct>
        </list>
      </struct>
    </list>
  </struct>
```

4. Add in initial conditions for the temperature after the boundary conditions

```
<struct name="temperatureICs">
  <param name="type">CompositeVC</param>
  <list name="vcList">
    <struct>
      <param name="type">AllNodesVC</param>
      <list name="variables">
        <struct>
          <param name="name">temperature</param>
          <param name="type">double</param>
          <param name="value">1.0</param>
        </struct>
      </list>
    </struct>
  </list>
</struct>
```

5. Specify the background material's thermal expansivity, thermal diffusivity, radiogenic heating rate, and radiogenic decay time scale by adding after

```
<struct name="viscous">
  <param name="Type">RheologyMaterial</param>
  <param name="Shape">boxShape</param>
  <param name="density">1.0</param>
```

   the lines

```
  <param name="alpha">1.0</param>
  <param name="diffusivity">1.0</param>
  <list name="heatingElements">
    <struct>
      <param name="Q">1.0</param>
      <param name="lambda">1.0</param>
    </struct>
  </list>
```

   For the sphere, after the lines

```
<struct name="sphereViscous">
  <param name="Type">RheologyMaterial</param>
  <param name="Shape">sphereShape</param>
  <param name="density">1.0</param>
```

add the lines

```
  <param name="alpha">10.0</param>
  <param name="diffusivity">10.0</param>
  <list name="heatingElements">
    <struct>
      <param name="Q">1000.0</param>
      <param name="lambda">10.0</param>
    </struct>
  </list>
```

This makes the sphere more expansive, conductive, and radioactive.

6. Modify the buoyancy force term by adding the temperature field after

```
<struct name="buoyancyForceTerm">
  <param name="Type">BuoyancyForceTerm</param>
  <param name="ForceVector">mom_force</param>
```

with the line

```
  <param name="TemperatureField">TemperatureField</param>
```

7. Finally, to highlight the effects of temperature, make the boundary move more slowly by changing the line after

```
<param name="type">WallVC</param>
  <param name="wall">right</param>
  <list name="variables">
    <struct>
      <param name="name">vx</param>
      <param name="type">double</param>
```

from

```
  <param name="value">1.0</param>
```

to

```
  <param name="value">0.01</param>
```

A worked example is in `thermal.xml`. Figure 5.19 shows the temperature and velocity.

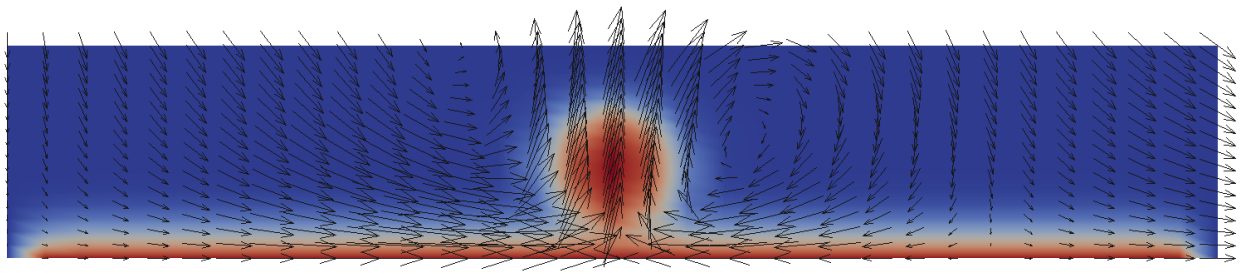


Figure 5.19: Temperature and velocity for the thermal convection example

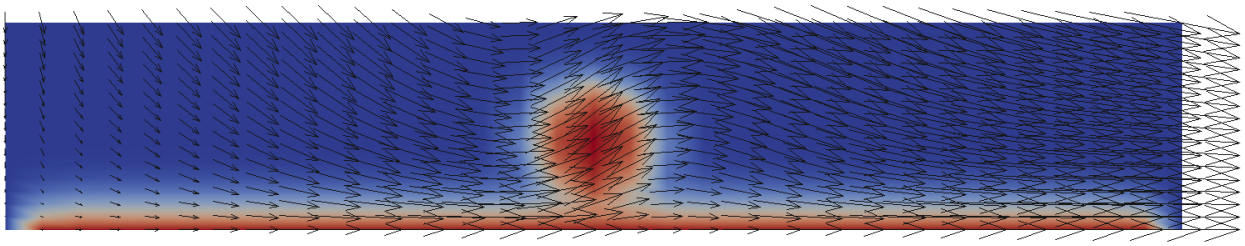


Figure 5.20: Temperature and velocity for the power-law creep model

5.15 Power Law Creep

A common approximation for the rheology of rocks is power law creep. This example shows how to implement this with the NonNewtonian rheology as described in Section A.5.2.4.

1. Copy `mythermal.xml` to `mynon_newtonian.xml`.
2. Add the NonNewtonian rheology after NonSphereShape

```
<struct name="nonNewtonian">
  <param name="Type">NonNewtonian</param>
  <param name="StrainRateInvariantField">StrainRateInvariantField</param>
  <param name="n">3.4</param>
  <param name="T_0">1.0</param>
  <param name="A">1.0</param>
  <param name="refStrainRate">0.01</param>
</struct>
```

3. Change

```
<list name="Rheology">
  <param>backgroundViscosity</param>
  <param>storeViscosity</param>
  <param>storeStress</param>
</list>
```

   to

```
<list name="Rheology">
  <param>nonNewtonian</param>
  <param>storeViscosity</param>
  <param>storeStress</param>
</list>
```

A worked example is in `non_newtonian.xml`. Figure 5.20 shows the temperature and velocity. The differences with the example in Figure 5.19 are mostly because the viscosity is higher everywhere.

Chapter 6

Geologic Example

The previous chapter gave examples on how to make simple problems with simple parameters (e.g., length=1, viscosity=1). Scaling these input parameters to realistic values (e.g., length=1000km, viscosity=10^{26}) should be as easy as changing the various parameters to the right number. In practice, it can be quite difficult, because you may have to change many different parameters at once to ensure a stable solution. To make that transition easier, there is a sample input file in `input/examples/dike.xml` which has a rough model of a magmatic dike [19]. A schematic of the simulation is shown in Figure 6.1.

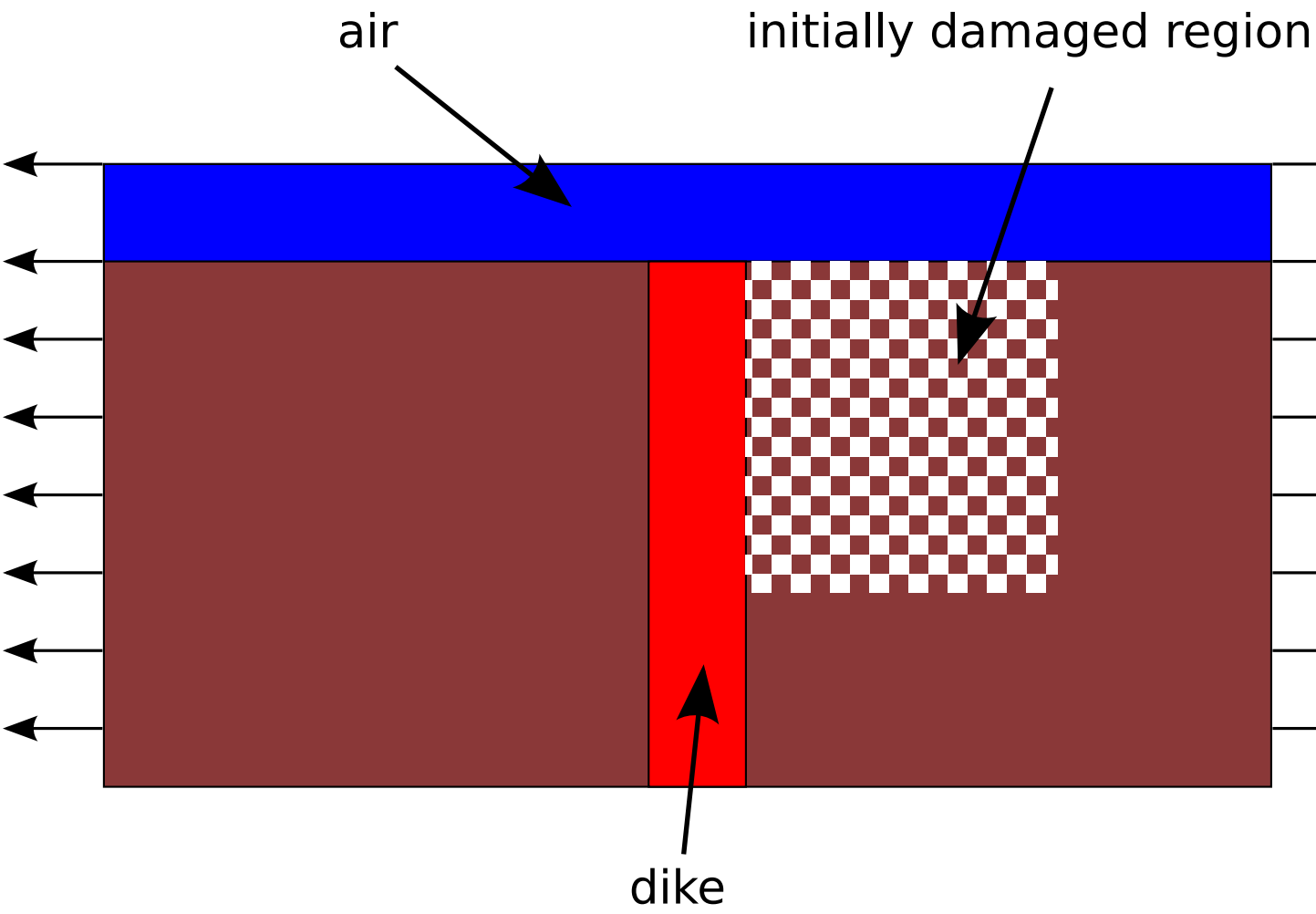


Figure 6.1: Schematic of dike example

The right and left sides are pulled at a constant rate. The region that is modeled is fixed, so material flows in from the bottom and out through the sides. The top layer is a low viscosity 'air' layer that flows in and out of the simulation as time progresses. The dike region is a region of constant divergence, so material is created there *ab nihilo*. This is to model a magma chamber that is fed from small channels from far away.

There is also a nonlinear temperature gradient, going from 273° K at the surface to 1473° K at the bottom. The dike is set to 1500° K. The temperature is fixed to the background and does not advect.

The mantle is modeled with a temperature and strain dependent non-newtonian viscosity and a Drucker-Prager plasticity.

Figure 6.2 shows the integrated strain of a model after 100 steps. The resolution is 120×36, and we used a direct solver on a laptop. There are three prominent faults which propagate out from the dike region.

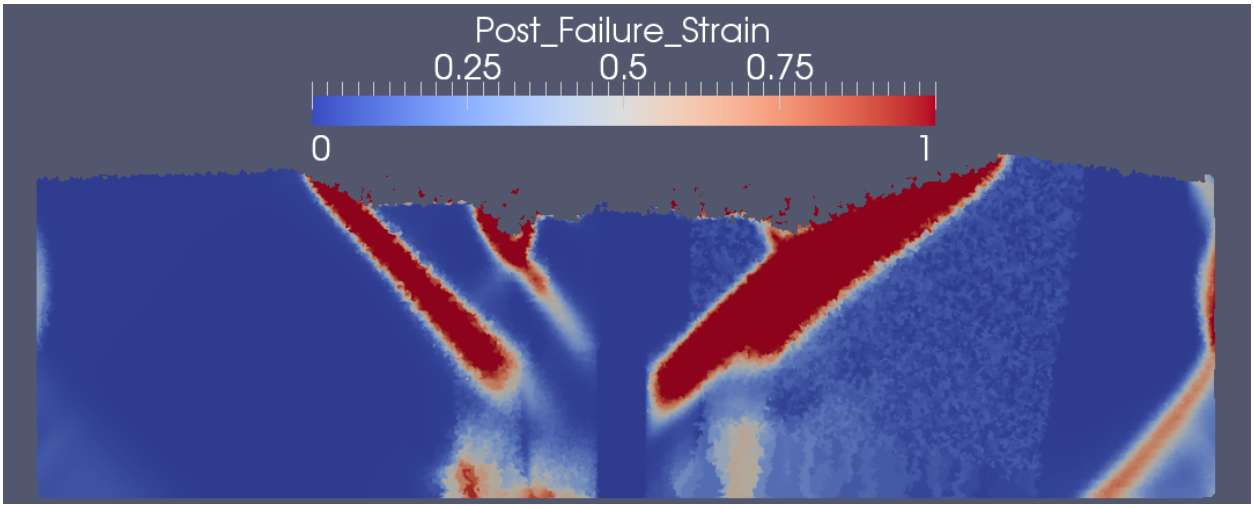


Figure 6.2: Integrated strain for the dike model

Chapter 7

Modifying Gale

7.1 Introduction

If you wish to change material properties, here is a brief overview of how to modify the code. The following is currently lacking in many areas, but will be expanded and refined in future releases.

7.2 Software Components of Gale

Gale makes use of several physics libraries, including StGermain, StgFEM, PiCellerator, and UnderWorld. These are open-source finite element method libraries written by the Victorian Partnership for Advanced Computing (VPAC) and Louis Moresi's group at Monash University (see Fig 7.1). Gale also makes use of PETSc, a suite of data structures and routines for the parallel solution of scientific applications modeled by partial differential equations.

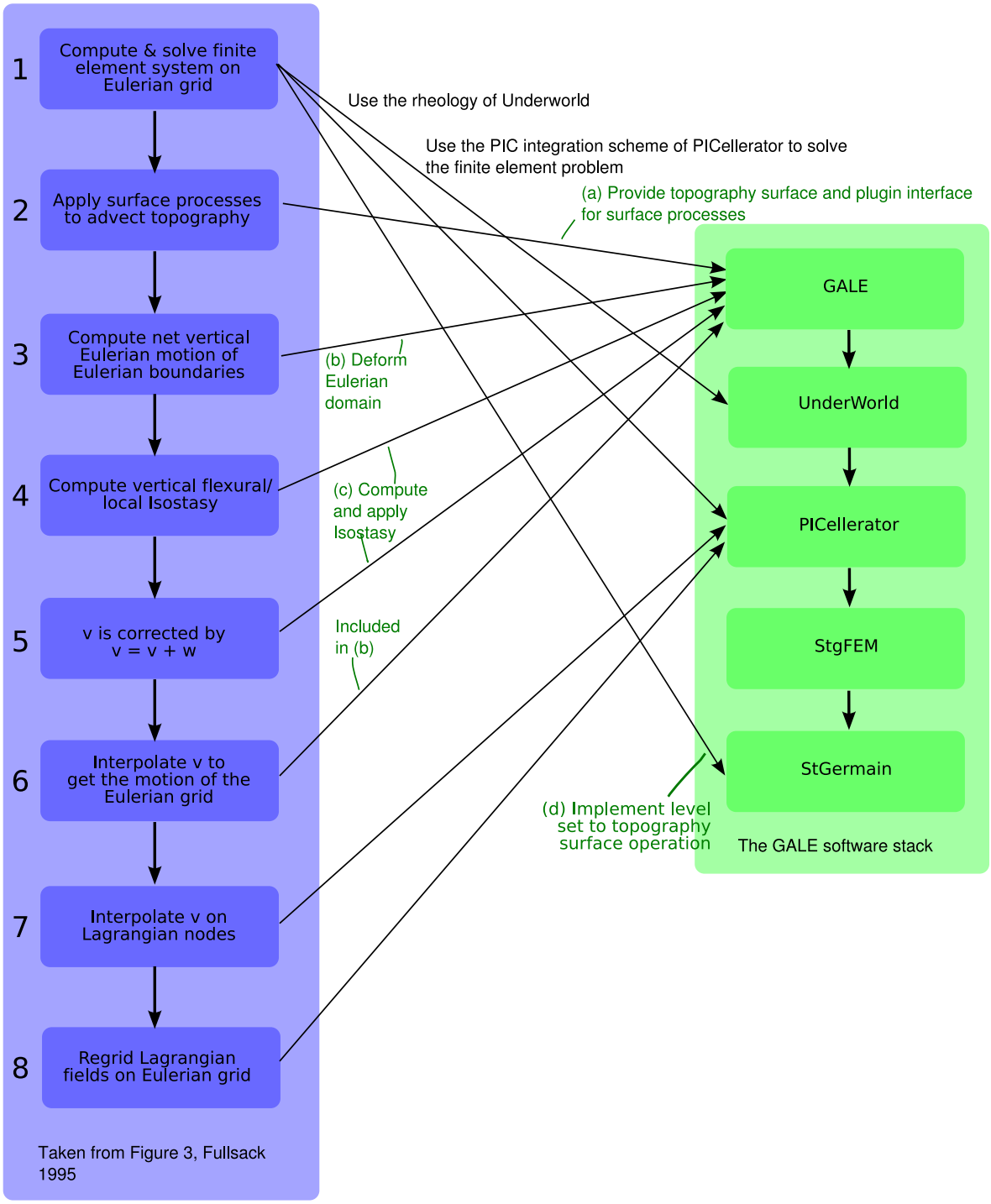
7.2.1 StGermain

StGermain (`csd.vpac.org/twiki/bin/view/Stgermain/WebHome`) provides an infrastructure that can be used to create reusable, collaborative computational development environments. It aims to provide the efficiency and style of coding near that of traditional HPC as well as new techniques and methods in scientific computing. Effectively, it is the application of contemporary software engineering on multi-disciplinary computational research. In particular, StGermain can be used in the development of computational finite element codes. It permits the interchanging of numerical schemes without having to change the problem description or the constitutive rules utilized. It also allows numerical schemes and constitutive rules to be reused for different problems in different disciplines. Scientists can then switch to new computational technologies as they become available. StGermain also capitalizes on the resources invested in software development on a research project, rendering that software effectively reusable for subsequent projects. In turn, intellectual property, skills and adaptability of the recipients develop over time.

7.2.2 PETSc

PETSc (`www-unix.mcs.anl.gov/petsc/petsc-as`), the Portable, Extensible Toolkit for Scientific Computation, is a suite of data structures and routines for the uni- and parallel-processor solution of large-scale scientific application problems modeled by partial differential equations. It employs the MPI standard for all message-passing communication.

Figure 7.1: Mapping between MicroFEM and Gale



7.2.3 StgFEM

StgFEM (`csd.vpac.org/twiki/bin/view/Stgfem/WebHome`) uses the StGermain philosophy of reusability and collaborative development to create a finite element problem composer in terms of both the linear system to be solved and the finite element discretization of the problem domain. The composition can be described in XML and could be represented in a network diagram with an appropriate tool. StgFEM

describes finite element systems for various formulations in a manner that can allow the underlying numerics to be interchanged.

7.2.4 PiCellerator

PICellerator (`csd.vpac.org/twiki/bin/view/PICellerator/WebHome`) (Particle In Cellerator), a Lagrangian Integration Point Finite Element framework, is implemented as an integration scheme substitute for the default Gaussian scheme implemented in StgFEM. The PICellerator concept has since grown to become a general Lagrangian integration scheme framework and a Lagrangian constitutive rule framework. The PIC scheme is provided and other Arbitrary Lagrangian Eulerian schemes are in development. Constitutive rules are reusable across these schemes.

7.2.5 UnderWorld

UnderWorld (`wasabi.maths.monash.edu.au/twiki/view/Software/Underworld`) is a StGermain parallel modeling framework Geoscience research code which utilizes a Lagrangian particle-in-cell finite element scheme (the prototype of which is the Ellipsis code), visualized using gLucifer. UnderWorld (Monash University), StGermain (Victorian Partnership for Advanced Computing or VPAC) and gLucifer (Monash University) are under development as part of the Australian Computational Earth Systems Simulator (ACcESS), an Australian Government National Research Facility, a node of which is located at the Australian Crustal Research Centre (ACRC) at Monash University (Clayton Campus).

7.3 System Description

Gale uses StgFEM to formulate implicit finite element systems, with the bulk of the information placed in a stiffness matrix and a force vector. Depending on the type of solver used, there may be several matrices and vectors. The stiffness matrix class has a member whose purpose is to assemble the entire matrix. It does this through the use of "stiffness matrix terms." The `StiffnessMatrixTerm` class provides an interface through which the elemental contributions to the matrix may be influenced. In typical fluid flow systems, the majority of the physics is applied to the model through constitutive laws which modify the stiffness matrix to reflect certain physical situations. The `ConstitutiveMatrix` class inherits from `StiffnessMatrixTerm`, providing an interface tailored to implementing material-based continuum physics.

So, where does the actual physics take place? The answer is in the Rheology class, but first look at the PIC (particle-in-cell) scheme. There are a number of benefits to using a PIC scheme, but the main use is to store material properties on each particle. These properties can then be used to drive our constitutive laws, in turn modifying the produced stiffness matrices. For each defined material in the domain, there may be a number of associated rheological laws. Whenever an element in the system is assembled, each stiffness matrix term is evaluated, implying that the constitutive matrix will be evaluated for each element. For each particle internal to an element, the `ConstitutiveMatrix` class will evaluate all associated rheologies, each rheology modifying the constitutive matrix.

Modifying the constitutive matrix takes place in the form of viscosity values. In this, probably the simplest of rheological laws, the `ModifyConstitutiveMatrix` method simply sets a specific viscosity value for the current element/material point tuple.

7.4 Sample Rheologies

7.4.1 Simple Viscous

This excerpt is taken from the file `src/Underworld/Rheology/src/MaterialViscosity.c`. It has been inherited from the Rheology class and thus possesses a virtual method named

```
MaterialViscosity_ModifyConstitutiveMatrix
```

which has been overridden to set the viscosity in the constitutive matrix, as follows:

```
void _MaterialViscosity_ModifyConstitutiveMatrix(
   void*                rheology,
   ConstitutiveMatrix*  constitutiveMatrix,
   MaterialPointsSwarm* swarm,
   Element_LocalIndex   lElement_I,
   MaterialPoint*       materialPoint,
   Coord                xi )
{
   MaterialViscosity* self = (MaterialViscosity*)rheology;
   ConstitutiveMatrix_SetIsotropicViscosity( constitutiveMatrix,
                                             self->eta0 );
}
```

The parameters passed to this method provide access to the rheology class's members/methods, the constitutive matrix, the swarm (material points), the index of the element currently being assembled, the material point currently being evaluated, and the material point's elemental coordinates.

7.5 Standard Condition Functions

If you need to write your own Standard Condition Functions (see Section A.13), then the easiest way is to copy and paste an existing function. For example, suppose you wanted to create a function `xSquared` that returns Ax^2, where A is a constant provided in the input file. You start by opening the file

```
src/StgFEM/plugins/StandardConditionFunctions/StandardConditionFunctions.c
```

and finding the function

```
StgFEM_StandardConditionFunctions_Extension
```

Make a copy of that function and rename it to

```
StgFEM_StandardConditionFunctions_xSquared
```

Then you only need to modify the last 6 lines of the function to get the desired behavior. Specifically, it will become

```
void StgFEM_StandardConditionFunctions_xSquared( Node_LocalIndex node_lI,
                                                 Variable_Index var_I,
                                                 void* _context, void* _result ) {
        DiscretisationContext*  context           = (DiscretisationContext*)_context;
        Dictionary*             dictionary        = context->dictionary;
        FeVariable*             feVariable        = NULL;
        FeMesh*                 mesh              = NULL;
        double*                 result            = (double*) _result;
        double*                 coord;
        double                  factor;

        feVariable = (FeVariable*)FieldVariable_Register_GetByName(
                                  context->fieldVariable_Register, "VelocityField" );
        mesh       = feVariable->feMesh;
        /* Find Centre of Solid Body Rotation */
        factor = Dictionary_GetDouble_WithDefault( dictionary, "xSquaredFactor", 1.0 );
        /* Find coordinate of node */
        coord = Mesh_GetVertex( mesh, node_lI );
        *result = factor * (coord[ I_AXIS ]*coord[ I_AXIS ];
}
```

Then you need to register your function at the top of the file. After the lines

```
condFunc = ConditionFunction_New( StgFEM_StandardConditionFunctions_Extension,
                                  "Velocity_Extension" );
ConditionFunction_Register_Add( context->condFunc_Register, condFunc );
```

add the lines

```
condFunc = ConditionFunction_New( StgFEM_StandardConditionFunctions_xSquared, "xSquared" );
ConditionFunction_Register_Add( context->condFunc_Register, condFunc );
```

The last thing to do is to modify

```
src/StgFEM/plugins/StandardConditionFunctions/StandardConditionFunctions.h
```

After the line

```
void StgFEM_StandardConditionFunctions_Extension( Node_LocalIndex node_lI, Variable_Index var_I,
                                                  void* _context, void* _result ) ;
```

add the line

```
void StgFEM_StandardConditionFunctions_xSquared( Node_LocalIndex node_lI, Variable_Index var_I,
                                                 void* _context, void* _result ) ;
```

Now recompile the code, and you should be able to use your new function from input files.

Appendix A

Input File Format

A.1 Structure

The input files are XML files. This leverages a well-known format to specify concepts like hierarchies, lists, parameters, and arbitrary structures. The entire document is placed withing a `StGermainData` structure.

