User Manual

Release 1.7

Monash University, VPAC

September 19, 2012

CONTENTS

CHAPTER

ONE

ACKNOWLEDGEMENTS

The following people have contributed content to the Underworld manual (in alphabetical order):

Cecile Duboz, Rebecca Farrington, Justin Freeman, Julian Giordani, Luke Hodkinson, Megan Hough, Kathleen Humble, Owen Kaluza, Walter Landry, David Lee, Vincent Lemiale, Alan Lo, John Mansour, Wendy Mason, Belinda May, David May, Catherine Meriaux, Louis Moresi, Steve Quenette, Jerico Revote, Wendy Sharples, John Spencer, Jay Stafford, Dave Stegman, Patrick Sunter, Robert Turnbull, Mirko Velic, Sergio Zlotnik.

1.1 Special Thanks

Timothy Barry, Julian Giordani, Wendy Mason, and *Patrick Sunter* for their advice on structure, content and LaTEX. Also thanks to *Owen Kaluza* for his work on *Visualisation* (page 63).

CHAPTER

TWO

PREFACE

2.1 Introduction

Underworld is a 3D-parallel geodynamic modelling framework capable of deriving viscous/visco-plastic thermal, chemical and thermochemical models consistent with tectonic processes, such as mantle convection and lithospheric deformation over long time scales.

Underworld utilises a Lagrangian particle-in-cell finite element scheme (the prototype of which is the Ellipsis code), and is visualised using gLucifer. The Underworld source code is written in C in an Object Oriented style, following the methodology of design for change applied to computational codes implemented in StGermain.

The Underworld to StGermain software stack is released under a mixture of BSD and LGPL licences. It uses PETSc (optimised numerical solvers) and MPI (parallelism) libraries.

All the projects that make up Underworld now use the CREDO toolkit for system-level testing. CREDO is an open-source toolkit for benchmarking and analysis of scientific modelling software written in Python, and is developed and maintained along with Underworld. CREDO uses the LGPLv2.1 licence.

2.2 Underworld Users

The main audience for Underworld, the 3D parallel geodynamic modelling software, are research scientists interested in modelling geodynamic processes on long time scales. Computer programmers and mathematical modellers would also find the underlying modelling framework of interest.

2.3 Citation

Underworld is made available under the GNU General Public Licence,. A number of individuals have contributed a significant proportion of their careers towards the development of Underworld. It is essential that you recognise these individuals in the normal scientific practice by making appropriate acknowledgements.

The code is based on the method described in [MoresiQuenetteLemiale2007] (page 95). And the core algorithms and structure are based on the method described in [MoresiDufourMuhlhaus2003] (page 95).

This code is developed by the Victorian Partnership of Advanced Computing (VPAC), computational software developers in collaboration with Monash University's Monash Geodynamics group, led by Louis Moresi, as part of the AuScope nation-wide Geosciences infrastructure program. The Underworld team requests that in your oral presentations and in your publications that you indicate your use of this code and acknowledge the authors of the code, Victorian Partnership for Advanced Computing and Monash University.

2.4 Support

Underworld is under development as part of AuScope Ltd, which is funded under the National Collaborative

Research Infrastructure Strategy (NCRIS), an Australian Commonwealth Government Programme. (Formerly Underworld was a project as part of the Australian Computational Earth Systems Simulator (ACcESS MNRF).

Underworld is jointly maintained by the Victorian Partnership for Advanced Computing (VPAC) and Monash University.

CHAPTER

THREE

INSTALLATION

3.1 Introduction

This section provides steps and links on obtaining and setting up Underworld. For running Underworld on a cluster, see the Underworld Cluster page. For running Underworld on the ARCS Grid, see the Underworld Grid page.

For installation on local machines, Underworld is available as:

- Binary packages, under Ubuntu and Mac OS X platfroms, no dependencies required.
- As a stable release source code tarball, dependencies required.
- From the source code repository, dependencies required.

See the Underworld Downloads page for more details.

The following instructions assume a beginner's level of familiarity with the Unix/Linux operating environment, including:

- How to open a terminal window
- How to navigate between directories using a terminal application
- How to open, edit, save and close a file using an editor through a terminal application

There are many beginners Unix / Linux guides available as textbooks and on the Internet, including the Unix Tutorial for Beginners.

3.2 Getting Help

For help, advice or the latest information on Underworld:

- Have a look through the Underworld homepage.
- Read the Underworld FAQ page.

3.3 System Requirements

3.3.1 Operating System

Underworld will work and is supported on:

- Linux-based OS (desktop and supercomputers)
- Mac OS X (Power PC and Intel)

Note: Underworld is currently not available on Windows.

Underworld requires:

- 300-500 MB for a compiled copy of Underworld. (x3 for source code component if using Mercurial)
- 340 MB approx. for software dependencies.
- Enough space for your output data (e.g. 500+ MB recommended for medium to large-scale problems).

3.3.2 Software Dependencies

Underworld relies on a suite of software applications, which need to be installed prior to installing Underworld. It is recommended that you install each program in the order listed. However, if you are going to install any of the Visualisation Software or Interactive Visualisation Software listed further below, install them before configuring PETSc.

See the Underworld Dependencies page for instructions on how to download and install each program.

When installing Underworld from one of the available Ubuntu binary packages, all dependencies are automatically downloaded and installed on the system. When using the Mac OS X binary packages, the dependencies are bundled together with Underworld, so no manual installation of the dependencies is required.

Required Software

If building Underworld from source either from the tarball or repository checkout, a number of software dependencies needs to be installed first.

The following list outlines the required software that needs to be installed prior to configuring and installing Underworld:

- **C Compiler** - Needed to compile several dependencies and Underworld; usually installed by default on Linux systems. On Macs this can be installed from the Developer Tools package that came with your Mac OS. C compilers are *gcc* and *icc*.
- **MPI Implementation** - Parallel MPI Message-passing library. Underworld supports MPICH and OpenMPI.
  - *MPICH* - See the Underworld Dependencies page and MPICH homepage for installation and version details.
  - *OpenMPI* - See the Underworld Dependencies page and OpenMPI homepage for installation and version details.
- **LAPACK and BLAS** - Used to provide basic *numerical kernels* for solving matrices etc. See the Underworld Dependencies page for details on how to install these libraries for various platforms.
- **Python** - Required to compile PETSc and Underworld components. To check if you need to install Python, see the Underworld Dependencies page or the Python homepage.
- **PETSc** - A set of libraries including equation solvers and assembly routines. To Install PETSc see the Underworld Dependencies page and the PETSc homepage.
- **HDF5** - The default application used for checkpointing meshes and particles. See the Underworld Dependencies page for further details.
- **SCons** - Package for building and compiling Underworld on your system. For versions of Underworld greater than 1.2.0, scons is packaged with the Underworld tarball or checkout.

Additional Software

To checkout the Underworld code from the repository, the following software is required:

- **Mercurial** - The version control system used by Underworld. See the Underworld Dependencies page for further details.

For visualisation output, these additional software are required:

- **libppng** and/or **libjpeg** and/or **libtiff** - For image output in png, jpeg and tiff formats.
- **libavcodec** or **libfame** - For producing animations.

For interactive visualisation, these additional windowing software are required:

- **X11** - Standard on Linux. Comes as an option with *Developer Tools* or separate package on MACs. See the Underworld Dependencies page for further details.
- **Carbon** - MACs only, comes already installed.
- **OSMesa** - Can be installed on most systems. Recommended only for *off-screen* rendering as this doesn't work with interactive visualisations. See the Underworld Dependencies page for further details.
- **SDL** - Can also be used.

To draw the output image inside the created window/s, one of the following additional software are required:

- **OpenGL** - Comes standard with Linux and MACs platforms.
- **Mesa** - Comes with OSMesa. See the Underworld Dependencies page for further details.

To output visualisation data into a database, the following software is required:

- **SQLite** - Comes standard with MACs. Available as a package from most Linux distros.

Note: For interactive visualisation, don't use OSMesa.

Note: For cluster installations, only use OSMesa.

When installing Underworld on a cluster environment, it is recommended to use OSMesa only, since most clusters will not be running the X11 daemon on compute nodes. All *input files* can be ran without interactive visualisation, in which case images are still created and can be opened manually. For further information on *input files*, see *Modelling* (page 20); gLucifer output, see *Images and Movies* (page 67).

3.4 Getting Underworld

See the Underworld Releases page for instructions on how to download different versions of Underworld, or go straight to Underworld Downloads page to download a tarball of the code straight away.

3.4.1 As a Binary Package (Ubuntu and MACs)

Binaries for Underworld are available at the Underworld Downloads page. Supported binaries are Ubuntu (.deb) and MAC (.dmg). The Ubuntu binary packages are available in i686 (32-bit) and amd64 (64-bit) architectures. Make sure that you're downloading the right package for your computer's architecture. The *input files*, available on the same page, need to be downloaded separately.

3.4.2 As a Release Tarball

Release tarballs are available at the Underworld Downloads page.

Note: Tarball copy of the code cannot be updated to a later version. There will be a separate tarball for each *stable release* of Underworld.

3.4.3 From the Mercurial Repository

Information on how to check-out Underworld is available at the Underworld Releases page. There are two options available: *stable release* and *bleeding-edge*. Unless you are interested in writing or contributing code, the *stable release* checkout is the recommended version to use.

3.4.4 Getting Underworld Input Files

If you are using the latest *stable release* version of Underworld, the *input files* can be downloaded from the Underworld Downloads page. The *input files* are the standard XML files that form the models for Underworld.

Underworld is designed to operate in a multi-user environment, in which users can have their own files located in their home directory which link to the files downloaded from the repository. This ensures that conflicts do not arise when running updates to Underworld and reduces the amount of work required for each user to produce their own working model.

3.5 Installing Underworld

The Underworld Releases page, updated on a regular-basis contains references for installing Underworld.

3.5.1 From a Binary Package (Ubuntu and MACs)

Once the binary package has been downloaded (see *Getting Underworld* (page 7)),

On Ubuntu:

1. Double-click on the file to launch the installation manager.
2. Click on the *Install Package* option to start the installation. This process might take a moment as this step also downloads and installs the required dependencies.
3. Once the installation is finished you can now close the manager.
4. The *Underworld* executable will be part of the environment so you can now issue *Underworld* on the command line and start running your models.

On MAC:

1. Double-click on the file to mount the package.
2. Once mounted, Double-click on the package to start the installation process.
3. Follow the standard prompt for installing the package.
4. Once the installation is finished you can now close the manager.
5. The *Underworld* executable will be part of the environment so you can now issue *Underworld* on the command line and start running your models.

3.5.2 From Source (Tarball, Repository)

Once all required and additional dependencies are installed (see *Software Dependencies* (page 6)), and the Underworld source code has been acquired either from the tarball or repository (see *Getting Underworld* (page 7)), you can now *compile* the code.

1. First navigate to the source code directory, for example of the directory name is Underworld-1.5:

   cd Underworld-1.5

2. Configure Underworld, here we're configuring in optimised mode:

```
./configure.py --with-debugging=0
```

   Typing `./configure.py --help` will list all avaiable configuration options.

3. Compile Underworld:

```
./scons.py
```

See the Underworld Documentation on SCons for a complete reference on installing Underworld from source using SCons.

3.5.3 Running & Testing Underworld

After installing Underworld, you can check the build directory for the created executables and libraries. The build directory will contain the imporant executables, *Underworld* and *FlattenXML*. To verify your installation:

- You can check that the executables are present:

```
ls Underowrld-1.5/build/bin
```

- You can also run the standard tests:

```
./scons.py check > results
```

  This will run the standard unit and system tests for Underworld.

See *Tests* (page 30) for more information on Underworld tests. If you encounter any issues running Underworld or if you get any errors in running the tests and you're using the *stable release* version of the code, contact users@underworldproject.org, otherwise if you're using the *bleeding edge* version of the code, submit your enquiry to development@underworldproject.org.

CHAPTER

FOUR

BASICS

4.1 Introduction

- Underworld runs on Unix-style systems, e.g. Linux, Mac OS X, etc., (see *System Requirements* (page 5)).
- There are a number of options available to install Underworld, and a number of package dependencies, (see *Installing Underworld* (page 8) and *System Requirements* (page 5)).
- It uses XML code contained in input files to define models.
- It is extensible, and is capable of dynamically loading objects declared in the XML file.
- It is generally ran from a terminal:

```
./Underworld ~/UnderworldFiles/MyInputFiles/Sample.xml
```

  Although other options exist for running jobs including using Python scripts with the CREDO system, submitting jobs over the Grid using the Grisu tool, and a work-in-progress web portal.

- It outputs results in three ways:
  - Outputs to screen
  - Outputs to user-defined files
  - Outputs images via gLucifer (visualisation package included with Underworld)
- It can run a large number of geophysical problems.
- It is written in a hierarchical style with higher-level programs built on the structure of lower-level ones. Underworld is the top-level program.

4.2 Scope

Underworld solves the Stokes flow problem and the energy (advection/diffusion) equation. Underworld allows for a modeller to customise these equations by modifying the assembly methods for these general equation i.e, add force terms, add constitutive behaviours, thus modelling complex geophysical behaviours.

4.2.1 General Equations

Incompressible Stokes Flow:

$$\frac{\partial \tau_{ij}}{\partial x_j} - \frac{\partial p}{\partial x_i} = -\alpha \rho g T \lambda_i \tag{4.1}$$

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{4.2}$$

Advection Diffusion:

$$\frac{\partial T}{\partial t} + u_i \frac{\partial T}{\partial x_i} = \kappa \frac{\partial^2 T}{\partial x_j^2} \tag{4.3}$$

Where x_i are the spatial coordinates, u_i is the velocity, T is the temperature, α is the thermal expansivity, ρ is the fluid density, g is the gravitational acceleration, λ_i is the unit vector in the direction of gravity and κ is the thermal diffusion.

4.3 Hierarchical Framework

Underworld is a program that sits at the top of a hierarchy of other programs. Each of these programs builds on the previous programs expanding the scientific or computational capacity of the underlying programs. The goal of this approach is to modularise the code at a design level, which enables users to work at a particular level of the code without having to modify higher or lower levels significantly.

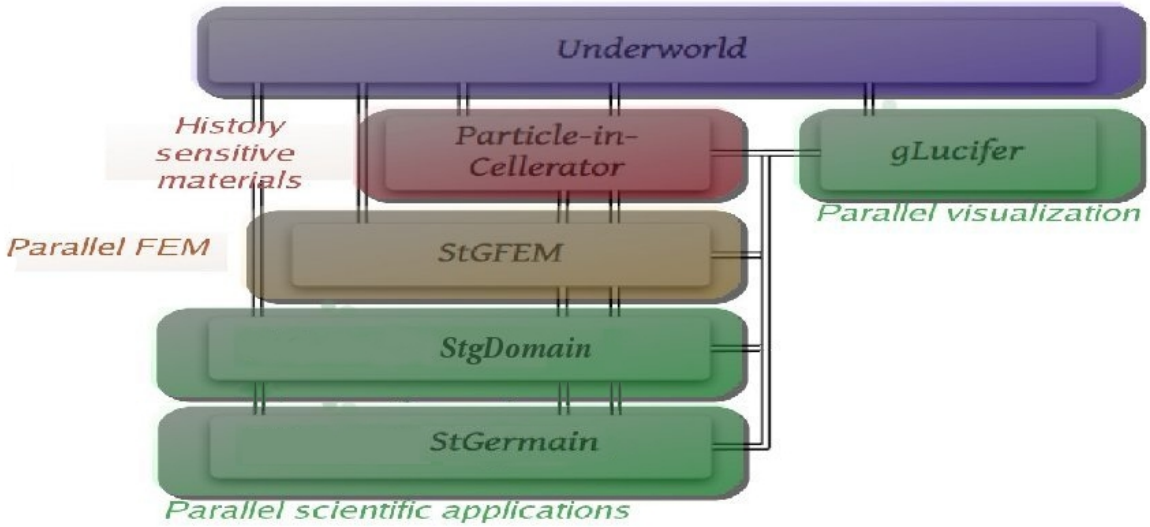


Figure 4.1: Underworld Hierarchical Structure

The following describes the components in each layer.

