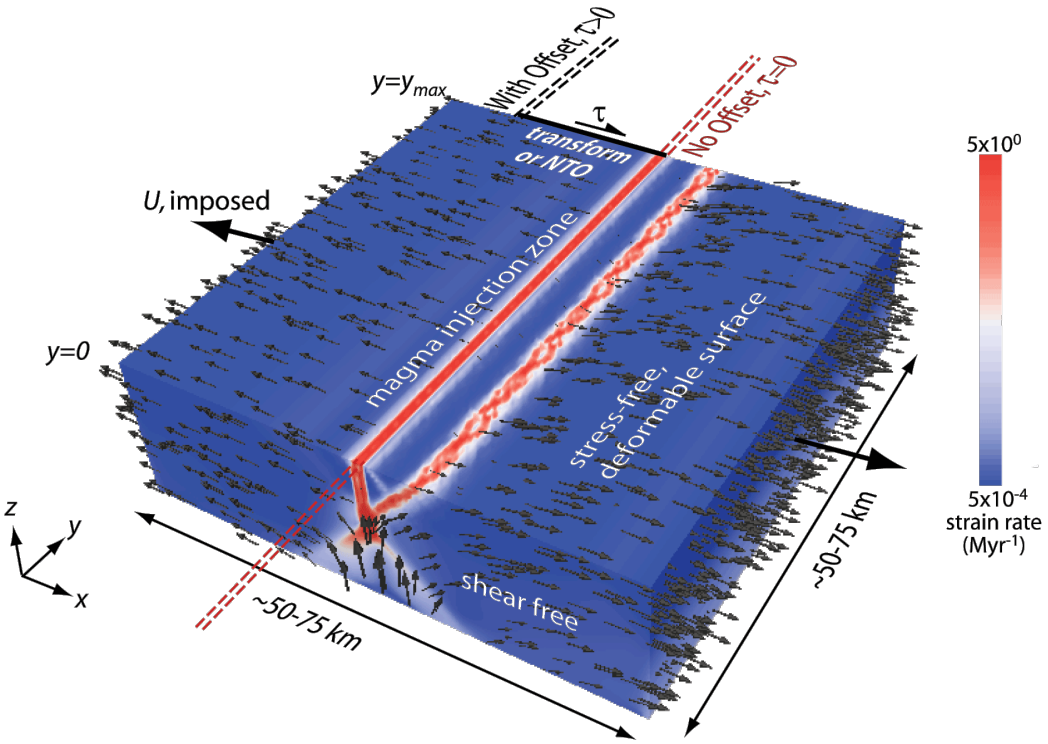
COMPUTATIONAL INFRASTRUCTURE FOR GEODYNAMICS (CIG)
VICTORIA PARTNERSHIP FOR ADVANCED COMPUTING (VPAC)
MONASH UNIVERSITY

Gale

User Manual
Version 2.0.1



Walter Landry
Luke Hodkinson
Susan Kientz

www.geodynamics.org

Gale

Walter Landry and Luke Hodkinson
Version 2.0.1

August 22, 2012

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, integrated 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 2.2.8.3).

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 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.4.

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 Finite Elements

Gale can use a few different types of finite elements to represent the solution. The recommended elements are quadratic (Q_2) elements for the velocity and temperature, and discontinuous linear (P_{-1}) elements for the pressure. These elements are mathematically well behaved and have been used in other computational codes with success.

If, for some reason, you wish use a different element type, Gale also supports linear (Q_1) and piecewise constant (P_0) elements. One formulation common in many solid earth modelling codes is to use Q_1 elements for the velocity and P_0 elements for the pressure. This formulation gives rise to a checkerboard instability. While this instability is not always fatal, dealing with it can be difficult and error prone.

Previous versions of Gale did not support Q_2 or P_{-1} elements, so the recommendation was to use Q_1 elements for both the velocity and pressure. This formulation has its own instability that is fixed by adding an artificial compressibility. In principle, this artificial compressibility should be small and get smaller as resolution increases. In practice, for realistic geologic problems, the artificial compressibility was far too large and dramatically altered the dynamics.

2.2.8.2 Thermal Advection and Diffusion

Gale uses the Stream Upwind Petrov-Galerkin (SUPG) method to solve the energy equation (eq. 2.12). This should normally work without any modification. However, if the elements in your model gets significantly distorted, you may see anomalously high temperature variations. To fix this, you can modify `supgFactor`, as detailed in Section A.2.

2.2.8.3 Scaling

One thing to note is that Equations 2.2 and 2.3 have different units. So, if you have a viscosity of $10^{25} Pa \cdot s$ and you express your viscosities in $Pa \cdot s$, the numbers in the two equations will be too disparate and cause the solver to fail. 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

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

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.4.2.4), you have to scale `A`, `refStrainRate`, `minViscosity`, and `maxViscosity`. For example, `A` has units of $s^{-1} Pa^{-n}$, so in this case $A_{new} = A_{old} 10^{25}$.

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

$$\begin{pmatrix} K & G \\ G^T & C \end{pmatrix} \begin{pmatrix} v \\ p \end{pmatrix} = \begin{pmatrix} f \\ d \end{pmatrix}, \tag{2.13}$$

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

$$\begin{array}{rcl} Kv + Gp & = & f \\ G^T v + Cp & = & d \end{array}. \tag{2.14}$$

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 2.14, 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 $K\,u_0 = f - G q_0$ 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 $Ku^* = Gs_k$ 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 $(u_{k+1} - u_k) / u_{k+1} > linear\, tolerance$

That will give us a single solution to Equation 2.14 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 2.14 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 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, and Windows.

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. In order to build Gale, you must have a C++ compiler and the headers and development libraries for

- MPI
- PETSc 3.0 (not 3.1!)
- libxml2
- HDF5

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.

HDF5 is not strictly required, but checkpointing and visualization will not work without it.

3.3.2 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.

3.3.2.1 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 -u
cd PICellerator
hg pull -u
cd ../StGermain
hg pull -u
cd ../StgDomain
hg pull -u
cd ../StgFEM
hg pull -u
cd ../Underworld
hg pull -u
cd ../config
hg pull -u
cd ../gLucifer
hg pull -u
```

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-2_0_1 input/cookbook/yielding.json
```

will output

```
TimeStep = 0, Time = 0
TimeStep = 1, Time = 0.021503
TimeStep = 2, Time = 0.0427746
TimeStep = 3, Time = 0.0638247
TimeStep = 4, Time = 0.0846619
TimeStep = 5, Time = 0.105288
TimeStep = 6, Time = 0.125705
TimeStep = 7, Time = 0.145914
TimeStep = 8, Time = 0.165918
TimeStep = 9, Time = 0.185726
TimeStep = 10, Time = 0.205284
```

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 on line 1 at column 1
not a value
Error: Could not read input file input/cookbook/foo.json. Exiting.
```

Due to quirks in some implementations of MPI, you may have to specify the complete path to the input file (e.g., `./Gale-2_0_1 /home/juser/gale/input/cookbook/yielding`.json).

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.json
```

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 numerical 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.

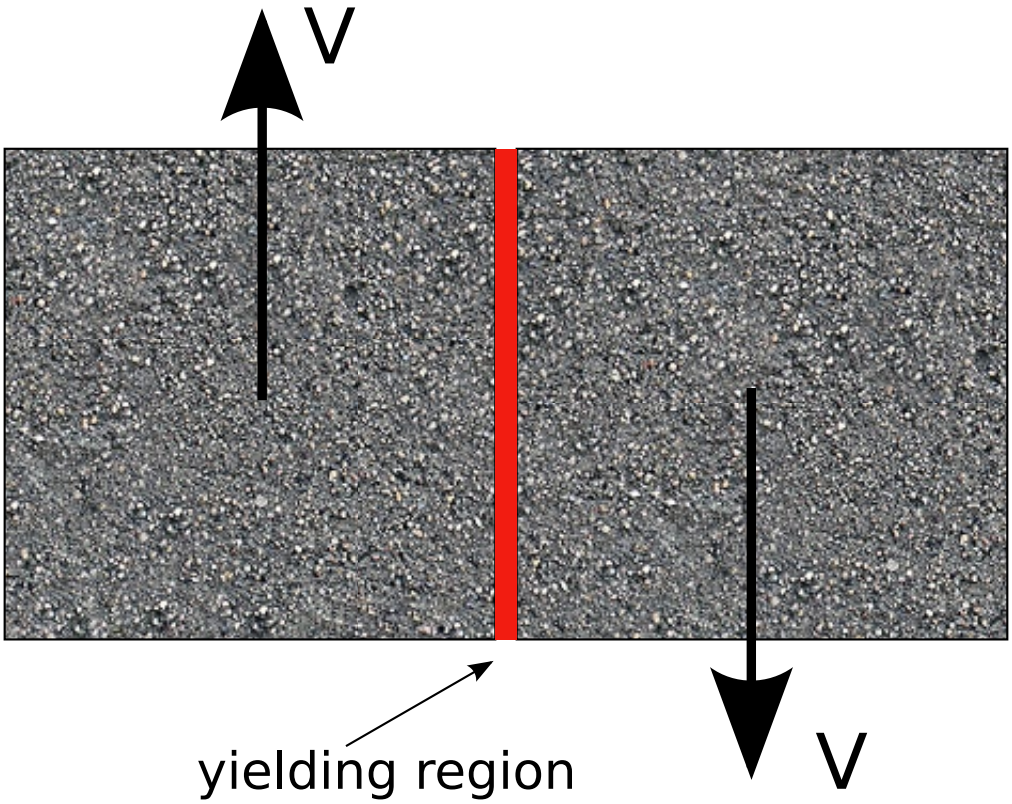


Figure 4.1: Two blocks sliding past each other with a yielding region between them.

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.

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 B.1), or when materials yield.

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-2_0_0  input/cookbook/yielding.json -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-2_0_0  input/cookbook/yielding.json -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 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-2_0_0  input/cookbook/yielding.json --maxTimeSteps=20
```

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

```
./Gale-2_0_0  input/cookbook/yielding.json --maxTimeSteps=20 --dim=3 --elementResI=64
--elementResJ=64 --elementResK=64 --particlesPerCell=60 --checkpointEvery=10
```

4.2.4 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