```
<?xml version="1.0"?>
<StGermainData xmlns="http://www.vpac.org/StGermain/XML_IO_Handler/Jun2003">
...
</StGermainData>
```

Within that structure, there are five main parts of every Gale input file: the components, the plugins, EulerDeform, Velocity Conditions, and the variables.

A.1.1 Components

The components section is separated off from the rest of the file with an enclosing `components` structure. This `components` structure is where the bulk of the file will be. It specifies things like what the mesh will be like, which material goes where, what the material properties are, what kind of solver to use, etc. Most of the ideas you need to specify your problem will go into the components. When adding a new component, it is important to remember to put the new component inside the components structure. Otherwise Gale will (silently) not use that component. For example, an input file such as

```
<struct name="components">
  <struct name="conditionFunctions">
    <param name="Type">StgFEM_StandardConditionFunctions</param>
  </struct>
</struct>
```

will correctly initialize `StgFEM_StandardConditionFunctions`, but the input file

```
<struct name="components">
</struct>
<struct name="conditionFunctions">
  <param name="Type">StgFEM_StandardConditionFunctions</param>
</struct>
```

will not, and no error message will alert you of the problem.

A.1.2 Plugins

Gale nominally has the ability to load modules during runtime. Dynamically loading modules is, in general, difficult to get working on multiple platforms. To simplify things, Gale compiles a number of plugins into the code:

```
StgFEM_FrequentOutput
StgFEM_CPUTime
Underworld_MovingMesh
Underworld_Vrms
Underworld_EulerDeform
StgFEM_PrintFeVariableDiscreteValues
Underworld_VTKOutput
```

To use these plugins, list them in a `plugins` section outside of the `components` structure. For example, the following lines will enable the `EulerDeform` and `VTKOutput` plugins

```
<list name="plugins">
  <param>Underworld_EulerDeform</param>
  <param>Underworld_VTKOutput</param>
</list>
```

You can add additional plugins by modifying the list of static plugins in `src/Gale/src/main.c` and making sure that the plugin is compiled into the Gale executable.

A.1.2.1 EulerDeform

The `Underworld_EulerDeform` plugin allows the upper surface to move freely or stay rigidly in place. An example `EulerDeform` struct is

```
<struct name="EulerDeform">
  <list name="systems">
    <struct>
      <param name="mesh">mesh-linear</param>
      <param name="remesher">velocityRemesher</param>
      <param name="velocityField">VelocityField</param>
      <param name="wrapTop">True</param>
      <list name="fields">
        <struct>
          <param name="field">VelocityField</param>
          <param name="variable">velocity</param>
        </struct>
        <struct>
          <param name="field">PressureField</param>
          <param name="variable">pressure</param>
        </struct>
      </list>
    </struct>
  </list>
</struct>
```

This commands Gale to interpolate both the pressure and velocity field onto the new mesh. This interpolation can sometimes go awry. Often, this is because something else is going wrong. For example, if your velocity solution is bad and you get large velocities, then the mesh can turn itself inside out. This, in turn, will cause `EulerDeform` to fail. If you are getting spurious problems with interpolation, then you can turn off this interpolation by removing these two fields from `EulerDeform`. It may slightly affect the speed of your solution, since Gale uses those pressure and velocity fields as a starting guess for the next time step.

Note the critical line

```
<param name="wrapTop">True</param>
```

that makes the top surface conform to the simulation.

Additionally, Gale can fix the positions of the boundaries. For example, if you are running a shortening model, normally Gale will move the boundaries inward as the simulation progresses. If different parts of the boundary are moving at different rates (such as if you were simulating one slab sliding over the other), then the side boundary would quickly become distorted and ruin the simulation. To fix the right boundary, set the variable `staticRight` to `True`

```
<param name="staticRight">True</param>
```

Similarly, you can independently set the left, top, bottom, front, and back boundaries.

Note that this will only fix the interior of that boundary. So setting `staticRight` will not fix the top right or bottom right corners (in 2D) and edges (in 3D). If you set both `staticRight` and `staticBottom`, then the bottom right corner will also be fixed. Otherwise, you can set `staticBottomRight` to specifically fix the bottom right corner.

If you set `staticRight` or `staticLeft` but do not fix the upper corners, then Gale will move the top right or left corner to the boundary and interpolate the height. This is useful if material is flowing out and you want the boundary of the mesh to vary as lumps go through. If material is actually flowing in, Gale will be unable to interpolate and will complain.

The `floatRightTop` and `floatLeftTop` variables are useful when you are using a boundary layer (see Sections A.5.3.3), and you want the height of the boundary layer to match the interior.

Also note that you must include `Underworld_EulerDeform` in the list of plugins (see Section A.1.2) in order for this section to have any effect.

Defaults	
velocityField	-
wrapTop	False
wrapLeft	False
wrapRight	False
staticRight	False
staticRightTop	False
staticRightBottom	False
staticRightFront	False
staticRightBack	False
staticRightTopFront	False
staticRightTopBack	False
staticRightBottomFront	False
staticRightBottomBack	False
staticLeft	False
staticLeftTop	False
staticLeftBottom	False
staticLeftFront	False
staticLeftBack	False
staticLeftTopFront	False
staticLeftTopBack	False
staticLeftBottomFront	False
staticLeftBottomBack	False
staticTop	False
staticTopFront	False
staticTopBack	False
staticBottom	False
staticBottomFront	False
staticBottomBack	False
staticFront	False
staticBack	False
floatLeftTop	False
floatRightTop	False
xRightCoord	-
xLeftCoord	-

A.1.3 Initial and Boundary Conditions

These sections specify the boundary conditions on the velocity, and the initial and boundary conditions for the temperature. See Sections A.6.1, A.6.4, and A.9 for more details.

A.1.4 Variables

The last section is where most of our numeric constants are placed. For example, how many time steps, how often to print output, etc. You may also declare variables for convenience (e.g., the number of grid points) and use it elsewhere, such as in the components. If you are using the basic set of components, then the more important parameters are:

maxTimeSteps The number of time steps to take in the simulation. Each time step can cover a different amount of time. Gale determines how big of a step to take by dividing the grid size by the largest velocity during that time step. Unfortunately, there is no way to stop at a maximum time.

dumpEvery How often to write VTK output (see Section B.1).

checkPointEvery How often to write the checkpoint files (see Section B.2).

outputPath The directory to put output files in. Due to quirks in MPI, you may need to specify this as a full path (e.g., `/home/juser/simulations/myoutput`) rather than a relative path (`myoutput`).

dim The number of dimensions of the problem (2 or 3).

minX,minY,minZ,maxX,maxY,maxZ The physical size of the box you are simulating. Note that this may be modified by `SurfaceAdaptor` (Section A.6.5).

elementResI,elementResJ,elementResK The number of elements in each direction. Note that the number of grid points is one larger (e.g., 64 elements ⇒ 65 grid points).

shadowDepth When running in parallel, every parameter only computes quantities over a portion of the grid. To do this, each processor must keep copies of points that belong to other processors. This parameter specifies how wide the region of copied points is. You should never need to change this from 1.

gaussParticlesX,gaussParticlesY,gaussParticlesZ The number of particles in each direction when putting down particles using a Gaussian distribution. This is used when mapping quantities from the particles to the grid. You should never need to change this number.

particlesPerCell The ideal number of particles in each element. Gale will attempt to keep the number of particles in each element close to this number. You need to vary this number to gauge how sensitive the results of our simulation are to this number.

dtFactor A factor to scale the time step. Ordinarily, Gale will automatically choose an appropriate step size to ensure a stable solution. If you find that to be too large of a step size, you can change `dtFactor` to a smaller number. The default is 1 (no scaling).

dt The size of the time step. Ordinarily, Gale will automatically choose an appropriate step size to ensure a stable solution. For some purposes, it may be convenient to explicitly specify the time step. Be careful! The time step will then be constant over the entire simulation. If the grid stretches and/or velocities become larger than you expect, you may end up with an unstable simulation. The default is 0, which means to use dynamic time stepping.

defaultDiffusivity This is the default diffusivity for all materials. It also indirectly sets the time step. See Section A.3.

maxTimeStepSize The maximum size of the time step. This limit is applied after dtFactor and dt.

seed A random number seed used when placing new particles. You should never need to change this variable, since changing it should not affect the simulation.

A.2 Basic Components

Gale is built on top of StGermain, which is a very general framework for scientific computation. Because StGermain is so general, you have to tell it fairly basic things that would be implicit in most codes. For example, you must tell StGermain that you want to set up a regular mesh and solve a finite element problem on it. This means you have to include a number of components in every input file. These components are

```
<struct name="mesh-linear">
  <param name="Type">FeMesh</param>
  <param name="elementType">linear</param>
</struct>
<struct name="linearMesh-generator">
  <param name="Type">CartesianGenerator</param>
  <param name="mesh">mesh-linear</param>
  <param name="dim">dim</param>
  <param name="shadowDepth">shadowDepth</param>
```

```
    <list name="size">
      <param>elementResI</param>
      <param>elementResJ</param>
      <param>elementResK</param>
    </list>
    <list name="minCoord">
      <param>minX</param>
      <param>minY</param>
      <param>minZ</param>
    </list>
    <list name="maxCoord">
      <param>maxX</param>
      <param>maxY</param>
      <param>maxZ</param>
    </list>
  </struct>
  <struct name="velocity">
    <param name="Type">MeshVariable</param>
    <param name="mesh">mesh-linear</param>
    <param name="Rank">Vector</param>
    <param name="DataType">Double</param>
    <param name="VectorComponentCount">dim</param>
    <list name="names">
      <param>vx</param>
      <param>vy</param>
      <param>vz</param>
    </list>
  </struct>
  <struct name="velocityBCs">
    <param name="Type">CompositeVC</param>
    <param name="Data">mesh-linear</param>
  </struct>
  <struct name="velocityICs">
    <param name="Type">CompositeVC</param>
    <param name="Data">mesh-linear</param>
  </struct>
  <struct name="velocityDofLayout">
    <param name="Type">DofLayout</param>
    <param name="mesh">mesh-linear</param>
    <param name="BaseVariableCount">dim</param>
    <list name="BaseVariables">
      <param>vx</param>
      <param>vy</param>
      <param>vz</param>
    </list>
  </struct>
  <struct name="VelocityField">
    <param name="Type">FeVariable</param>
    <param name="FEMesh">mesh-linear</param>
    <param name="DofLayout">velocityDofLayout</param>
    <param name="BC">velocityBCs</param>
    <param name="IC">velocityICs</param>
    <param name="LinkedDofInfo">velocityLinkedDofs</param>
  </struct>
```

```
<struct name="VelocityMagnitudeField">
  <param name="Type">OperatorFeVariable</param>
  <param name="Operator">Magnitude</param>
  <param name="FeVariable">VelocityField</param>
</struct>
<struct name="VelocityGradientsField">
  <param name="Type">OperatorFeVariable</param>
  <param name="Operator">Gradient</param>
  <param name="FeVariable">VelocityField</param>
</struct>
<struct name="VelocityGradientsInvariantField">
  <param name="Type">OperatorFeVariable</param>
  <param name="Operator">TensorInvariant</param>
  <param name="FeVariable">VelocityGradientsField</param>
</struct>
<struct name="VelocityXXField">
  <param name="Type">OperatorFeVariable</param>
  <param name="Operator">TakeFirstComponent</param>
  <param name="FeVariable">VelocityField</param>
</struct>
<struct name="VelocityYYField">
  <param name="Type">OperatorFeVariable</param>
  <param name="Operator">TakeSecondComponent</param>
  <param name="FeVariable">VelocityField</param>
</struct>
<struct name="StrainRateField">
  <param name="Type">OperatorFeVariable</param>
  <param name="Operator">TensorSymmetricPart</param>
  <param name="FeVariable">VelocityGradientsField</param>
</struct>
<struct name="VorticityField">
  <param name="Type">OperatorFeVariable</param>
  <param name="Operator">TensorAntisymmetricPart</param>
  <param name="FeVariable">VelocityGradientsField</param>
</struct>
<struct name="StrainRateInvariantField">
  <param name="Type">OperatorFeVariable</param>
  <param name="Operator">SymmetricTensor_Invariant</param>
  <param name="FeVariable">StrainRateField</param>
</struct>
<struct name="pressure">
  <param name="Type">MeshVariable</param>
  <param name="mesh">mesh-linear</param>
  <param name="Rank">Scalar</param>
  <param name="DataType">Double</param>
</struct>
<struct name="pressureDofLayout">
  <param name="Type">DofLayout</param>
  <param name="mesh">mesh-linear</param>
  <list name="BaseVariables">
    <param>pressure</param>
  </list>
</struct>
<struct name="PressureField">
```

```
  <param name="Type">FeVariable</param>
  <param name="FEMesh">mesh-linear</param>
  <param name="DofLayout">pressureDofLayout</param>
  <param name="LinkedDofInfo">pressureLinkedDofs</param>
</struct>
<struct name="StressField">
  <param name="Type">StressField</param>
  <param name="StrainRateField">StrainRateField</param>
  <param name="Context">context</param>
  <param name="ConstitutiveMatrix">constitutiveMatrix</param>
  <param name="Swarm">picIntegrationPoints</param>
  <param name="Mesh">mesh-linear</param>
  <param name="IC">stressICs</param>
</struct>
<struct name="ViscosityField">
  <param name="Type">ViscosityField</param>
  <param name="Context">context</param>
  <param name="Swarm">picIntegrationPoints</param>
  <param name="Mesh">mesh-linear</param>
  <param name="ConstitutiveMatrix">constitutiveMatrix</param>
</struct>
<struct name="cellLayout">
  <param name="Type">SingleCellLayout</param>
</struct>
<struct name="particleLayout">
  <param name="Type">GaussParticleLayout</param>
</struct>
<struct name="gaussSwarm">
  <param name="Type">IntegrationPointsSwarm</param>
  <param name="CellLayout">cellLayout</param>
  <param name="ParticleLayout">particleLayout</param>
  <param name="FeMesh">mesh-linear</param>
  <param name="TimeIntegrator">timeIntegrator</param>
  <param name="IntegrationPointMapper">gaussMapper</param>
</struct>
<struct name="gaussMapper">
  <param name="Type">GaussMapper</param>
  <param name="IntegrationPointsSwarm">gaussSwarm</param>
  <param name="MaterialPointsSwarm">gaussMaterialSwarm</param>
</struct>
<struct name="backgroundLayout">
  <param name="Type">BackgroundParticleLayout</param>
</struct>
<struct name="gaussMaterialSwarm">
  <param name="Type">MaterialPointsSwarm</param>
  <param name="CellLayout">cellLayout</param>
  <param name="ParticleLayout">backgroundLayout</param>
  <param name="FeMesh">mesh-linear</param>
</struct>
<struct name="timeIntegrator">
  <param name="Type">TimeIntegrator</param>
  <param name="order">1</param>
  <param name="simultaneous">t</param>
  <param name="Context">context</param>
```

```
</struct>
<struct name="elementCellLayout">
  <param name="Type">ElementCellLayout</param>
  <param name="Mesh">mesh-linear</param>
</struct>
<struct name="weights">
  <param name="Type">PCDVC</param>
  <param name="resolutionX">10</param>
  <param name="resolutionY">10</param>
  <param name="resolutionZ">10</param>
  <param name="lowerT">0.6</param>
  <param name="upperT">25</param>
  <param name="maxDeletions">3</param>
  <param name="maxSplits">3</param>
  <param name="MaterialPointsSwarm">materialSwarm</param>
</struct>
<struct name="localLayout">
  <param name="Type">MappedParticleLayout</param>
</struct>
<struct name="picIntegrationPoints">
  <param name="Type">IntegrationPointsSwarm</param>
  <param name="CellLayout">elementCellLayout</param>
  <param name="ParticleLayout">localLayout</param>
  <param name="FeMesh">mesh-linear</param>
  <param name="WeightsCalculator">weights</param>
  <param name="TimeIntegrator">timeIntegrator</param>
  <param name="IntegrationPointMapper">mapper</param>
</struct>
<struct name="mapper">
  <param name="Type">CoincidentMapper</param>
  <param name="IntegrationPointsSwarm">picIntegrationPoints</param>
  <param name="MaterialPointsSwarm">materialSwarm</param>
</struct>
<struct name="materialSwarmParticleLayout">
  <param name="Type">MeshParticleLayout</param>
  <param name="mesh">mesh-linear</param>
  <param name="cellParticleCount">particlesPerCell</param>
</struct>
<struct name="materialSwarm">
  <param name="Type">MaterialPointsSwarm</param>
  <param name="CellLayout">elementCellLayout</param>
  <param name="ParticleLayout">materialSwarmParticleLayout</param>
  <param name="FeMesh">mesh-linear</param>
  <param name="SplittingRoutine">splittingRoutine</param>
  <param name="RemovalRoutine">removalRoutine</param>
  <param name="EscapedRoutine">escapedRoutine</param>
</struct>
<struct name="materialSwarmAdvector">
  <param name="Type">SwarmAdvector</param>
  <param name="Swarm">materialSwarm</param>
  <param name="TimeIntegrator">timeIntegrator</param>
  <param name="VelocityField">VelocityField</param>
  <param name="PeriodicBCsManager">periodicBCsManager</param>
  <param name="allowFallbackToFirstOrder">True</param>
```

```
</struct>
<struct name="splittingRoutine">
  <param name="Type">ReseedSplitting</param>
  <param name="idealParticleCount">particlesPerCell</param>
</struct>
<struct name="solutionVelocity">
  <param name="Type">SolutionVector</param>
  <param name="FeVariable">VelocityField</param>
</struct>
<struct name="solutionPressure">
  <param name="Type">SolutionVector</param>
  <param name="FeVariable">PressureField</param>
</struct>
<struct name="mom_force">
  <param name="Type">ForceVector</param>
  <param name="FeVariable">VelocityField</param>
  <param name="ExtraInfo">context</param>
</struct>
<struct name="cont_force">
  <param name="Type">ForceVector</param>
  <param name="FeVariable">PressureField</param>
  <param name="ExtraInfo">context</param>
</struct>
<struct name="k_matrix">
  <param name="Type">StiffnessMatrix</param>
  <param name="RowVariable">VelocityField</param>
  <param name="ColumnVariable">VelocityField</param>
  <param name="RHS">mom_force</param>
  <param name="allowZeroElementContributions">False</param>
</struct>
<struct name="constitutiveMatrix">
  <param name="Type">ConstitutiveMatrixCartesian</param>
  <param name="Swarm">picIntegrationPoints</param>
  <param name="StiffnessMatrix">k_matrix</param>
</struct>
<struct name="g_matrix">
  <param name="Type">StiffnessMatrix</param>
  <param name="RowVariable">VelocityField</param>
  <param name="ColumnVariable">PressureField</param>
  <param name="RHS">cont_force</param>
  <param name="allowZeroElementContributions">False</param>
</struct>
<struct name="gradientStiffnessMatrixTerm">
  <param name="Type">GradientStiffnessMatrixTerm</param>
  <param name="Swarm">gaussSwarm</param>
  <param name="StiffnessMatrix">g_matrix</param>
</struct>
<struct name="preconditioner">
  <param name="Type">StiffnessMatrix</param>
  <param name="RowVariable">PressureField</param>
  <param name="ColumnVariable">PressureField</param>
  <param name="RHS">cont_force</param>
  <param name="allowZeroElementContributions">True</param>
</struct>
```

```
<struct name="preconditionerTerm">
  <param name="Type">UzawaPreconditionerTerm</param>
  <param name="Swarm">picIntegrationPoints</param>
  <param name="StiffnessMatrix">preconditioner</param>
</struct>
<struct name="uzawa">
  <param name="Type">Stokes_SLE_UzawaSolver</param>
  <param name="Preconditioner">preconditioner</param>
  <param name="tolerance">1.0e-5</param>
  <param name="maxIterations">5000</param>
</struct>
<struct name="stokesEqn">
  <param name="Type">Stokes_SLE</param>
  <param name="SLE_Solver">uzawa</param>
  <param name="Context">context</param>
  <param name="StressTensorMatrix">k_matrix</param>
  <param name="GradientMatrix">g_matrix</param>
  <param name="DivergenceMatrix"></param>
  <param name="CompressibilityMatrix">c_matrix</param>
  <param name="VelocityVector">solutionVelocity</param>
  <param name="PressureVector">solutionPressure</param>
  <param name="ForceVector">mom_force</param>
  <param name="ContinuityForceVector">cont_force</param>
  <param name="killNonConvergent">false</param>
  <param name="nonLinearMaxIterations">nonLinearMaxIterations</param>
  <param name="nonLinearTolerance">nonLinearTolerance</param>
  <param name="makeConvergenceFile">false</param>
</struct>
<struct name="c_matrix">
  <param name="Type">StiffnessMatrix</param>
  <param name="RowVariable">PressureField</param>
  <param name="ColumnVariable">PressureField</param>
  <param name="RHS">cont_force</param>
  <param name="allowZeroElementContributions">True</param>
</struct>
<struct name="mixedStabiliser">
  <param name="Type">MixedStabiliserTerm</param>
  <param name="Swarm">gaussSwarm</param>
  <param name="picSwarm">picIntegrationPoints</param>
  <param name="storeVisc">storeViscosity</param>
  <param name="StiffnessMatrix">c_matrix</param>
</struct>
<struct name="background">
  <param name="Type">Everywhere</param>
</struct>
<struct name="escapedRoutine">
  <param name="Type">EscapedRoutine</param>
  <param name="idealParticleCount">0</param>
</struct>
<struct name="velocityRemesher">
  <param name="Type">StripRemesher</param>
  <param name="mesh">mesh-linear</param>
  <param name="meshType">regular</param>
  <list name="dim">
```

```
    <param>true</param>
    <param>true</param>
    <param>true</param>
  </list>
</struct>
<struct name="storeViscosity">
  <param name="Type">StoreVisc</param>
  <param name="MaterialPointsSwarm">materialSwarm</param>
</struct>
<struct name="storeStress">
  <param name="Type">StoreStress</param>
  <param name="MaterialPointsSwarm">materialSwarm</param>
</struct>
```

For almost all simulations, you will not need to change these components.

A.3 Temperature components

To configure Gale to use and evolve the temperature, you need to add the following components

```
<!-- Temperature components -->
<struct name="temperature">
  <param name="Type">MeshVariable</param>
  <param name="Rank">Scalar</param>
  <param name="DataType">Double</param>
  <param name="mesh">mesh-linear</param>
</struct>
<struct name="temperatureBCs">
  <param name="Type">CompositeVC</param>
  <param name="Data">mesh-linear</param>
</struct>
<struct name="temperatureICs">
  <param name="Type">CompositeVC</param>
  <param name="Data">mesh-linear</param>
</struct>
<struct name="temperatureDofLayout">
  <param name="Type">DofLayout</param>
  <param name="mesh">mesh-linear</param>
  <list name="BaseVariables">
    <param>temperature</param>
  </list>
</struct>
<struct name="TemperatureField">
  <param name="Type">FeVariable</param>
  <param name="FEMesh">mesh-linear</param>
  <param name="DofLayout">temperatureDofLayout</param>
  <param name="BC">temperatureBCs</param>
  <param name="IC">temperatureICs</param>
  <param name="LinkedDofInfo">temperatureLinkedDofs</param>
</struct>
<struct name="TemperatureGradientsField">
  <param name="Type">OperatorFeVariable</param>
  <param name="Operator">Gradient</param>
  <param name="FeVariable">TemperatureField</param>
```

```
</struct>

<!-- Energy Equation -->
<struct name="residual">
  <param name="Type">ForceVector</param>
  <param name="FeVariable">TemperatureField</param>
</struct>
<struct name="massMatrix">
  <param name="Type">ForceVector</param>
  <param name="FeVariable">TemperatureField</param>
</struct>
<struct name="predictorMulticorrector">
  <param name="Type">AdvDiffMulticorrector</param>
</struct>
<struct name="EnergyEqn">
  <param name="Type">AdvectionDiffusionSLE</param>
  <param name="SLE_Solver">predictorMulticorrector</param>
  <param name="Context">context</param>
  <param name="PhiField">TemperatureField</param>
  <param name="Residual">residual</param>
  <param name="MassMatrix">massMatrix</param>
  <param name="courantFactor">0.25</param>
</struct>
<struct name="lumpedMassMatrixForceTerm">
  <param name="Type">LumpedMassMatrixForceTerm</param>
  <param name="Swarm">gaussSwarm</param>
  <param name="ForceVector">massMatrix</param>
</struct>
<struct name="defaultResidualForceTerm">
  <param name="Type">AdvDiffResidualForceTerm</param>
  <param name="Swarm">gaussSwarm</param>
  <param name="ForceVector">residual</param>
  <param name="ExtraInfo">EnergyEqn</param>
  <param name="VelocityField">VelocityField</param>
  <param name="defaultDiffusivity">defaultDiffusivity</param>
  <param name="UpwindXiFunction">Exact</param>
</struct>
<struct name="internalHeatingTerm">
  <param name="Type">RadiogenicHeatingTerm</param>
  <param name="ForceVector">residual</param>
  <param name="Swarm">picIntegrationPoints</param>
</struct>
```

You should never need to modify these components.

You need to specify the thermal diffusivity. You can specify a single diffusivity for all materials by adding a line like

```
<param name="defaultDiffusivity">1</param>
```

to the list of variables. You can also override this default for each material (see Section A.5). Gale also uses `defaultDiffusivity` when computing the time step. Specifically, it uses the smaller the time step from the Stokes solve and $courantFactor * dx * dx/defaultDiffusity$ (dx is the smallest grid spacing).

You will also need to add in initial and boundary conditions (see Sections A.6.4 and A.9). Finally, you may want to set material properties for the buoyancy forces (see Section A.11) and radiogenic heating (see Section A.5).

Defaults	
defaultDiffusivity	1

A.4 Shapes

When setting up a simulation, Gale reads in shapes to determine what to put where. For example, you can create a simulation with different materials by creating different shapes and putting different materials in them. As a simple example, you can create a 3D box

```
<struct name="simpleBox">
  <param name="Type">Box</param>
  <param name="startX">0.0</param>
  <param name="endX">1.0</param>
  <param name="startY">0.0</param>
  <param name="endY">1.0</param>
  <param name="startZ">0.0</param>
  <param name="endZ">1.0</param>
</struct>
```

You can perform operations on shapes to create new shapes. For example, if you also create a sphere

```
<struct name="simpleSphere">
  <param name="Type">Sphere</param>
  <param name="radius">1.0</param>
</struct>
```

then you can compose it with the box to create a new shape

```
<struct name="nonSphere">
  <param name="Type">Intersection</param>
  <list name="shapes">
    <param>simpleBox</param>
    <param>!simpleSphere</param>
  </list>
</struct>
```

Note that the exclamation point "!" in front of `simpleSphere` means "not." So `Intersection` creates a shape that is the intersection of the box and everywhere outside of the sphere. You can list an arbitrary number of shapes in `Intersection`. Also, you can use `Union` to create a shape that covers all of the input shapes.

In addition, every shape accepts the translation variables `CentreX`, `CentreY`, and `CentreZ`, and the Euler angles `alpha`, `beta`, and `gamma`. So if you modify the Box example above to