- **StGermain** - Core parallel programming architecture, memory management and OO frameworks.
- **StgDomain** - Basic mathematical formulations, geometry, mesh, shape and basic swarm framework.
- **StgFEM** - Finite Element Method (FEM) framework (Eulerian).
- **PICellerator** - Particle in Cell methods (Lagrangian / Eulerian framework).
- **gLucifer** - Visualisation package.
- **Underworld** - Geophysical formulations, e.g. rheologies.

4.3.1 StGermain

StGermain is designed to handle the core parallel programming architecture, memory management and Object Oriented (OO) frameworks. This is where the code that runs journalling, memory management, and the various definitions of *containers* such as variables, resides. This is the layer that allows higher end users to use the same model files across one or multiple computers. StGermain forms part of the base level of the hierarchy. Most users will never need to look into this code base.

4.3.2 StgDomain

StgDomain handles the geometry and meshing definitions, (located in the *Geometry* and *Mesh* sub-directories respectively) as well as basic mathematical formulas (located in *Geometry* sub-directory). StgDomain also contains all the code on how to move particle swarms at the most basic level. Users writing their own plugins or components may need to use this layer for it's available pre-written mathematical functions.

4.3.3 StgFEM

StgFEM follows StGermain's approach of breaking the Finite Element Method (FEM) into a component-based, declarative approach, making considerable use of the *Strategy Design Pattern*.

The motivation behind this approach are:

- Setting it up as a framework that can solve different sets of linear equations.
- In particular, allowing declarative specification of commonly used elements in a FEM system-important for complex systems such as Stokes, and coupled systems.
- High degree of modularity for each phase of the FEM process - with the view of other codes being able to modify all of them.

It is organised around the concept of *System of Linear Equations* and:

- Assumes the numerical modeller has already translated from weak form to a matrix, vector system.
- It also puts responsibility on the numerical modeller to be able to translate algebraic terms as they appear in the weak form, to assembly term components.

StgFEM has a number of input files/basic models in the *Apps* directory that are well worth looking at and running. These include the base models for many more complicated XML models.

Some of the XMLs include:

- The Energy Solver
- Stokes Momentum Uzawa
- Thermal Convection
- Temperature Diffusion
- and a number of smaller XML files in *StgFEM_Components* directory.

StgFEM has a number of available *plugins* in the plugins sub-directory that can be of use to the user.

4.3.4 PICellerator

PICellerator is a fairly *lightweight* layer in the framework, in terms of number of new components provided. It is closely related to the StgFEM layer immediately below it. PICellerator uses a Particle-In-Cell approach, with the collection of particles and associated properties known as a *swarm*.

PICellerator's approach has the following key concepts:

- Particle-In-Cell material points
- Integration of FEM at Gauss points is replaced by scheme related to material points (by default, integration points and material points are co-located, but this can be changed).
- Material points are then advected based on the solution of the FEM.

Note: A material point does not map to physical particles, but is designed to show the material properties in a local region of the model.

Material and Integration Swarms

There are two types of swarms in PICellerator:

- **Material swarms** - for physical material properties that move.
- **Integration swarms** - defines discrete integration points.

These two swarms are related, since material properties are often required when integrating, but the code does not necessarily require that the two be co-located. PICellerator deals with the difficult problem of locating to which cell a given particle belongs, particularly across multiple processors. The cells correspond to an *Eulerian Mesh*, with the *Material* and *Integration swarms* handling the *Lagrangian particle advection*. PICellerator also deals with population control as particles are advected in and out of the modelling space; and weighting schemes for integration using both simple and more complicated methods (i.e. *Voronoi schemes*).

There are a number of XML examples available in the *Apps* sub-directory, including:

- The Rayleigh Taylor Benchmark
- Lid Driven PIC convection
- Thermochemical convection
- and the Buoyancy Benchmark

PICellerator has a number of available plugins in the *plugins* sub-directory that can be of use to the user.

4.3.5 Underworld

Underworld forms the top layer in the hierarchy of programs. Underworld at this level, is designed to deal with the models governing various geophysical rheologies. Most of the component models in Underworld are located in the *Rheology* sub-directory.

Examples of problems that can be solved, for which a number of *template* XML models are provided are found in the *InputFiles* sub-directory:

- **Thermal convection**
  - Temperature and stress-dependent convection; (e.g. *NonNewtonian.xml*, *MultiComponent.xml*)
  - Temperature-dependent convection; (e.g. *Arrhenius.xml*, *FrankKanenetskii.xml*)
  - Anisotropic rheology; (e.g. *Anisotropic.xml*)
  - Plastic rheology; (e.g. *MobileLid.xml*)
  - Radiogenic heating; (e.g. *InternalHeating.xml*)
  - Depth-dependent viscosity; (e.g. *DepthDependentViscosity.xml*)
  - and Multi-thermal diffusivity. (e.g. *MultiThermalDiffusivity.xml*)
- **Chemical convection**
  - The Rayleigh-Taylor benchmark; (e.g. *RayleighTaylorBenchmark.xml*)
  - and subducting / falling viscoplastic slab. (e.g. *SlabSubduction.xml*)
- **Thermochemical convection**
  - D” (core-mantle boundary) of the Earth. (e.g. *ThermoChemBenchmark.xml*)
- **Viscoplastic extension**
  - Von-Mises Extension; (e.g. *Extension.xml*)
  - and Faulting Moresi-Muhlhaus, 2006 Extension. (e.g. *ExtensionFMM.xml*, *ExtensionFMM3D.xml*)

CHAPTER

FIVE

UNDERWORLD

5.1 Introduction

This chapter details the essentials of Underworld modelling, namely, running the program and modifying XML input files used to setup models. This introduces the general declarative XML file structure. The chapter concludes with an introduction on how to run the testing scripts and basic test models that are distributed with the code. The following chapter *Scientific Benchmarks* (page 32) introduces how to create reproducible work flow scripts that can set up, run and analyse a model using the CREDO benchmarking and analysis toolkit. Succeeding chapters talks about other functionalities in Underworld like *Journalling* (page 38), *Checkpointing* (page 43), *Timestepping* (page 46) and *Units Scaling* (page 47).

5.2 Running Underworld

Underworld is run from the command line, executing the Underworld executable with appropriate XML files and/or command line arguments.

For example, running the following line:

```
./Underworld-1.5/build/bin/Underworld ./InputFile/Arrhenius.xml
--outputPath=./uw_results/exp1
```

will read the model declarations from the file located at *./InputFile/Arrhenius.xml*, execute the program and put the output to directory *./uw_results/exp1*.

5.3 Saving Models and Data

Underworld is designed to operate in a multi-user environment, in which users can have their own model files (input and output) located in their separate home directories, but all users execute the same Underworld executable installed in a single shared location. This recommended setup is how Underworld is packaged on cluster machines and in Package binaries. Users should always keep their XML input files, and output files, in separate locations to the source code itself. This allows users to try different version of Underworld against the same input files, and ensures conflicts do not arise if the code is modified or removed.

For example, the Underworld directory structure could be:

- Input files in *~/UnderworldFiles/MyInputFiles*.
- Visualisation files in *~/UnderworldFiles/MyInputFiles/MyViewports*.
- Output subdirectories for each model within *~/UnderworldFiles/MyOutput*.

It is also wise to create a soflink to the *Underworld* executable, to save you from always having to point to its absolute path.

For example:

```
ln -s Underworld-1.5/build/bin/Underworld ./Underworld
```

5.4 Running Models

Assuming the directory structure is as specified in *Saving Models and Data* (page 15), the job can be executed as follows:

```
./Underworld ~/UnderworldFiles/MyInputFiles/Sample.xml \
--outputPath=~/UnderworldFiles/MyOutput/exp1 \
-Uzawa_velSolver_ksp_type preonly -Uzawa_velSolver_pc_type lu
```

This executes the *Sample.xml* input file, saves the output in the directory *~/UnderworldFiles/MyOutput/exp1*, and uses the direct solver (single processor only, but very fast) by passing a PETSc option through Underworld to the numerical package.

For multi-processor runs, you can execute the following:

```
mpiexec -np 2 ./Underworld ~/UnderworldFiles/MyInputFiles/Sample.xml \
--outputPath=~/UnderworldFiles/MyOutput/exp1
```

5.4.1 On a Cluster

If you wish to run Underworld on an Australian supercomputer, it's recommended to use the latest pre-installed Underworld release module installed by the development team and cluster administrators.

There are broadly three different ways to use Underworld on computer clusters:

- Use pre-installed Underworld release modules using a grid submission client.
- Use pre-installed Underworld release modules via a traditional shell login. This will require a cluster account.
- Install your own copy of Underworld via a traditional shell login. This will also require a cluster account.

For information on setting up and running Underworld on a cluster, see the following pages: Underworld Cluster, Underworld Releases and Underworld Modules.

5.4.2 On Grisu

For details on how to setup an account for Grisu on the ARCS Grid, as well as details on how to run a job, please refer to the Underworld Grid page. This section will briefly run through some of the work-flow differences when running Underworld on Grisu (Web-start Application Version 0.2.2).

In Grisu, the Underworld job submission template is used to submit and monitor an Underworld job. There is no need to create a link to an Underworld executable, or to write a PBS script to submit the job. This is all handled by the job submission template. A user may select the version of Underworld they wish to use based on the available options on their selected cluster.

See *Grisu's Main Panel* (page 17), *Grisu's Job Parameter Panel* (page 18) and *Grisu's Post Processing Panel* (page 19) for screen shots of Grisu's Underworld job submission panels.

Note: Grisu's Underworld Job submission panel also allows you to add in command-line override options to the XML.

Another point of difference in Grisu is that all paths to other XML's within your submitted XML files MUST be relative. If the other XML files are submitted with the job.

The path reference can be:

```
<include>./MyReferenceXML.xml</include>
```

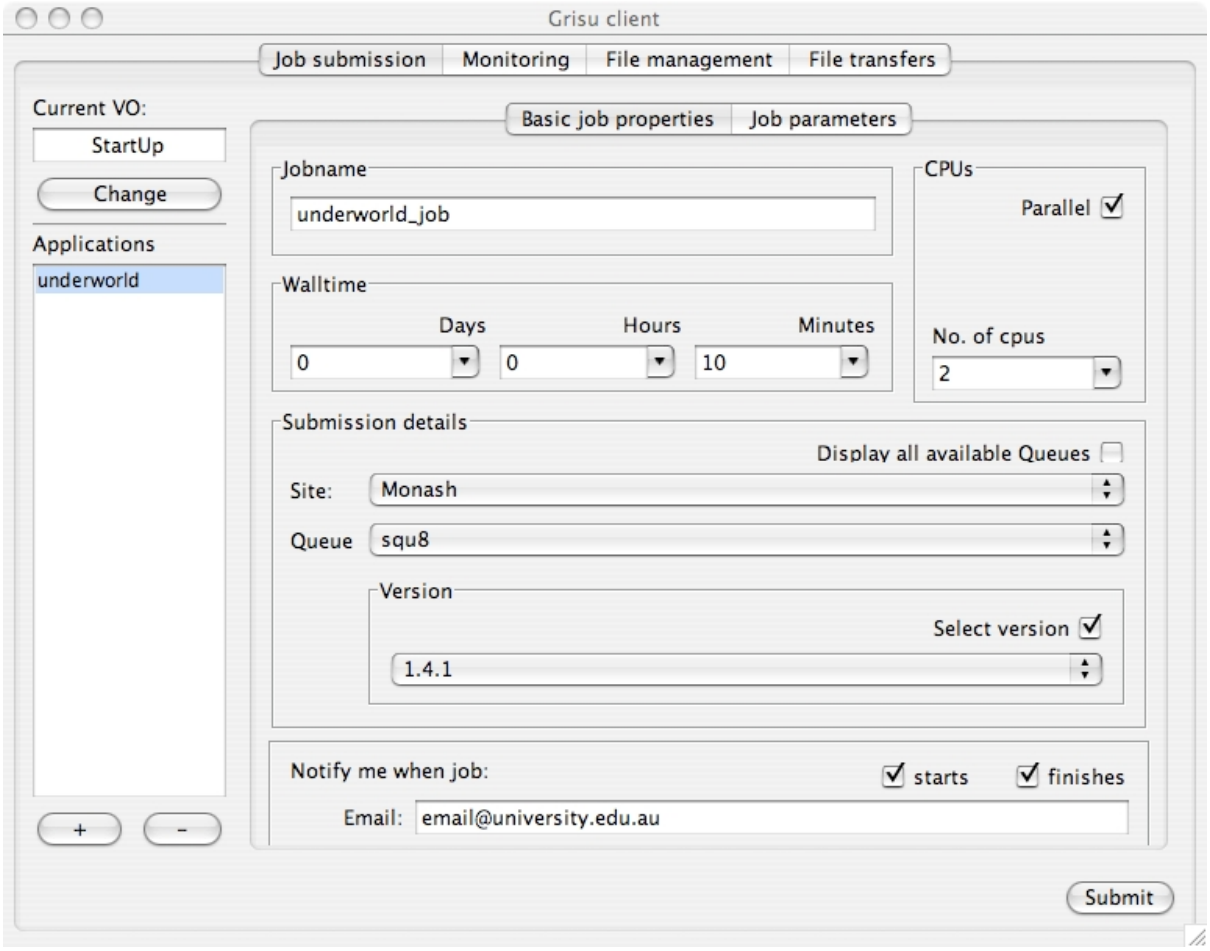


Figure 5.1: Grisu's Main Panel

If instead you wish to reuse files from previous jobs, you will need to specify the relative path to those jobs. Reasons to do this include appending results to a previous job and loading a checkpointed job.

Now you can load all these into the current job, or you can specify the path to your previous jobs. Each job will have its own directory which will contain all output files and input files for that particular job. The *grisu-jobs* subdirectory, located in a users directory of their virtual organisation's home directory, contains all the job directories.

On Grisu, you may also want to set up a directory of *Common Files* if you have a large number of Base XML's. Using relative paths you will be able to reference these in submitted jobs without having to upload them each time. You may like to set your checkpoint files to be saved in a separate output subdirectory, to make them easier to manage.

Most output from Underworld jobs can be previewed on Grisu in realtime. These files are downloaded to a local Grisu cache. Grisu can preview text files and image files but is unable to preview movie files. Any output files can also be downloaded during or after a job run has completed. All results from Grisu can be downloaded to your local machine, or an account on another cluster.