```
"checkpointEvery": "1"
```

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.

Not all of the example input files save and restore the temperature. To enable that, see Section A.2.

4.2.5 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 clearly 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

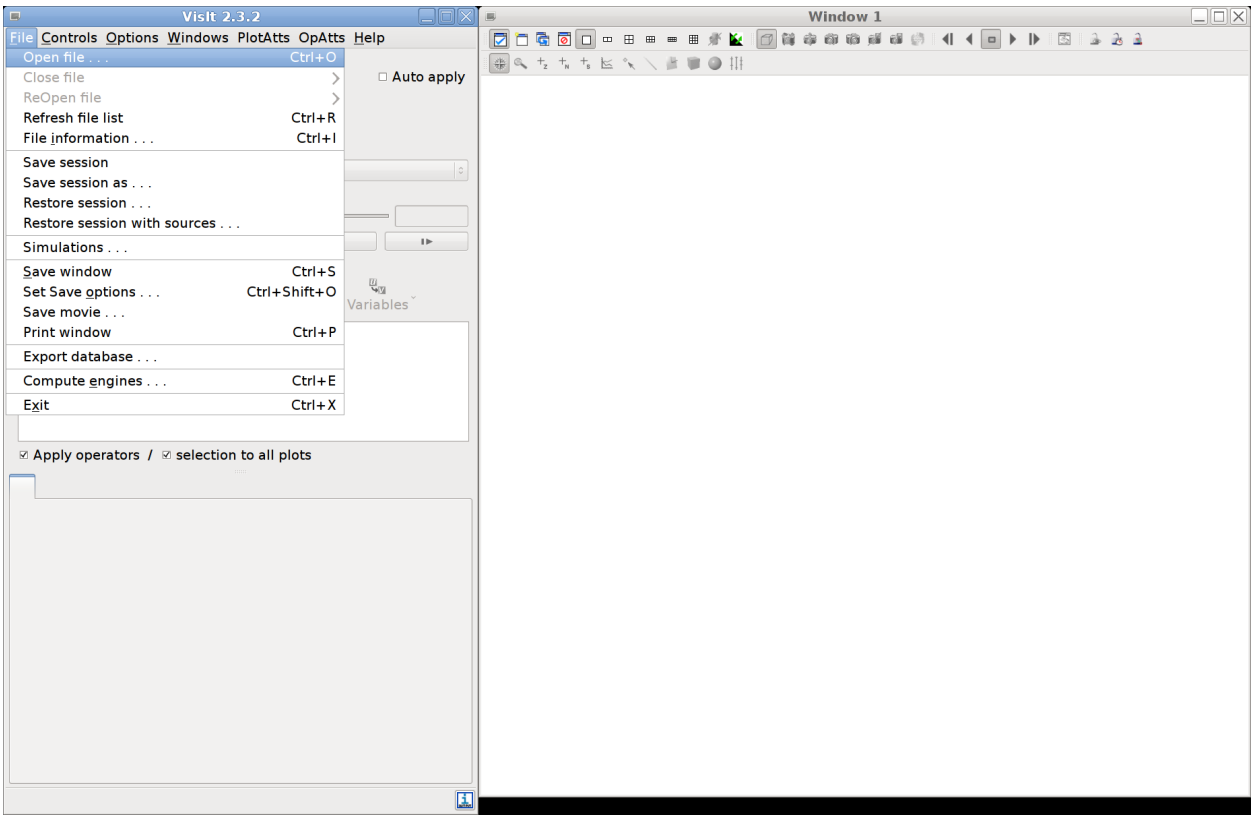
The sample input files will produce a directory in which you will find a number of files. The easiest way to visualize results is to use the XDMF files. These files are in a standard, self-describing file format that can be easily visualized with several different visualization programs, e.g., ParaView (`paraview.org`) and Visit (`www.llnl.gov/visit`). Visit is recommended as it is easy to get working, easy to use, and scales to large data sets.

XDMF visualization files are created at the same time as checkpoints. So to change the frequency at which Gale outputs XDMF files, change the parameter `checkpointEvery`.

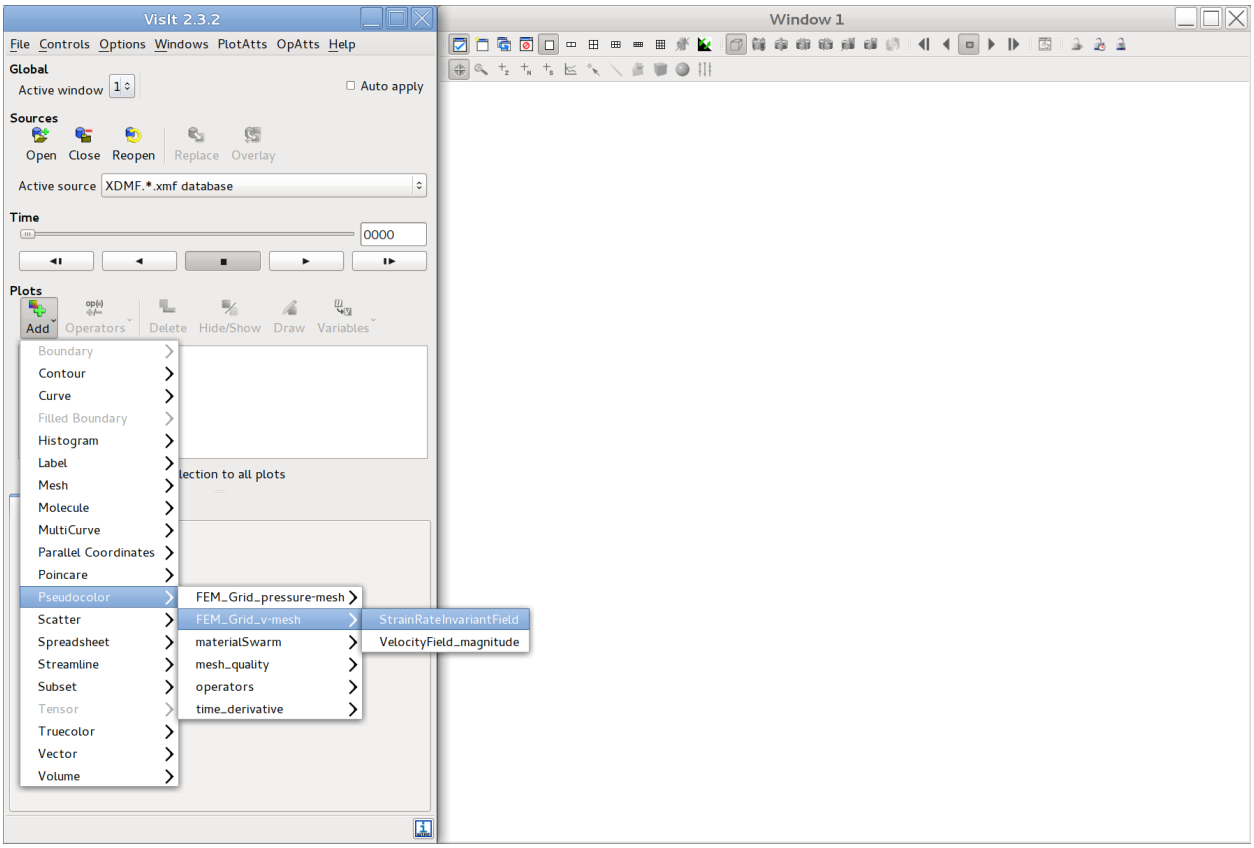
4.3.1 Basic Visualization with Visit

These instructions are for Visit version 2.3.2. To visualize the output of `input/cookbook/yielding.json`,

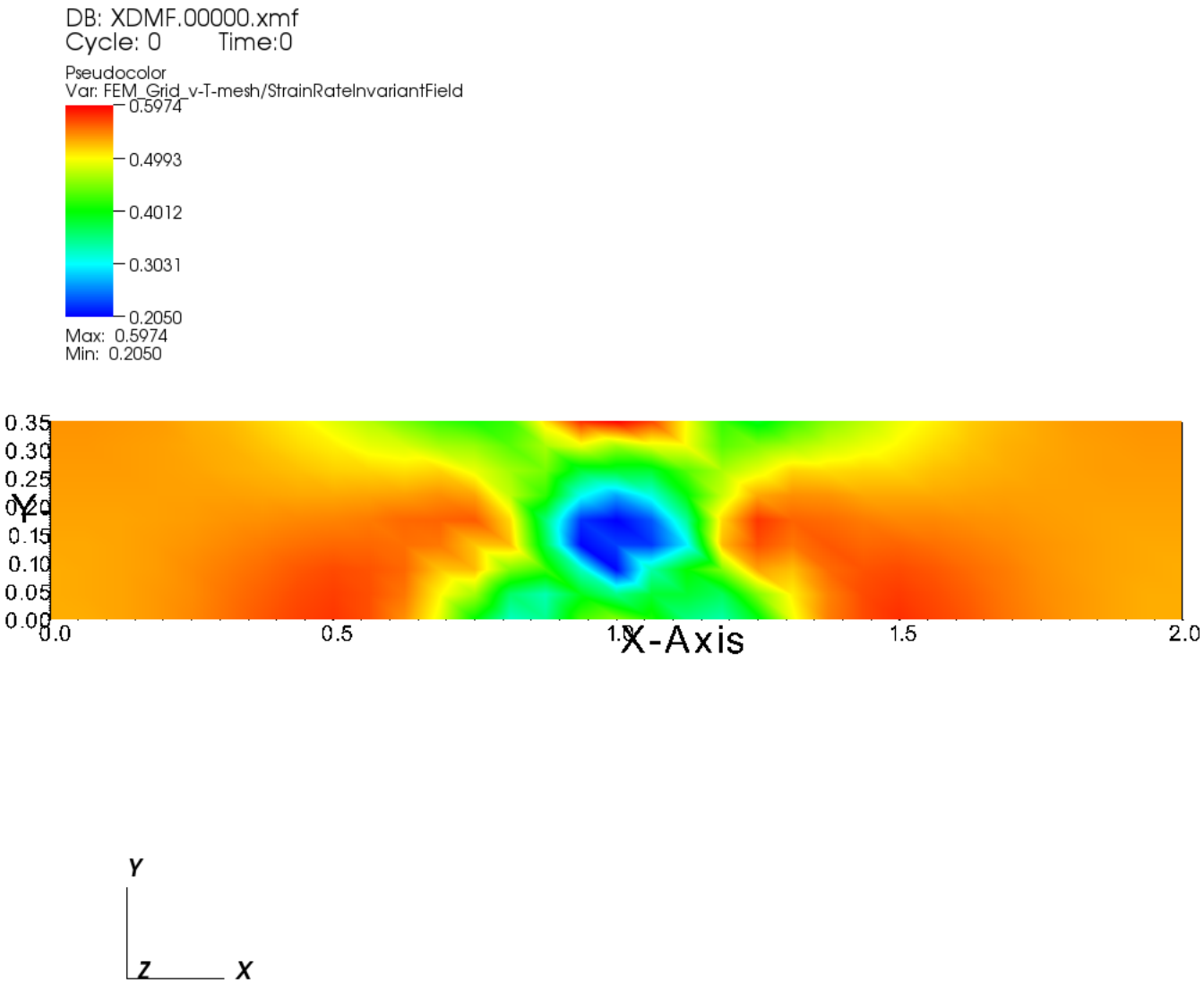
1. Start Visit and open a new data file: File ▷ Open



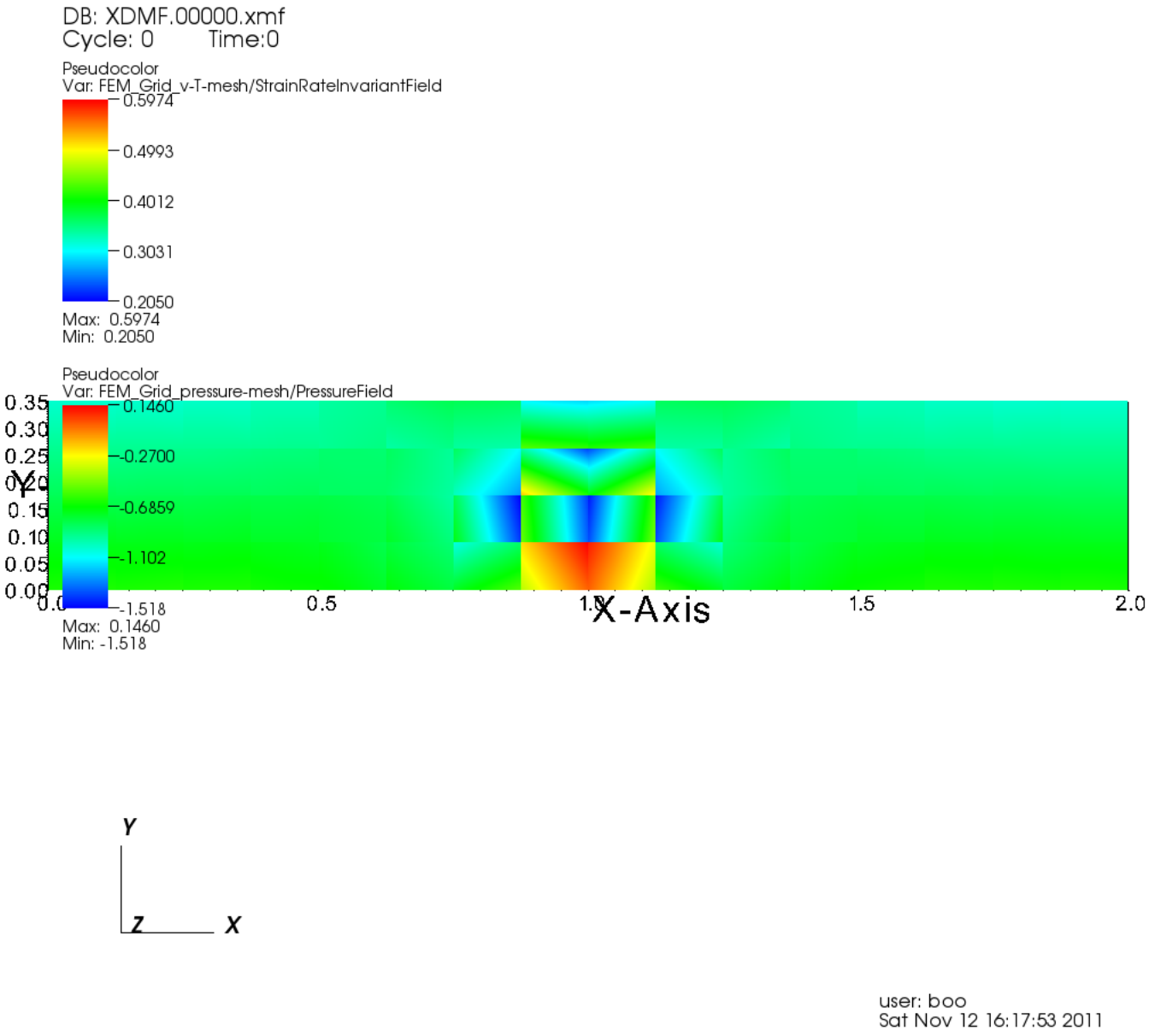
2. A file dialog screen will appear. Navigate to the output directory. Visit will automatically group similar files together. Select the `XDMF` files. Now click the `Add` button under `Plots`. Select `Pseudocolor`, then `FEM_Grid_v-mesh`, then `StrainRateInvariantField`.



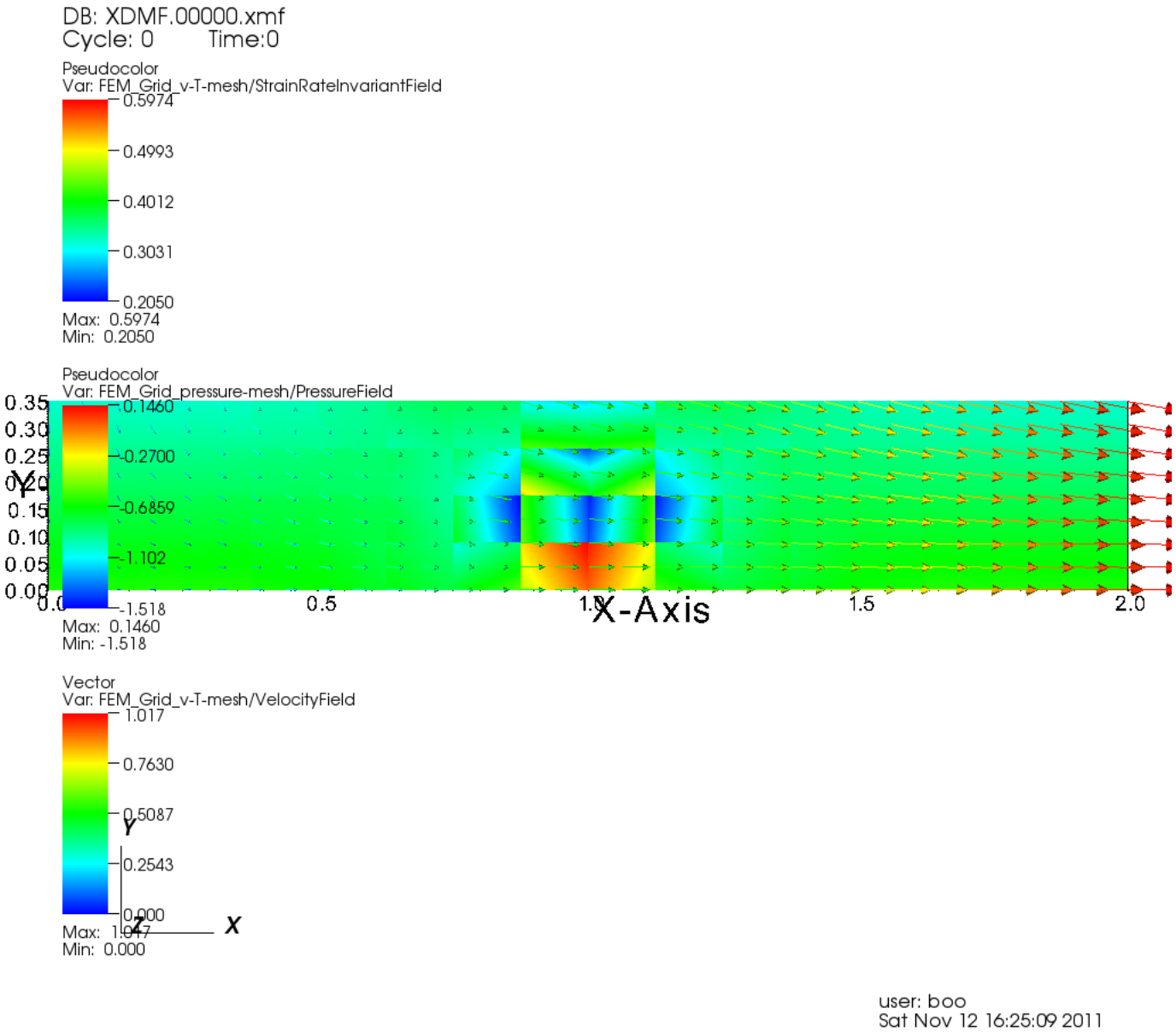
3. Now click on `Draw` and you will get a picture of the StrainRateInvariant at the first time step.



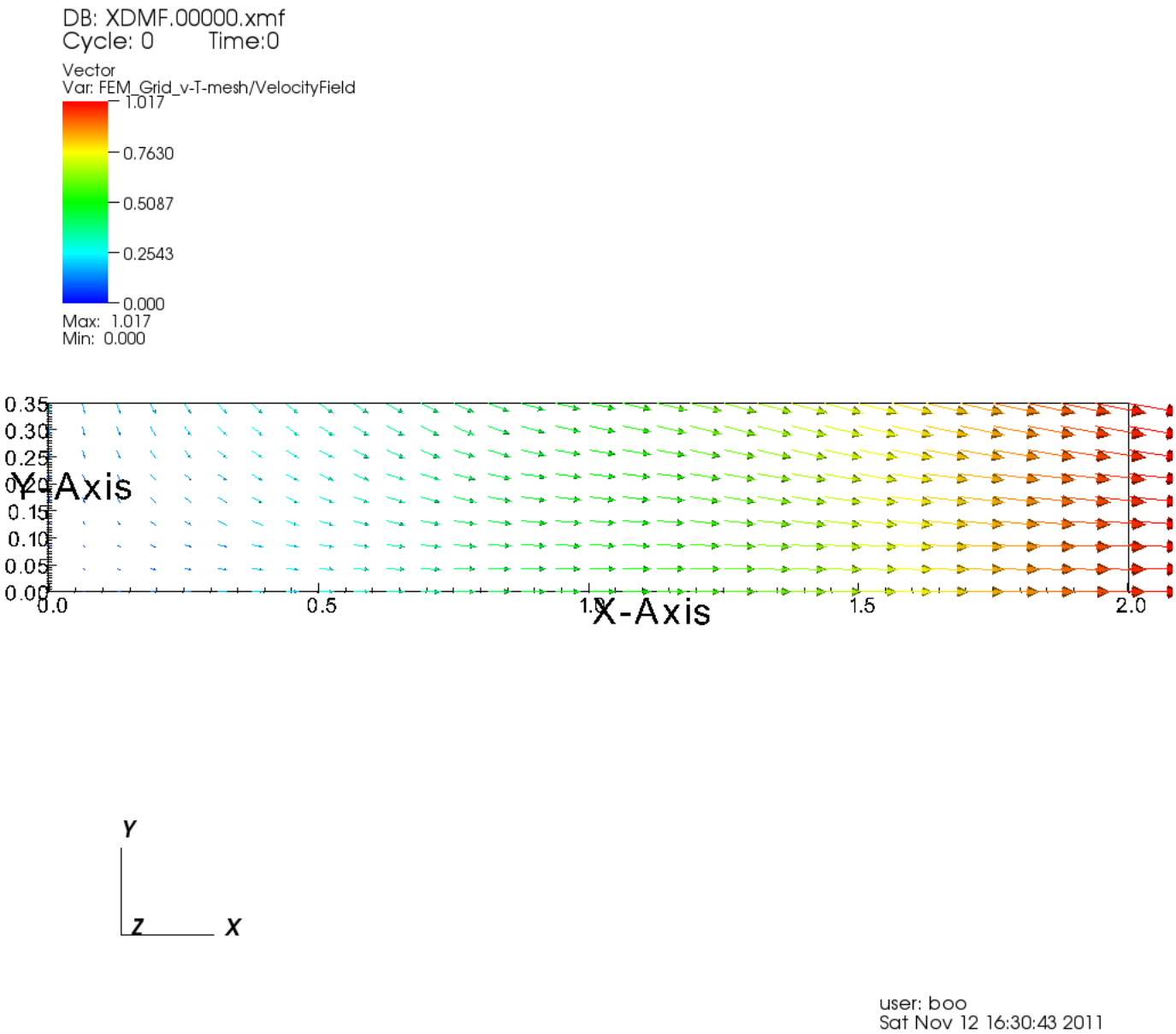
4. You can plot the pressure by clicking again on the `Add` button, selecting `Pseudocolor`, then `FEM_Grid_pressure-mesh`, then `PressureField`. Finally, click on `Draw`. The resolution is rather low, so the pressure solution is very rough.



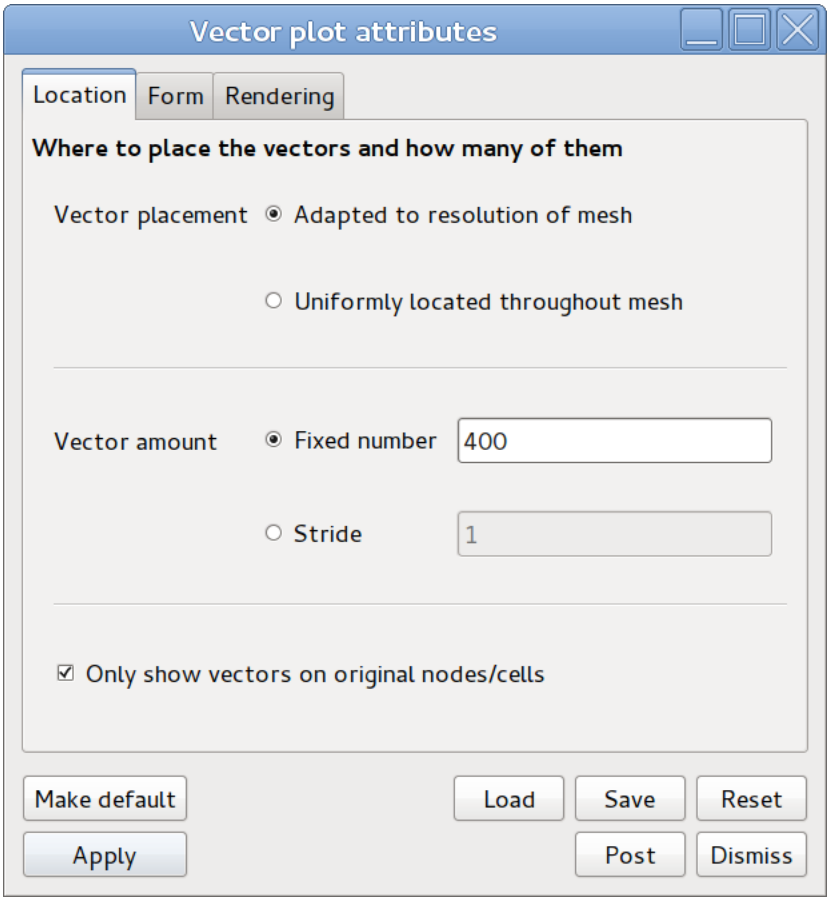
5. Now you can plot the velocity as arrows on top of the pressure: Click on the `Add` button, select `Vector`, then `FEM_Grid_v-mesh`, then `VelocityField`. Press `Draw` and you will see the velocity arrows colored by the velocity magnitude.



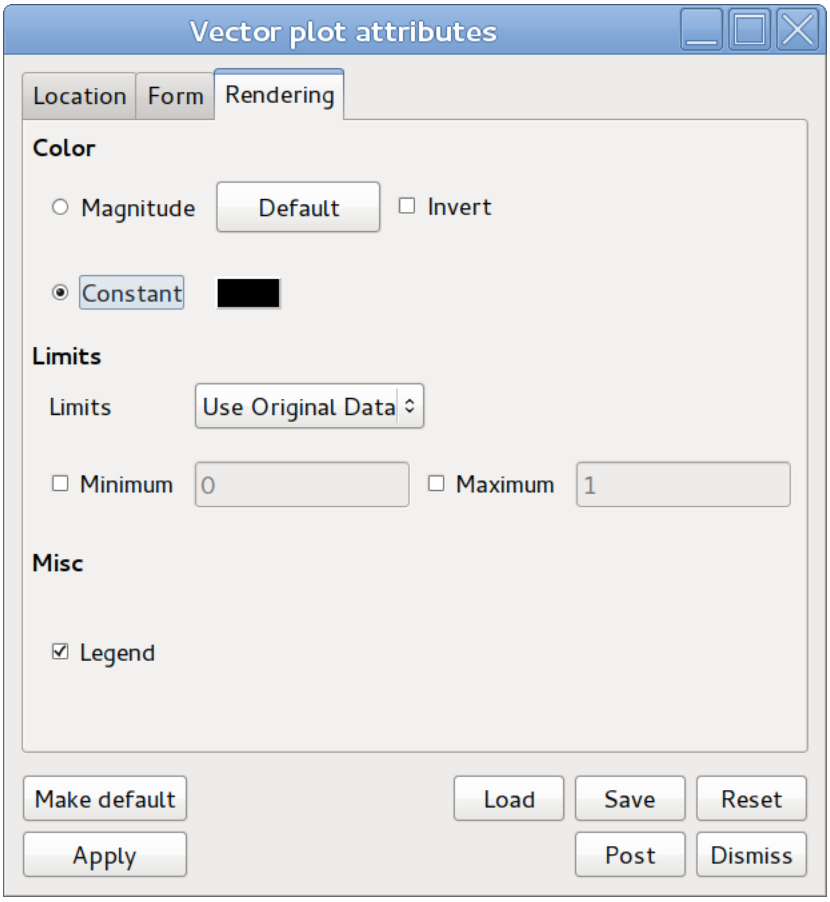
6. You can temporarily hide the pressure by clicking on `Pseudocolor - FEM_Grid_pressure-mesh/PressureField` and then clicking on the `Hide/Show` button. Repeat this with `Pseudocolor - FEM_Grid_v-mesh/StrainRateField` to see a clearer view of the velocity arrows.



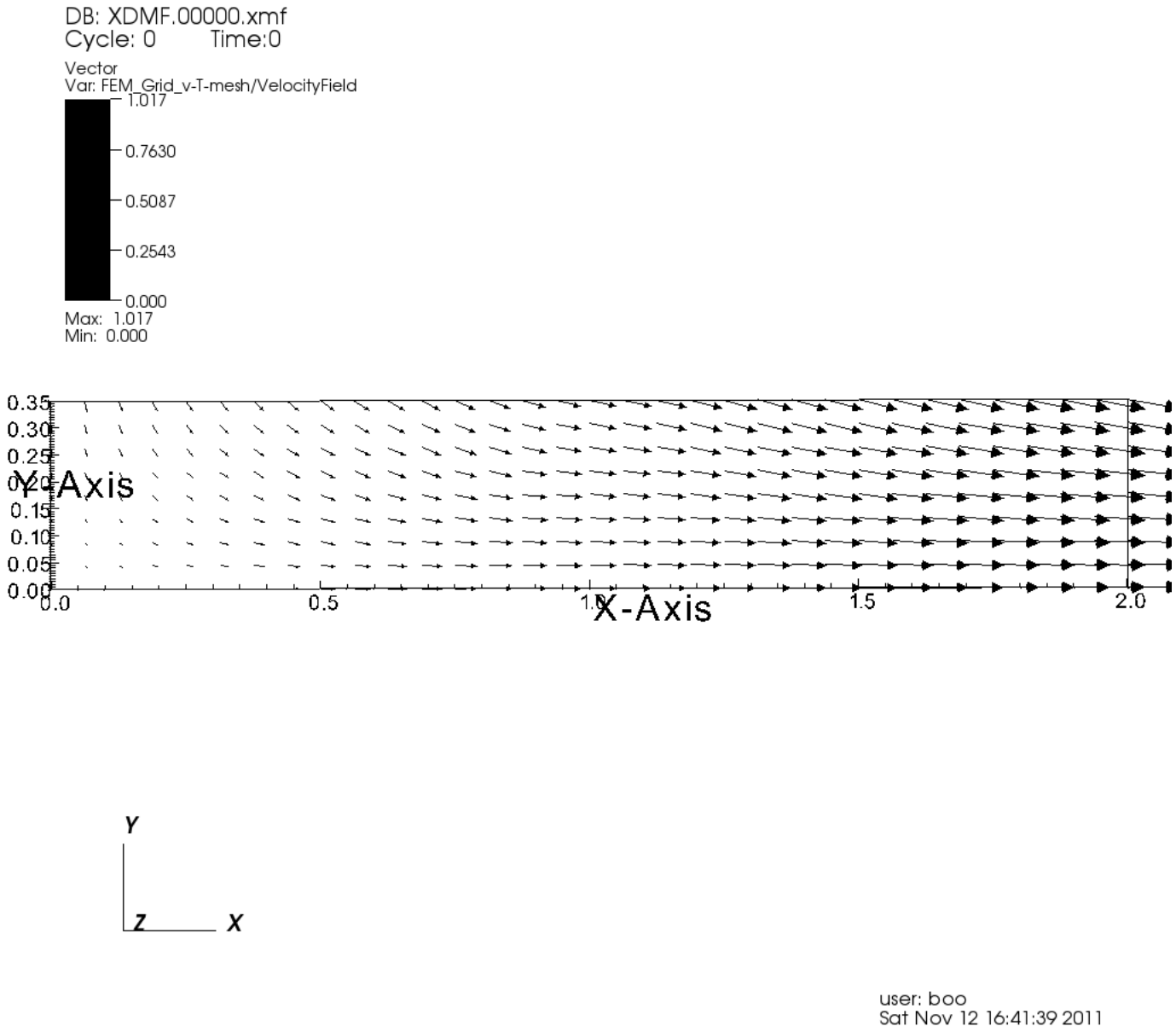
7. To change the color of the arrows, click on the `PlotAtts` menu item near the top. Select `Vector` in the drop down menu. This will bring up a new `Vector plot attributes` window.



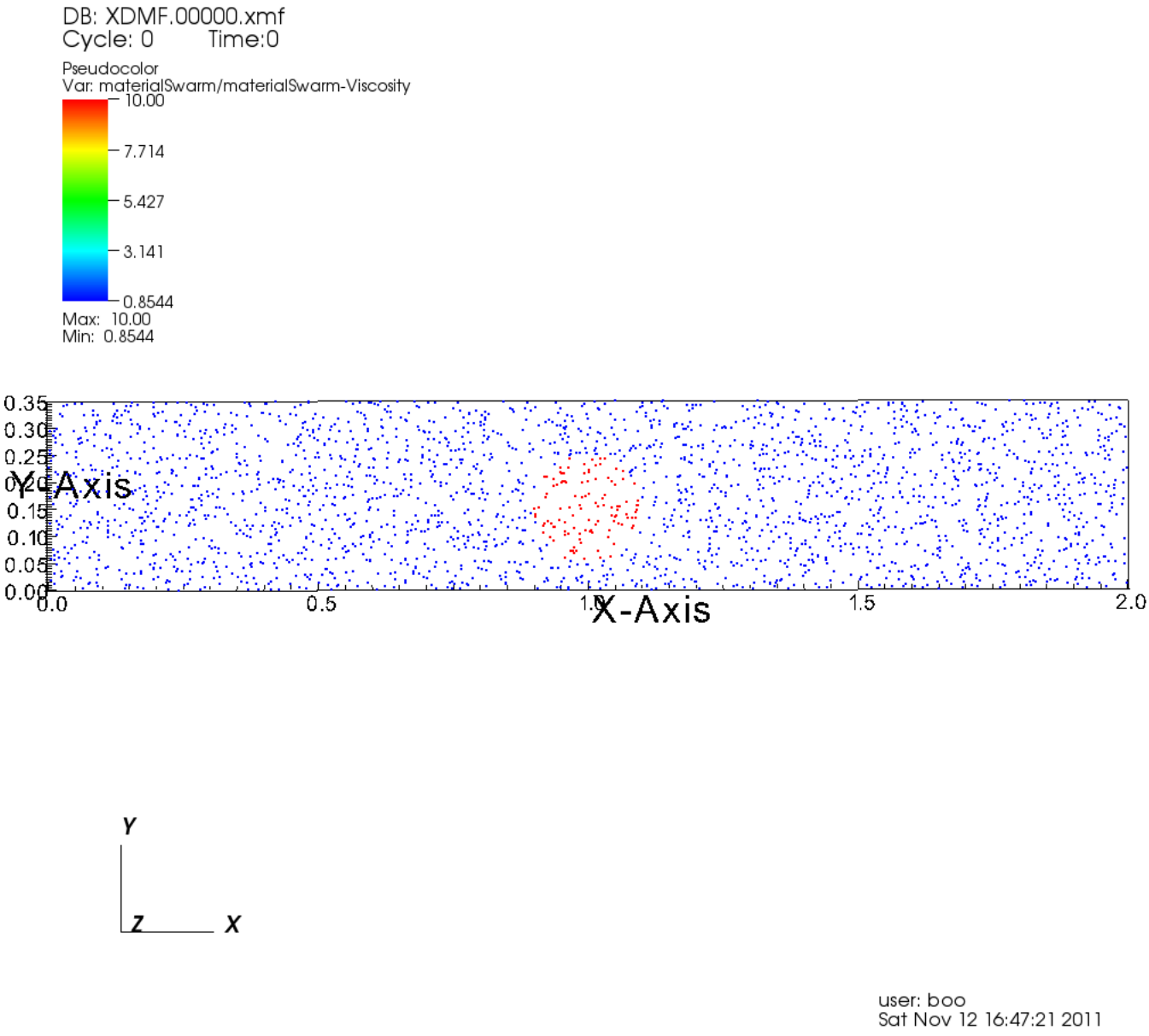
Click on the `Rendering` tab at the top. Under the `Color` section, select `Constant`.



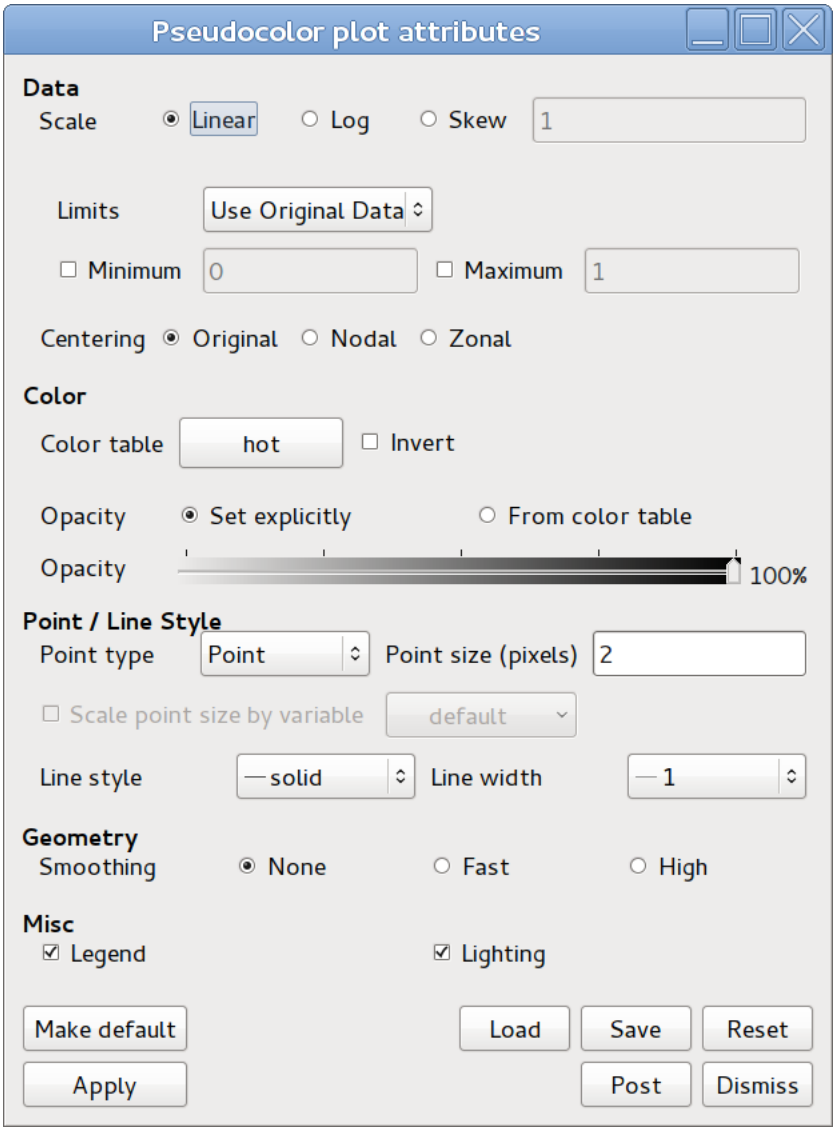
Click on `Apply` to apply the changes, then `Dismiss` to get rid of the `Vector plot attributes` window.



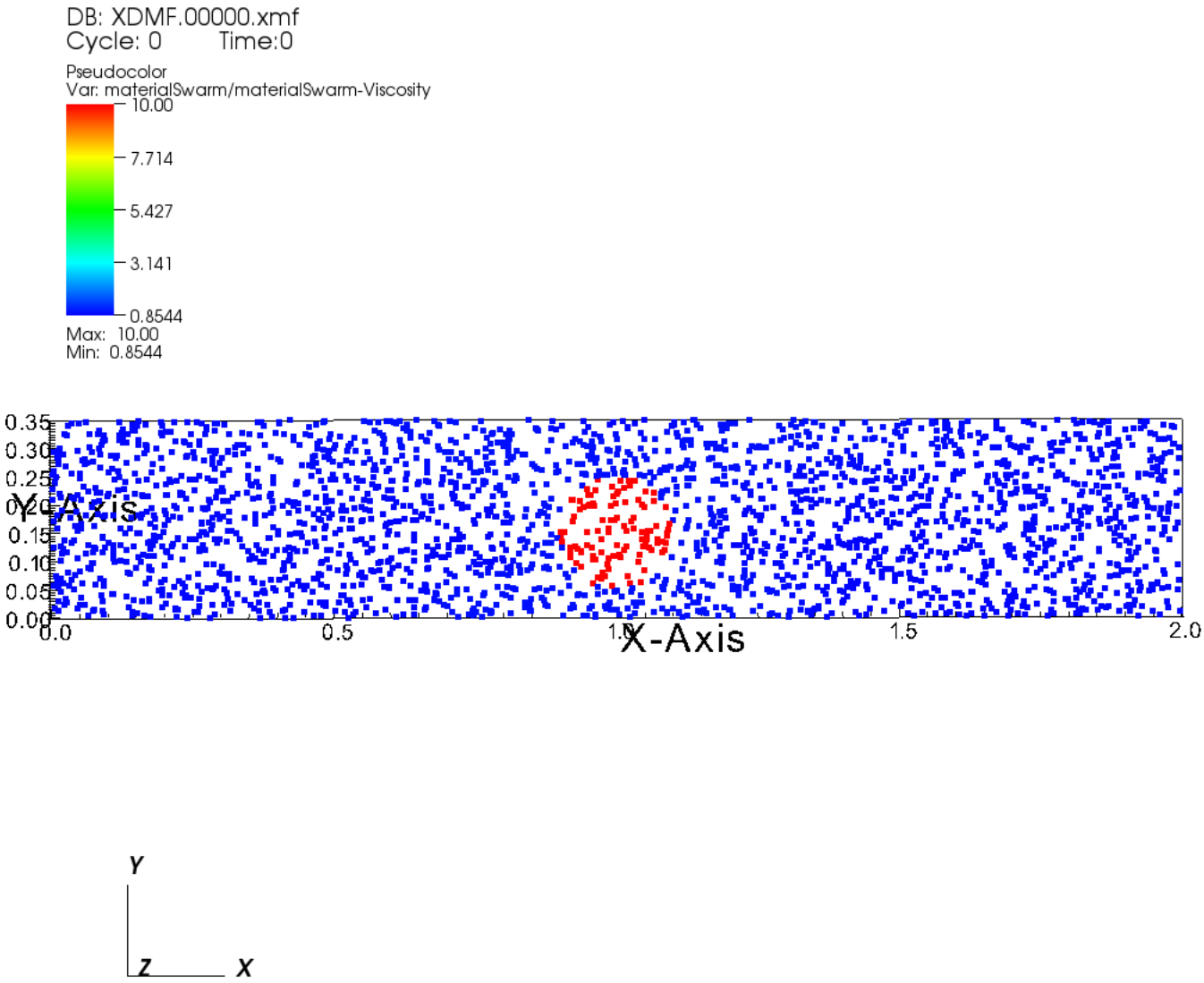
8. Now you can look at the particles. Hide the velocity arrows by clicking on `Vector - FEM_Grid_v-mesh/VelocityField` and then the `Hide/Show` button. Click the `Add` button, select `Pseudocolor`, then `materialSwarm`, then `materialSwarm-Viscosity`. As before, finish by clicking the `Draw` button.



9. The particles are a bit small to see. To increase their size, start by clicking on the `PlotAtts` menu item and selecting `Pseudocolor`. This brings up the `Pseudocolor plot attributes` window.

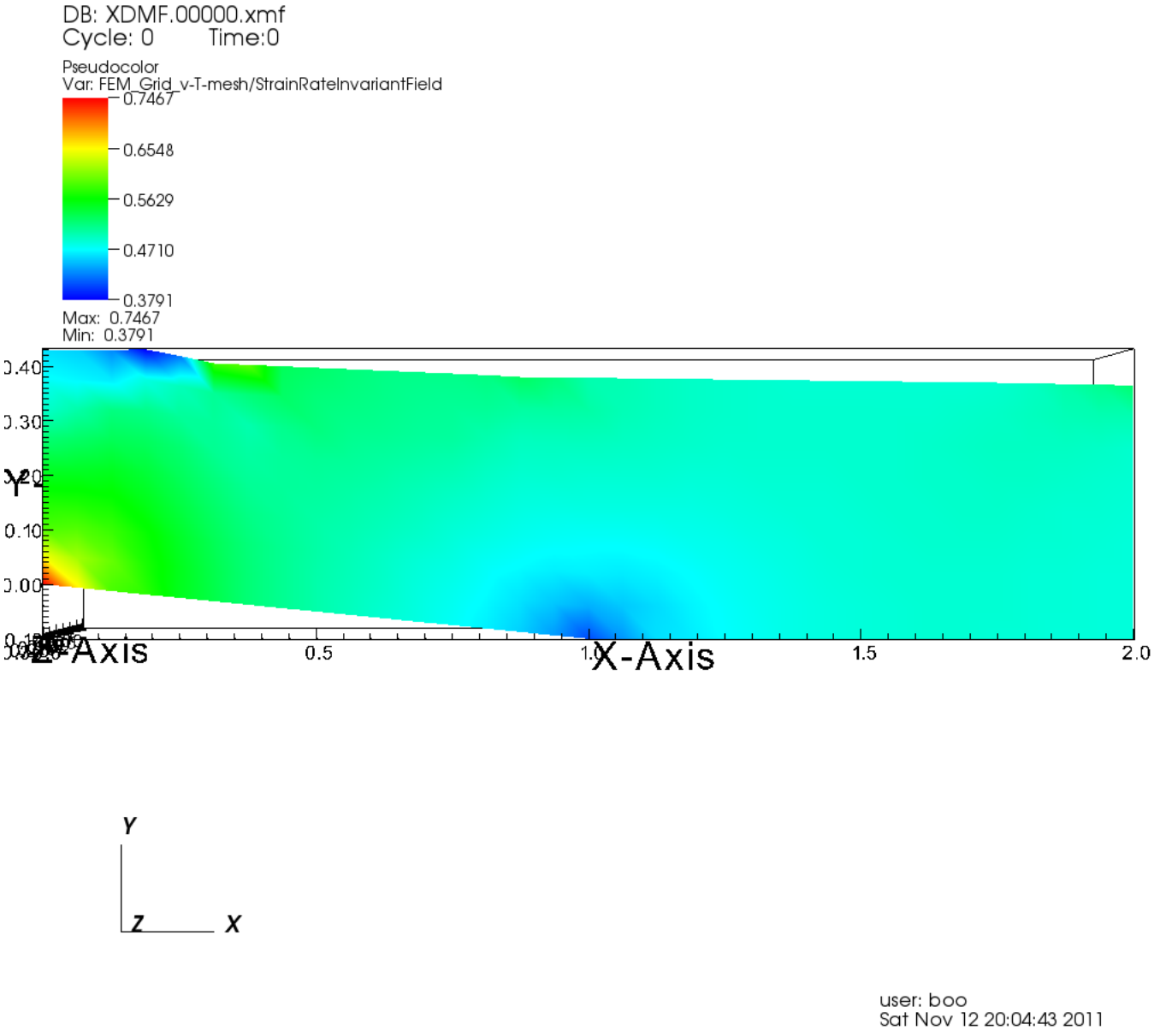


Change `Point size` to 5, press `Apply` and then `Dismiss` to get a view with larger particles.



This displays the viscosity of the particles. The red points represent the high viscosity ball, while the blue points represent material that has yielded. You can animate the view by pressing the arrow button  .

10. Now we will visualize a 3D simulation. First delete all of the existing plots by clicking the `Delete` button until they are all gone. Then click the `Close` button to get rid of the 2D plots. Run Gale with the input file `input/cookbook/extension3D.json` and open the XDMF files. As with the 2D input, add the strain rate invariant and click `Draw`.



Press and hold button 1 on your mouse to rotate the figure. Hold button 1 and the Control or Shift key to translate. Hold button 3 or use the mouse wheel to zoom in and out. To reset the view, press button 2 to bring up the context menu and select `Reset View`.

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 ensure 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 defined on a mesh (e.g., velocity, pressure, strain rate, etc.). The resolution of the grid is determined by `nx`, `ny`, and `nz`.

But sometimes the mesh resolution is not the principal source of error. For example, for the 2D Divergence benchmark (Section B.3), 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`. However, the solver can still initially settle on a wrong solution. Then, after many iterations, it may find the correct solution. To enforce this, you can set `nonLinearMinIterations` and the solver will keep iterating even if it thinks it has already found a solution.

It is also possible that the number of particles determines the error. 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 B.1, B.2), 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.

Chapter 5

Cookbooks

5.1 Introduction

In this chapter, you will edit a template file (`input/cookbook/template.`json) to create a series of input files. The template file is in JSON format ((`http://json.org`)). JSON is a lightweight data-interchange format that is easy for humans and machines to read and write.

5.1.1 Adding Lines to the Template File

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