```
<struct name="simpleBox">
  <param name="Type">Box</param>
  <param name="CentreX">1.0</param>
  <param name="startX">0.0</param>
  <param name="endX">1.0</param>
  <param name="startY">0.0</param>
  <param name="endY">1.0</param>
  <param name="startZ">0.0</param>
  <param name="endZ">1.0</param>
</struct>
```

then the box will actually span from 1 to 2.

The Euler angles use the y convention, first rotating about the original z axis an angle γ, then rotating around the new y axis an angle β, and finally a rotation around the new z axis an angle α. Specifically, these rotations are expressed through the rotation matrix

$$R = \begin{pmatrix} -\sin\alpha\sin\gamma + \cos\alpha\cos\beta\cos\gamma & \sin\alpha\cos\gamma + \cos\beta\sin\gamma\cos\alpha & -\cos\alpha\sin\beta \\ -\cos\alpha\sin\gamma - \cos\beta\cos\gamma\sin\alpha & \cos\alpha\cos\gamma - \cos\beta\sin\gamma\sin\alpha & \sin\alpha\sin\beta \\ \sin\beta\cos\alpha & \sin\beta\sin\alpha & \cos\beta \end{pmatrix}.$$

So when Gale attempts to figure out whether a coordinate (x, y, z) is inside a shape, it creates a new coordinate

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \left(\begin{pmatrix} x \\ y \\ z \end{pmatrix} - \begin{pmatrix} CentreX \\ CentreY \\ CentreZ \end{pmatrix} \right) R,$$

which it uses in the formulas below.

Finally, you can command Gale to invert the shape with the `invert` variable, making the inside the outside and vice versa.

Defaults	
CentreX	0
CentreY	0
CentreZ	0
alpha	0
beta	0
gamma	0
invert	False

A.4.1 BelowCosinePlane

$$y < offset + delta * \cos\left(\frac{\pi period}{180} * x + phase\right)$$

Defaults	
offset	0
delta	0.5
period	1
phase	0

See also the notes for BelowPlane.

A.4.2 BelowPlane

$$y < offset$$

This shape also uses the variables $minX$, $minY$, $minZ$, $maxX$, $maxY$, and $maxZ$, which are only used when computing the volume of this shape.

Defaults	
offset	0
minX	0
minY	0
minZ	0
maxX	1
maxY	1
maxZ	1

A.4.3 Box

$$\begin{array}{c} startX < x < endX \\ startY < y < endY \\ startZ < z < endZ \end{array}$$

Alternately, you can use widths, in which case

$$\begin{array}{c} |x| < widthX/2 \\ |y| < widthY/2 \\ |z| < widthZ/2 \end{array} .$$

You may mix and match these specification (e.g., use start/end for x, and width for y). If both are specified for one coordinate, Gale will use start and end.

Defaults	
widthX	0
widthY	0
widthZ	0

A.4.4 ConvexHull

This shape is a convenience function for creating intersections of `BelowPlane`. The input is a list of vertices. From each vertex, Gale computes a vector and plane normal to this vector. The half spaces below the planes are then intersected to create a shape. At present, you may only use three vertices in 2D, and four vertices in 3D. As an example, the shape

```
<struct name="triangleShape">
  <param name="Type">ConvexHull</param>
  <list name="vertices">
    <asciidata>
      <columnDefinition name = "x" type="double"/>
      <columnDefinition name = "y" type="double"/>
                                      1.0      1.0
                                     -1.0      1.0
                                      0.0     -1.0
    </asciidata>
  </list>
</struct>
```

creates a shape with vertices at (0,2), (4,-2), (-4,-2).

A.4.5 Cylinder

This shape uses a variable *perpendicularAxis* to determine which direction is perpendicular to the axis of the cylinder. This variable accepts any of (x, y, z, X, Y, Z, i, j, k, I, J, K, 0, 1, 2). For the case where $perpendicularAxis = z$, then

$$\begin{array}{c} radius^2 > x^2 + y^2 \\ startX < x < endX \\ startY < y < endY \\ startZ < z < endZ \end{array} .$$

Defaults	
radius	0
startX	$-\infty$
startY	$-\infty$
startZ	$-\infty$
endX	∞
endY	∞
endZ	∞

A.4.6 Everywhere

This is a convenience shape simply meaning everywhere.

A.4.7 PolygonShape

This is primarily a two-dimensional shape. The input to this shape is a list of vertices. To figure out whether a point is inside the polygon, Gale adds up all of the angles of the vectors going to the vertices. If the point is inside the polygon, then the angles will sum to $\pm 2\pi$, depending on which direction you specify the vertices. If the point is outside the polygon, then the angles sum to 0. A simple example is a triangle

```
<struct name="triangleShape">
  <param name="Type">PolygonShape</param>
  <list name="vertices">
    <asciidata>
    <columnDefinition name = "x" type="double"/>
    <columnDefinition name = "y" type="double"/>
                                    0.0      0.0
                                    1.0      0.0
                                    1.0      1.0
    </asciidata>
  </list>
</struct>
```

This creates a triangle with vertices at (0,0), (1,0), (1,1).

You can extrude this shape into three dimensions by specifying startZ and endZ.

Defaults	
startZ	0
endZ	0

A.4.8 Sphere

$$x^2 + y^2 + z^2 < radius^2$$

Defaults	
radius	0

A.4.9 Superellipsoid

In two dimensions

$$\left(\frac{x}{radiusX}\right)^{2/epsilon1} + \left(\frac{y}{radiusY}\right)^{2/epsilon1} < 1,$$

and in three dimensions

$$\left(\left(\frac{x}{radiusX}\right)^{2/epsilon2} + \left(\frac{y}{radiusY}\right)^{2/epsilon2}\right)^{epsilon2/epsilon1} + \left(\frac{z}{radiusZ}\right)^{2/epsilon1} < 1.$$

Defaults	
radiusX	1
radiusY	1
radiusZ	1
epsilon1	1
epsilon2	1

A.5 Materials

Gale supports two kinds of rheologies: viscous and yielding. You can combine these two rheologies to create a more realistic composite rheology. You then pair this composite rheology with a shape to actually lay down material on the grid. As a simple example, you can create a viscous rheology

```
<struct name="viscousRheology">
  <param name="Type">MaterialViscosity</param>
  <param name="eta0">10.0</param>
</struct>
```

and a Von Mises yielding rheology

```
<struct name="strainWeakening">
  <param name="Type">StrainWeakening</param>
  <param name="TimeIntegrator">timeIntegrator</param>
  <param name="MaterialPointsSwarm">materialSwarm</param>
  <param name="softeningStrain">0.1</param>
  <param name="initialDamageFraction">0.0</param>
  <param name="initialDamageWavenumber">0.5</param>
  <param name="initialDamageFactor">0.5</param>
  <param name="healingRate">0.0</param>
</struct>

<struct name="yieldingRheology">
  <param name="Type">VonMises</param>
  <param name="cohesion">10.0</param>
  <param name="cohesionAfterSoftening">1.0</param>
</struct>
```

and combine them together with `materialShape` (see Section A.4 on how to create shapes)

```
<struct name="yieldingMaterial">
  <param name="Type">RheologyMaterial</param>
  <param name="Shape">yieldingShape</param>
  <list name="Rheology">
```

```
    <param>viscousRheology</param>
    <param>yieldingRheology</param>
  </list>
</struct>
```

For each material, you can specify a density, a coefficient of thermal expansivity (α), and a thermal diffusivity.

The density and expansivity are used by the `BuoyancyForceTerm` component (see Section A.11.1) to create buoyancy forces. The diffusivity is used by the temperature solver (see Section A.3).

You can also specify multiple radiogenic heating rates (Q) and radiogenic timescales (λ). This simulates the action of multiple radioactive materials with different half-lives. To enable this, you must provide a list of Qs and λs. For example, to specify two different radioactive species, add something like

```
<list name="heatingElements">
  <struct>
    <param name="Q">1.0</param>
    <param name="lambda">1.0</param>
  </struct>
  <struct>
    <param name="Q">2.0</param>
    <param name="lambda">2.0</param>
  </struct>
</list>
```

At a given time t, each radioactive element will generate

$$Qe^{-\lambda t}$$

units of energy.

Defaults	
density	0
alpha	0
diffusivity	1
Q	0
lambda	0

A.5.1 StoreVisc and StoreStress

These are not rheologies per se, but rather extra fields where Gale saves the effective isotropic viscosity and components of the stress tensor. For pure viscous materials, the effective viscosity will be the same as the viscosity you supply. For yielding rheologies, the effective viscosity will change as the particle yields. These components needs a `MaterialPointsSwarm`, which in all of the sample input files is called `materialSwarm`.

Defaults	
MaterialPointsSwarm	none

A.5.2 Viscous

A.5.2.1 MaterialViscosity

This is the simplest rheology. There is only one variable, the viscosity `eta0`.

Defaults	
eta0	1

A.5.2.2 Frank-Kamenetskii

This is a temperature-dependent viscosity

$$eta = eta0 * \exp\left(-theta * T\right).$$

Defaults	
eta0	1
theta	0

A.5.2.3 Arrhenius

This is another temperature dependent viscosity

$$eta = eta0 * \exp\left(\left(activationEnergy + activationVolume * \left(height - y\right)\right) / \left(T + referenceTemperature\right)\right).$$

Note that *height* is the height of the column, not the overall maximum height of the material. Also, *height* does not consider material boundaries. So if you have an air layer, you may get surprising results.

Defaults	
eta0	1
activationEnergy	0
activationVolume	0
referenceTemperature	1

A.5.2.4 NonNewtonian

This is a strain rate dependent rheology. It assumes that the material obeys the relation

$$\dot{\varepsilon} = A\tau^n \exp\left(-T_0/T\right),$$

where $\dot{\varepsilon}$ is the strain rate, τ is the stress, and A , T_0, and n are constants. Using

$$\tau = 2\eta\dot{\varepsilon},$$

we can write the viscosity as

$$\eta = \frac{\dot{\varepsilon}^{\frac{1}{n}-1} \exp\left(T_0/nT\right)}{2A^{\frac{1}{n}}}.$$

When setting the viscosity for the first solve, the strain rate has not been calculated yet. So you must supply a reference strain rate for that first step. Gale uses this viscosity to find a solution and thus a new strain rate. Gale then iterates until the strain rate converges.

You may set maximum and minimum values for the resulting viscosity. If the temperature is greater than the melting temperature, then the viscosity is just set to the minimumViscosity.

Defaults	
n	1
T_0	0
T_melt	∞
A	1
refStrainRate	-
minViscosity	-
maxViscosity	-

A.5.3 Yielding

Yielding rheologies are a bit more complicated.

A.5.3.1 StrainWeakening

First you need to create a `StrainWeakening` component. `StrainWeakening` is mainly used to define an initial distribution of strain in a material and to calculate the accumulated strain on each particle. To that end, it requires a number of parameters.

TimeIntegrator This is the component used for time integration to accumulate strain. Given the standard components in Section A.2, this will be `timeIntegrator`.

MaterialPointsSwarm This is the swarm of particles associated with this rheology. Given the standard components in Section A.2, this will be `materialSwarm`.

healingRate With this parameter, accumulated strain can decrease. Specifically, the time derivative of accumulated strain becomes

$$\frac{\sigma_{yield}}{\eta}\left(\frac{\beta}{1-\beta} - healingRate\right),$$

where $\beta \equiv \sigma_{yield}/\sigma$, σ_{yield} is the yield stress, σ is some measure of the current stress (e.g., the second invariant of the stress tensor), and η is the isotropic viscosity. Note that the healing rate should be between 0 and 1.

initialSofteningStrain The strain at which the material starts to yield.

finalSofteningStrain The strain at which the material has fully yielded.

initialDamageFraction The chance that an individual material particle will have a non-zero initial strain.

initialDamageWaveNumber The wavenumber for the initial random strain. To avoid having initial strain on the edges of the box, this should be set to the inverse of the horizontal length of the box.

initialDamageFactor The maximum initial random strain for a particle is `initialDamageFactor`*`finalSofteningStrain`.

randomSeed A random number seed used when computing which particles are initially strained.

initialStrainShape If defined, the initial random strain will only occur within this shape (outside the shape the initial random strain will be zero).

strainLimitedShape If defined, the strain within this shape will not grow beyond `strainLimit`.

strainLimit The maximum amount of strain allowed within `strainLimitedShape`.

You can also define a strain weakening ratio $\alpha \equiv \min(1, \gamma/\gamma_{softening})$, where γ is the accumulated strain, and $\gamma_{softening}$ is the softening strain. This allows us to define quantities like the effective cohesion $C' \equiv C_{pristine}(1-\alpha) + C_{yielded}\alpha$ and effective friction coefficient $\tan\phi' = \tan\phi_{pristine}(1-\alpha) + \tan\phi_{yielded}\alpha$.

Defaults	
TimeIntegrator	none
MaterialPointsSwarm	none
healingRate	0
initialsofteningStrain	0
finalsofteningStrain	∞
initialDamageFraction	0
initialDamageWaveNumber	-1.0
initialDamageFactor	1.0
randomSeed	0
initialStrainShape	none

A.5.3.2 VonMises

This is the simplest yielding rheology in Gale. The yielding stress is simply the effective cohesion. Specifically, the yielding condition specifies

$$\sqrt{J_2} = C'$$

where J_2 is the second invariant of the deviatoric stress tensor. This rheology only has a few input parameters:

- `cohesion` and `cohesionAfterSoftening` have the obvious meanings.
- `minimumYieldStress` sets an absolute minimum to the stress required to make the material yield.
- `StrainRateSoftening` is a Boolean variable that changes how the constitutive matrix is modified when the material has yielded. If `StrainRateSoftening` is `True`, then the viscosity is set to

  $$\eta_{new} = 2C'^2\eta / \left(C'^2 + J_2\right).$$

  This is a way of creeping up on the correct viscosity to avoid setting the viscosity too low. Otherwise the viscosity is set to

  $$\eta_{new} = \eta C' / \sqrt{J_2},$$

  which essentially sets the stress of the particle to the yield stress.

Defaults	
cohesion	0
cohesionAfterSoftening	0
minimumYieldStress	0
StrainRateSoftening	False

A.5.3.3 DruckerPrager

This rheology uses the same parameters as Von Mises, but also adds a friction coefficient that can soften. Specifically, the yield condition is

$$\sqrt{J_2} = Ap + B,$$

where p is the pressure. The value of the constants A and B are different from 2D and 3D. In 2D, Drucker-Prager and Mohr-Coulomb are identical. Specifically, if we write the Mohr-Coulomb yield stress as

$$\sigma_{MC} = C' + \sigma_\perp \tan\phi',$$

then

$$\begin{array}{rcl} A & = & \sin\phi' \\ B & = & C'\cos\phi' \end{array}.$$

In 3D, the mapping between friction angles and cohesion to A and B is more complicated

$$\begin{array}{rcl} A & = & \frac{2\sin\phi'}{\sqrt{3}(3-\sin\phi')} \\ B & = & \frac{6C'\cos\phi'}{\sqrt{3}(3-\sin\phi')} \end{array}.$$

You can also write a Mohr-Coulomb rheology in this form, but then the constants A and B depend on J_2. So reducing the viscosity does not result in a linear decrease in J_2. This makes it difficult for the code to find a solution. In practice, the yield surface for Drucker-Prager and Mohr-Coulomb are not too dissimilar. Mohr-Coulomb's yield surface is a six-sided cone, while Drucker-Prager's yield surface is the smooth cone inscribing the Mohr-Coulomb segmented cone.

Note that `minimumYieldStress` is interpreted differently. If it is zero (the default), then the actual minimum yield stress will be the effective cohesion. This is because there tends to be numerical problems when using a very small minimum yield stress under tension.

When reducing the viscosity, if the second invariant of the strain rate tensor $\dot{\epsilon}$ is greater than `maximumStrainRate` ($\dot{\epsilon}_{max}$) and $\dot{\epsilon}_{max} \neq 0$, then Drucker-Prager sets the new viscosity to

$$\eta_{new} = \frac{Ap + B}{\sqrt{\dot{\epsilon}_{max}}}.$$

Otherwise, Drucker-Prager sets the new viscosity such that the stress will equal the yield stress

$$\eta_{new} = \frac{Ap + B}{\sqrt{\dot{\epsilon}}}.$$

After that, if η_{new} is less than `minimumViscosity`, then η_{new} is set to `minimumViscosity`. See Section 4.2.1 for more details on how to use `maxStrainRate` and `minimumViscosity`.

Also, the Drucker-Prager implementation allows you to specify that material near the boundary will have different yielding properties. This is useful for simulating frictional boundaries. For example, if `boundaryLeft` is `True`, then in the element on the left boundary, Gale will use `boundaryCohesion` instead of `cohesion`, `boundaryFrictionCoefficient` instead of `frictionCoefficient`, etc.

Finally, DruckerPrager requires a pressure. If you are using `HydrostaticTerm` (see Section A.10), you have to remember to give it that as well.

Defaults	
PressureField	none
HydrostaticTerm	none
frictionCoefficient	0
frictionCoefficientAfterSoftening	0
minimumYieldStress	0 (see above)
minimumViscosity	0
maxStrainRate	0
boundaryCohesion	0
boundaryCohesionAfterSoftening	0
boundaryFrictionCoefficient	0
boundaryFrictionCoefficientAfterSoftening	0
boundaryLeft	False
boundaryRight	False
boundaryTop	False
boundaryBottom	False
boundaryFront	False
boundaryBack	False

See also Section A.5.3.2.

A.5.3.4 FaultingMoresiMulhaus2006

This is a fairly complicated non-isotropic rheology. The full details can be found in Moresi and Mülhaus (2006) [4], but essentially it keeps track of which direction a material is strained. To do so, it uses a component called `Director`. For the standard components given in Section A.2, this would be

```
<struct name="director">
  <param name="Type">Director</param>
  <param name="TimeIntegrator">timeIntegrator</param>
  <param name="VelocityGradientsField">VelocityGradientsField</param>
  <param name="MaterialPointsSwarm">materialSwarm</param>
```

```
  <param name="initialDirectionX">0.0</param>
  <param name="initialDirectionY">1.0</param>
  <param name="initialDirectionZ">0.0</param>
  <param name="dontUpdate">True</param>
</struct>
```

Otherwise, it adds one variable not present in `DruckerPrager`: `ignoreOldOrientation`. This tells Gale whether it should check to see whether material will weaken further in the current direction, or if it should try every direction equally each time step.

Defaults	
cohesion	0
cohesionAfterSoftening	0
frictionCoefficient	0
frictionCoefficientAfterSoftening	0
minimumYieldStress	0
ignoreOldOrientation	False

A.6 Boundary Conditions

Gale's computational domain is logically Euclidean. So in 2D, there are four boundaries: `right`, `left`, `top`, and `bottom`. 3D adds `front` and `back`. Note that the boundaries in the z axis are `front` and `back`, not `top` and `bottom`. In many cases, this makes it simple to switch between 2D and 3D. When doing this, you may ignore the warning that the z boundaries are empty in 2D.

A.6.1 Velocity Boundary Conditions

To impose boundary conditions on the velocity, add a composite variable condition (`CompositeVC`) to the input file. Within that `CompositeVC`, add a list of conditions by using `WallVC`s. Within each `WallVC`, specify which boundary and what the velocity's value is. For example, to set the y velocity on the bottom to zero, add

```
<struct name="velocityBCs">
  <param name="type">CompositeVC</param>
  <list name="vcList">
    <struct>
      <param name="type">WallVC</param>
      <param name="wall">bottom</param>
      <list name="variables">
        <struct>
          <param name="name">vy</param>
          <param name="type">double</param>
          <param name="value">0</param>
        </struct>
      </list>
    </struct>
  </list>
</struct>
```

If, instead, you set `vy` to a non-zero value, then the boundary will move as the simulation proceeds. If you want the sides to remain fixed, then you probably want flux boundaries, in which case you will also have to specify a few more things (see Section A.6.2).

You can also set the velocity to a function. For example, to also set the x velocity to have a Gaussian distribution $\exp\left(-\left(\frac{x-0.5}{0.1}\right)^2\right)$

```
<struct name="velocityBCs">
  <param name="type">CompositeVC</param>
  <list name="vcList">
    <struct>
      <param name="type">WallVC</param>
      <param name="wall">bottom</param>
      <list name="variables">
        <struct>
          <param name="name">vy</param>
          <param name="type">double</param>
          <param name="value">0</param>
        </struct>
        <struct>
          <param name="name">vx</param>
          <param name="type">func</param>
          <param name="value">Gaussian</param>
        </struct>
      </list>
    </struct>
  </list>
</struct>
<param name="GaussianHeight">1.0</param>
<param name="GaussianWidth">0.1</param>
<param name="GaussianCenter">0.5</param>
<param name="GaussianDim">0</param>
```

Note that the parameters are separated out into the variables section (see Section A.1.4). In general, you can use any of the Standard Condition Functions (see Section A.13) to specify the velocities.

If you need to specify velocities for only part of the boundary (e.g., the left half moves at vx=1, the right half is unconstrained), then you should use a `ShapeVC` (see Section A.8).

A.6.2 Flux Boundary Conditions

Let's assume you wish to have material flow across the boundary instead of having the boundary move. A simple example would be like Figure 5.11, where material flows in from the left and out through the bottom. There are three things that you must specify for this to work.

1. **The boundaries do not move.** For this model, you need to ensure that, while the material moves, neither the bottom nor left boundaries move. Do this by specifying

   ```
   <param name="staticBottom">True</param>
   <param name="staticLeft">True</param>
   ```

   in EulerDeform (see Section A.1.2.1).

2. **Velocity conditions on the boundaries.** Again, for slab subduction this involves inflow conditions on the left boundary and outflow conditions on bottom. See Section A.6.1 for details. The other boundaries have no-slip conditions.

3. **Special parameter for population control.** You must specifically tell Gale that you are running a simulation where particles may be created by inflow. This is done by specifying

   ```
   <param name="Inflow">True</param>
   ```

   in the `PCDVC` struct.

A.6.3 Stress Boundary Conditions

If the nature of your problem is that stresses are specified on the boundary rather than velocities, you can specify those conditions using the `StressBC` component. For example, if you want to simulate an extension model with isostasy, this is equivalent to adding a supporting stress on the bottom. In equilibrium, the supporting stress cancels the force of gravity, and material does not flow across the boundary. When material piles up, the supporting stress is too weak to support the material, and material flows out. Similarly, when material thins out, the supporting stress overcomes gravity and material flows in.

StressBC is a component, so it must be inside the list of components (see Section A.1.1), not outside the list like the velocity boundary conditions. For example, to incorporate an isostatic bottom boundary condition, you would specify the stress on the bottom boundary in the y direction as a constant.