5.4.3 Using an Underworld binary package

This section demonstrates how to run Underworld models using the *Underworld* executable installed from one of the Underworld binary packages. For this example, we're using the *Underworld-1.5.2 Binary Package for Ubuntu*. It's assumed that you've downloaded the *InputFiles* tarball from the Underworld Downloads page, (see *Getting Underworld Input Files* (page 8)).

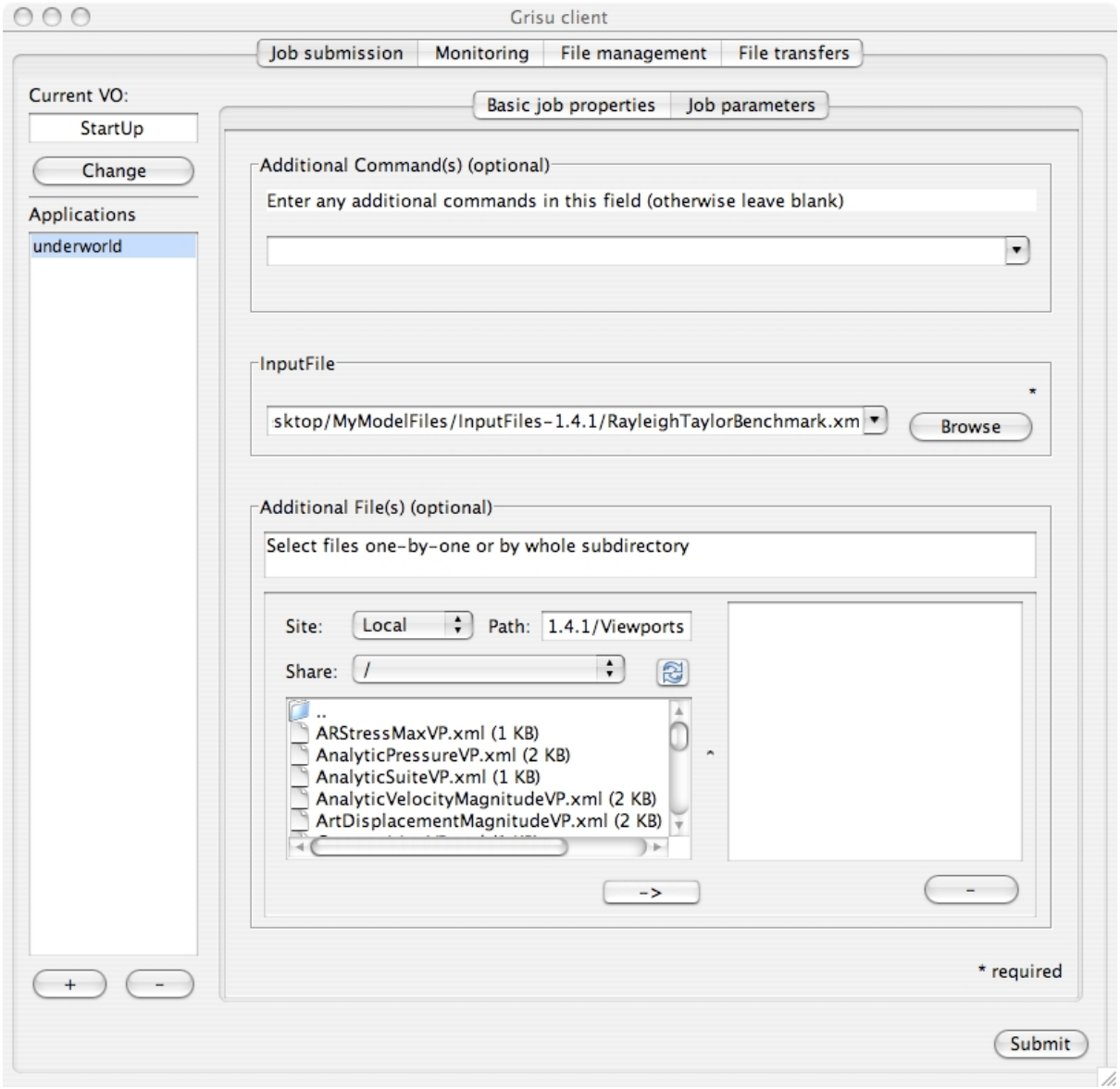
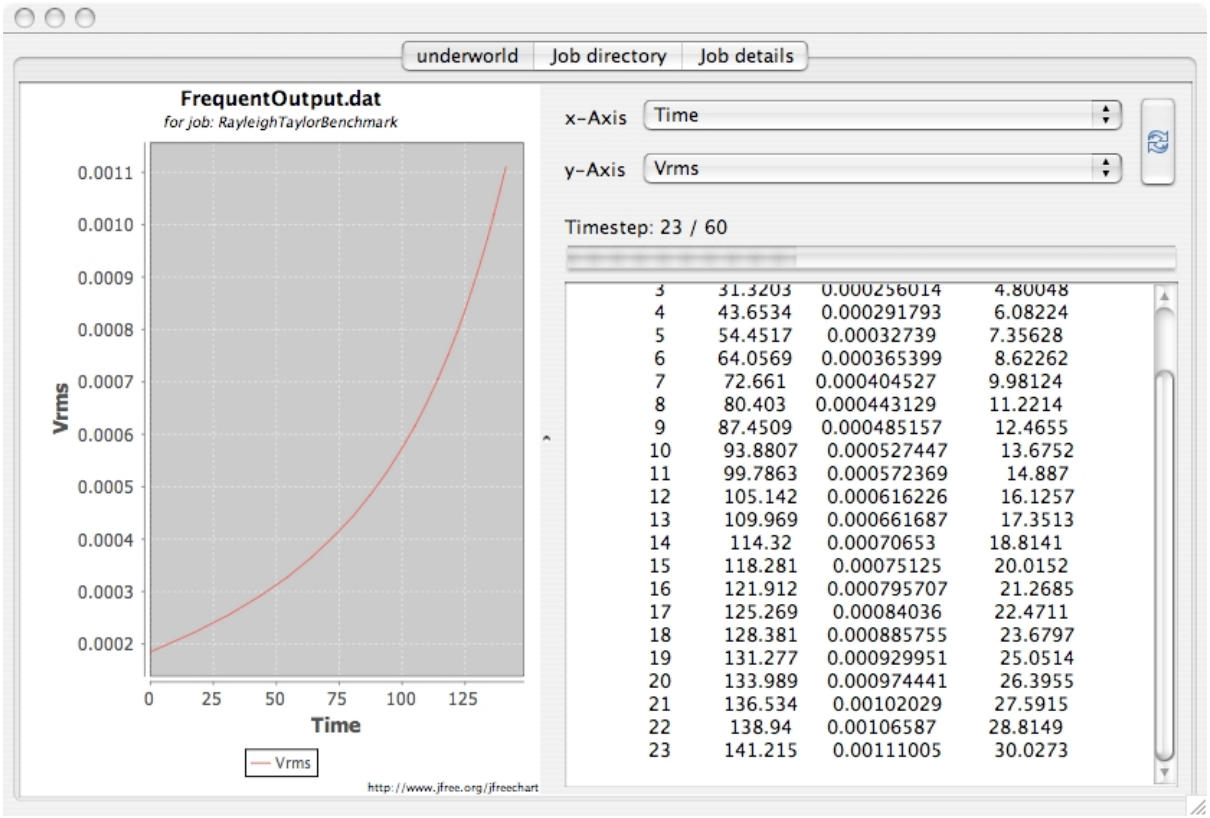


Figure 5.2: Grisu's Job Parameter Panel



3	31.3203	0.000256014	4.80048
4	43.6534	0.000291793	6.08224
5	54.4517	0.00032739	7.35628
6	64.0569	0.000365399	8.62262
7	72.661	0.000404527	9.98124
8	80.403	0.000443129	11.2214
9	87.4509	0.000485157	12.4655
10	93.8807	0.000527447	13.6752
11	99.7863	0.000572369	14.887
12	105.142	0.000616226	16.1257
13	109.969	0.000661687	17.3513
14	114.32	0.00070653	18.8141
15	118.281	0.00075125	20.0152
16	121.912	0.000795707	21.2685
17	125.269	0.00084036	22.4711
18	128.381	0.000885755	23.6797
19	131.277	0.000929951	25.0514
20	133.989	0.000974441	26.3955
21	136.534	0.00102029	27.5915
22	138.94	0.00106587	28.8149
23	141.215	0.00111005	30.0273

Figure 5.3: Grisu's Post Processing Panel

Once the *InputFiles* tarball has been untarred, you can now perform the followin operations:

- Go to the *InputFiles* directory:

```
cd ~/InputFiles/
```

- The `Underworld` executable should be already part of the environment, so you can issue the command right away:

```
Underworld Arrhenius.xml
```

  This will run the standard simulation.

- To quit the simulation at any time white it's running, you can perform:

```
<Ctrl> + c
```

Underworld's input files are written in XML format and contain all the information required to set up a simulation. In the *InputFiles* directory you will find a listing of the available standard input files. When you run a simulation using Underworld, all the output created from the simulation will be dumped into a directory in your file system. The output directory is specified in the XML InputFile you run.

The default output directory is:

```
./output
```

The output directory will contain main files depending on what output options are enabled on the input file.

The default output directory will contain the following:

- A time-stamped copy of the input files you used, combined into one XML file (flattended).
- A *FrequentOutput.dat* file that contains statistical information on the simulation run.
- Output images (*.png, *.jpg, or *.tiff) of each timestep.

- A movie of the whole simulation called *window.mpeg* if you enabled movie output on the input file.

See *Sample image output from running Arrhenius.xml in Underworld.* (page 20) for a sample image output produced by Underworld.

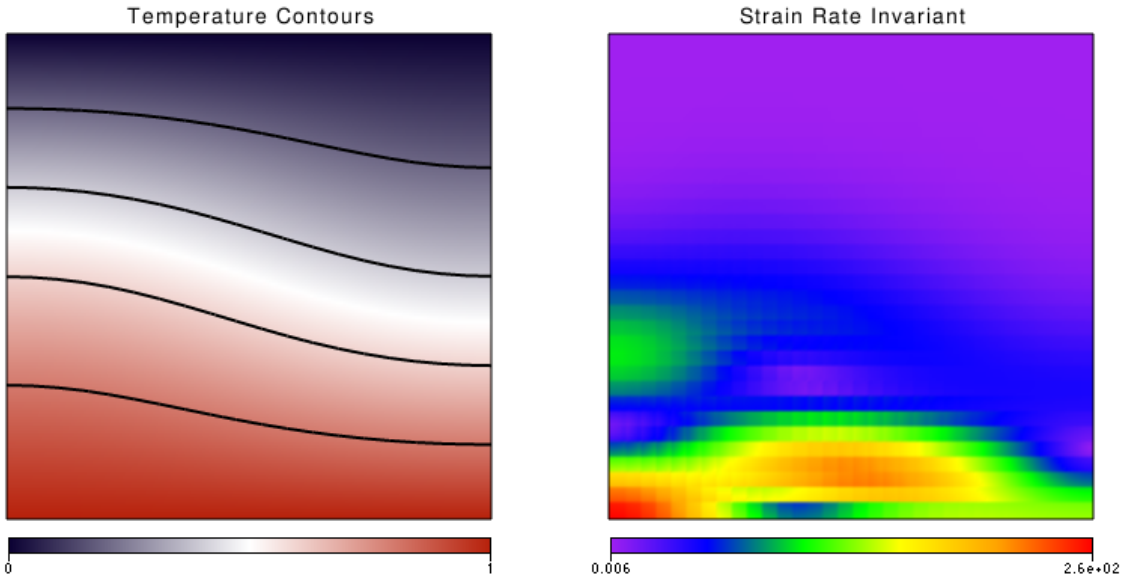


Figure 5.4: Sample image output from running *Arrhenius.xml* in Underworld.

5.5 Modelling

Underworld is designed to work using declarative programming. This is done through the XML input files and components (components are Underworld object-oriented objects). Most users should not need to look at the underlying C code. What this means for the user, is that in order to create a model, you need to *declare* all the important aspects of your model in the XML and the program handles the rest.

This design is incredibly flexible, but this can also mean that a new user can get easily lost in the number of options. For this reason, Underworld has collections of tempalte XML files that can be used to create models.

XML input files (page 20) and *Components* (page 28) will run you through the basics of what exactly input files and components are, and how to use them to create models in Underworld. Then *XML Examples* (page 25) will run through a basic tutorial on how to use and adapt the template XML input files.

5.5.1 XML input files

Underworld input files are XML files. It contain all the information and parameters needed to create your model. XML files can include other XML files, and a common setup is to have a single custom XML that calls template XML files.

Template XML files and the include command

Underworld comes with a series of XML files that activate various aspects of a model in isolation to one another, such as: *boundary conditions*, *numerical solver types*, *visualisation options*. The idea is template XML files can be swapped in and out from a main XML file to quickly and simply modify a simulation. These template XML files are located in several directories under the build directory of an Underworld installation. For example, if the Underworld installation directory is *~/myUnderworld*, the template XML files will be located in *~/myUnderworld/build/lib/StGermain/*.

The most commonly-used XML files will be located the following paths:

- *~/myUnderworld/build/lib/StGermain/Underworld* - For general pieces of functionality.
- *~/myUnderworld/build/lib/StGermain/Underworld/Viewports/* - For visualisation definitions.
- *~/myUnderworld/build/lib/StGermain/Underworld/VariableConditions/* - For boundary condition and initial condition definitions.

Template XML files can be included from other XML files using the <*include*></*include*> tag.

For example, having the following line in your XML:

```
<include>Underworld/Viewports/VelocityMagnitudeVP.xml</include>
```

will include the template file located in:

```
~/myUnderworld/build/lib/StGermain/Underworld/Viewports/VelocityMagnitudeVP.xml
```

The <*include*></*include*> tag in this case assumes the search path for the template file starts at *~/myUnderworld/build/lib/StGermain/*. Relative and absolute paths can also be used in the <*include*></*include*> tag.

For example:

```
<include>./myViewport.xml</include>
```

Creating your own XML file

The following list outlines the required steps for a user to know when using/creating XML files for Underworld:

- **Creating a blank input file** - Users should have their own XML files located in their home directory which links to the template XML files downloaded from the repository. This ensures that conflicts do not arise when running updates to Underworld and reduces the amount of work the user needs to produce their own working models.

  An empty XML file will require the root <*StGermainData*></*StGermaindata*> tag.

  For example:

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

...

</StGermainData>
```

- **Including a template XML file** - Users can add aspects to their model by including template XML files.

  For example, after including the *Rayleigh-Taylor Benchmark* template XML, the XML should look something like this:

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

  <include>$PATHTOUNDERWORLD/Underworld/InputFiles/
  RayleighTaylorBenchmark.xml</include>

</StGermainData>
```

  Where *$PATHTOUNDERWORLD* is the path to the Underworld directory.

  For example:

```
<include>/home/Me/Underworld_1.5/Underworld/InputFiles/
RayleighTaylorBenchmark.xml</include>
```

If you are on a cluster, you will be using a reference to your locally checked out input files directory, (see *Getting Underworld Input Files* (page 8)).