```
"components":
{
```

at the beginning of the file, and before the matching brace just before `"velocityBCs"`.

All items are separated by commas ",". So if you are adding something to the end of a section, you will have to add a comma after the last item before adding your item. If you delete an item at the end, you must also delete the trailing comma. It is very easy to forget to add or delete a comma. If you do so, Gale should give you an error telling you what line the error is on.

The template file is indented to make it easier to for you to understand. This is solely for your benefit. Gale does not pay attention to indentation when reading the files. You may also add comments with a syntax like

```
// This is a comment
```

Everything on the line following '//' will be ignored.

5.1.2 Adding Variables to the Template File

When you are instructed to add a variable, add it at the end of the file before the closing bracket. As with components, if you add a variable at the end, you must first add a comma and then add the new item.

5.2 Viscous Material

This example simply fills up the computational domain with a single viscous material. It is a valid input file, but it is not very interesting 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.json` to `myviscous.json` to edit as follows.

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

```
,
"backgroundShape":
{
    "Type": "EquationShape",
    "equation": "1"
},
```

   `EquationShape` defines a shape to be wherever `equation`>=0. Since `equation`=1, that is true everywhere. Notice that we added a comma before `backgroundShape`. In anticipation of more items, we also added a comma after the closing brace of `backgroundShape`.

3. Then set the material's viscosity

```
"backgroundViscosity":
{
    "Type": "MaterialViscosity",
    "eta0": "1.0"
},
```

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

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

```
"viscous":
{
    "Type": "RheologyMaterial",
    "Shape": "backgroundShape",
    "density": "1.0",
    "Rheology": [
        "backgroundViscosity",
        "storeViscosity",
        "storeStress"
    ]
}
```

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

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

5.3 Viscous Material in Simple Extension

The input file you created in Section 5.2 is valid, but nothing moves. In this example, you will make the material extend by having the right boundary move.

1. Copy `myviscous.json` to `myextension.json`.

2. Make the right boundary move by changing the line after this section

```
"type": "WallVC",
"wall": "right",
"variables": [
  {
    "name": "vx",
```

from