```
<struct name="stressBC">
  <param name="Type">StressBC</param>
  <param name="ForceVector">mom_force</param>
  <param name="Swarm">picIntegrationPoints</param>
  <param name="wall">bottom</param>
  <param name="y_type">double</param>
  <param name="y_value">1.0</param>
</struct>
```

You can also use the Standard Condition Functions (see Section A.13), but due to technical issues, you must list Standard Condition Functions before StressBC in the list of components. So, for example, to add a stress condition to the left wall with a Gaussian shape, the complete list of components would be

```
<struct name="conditionFunctions">
  <param name="Type">StgFEM_StandardConditionFunctions</param>
</struct>
<struct name="stressBCBottom">
  <param name="Type">StressBC</param>
  <param name="ForceVector">mom_force</param>
  <param name="Swarm">picIntegrationPoints</param>
  <param name="wall">bottom</param>
  <param name="y_type">double</param>
  <param name="y_value">1.0</param>
</struct>
<struct name="stressBCLeft">
  <param name="Type">StressBC</param>
  <param name="ForceVector">mom_force</param>
  <param name="Swarm">picIntegrationPoints</param>
  <param name="wall">left</param>
  <param name="x_type">func</param>
  <param name="x_value">Gaussian</param>
</struct>
```

And then in the list of variables (see Section A.1.4), add the parameters for the `Gaussian`:

```
<param name="GaussianHeight">1.0</param>
<param name="GaussianWidth">0.1</param>
<param name="GaussianCenter">0.5</param>
<param name="GaussianDim">0</param>
```

The type can also be `HydrostaticTerm`. If you are using a `HydrostaticTerm` component (see Section A.10), then you need a `StressBC` component on the top to act as a restoring force when the surface of the material dips below equilibrium. So it would be something like

```
<struct name="stressBCTop">
  <param name="Type">StressBC</param>
  <param name="ForceVector">mom_force</param>
  <param name="Swarm">picIntegrationPoints</param>
  <param name="wall">top</param>
  <param name="y_type">HydrostaticTerm</param>
  <param name="y_value">hydrostaticTerm</param>
</struct>
```

If the bottom boundary can move (that is, you have not set `StaticBottom` in `EulerDeform` and $v_y \neq 0$), and the material outside the simulation is different from inside, then you must also set a StressBC on the bottom. For this, you must also set `bottomDensity`, giving something like

```
<struct name="stressBCTop">
  <param name="Type">StressBC</param>
  <param name="ForceVector">mom_force</param>
  <param name="Swarm">picIntegrationPoints</param>
  <param name="wall">bottom</param>
  <param name="y_type">HydrostaticTerm</param>
  <param name="y_value">hydrostaticTerm</param>
  <param name="bottomDensity">3000</param>
</struct>
```

Note that you must not have a separate `StressBC` for each direction of the normal stress (x,y,z). `StressBC` will automatically apply the proper force in all directions.

A.6.4 Temperature Boundary Conditions

Setting the boundary conditions on the temperature works almost exactly the same as velocity boundary conditions (see Section A.6.1). You need only change `velocityBCs` to `temperatureBCs` and the velocity variable (e.g., `vx`) to `temperature`. For example, to set the bottom temperature to 1, you would add

```
<struct name="temperatureBCs">
  <param name="type">CompositeVC</param>
  <list name="vcList">
    <struct>
      <param name="type">WallVC</param>
      <param name="wall">bottom</param>
      <list name="variables">
        <struct>
          <param name="name">temperature</param>
          <param name="type">double</param>
          <param name="value">1.0</param>
        </struct>
      </list>
    </struct>
  </list>
</struct>
```

A.6.5 Deformed Upper and Lower Boundaries

Normally, Gale starts the simulation in a rectangular box. As the simulation proceeds, the boundaries can become distorted, in particular the upper boundary. However, you can also configure Gale to start with an initially deformed upper or lower boundary by adding a `SurfaceAdaptor` component. A simple example is to make the top a sinusoid

```
<struct name="surfaceAdaptor">
  <param name="Type">SurfaceAdaptor</param>
  <param name="mesh">mesh-linear</param>
  <param name="sourceGenerator">cartesianGenerator</param>
  <param name="topSurfaceType">cosine</param>
  <list name="topOrigin">
    <param>0.0</param>
  </list>
  <param name="topAmplitude">0.1</param>
  <param name="topFrequency">6.28318530718</param>
</struct>
```

This sets the height of the surface to

$$h = h_0 + amplitude * \cos\left(x * frequency\right),$$

where h_0 is the original height.

Note that many of the variables are prefaced with "top". You can also use "bottom" there, and thus modify the height of the bottom boundary. So if you modified the example above to

```
<struct name="surfaceAdaptor">
  <param name="Type">SurfaceAdaptor</param>
  <param name="mesh">mesh-linear</param>
  <param name="sourceGenerator">cartesianGenerator</param>
  <param name="topSurfaceType">cosine</param>
  <list name="topOrigin">
    <param>0.0</param>
  </list>
  <param name="topAmplitude">0.1</param>
  <param name="topFrequency">6.28318530718</param>
  <param name="bottomSurfaceType">cosine</param>
  <list name="bottomOrigin">
    <param>0.0</param>
  </list>
  <param name="bottomAmplitude">0.1</param>
  <param name="bottomFrequency">6.28318530718</param>
</struct>
```

then the top and bottom will follow similar curves.

The other supported surface types are `sine`, `wedge`, `cylinder`, `plateau` and `topo_data`. `sine` takes the same arguments as the `cosine` example above. `wedge` takes three arguments, `BeginOffset`, `EndOffset`, and `Gradient`, and sets the height to

$$h = \begin{cases} h_0 & x < BeginOffset \\ h_0 + Gradient\left(x - BeginOffset\right) & BeginOffset < x < EndOffset \\ h_0 + Gradient\left(EndOffset - BeginOffset\right) & x > EndOffset \end{cases}.$$

`cylinder` takes 6 arguments: `X0`, `Y0`, `Radius`, `MinX`, `MaxX`, and `Sign`. If we define

$$\begin{array}{c} x' \equiv \max\left(\min\left(x, MaxX\right), MinX\right) \\ d \equiv \sqrt{Radius^2 - \left(x - x'\right)^2} \end{array},$$

then the height is set to

$$h = \begin{cases} h_0 + Y0 + d & Sign = true \\ h_0 + Y0 - d & Sign = false \end{cases}.$$

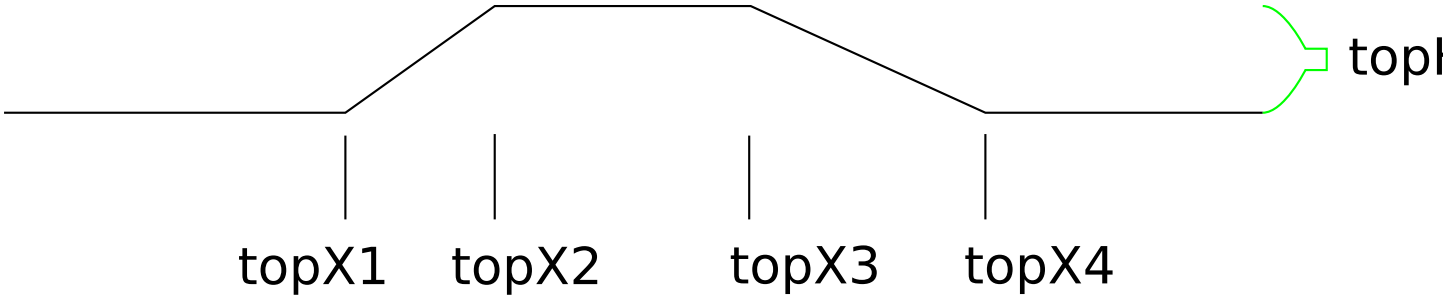


Figure A.1: Height of plateau as a function of the parameters

`plateau` takes up to 9 arguments: `X1`, `X2`, `X3`, `X4`, `Z1`, `Z2`, `Z3`, `Z4`, and `Height`. In 2D, it sets the height as shown in Figure A.1. In 3D, an equivalent thing is done in z, so you end up with a rectangular plateau.

`topo_data` reads in an ascii file with the name from `SurfaceName` ("ascii_topo" by default). The file has a grid with `Nx*Nz` points covering the area from (`MinX`,`MinY`) to (`MaxX`,`MaxY`). Gale then interpolates the heights from that grid to its own grid.

If you want to implement your own surface functions, look in

```
src/StgDomain/Mesh/src/SurfaceAdaptor.c
```

A.6.6 Erosion

Gale has two different models for modeling erosion. After Gale computes a solution to the Stokes flow, both of these work by modifying the velocity of the top nodes of the mesh. So it does not keep track of where material comes from and where it goes.

A.6.6.1 Diffusion

This plugin applies a diffusive operator to the top. Specifically,

$$\frac{\partial y}{\partial t} = -diffusionCoefficient \frac{\partial^2 y}{\partial x^2}.$$

You enable diffusion by adding the plugin `SurfaceProcess`.

A.6.6.2 HRS Erosion

This plugin applies the erosion law as described in Hilley and Strecker [20]. In particular, it forces the slope to be

$$\alpha = \overline{a}_{old} + \tan^{-1}\left(2vTW^{-2} - \frac{\left(2Kk_a^m\right)W^{hm-1}S^n}{(hm+1)\,dt_{erosion}}\right),$$

where

$$\begin{aligned} S &\equiv \tan^{-1}\left(\overline{a}_{old}\right), \\ \overline{a} &\equiv \left(y_{max} - y_0\right)/W, \end{aligned}$$

W, y_{max} and y_0 are determined by the geometry as in Figure A.2, and vT, K, k_a, h, m, n, and $dt_{erosion}$ are specified by the input file. Erosion is only applied at intervals of $dt_{erosion}$ and does not start eroding until after `first_t_erosion`.

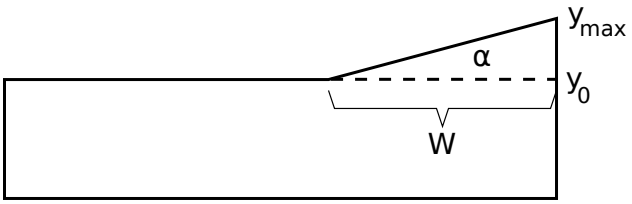


Figure A.2: Geometry for HRS Erosion

Defaults	
vT	-
K	-
ka	-
h	-
m	-
dt_erosion	-
first_t_erosion	-

A.7 Solver Parameters

There are a number of parameters that control solver behavior. Pseudo-code for how it works is

```
for (i=0; i<=nonLinearMaxIterations; ++i)
  {
    for(j=0; j<=maxIterations; ++j)
      {
        Apply one linear iteration;
        if(monitor)
          print out residual and cpu time;
        if(j>=minIterations)
          {
            if((useAbsoluteTolerance
                && absolute_residual<tolerance)
               || (!useAbsoluteTolerance
                   && relative_residual<tolerance))
              break;
          }
      }
    compute non-linear_residual;
    if(i>=nonLinearMinIterations
       && non-linear_residual<nonLinearTolerance)
      break;
    if(i==nonLinearMaxIterations && killNonConvergent)
      abort();
  }
```

The linear iteration step is described more fully in Section 2.2.8.4. The parameters for the linear solve are set in the `Stokes_SLE_UzawaSolver` component

Defaults	
tolerance	10^{-5}
maxIterations	1000
minIterations	1
useAbsoluteTolerance	False
monitor	False

Note that in all of the example input files, `tolerance` is set equal to the global parameter `linearTolerance`. The parameters for the non-linear solve are set in the `Stokes_SLE` component

Defaults	
nonLinearTolerance	10^{-2}
nonLinearMaxIterations	500
nonLinearMinIterations	1
killNonConvergent	True

A.8 Fixing Internal Degrees of Freedom

While the velocity and temperature boundary conditions (see Sections A.6.1 and A.6.4) can be used to specify values on the boundary, it is sometime necessary to specify values within the domain as well. For example, the region that you want to simulate may not map nicely to a rectangular domain. You can fix the internal degrees of freedom for the areas outside of your irregular domain with a `MeshShapeVC`. It works very similar to `WallVC`, except that you supply a shape rather than a wall for the condition to work on. For example, adding

```
<struct>
  <param name="type">MeshShapeVC</param>
  <param name="Shape">fixedShape</param>
  <list name="variables">
    <struct>
      <param name="name">vy</param>
      <param name="type">double</param>
      <param name="value">0</param>
    </struct>
  </list>
</struct>
```

to the list of `WallVC`s in the `CompositeVC` will fix the y velocity in the `fixedShape` region. Note that you can also employ this as a boundary condition by making `fixedShape` only cover a wall. The main advantage of this approach over a `WallVC` is that you can have it only cover a part of the wall, thus constraining only part of the boundary. So if you wanted half of the boundary to move at a certain velocity, but wanted the other half unconstrained, you would use a `ShapeVC`.

There is one important drawback to using a `MeshShapeVC`. `MeshShapeVC` constrains mesh points defined by a shape initially. However, if the mesh deforms, then `MeshShapeVC` will still constrain the same points on the grid. These points will be at a different location in space, so the constraint is now operating on a different area. The only way to really prevent the mesh from deforming is to use static sides (see Section A.1.2.1) everywhere.

A.9 Temperature Initial Conditions

For temperature dependent problems, you need to set initial conditions for the temperature. Because we are ignoring inertial effects, the interior velocity is completely determined by the boundary conditions. Setting initial conditions is similar to setting boundary conditions. The only difference is to change the condition type from `WallVC` to `AllNodesVC`. As an example, to set the initial temperature everywhere to 1, you would add

```
<struct name="temperatureICs">
  <param name="type">CompositeVC</param>
  <list name="vcList">
    <struct>
      <param name="type">AllNodesVC</param>
```

```
        <param name="wall">bottom</param>
        <list name="variables">
          <struct>
            <param name="name">temperature</param>
            <param name="type">double</param>
            <param name="value">1.0</param>
          </struct>
        </list>
      </struct>
    </list>
  </struct>
```

A.10 HydrostaticTerm

This term subtracts out the hydrostatic part of the pressure as described in Section 2.2.8.3. It can subtract out the pressure from a two layer model with a temperature profile described by `the TemperatureProfile` standard condition function (see Section A.13). Specifically, it subtracts out a density given by

$$\begin{array}{cc} 0 & x > height \\ upperDensity \cdot (1 - upperAlpha \cdot T) & materialBoundary > x > height \\ lowerDensity \cdot (1 - lowerAlpha \cdot T) & x > materialBoundary \end{array} ,$$

where

$$T = \begin{cases} T\_0 & x > height \\ T\_0 + linearCoefficient\,(height - x) & x < height \\ +exponentialCoefficient1\,(1 - \exp\,(-exponentialCoefficient2\,(height - x))) & \end{cases} .$$

This component also computes a pressure by integrating this density profile analytically and multiplying by *gravity*. Once you have created a `HydrostaticTerm`, you have to remember to pass it along to `BuoyancyForceTerm` (Section A.11.1) and a `StressBC` (Section A.6.3) in order for it to take effect. You also have to pass it to any `DruckerPrager` rheologies (Section A.5.3.3) that you are using, so that the rheology will have the correct pressure.

Defaults	
upperDensity	0
upperAlpha	0
lowerDensity	0
lowerAlpha	0
height	0
materialBoundary	0
T_0	0
linearCoefficient	0
exponentialCoefficient1	0
exponentialCoefficient2	0
gravity	0

A.11 Buoyancy Forces

Gales supports two types of buoyancy forces. The first one, `BuoyancyForceTerm`, is more general, allowing you to specify buoyancy properties for each material.

A.11.1 BouyancyForceTerm

If you add this component, then there will be a force on each particle of

$$F = -\rho g.$$

If you specify a `TemperatureField`, then the force becomes

$$F = -\rho g \left(1 - \alpha T\right).$$

The density (ρ) and coefficient of thermal expansivity (α) are taken from the material properties (see Section A.5). The vector `gravityDirection` determines the direction of the force. In the sample input files, `ForceVector` is always `mom_force`, and `Swarm` is always `picIntegrationPoints`.

If you have defined a `HydrostaticTerm` (Section), then you need to pass it in to the `BuoyancyForceTerm` for it to take effect.

Defaults	
gravity	0
gravityDirection	(0,1,0)
TemperatureField	none
ForceVector	none
Swarm	none
HydrostaticTerm	none

A.11.2 BuoyancyForceTermThermoChem

If you add this component, then there will be a vertical force on each particle of

$$F = -\rho Ra_C.$$

If you specify a `TemperatureField`, then the force becomes

$$F = Ra_T T - \rho Ra_C.$$

The thermal (Ra_T) and chemical (Ra_C) Rayleigh numbers are the same for all materials. In contrast to `BuoyanceForceTerm`, the force is always in the vertical (y) direction. In the sample input files, `ForceVector` is always `mom_force`, and `Swarm` is always `picIntegrationPoints`.

Note that this component does not take a `HydrostaticTerm`, so you can not use this component if you are using `HydrostaticTerm`.

Defaults	
RaC	0
RaT	0
TemperatureField	none
ForceVector	none
Swarm	none

A.12 Divergence Forces

As mentioned in Section 2.2.5, it is possible to add a divergence force to the continuity equation so that material is created anew. The first three parameters will always be the same between input files.

ForceVector cont_force

Swarm picIntegrationPoints

GeometryMesh mesh-linear

The last three parameters specify the divergence.

DomainShape The divergence is only non-zero outside of this shape.

force_type This can be either "double" or "func." If set to "double," then the divergence force will be a constant. If set to "func," then the divergence force can be any of the Standard Condition Functions (see Section A.13).

force_value If "force_type" is "double," then this must be a number. If "force_type" is "func," then it must be the textual name of one of the Standard Condition Functions (e.g., StepFunction).

A.13 Standard Condition Functions

Standard Condition Functions are functions that you can use to specify initial conditions and boundary conditions. At present, they take in a coordinate (x, y, z) and output a function $f(x, y, z)$. They are all defined in the directory `src/StgFEM/plugins/StandardConditionFunctions/`. For the following descriptions, the sides of the box are at x_{min} and x_{max}, and $L \equiv x_{max} - x_{min}$, and all names with CamelCase capitalization are variables from the input file. If you need to write your own function, see Section 7.5.

Velocity_SolidBodyRotation Returns the coordinates rotated by `SolidBodyRotationOmega` around the z axis, centered at the coordinate (`SolidBodyRotationCentreX`, `SolidBodyRotationY`), out to a radius `RadiusCylinder`. Specifically, if $r < RadiusCylinder$

$$\begin{aligned} f(x,y,z)_x &= -SolidBodyRotationOmega(y - SolidBodyRotationCentreY) \\ f(x,y,z)_y &= SolidBodyRotationOmega(x - SolidBodyRotationCentreX) \end{aligned} ,$$

otherwise returns 0.

Velocity_PartialRotationX Returns the x component of Velocity_SolidBodyRotation.

Velocity_PartialRotationY Returns the y component of Velocity_SolidBodyRotation.

TaperedRotationX If $r < RadiusCylinder$, returns Velocity_PartialRotationX. If $RadiusCylinder < r < TaperedRadius$, returns

$$VelocityPartialRotationX\left(\frac{TaperedRadius - r}{TaperedRadius - RadiusCylinder}\right).$$

If $r > TaperedRadius$, returns 0.

TaperedRotationY Similar to TaperedRotationX, but returns Velocity_PartialRotationY.

Velocity_SimpleShear $f(x, y, z) = SimpleShearFactor\,(y - SimpleShearCentreY)$

Velocity_Extension $f(x, y, z) = ExtensionFactor\,(x - ExtensionCentreX)$

Velocity_PartialLid_TopLayer Returns 0 if the x coordinate is within one grid point of the boundary; 1 otherwise.

Velocity_LinearInterpolationLid

$$f(x,y,z) = bcLeftHandSideValue + \left(\frac{bcRightHandSideValue - bcLeftHandSideValue}{L}\right)x$$

Velocity_Lid_RampWithCentralMax $f(x,y,z) = \begin{cases} 2\,(x - x_{min})\,/L & x < L/2 + x_{min} \\ 1 - 2(x - x_{min} - L/2) & x > L/2 + x_{min} \end{cases}$

Velocity_SinusoidalLid

$$f(x,y,z) = \sin\left(\frac{\pi\left(x - x_{min}\right)}{L} sinusoidalLidWavenumber\right)$$

Velocity_Lid_CornerOnly Returns 1 if the coordinate is on the right boundary.

Temperature_CosineHill Returns a hill defined by a cosine. Specifically, if we define pre-rotated coordinates

$$\begin{array}{rcl} x_{pre-rotate} & = & CosineHillCentreX - SolidBodyRotationCentreX \\ y_{pre-rotate} & = & CosineHillCentreY - SolidBodyRotationCentreY \\ z_{pre-rotate} & = & CosineHillCentreZ - SolidBodyRotationCentreZ \end{array} ,$$

and then rotate them around the z axis by the angle $\theta = SolidBodyRotationOmega * t$

$$\begin{array}{rcl} x_{hill} & = & x_{pre-rotate}\cos\theta - y_{pre-rotate}\sin\theta \\ y_{hill} & = & x_{pre-rotate}\sin\theta + y_{pre-rotate}\cos\theta \\ z_{hill} & = & z_{pre-rotate} \end{array} ,$$

then

$$f(x,y,z) = \begin{cases} \frac{CosineHillHeight}{4}\cos\left(\frac{2\pi r}{CosineHillDiameter}\right) & r < CosineHillDiameter \\ 0 & r > CosineHillDiameter \end{cases} ,$$

where r is the distance from the center of the hill

$$r \equiv \sqrt{\left(x - x_{hill}\right)^2 + \left(y - y_{hill}\right)^2 + \left(z - z_{hill}\right)^2}.$$

LinearWithSinusoidalPerturbation If you scale the y coordinate

$$y_{scaled} = \left(y - y_{min}\right) / \left(y_{max} - y_{min}\right),$$

then this returns

$$\begin{array}{rcl} f(x,y,z) & = & SinusoidalTempIC\_TopLayerBC \\ & & + \left(SinusoidalTempIC\_TopLayerBC - SinusoidalTempIC\_TopLayerBC\right)\left(1 - y_{scaled}\right) \\ & & + SinusoidalTempIC\_PerturbationAmplitude \\ & & * \left(\cos\left(\pi x * SinusoidalTempIC\_HorizontalWaveNumber\right)\right. \\ & & \left. + sin\left(\pi y SinusoidalTempIC\_VerticalWaveNumber\right)\right) \end{array} .$$

Temperature_Trigonometry

$$f(x,y,z) = 1 - \frac{\pi y}{2}\sin\left(\frac{\pi x}{x_{max} - x_{min}}\right).$$

VelicTemperatureIC

$$f(x,y,z) = sigma * \cos\left(\pi wavenumberX\left(\frac{x - x_{min}}{x_{max} - x_{min}}\right)\right)\sin\left(\pi\left(y - y_{min}\right)wavenumberY\right),$$

where the height of the box is constrained to $y_{max} - y_{min} = 1$.

VelicTemperatureIC_SolB

$$f(x,y,z) = sigma * \cos\left(\pi wavenumberX\left(\frac{x - x_{min}}{x_{max} - x_{min}}\right)\right)\sinh\left(\pi wavenumberY\left(\frac{y - y_{min}}{x - x_{min}}\right)\right),$$

where the height of the box is constrained to $y_{max} - y_{min} = 1$.

AnalyticalTemperatureIC First, define

$$\begin{array}{rcc} x_0 & \equiv & x - x_{min} \\ y_0 & \equiv & y - y_{min} \\ L & \equiv & x_{max} - x_{min} \\ H & \equiv & y_{max} - y_{min} \\ \lambda & \equiv & L/H \end{array} ,$$

and then compute some intermediate quantities

$$\begin{array}{rcc} u_0 & = & \frac{\lambda^{7/3}}{(1+\lambda^4)^{2/3}} \left(\frac{Ra}{2\sqrt{pi}}\right)^{2/3} \\ v_0 & = & u_0/\lambda \\ Q & = & 2\sqrt{\lambda/\left(\pi u_0\right)} \\ T_u & = & \frac{1}{2} erf(\left(\frac{1}{2}\left(1-y_0\right)\sqrt{u_0/x_0}\right) \\ T_l & = & 1 - \frac{1}{2} erf\left(\frac{1}{2} y_0 \sqrt{u_0/\left(\lambda - x_0\right)}\right) \\ T_r & = & \frac{1}{2} + \frac{1}{2}\left(Q/\sqrt{\pi}\right)\sqrt{v_0/\left(y_0+1\right)}\exp\left(-x_0^2 v_0/\left(4y_0+4\right)\right) \\ T_s & = & \frac{1}{2} - \frac{1}{2}\left(Q/\sqrt{\pi}\right)\sqrt{v_0/\left(2-y_0\right)}\exp\left(-\left(\lambda - x_0\right)^2 v_0/\left(8-4y_0\right)\right) \\ g & = & T_u + T_l + T_r + T_s - 1.5 \end{array} .$$

The result is

$$f(x,y,z) = \begin{cases} 0 & g < 0 \\ g & 0 < g < 1 \\ 1 & g > 1 \end{cases} .$$

SinusoidalExtension

$$\begin{array}{rcc} f(x,y,z) & = & SinusoidalExtensionVelocity + SinusoidalExtensionAmplitude \\ & & * \cos\left(2\pi SinusoidalExtensionFrequency\left(t + dt - SinusoidalExtensionPhaseShift\right)\right) \end{array}$$

StepFunction This function returns a ramp function in the axis prescribed by the integer $StepFunctionDim$, where $0 \Rightarrow x$, $1 \Rightarrow y$, $2 \Rightarrow z$. Defining some convenient constants

$$\begin{array}{rcc} w & = & coord[dim] \\ w_- & = & StepFunctionLowerOffset \\ w_+ & = & StepFunctionUpperOffset \\ V_- & = & StepFunctionLowerValue \\ V_+ & = & StepFunctionUpperValue \end{array} ,$$

if $StepFunctionLessThan$ is `True`, then

$$f(x,y,z) = \begin{cases} V_- & w < w_- \\ V_- + \left(V_+ - V_-\right)\left(\frac{w-w_-}{w_+-w_-}\right) & w_- < w < w_+ \\ V_+ & w_+ < w \end{cases} ,$$

otherwise it is reversed

$$f(x,y,z) = \begin{cases} V_+ & w < w_- \\ V_- + \left(V_+ - V_-\right)\left(\frac{w_+-w}{w_+-w_-}\right) & w_- < w < w_+ \\ V_- & w_+ < w \end{cases} ,$$

StepFunctionProduct1 Using a similar scheme as StepFunction to specify the dimension with $StepFunctionProduct1Dim$,

$$f(x,y,z) = \begin{cases} 0 & w < StepFunctionProduct1Start \\ StepFunctionProduct1Value & StepFunctionProduct1Start < w < StepFunctionProduct1End \\ 0 & w > StepFunctionProduct1End \end{cases}$$

StepFunctionProduct2

StepFunctionProduct3

StepFunctionProduct4 These are the same as StepFunctionProduct1 except that they use different variables (e.g., $StepFunctionProduct2Dim$ instead of $StepFunctionProduct1Dim$).

Gaussian

$$f(x,y,z) = (GaussianHeight)\exp\left[-\left(\frac{GaussianCenter - coord[GaussianDim]}{GaussianWidth}\right)^2\right]$$

TemperatureProfile

$$f(x,y,z) = \begin{cases} T_0 & y > maxY \\ T_0 + A\left(maxY - y\right) + B\left(1 - \exp\left(-C\left(maxY - y\right)\right)\right) & y < maxY, H_0 < 0 \\ \min\left(T_m, T_0 + \frac{(T_m - T_0)}{H}\left(maxY - y\right) + B\left(1 - \exp\left(-C\left(maxY - y\right)\right)\right)\right) & y < maxY, H_0 > 0 \end{cases},$$

where

$$H = \min\left(H_m, H_o + 2dH\left|x - x_c\right| / \left(maxX - minX\right)\right),$$

and

$$\begin{aligned} T_0 &\equiv TemperatureProfileTop, \\ A &\equiv TemperatureLinearCoefficient, \\ B &\equiv TemperatureExponentialCoefficient1, \\ C &\equiv TemperatureExponentialCoefficient2, \\ T_m &\equiv TemperatureProfileMax, \\ H_0 &\equiv TemperatureProfileH0, \\ H_m &\equiv TemperatureProfileHm, \\ dH &\equiv TemperatureProfiledH, \\ x_c &\equiv ExtensionCentreX. \end{aligned}$$

File1,...,File10 This reads `File1_N` elements from `File1_Name`. The format is two columns, with the first column being the coordinate along the direction `File1_Dim` and the second being the value. The coordinates must be sorted and increasing. Gale linearly interpolates between values as necessary. So a file with the two lines

```
0 10
100 20
```

will create a linear gradient between 0 and 100.

A.14 Verbosity Options

By default, Gale prints out very little when running. To get more information, insert