- **Adding or Overwriting a component** - Users can change or add component parameters by simply entering them in the XML file. Relevant sections from a template XML file can be copied and its parameters changed.

  For example:

```
<struct name="components" mergeType="merge">
   <struct name="name in InputFile" mergeType="merge">
      <param name="parameter to overwrite" mergeType="replace">value</param>
   </struct>
</struct>
```

  In most cases, the name of the *Type* matches the file name of the component.

  The example below changes the background viscosity:

```
<struct name="components" mergeType="merge">
   <struct name="backgroundViscosity" mergeType="merge">
      <param name="eta0" mergeType="replace">2.0</param>
   </struct>
</struct>
```

- **Adding or Overwriting a variable** - Users can add their own variable parameters in their XML files.

  The example below changes the location of the output directory:

```
<param name="outputPath">./outputNew/</param>
```

  Make sure you are NOT within the *components* struct. Variables are usually added towards the end of the file, after the *components* struct.

Standard XML input file format

A typical XML input file is composed of the following sections:

- **StGermainData tag** - This section *opens* the XML file, letting the compiler know that it is an XML file, and that it is a *StGermainData* XML type.

  For example:

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

...

</StGermainData>
```

  Where ... represents entries mentioned below.

- **A list of toolboxes to import** - The toolbox listed here will be the highest framework in the hierarchy that the model will use. For most users, all necessary programs will be loaded by default and will not need to be listed here.

  For example:

```
<list name="import">
   <param>Underworld</param>
</list>
```

- **Context of the model** - The user must specify the *context* of the model. This must be done first before other components are read into the dictionary.

  For example:

```
<struct name="component" mergeType="merge">
   <struct name="context">
       <param name="Type">UnderworldContext</param>
   </struct>
</struct>
```

This should be added directly after *<list name="import">...</list>* block.

- **A list of other XML files to include** - These are the template XML files that help to define exactly what the model is doing. Underworld is set up to use a hierarchy of XML files so that your XML doesn't need to have everything in one file.
- **A list of plugins to include** - These plugins provides additional functionality that can be be included in the input file. Plugins can define what values to output. They can also modify how the model operates.

  For example:

```
<list name="plugins" mergeType="merge">
   <struct>
      <param name="Type">PluginName</param>
      <param name="Context">context</param>

      ...

   </struct>
</list>
```

Where ... represents arguments to pass into the plugin. This structure requires the user to know what parameters to pass to a plugin. As with the *components*, parameters passed in are error checked and the code will halt and report a problem if invalid parameters are found.

A complete example:

```
<list name="plugins" mergeType="merge">
   <struct>
      <param name="Type">StgFEM_FrequentOutput</param>
      <param name="Context">context</param>
   </struct>
   <struct>
      <param name="Type">Underworld_Vrms</param>
      <param name="Context">context</param>
      <param name="GaussSwarm">gaussSwarm</param>
      <param name="VelocityField">VelocityField</param>
   </struct>
   <struct>
      <param name="Type">StgFEM_CPUTime</param>
      <param name="Context">context</param>
   </struct>
   <struct>
      <param name="Type">lucPlugin</param>
      <param name="Context">lucPluginContext</param>
   </struct>
</list>
```

Plugins can be found inside the *StgFEM/plugins*, *PICellerator/plugins*, and *Underworld/plugins* directories. To include a plugin, use the plugins' name within the params struct. If referencing a template XML file, your input file will inherit all the template's plugins.

- **A list of components** - These are the values, parameters and shapes the specified model.

  For example:

```
<!-- Specialised components for this particular model -->
<struct name="components" mergeType="merge">
   <struct name="shape">
```

```
        <param name="Type">Everywhere</param>
    </struct>
</struct>
```

- **Simulation control parameters** - These parameters specify variables such as: how long the model will run; when values are saved; whether to restart from previously checkpointed values; where the output should be located; how many iterations should the linear and non-linear solvers attempt per timestep; and other *single value* scientific parameters important to the simulation.

  For example:

```
<!-- Simulation control -->
<param name="maxTimeSteps">10</param>
<param name="outputPath">./output</param>
<param name="dumpEvery">100</param>
<param name="checkpointEvery">100</param>
```

  Where *maxTimeSteps* defines the maximum number of timesteps, *outputPath* defines the directory in which to save all data, *dumpEvery* defines the timestep increment at which data is outputted, and *checkpointEvery* defines how often to checkpoint data.

- **Journal settings** - The journal controls the information that is *printed to screen* during the simulation. If running remotely on a HPC cluster, it is written to the standard output file.

  For example:

```
<!-- Control Journal output -->
<param name="journal.info.Context-verbose">True</param>
<param name="journal-level.info.Stg_ComponentFactory">2</param>
<param name="journal.debug">f</param>
```

- **Geometry and mesh setup** - This block outlines the physical parameters of the model: box, size, dimensions and resolution.

  For example:

```
<!-- Geometry & Mesh setup -->
<param name="dim">2</param>
<param name="elementResI">32</param>
<param name="elementResJ">32</param>
<param name="shadowDepth">1</param>
<param name="minX">0.0</param>
<param name="minY">0.0</param>
<param name="maxX">1.0</param>
<param name="maxY">1.0</param>
```

- **Initial and boundary conditions** - This section will list the template XML files that define initial conditions *(ICs)* and boundary conditions *(BCs)* for the model. For a listing of available ICs and BCs see the directory *Underworld/InputFiles/VariableConditions*.

  For example:

```
<include>Underworld/InputFiles/VariableConditions/temperatureBCs.xml
</include>
```

- **Visualisation settings** - This section will either list the XML files that define the visualisation settings for the model, or will actually list the parameters and values of the visualisation. It is common practice to regroup all the parameters relative to a Viewport in another file, that will live in the *Viewports* directory (a sub-directory of the *InputFiles* directory where the template XML files are located). This avoids lengthy input files and allows some viewports settings to then be re-used in other input files.

  For example:

```
<include>Underworld/Viewports/TemperatureAndVelocityArrowsVP.xml</include>
```

If enabling gLucifer visualisation and not including visualisation template XML files, the following line needs to be added into the input file:

```
<include>glucifer/window.xml</include>
```

XML Examples

This section will run through a few simple Underworld models to give the user a feel for how Underworld works. It will only focus on using and modifying input XML files. XML files used in this section are only example input files, and some parameter alterations may be needed in order to do specific scientific modelling jobs.

The following examples assumes that the user is working with a checkout of Underworld on a local machine. The initial input file will be called *Example.xml* and is based on the *RayleighTaylorBenchmark.xml* XML file as per defined in the *<include></include>* tag.

The *Example.xml* file will have the initial XML declarations:

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

   <include>$PATHTOUNDERWORLD/Underworld/InputFiles/RayleighTaylorBenchmark.xml
   </include>
   <list name="plugins" mergeType="merge">
      <!-- Output Plugins -->
   </list>
   <struct name="components" mergeType="merge">

   ...

   </struct>

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

</StGermainData>
```

Where *$PATHTOUNDERWORLD* is the path to the Underworld directory, (see *Creating your own XML file* (page 21)). The ... inside the *components* block will be replaced by various component definitions in the following examples:

- **Viscous Material Example** - In this example, the entire region is assigned a single viscous material.

  1. Add the shape:

```
<struct name="backgroundShape" mergeType="replace">
   <param name="Type">Everywhere</param>
</struct>
```

  2. Set the material viscosity:

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

     MaterialViscosity is a predefined type in Underworld, it requires a value for *eta0*.

  3. Add the material:

```
<struct name="background" mergeType="replace">
   <param name="Type">RheologyMaterial</param>
   <param name="Shape">backgroundShape</param>
```

```
    <param name="density">1.0</param>
    <param name="Rheology">backgroundViscosity</param>
</struct>
```

RheologyMaterial is a predefined type of material. It requires: a *Shape*, defined in backgroundShape); a *density*, defined as 1.0; and a *Rheology*, defined in *backgroundViscosity*.

4. Save the resulting *Example.xml* file. These declarations will override the default values in *RayleighTaylorBenchmark.xml*. It will now override any other values not re-defined by the declarations

- **Multiple Viscous Materials Example** - This example will show how to create more than one viscous material in a simulation. This will use the *Example.xml* From the *Viscous Material Example*.

1. Add the shape:

```
<struct name="lightLayerShape" mergeType="replace">
    <param name="Type">Sphere</param>
    <param name="CentreX">0.5</param>
    <param name="CentreY">0.5</param>
    <param name="radius">0.10</param>
</struct>
```

This simulation will be run in 2D, so the result is a circle.

2. Add the material:

```
<struct name="lightLayerViscosity" mergeType="replace">
    <param name="Type">MaterialViscosity</param>
    <param name="eta0">1.0</param>
</struct>
<struct name="lightLayer" mergeType="replace">
    <param name="Type">RheologyMaterial</param>
    <param name="Shape">lightLayerShape</param>
    <param name="density">0.1</param>
    <param name="Rheology">lightLayerViscosity</param>
</struct>
```

3. Override the shape of the original material so that it doesn't include the sphere. The new shape is created from the old shape minus the sphere:

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

4. Modify the viscous material *background* to use *nonSphereShape* as its *Shape*.

From this:

```
<struct name="background" mergeType="replace">
    <param name="Type">RheologyMaterial</param>
    <param name="Shape">backgroundShape</param>
    <param name="density">1.0</param>
    <param name="Rheology">backgroundViscosity</param>
</struct>
```

To this:

```
<struct name="background" mergeType="replace">
   <param name="Type">RheologyMaterial</param>
   <param name="Shape">nonSphereShape</param>
   <param name="density">1.0</param>
   <param name="Rheology">backgroundViscosity</param>
</struct>
```

5. Save the resulting Example.xml file.

See *Sample image output from running Example.xml in Underworld.* (page 28) for an example output image from running *Example.xml* in Underworld.

- **Parameters Only Example** - This last example shows how to create a *parameters only* XML file that can be used to override existing parameters in other XML files. This is useful for creating variations of a standard model and experimenting with its parameters. For this example, *Parameters.xml* will be created. As a starting point, it will have the same content as the base *Example.xml* file in *XML Examples* (page 25).

  1. Remove the *<include>...</include* line from the file. This include line is not required as *Parameters.xml* will be ran together with *Example.xml*.
  2. Remove the *<param name="outputPath">./output/</param>* line from the file. This will be re-added in the next step at a different order.
  3. Add parameters towards the end of the file and not inide the *components* block:

```
<!-- Simulation Parameters -->
<param name="outputPath">./MyPath/</param>
<param name="maxTimeSteps">60</param>
<param name="dumpEvery">1</param>
<param name="checkpointEvery">0</param>

<!-- Swarm Parameters -->
<param name="particleLayoutType">random</param>
<param name="particlesPerCell">20</param>

<!-- Mesh Marameters (Model Resolution)-->
<param name="elementResI">64</param>
<param name="elementResJ">64</param>
<param name="elementResK">1</param>

<!-- Model Parameters -->
<param name="gravity">1.0</param>
<param name="dim">2</param>

<!-- Box Size -->
<param name="minX">0.0</param>
<param name="minY">0.0</param>
<param name="minZ">0.0</param>
<param name="maxX">1.0</param>
<param name="maxY">1.0</param>
<param name="maxZ">1.0</param>
```

  4. Modifiable parameters on each component used in the model can be included. For example, shape, viscosity and density parameters can be included.

     Inside the *components* struct, the parameters can be adjusted by changing the shape of the sphere:

```
<struct name="lightLayerShape" mergeType="merge">
   <param name="CentreX" mergeType="replace">0.5</param>
   <param name="CentreY" mergeType="replace">0.5</param>
```

```
    <param name="radius" mergeType="replace">0.30</param>
</struct>
```

Or changing the viscosities of each shape:

```
<struct name="backgroundViscosity" mergeType="merge">
    <param name="eta0" mergeType="replace">2.0</param>
</struct>
<struct name="lightLayerViscosity" mergeType="merge">
    <param name="eta0" mergeType="replace">2.0</param>
</struct>
```

Or changing the densities:

```
<struct name="background" mergeType="merge">
    <param name="density" mergeType="replace">2.0</param>
</struct>
<struct name="lightLayer" mergeType="merge">
    <param name="density" mergeType="replace">0.1</param>
</struct>
```

5. Save the resulting *Parameters.xml* file. This file can then be ran together with *Example.xml* in Underworld.

   For example:

   ./Underworld Example.xml Parameters.xml

   See *Sample image output from running Example.xml and Parameters.xml in Underworld.* (page 29) for an example output image from running *Example.xml* and *Parameters.xml* in Underworld.

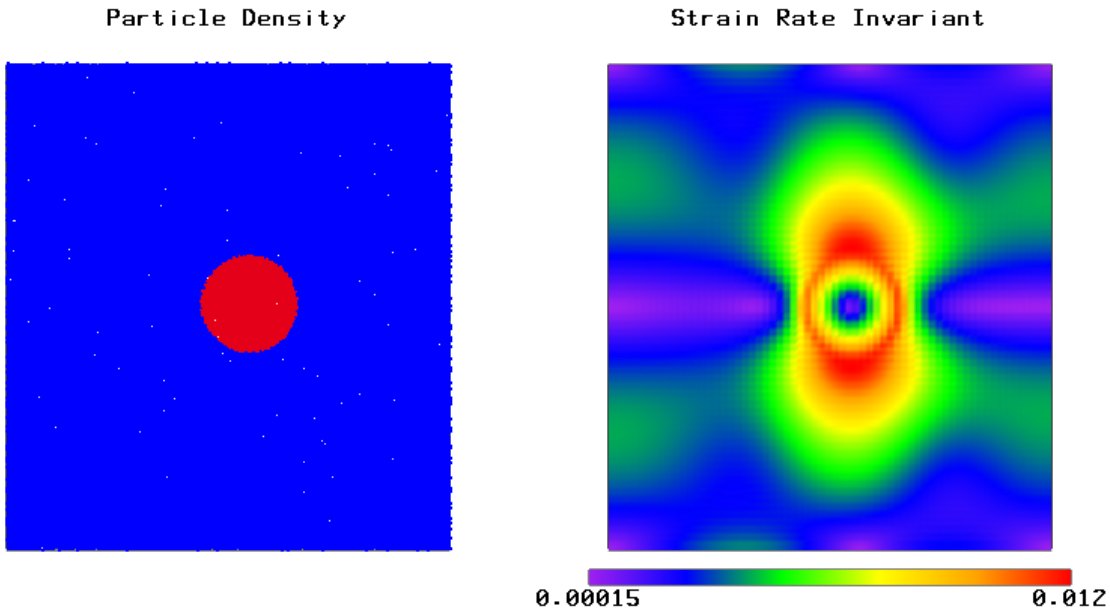


Figure 5.5: Sample image output from running *Example.xml* in Underworld.