```
"value": "0"
```

to

```
"value": "1.0"
```

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

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

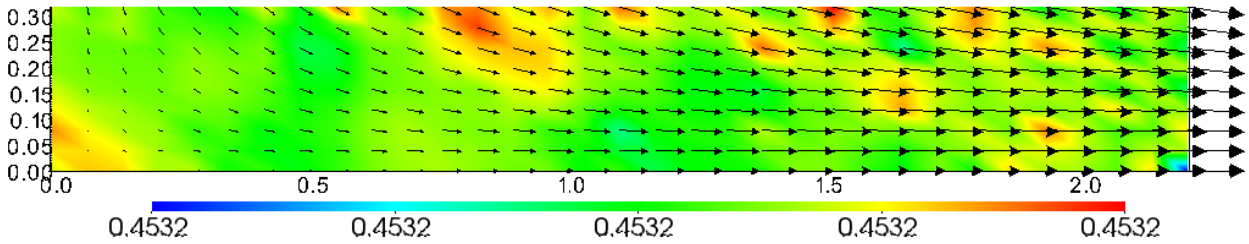


Figure 5.1: Strain rate invariant and velocity of viscous material in extension

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.

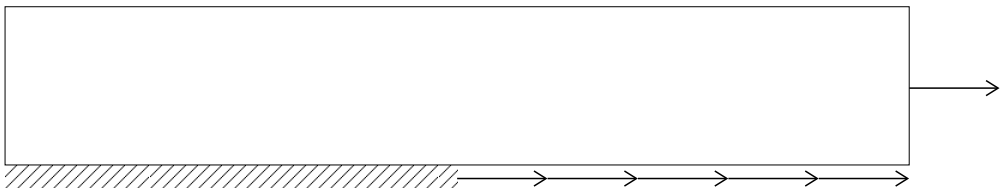


Figure 5.2: Split Boundary

1. First, copy `myextension.json` to `mysplit.json`
2. Modify the bottom boundary condition of WallVC to

```
"type": "WallVC",
"wall": "bottom",
"variables": [
    {
        "name": "vx",
        "value": "step(x-1)"
    },
    {
        "name": "vy",
        "value": "0.0"
    }
]
```

A worked example is in the file `input/cookbook/split.json`. 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.

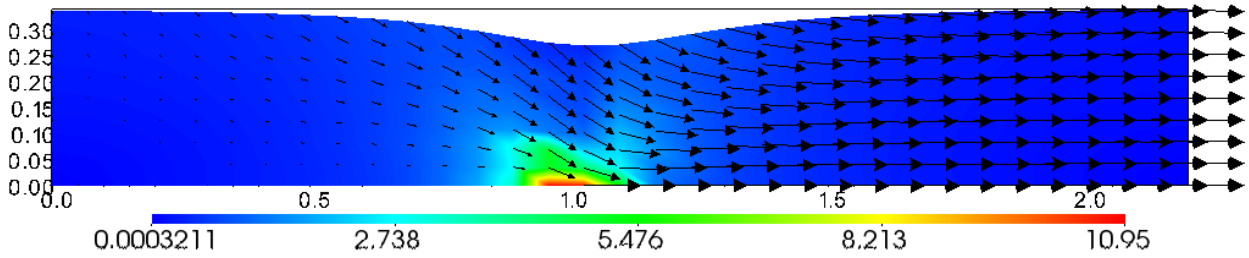


Figure 5.3: Strain rate invariant and velocity with complex boundaries

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 `myextension.json` to `myfile.json`
2. Modify the bottom boundary condition of WallVC to

```
"type": "WallVC",
"wall": "bottom",
"variables": [
    {
        "name": "vx",
        "type": "func",
        "value": "File1"
    },
    {
        "name": "vy",
        "value": "0"
```

```
        }
    ]
```

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

```
,"File1_Name": "input/cookbook/velocities",
"File1_Dim": "0",
"File1_N": "102"
```

to the end of the file (just before the last bracket "}").

There is a fully worked out example in `input/cookbook/file.json`.

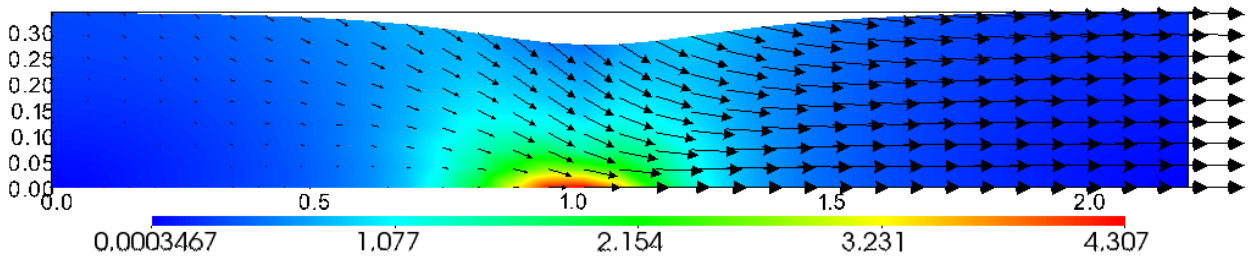


Figure 5.4: Strain rate invariant and velocity with boundary conditions read from a file

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 5.5. 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.

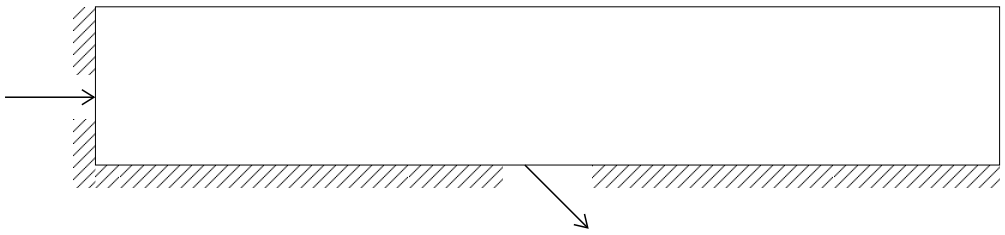


Figure 5.5: Inflow/Outflow Boundary

1. Copy the file `myviscous.json` that you created in Section 5.2 to `myinflow_outflow.json`.
2. Then, add the following lines after the `wrapTop` line so that Gale keeps the left and bottom sides fixed:

```
,"staticLeft" : "True",
"staticBottom" : "True"
```

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

```
"type": "WallVC",
"wall": "left",
"variables": [
    {
        "name": "vx",
        "value": "step(y-0.1)*step(0.2-y)"
    }
]
```

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

```
"type": "WallVC",
"wall": "bottom",
"variables": [
    {
        "name": "vx",
        "value": "step(x-0.9)*step(1.1-x)"
    },
    {
        "name": "vy",
        "value": "-step(x-0.9)*step(1.1-x)"
    }
]
```

A worked example is in the file `input/cookbook/inflow_outflow.json`. Figure 5.6 shows the strain rate invariant and velocity.

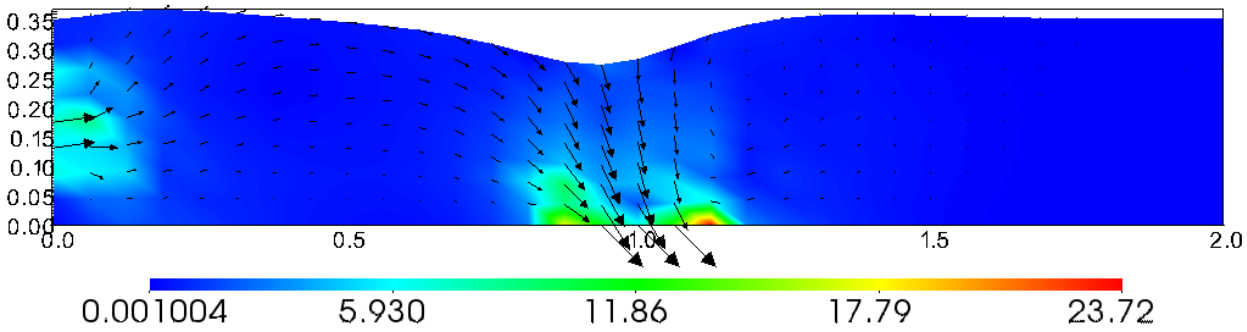


Figure 5.6: Strain rate invariant and velocity with inflow/outflow boundaries

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 `myextension.json` to `mynormal_stress.json`

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

```
},
{
  ”type”: ”WallVC”,
  ”wall”: ”bottom”,
  ”variables”: [
    {
        ”name”: ”vy”,
        ”value”: ”0”
    }
  ]
```

Notice that we removed the preceding bracket with comma "},” and left the trailing bracket "}".

3. Add in a StressBC component

```
,”stressBC”:
{
    ”Type”: ”StressBC”,
    ”ForceVector”: ”mom_force”,
    ”wall”: ”bottom”,
    ”normal_value”: ”0.35-y”
}
```

This force emulates a hydrostatic pressure which increases with depth. The height of the material above $y = 0$ is 0.35, and the density of the material is 1, so the stress needed to counteract gravity is $0.35 - y$.

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

```
”systems”: [
    {
        ”mesh”: ”v-mesh”,
        ”p-mesh”: ”p-mesh”,
        ”remesher”: ”velocityRemesher”,
        ”velocityField”: ”VelocityField”,
        ”wrapTop”: ”True”
```

the line

```
,”staticBottom”: ”True”
```

5. 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

```
,{
  ”name”: ”vy”,
  ”value”: ”0.0”
}
```

in two places: after

```
"type": "WallVC",
"wall": "left",
"variables": [
    {
        "name": "vx",
        "value": "0.0"
    },
```

and after

```
"type": "WallVC",
"wall": "right",
"variables": [
    {
        "name": "vx",
        "value": "1.0"
    },
```

A worked example is at `input/cookbook/normal_stress.json`. Figure 5.7 shows the strain rate invariant and velocity. Notice that material is now flowing in from the bottom.

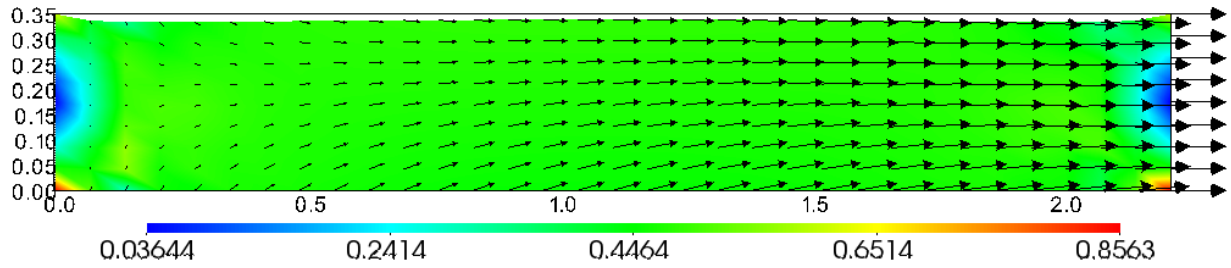


Figure 5.7: Strain rate invariant and velocity of viscous material in extension with a normal stress boundary

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

```
"staticBottom": "True"
```

to

```
"wrapBottom": "True"
```

A worked example is in `input/cookbook/deforming_bottom.json`. Figure 5.8 shows the strain rate invariant and velocity.

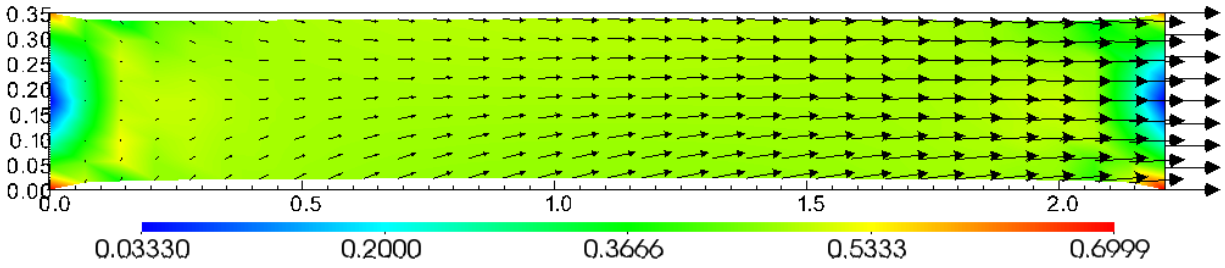


Figure 5.8: Strain rate invariant and velocity of viscous material with a deformable bottom boundary

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 5.9. This example has no moving boundaries, so the material will simply relax.

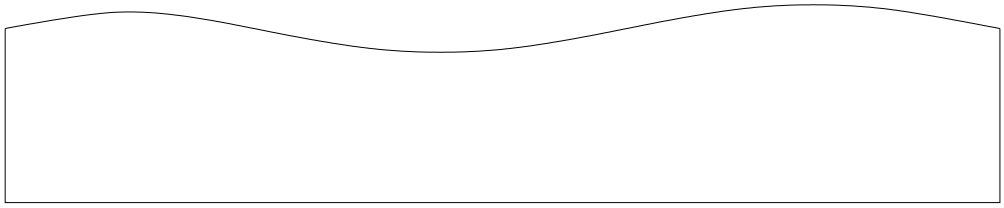


Figure 5.9: Sinusoidal Top

1. Copy `myviscous.json` to `mysinusoid.json`.
2. Add a `SurfaceAdaptor` component:

```
,"surfaceAdaptor":
{
    "Type": "SurfaceAdaptor",
    "mesh":"v-mesh",
    "sourceGenerator": "v-mesh-generator",
    "topEquation": "0.1*sin(2*pi*x)"
}
```

A worked example is in `input/cookbook/sinusoid.json`. Figures 5.10 and 5.11 shows the strain rate invariant and velocity (see Section 4.3.1) at the beginning and after the tenth timestep. Note that the material has flattened out and the magnitude of the velocity and strainrate has reduced considerably.

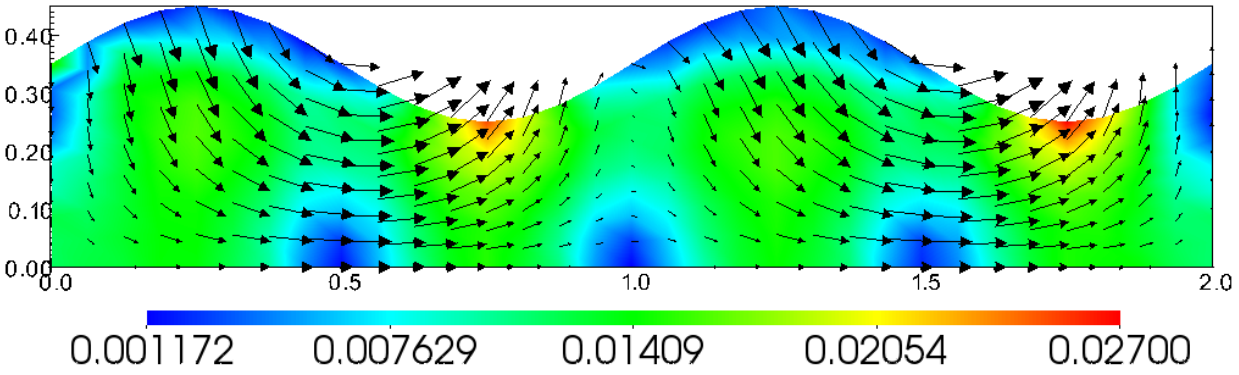


Figure 5.10: Strain rate invariant and velocity with initially deformed upper boundary

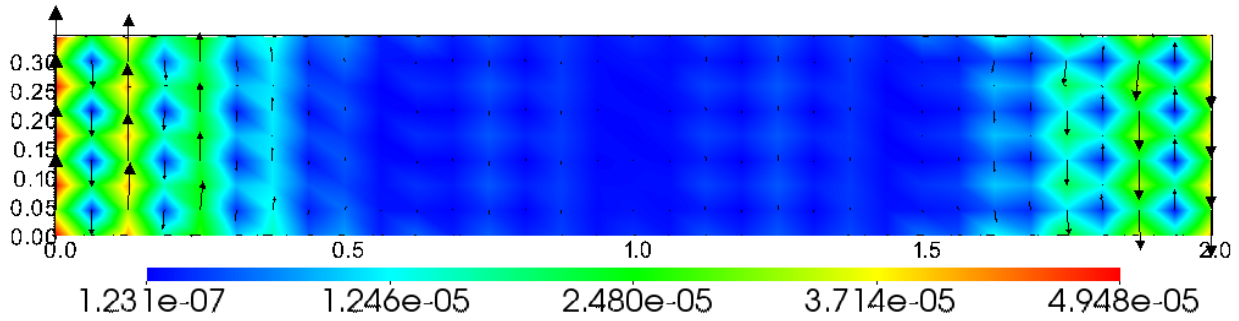


Figure 5.11: Strain rate invariant and velocity with initially deformed upper boundary

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 5.12. This is meant to approximate one slab subducting under another.

1. Copy `myinflow_outflow.json` to `myfixed_bottom.json`
2. Add a `SurfaceAdaptor` component for the bottom boundary:

```
,"surfaceAdaptor":
{
    "Type": "SurfaceAdaptor",
    "mesh": "v-mesh",
    "sourceGenerator": "v-mesh-generator",
    "bottomEquation": "step(0.960468635615-x)*(-3 + sqrt(3.15*3.15 - x*x))"
}
```

3. In the boundary conditions, replace

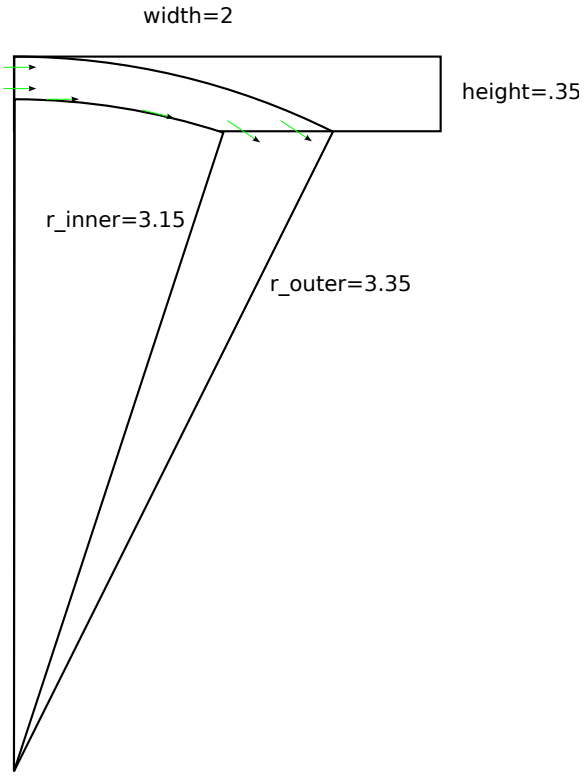


Figure 5.12: Geometry and boundary conditions for the fixed, deformed bottom boundary

```
    ”value”: ”step(y-0.1)*step(0.2-y)”
```

with

```
    ”value”: ”r=hypot(x,y+3), step(3.35-r)*(y+3)”
```

and replace

```
    ”type”: ”WallVC”,
    ”wall”: ”bottom”,
    ”variables”: [
        {
            ”name”: ”vx”,
            ”value”: ”step(x-0.9)*step(1.1-x)”
        },
        {
            ”name”: ”vy”,
            ”value”: ”-step(x-0.9)*step(1.1-x)”
        }
    ]
```

with

```
    ”type”: ”WallVC”,
    ”wall”: ”bottom”,
    ”variables”: [
        {
            ”name”: ”vx”,
```

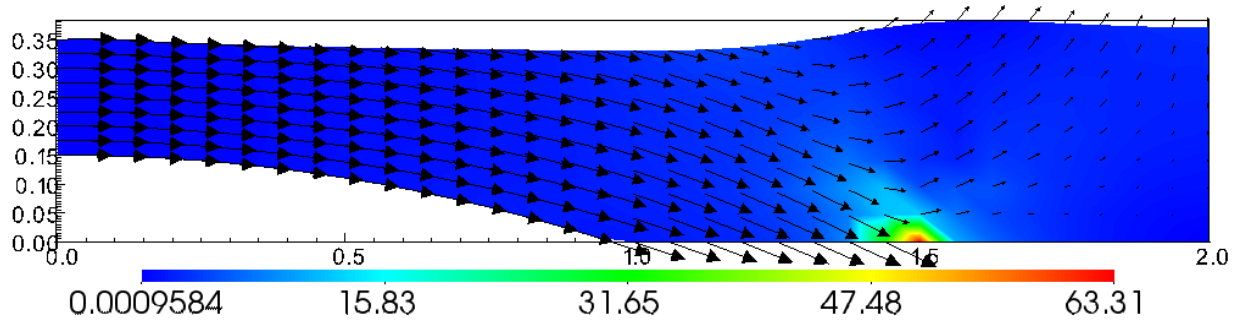


Figure 5.13: Strain rate invariant and velocity for a deformed bottom boundary