```
<param name="journal.info">True</param>
<param name="journal.debug">True</param>
<param name="journal-level.info">2</param>
<param name="journal-level.debug">2</param>
```

into the variables section (see Section A.1.4). This will print out more information than you need about the components, the solvers, and the number of iterations. In addition, you can get even more information about the solvers from PETSc by appending "`-ksp_monitor`" to the command line.

Appendix B

Output File Format

Gale outputs two types of files: VTK files for data analysis and visualization, and checkpointing files for restarting a run.

B.1 VTK Files: `.vts`, `.pvts`, `.vtu`, and `.pvtu` (Visualization)

These files are output by the `Underworld_VTKOutput` plugin. The `.vts` and `.pvts` files contain information about quantities on the grid, and the `.vtu` and `.pvtu` contain information about quantities on the particles. Every processor outputs its own `.vts` and `.vtu` file, and the `.pvts` and `.pvtu` are small files that have information on how to stitch them all together. Ordinarily, you would only open the `.pvts` and `.pvtu` files. These files are in a format understood by a wide variety of visualization programs, such as ParaView (`paraview.org`) (recommended) or MayaVi (`mayavi.sf.net`). VisIt (`www.llnl.gov/visit`) can understand the raw `.vts` and `.vtu` files, but not the `.pvtu` and `.pvts` files. So VisIt is mostly useful for visualizing serial runs.

The `Underworld_VTKOutput` plugin is activated by adding the lines