5.5.2 Components

Components form the major part of an Underworld model. Each template XML file creates objects of different types, based on components. Each component, composed of one *.h* and one *.c* file, defines a set of parameters with default values. If the values of the parameters are not redefined by the type within the template InputFile or your own input file (which calls a template XML file), the default values from the components will be applied when the

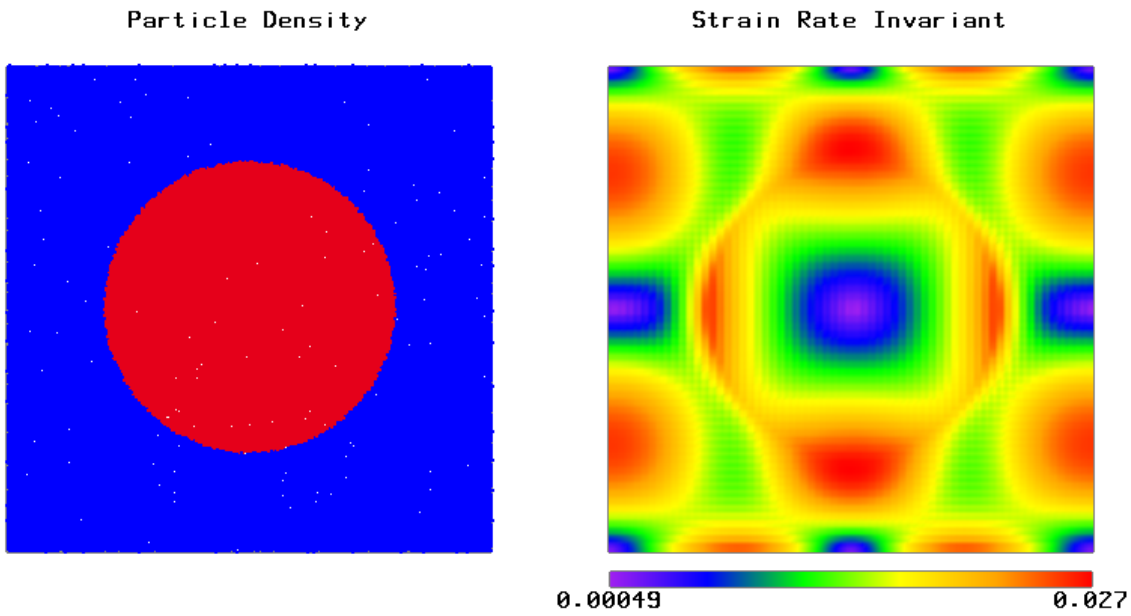


Figure 5.6: Sample image output from running *Example.xml* and *Parameters.xml* in Underworld.

model is run. Components are created and linked together at run-time. Which components are created and what properties they will have are defined within the XML input files.

Where are they located?

Components are defined throughout the code base in all levels of the hierarchy. If you are interested in looking up the properties of an individual component, the Underworld Documentation page contains links to the component codex for each *stable release* of Underworld.

Component Examples

This section provides some examples on the use of components in the XML files.

- **Defining a Geometric Shape in the Model Domain** - In broad terms, the shape of an object to be modelled must be defined prior to its material properties e.g. density, rheology. Users can choose from a series of preset shapes or build their own based on those presets.

  For example:

```
<struct name="backgroundShape" mergeType="replace">
   <param name="Type">Everywhere</param>
</struct>
```

  The component codex has detailed examples of the XML entries for the different available shapes. Users can look at the component *Stg_Shape* in the StGermain component codex.

- **Union** - Shapes can be combined using a *Union*. A new shape called *shapeUnion* will be created by combining *mainShape* and *extraShape*.

  For example:

```
<struct name="shapeUnion">
   <param name="Type">Union</param>
   <list name="shapes">
      <param>mainShape</param>
      <param>extraShape</param>
   </list>
</struct>
```

Shapes being combined in a union must have already been definea. It could be defined above the actual union or in an included XML file.

- **Intersection** - A void can be created where two shapes intersect. A new shape *intersectionShape* which includes *mainShape* is created but excludes the region defined by *voidShape*.

  For example:

```
<struct name="shapeIntersection">
   <param name="Type">Intersection</param>
   <list name="shapes">
      <param>mainShape</param>
      <param>!voidShape</param>
   </list>
</struct>
```

  Where *!* in front of *voidShape* means *not*.

- **Material Definition** - Once a shape has been defined, users can specify its physical properties such as rheology and density.

  For example:

```
<struct name="mainShapeViscosity">
   <param name="Type">MaterialViscosity</param>
   <param name="eta0">1.0</param>
</struct>
<struct name="materialOne">
   <param name="Type">RheologyMaterial</param>
   <param name="Shape">mainShape</param>
   <param name="density">1.0</param>
   <param name="Rheology">mainShapeViscosity</param>
</struct>
```

5.5.3 Override Parameters from the Command Line

Users can override all XML parameters from the command line when running the *Underworld* executable.

For example:

```
./Underworld Example.xml --outputPath=./newPath
```

Command line options can be used for any XML parameter in Underworld as well as for selecting some PETSc solver options.

For example:

- Override a parameter - *–elementResI=5*.
- Navigate through the struct hierarchy - *–components.lightLayerShape.wavelength=1.7*. Dots . are used to navigate down through the struct hierarchy.
- Access lists:
  - To override an entry - *–plugins[2].Context=context2*.
  - To add an entry - *–FieldVariablesToCheckpoint[]=PressureField.*

  Square brackets *[]* are used to select or add items into a list.

5.6 Tests

Tests are an important part of any software project, both for ensuring an installation is running as expected, and for providing examples of expected behaviour. Underworld provides an extensive set of tests, grouped into several categories.

5.6.1 Running

The tests in Underworld are currently categorised into *unit*, *integration* and *convergence* tests. The tests are spread across StGermain, StgDomain, StgFEM, PICellerator and Underworld. The tests can be executed from the root directory of the Underworld installation where *./scons.py* and *./configure.py* are normally executed. For example, in *$PATHTOUNDERWORLD*.

The following list outlines the type of test suites and how to run them:

- *./scons.py check* - This will run the unit and low-resolution integration tests only.
- *./scons.py check-complete* - All the configured tests (unit, low-resolution and normal-resolution integration, convergence).
- *./scons.py check-unit* - Unit tests only.
- *./scons.py check-convergence* - Convergence tests only.
- *./scons.py check-integration*- Normal-resolution integration tests only.
- *./scons.py check-lowres* - Low-resolution version of the integration tests only.

5.6.2 Unit Tests

The unit tests are encapsulated into a binary executable for each of the components: StGermain, StgDomain, StgFEM, PICellerator and Underworld and can be located under the *$PATHTOUNDERWORLD/build/tests* directory.

Executing the following commands will go to the unit test directory and run the test suites for each of the components:

```
cd $PATHTOUNDERWORLD/build/tests
./testStGermain
./testStgDomain
./testStgFEM
./testPICellerator
./testUnderworld
```

Underworld uses a custom unit testing system called PCU *(Parallel C-Unit)*, to provide and manage these tests. More information about PCU, particularly relevant for those developing new tests, can be found at the PCU Documentation.

5.6.3 CREDO

CREDO is benchmarking and analysis tool for Underworld. It provides methods and workflows on running Underworld. See the Underworld Documentation page for links on the CREDO documentation.

CREDO provies functionality to users to:

- Run one or more Underworld models in a defined, repeatable way.
- Define overrides, or modifications to apply to the models.
- Require certain analysis operations, or tests, to be applied to the models.
- Post-process the results of the Underworld outputs for scientific analysis.

Example CREDO script and workflow can be found at:

```
$PATHTOUNDERWORLD/Underworld/InputFiles/credo-rayTayBasic.py
```

5.7 Scientific Benchmarks

This chapter describes benchmark models available in Underworld as well as instructions on setting up and running the science benchmarks. A few examples with their results are also included in this chapter.

5.7.1 Models

Available models for benchmarking include (but are not limited to):

- Geothermal
- Groundwater
- Lithosphere Instabilities
- Magma
- Mantle Convection
- Numerical Capabilities
- Melt
- Subduction
- Surface Processes

See the Underworld Models page for a list of of available models in Underworld.

5.7.2 Science Benchmarks with CREDO

The CREDO workflow tool, mentioned in the context of scripts to run Underworld tests (see *Tests* (page 30)), can also be used to run and report on Underworld benchmarks. One of the design goals of CREDO is the ability to treat benchmarks as a special type of system test, that requires a higher level of scientific description and analysis.

See the CREDO Configuration page for more detailed information on how to write and run benchmarks. The benchmarks currently available are located in the *SysTest/ScienceBenchmarks* sub-directory of each project. In Underworld's case, this includes several *Rayleigh-Taylor* and *Thermal Convection* benchmarks.

5.7.3 Benchmark Examples

Rayleigh-Taylor Benchmark

This benchmark models the evolution of a sinusoidal Rayleigh-Taylor instability. It has a lower density fluid on the bottom of the box, and a higher density fluid above.

The XML file for the benchmark is *Underworld/InputFiles/RayleighTaylorBenchmark.xml*.

Detail on the benchmark can be found in [VanKeken1997] (page 95).

- **Components** - The free-slip-sides boundary condition *Underworld/InputFiles/RayleighTaylor.xml*, which applies a zero condition perpendicular to the box walls, restricting movement so that a material can flow along a box wall but not away from it, but also applies a zero condition along the top and bottom walls restricting movement along these walls in the *X* direction.
- **General Equations** - In both analytic and numerical approaches to modelling such phenomena, the mathematical description has been a Rayleigh-Taylor instability - a gravitational instability of a layer of heavy fluid overlying a lighter one,

$$\nabla \cdot \tau - \nabla p = -\mathbf{g}\rho\hat{z} \tag{5.1}$$

  Where τ is deviatoric stress, p is pressure, $\mathbf{g}$ is gravity, ρ is density, and $\hat{z}$ is the unit vector in z direction.

For approximately incompressible materials (Boussinesq approximation),

$$\nabla \cdot \mathbf{u} = 0 \tag{5.2}$$

Where $\mathbf{u}$ is velocity.

The advection equation for composition is

$$\frac{\partial \Gamma}{\partial t} + (\mathbf{u} \cdot \nabla)\Gamma = 0 \tag{5.3}$$

Where Γ is composition, t is time, and $\mathbf{u}$ is velocity.

- **Results** - The following material properties were used in this sample:
  - *viscosity* = 1.0 (background and light layer) (dimensionless)
  - *density* = 1.0 (background), 0.0 (light layer) (dimensionless)
  - *resolution* = 64 x 64

  See *Rayleigh-Taylor Benchmark at 1st Timestep (2nd image)* (page 33), *Rayleigh-Taylor Benchmark at 638th Timestep (2001.22 dimensionless time)* (page 34) and *Graph of Dimensionless Time vs. Velocity RMS (VRMS): of Rayleigh-Taylor Benchmark* (page 34) for output images produced using Underworld-1.0.0.

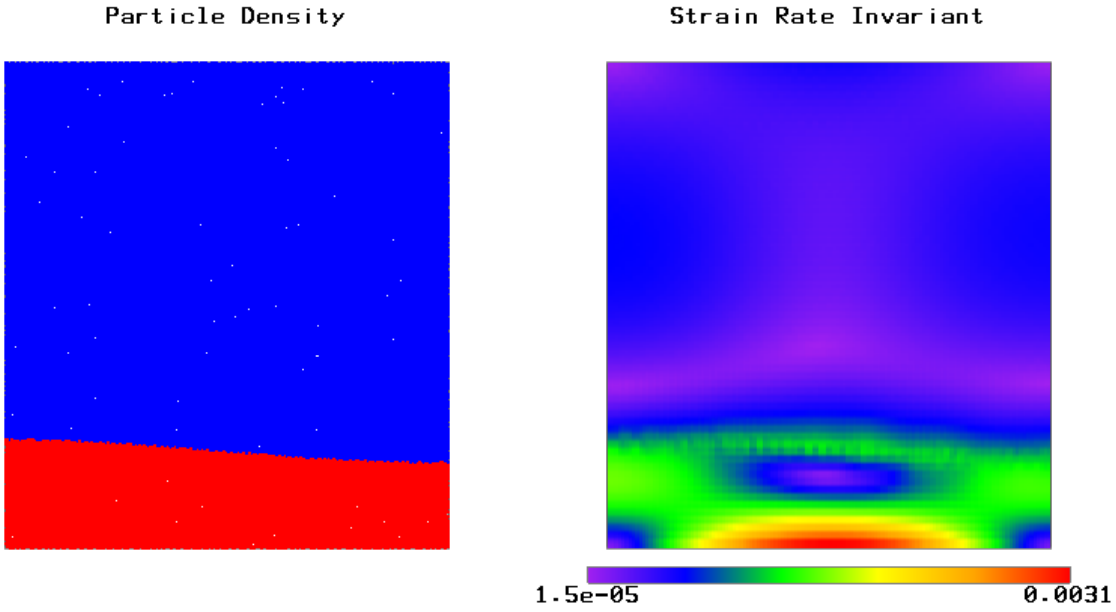


Figure 5.7: Rayleigh-Taylor Benchmark at 1st Timestep (2nd image)

Thermo-Chemical Benchmark

The D" (core-mantle boundary) of the Earth is modelled with the ThermoChemBenchmark model. A dense layer is placed at the bottom of the box. This models the entrainment of a low lying dense layer.

The XML file for the benchmark is *Underworld/InputFiles/ThermoChemBenchmark.xml*.

Detail on the benchmark can be found in [VanKeken1997] (page 95).

- **General Equations** - We solve the equations of thermal convection for a creeping fluid with the Bousinnesq approximation and infinite Prandtl number.