```
            "value": "r=hypot(x,y+3), step(3.35-r)*(y+3)"
        },
        {
            "name": "vy",
            "value": "r=hypot(x,y+3), -step(3.35-r)*x"
        }
    ]
```

4. Ensure that the height of the incoming material remains fixed by adding

   ```
   "staticLeftTop" : "True",
   ```

   in the `EulerDeform` struct, right after

   ```
   "staticLeft" : "True",
   ```

A worked example is in `input/cookbook/fixed_bottom.json`. Figure 5.13 shows the strain rate invariant and velocity.

5.11 Extension in 3D with topography

This example extends the simulation into 3D, adding initial topography and a deformed bottom.

1. Copy `myextension.json` to `myextension3D.json`.
2. Add a SurfaceAdaptor component

```
,"surfaceAdaptor":
{
    "Type": "SurfaceAdaptor",
    "mesh":"v-mesh",
    "sourceGenerator": "v-mesh-generator",
    "topSurfaceType": "topo_data",
    "topSurfaceName": "input/cookbook/test.topo",
    "topNx": "32",
    "topNz": "12",
    "topMinX": "minX",
    "topMaxX": "maxX",
    "topMinZ": "minZ",
```

```
    "topMaxZ": "maxZ",

    "bottomEquation": "x<1 ? -0.1*x : -0.1"
}
```

This component reads in data from the file `test.topo` to set the initial height. It also sets the bottom to have a slope that flattens out.

3. Add velocity conditions for the front and back

```
{
    "type": "WallVC",
    "wall": "front",
    "variables": [
        {
            "name": "vz",
            "value": "0.0"
        }
    ]
},
{
    "type": "WallVC",
    "wall": "back",
    "variables": [
        {
            "name": "vz",
            "value": "0.0"
        }
    ]
},
```

4. Change `dim` from 2 to 3.

A worked example is in `input/cookbook/extension3D.json`. Figure 5.14 shows the strain rate invariant and velocity.

5.12 Tracers

This example adds tracer particles to track where material moves. These tracers play no active part in the simulation, only observing the fields as they follow the movements of the material.

1. Copy `myfixed_bottom.json` to `tracers.json`.
2. Enable tracers by adding

```
”enable-tracers”: true,
```

after

```
”FieldVariablesToCheckpoint”: [
    ”StrainRateInvariantField”,
    ”VelocityField”,
    ”PressureField”
],
```

3. Add a component laying out the initial positions of the particles

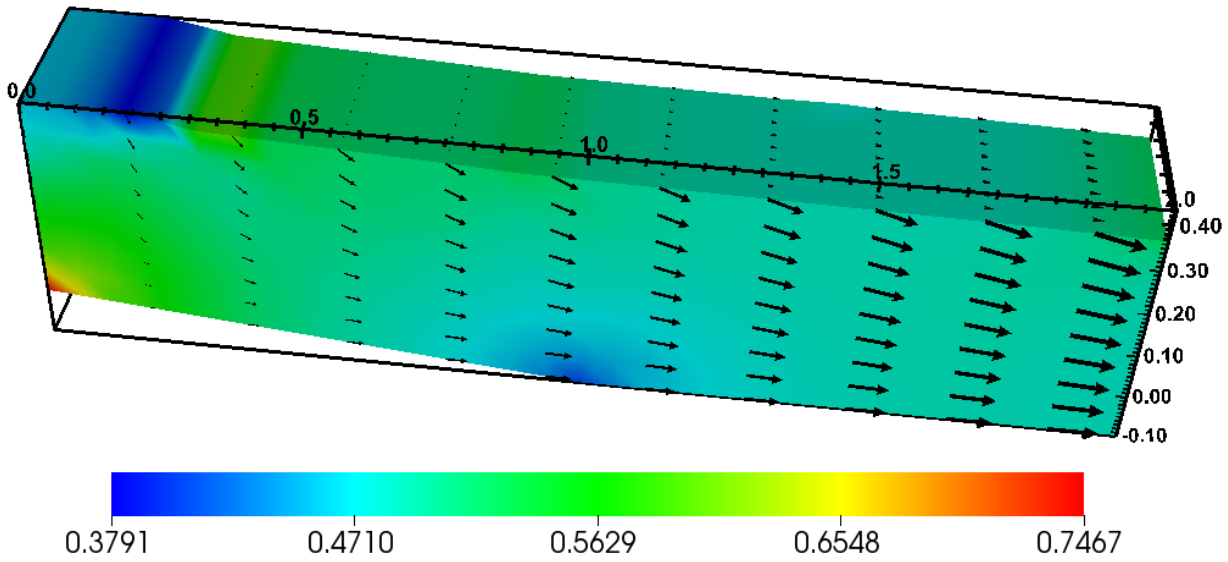


Figure 5.14: Strain rate invariant and velocity for a deformed bottom boundary

```
,"pLayout":
{
    "Type": "ManualParticleLayout",
    "manualParticlePositions": [
        "asciidata",
        ["x", "y"],
        1.0, .1,
        1.3, .1,
        1.6, .1,
        1.9, .1,
        1.0, .2,
        1.3, .2,
        1.6, .2,
        1.9, .2
    ]
},
```

and another component for controlling what fields are output

```
"swarmOutput":
{
    "Type": "TracerOutput",
    "Swarm": "passiveTracerSwarm",
    "Fields" : [
        "PressureField",
        "StrainRateInvariantField"
    ]
}
```

4. In order to see nice tracks, increase the number of timesteps by changing the line

```
"maxTimeSteps": "10",
```

to

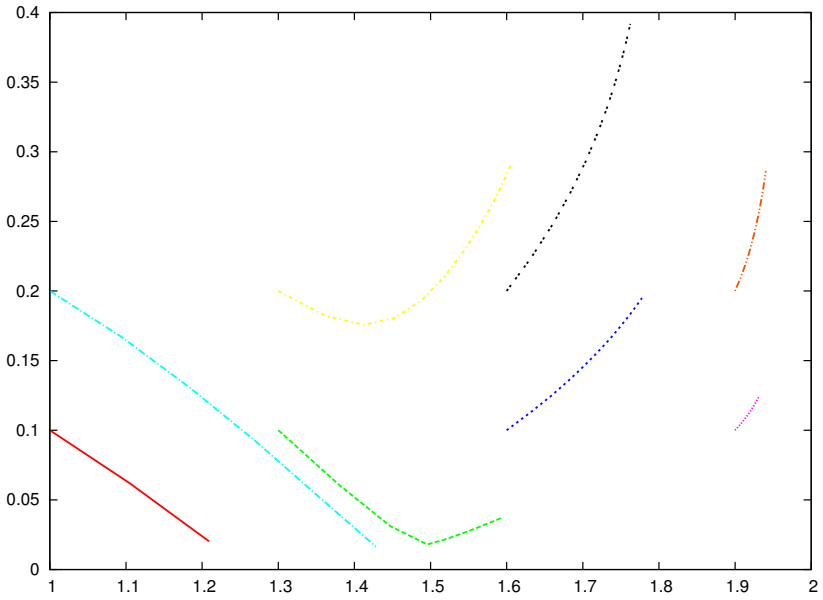


Figure 5.15: Particle tracks of tracers

```
"maxTimeSteps": "100",
```

After running this input file you will see eight new files in the output directory: `swarmOutput.00000.dat`, ... `swarmOutput.00007.dat`. Inside each of these files is a record of the time, position, pressure, and strain-rate invariant that each particle saw as it traveled along. Plotting the particle tracks of all of these tracers gives us Figure 5.15.

5.13 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 5.16.

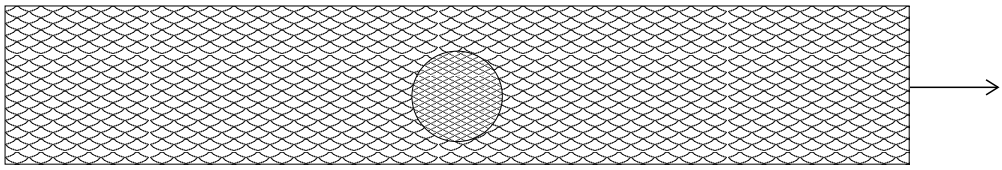


Figure 5.16: Multiple Viscous Materials

1. Copy `myextension.json` (see Section 5.3) to `mymulti_material.json`.
2. Add the sphere.

```
"sphereShape":
{
    "Type": "EquationShape",
    "equation": ".1^2 - ((x-1)^2 + (y-.15)^2)"
},
```

3. Then add the new material.

```
,''sphereViscosity'':
{
    ''Type'': ''MaterialViscosity'',
    ''eta0'': ''10.0''
},
''sphereViscous'':
{
    ''Type'': ''RheologyMaterial'',
    ''Shape'': ''sphereShape'',
    ''density'': ''1.0'',
    ''Rheology'': [
        ''sphereViscosity'',
        ''storeViscosity'',
        ''storeStress''
    ]
}
```

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:

```
''nonsphereShape'':
{
    ''Type'': ''Intersection'',
    ''shapes'':
    [
        ''backgroundShape'',
        ''!sphereShape''
    ]
},
```

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

```
''viscous'':
{
    ''Type'': ''RheologyMaterial'',
```

   from

```
''Shape'': ''backgroundShape'',
```

   to

```
''Shape'': ''nonsphereShape'',
```

A worked example is in `input/cookbook/multi_material.json`. Figure 5.17 shows the strain rate invariant and velocity,

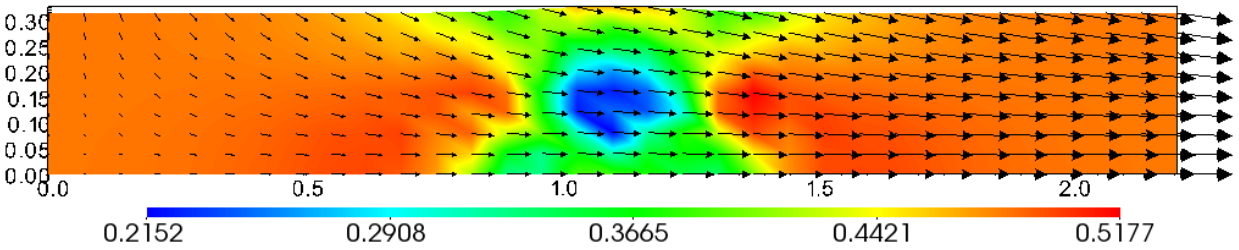


Figure 5.17: Strain rate invariant and velocity with multiple viscous materials

and Figure 5.18 shows the viscosity of the particles.

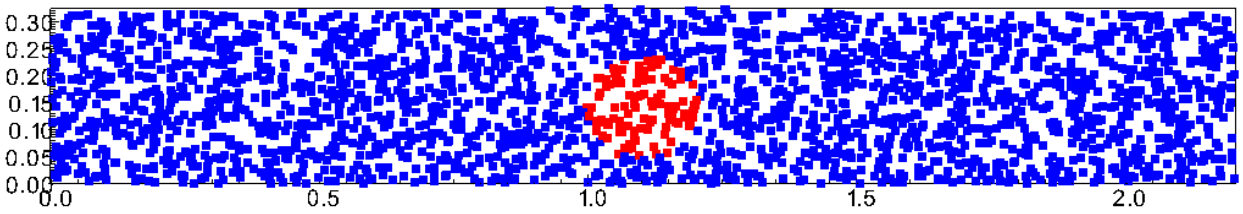


Figure 5.18: Viscosities with multiple viscous materials

5.14 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.json` to `myyielding.json`
2. Add a `StrainWeakening` component and a `DruckerPrager` component

```
"strainWeakening":
{
    "Type": "StrainWeakening",
    "TimeIntegrator": "timeIntegrator",
    "MaterialPointsSwarm": "materialSwarm",
    "softeningStrain": "0.1",
    "initialDamageFraction": "0.0",
    "initialDamageWavenumber": "0.5",
    "initialDamageFactor": "0.5",
    "healingRate": "0.0"
},
"yielding":
{
    "Type": "DruckerPrager",
    "PressureField": "PressureField",
    "VelocityGradientsField": "VelocityGradientsField",
    "MaterialPointsSwarm": "materialSwarm",
    "Context": "context",
    "StrainWeakening": "strainWeakening",
```

```
        "StrainRateField": "StrainRateField",
        "cohesion": "1.0",
        "cohesionAfterSoftening": "0.0001",
        "frictionCoefficient": "0.0",
        "frictionCoefficientAfterSoftening": "0.0",
        "minimumViscosity": "1.0e-4"
    },
```

after `backgroundViscosity`.

3. Add this yielding rheology to the existing background material by inserting

```
"yielding",
```

after

```
"viscous":
{
    "Type": "RheologyMaterial",
    "Shape": "nonsphereShape",
    "density": "1.0",
    "Rheology": [
        "backgroundViscosity",
```

A worked example is in `input/cookbook/yielding.json`. Figure 5.19 shows the strain rate invariant and velocity. A fault has developed on the left side.

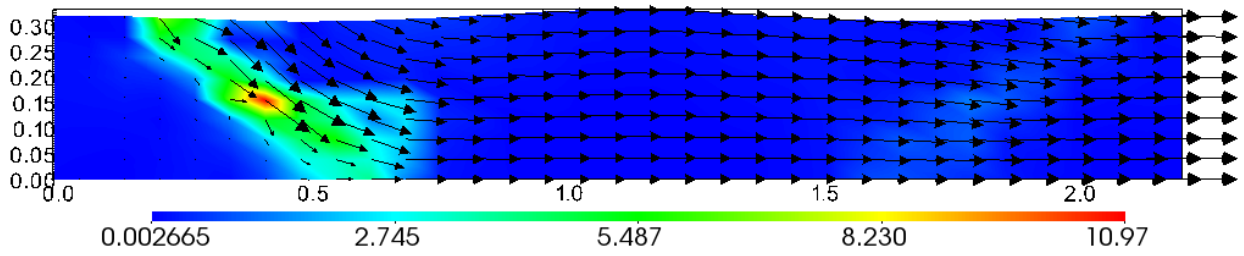


Figure 5.19: Strain rate invariant and velocity of yielding material in extension

Figure 5.20 shows the viscosity of the particles,

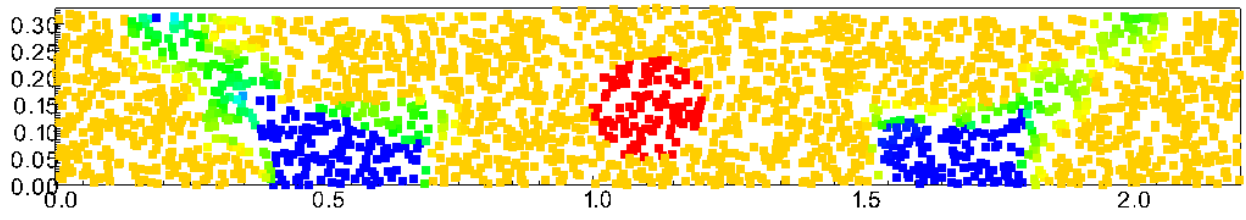


Figure 5.20: Viscosity of yielding material in extension

and Figure 5.21 shows the accumulated post-yielding strain of the particles.

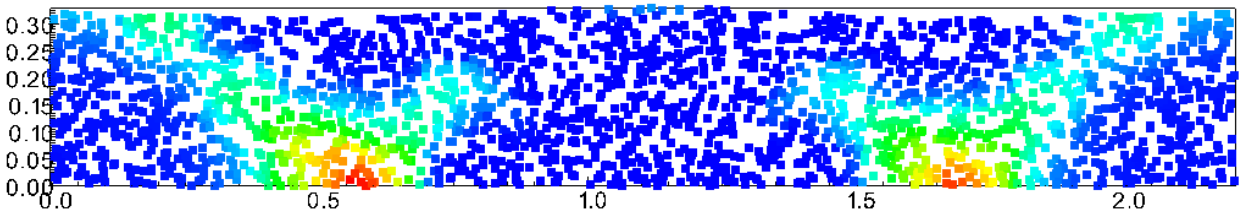


Figure 5.21: Accumulated post-yielding strain of yielding material in extension

5.15 Thermal Convection

Temperature can play a decisive role in geophysical processes. This example takes the multiple viscous material example from Section 5.13, heats it on the bottom, and adds in radiogenic heating throughout.

1. Copy `mymulti_material.json` to `mythermal.json`
2. Enable the thermal components by adding

   ```
   "enable-thermal": true,
   ```

   after

   ```
   "FieldVariablesToCheckpoint": [
       "StrainRateInvariantField",
       "VelocityField",
       "PressureField"
   ],
   ```

3. Add in temperature boundary conditions after the velocity boundary conditions

   ```
   "temperatureBCs": {
       "type": "CompositeVC",
       "vcList": [
           {
               "type": "WallVC",
               "wall": "left",
               "variables": [
                   {
                       "name": "temperature",
                       "value": "1.0"
                   }
               ]
           },
           {
               "type": "WallVC",
               "wall": "right",
               "variables": [
                   {
                       "name": "temperature",
   ```

```
                    "value": "1.0"
                }
            ]
        },
        {
            "type": "WallVC",
            "wall": "top",
            "variables": [
                {
                    "name": "temperature",
                    "value": "1.0"
                }
            ]
        },
        {
            "type": "WallVC",
            "wall": "bottom",
            "variables": [
                {
                    "name": "temperature",
                    "value": "2.0"
                }
            ]
        }
    ]
},
```

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

```
"temperatureICs":
{
    "type": "CompositeVC",
    "vcList": [
        {
            "type": "AllNodesVC",
            "variables":[
                {
                    "name": "temperature",
                    "value": "1.0"
                }
            ]
        }
    ]
},
```

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

```
"viscous":
{
    "Type": "RheologyMaterial",
    "Shape": "nonsphereShape",
    "density": "1.0",
```

the lines

```
"alpha": "1.0",
"diffusivity": "1.0",
"heatingElements": [
    {
        "Q": "1.0",
        "lambda": "1.0"
    }
],
```

For the sphere, after the lines

```
"sphereViscous":
{
    "Type": "RheologyMaterial",
    "Shape": "sphereShape",
    "density": "1.0",
```

add the lines

```
"alpha": "10.0",
"diffusivity": "10.0",
"heatingElements": [
    {
        "Q": "1000.0",
        "lambda": "10.0"
    }
],
```

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

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

```
,"TemperatureField": "TemperatureField"
```

after the lines

```
"buoyancyForceTerm":
{
    "Type": "BuoyancyForceTerm",
    "ForceVector": "mom_force",
    "Swarm": "gaussSwarm",
    "gravity": "gravity"
```

7. The deforming mesh requires some adjustments to the advection terms. Enable this by adding

```
"T-mesh": "T-mesh",
"displacementField": "DisplacementField",
```

after

```
"EulerDeform":
{
    "systems": [
        {
            "mesh": "v-mesh",
            "p-mesh": "p-mesh",
```

8. Add temperature as a checkpoint variable by inserting

```
"TemperatureField",
```

after

```
"FieldVariablesToCheckpoint": [
    "StrainRateInvariantField",
    "VelocityField",
```

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

```
"type": "WallVC",
"wall": "right",
"variables": [
    {
        "name": "vx",
```

from

```
"value": "1.0"
```

to

```
"value": "0.01"
```

A worked example is in `thermal.json`. Figure 5.22 shows the temperature and velocity.

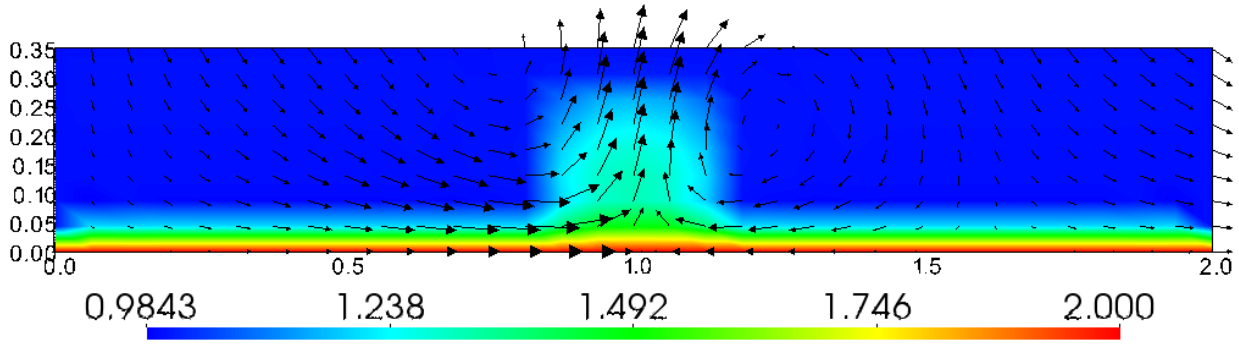


Figure 5.22: Temperature and velocity for the thermal convection example

5.16 Thermal Convection with Initial Conditions from a File

There are a number of different functions that you can use as initial conditions for the temperature (see Appendix A.11). This example shows how to use data from a file as your initial condition. The data used for this file is in `input/cookbook/temperature`, and sets the initial temperature inside the box to

$$1 + 0.05\cos(6x)\cos(10y).$$

1. Copy `mythermal.json` to `mythermal_file.json`.

2. In the `temperatureICs` struct, change the line after