```
<struct>
  <param name="Type">Underworld_VTKOutput</param>
  <param name="Context">context</param>
</struct>
```

to the list of plugins. This line is already in the example input files.

The particle files can get quite large. Many times, you do not need to see every single particle. You can configure Gale to only output every Nth particle by setting `particleStepping` to N.

Defaults	
particleStepping	1

B.2 `Checkpoint` Files: .h5, .dat and .xmf

These files are mostly useful for checkpointing. They are in a machine dependent format and have the bare minimum needed to restart the run.

Appendix C

Benchmarks

Gale has been tested against a number of different benchmarks. Each benchmark tests different parts of the code, although there is some overlap. Specifically, Table C.1 summarizes which parts of the code are tested by which benchmark.

Code Functionality	Benchmark Section
Stokes solver and interpolate between particles and mesh in 2D	C.2,C.3, C.4 , C.5, C.6, C.7
Stokes solver and interpolate between particles and mesh in 3D	C.1, C.4
Time stepping	C.3, C.6, C.7
Gravity	C.1, C.3, C.6, C.7
Free surface	C.3, C.6, C.7
Drucker Prager rheology in 2D	C.5, C.6, C.7

Table C.1: Summary of which parts of the code are tested by which benchmarks

With the exception of the GeoMod benchmarks (Sections C.6 and C.7), the benchmarks can be carried out to high precision (~1%). In particular, the error should follow the relation

$$error \propto h + O(h^2),$$

where h is the size of the element. This means that if we plot the error from three different resolutions (high, medium and low) and scale it by h, we should see that the high-resolution error is closer to the medium-resolution error than the low-resolution error. In practice, this may be difficult to achieve because there are almost always other sources of error besides resolution.

Altogether, these benchmarks give us a high degree of confidence in the code.

C.1 Falling Sphere

This benchmark simulates a rigid sphere falling through a cylinder filled with a viscous medium as in Figure C.1.

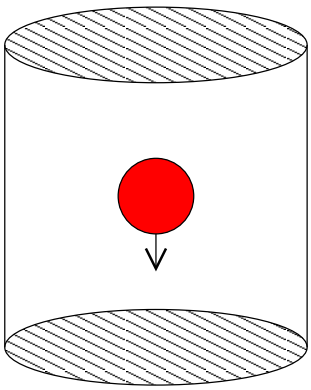


Figure C.1: Schematic of a Sphere falling through a Cylinder

The file

`input/benchmarks/falling_sphere/README`

has instructions on running this benchmark. In an infinitely large cylinder, the analytic solution for the drag on a sphere is

$$F = 6\pi\eta r u,$$

where η is the viscosity of the medium, r is the radius of the sphere, and u is the velocity of the sphere. Conversely, the buoyancy force is

$$F = \frac{4}{3}\pi r^3 g\delta\rho,$$

where g is the gravitational constant and $\delta\rho$ is the density difference between the sphere and the medium. Balancing these two forces and solving for the velocity gives

$$u = \frac{2}{9}r^2 g\delta\rho/\eta.$$

Setting $g = 1$, $r = 1$, $\delta\rho = 0.01$, and $\eta = 1$ gives a velocity of

$$u = 0.00222.$$

In our case, we simulate a rigid sphere with a high viscosity sphere. This allows some internal circulation within the sphere, and so the expression for the velocity becomes [9]

$$u = \frac{1}{3}\frac{r^2 g\delta\rho}{\eta}\frac{\eta+\eta'}{\eta+\frac{3}{2}\eta'},$$

where η' is the viscosity of the sphere. For our case, the background medium's viscosity is 1 and the sphere's viscosity is 100, so the correction is about 1%.

When the boundaries are not infinitely far away, we can expand the solution in terms of the ratio of the radius of the sphere (r) to the radius of the cylinder (R). One solution by Habermann [12] gives a drag force of

$$F_H = 6\pi\eta r u\frac{1-0.75857\cdot\left(\frac{r}{R}\right)^5}{1+f_H\left(\frac{r}{R}\right)},$$

where

$$f_H\left(\frac{r}{R}\right) = -2.1050(r/R) + 2.0865(r/R)^3 - 1.7068(r/R)^5 + 0.72603(r/R)^6.$$

For our case with $r = 1$, $R = 4$, this gives a velocity of

$$u = 1.122747319 \cdot 10^{-3}.$$

The walls reduce the speed by about a factor of two.

Another solution by Faxen [12] gives a drag force of

$$F_F = 6\pi\eta r u / \left(1 + f_F(r/R)\right),$$

where

$$\begin{aligned} f_f(r/R) &= -2.10444(r/R) + 2.08877(r/R)^3 - 0.94813(r/R)^5 \\ &\quad -1.372(r/R)^6 + 3.87(r/R)^8 - 4.19(r/R)^{10}. \end{aligned}$$

For our case, this gives a speed of

$$u = 1.12293603939 \cdot 10^{-3},$$

which agrees closely with the result from Habermann.

Another possible artifact is that we do not simulate an infinitely long cylinder. This turns out to be a small effect. We use a cylinder with a height of 8, and place the sphere halfway down. We did runs where the cylinder was twice as tall, and the results were essentially unchanged.

Since we do not precisely track the surface of the sphere, there is some ambiguity as to where the sphere ends and the background medium begins. The sphere viscosity is 100 and the background viscosity is 1, so we decide, somewhat arbitrarily, to define the sphere to be those cells where the viscosity is greater than 99. Also, because we are simulating a high viscosity sphere rather than a completely rigid sphere, the velocity inside the sphere is not uniform. The error bars indicate the variation in velocity across the sphere.

The errors in the computed velocity compared to the Faxen solution are plotted in Figure C.2. These were done with resolutions of 8×16×8, 16×32×16, 32×64×32, and 64×128×64, corresponding to grid sizes (h) of 0.5, 0.25, 0.125, and 0.0625. Because of the symmetries of the problem we only have to simulate a quarter of the domain. As h decreases, the error decreases. It does not decrease linearly with h, suggesting that some other factor is contributing to the error (e.g. the finite viscosity of the sphere).

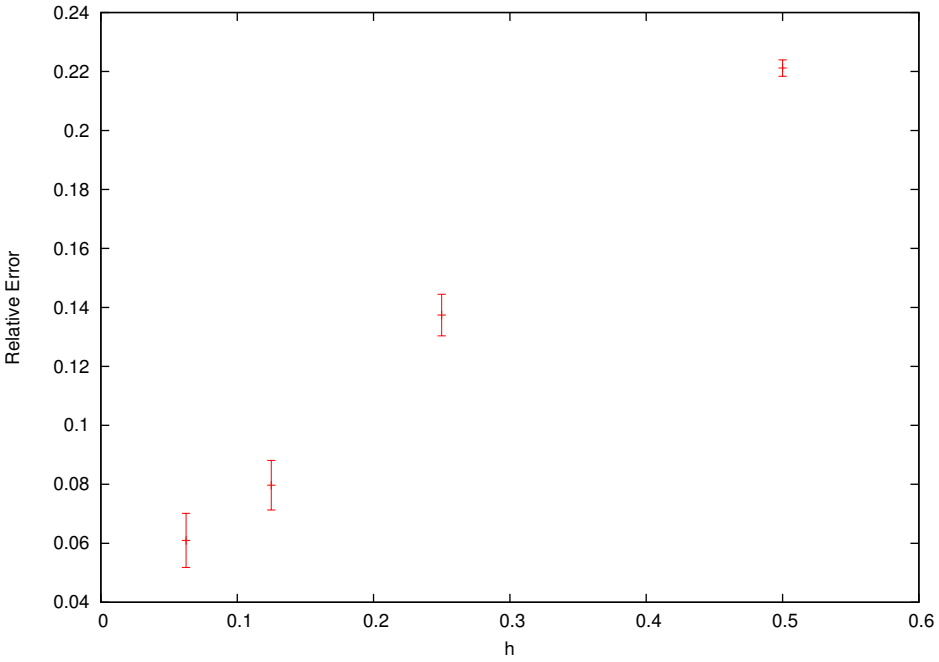


Figure C.2: Relative Error in computed velocity vs. resolution

C.2 Circular Inclusion

Schmid and Podladchikov [8] derived a simple analytic solution for the pressure and velocity fields for a circular inclusion under simple shear as in Figure C.3.

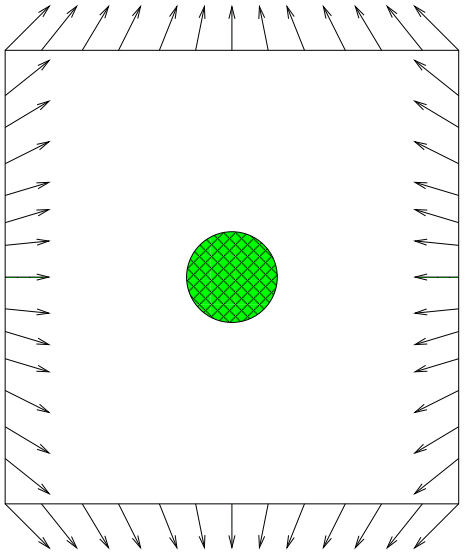


Figure C.3: Schematic for the circular inclusion benchmark

The file `input/benchmarks/circular_inclusion/README` has instructions on how to run this benchmark.

Because of the symmetry of the problem, we only have to solve over the top right quarter of the domain. For the velocity boundary conditions, the analytic solution is a bit complicated. So we used the simple relation

$$\begin{aligned} v_x &= -\dot{\epsilon} y, \\ v_y &= \dot{\epsilon} x, \end{aligned}$$

for the boundaries, where $\dot{\epsilon} = 1$ is the magnitude of the shear and x and y are the coordinates. This induces an error of order r_i^2/r^2, where $r_i = 0.1$ is the radius of the inclusion, and r is the radius. We have the boundaries at 80 times the radius of the inclusion, giving an error of about 0.01%, which is much smaller than the other errors we are looking at. Just to make sure, we did runs with the boundaries at 40 times the radius of the inclusion and got very similar results.

A characteristic of the analytic solution is that the pressure is zero inside the inclusion, while outside it follows the relation

$$p_m = 4\dot{\epsilon} \frac{\mu_m \left(\mu_i - \mu_m\right)}{\mu_i + \mu_m} \frac{r_i^2}{r^2} \cos\left(2\theta\right),$$

where $\mu_i = 2$ is the viscosity of the inclusion and $\mu_m = 1$ is the viscosity of the background media. Many numerical codes that solve Stokes flow (Eq. 2.2 and 2.3), including Gale, assume that pressure, velocity, and viscosity are continuous. The pressure discontinuity at the surface of the inclusion violates that assumption, so the error tends to concentrate near the surface of the inclusion.

Figure C.4 plots the error in the pressure along the line $y = x/2$ for different resolutions. Inside the inclusion near the surface, the pressure is consistently wrong. The pressure does not converge with higher resolution, giving us a clue that the numerical scheme is not completely accurate.

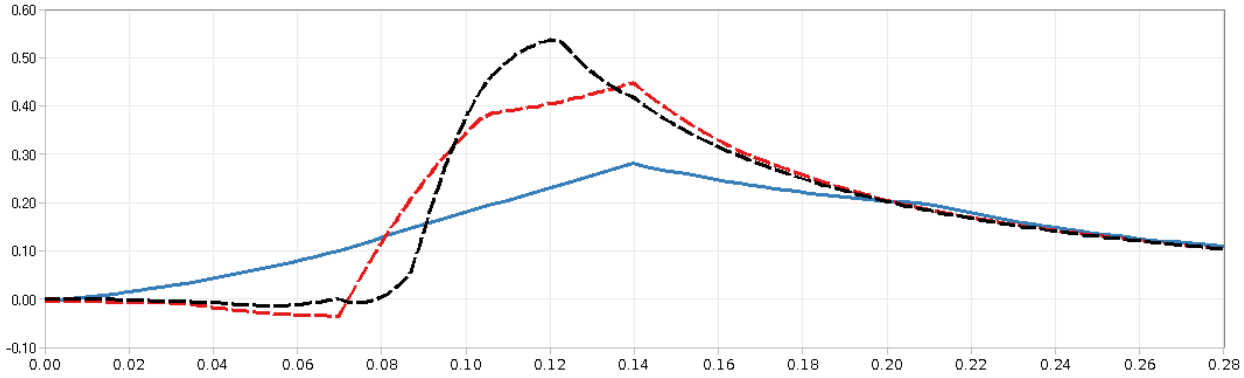


Figure C.4: Pressure along the line $y = x/2$ for resolutions of 128×128 (blue), 256×256 (red), and 512×512 (black). The inclusion has a radius $r_i = 0.1$. Note that the pressure should be zero inside the inclusion, but the numerical solutions consistently underestimate the pressure.

Outside the inclusion, the error is better behaved. Figure C.5 plots the error in the pressure along the line $y = x/2$ outside the inclusion for different resolutions. While there are still problems near the surface, away from the surface the solutions are quite good. Figure C.6 zooms in on the error farther out, and we can see that the error scales continues to reduce with resolution. This gives us confidence that, at least away from the inclusion, the code is giving the right answer. This kind of result, where the solution is bad close to the surface, but good otherwise, is typical for numerical solutions of this problem [13].

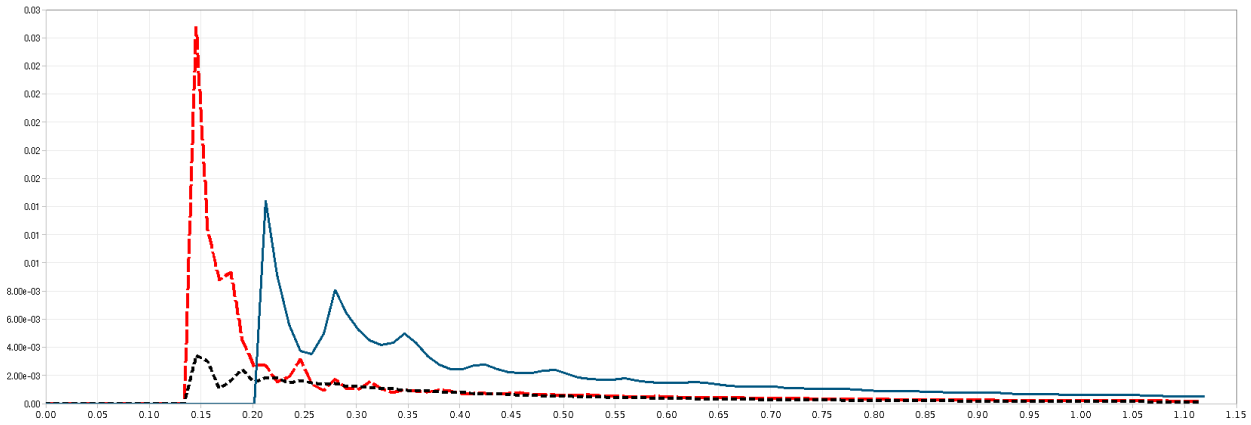


Figure C.5: Error in the pressure outside the inclusion along the line $y = x/2$ for resolutions of 128×128 (blue), 256×256 (red), and 512×512 (black). The inclusion has a radius $r_i = 0.1$.

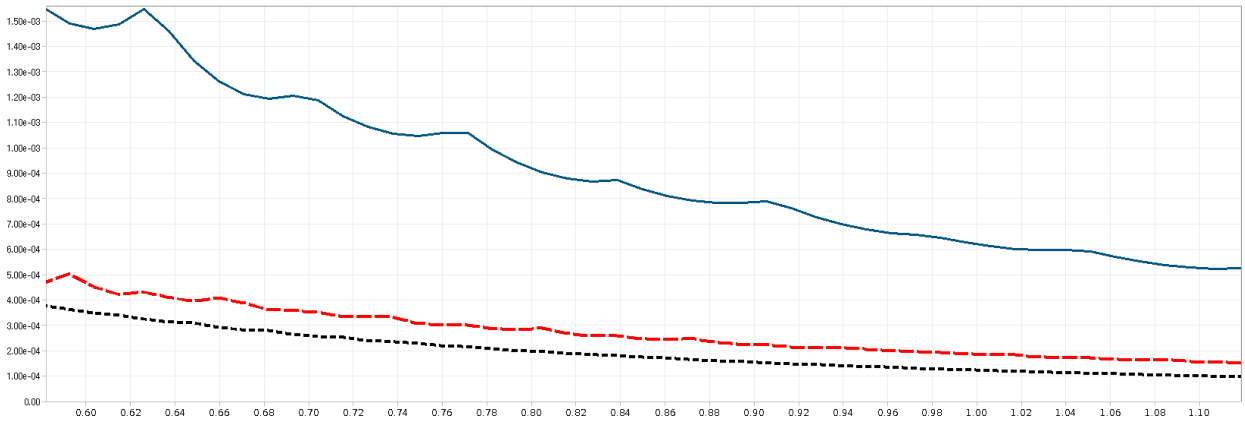


Figure C.6: As in Figure C.5, but zoomed in on a part a little away from the inclusion.

C.3 Relaxation of Topography

Given an infinitely deep purely viscous medium with an infinitesimal initial sinusoidal height profile, the topography will decay exponentially with the timescale [10]

$$t_r = \frac{4\pi\eta}{gL},$$

where η is the viscosity, g is the gravitational constant, and L is the wavelength of the initial sinusoid.

In our case, we simulate a medium with non-infinite depth (depth=L) and a sinusoid with a non-zero amplitude ($A = 0.01$). The internal fields decay exponentially with depth with a length scale of $L/2\pi$, giving an error of 0.2%. A non-zero amplitude creates errors of order $(2\pi A/L)^2$, which in this case is 0.4%.

The file `input/benchmarks/sinusoid/README` explains how to run this benchmark. Figure C.7 shows the results of a high-resolution run. This run is relatively large (512×1024), because we need fairly high resolution to be able to accurately resolve the small (1%) height difference. Also note that we use symmetry to only simulate half of the wavelength.