  The deviatoric stress tensor, τ_{ij} is given by,

$$\tau_{ij} = \eta \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \tag{5.4}$$

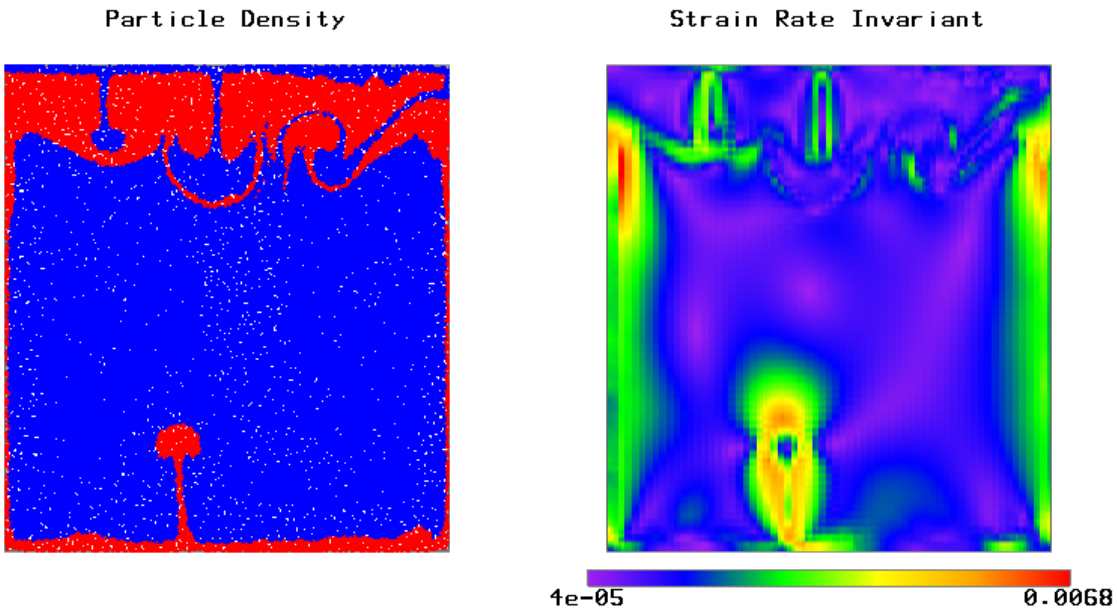


Figure 5.8: Rayleigh-Taylor Benchmark at 638th Timestep (2001.22 dimensionless time)

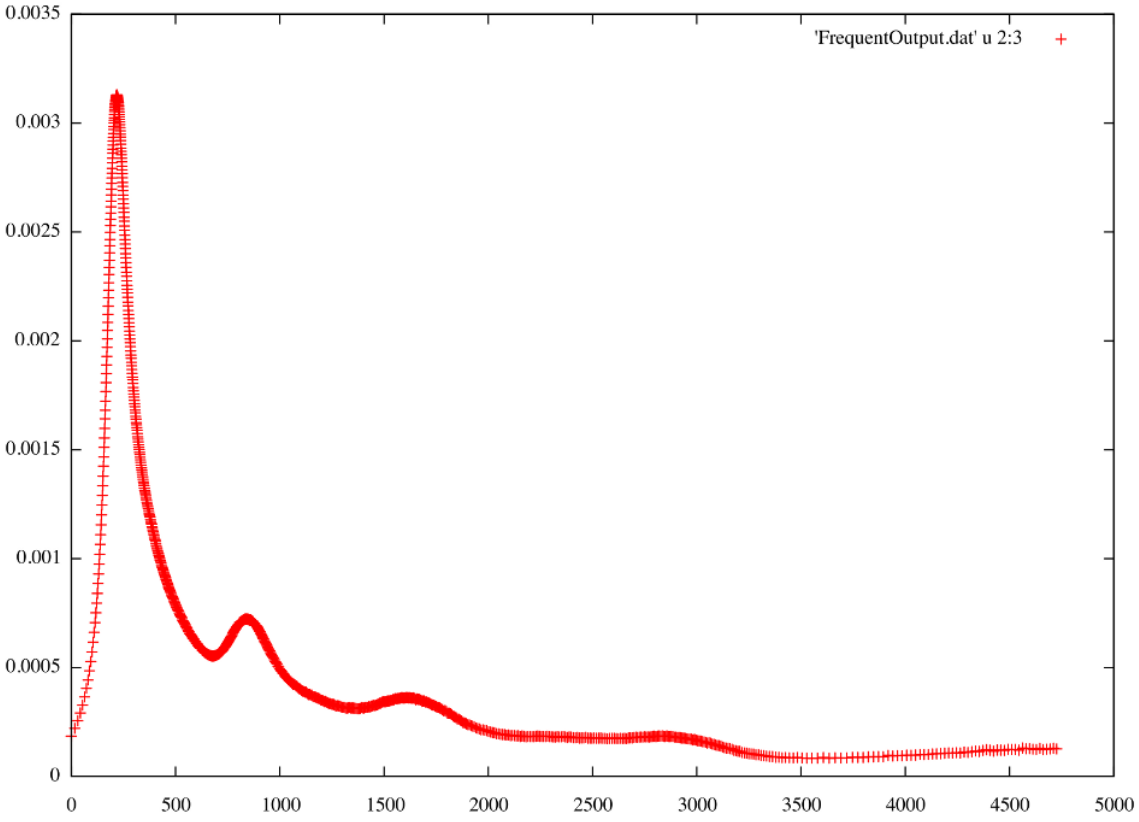


Figure 5.9: Graph of Dimensionless Time vs. Velocity RMS (VRMS): of Rayleigh-Taylor Benchmark

Where η is the viscosity. This is a constitutive relationship.

The surface temperature is $T_0 = 0$ and the bottom temperature is $T_l = \Delta T$. All boundaries are free-slip.

Choosing the following scales,

$$x_i \rightarrow lx_i, \tag{5.5}$$
$$T \rightarrow \Delta T T, \tag{5.6}$$
$$u_i \rightarrow \frac{\kappa}{l} u_i \tag{5.7}$$

results in a non-dimensional form of the equations, given as,

$$-\frac{\partial \tau_{ij}}{\partial x_j} + \frac{\partial p}{\partial x_i} = \rho g (1 - \alpha T)\lambda_i, \tag{5.8}$$
$$\frac{\partial u_i}{\partial x_i} = 0, \tag{5.9}$$
$$\frac{\partial T}{\partial t} + u_i \frac{\partial T}{\partial x_i} = \frac{\partial^2 T}{\partial x_j^2} \tag{5.10}$$

Where ρ and α are material properties.

- **Components** - The base application is *ThermoChem.xml*, but in ThermoChemBenchmark, *BuoyancyFormTerm* is replaced with *BuoyancyForceTermThermoChem*.

  The temperature boundary condition applies a temperature of 0 (zero) to the top, and 1 (one) to the bottom of the box (dimensionless values). A dense layer is placed at the bottom of the box, and then entrained by thermal convection. This model solves for an energy equation, as well as the standard Stokes Equation.

  The following line is added to the model's input file:

```
<include>Underworld/VariableConditions/temperatureBCs.xml</include>
```

  The free-slip boundary condition applies a zero condition perpendicular to the box wall, which restricts movement so that a material can flow along a box wall but not away from it. In the example , this condition is applied to every box wall (i.e., in the 3D case, top, bottom, left, right, front and back). This produces the effect of a mirror of the box outside the box wall to which this boundary condition is applied.

  The following line is added to the model's input file:

```
<include>Underworld/VariableConditions/velocityBCs.freeslip.xml</include>
```

  The thermochemical convection template input files contain an analytic temperature initial condition.

- **Results** - The following material properties were used in this sample:
  - *viscosity* = 1.0 (ambient and dense layer) (dimensionless)
  - *density* = 0.0 (ambient), 1.0 (dense layer) (dimensionless)
  - *RaT* = 3.0e5
  - *RaC* = 4.5e5
  - *alpha* = 1.0 (ambient and dense layer) (dimensionless)
  - *courantFactor* = 0.25
  - *resolution* = 64 x 32

  See *Thermo-Chemical Benchmark at 10th Timestep (11th image)* (page 36), *Thermo-Chemical Benchmark at 2080th Timestep (0.0200099 dimensionless time)* (page 36) and *Graph of Dimensionless Time vs. Velocity RMS (VRMS) for Thermo-Chemical Benchmark* (page 37) for output images produced using Underworld-1.0.0.

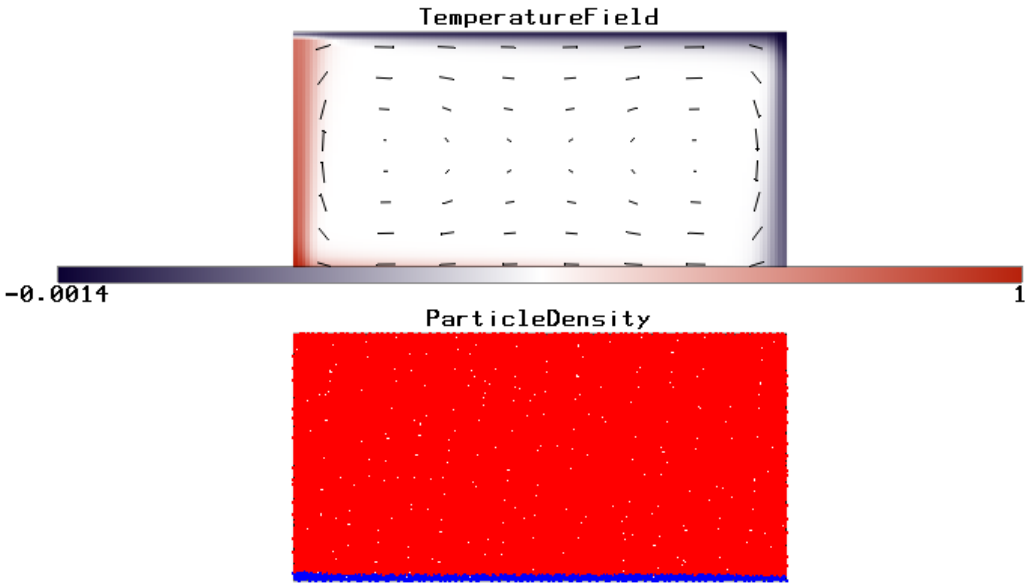


Figure 5.10: Thermo-Chemical Benchmark at 10th Timestep (11th image)

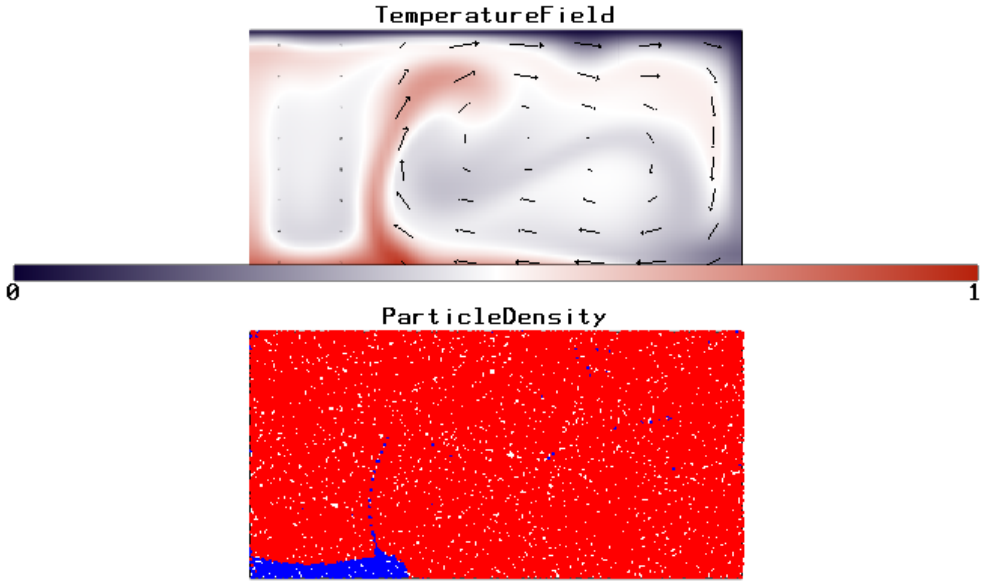


Figure 5.11: Thermo-Chemical Benchmark at 2080th Timestep (0.0200099 dimensionless time)

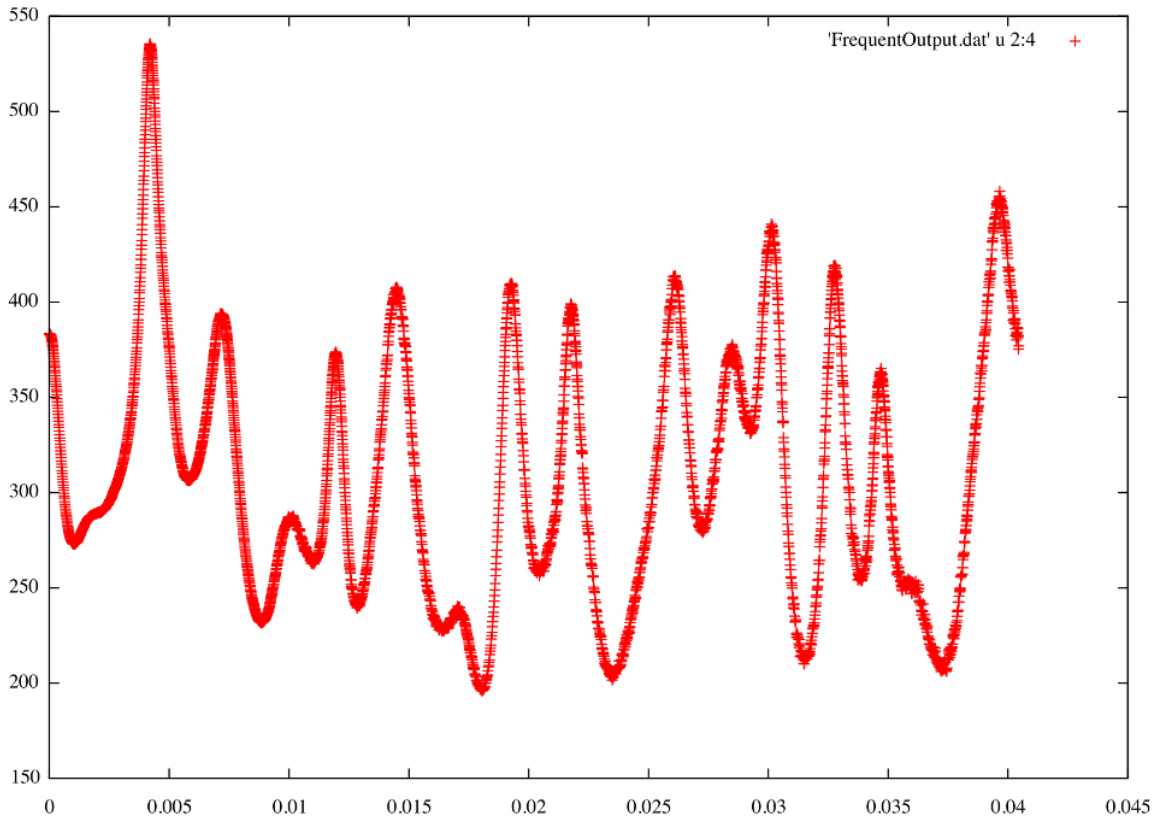


Figure 5.12: Graph of Dimensionless Time vs. Velocity RMS (VRMS) for Thermo-Chemical Benchmark

Faulting Moresi-Mühlhaus-2006 3D Extension Model

This model simulates the yielding of the earth's crust in three dimensions based on the Mohr-Coulomb criterion, according to the FaultingMoresiMuhlhaus2006 model. One (2D) or two (3D) weak zones are placed in the top of the upper mantle in order to facilitate the localisation of the deformation near the weak zone(s).

The XML files for the benchmark are:

- *Underworld/InputFiles/ExtensionFMM.xml*
- *Underworld/InputFiles/ExtensionFMM3D.xml*

Although the yielding component makes use of a Mohr-Coulomb type failure criterion, it also implies other physical assumptions.

Detail on the formulation can be found in [MoresiMuhlhaus2006] (page 95).

- **Components** - The base application of the model is *Underworld/InputFiles/ExtensionBaseApp.xml*. The extension boundary condition is free-slip, with velocity conditions applied to the left (-0.5) and right (0.5) walls of the box to allow for extension (dimensionless values).

  The conditions are contained in the *Underworld/VariableConditions/velocityBCs.extension.xml* XML file.

- **Results** - The following additional rheological parameters were used in this sample:
  - *viscosity* - background and incompressible layer 0.1, crust 10.0, mantle (including weak zone) 1.0 (dimensionless)
  - $\frac{1}{lambda} = 10.0$
  - *Crust*:
    - softening strain = 0.1

- initial damage fraction = 0.0
- initial damage wavenumber = 0.5
- initial damage factor = 0.5
- healing rate = 0.0
- cohesion = 10.0
- cohesion after softening = 0.0001
- friction coefficient = 0.0
- friction coefficient after softening = 0.0
- minimum yield stress = 0.00001

See *Faulting Moresi-Mühlhaus 2006 Benchmark at 1st Timestep (2nd image)* (page 39), *Faulting Moresi-Mühlhaus 2006 Benchmark at 13th Timestep* (page 39) for output images produced using Underworld-1.0.0.