```
"type": "AllNodesVC",
"variables":[
    {
        "name": "temperature",
```

from

```
"value": "1.0"
```

to

```
"type": "func",
"value": "File1"
```

3. Add in the lines

```
,"File1_Name": "input/cookbook/temperatures",
"File1_Dim": "0",
"File1_N": "202",
"File1_Dim2": "1",
"File1_N2": "37"
```

at the end of the file, just before the last bracket "}".

4. Increase the vertical resolution a little by changing

```
"ny": "4",
```

to

```
"ny": "8",
```

A worked example is in `thermal_file.json`. Figure 5.23 shows the temperature and velocity at the end of the calculation.

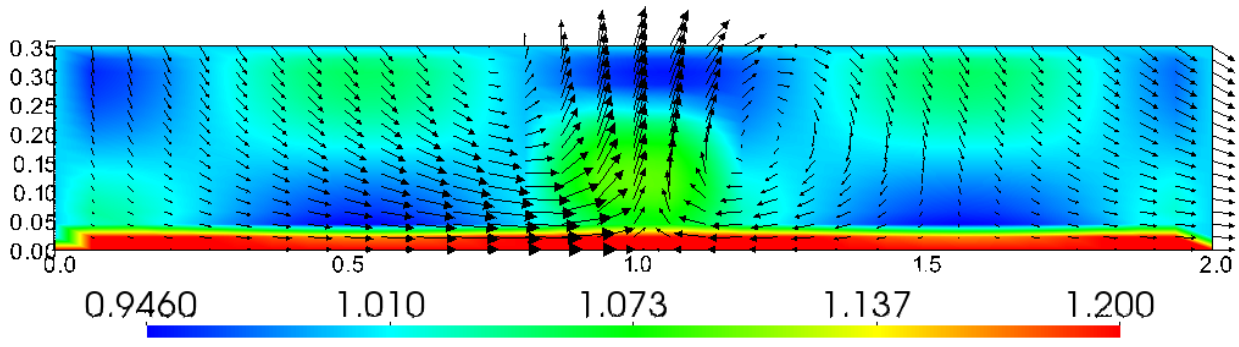


Figure 5.23: Temperature and velocity when using temperature initial data from a file.

5.17 Pure Thermal

This example turns off the Stokes solver and only evolves the temperature. Since the Stokes equations are not solved, the velocity must be specified independently. This also means that variables like strain rate and viscosity are no longer needed.

1. Copy `mythermal.json` to `mythermal_only.json`.

2. Delete the `buoyancyForceTerm` component.

3. Remove the references to viscosity and stress in the materials by changing the `viscous` component to

```
"viscous":
{
    "Type": "Material",
    "Shape": "nonsphereShape",
    "diffusivity": "1.0",
    "heatingElements": [
        {
            "Q": "1.0",
            "lambda": "1.0"
        }
    ]
},
```

   and change the `sphereViscous` component to

```
"sphereViscous":
{
    "Type": "Material",
    "Shape": "sphereShape",
    "diffusivity": "10.0",
    "heatingElements": [
        {
            "Q": "1000.0",
            "lambda": "10.0"
        }
    ]
}
```

4. Replace the velocity boundary condition `velocityBCs` with

```
”velocityICs”:
{
    ”type”: ”CompositeVC”,
    ”vcList”: [
        {
            ”type”: ”AllNodesVC”,
            ”variables”:[
                {
                    ”name”: ”vx”,
                    ”value”: ”0.0”
                },
                {
                    ”name”: ”vy”,
                    ”value”: ”0.0”
```

```
                    }
                ]
            }
        ]
    },
```

5. Remove the strain rate and pressure as checkpointed variables by replacing

```
"FieldVariablesToCheckpoint": [
    "StrainRateInvariantField",
    "VelocityField",
    "TemperatureField",
    "PressureField"
],
```

with

```
"FieldVariablesToCheckpoint": [
    "VelocityField",
    "TemperatureField"
],
```

6. Disable Stokes by adding

```
"enable-stokes": false,
```

right after

```
"enable-thermal": true,
```

Figure 5.24 shows the temperature at the end of the calculation.

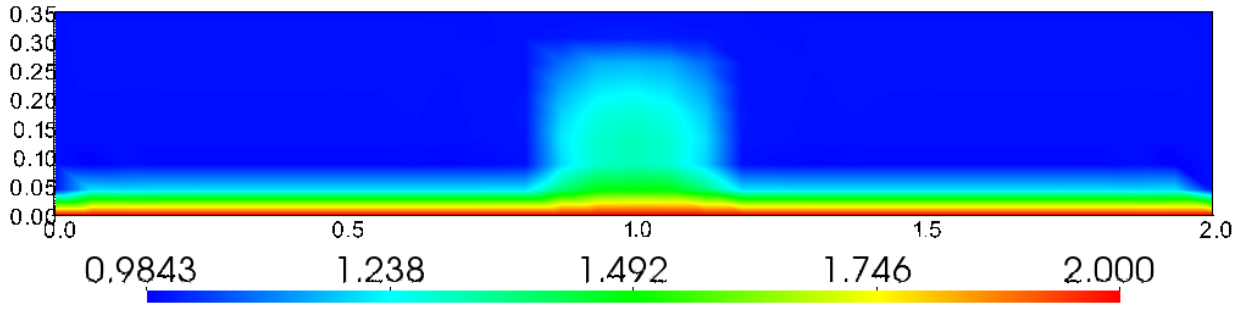


Figure 5.24: Temperature and velocity when using temperature initial data from a file.

5.18 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.4.2.4.

1. Copy `mythermal.json` to `mynon_newtonian.json`.

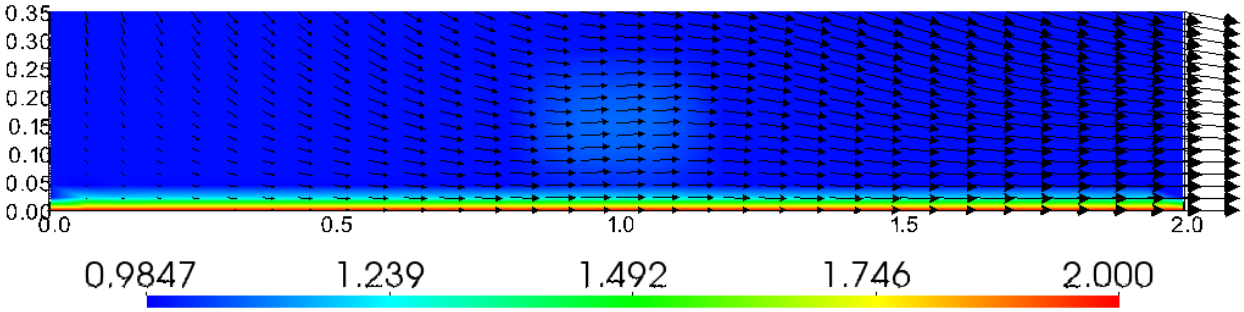


Figure 5.25: Temperature and velocity for the power-law creep model

2. Replace the `backgroundViscosity` component with

```
”nonNewtonian”:
{
    ”Type”: ”NonNewtonian”,
    ”StrainRateInvariantField”: ”StrainRateInvariantField”,
    ”TemperatureField”: ”TemperatureField”,
    ”n”: ”3.4”,
    ”T_0”: ”1.0”,
    ”A”: ”1.0”,
    ”refStrainRate”: ”0.01”
},
```

3. In the `viscous` material, change `backgroundViscosity` to `nonNewtonian`.

A worked example is in `non_newtonian.json`. Figure 5.25 shows the temperature and velocity. The differences with the example in Figure 5.22 are mostly because the viscosity is higher everywhere.

Appendix A

Input File Format

A.1 Structure

The input files are in the JSON format ((`http://json.org`)). This leverages a well-known format to specify concepts like hierarchies, lists, parameters, and arbitrary structures. The entire file is enclosed within brackets "{" and "}". Within those brackets, there are four parts of a Gale input file: components, EulerDeform, variable conditions, and variables.

Internally, the JSON input files are converted to XML and then parsed. During that process, a number of components are added by default, such as the mesh, the velocity field, and particles. Gale writes out this XML into the file `input.xml` in the output directory. If you create your own version of these default components in your JSON input file, your versions will take precedence.

Previous versions of Gale used XML as the input file format, and this scheme allows Gale to accept either JSON or XML as input. Existing XML input files will still work with minor modifications. See the file UPGRADE for the details of these modifications.

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 which material goes where, what the material properties are, 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

```
''components'':
{
  ''sphereShape'':
  {
    ''Type'': ''EquationShape'',
    ''equation'': ''.1^2 - ((x-1)^2 + (y-.15)^2)''
  }
}
```

will correctly initialize `sphereShape`, but the input file

```
''components'':
{
},
  ''sphereShape'':
  {
    ''Type'': ''EquationShape'',
```

```
        "equation": ".1^2 - ((x-1)^2 + (y-.15)^2)"
    }
```

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

A.1.2 EulerDeform

`EulerDeform` allows the upper surface to move freely or stay rigidly in place. If you do not have an `EulerDeform` struct, then the mesh will not deform. An example `EulerDeform` struct is

```
"EulerDeform":
{
    "systems": [
        {
            "mesh": "v-mesh",
            "p-mesh": "p-mesh",
            "remesher": "velocityRemesher",
            "velocityField": "VelocityField",
            "wrapTop": "True"
        }
    ]
},
```

Note the critical line

```
"wrapTop": "True"
```

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`

```
"staticRight": "True"
```

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

Setting staticRight will make the right boundary immobile. You can make the whole boundary move with a fixed velocity, by setting `right_equation`. So setting

```
"right_equation": "10 - 0.01 * t"
```

will make the right side move from 10 inwards with a velocity 0.1. Similarly, you can make the left side move by setting `left_equation`. If you are scaling units as in Section 2.2.8.3, be sure to scale the velocity here.

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.4.3.3), and you want the height of the boundary layer to match the interior.

In general, Gale has three different meshes: velocity, pressure, and temperature. Pure Stokes flow only has velocity and pressure meshes. Pure thermal flow only has velocity and temperature meshes. The active

ones must be supplied to EulerDeform using the `v-mesh`, `p-mesh`, and `T-mesh` variables. Unless you change something, these will be `v-mesh`, `p-mesh`, and `T-mesh`.

In addition, the energy equation (2.12) is an advection-diffusion equation. When the mesh distorts, the advection needs to be modified for consistency. When you enable thermal evolution, Gale automatically creates `DisplacementField`. Set `displacementField` to `DisplacementField` and Gale will make the necessary corrections for advection.

Defaults	
velocityField	-
v-mesh	-
p-mesh	-
T-mesh	-
DisplacementField	-
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 initial and boundary conditions for the velocity and temperature. See Sections A.5.1, A.5.4, and A.8 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. 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.

enable-stokes Enable solution of the Stokes equations. The default is true.

enable-thermal Enable temperature evolution. The default is false.

enable-tracers Enable tracer particles. The default is false.

checkPointEvery How often to write the checkpoint files (see Section 4.2.4).

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.5.5).

nx,ny,nz The number of elements in each direction. Note that the number of grid points depends on the type of element. The pressure mesh uses discontinuous linear elements (P_{-1}) which have three grid points per element in 2D and four grid points in 3D. So if nx=16, ny=32, then there will be 16*32*3=1536 pressure grid points. The temperature mesh uses linear elements (Q_1) which have their grid points on the corners. So the number of grid points is one larger than the number of elements (e.g., 64 elements $\Rightarrow$ 65 grid points). Finally, the velocity mesh uses quadratic elements (Q_2) which has grid points at the corners and in between. So if nx=16, ny=32, there will be (16*2+1)*(32*2+1)=2145 velocity 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.

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 will probably never need to change this from the default (40).

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 shrinks 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.2.

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.

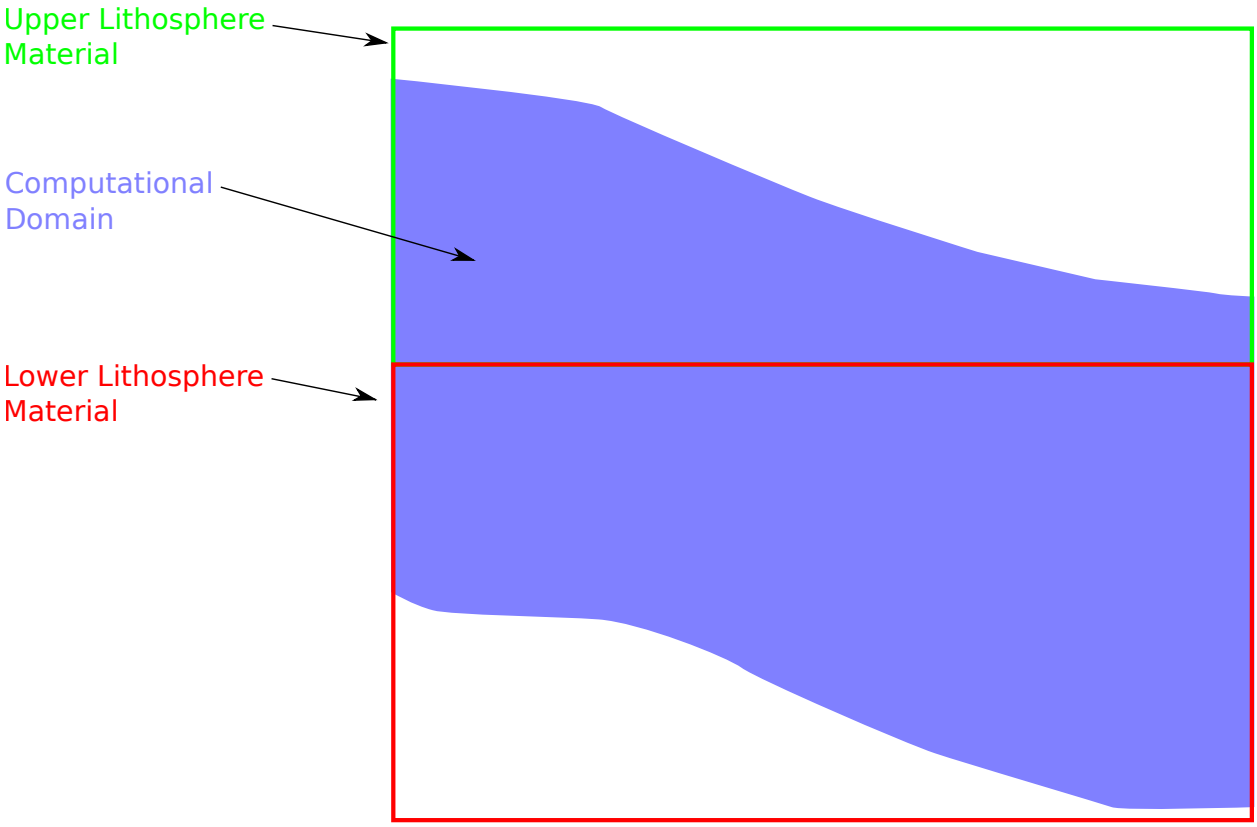


Figure A.1: Areas covered by material box shapes and the computational domain.

A.2 Temperature components

To enable temperature evolution, set the variable `enable-thermal` to `true`. You will also need to enable checkpointing for the temperature by adding the line `TemperatureField` to the list `FieldVariablesToCheckpoint`.

You need to specify the thermal diffusivity. You can specify a single diffusivity for all materials by setting the variable `defaultDiffusivity`. You can override this default for each material (see Section A.4).

You will also need to add in initial and boundary conditions (see Sections A.5.4 and A.8). Finally, you will need to set material properties for the buoyancy forces (see Section A.9) and radiogenic heating (see Section A.4).

This should normally work without any tweaking. However, if your model has strongly distorted elements, then you may see anomalously high temperature variations. To fix that, modify the prefactor for SUPG (see Section 5.15) by setting the variable `supgFactor` to something smaller. A good first guess is to try 0.5. Note that if you set `supgFactor` too small, then you may see other numerical artifacts.

A.3 Shapes

When setting up the simulation, Gale first creates the computational domain. That domain may be irregular if you are using a `SurfaceAdaptor` (Section A.5.5). Gale then starts putting down materials within that domain. When putting down a material at a particular point, Gale asks all of the materials (Section A.4) whether that point belongs to that material. So it is perfectly fine to have material shapes that cover more than the computational domain. Figure A.1 shows an example with irregular top and bottom materials. Materials for the upper and lower lithosphere are defined in large, regular boxes, but material is only created within the blue region.

As a simple example, you can create a 3D box

```
"box":
{
  "Type": "Box",
  "startX": "0.0",
  "endX": "1.0",
  "startY": "0.0",
  "endY": "1.0",
  "startZ": "0.0",
  "endZ": "1.0"
}
```

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

```
"sphere":
{
  "Type": "EquationShape",
  "equation": "1-(x*x + y*y + z*z)"
}
```

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

```
"nonSphere":
{
  "Type": "Intersection",
  "shapes": [
    "box",
    "!sphere"
  ]
}
```

Note that the exclamation point "!" in front of `simpleSphere` means "not." So this `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

```
"simpleBox": {
  "Type": "Box",
  "CentreX": "1.0",
  "startX": "0.0",
  "endX": "1.0",
  "startY": "0.0",
  "endY": "1.0",
  "startZ": "0.0",
  "endZ": "1.0"
}
```

then the box will actually span from $x = 1$ to $x = 2$.

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. Note that the rotation is around the center of the shape. So, for example, a box will rotate around the center of the box, not one of its corners.

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.3.1 EquationShape

This shape is defined by a user-defined equation. Specifically, a point is inside the shape if

$$equation >= 0.$$

So to define a sphere centered at (1,2,1.5) with radius=5, set

```
"equation": "5^2-((x-1)^2 + (y-2)^2 + (z-1.5)^2)"
```

A.3.2 Box

This is a simple rectangular box. A point is inside the shape if

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

Alternately, you can use widths, in which case

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

You may mix and match these specifications (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.3.3 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

```
''triangleShape'':
{
  ''Type'': ''PolygonShape''
  ''vertices'': [
    ''asciidata'',
    [''x'', ''y''],
    0.0, 0.0,
    1.0, 0.0,
    1.0, 1.0
  ]
}
```

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 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

```
''viscousRheology'':
{
  ''Type'': ''MaterialViscosity'',
  ''eta0'': ''10.0''
}
```

and a Von Mises yielding rheology

```
''strainWeakening'':
{
  ''Type'': ''StrainWeakening'',
  ''TimeIntegrator'': ''timeIntegrator'',
  ''MaterialPointsSwarm'': ''materialSwarm'',
  ''softeningStrain'': ''0.1'',
  ''initialDamageFraction'': ''0.0'',
  ''initialDamageWavenumber'': ''0.5'',
  ''initialDamageFactor'': ''0.5'',
  ''healingRate'': ''0.0''
},

''yieldingRheology'':
{
  ''Type'': ''VonMises'',
```

```
  "cohesion": "10.0",
  "cohesionAfterSoftening": "1.0"
}
```

and combine them together with `materialShape` (see Section A.3 on how to create shapes)

```
"yieldingMaterial":
{
  "Type": "RheologyMaterial",
  "Shape": "materialShape",
  "Rheology": [
    "viscousRheology",
    "yieldingRheology"
  ]
}
```

For each material, you can specify a density, a coefficient of thermal expansivity (α), and a thermal diffusivity. To make a pressure or temperature dependent density, set `densityEquation` instead of `density`. For example, specifying

```
"densityEquation": "p<1 ? 2 : 1"
```

will set the density to 2 when the pressure (`p`) is less than 1, and 1 otherwise. For temperature dependence, use the variable `T`. Similarly, to set a pressure or temperature dependent thermal expansity, specify `alphaEquation` instead of `alpha`.

The density and expansivity are used by the `BuoyancyForceTerm` component (see Section A.9.1) to create buoyancy forces. The diffusivity is used by the temperature solver (see Section A.2).

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 `Q`'s and λ's. For example, to specify two different radioactive species, add something like