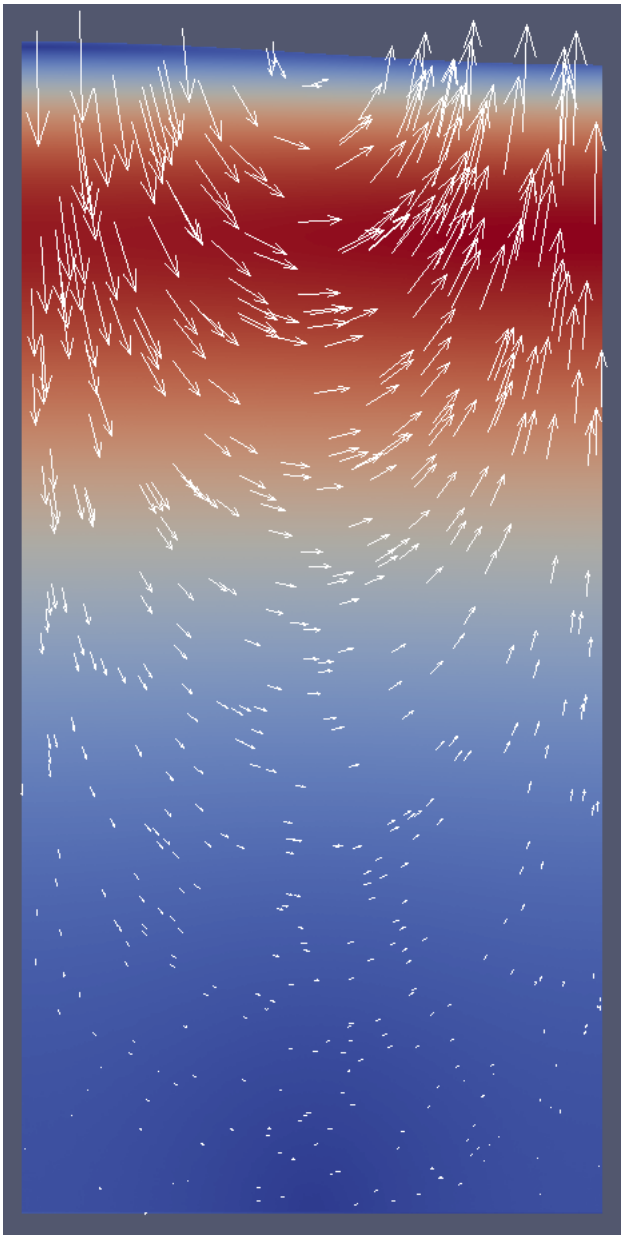


Figure C.7: Strain rate and velocities for a sinusoidal topography relaxing under gravity

Running the code with multiple resolutions and measuring the error in the height in the trough gives Figure C.8. Scaling the error with resolution gives Figure C.9. The error decreases linearly with increasing resolution, giving us confidence in our ability to accurately track topography.

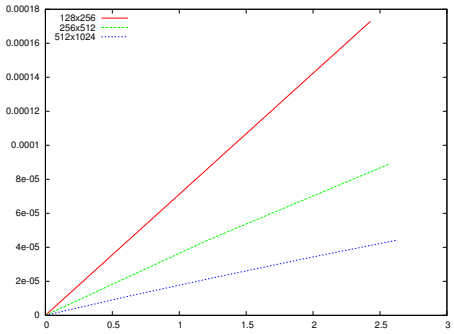


Figure C.8: Error in the height at the trough

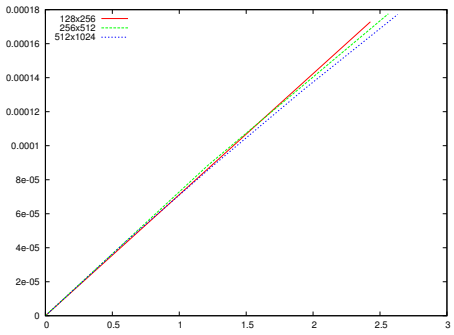


Figure C.9: As in Figure C.8, but with the error scaled with h. So the medium-resolution error is multiplied by 2 and the high-resolution error is multiplied by 4.

C.4 Divergence

This benchmark tests the implementation of the divergence term in equation 2.8. In 2D, a constant divergence is applied to a square domain, and the velocity on the corners is set to enforce a spreading from the center of the square. Figure C.10 shows the velocity and strain rate invariant for a numerical solution. For a constant divergence d, the analytic solution for this setup is

$$\begin{aligned} v_x &= x \cdot d/2 \\ v_y &= y \cdot d/2 \end{aligned} .$$

In 3D, the analytic solution is

$$\begin{aligned} v_x &= x \cdot d/3 \\ v_y &= y \cdot d/3 \\ v_z &= z \cdot d/3 \end{aligned} .$$

In both cases, the strain rate invariant equals $\sqrt{d/2}$. As shown in Figure C.11, the main source of error in 2D comes from inaccuracies in the solver. Figure C.12 paints a different picture in 3D, where the main source of error comes from having a finite number of particles.

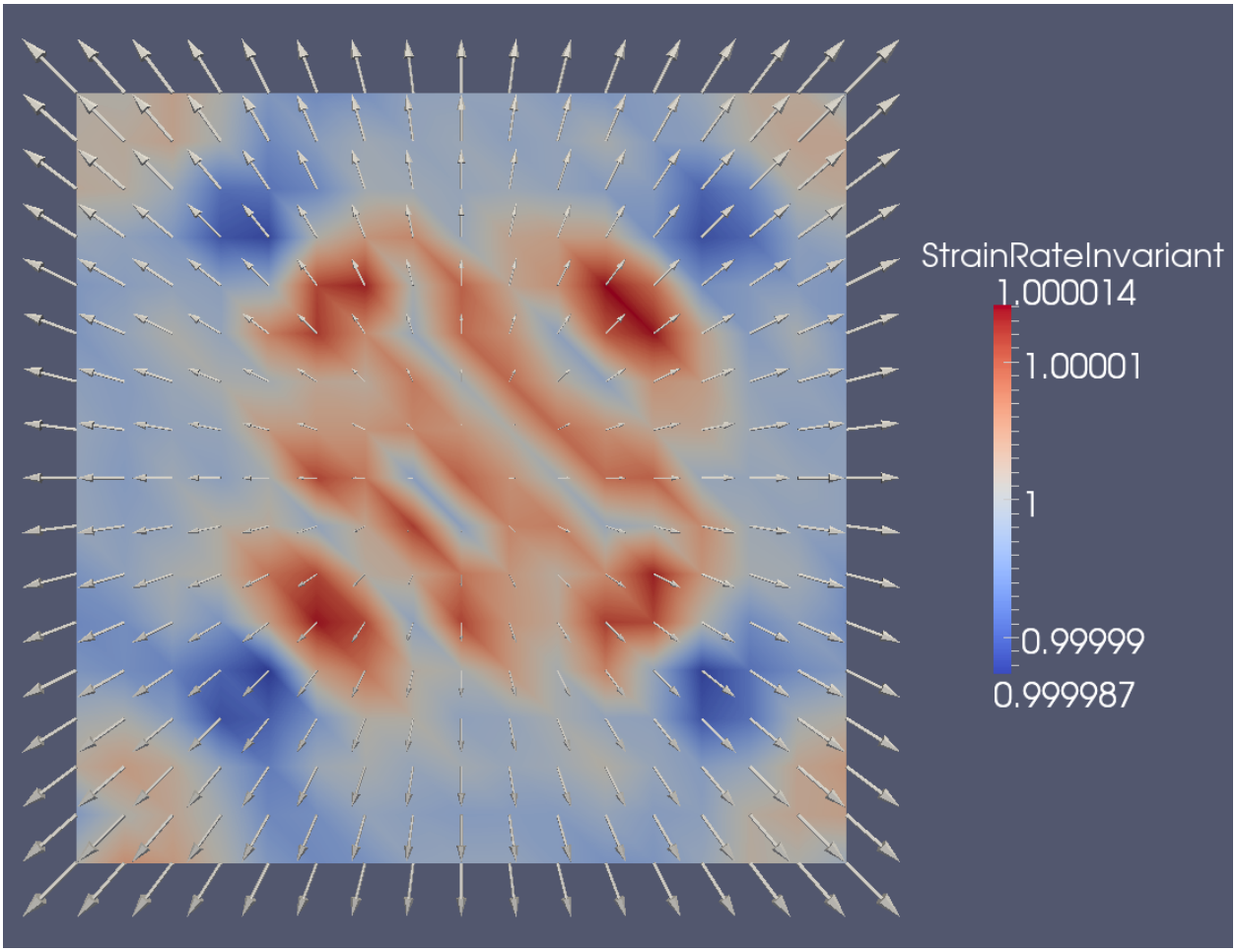


Figure C.10: Velocity and Strain Rate Invariant solution for the 2D Divergence benchmark. The variation in the strain rate invariant is uniformly small.

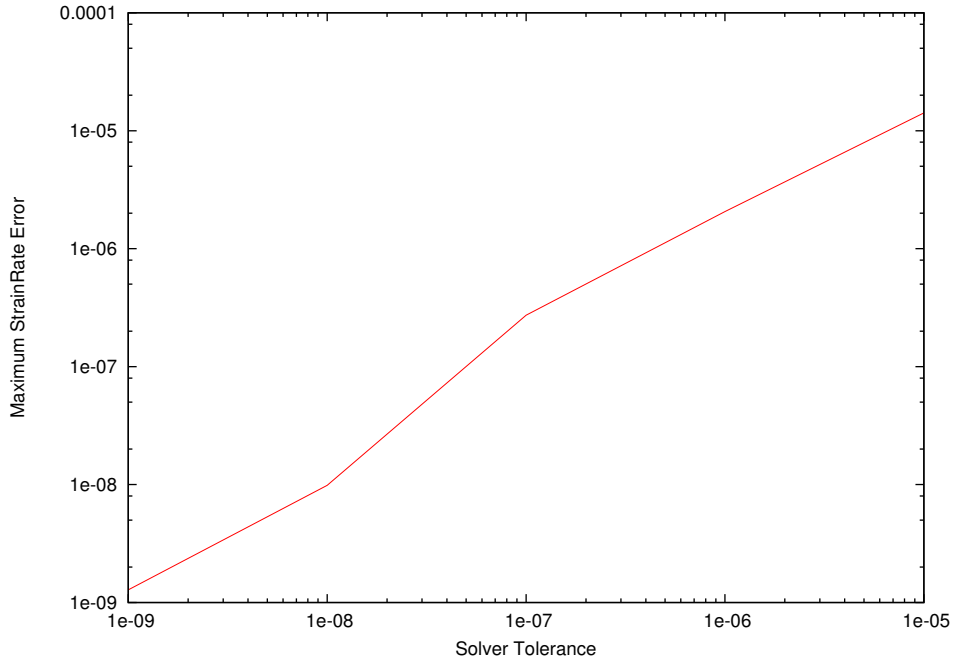


Figure C.11: Maximum error in the strain rate invariant for the 2D Divergence benchmark vs tolerance in the linear solver. The resolution is kept at 32×32, and the number of particles per cell is kept at 30.

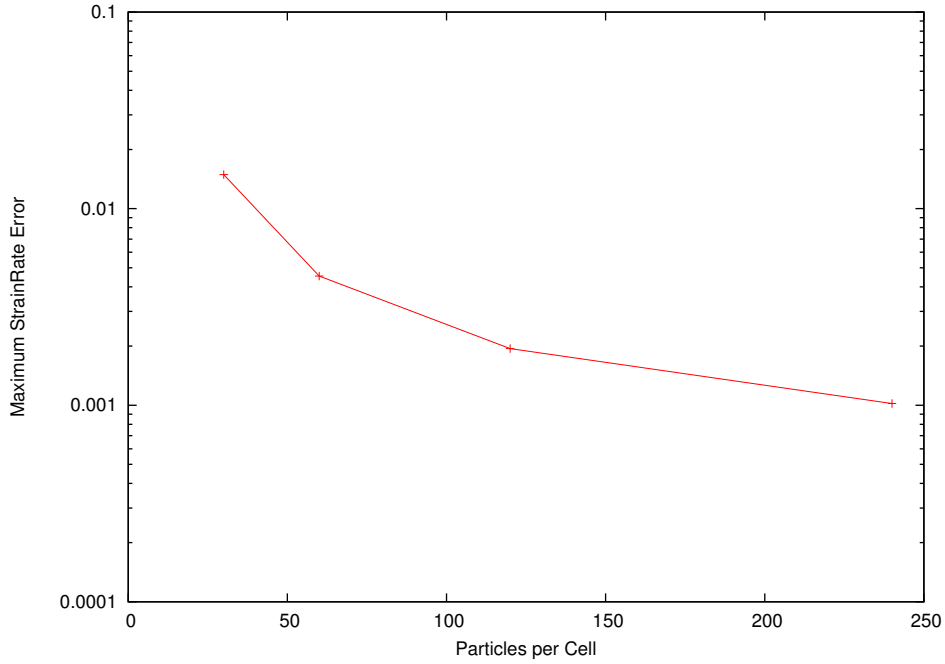


Figure C.12: Maximum error in the strain rate invariant for the 3D Divergence benchmark vs the number of particles in each cell. The resolution is kept at 16×16×16, and the tolerance in the linear solver is kept at 10^{-7}.

C.5 Drucker-Prager

C.5.1 Analytic Treatment

For the Drucker-Prager rheology in 2D, we can write the yielding relation as

$$\sigma_{ns} = \sigma_{nn} \tan \varphi + C,$$

where σ_{ns} is the shear stress perpendicular to the fault plane, σ_{nn} is the shear stress parallel to the fault plane, ϕ is the internal angle of friction, and C is the cohesion. Decomposing this into principal stresses σ_I, σ_{II}, and σ_{III} gives

$$\sin(2\Theta)(\sigma_I - \sigma_{III})/2 = \tan\varphi((\sigma_I + \sigma_{III})/2 + \cos(2\Theta)(\sigma_I - \sigma_{III})/2) + C,$$

where Θ is the angle that the fault makes relative to the maximum shear stress. Assuming that the fault forms where the shear stress $\sigma_I - \sigma_{III}$ is a minimum, a little algebra gives us

$$\Theta = \pm\left(\frac{\pi}{4} + \frac{\varphi}{2}\right).$$

Using this, we can construct a simple plasticity experiment and make sure that Gale gives the correct faulting angle.

C.5.2 Model Setup

We performed a shortening experiment as shown in Figure C.13. We only solve the Stokes equation and look at the strain rate invariant to find incipient faults. We do not take any time steps, removing any confounding effects they may cause. We made the weak region have a resolvable size, because faults arising from unresolved weak regions always tend to be at 45° [18].

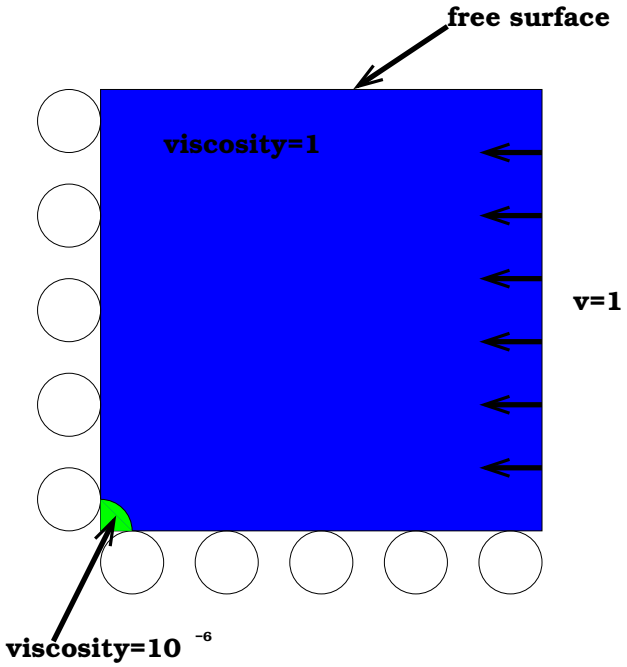


Figure C.13: The setup for the shortening experiment. The box is 1 unit on a side, and the low viscosity region has a radius of 0.01 (its size is exaggerated).

C.5.3 Numerical Results

Figure C.14 shows the results for two different resolutions for $\varphi = 45$. The uncertainties in measuring the fault angle are far larger than the differences between the two resolutions.

On the other hand, Figure C.15 shows results for when we vary the maximum strain rate. If the maximum strain rate is too low, then the fault angles are no longer correct. Setting the maximum strain rate to a more moderate value (10), on the other hand, does not seem to affect the fault angle for this case. However, if we look at a different friction coefficient in Figure C.16, then the fault angles are wrong.

Similarly, Figure C.17 shows results for when you vary the minimum viscosity. With a very high minimum viscosity, faults do not develop. With a little lower setting, faults do develop, but not at the correct angle. With a still lower setting, the angles become correct, but the structure still differs markedly from the result with unconstrained minimum viscosity. It is only with the lowest setting that the structure converges to the unconstrained answer.

Figure C.18 shows a plot of the numerical vs. analytic results for several different angles. This gives us confidence that, at least in compression in 2D, our Drucker-Prager implementation gives the correct results.

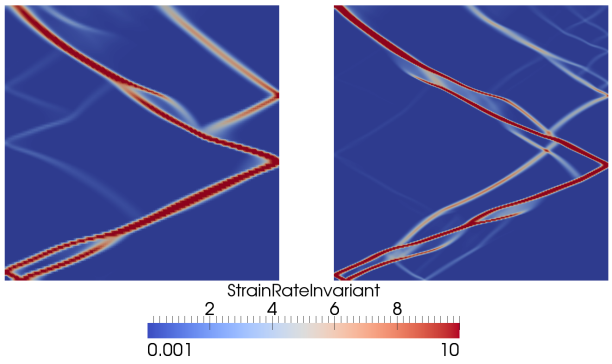


Figure C.14: Strain rate invariant for the yielding experiment with $\varphi = 45$ with two different resolutions: 128×128 and 256×256. Any differences in the fault angle between the two resolutions are swamped by uncertainties in determining the overall angle of faulting.

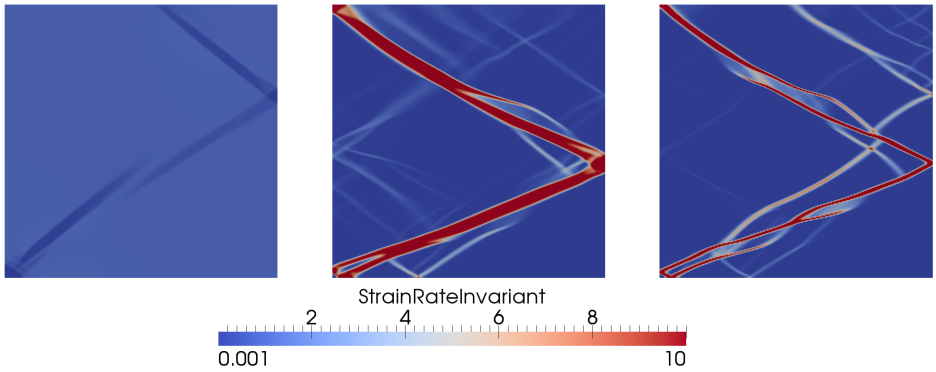


Figure C.15: Strain rate invariant for the yielding experiment with three different `maxStrainRate`'s: 1, 10, ∞. The resolution for all three cases is 256×256. The highest strain rate observed in the case for an infinite `maxStrainRate` is 74.

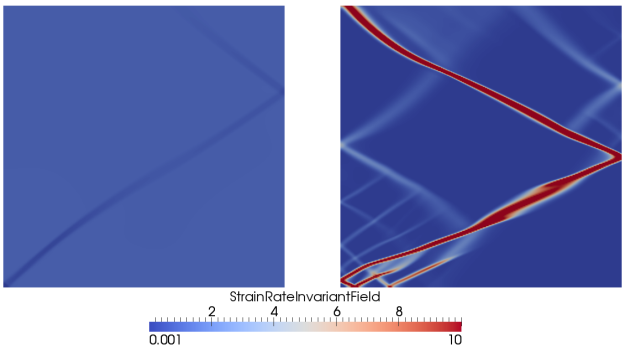


Figure C.16: Strain rate invariant for the yielding experiment with `maxStrainRate`'s of 10 and ∞ but with a friction angle $\varphi = 37°$. The resolution for both cases is 256×256.

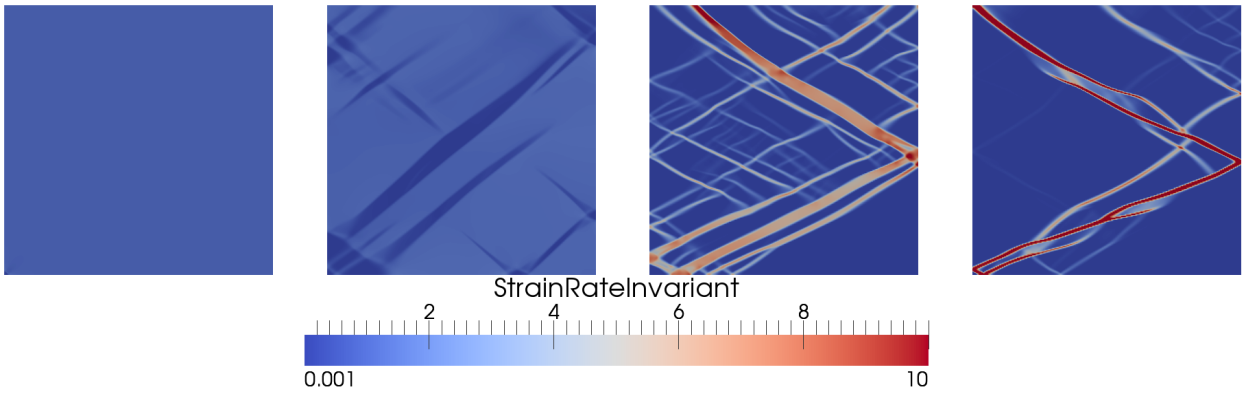


Figure C.17: Strain rate invariant for the yielding experiment with four different `minimumViscosity`'s:10^{-5}, 10^{-6}, 10^{-7}, 10^{-8}. The resolution for all three cases is 256×256.

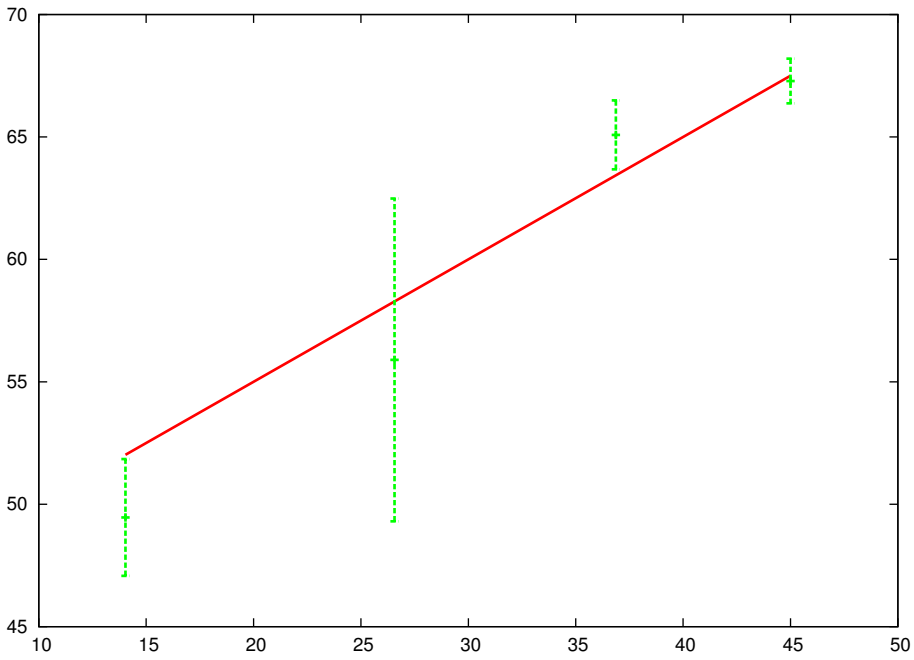


Figure C.18: Numerical vs analytic results for fault angles as a function of internal angle of friction.

C.6 Geomod 2004

Two benchmarks were created to validate numerical codes against analog sandbox experiments [11]: one benchmark simulates extension, and the other simulates shortening. A number of investigators with different codes ran these benchmarks, giving us a good standard against which to compare.

C.6.1 Extension

This benchmark simulates a sandbox being extended as in Figure C.19. The right side and half of the bottom are translated to the right. This creates a velocity discontinuity at the center which is the initial seed for localization. Gale's implementation of this benchmark is in `input/benchmarks/extension.xml`.

Like half of the codes in the benchmark, boundary friction was not included. Rather, the material is held fixed to the bottom boundary, and the velocity discontinuity is smoothed over 0.2 cm. In addition, the exact background viscosity is not prescribed by the benchmark. We have used $10^{12} Pa \cdot s$, the same as used in the I2ELVIS calculations. This value is somewhere in the middle of the range of values used in the calculations for other codes.

Figure C.20 shows the results for different values of `minimumViscosity` and `maxStrainRate` (see Section 4.2.1). Runs with a minimum viscosity of 10^4 and a maximum strain rate of $5 \cdot 10^{-4}$ do not lose any significant details while improving solver performance. Figure C.21 shows the results for different resolutions. The code is still not quite convergent, probably because we chose `maxStrainRate` such that it did not lose any details on the 512×128 grid.

A comparison against the other codes is in Figure C.22. While it is difficult to perform exact comparisons, Gale produces similar fault patterns.

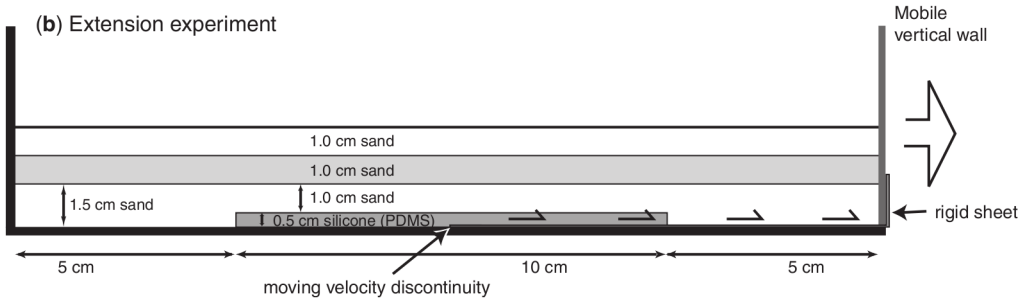


Figure C.19: Extension model setup. Reproduced, with permission, from Buiter et al. [11].

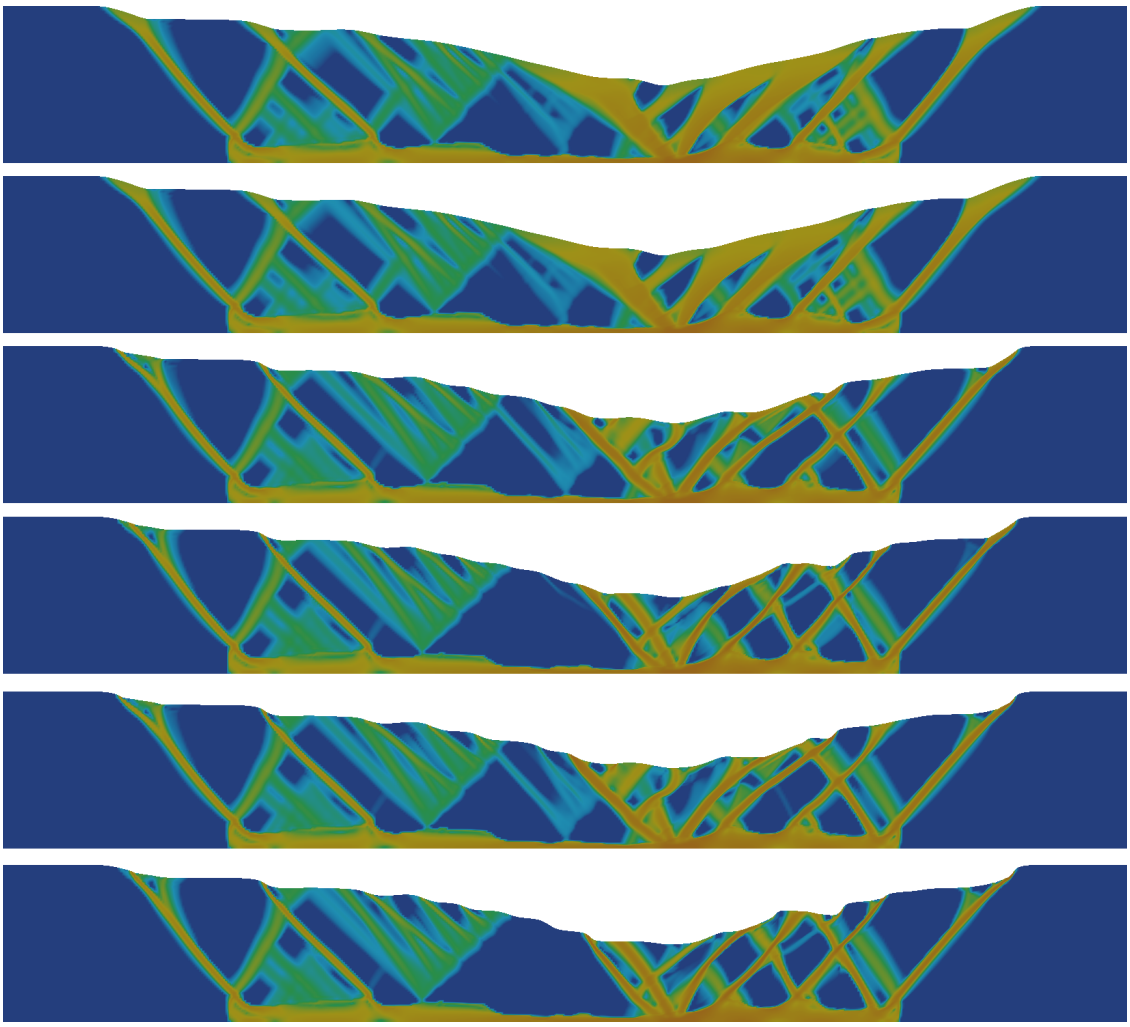


Figure C.20: Strain rate invariant for the extension model for varying η_{min} and $\dot{\epsilon}_{max}$. From top to bottom, they are: $\eta_{min} = 10^5$, $\dot{\epsilon}_{max} = 5 \cdot 10^{-4}$; $\eta_{min} = 10^5$, $\dot{\epsilon}_{max} = 2 \cdot 10^{-3}$; $\eta_{min} = 10^4$, $\dot{\epsilon}_{max} = 5 \cdot 10^{-4}$; $\eta_{min} = 10^4$, $\dot{\epsilon}_{max} = 2 \cdot 10^{-3}$; $\eta_{min} = 10^3$, $\dot{\epsilon}_{max} = 5 \cdot 10^{-4}$; $\eta_{min} = 10^3$, $\dot{\epsilon}_{max} = 2 \cdot 10^{-3}$;

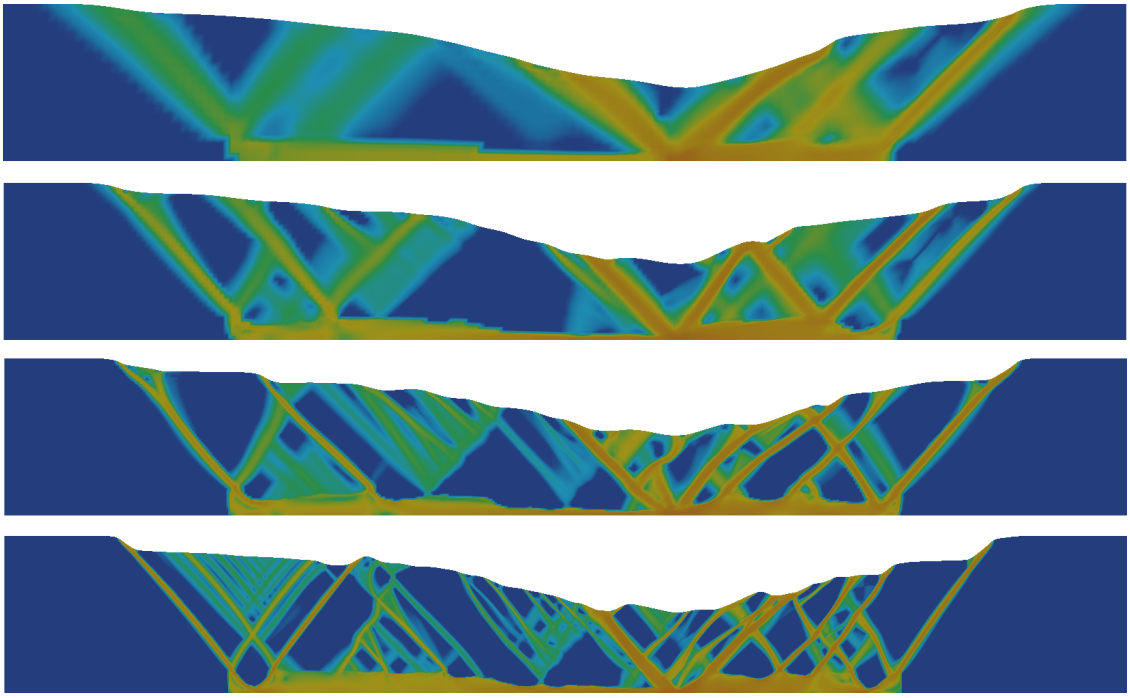


Figure C.21: Strain rate invariant for the extension model after 5 cm of extension for four different resolutions: 128x16, 256x32, 512x64, and 1024x128.

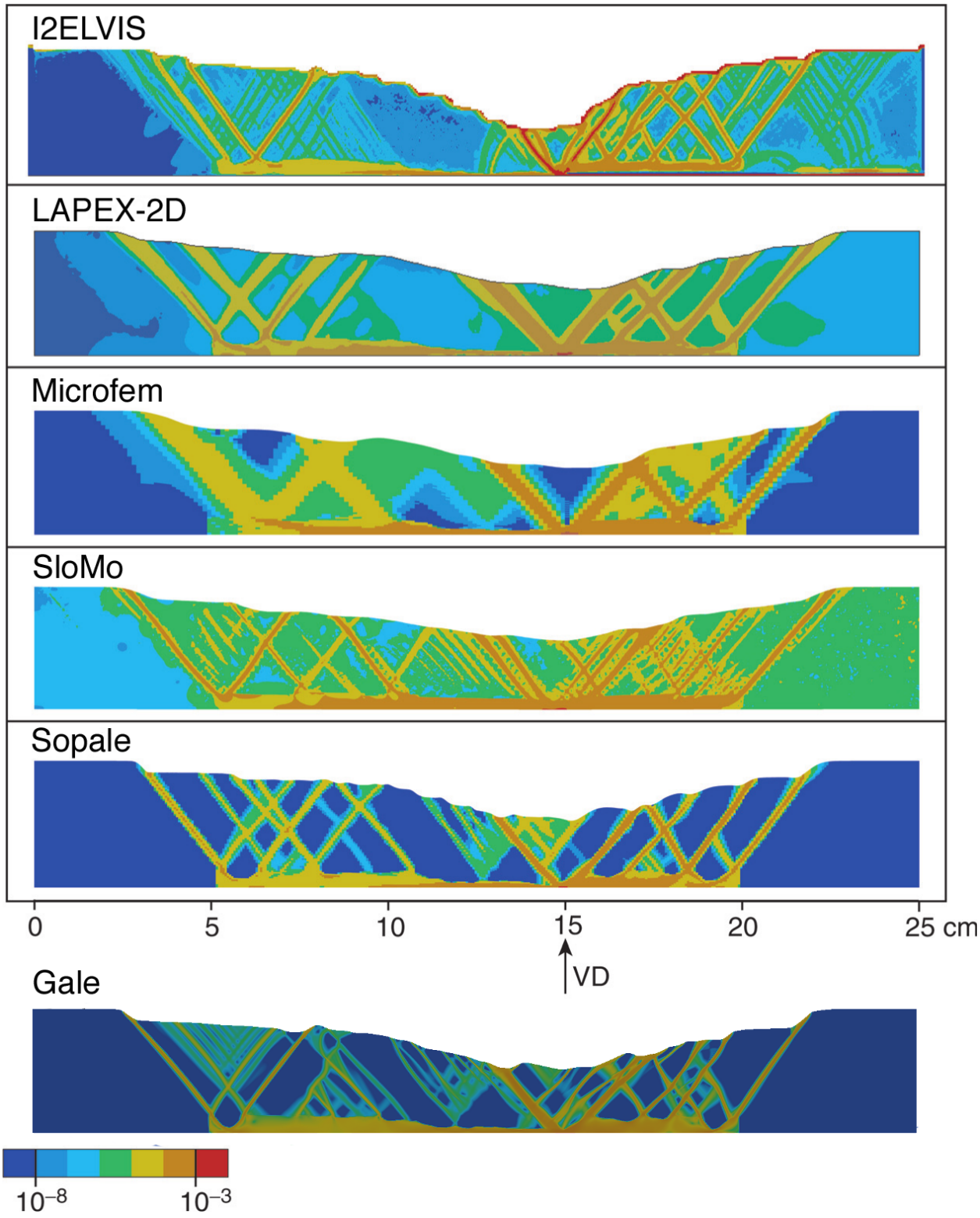


Figure C.22: Strain rate invariant for the numerical extension models after 5 cm of extension. The resolutions of the various models are: I2ELVIS: 400×75, LAPEX-2D: 301×71, Microfem: 201×61, SloMo: 401×71, Sopale: 401×71, Gale: 1024×128. Upper images reproduced, with permission, from Buiter et al. [11].

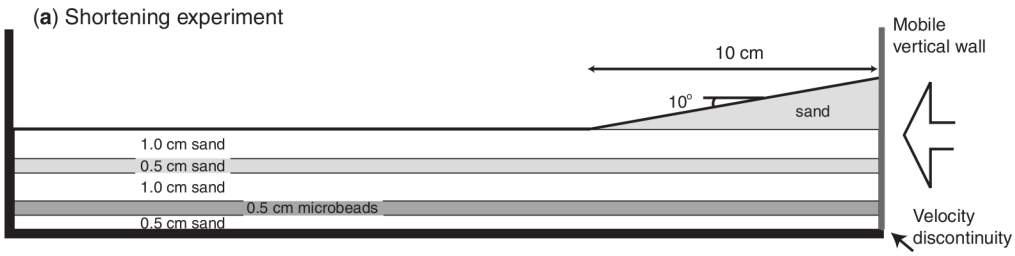


Figure C.23: Shortening model setup. Reproduced, with permission, from Buiter et al. [11].

C.6.2 Shortening

This benchmark simulates a sandbox being shortened as in Figure C.23. The right side is moved to the left, creating a velocity discontinuity at the bottom right corner. Gale's implementation of this benchmark is in `input/benchmarks/shortening.xml`.

As with the extension benchmark, we used values of 10^4 for `minimumViscosity` and $5{\cdot}10^{-4}$ for `maxStrainRate`. Unlike some of the other codes in the benchmark, we did not apply diffusion to the top surface. Without diffusion, steep slope angles develop, leading to landslides. Landslides occur over a very short time and length scale, posing particular difficulties for solvers. The `minimumViscosity` setting moderated these landslides, so explicit diffusion was not needed.