- **Yielding** - Basically, a Mohr-Coulomb criterion is used on each Lagrangian points (particles) to determine if a failure plane is (re)activated at this particular point. If yes, the preferred plane of sliding (assumed to be the one that is more closely aligned with the local sense of shear strain rate) defines a direction of anisotropy. The material properties can then be softened to model the localization of deformation. Once a plane of failure is activated, it is assumed to be a preferred orientation for subsequent sliding, but in case it is no longer active, new orientations are tested. As one can see, this formulation does not simply model a Mohr-Coulomb failure criterion in the strict sense. It is actually a way of incorporating brittle (or semi-brittle) material behavior in a viscous code. If anyone wants to use it, it is important to bear in mind the underlying assumptions of this model.

  In a material which has the lots of randomly directed faults, and therefore the potential to fail in any direction, the faults on which slip occur for the minimum stress orientated at an angle $\pm\theta$ to the most compressive principle stress direction.

  This angle is found using the equation:

$$\tan(2\theta) = \frac{1}{\mu} \tag{5.11}$$

  Where μ is the frictional coefficient.

  The principle stress directions are found using the functions:

```
void UnderworldEigenvalues( double* vector, double* eigenvalues );
void UnderworldEigenvectors( double* vector,
                             double* eigenvalue,
                             double** eigenvector );
```

  Where the most compressive eigenvector is given by the greatest eigenvector, which is given as the first vector in the array.

  Both directions $\pm\theta$ to this most compressive principle stress direction are equivalent in terms of weakness, but generally, only one of these directions is stable. One direction is aligned parallel to the sense of shear, this direction unstable because it will rotate in a hardening direction. The other orientation is stable because the director will no longer be rotating if it is perpendicular to the shear. This then becomes part of the failure criteria, that the material may only fail in the softening direction.

5.8 Journalling

Journalling can provide different levels of information about how the model is progressing. In the case of a desktop machine, this information will display in the terminal window on the screen as the model runs. When running a model on a computer cluster, journalling information will not be displayed on the screen but will be dumped to a file *queuename.o* where *queuename* is the up-to-ten-letter name assigned to the job in the pbs script.

Each category consists of an *info*, *debug* and *error* journal. Journals are controlled from within an input file as follows:

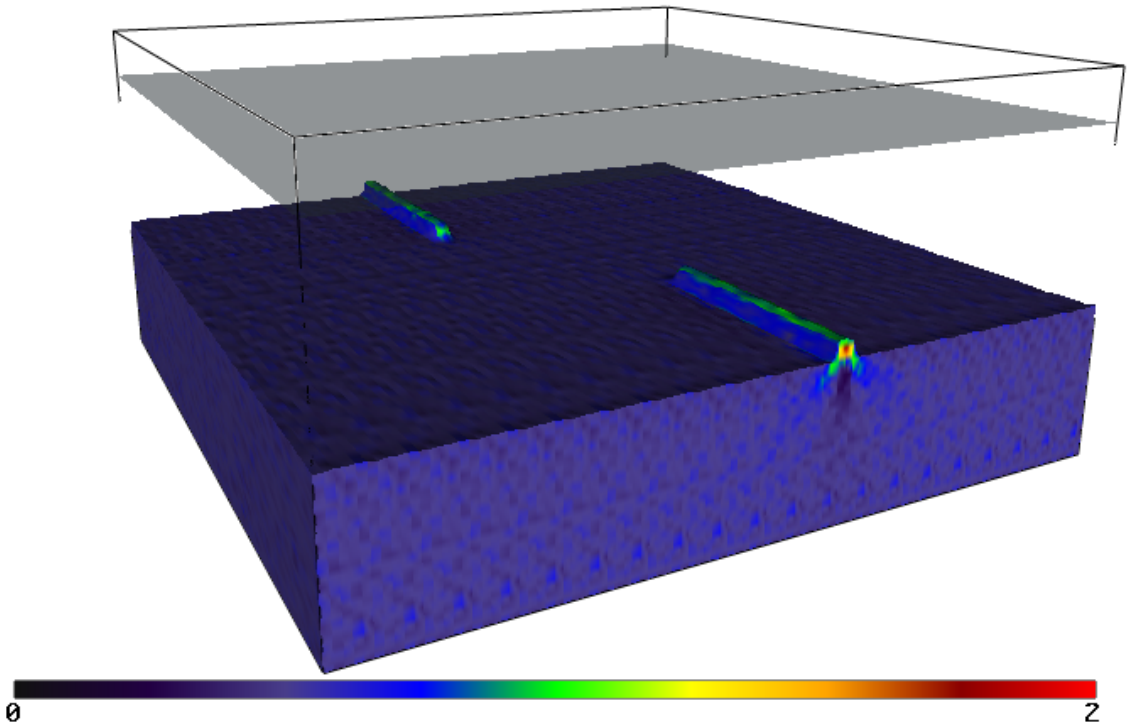


Figure 5.13: Faulting Moresi-Mühlhaus 2006 Benchmark at 1st Timestep (2nd image)

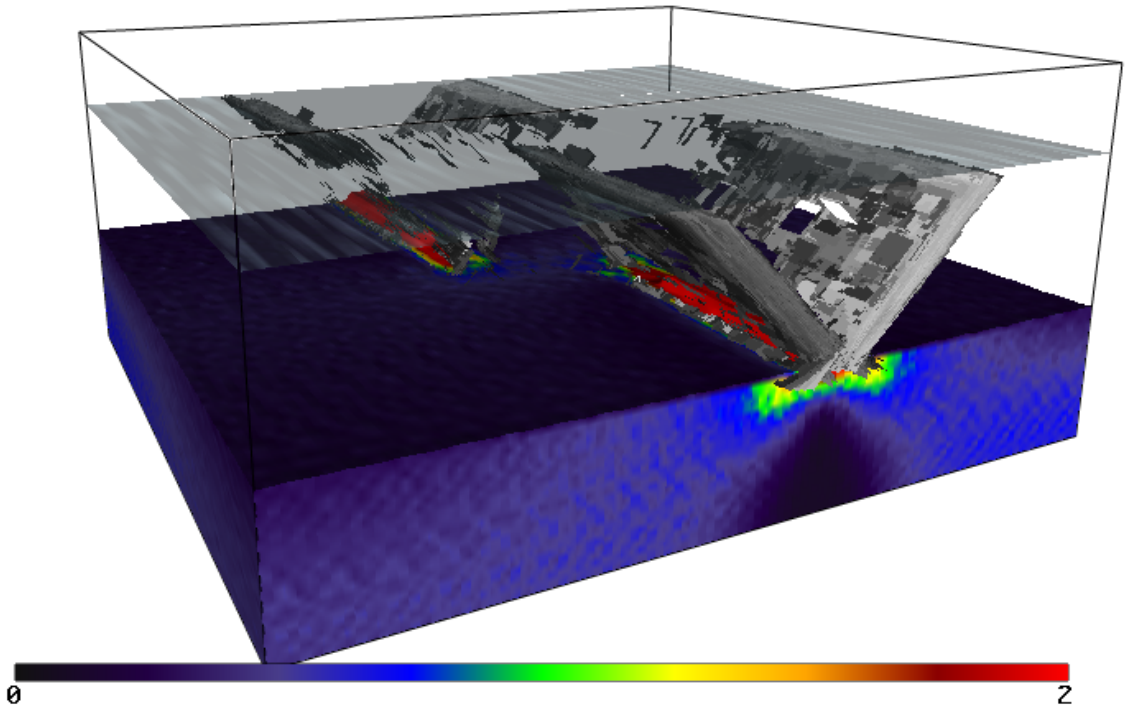


Figure 5.14: Faulting Moresi-Mühlhaus 2006 Benchmark at 13th Timestep

- The *info* journal is enabled (t or true) or disabled (f or false)
- the level of information required for each category is specified with an integer, where 1 is the least amount of information. Available levels are as follows:
  - *1* - General (one line).
  - *2* - Medium (per timestep).
  - *3* - Verbose (lots of information, per inner loop per time step).
  - *< 3* - Even more information ...

5.8.1 Journal Status

Journal and stream status at the start of a simulation on a desktop machine can be viewed to know which ones are enabled using the *–showJournalStatus* command line argument.

For example:

```
--showJournalStatus=True
```

This will give a print out of all available streams and their activation status.

5.8.2 Outputting

On a Desktop

By default journalling information will be displayed in the terminal window as the model runs. This information can be instead redirected to an output file by simply redirecting the output to a file.

For example:

```
./Underworld MyInputFiles/Sample.xml > filename.txt
```

This will redirect the journal output into *filename.txt* file.

On a Cluster

When running a model on a computer cluster, journalling information will not be displayed on the screen but will be instead dumped to a file *queuename.o*. For this reason, when running a long-running parallel job on a cluster, it is strongly advised to reduce the level of journal output to avoid filling up the stdout buffer.

5.8.3 Journal XML Controls

The following list outlines the various journal controls that can be toggled by the user.

Enabling/Disabling

- Enable or disable the entire journal.

```
<param name="journal">on/off</param>
```

- Enable or disable a typed stream.

```
<param name="journal.TypedStream">on/off</param>
```

- Enable or disable a named stream or its sub-streams.

```
<param name="journal.TypedStream.NamedStream...">on/off</param>
```

- Enable or disable a named stream, and all its children.

```
<param name="journal-enable-branch.TypedStream.NamedStream...">on/off</param>
```

Level of Output

- Set the level of output for a typed stream.

```
<param name="journal-level.TypedStream">integer</param>
```

- Set the level of output for a named stream.

```
<param name="journal-level.TypedStream.NamedStream...">integer</param>
```

- Set the level of output for a typed stream, and all children of that type.

```
<param name="journal-level-branch.TypedStream">integer</param>
```

- Set the level of output for a named stream, and all its children.

```
<param name="journal-level-branch.TypedStream.NamedStream...">integer</param>
```

Printing Rank

- Set the rank number which will be printing from this typed stream.

```
<param name="journal-rank.TypedStream">integer</param>
```

- Set the rank number which will be printing from this named stream.

```
<param name="journal-rank.TypedStream.NamedStream...">integer</param>
```

Dump Every

- How often a typed stream dumps output.

```
<param name="journal-dumpevery.TypedStream">integer</param>
```

- How often a named stream or sub-stream dumps output.

```
<param name="journal-dumpevery.TypedStream.NamedStream...">integer</param>
```

Auto Flush

- Set whether this typed stream will flush after every output call.

```
<param name="journal-autoflush.TypedStream">on/off</param>
```

- Set whether this named stream will flush after every output call.

```
<param name="journal-autoflush.TypedStream.NamedStream...">on/off</param>
```

Output Redirection

- Redirect a typed stream to a file output.

```
<param name="journal-file.TypedStream">filename</param>
```

- Redirect a named stream to a file output.

```
<param name="journal-file.TypedStream.NamedStream...">filename</param>
```

MPI File Offsets

- Set the number of bytes offset for the MPI file associated with the stream.

```
<param name="journal-mpi-offset.TypedStream">filename</param>
```

- Set the number of bytes offset for the MPI file associated with the stream.

```
<param name="journal-mpi-offset.TypedStream.NamedStream...">filename</param>
```

Examples

- **debug**
  - Enables (true) or disables (false) the *debug* stream.

```
<param name="journal.debug">true</param>
```

  - Enables (true) or disables (false) the *Plugin* stream of *debug* or its sub-streams.

```
<param name="journal.debug.Plugin">true</param>
```

- **Context-verbose**
  - Enables (true) or disables (false) the *Context-verbose* stream or its sub-streams.

```
<param name="journal.info.Context-verbose">true</param>
```

- **Components**
  - Sets the level (an integer, e.g. 2) of output for the *Stg_ComponentFactory* stream.

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

- **StgFEM**
  - Enables (true) or disables (false) the *StgFEM* stream and all of its children.

```
<param name="journal-enable-branch.debug.StgFEM">true</param>
```

  - Sets the level (an integer, e.g. 2) of output for the *StgFEM* stream and all of its children.

```
<param name="journal-level-branch.debug.StgFEM">2</param>
```

- **Swarm**
  - Enables (true) or disables (false) the *Swarm* stream and all of its children.

```
<param name="journal-enable-branch.debug.Swarm">true</param>
```

  - Sets the level (an integer, e.g. 2) of output for the *Swarm* stream and all of its children.
- **gLucifer**
  - Enables (true) or disables (false) the *lucDebug* stream and all of its children.

```
<param name="journal-enable-branch.debug.lucDebug">true</param>
```

  - Sets the level (an integer, e.g. 2) of output for the *lucDebug* stream and all of its children.

```
<param name="journal-level-branch.debug.lucDebug">2</param>
```

– Sets the level (an integer, e.g. 2) of output for the *lucDebug* stream.

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

– Sets the level (an integer, e.g. 2) of output for the *lucInfo* stream.

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

5.9 Checkpointing

Underworld provides the facility to *check point* simulation runs periodically, which saves key information about the application to disk. This is useful for analysing results in detail after a simulation has completed, or restarting the simulation. It allows key parameters to be changed when simulations are restarted.

5.9.1 Scope

When checkpointing is enabled for StgFEM, it will dump the values of certain *FeVariables* in the current simulation into separate files in the output directory. The output directory can be specified by setting the *outputPath* in the XML file (see *Creating your own XML file* (page 21)). By default, the system will save only the *FeVariables* required for a restart of the model. Additional fields can be included for checkpointing by declaring them in the *FieldVariablesToCheckpoint* list of the XML input file.

For example:

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

Note: Specifying an own *FieldVariablesToCheckpoint* list may result in insufficient data being stored for checkpoint restarts. If storing auxiliary fields for general analysis is required, it is best to use the *FieldVariablesToSave* list.

At times it is also desirable to specify the data to store to file. This is distinct from checkpointed data in that it will not be reloaded when a job is resumed. Currently only field data and not swarm data may be stored. The fields which the user wishes to store are defined using the *FieldVariablesToSave* list.

For example:

```
<list name="FieldVariablesToSave">
   <param>VelocityGradientsField</param>
   <param>StrainRateField</param>
</list>
```

A list of available fields to save may be generated using the *StgFEM_FeVariableList* plugin.

5.9.2 Data Options

There are two modes for checkpointing in Underworld:

- **HDF5** - This mode relies upon the HDF5 libraries for data storage. If during the compile configuration stage the HDF5 libraries are found, this is used as the default mode.

  The HDF5 mode is the preferred mode for checkpointing. Files stored using the HDF5 mode receive the *.h5* extension. In this mode, each field is stored in its own file, and only one file is create irrespective of how many processors the jobs utilises. Swarm data is treated differently however, with each processor dumping

a file for each swarm in the simulation. An eight processor simulation would for example result in eight output files for each swarm.

Restarting using arbitrary number of processors is still allowed and will occur automatically. Users don't need to specify how many files were used for the original checkpoint. One key benefit of HDF5 mode is that checkpoints are portable, and may be restarted on any machine irrespective of the architecture. File sizes are also smaller for swarm data, as only the required data is stored. The *H5utils* package may prove useful to the user wishing to query and manipulate HDF5 files.