```
"heatingElements": [
  {
    "Q": "1.0"
    "lambda": "1.0"
  },
  {
    "Q": "2.0"
    "lambda": "2.0"
  }
]
```

At 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.4.1 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.

A.4.2 Viscous

A.4.2.1 MaterialViscosity

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

Defaults	
eta0	1

A.4.2.2 Frank-Kamenetskii

This is a temperature-dependent viscosity

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

Defaults	
eta0	1
theta	0

A.4.2.3 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.4.2.4 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 `minViscosity`.

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

A.4.3 Yielding

Yielding rheologies are a bit more complicated.

A.4.3.1 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. This will usually be `timeIntegrator`.

MaterialPointsSwarm This is the swarm of particles associated with this rheology. This will usually 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`.

For further reference, we 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. From that we define 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.4.3.2 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.4.3.3 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

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

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

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

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.

Defaults	
PressureField	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.4.3.2.

A.4.3.4 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`. This would usually be

```
"director":
{
  "Type": "Director",
  "TimeIntegrator": "timeIntegrator",
  "VelocityGradientsField": "VelocityGradientsField",
  "MaterialPointsSwarm": "materialSwarm",
  "initialDirectionX": "0.0",
  "initialDirectionY": "1.0",
  "initialDirectionZ": "0.0",
  "dontUpdate": "True"
}
```

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.5 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.5.1 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 `WallVCs`. 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

```
"velocityBCs": {
  "type": "CompositeVC",
  "vcList": [
    {
      "type": "WallVC",
      "wall": "bottom",
      "variables": [
        {
          "name": "vy",
          "value": "0"
        }
```

```
      ]
    }
  ]
}
```

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.5.2).

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

```
"velocityBCs": {
  "type": "CompositeVC",
  "vcList": [
    {
      "type": "WallVC",
      "wall": "bottom",
      "variables": [
        {
          "name": "vy",
          "value": "0"
        },
        {
          "name": "vx",
          "value": "exp(-((x-0.5)/0.1)^2)"
        }
      ]
    }
  ]
}
```

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 `MeshShapeVC` (see Section A.7).

A.5.2 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 5.13, where material flows in from the left and out through the bottom. There are two 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

   ```
   "staticBottom": "True"
   "staticLeft": "True"
   ```

   in EulerDeform (see Section A.1.2).

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.5.1 for details. The other boundaries have no-slip conditions.

A.5.3 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 normal stress on the bottom boundary as a linear function of the height. So if gravity is 9.81, the density of the supporting material is 2.3, and the height of the material is 1.2, then the `StressBC` should be

```
"stressBC":
{
  "Type": "StressBC",
  "ForceVector": "mom_force",
  "wall": "bottom",
  "normal_value": "9.81*2.3*(1.2-y)"
}
```

You can apply a shear stress to the boundary by specifying `x_value`, `y_value`, or `z_value` instead of or in addition to `normal_value`.

A.5.4 Temperature Boundary Conditions

Setting the boundary conditions on the temperature works almost exactly the same as velocity boundary conditions (see Section A.5.1). 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

```
"temperatureBCs":
{
  "type": "CompositeVC",
  "vcList": [
    {
      "type": "WallVC"
      "wall": "bottom",
      "variables": [
        {
          "name": "temperature",
          "value": "1.0"
        }
      ]
    }
  ]
}
```

A.5.5 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

```
''surfaceAdaptor'':
{
  ''Type'': ''SurfaceAdaptor'',
  ''mesh'': ''v-mesh'',
  ''sourceGenerator'': ''v-mesh-generator'',
  ''topEquation'': ''0.1*sin(2*pi*x)''
}
```

This sets the height of the surface to

$$h = h_0 + 0.1 \cdot \sin(2\pi x),$$

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

```
''surfaceAdaptor'': {
  ''Type'': ''SurfaceAdaptor'',
  ''mesh'': ''v-mesh'',
  ''sourceGenerator'': ''v-mesh-generator'',
  ''topEquation'': ''0.1*sin(2*pi*x)'',
  ''bottomEquation'': ''0.1*sin(2*pi*x)''
}
```

then the top and bottom will follow similar curves.

You can also read in topographic data from a file by setting the `SurfaceType` to `topo_data`.

```
"surfaceAdaptor":
{
    "Type": "SurfaceAdaptor",
    "mesh":"v-mesh",
    "sourceGenerator": "v-mesh-generator",
    "topSurfaceType": "topo_data",
    "topSurfaceName": "input/cookbook/test.topo",
    "topNx": "32",
    "topNz": "12",
    "topMinX": "minX",
    "topMaxX": "maxX",
    "topMinZ": "minZ",
    "topMaxZ": "maxZ"
}
```

This will read 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.

A.5.6 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.5.6.1 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`. For example to apply diffusion with a coefficient of 1, add

```
"plugins": [
  {
    "Type": "Underworld_SurfaceProcess"
  }
],
"SurfaceProcess":
  {
    "mesh": "v-mesh",
    "VelocityField": "VelocityField",
    "diffusionCoefficient": "0.1"
  },
```

just before the `EulerDeform` struct.

A.5.6.2 HRS Erosion

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

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

where

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

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`.

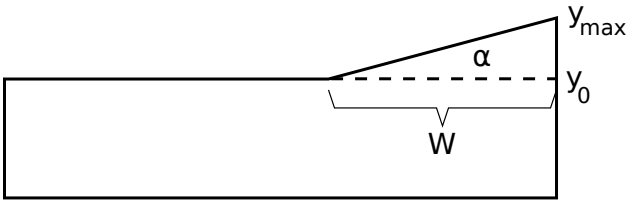


Figure A.2: Geometry for HRS Erosion

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

A.6 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.7 Fixing Internal Degrees of Freedom

While the velocity and temperature boundary conditions (see Sections A.5.1 and A.5.4) 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

```
{
  ”type”: ”MeshShapeVC”
  ”Shape”: ”fixedShape”
  ”variables”: [
    {
      ”name”: ”vy”
      ”value”: ”0”
    }
  ]
}
```

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 `MeshShapeVC`.

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) everywhere.

A.8 Initial Conditions

For temperature dependent problems, you need to set initial conditions for the temperature. Also, for pure thermal problems, the velocity is not solved for, so it must be set at the beginning. Setting initial conditions is similar to setting boundary conditions. In general, the only difference is changing the condition type from `WallVC` to `AllNodesVC`. As an example, to set the initial temperature everywhere to 1, you would add

```
”temperatureICs”: {
  ”type”: ”CompositeVC”
  ”vcList”: [
    {
      ”type”: ”AllNodesVC”,
      ”wall”: ”bottom”,
      ”variables”: [
        {
          ”name”: ”temperature”,
          ”value”: ”1.0”
        }
      ]
    }
  ]
}
```

A.9 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.9.1 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.4). The vector `gravityDirection` determines the direction of the force. In the sample input files, `ForceVector` is always `mom_force`, and `Swarm` is always `picIntegrationPoints`.

`damping` is whether to enable a damping term to fix a sloshing, "drunken seaman" instability often seen in models with a free surface. Adding the damping term with an adaptive step size makes the problem non-linear. If you have problems with convergence, try setting `dtFactor` to something less than 1 (e.g. 0.5), or use a fixed step size by setting `dt` (see Section A.1.4).

Defaults	
gravity	0
gravityDirection	(0,-1,0)
TemperatureField	none
ForceVector	none
Swarm	none
damping	True

A.9.2 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`.

Defaults	
RaC	0
RaT	0
TemperatureField	none
ForceVector	none
Swarm	none

A.10 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 parameter will always be the same between input files.

ForceVector cont_force

The last three parameters specify the divergence.

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

force_type This can be any one of "equation" (the default), "double", or "func" (mostly used for input from a file).

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., `File1`).

A.11 Equation Input

Gale includes the equation parser muParserX (`http://code.google.com/p/muparserx/`). This allows you to enter initial conditions, boundary conditions, and shapes using natural mathematical notation. The syntax is meant to be as close as possible to natural notation as possible. For example,

```
exp(-2*(x^2 + y^2))
```

is equivalent to the formula

$$e^{-2\left(x^2+y^2\right)}.$$

Within each equation, the coordinates are the only predefined variables: `x`, `y`, `z`, `t`. For your convenience, you can also define variables within an equation. The equation

```
r=hypot(x^2+y^2), r*exp(-(r/10)^2)
```

defines a radius and then uses it. Statements are separated by commas ",", and the return value of the equation is the last statement.

The available unary and binary operators are

+	addition
-	subtraction or unary minus
*	multiplication
/	division
^	raise to the power:x^y

In addition, the available functions are

sin(x)	sin
cos(x)	cos
tan(x)	tan
asin(x)	arcsin
acos(x)	arccos
atan(x)	arctan
sinh(x)	hyperbolic sin
cosh(x)	hyperbolic cos
tanh(x)	hyperbolic tan
asinh(x)	hyperbolic arcsin
acosh(x)	hyperbolic arccos
atanh(x)	hyperbolic arctan
sqrt(x)	Square root: $\sqrt{x}$
cbrt(x)	cube root: $\sqrt[3]{x}$
sqrt1pm1(x)	$\sqrt{1+x}-1$, optimised for when x is small
hypot(x,y)	$\sqrt{x^2+y^2}$
erf(x)	error function
erfc(x)	complementary error function
log(x)	natural logarithm:$\log(x)$
log1p(x)	$\log(1+x)$, optimised for when x is small
log10(x)	$\log_{10}(x)$
log2(x)	$\log_2(x)$
exp(x)	e^x
expm1(x)	e^x-1, optimised for when x is small
abs(x)	Absolute value: $\lvert x\rvert$
step(x)	0 if x<0, 1 otherwise
floor(x)	largest integer not greater than x
ceil(x)	smallest integer not less than x
sum(x1,x2,x3,...)	Sum of individual values: x1+x2+x3+...
min(x1,x2,x3,...)	Minimum of all values
max(x1,x2,x3,...)	Maximum of all values

In addition, you can use the syntax `a ? b : c` as a conditonal. If the first element `a` is true, then return `b`. Otherwise return `c`. For the condition `a`, you can use all of the normal relational operators. To be specific, the allowed relational operators are

<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	is equal
!=	is not equal
or	logical or
and	logical and
xor	exclusive or

Note that this will only evaluate the element as needed. So if x<1, `a` will be evaluated, but not `b`. This can come in handy if `a` or `b` are difficult to compute or not valid for certain ranges.

If you are unsure whether the equations you entered are producing the desired numbers, you can turn on the verbose option (Section A.14). This will output the equation that is being evaluated, the coordinates, and the result.

A.12 File Input Data

The other way to set initial and boundary conditions is by reading it in from a file. You can use up to 10 different files as input data (`File1`, `File2`, ... `File10`). For each `File`, there are a number of associated parameters. As a concrete example, `File1` will have data along the axis `File1_dim` and, if defined, the axes `File1_dim2` and `File1_dim3`. Depending on how many dimensions are defined, `File1` will be an array of `File1_N`, `File1_N*File1_N2`, or `File1_N*File1_N2*File1_N3` elements. Gale reads these elements from a file. The format of the file is one column for each of the coordinates (1, 2, or 3), and one column for 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, starting with 10 at 0 and ending with 20 at 100. For points less than 0, Gale uses the value of the lowest point (10). For points greater than 100, Gale uses the value of the highest point (20).

For 2D and 3D input files, the coordinate mesh defined by the input file must be a rectangular grid.

A.13 Tracers

You can add tracer particles to the simulation to help you track where material is flowing. The existing material particles are not suitable for that because they may get duplicated or removed as the simulation proceeds. This is necessary to keep the number of particles down to a reasonable level. Tracers, on the other hand, are merely silent observers, playing no role in the evolution of the system.

To add tracers to your simulation, first enable tracers by adding the variable

```
"enable-tracers": true,
```

Note that there are no quotes around `true`. Next set up the initial position of the tracers. To put the tracers exactly where you want them, use a `ManualParticleLayout` component. An example of one that puts down 8 tracers is

```
"pLayout": {
  "Type": "ManualParticleLayout",
  "totalInitialParticles": "1",
  "manualParticlePositions": [
    "asciidata",
    ["x", "y", "z"],
    1.0, .1, .1,
    1.3, .1, .1,
    1.6, .1, .1,
    1.9, .1, .1,
    1.0, .2, .1,
    1.3, .2, .1,
    1.6, .2, .1,
    1.9, .2, .1
  ]
}
```

Finally, add a `TracerOutput` component to output values of various fields (e.g. pressure, temperature) as the simulation progresses.

```
"swarmOutput":
{
```

```
    "Type": "TracerOutput",
    "Swarm": "passiveTracerSwarm",
    "Fields": [
      "PressureField",
      "StrainRateInvariantField"
    ]
  }
```

This component will create eight plain text files in the output directory, `swarmOutput.00000.dat`, `swarmOutput.00001.dat`, ... `swarmOutput.00007.dat`. Each file will contain the positions of the particle through time and the values of the pressure and strain rate invariant at those positions.

A.14 Verbosity Options

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