A comparison against the other codes' calculations at 14 cm of cumulative shortening is in Figure C.24. There is more variance among the different codes, so it is difficult to tell whether Gale's behavior agrees with the other codes. Figure C.25 shows a run with a few different resolutions, and even there we see marked differences in behavior as we increase resolution.

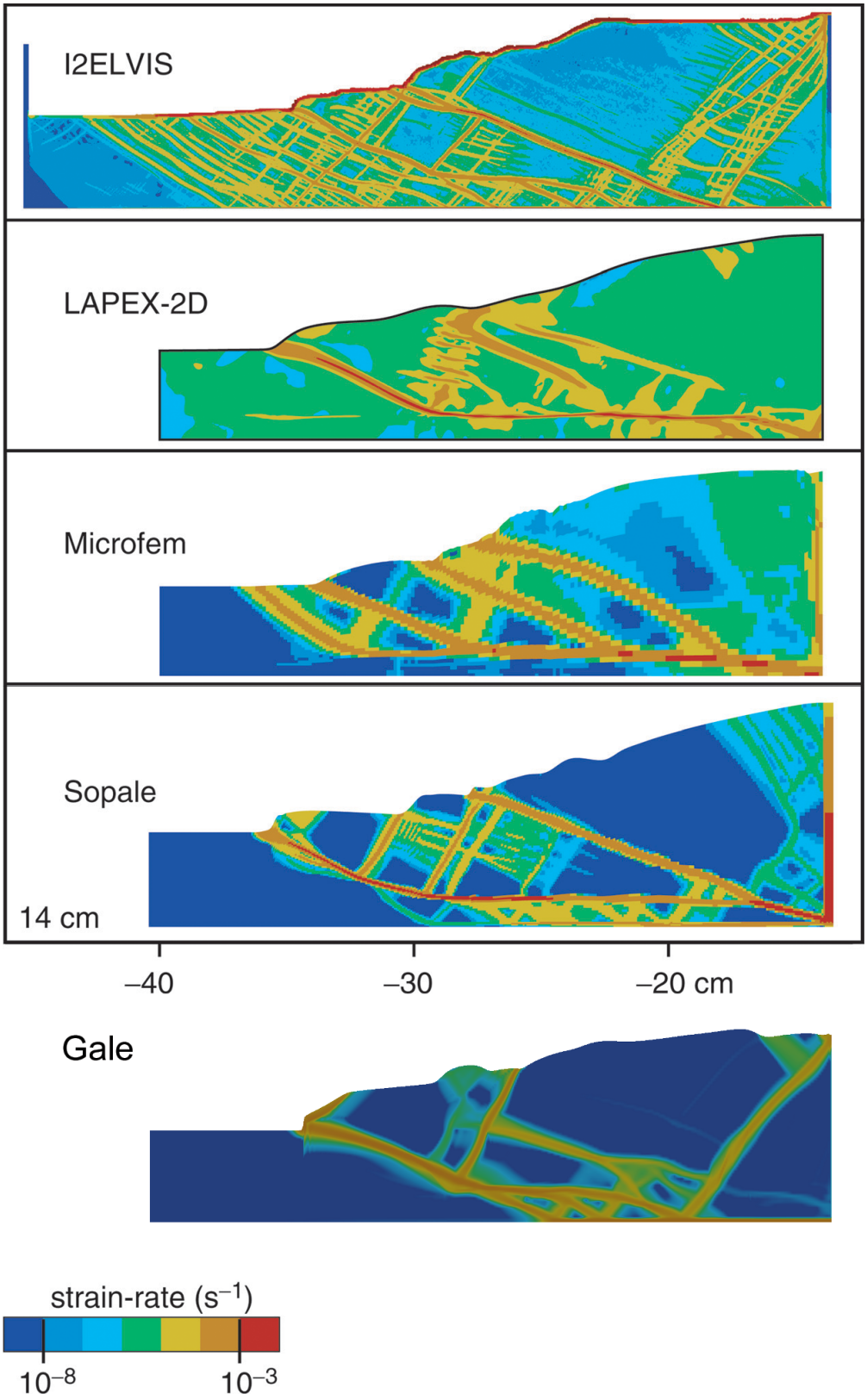


Figure C.24: Strain rate invariant for the numerical shortening models after 14 cm of shortening. The resolutions of the various models are: I2ELVIS: 900×75, LAPEX-2D: 351×71, Microfem: 201×36, Sopale: 401×71, Gale: 512×128. The upper portion of the figure is reproduced, with permission, from Buiter et al. [11].

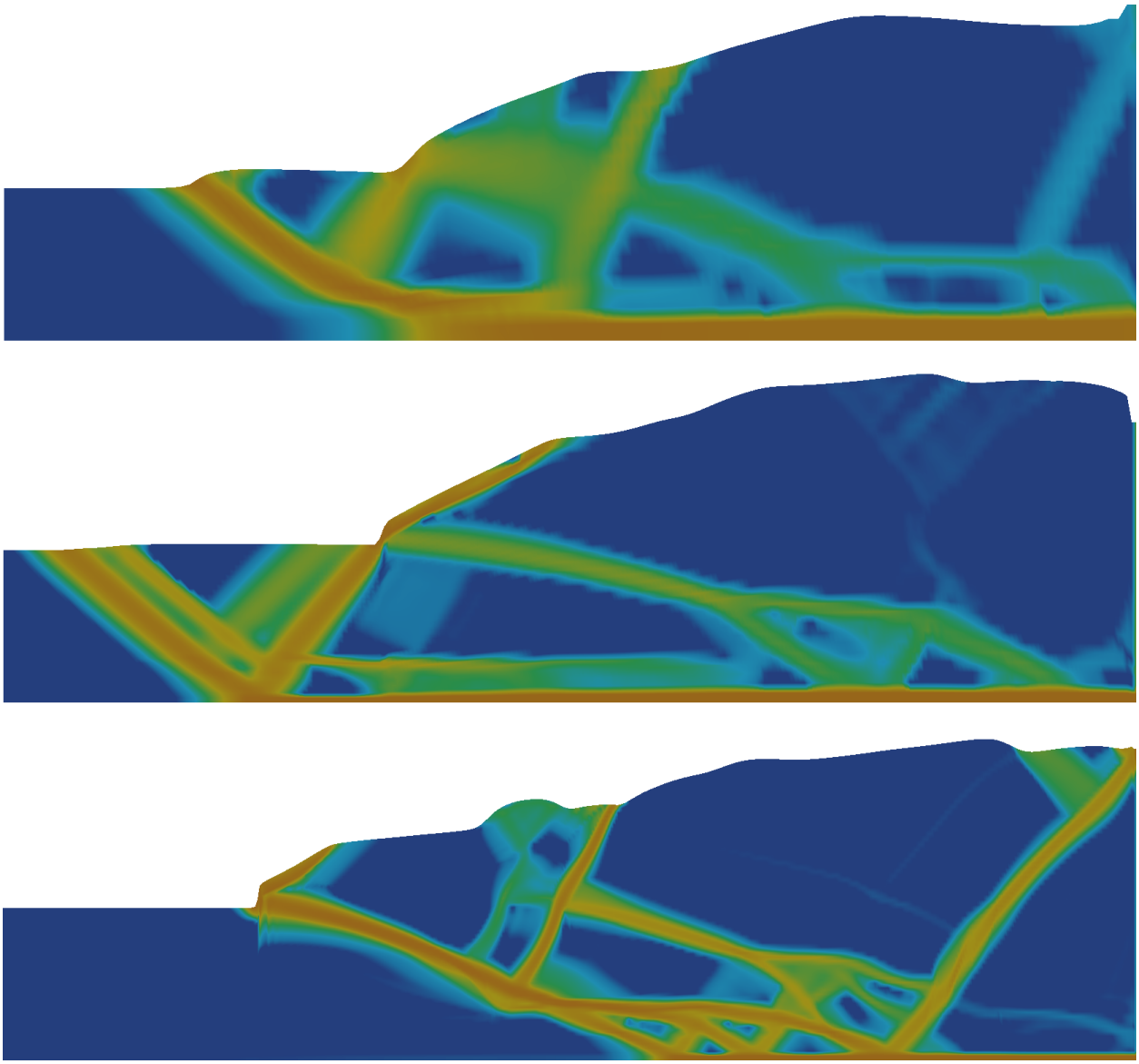


Figure C.25: Strain rate invariant for the shortening model after 14 cm of shortening for three different resolutions: (a) 128×32, (b) 256×64, and (c) 512×128.

C.7 Geomod 2008

Using the lessons learned from the Geomod 2004 benchmarks, new benchmarks were created that would make it easier to compare numerical experiments with each other and with analog experiments [16].

C.7.1 Stable Wedge

This benchmark simulates a wall pushing a wedge as in Figure C.26. There is an analytic solution [17] which details what the friction on the bottom and sides should be to provide enough resistance so that the wedge does not collapse under its own weight, but not so much as to cause any internal deformation as it slides. The derivation of the solution assumes that the friction along the sides has no cohesion. So the force at the tip will go to zero as the thickness of the material goes to zero. However, analog experiments suggest a finite cohesion, so this benchmark specifies a boundary cohesion.

We modeled the wedge using a relatively low viscosity ($1Pa \cdot s$) air layer on top. This low viscosity region does not, for the most part, affect the dynamics. We did not set `minimumViscosity` or `maxStrainRate`.

We modeled boundary friction by first fixing the sand to the boundary. We then modify the material properties in the element next to the boundary so that it provides the correct resistance. So in the bulk, the sand's internal angle of friction is 36 weakening to 31, while in the element at the boundary the internal angle of friction is 16 weakening to 14. Similarly, in the bulk, the cohesion is $10\,Pa$, while at the boundary the cohesion is $10\,Pa$ weakening to $0.01\,Pa$. If we do not weaken the cohesion, when we try to model an unstable wedge by reducing the internal angle of friction, the wedge never collapses on itself.

Figure C.27 shows the strain rate invariant after the wall has moved 4 cm, and Figure C.28 shows the particles. The bulk translates with almost no deformation, although, as expected, the tip deforms. The odd structures at the tip are below the grid resolution.

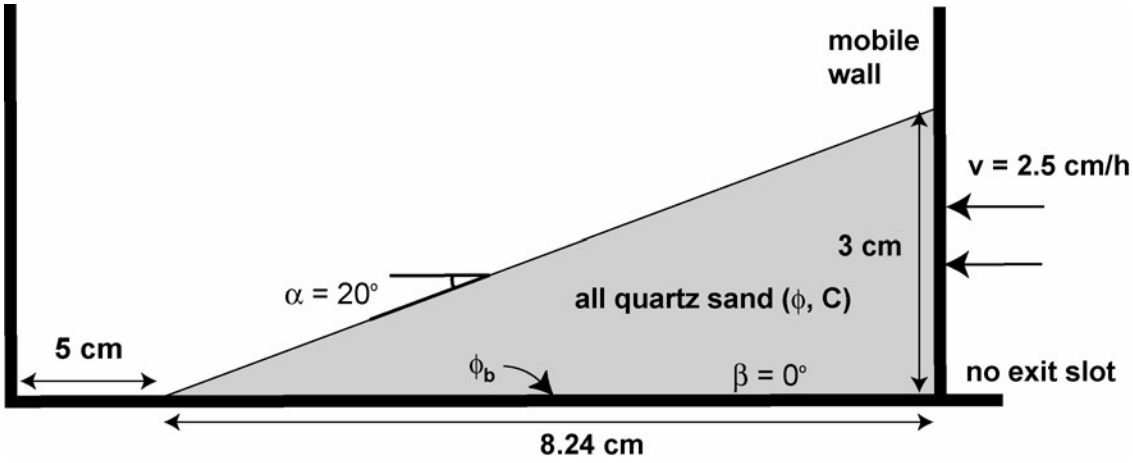


Figure C.26: Set up for the stable wedge benchmark. Image courtesy of Susanne Buiter.

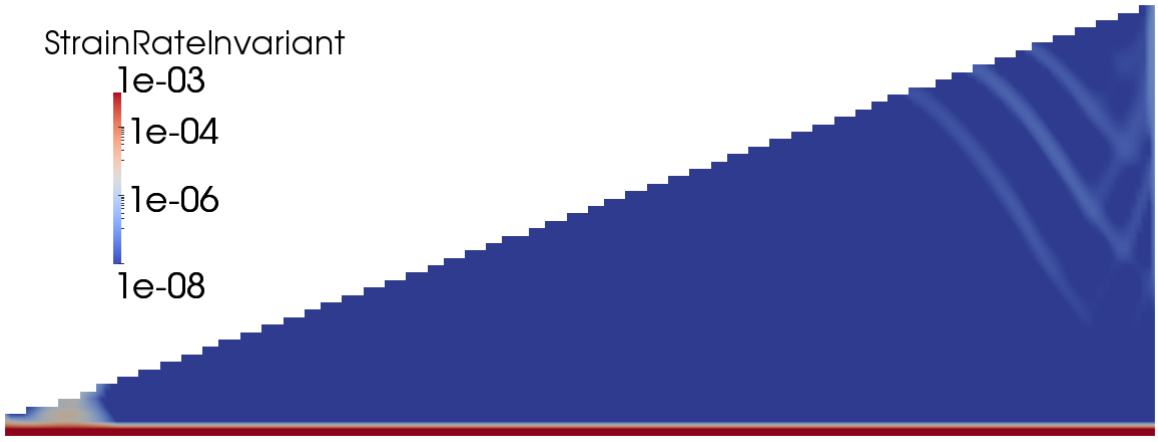


Figure C.27: Strain rate invariant for the stable wedge benchmark within the wedge. Outside the wedge, the strain rates are large because of the air's low viscosity. The resolution is 256x64, and the wedge has translated 4 cm.

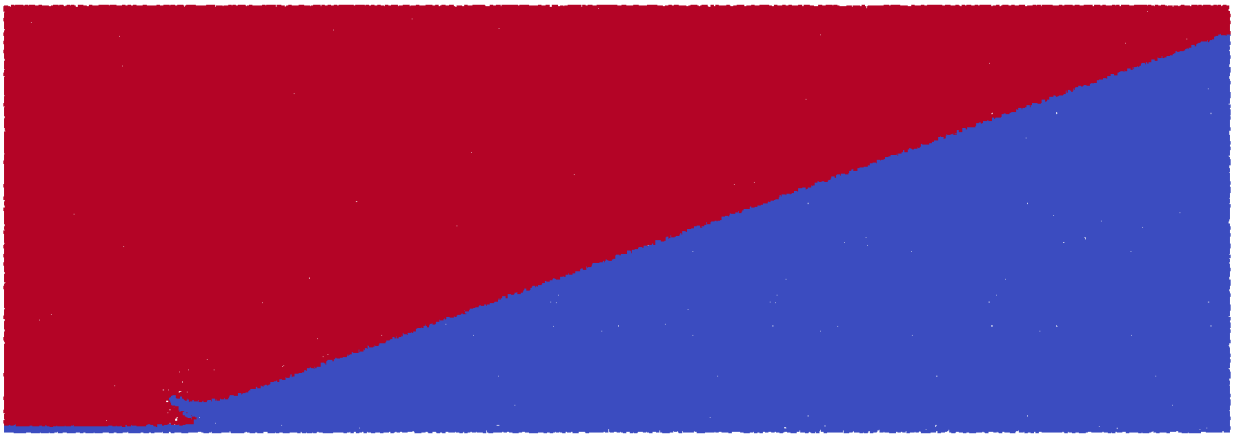


Figure C.28: Material particles for the stable wedge benchmark. Deformation at the tip is caused by a finite boundary cohesion. The odd structure at the tip is the result of the finite air viscosity, although the actual structure is not well resolved. The resolution is 256x64, and the wedge has translated 4 cm.

C.7.2 Unstable Shortening

This benchmark simulates a wall pushing against a wall of sand as in Figure C.29. There are three layers of sand, with the middle layer being a little heavier and sticking a little more to the boundary. Figures C.30, C.31, and C.32 show results for different resolutions.

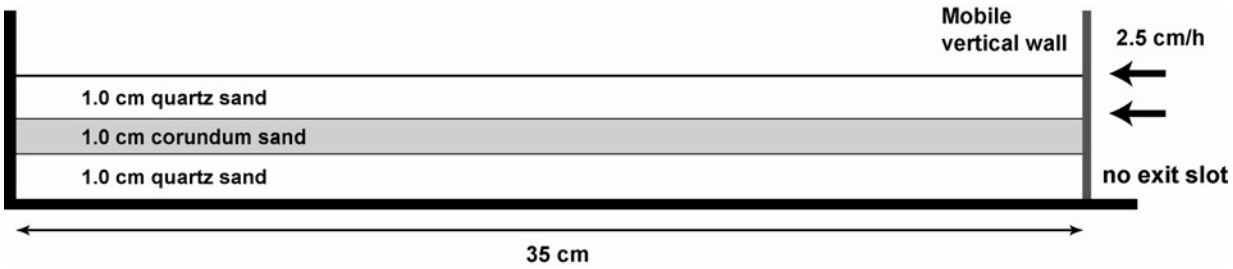


Figure C.29: Set up for the unstable shortening benchmark. Image courtesy of Susanne Buiter.

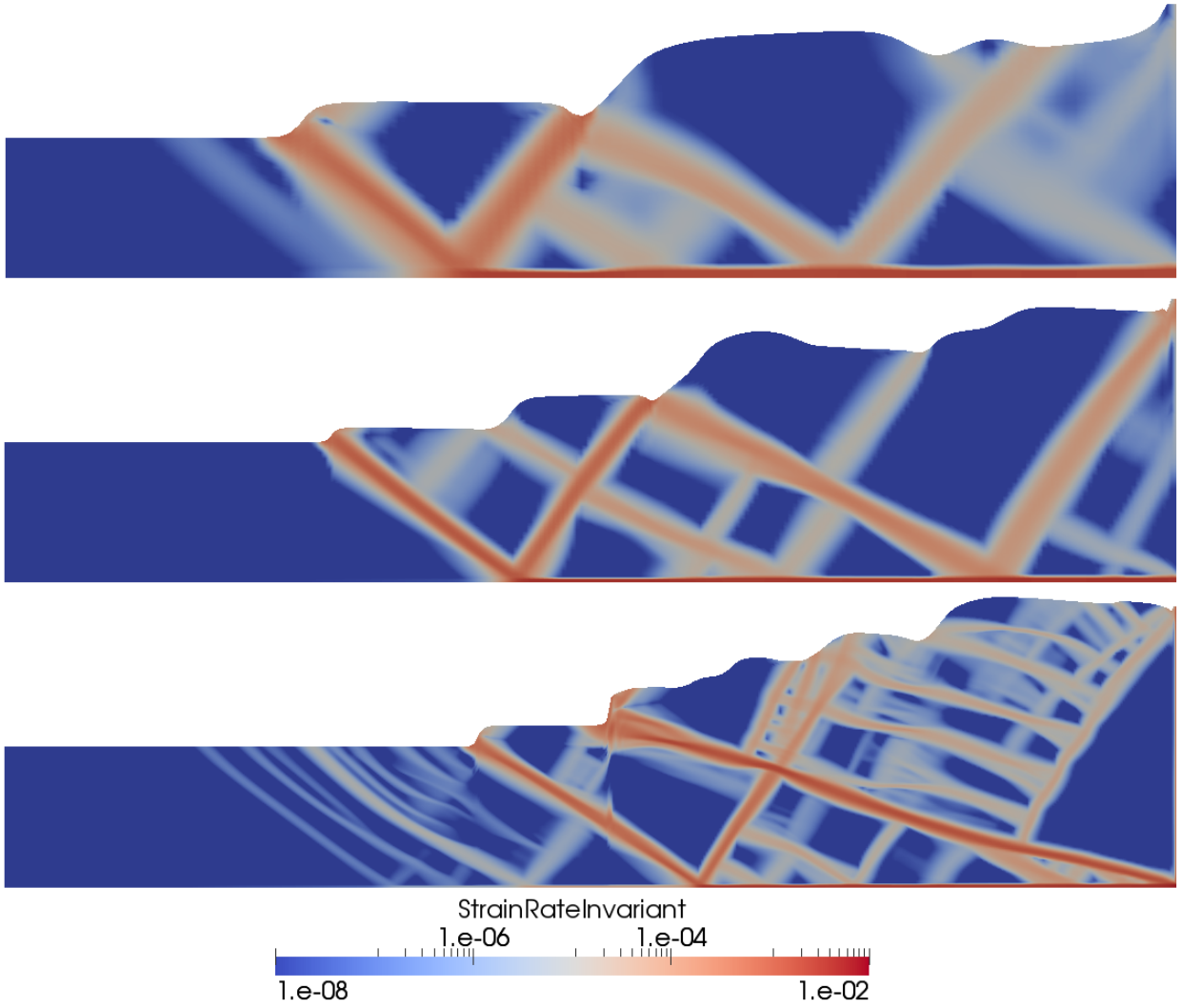


Figure C.30: Strain rate invariant for the unstable shortening benchmark at 10 cm of shortening with resolutions of 128×32, 256×64, and 512×128.

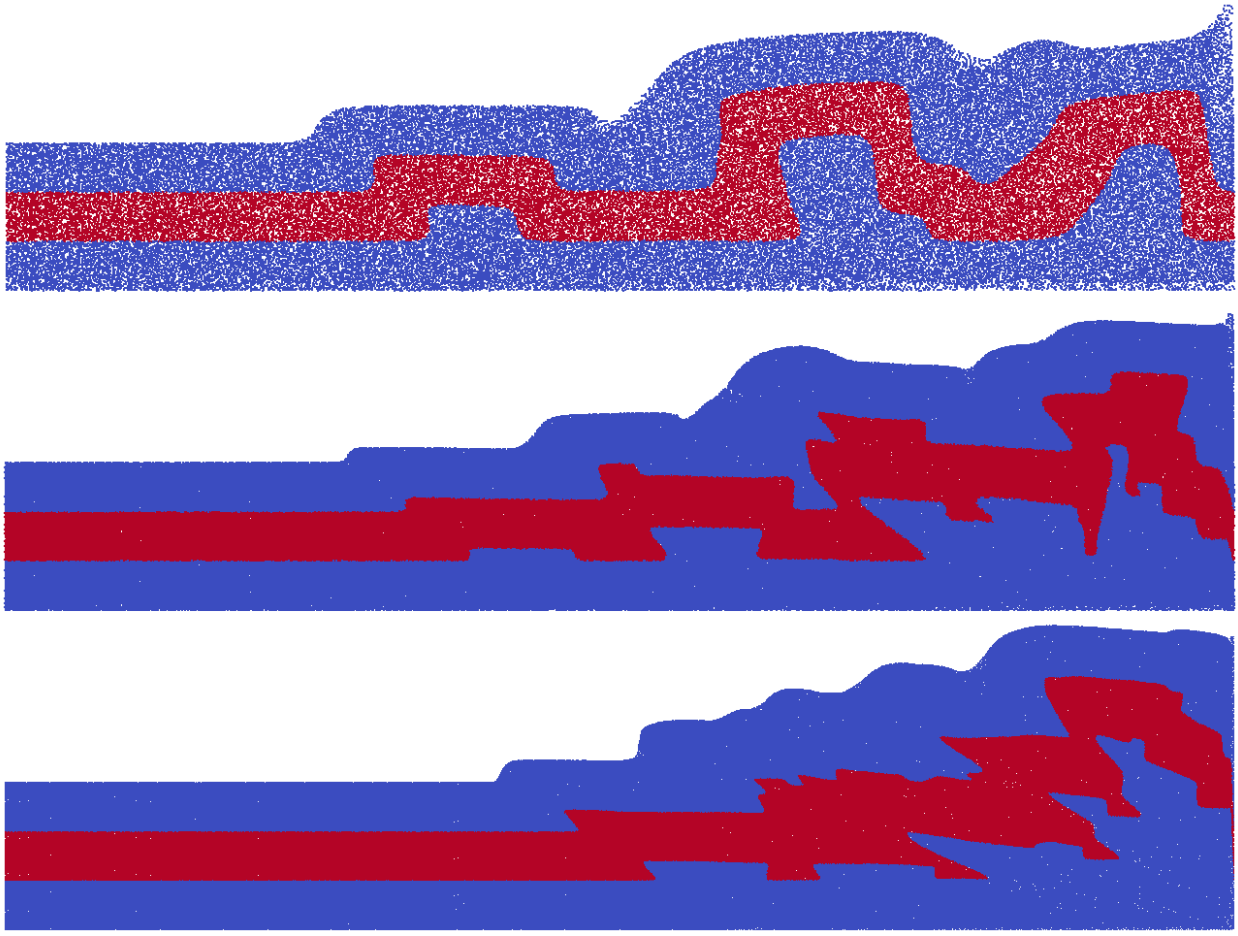


Figure C.31: Material particles for the unstable shortening benchmark at 10 cm of shortening with resolutions of 128×32, 256×64, and 512×128.

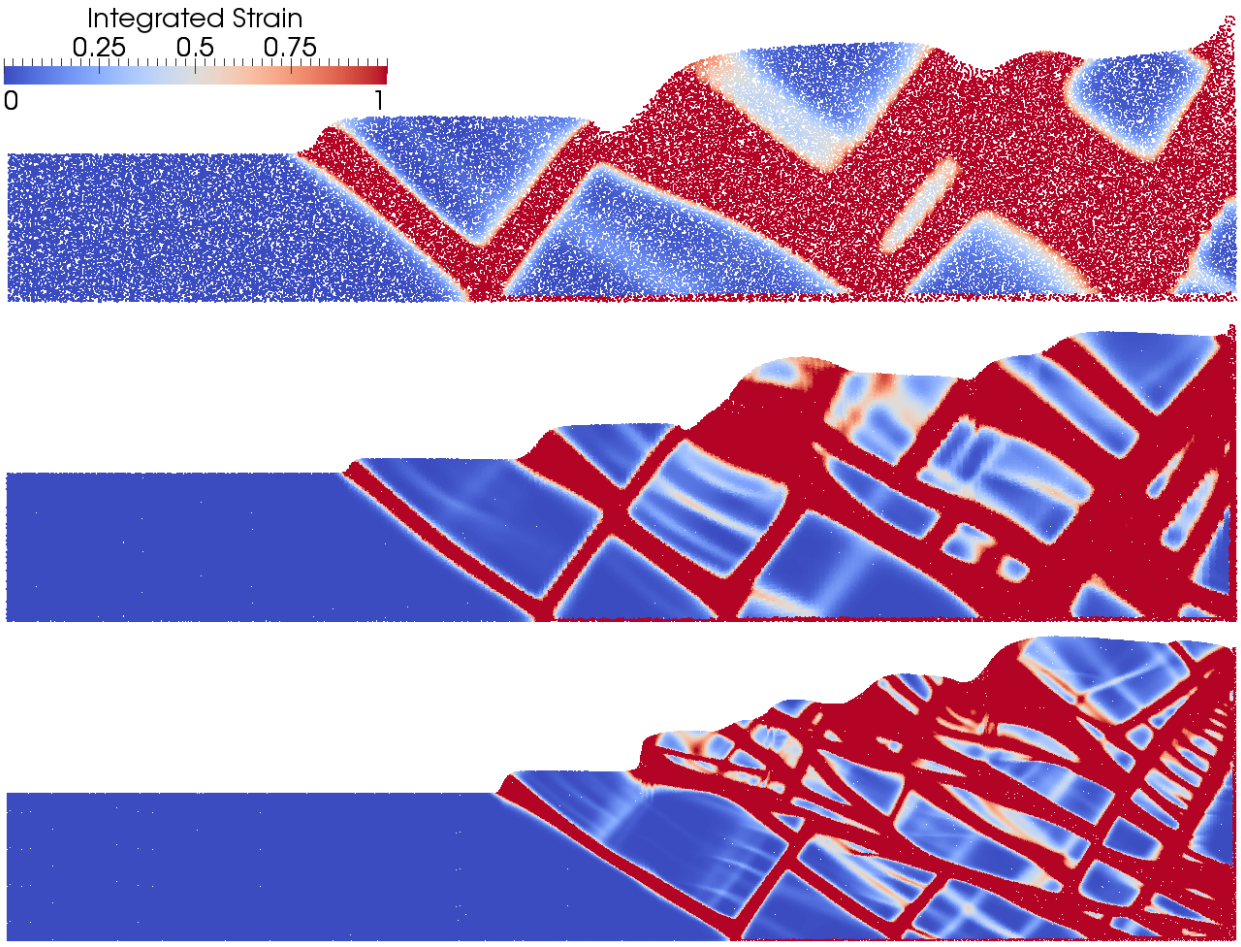


Figure C.32: Integrated strain for the unstable shortening benchmark at 10 cm of shortening with resolutions of 128×32, 256×64, and 512×128.

C.7.3 Brittle Shortening

This benchmark is very similar to unstable shortening. The only difference is that part of the bottom is also moving along as shown in Figure C.33. This causes the deformation to start from inside the sand box, rather than along the walls. Figures C.34, C.35, and C.36 show results for different resolutions.

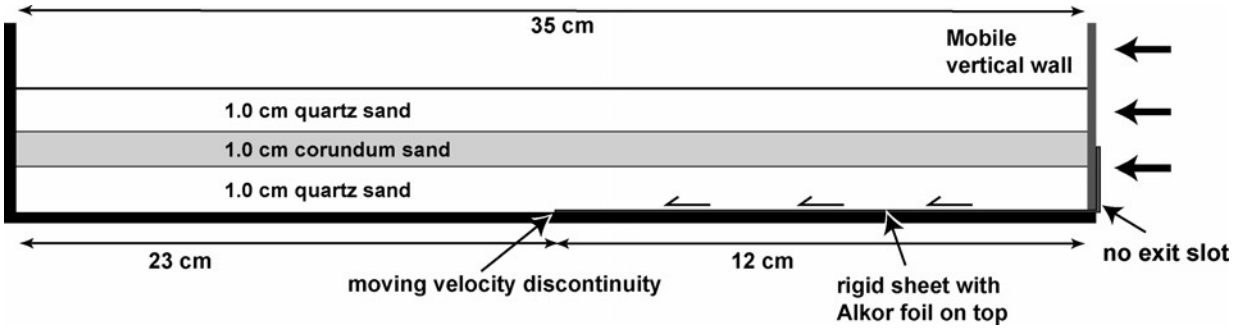


Figure C.33: Set up for the brittle shortening benchmark. Image courtesy of Susanne Buiter.

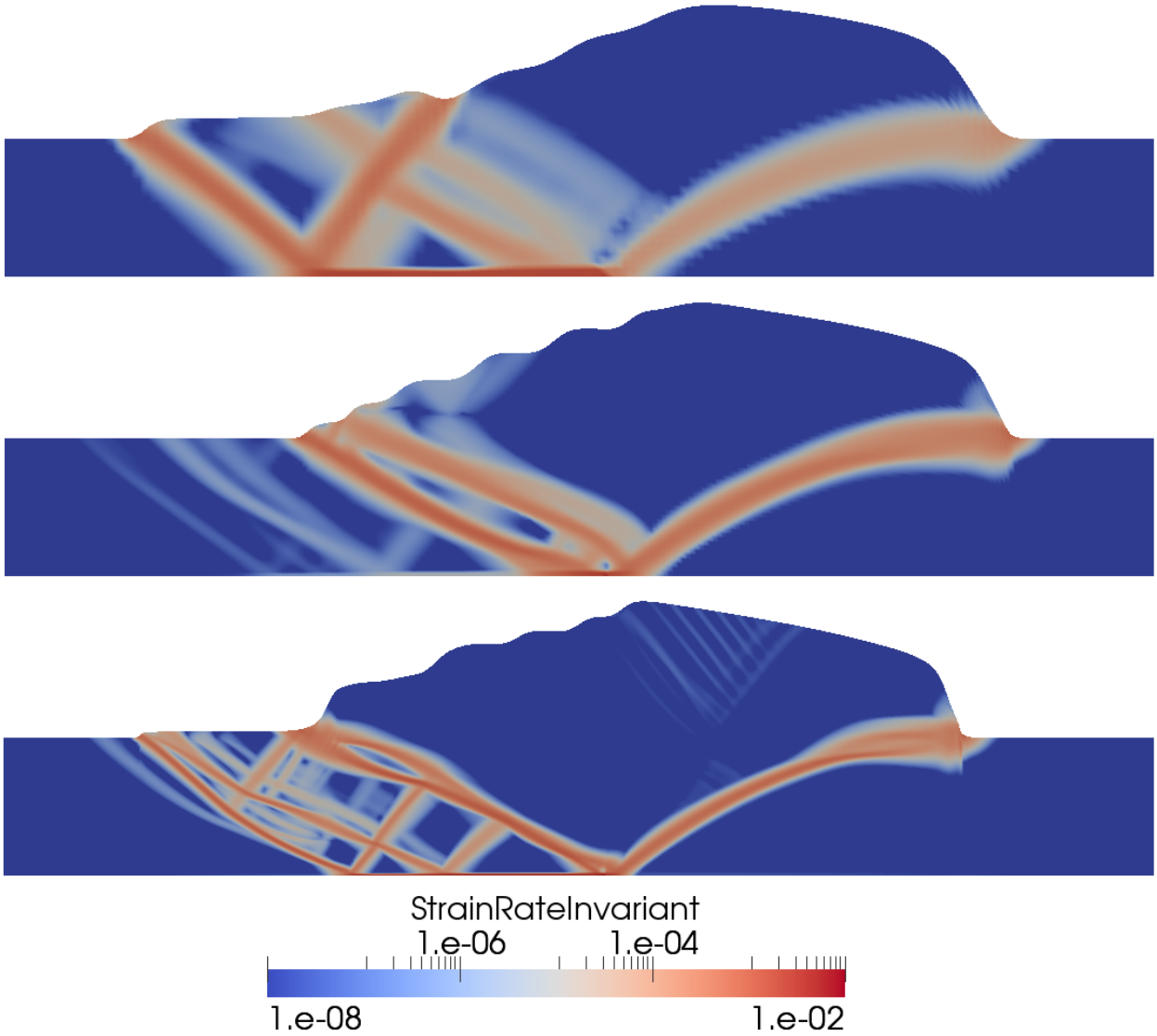


Figure C.34: Strain rate invariant for the brittle shortening benchmark at 10 cm of shortening with resolutions of 128×32, 256×64, and 512×128.

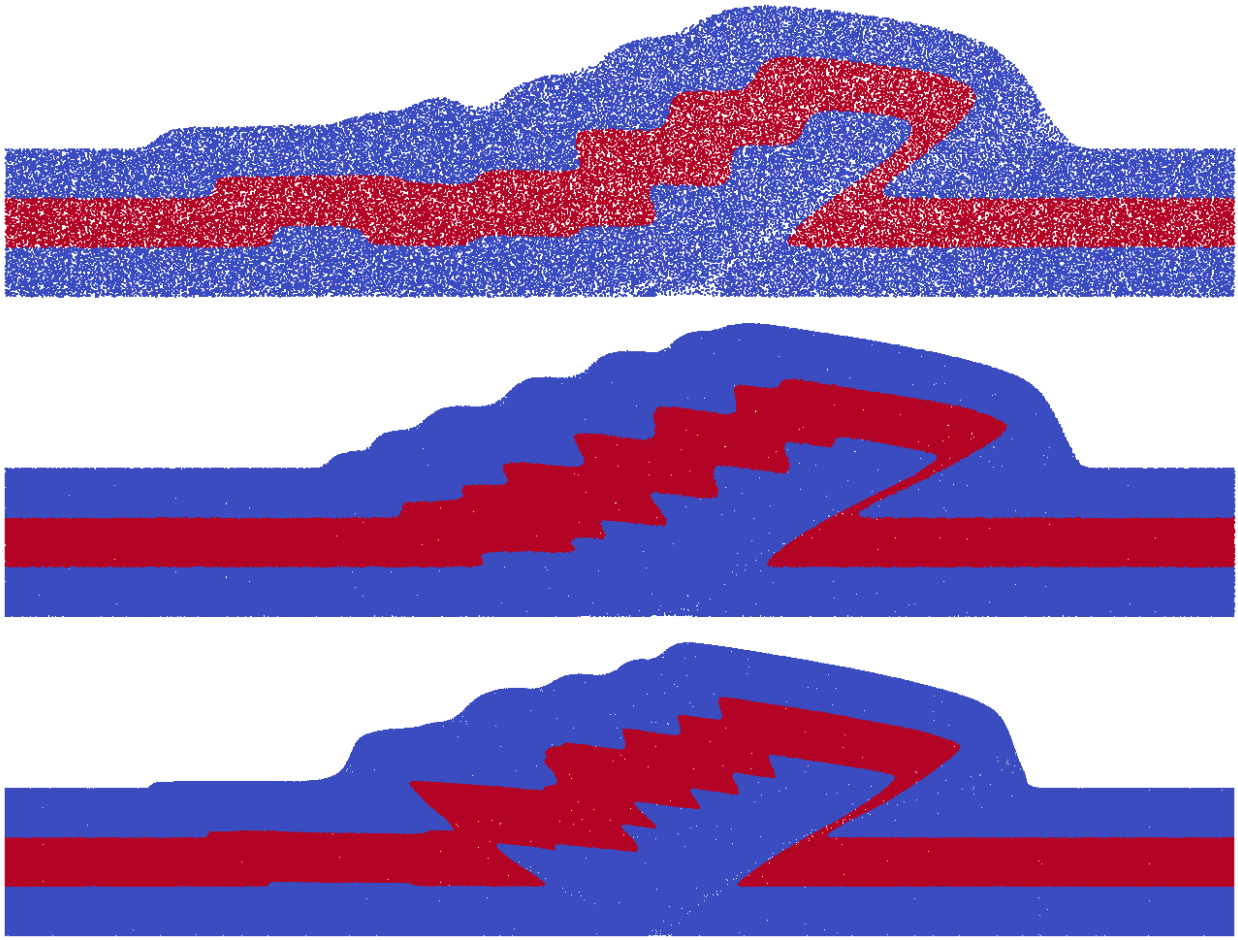


Figure C.35: Material particles for the brittle shortening benchmark at 10 cm of shortening with resolutions of 128×32, 256×64, and 512×128.

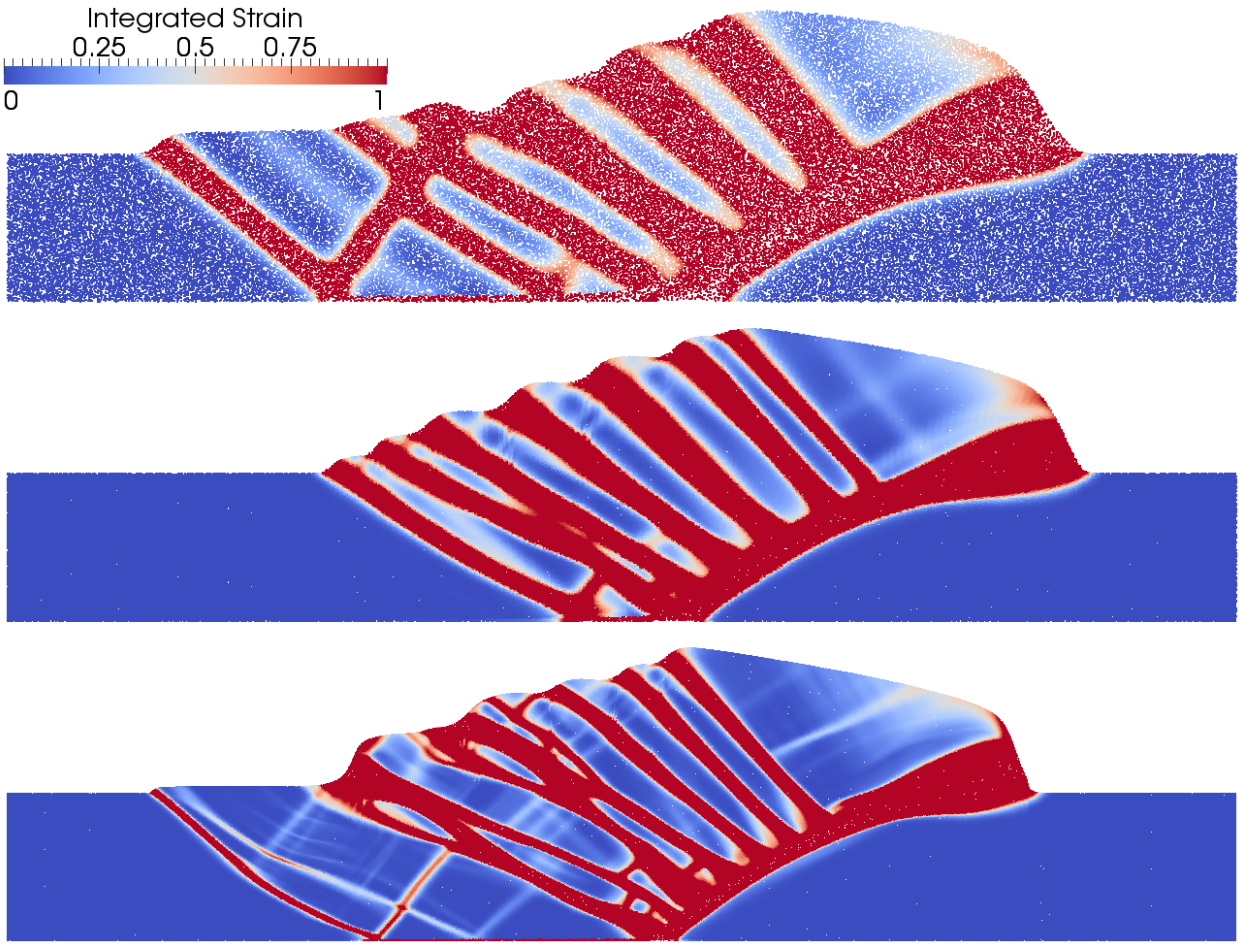


Figure C.36: Integrated strain for the brittle shortening benchmark at 10 cm of shortening with resolutions of 128×32, 256×64, and 512×128.

Appendix D

License

GNU GENERAL PUBLIC LICENSE Version 2, June 1991. Copyright (C) 1989, 1991 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software – to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps:

1. Copyright the software, and

2. Offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program" below refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification.") Each licensee is addressed as "you."

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version," you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found. For example:

> One line to give the program's name and a brief idea of what it does. Copyright © (year) (name of author)
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright © year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items – whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> (signature of Ty Coon)
> 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Bibliography

[1] Fullsack, Phillipe (1995). An arbitrary Lagrangian-Eulerian formulation for creeping flows and its application in tectonic models, *Geophys. J. Int., 120*, 1-23.

[2] Quenette, S., B. Appelbe, M. Gurnis, L. Hodkinson, L. Moresi, and P. Sunter (2005), An Investigation into Design for Performance and Code Maintainability in High Performance Computing, *ANZIAM J., 46*(e), C1001-C1016.

[3] Moresi, L.N., F. Dufour, and H.-B. Mühlhaus (2003), A Lagrangian integration point finite element method for large deformation modeling of viscoelastic geomaterials, *J. Comp. Phys., 184*, 476-497.

[4] Moresi, L.N., and H.-B. Mühlhaus (2006), Anisotropic viscous models of large-deformation Mohr-Coulomb failure, *Philosophical Magazine, 86*(21), 3287-3305.

[5] Moresi, L.N., and V.S. Solomatov (1995), Numerical investigation of 2D convection with extremely large viscosity variations, *Phys. Fluids, 7*(9), 2154-2162.

[6] O'Neill, C., L. Moresi, D. Müller, R. Albert, and F. Dufour (2006), Ellipsis 3D: a particle-in-cell finite element hybrid code for modelling mantle convection and lithospheric deformation, *Comput. Geosci. 32*(10), 1769-1779.

[7] Zhong, S., M.T. Zuber, L.N. Moresi, and M. Gurnis (2000), The role of temperature-dependent viscosity and surface plates in spherical shell models of mantle convection, *J. Geophys. Res.*, *105*, 11,063-11,082.

[8] Schmid, D.W., and Y.Y. Podladchikov (2003), Analytical solutions for deformable elliptical inclusions in general shear, *Geophys. J. Int.*, *155*, 269-288.

[9] Landau, L.D., and E.M. Lifshitz (1987), *Fluid Mechanics*, Pergamon Press, 61-62.

[10] Johnson, A.M., and R.C. Fletcher (1994), *Folding of Viscous Layers*, Columbia University Press, 19.

[11] Buiter, S.J.H., and A.Y. Babeyko, S. Ellis, T.V. Gerya, B.J.P. Kaus, A. Kellner, G. Schreurs, and Y. Yamada (2006), The numerical sandbox: comparison of model results for a shortening and an extension experiment, *Analogue and Numerical Modelling of Crustal-Scale Processes*, *253*, edited by S.J.H. Buiter and G. Schreurs, pp. 29-64, Geological Society, London, Special Publications, doi: 10.1144/GSL.SP.2006.253.01.02.

[12] Lindgren, E.R. (1999), The Motion of a Sphere in an Incompressible Viscous Fluid at Reynolds Numbers Considerably Less Than One, *Physica Scriptae*, *60*, 97-110.

[13] Deubelbeiss, Y., and B.J.P. Kaus (2007), A comparison of finite difference formultions for the Stokes equations in presence of strongly varying viscosity, poster presented at 2007 AGU.

[14] Dohrmann, C., and P. Bochev (2004), A stabilized finite element method for the Stokes problem based on polynomial pressure projections, *Int. J. Num. Meth. Fluids.*, *46,* 183-201

[15] Elman, H.C., D.J. Silvester, and A.J. Wathen (2005), *Finite Elements and Fast Iterative Solvers: with Applications in Incompressible Fluid Dynamics*, Oxford University Press

[16] Buiter, S., and G. Schreurs, http://www.geodynamics.no/benchmarks/benchmark-annum2008.html

[17] Dahlen, F.A. (1984), Noncohesive Critical Wedges: An Exact Solution, *J. Geophys. Res., 89*, B12, 10125-10133

[18] Kaus, B.J.P. (2009), Factors that control the angle of shear bands in geodynamic numerical models of brittle deformation, *Tectonophysics*, *484*, 36-47

[19] Buck, W.R., L.L. Lavier, and A.N.B. Poliakov (2005), Modes of faulting at mid-ocean ridges, *Nature*, *434*, 719-723

[20] Hilley, G.E. and M.R. Strecker (2004), Growth and erosion of fold-and-thrust belts with an application to the Aconcagua fold-and-thrust belt, Argentina, *J. Geophys. Res.*, *109*, B01410, doi:10.1029/2002JB002282