```
"journal.info": "True",
"journal.debug": "True",
"journal-level.info": "2",
"journal-level.debug": "2"
```

into the variables section (see Section A.1.4). This will print out more information than you need about the equations, components, solvers, and 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

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 B.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	B.2, B.3
Stokes solver and interpolate between particles and mesh in 3D	B.1, B.3
Time stepping	B.2
Gravity	B.1, B.2
Free surface	B.2
Thermal Advection & Diffusion	B.4, B.5, B.6

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

Many of these 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.

B.1 Falling Sphere

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

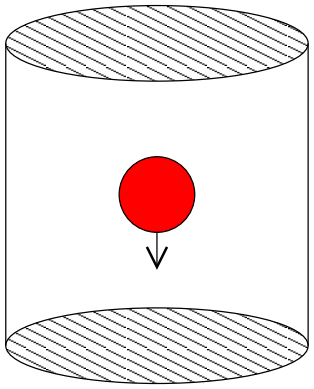


Figure B.1: Schematic of a Sphere falling through a Cylinder

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

$$u = 0.222.$$

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

$$u = 0.1122747319.$$

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

$$u = 0.112293603939,$$

which agrees closely with the result from Habermann.

Figure B.2 shows the velocity solution for the resolution 32×64×32. Because of the symmetries of the problem we only have to simulate a quarter of the domain. Since the sphere is not completely rigid, the velocity inside the sphere is not uniform. In particular, the velocity is largest in the center of the sphere and decreases outward.

We plot the error in the computed velocity compared to the Faxen solution in Figure B.3. The error bars correspond to the range of velocities for $r < 0.7$. As the element size h decreases, the error decreases.

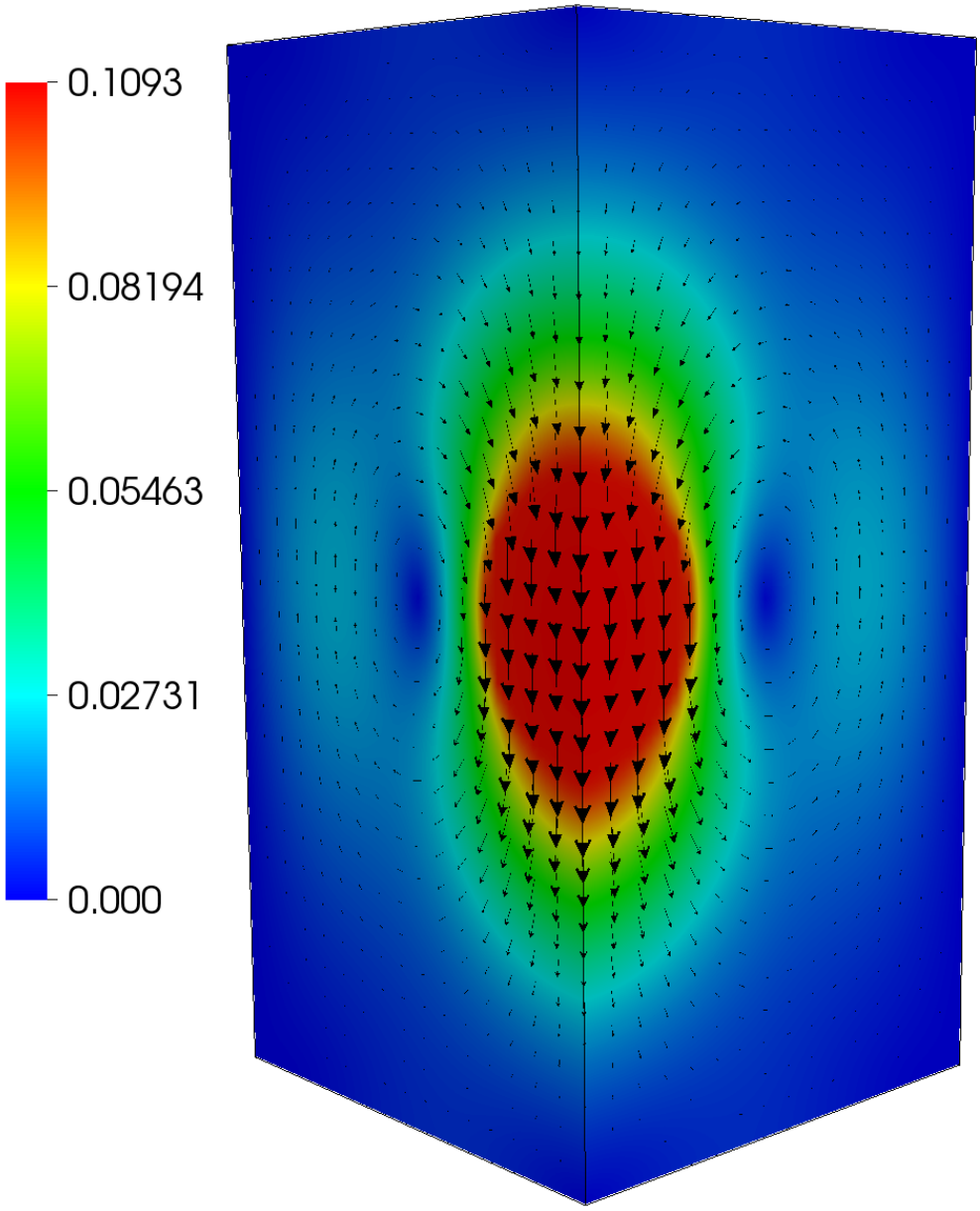


Figure B.2: Velocity in the sphere and surrounding medium

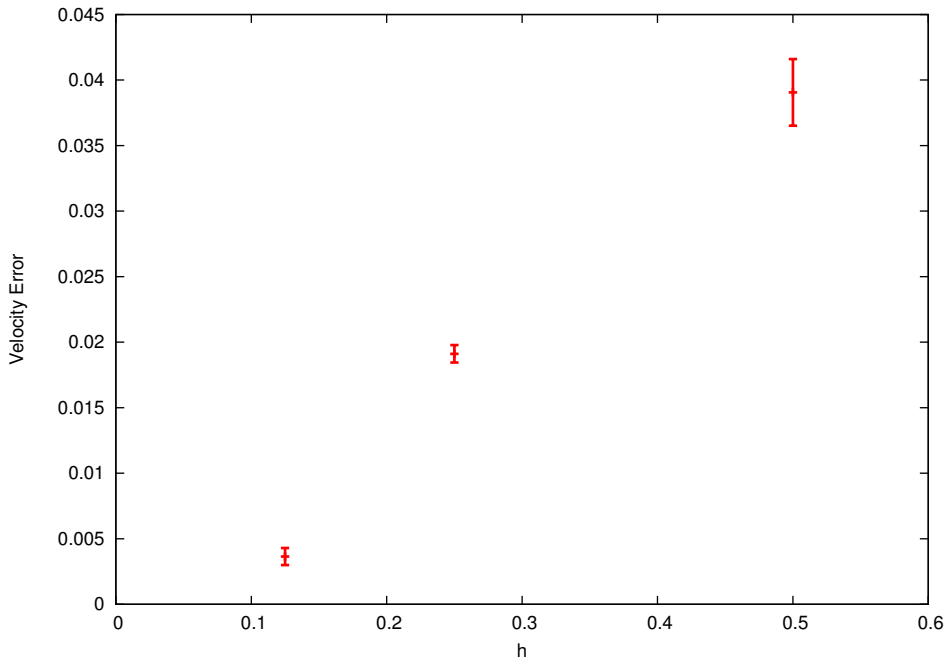


Figure B.3: Error in computed velocity vs. resolution

B.2 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]

$$t_r = \frac{4\pi\eta}{\rho g L},$$

where η is the viscosity, ρ is the density, 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%.

Figure B.4 shows the results of a high-resolution (256×512) run. Note that we use symmetry to only simulate half of the wavelength.

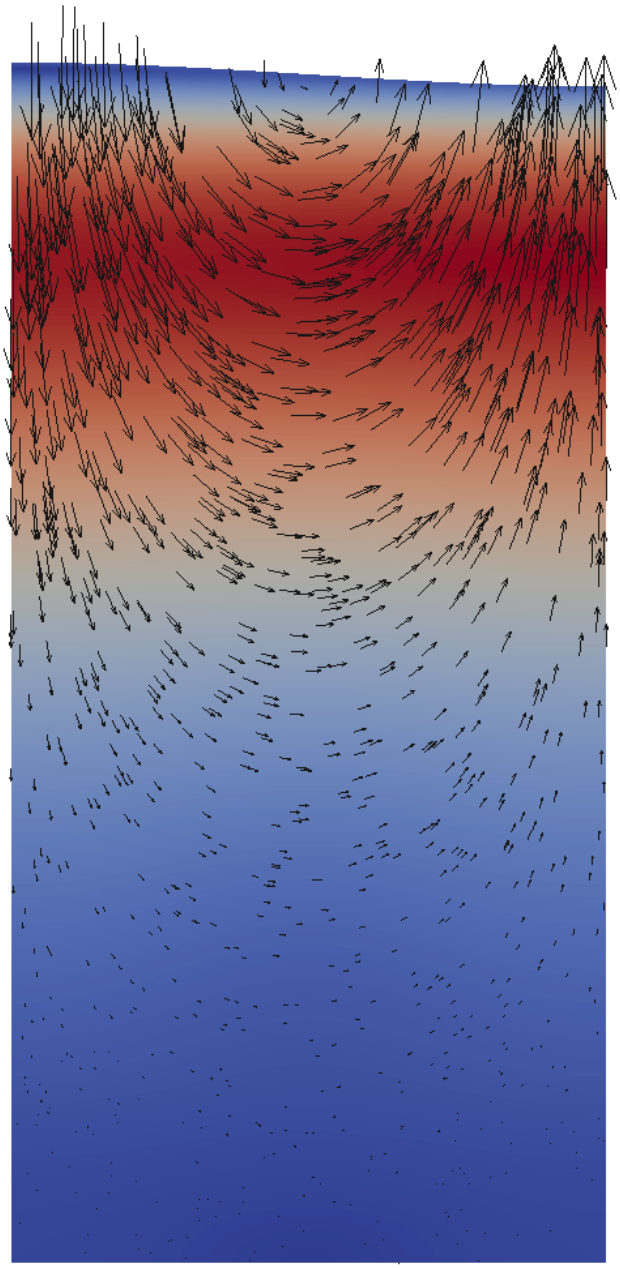


Figure B.4: Strain rate and velocities for a sinusoidal topography relaxing under gravity

Running the code with multiple resolutions and measuring the error in the height in the peak gives Figure B.5. The error behaves a bit erratically because of the damping term applied to the free surface (Section A.9.1). Even so, the error decreases linearly with increasing resolution, giving us confidence in our ability to accurately track topography.

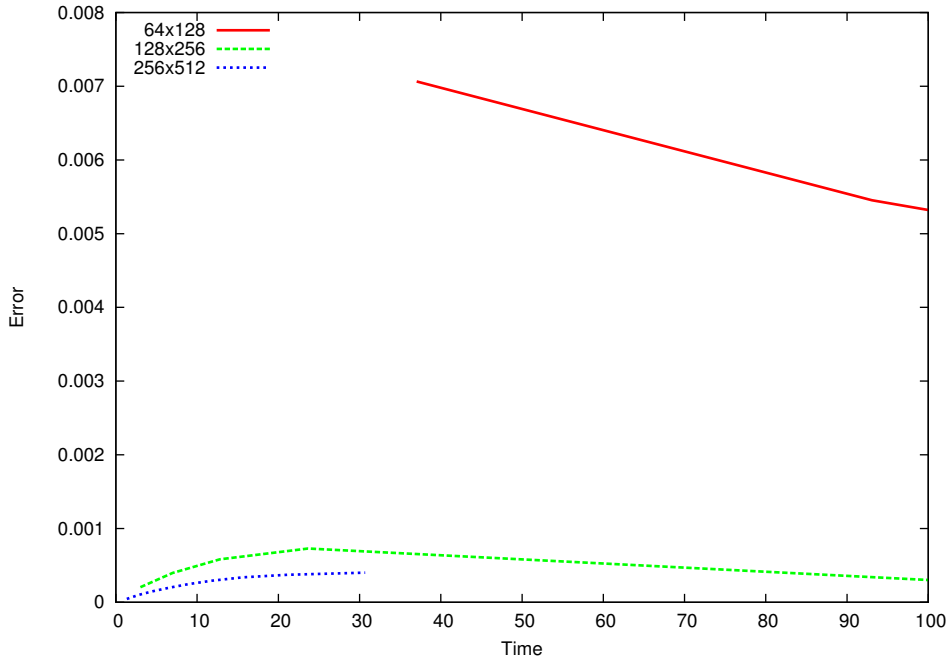


Figure B.5: Error in the height at the peak

B.3 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. 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 $\sqrt{d/2}$. The error is completely determined by the solver. In both 2D and 3D, decreasing `linearTolerance` to 10^{-9} results in a solution with zero error.

B.4 Thermal Diffusion

This is a pure thermal benchmark. The Stokes equations are not solved. Rather, the benchmark simulates a box relaxing from an initial sinusoidal temperature distribution. We set the velocity to zero and the initial temperature to

$$T_{t=0} = \cos(\pi x)\sin(2\pi y).$$

The temperature on the bottom and top are fixed to zero. The temperature on the left and right side are left free, implying the boundary conditions

$$\left.\frac{\partial T}{\partial x}\right|_{x=0,1} = 0.$$

The complete solution decays with time

$$T = \exp\left(-\kappa\left(5\pi^2\right)t\right)\cos(\pi x)\sin(2\pi y),$$

where $\kappa = 1.7$ is the diffusion coefficient. Figures B.6 and B.7 show the results at $t = 0$ and $t = 0.0011489$ for a run with 16×16 elements. Figure B.8 plots the error in the maximum temperature at $t = 0.0011489$ as a function of the grid spacing h. The error decreases linearly as the spacing decreases.

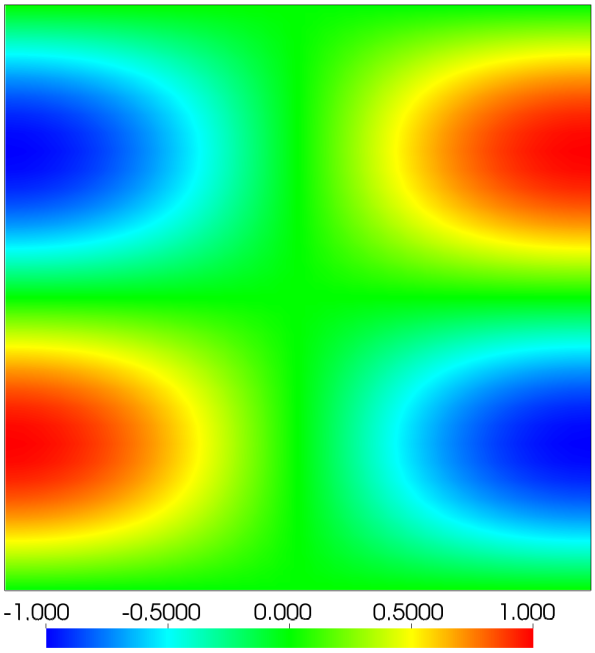


Figure B.6: Temperature at the beginning of the thermal diffusion benchmark. The mesh is 16×16 elements.

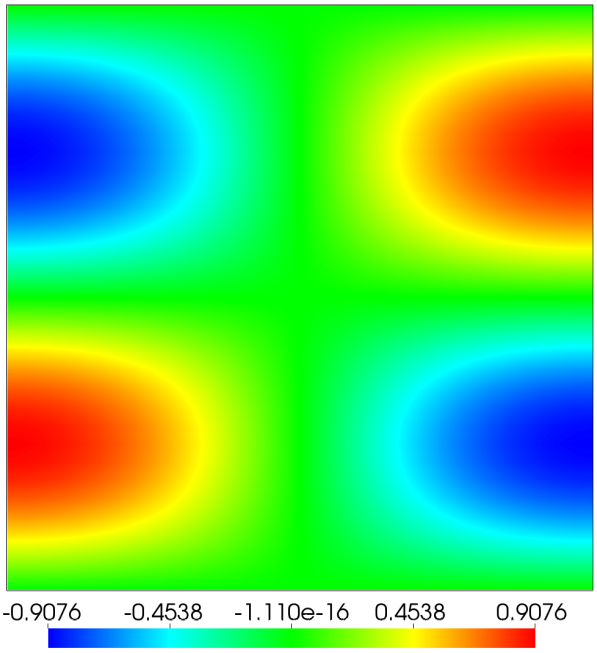


Figure B.7: Temperature at the end of the thermal diffusion benchmark. The mesh is 16×16 elements.

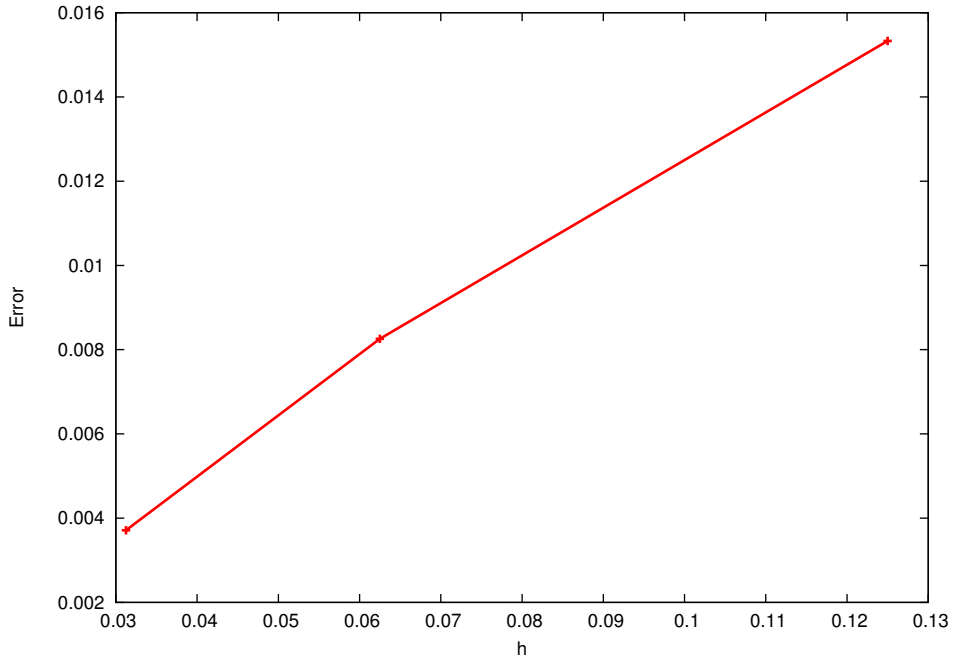


Figure B.8: Error in the maximum temperature at $t = 0.0011489$ as a function of resolution.

B.5 Lagrangian Thermal Advection

This is another pure thermal benchmark. In this case, the velocity is specified and the diffusivity is set to zero. The temperature is initially set to be 2 inside a box and 1 outside. We set the velocity to

$$\begin{aligned} v_x &= 0.3x - 0.2 \\ v_y &= (x + 0.3)(1.5 - x)y + (x - 0.15)(0.7 - x) \end{aligned} .$$

This velocity has been constructed such that the natural advection of the mesh will not be disturbed by the remesher. This means that the temperature field should not advect relative to the mesh. Figures B.9 and B.10 show the initial and final temperatures. The initial temperature distribution is kept sharp and intact.

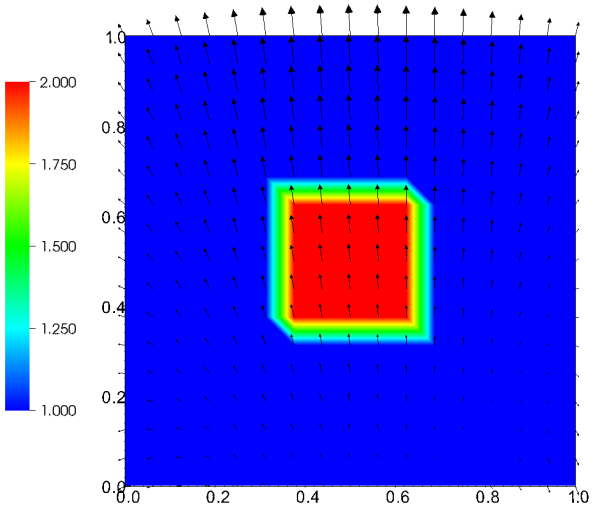


Figure B.9: Initial temperature and velocity of the lagrangian thermal advection benchmark.

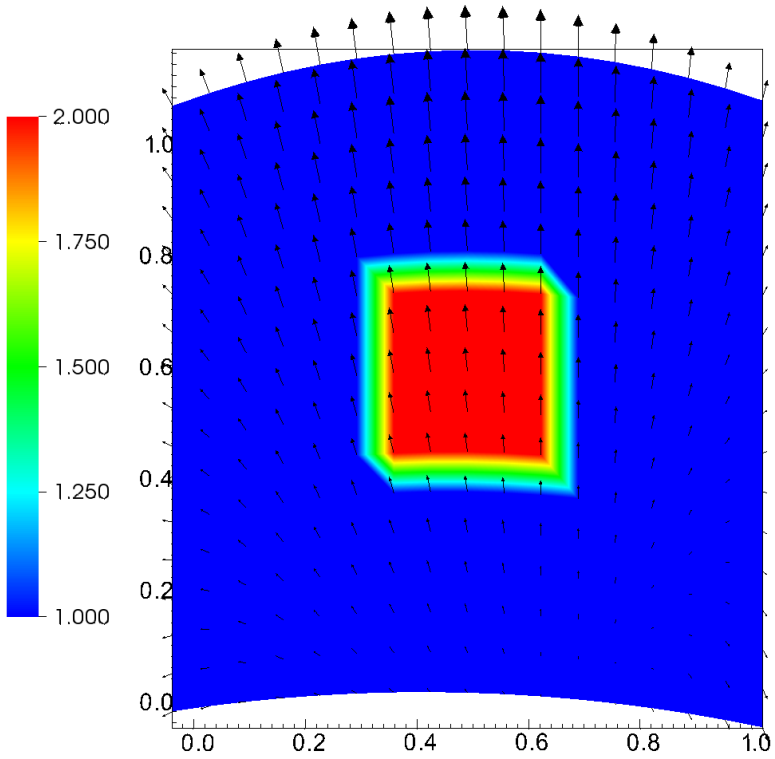


Figure B.10: Final temperature and velocity of the lagrangian thermal advection benchmark.

B.6 Eulerian Thermal Advection

This is another pure thermal benchmark. In contrast to the previous benchmark, the mesh is fixed and the temperature is advected across the grid. The velocity is set to $v_x = 1$, $v_y = 1$. Figure B.11 shows the initially sharp temperature distribution. Figures B.12, B.13, and B.14 show the result at $t = 0.25$ for runs with 16×16, 32×32, and 64×64 elements. While there is significant diffusion, it does improve with resolution.

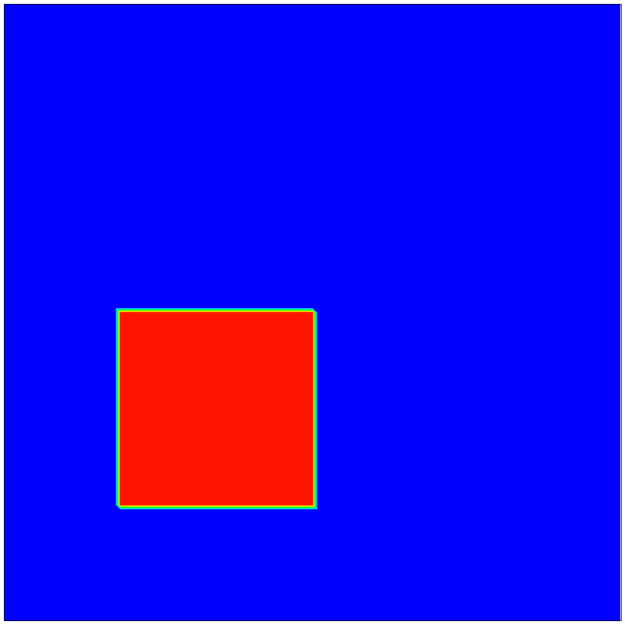


Figure B.11: Initial temperature of the eulerian thermal advection benchmark.

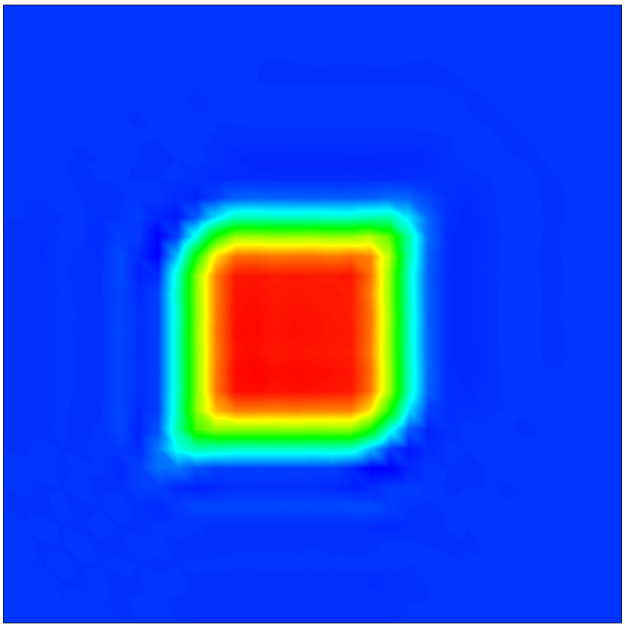


Figure B.12: Temperature at $t = 0.25$ for a run with 16×16 elements.

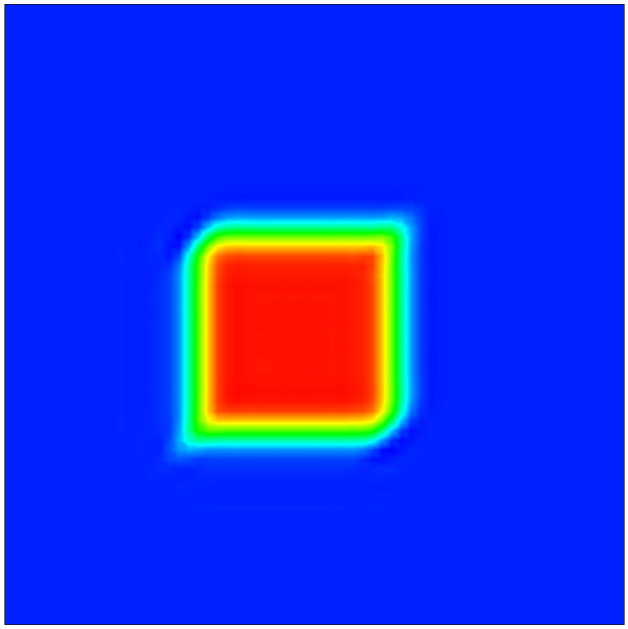


Figure B.13: Temperature at $t = 0.25$ for a run with 32×32 elements.

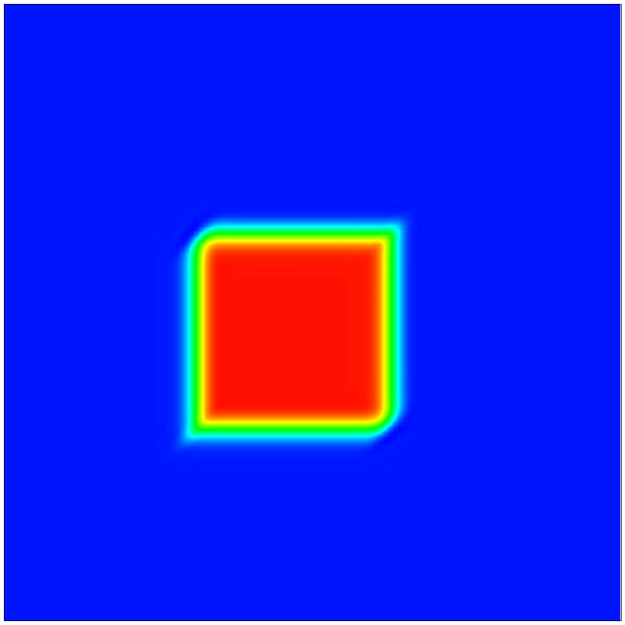


Figure B.14: Temperature at $t = 0.25$ for a run with 64×64 elements.

Appendix C

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