- **ASCII** - This mode stores all field data in ASCII format, while swarm data is stored in the native binary format of the machine. ASCII mode does not require any external package.

  The ASCII checkpoint mode stores each checkpointed field/swarm in its own file with the *.dat* extension. For parallel simulations this is still the case, with only one file per field/swarm created regardless of how many processors are used. Restarts are possible using any number of processors, irrespective of how many processors the simulation was originally run across.

  Generally, checkpoints created using ASCII mode are not portable due to the binary mode in which swarms are stored. The exceptions to this are where no swarm checkpointing is required, in which case data is only stored in ASCII format which may be re-read on other machines. The other exception is where machines have identical architectures in which case the binary files should be re-read without issue. Note also that some cluster machines do not have homogeneous architectures, and so binary checkpoint files may fail to be read correctly. Generally this will result in terminal failure of the simulation.

If required, HDF5 can be disabled when configuring and compiling Underworld (see *Installing Underworld* (page 8)).

5.9.3 Mesh and Node Locations

Locations of the nodes are stored in a separate mesh checkpoint file. For HDF5 checkpoints, the mesh connectivity is also contained within this file. The nodes' locations can be stored within the field checkpoint files by setting the *saveCoordsWithFields* flag in the XML file.

For example:

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

5.9.4 Format

The format of the nodal field dump files is:

```
value1 [value2] [value3]
```

If *saveCoordsWithFields* is set to *True*, the format is:

```
xCoord yCoord zCoord value1 [value2] [value3]
```

For example:

```
0.5 0.5 0.6 32.0 12.1 0.788
```

This is a 3D velocity dump with *saveCoordsWithFields* set to *True* where the data is at coord (0.5, 0.5, 0.6) and the value is (32, 12.1, 0.788). The line number within the dataset of the checkpoint file will correspond to the global node index.

5.9.5 Enabling Checkpointing

Checkpointing is disabled by default. To activate it, the parameter *checkpointEvery* in the XML input file or command line must be set.

For example:

```
<param name="checkpointEvery">5</param>
```

This will checkpoint every 5 timesteps. Checkpointing can be disabled by setting this parameter to `0` (zero).

5.9.6 Restarting a Simulation

To restart a simulation from a saved set of checkpoint information, the simulation must be re-launched with the compatible input files with an extra input parameter *restartTimestep* which must be specified to refer to the timestem to load. The simulation will start running from the next timestep.

For example:

```
./Underworld myJob.xml --restartTimestep=8
```

Model *myJob.xml* will be restarted from timestep 8. The *restartTimestep* can also be set in the XML input file.

For example:

```
<param name="restartTimestep">8</param>
```

The model can then be ran without the *restartTimestep* command line argument.

For example:

```
./Underworld myJob.xml
```

5.9.7 Restarting at a Different Resolution

Often it is useful to restart jobs at a resolution different to that originally used. This is possible by using the *interpolateRestart flag*. An example run of a standard Rayleigh-Taylor simulation at resolution of 64 elements in each direction and checkpointed at timestep 10 can be restarted to a higher resolution of 128 elements in each direction.

For example:

```
Underworld RayleighTaylorBenchmark.xml --restartTimestep=10 --interpolateRestart=1
```

Field data is interpolated to the new resolution using the element interpolation functions, and so interpolation order will correspond to the element order of the field being interpolated (a field constructed of linear basis functions will for example result in linear interpolation). The original particle data is used, with new particles being copied from those existing where required. For best results, it is recommended that steps up or down in resolution are kept modest. Only regular meshes are currently supported.

5.9.8 Changeable Attributes in Restart

The following attributes can be changed when restarting from checkpoint files:

- Visualisation paramters used in gLucifer visualisation can be completely changed since these are all generated on the fly and don't affect the core components of the model. This could be potentially used to change the camera angle to highlight different features of the model as they evolve.
- Global parameters such as diffusivity, solver tolerances, etc.
- Values applied at boundary conditions.
- Properties of the previously defined materials like the density of a material or rheologies applied to it. Material particles assignment can't be changed after the restart.
- Model resolution can be interpolated to a new resolution on restart.

5.9.9 Commonly Checkpointed Fields

The following list outlines the commonly checkpointed fields:

- *PressureField*
- *StrainRateField*
- *StrainRateInvariantField*
- *TemperatureField*
- *VelocityField*
- *VelocityGradientsField*
- *VelocityMagnitudeField*
- *VorticityField*

5.9.10 Checkpointing Flags

The following list outlines the available checkpointing flags

- **checkpointAppendStep** - *Boolean*. Set to *True* to store all checkpoint files in a per timestep directory.
- **checkpointAtTimeInc** - *Float*. Sets a time increment at which checkpointing should occur.
- **checkpointEvery** - *Unsigned*. Timestep interval at which checkpointing should occur.
- **checkPointPrefixString** - *String*. A string prefix which is appended to checkpoint files.
- **checkpointReadPath** - *String*. Path where checkpointing files may be found.
- **checkpointWritePath** - *String*. Path where checkpointing files may be stored.
- **interpolateRestart** - *Boolean*. Set to *True* to interpolate checkpointed data to the new resolution.
- **restartTimestep** - *Unsigned*. Timestep to resume from checkpointed data from.
- **saveCoordsWithFields** - *Boolean*. Set to *True* to include node locations with field checkpoint files.
- **saveDataEvery** - *Unsigned*. Timestep interval at which data saving should occur (not necessarily for checkpointing).

5.10 Timestepping

Depending on the equations used in a simulation, Underworld uses different algorithms to calculate the timestep, ΔT, required to maintain numerical stability. When modelling time independent Stokes Flow with particle advection, the stability criterion is,

$$\Delta T < 0.5 \cdot \frac{\Delta x}{|\mathbf{u}|} \tag{5.12}$$

Where Δx is the minimum width of an element and $|\mathbf{u}|$ is the maximum velocity magnitude measured in the entire domain. This method ensures a stable timestep is always selected.

For models with both Stokes Flow and Energy (advection diffusion) equations an added diffusion stability criterion is calculated,

$$\Delta T < C \cdot \frac{(\Delta x)^2}{K} \tag{5.13}$$

Where C is the courant factor and K is the maximum value of thermal diffusivity. The minimum ΔT from the above criteria is used as the timestep to update Underworld. To see the timestep values chosen by Underworld look at the *FrequentOutput.dat* file in the output directory.

5.10.1 Controls

The following list outlines available timestepping controls:

- **courantFactor** - *Default is 0.5*. The courant factor, C.
- **timestepFactor** - *Default is 1.0*. Scales the minimum timestep returned by the stability criteria.

  For example, *–timestepFactor=0.1* would use a timestep one tenth the original timestep calculated from the stability criteria.
- **limitTimeStepIncreaseRate** - *Default is False*. Determines whether the increase in ΔT should be limited.
- **maxTimeStepIncreasePercentage** - *Default is 10.0*. Sets a limit on the percentage increase in ΔT per timestep. This option requires *limitTimeStepIncreaseRate* to be *True*.
- **DtGranularity** - *Default is 0.0*. Sets the timestep to the nearest granularity of the problem, so long as *DtGranularity* is less than the calculated timestep from the criteria.
- **maxTimeStepSize** - *Default is 0.0*. Sets limit on the maximum time step possible. Timesteps greater than the limit are clipped to the limit.

All the options mentioned above can be set from the XML file or passed as command line arguments.

For example:

```
<param name="courantFactor">0.5</param>
```

See *AbstractContext Time Sequence.* (page 48) for the timestepping and checkpointing states that is configured in Underworld.

5.11 Units Scaling

As of version 1.5, Underworld comes with the capability to scale input and output parameters in *SI* units automatically. This allows users to input parameters using a variety of *SI* values, as well as output FEM field values in desired values.

5.11.1 Enabling Scaling

To activate this functionality, the setup of a *Scaling* component is required, then all relevant input and output units can be defined.

For example:

```
<struct name="default_scaling">
   <param name="Type">Scaling</param>
   <param name="spaceCoefficient_meters">1.0e6</param>
   <param name="timeCoefficient_seconds">3.30e-5</param>
   <param name="massCoefficient_kilograms">3.30e21</param>
   <param name="temperatureCoefficient_kelvin">2044</param>
</struct>
```

Here the fundamental scaling coefficients are defined (electric current is left out of this example). These are used to convert numbers between dimensional or scaled values based on a user specified SI units string.

If the following is defined in the XML file:

```
<param name="referenceTemperature" units="K">1022</param>
```

Where *K* means kelvin scaling, Underworld will store the referenceTemperature as $1022/2044 = 0.5$

And,

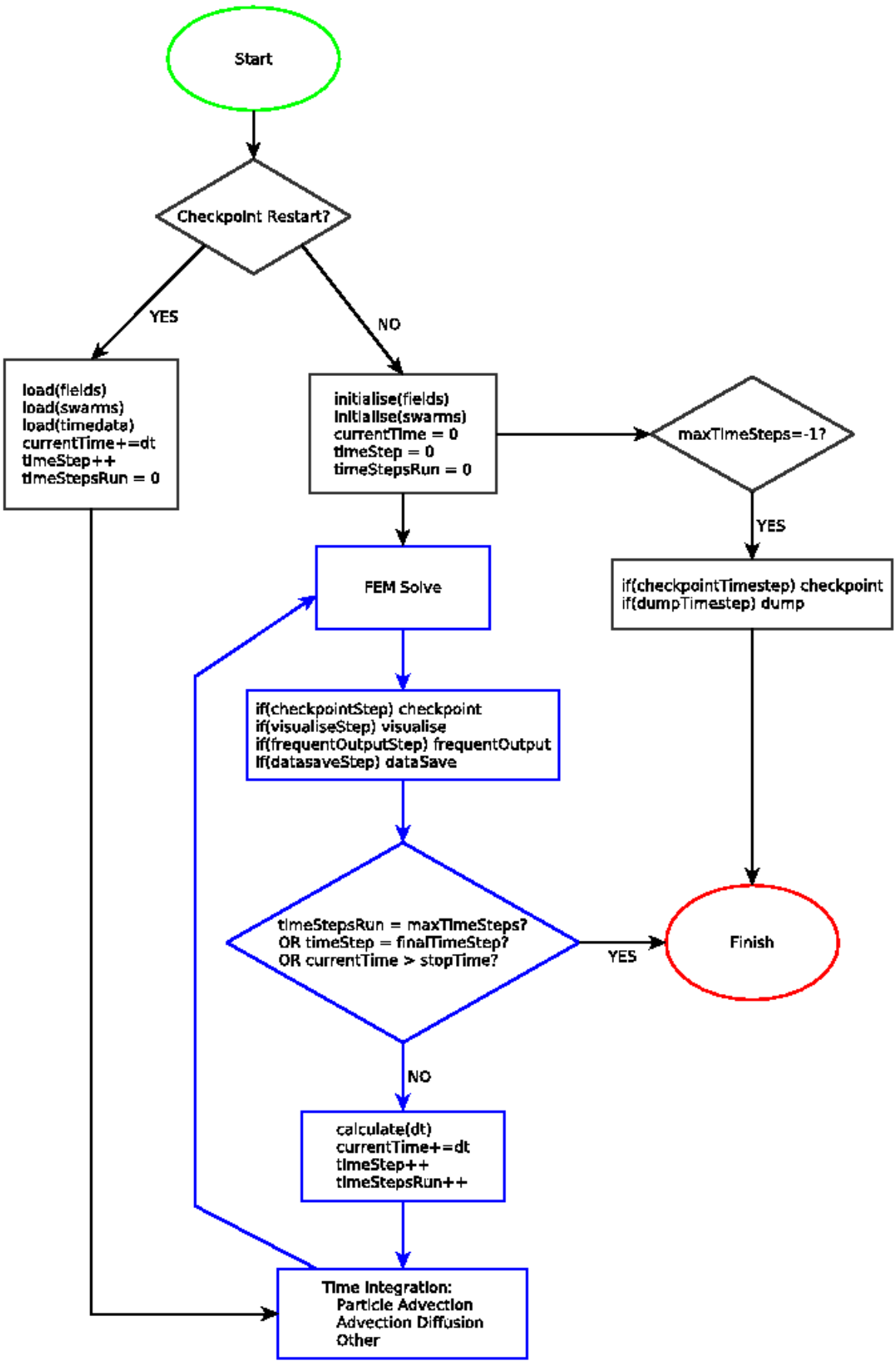


Figure 5.15: *AbstractContext* Time Sequence.

```
<param name="myViscosity" units="Pa*s">1e20</param>
```

Where *Pa*s* means pascal second scaling, Underworld will store the myViscosity as $1 * 10^{20}/(3.30 * 10^{21}/(3.30 * 10^{-5} * 1.0 * 10^{6})) = 1$.

5.11.2 Scaling gLucifer Output

A FeVariable component can be given an *outputUnits* string parameter to automatically scale any gLucifer output from the field.

For example:

```
<struct name="VelocityField" mergeType="merge">
   <param name="outputUnits">cm/yr</param>
</struct>
```

All velocity field output will be given a cm per year scaling.

5.11.3 Examples

Example of an XML file using scaling is found in:

`$PATHTOUNDERWORLD/Underworld/InputFiles/Woidt78_model_a.xml`.

5.11.4 Important Notes

Input scaling is only applied to parameters with explicit *units* attributes. Default values which the user may be using are not appropriately scaled and will most likely cause incorrect behaviour due to improper scaling. If the scaling mechanism is used, one must avoid using default values like *defaultDiffusivity*.

The *units* string doesn't perform any error checking yet on the type of number it is defining.

For example:

```
<param name="referenceTermperature" units="kg">1022</param>
```

is legal input, but the scaling will be incorrect and the model will misbehave. The *units* string needs to be checked with the number it is defining.

Only gLucifer output can be dimensionalised with *outputUnits*. All other data, such as *FrequentOutput.dat* and checkpoint data, will remain scaled.

5.12 Non-Linear Convergence Plot

The *Convergence tool* allows the user to see if their code is converging for a certain tolerance on the Uzawa solver. The default tolerance is non-linear. This works by the residual for each non-linear iteration being recorded in the file *convergence.dat*. The tool can be enabled in the XML input file.

For example:

```
<struct name="stokesEqn" mergeType="merge">
   <param name="makeConvergenceFile" mergeType="replace">true</param>
</struct>
```

This creates the *Convergence.dat* file when the model is ran. The file will have the following format:

```
1 2 0.002 0.001
1 3 0.0015 0.001
```

Where the first column represents the *timestep*, the second column the *number of non-linear iterations at that time step*, the third column the *residual of non-linear iterations* and the fourth column the *tolerance condition for non-linear solver*.

The maximum amount of non-linear iterations can also be set in the XML input file as a parameter.

For example:

```
<param name="nonLinearMaxIterations">5</param>
```

The default value is quite low, and could be too low to get a convergence. The output file, *Convergence.dat* helps the user decide if this number needs to be higher to get a convergence.

5.13 Recovery Methods

Underworld currently supports two recovery methods, *REP* and *SPR*. These methods recover stress and/or strain rate fields so they become continuous across the computational domain. Generally with the default element types in Underworld - linear velocity and constant pressure elements - the numeric fields for the *raw* stress and strain rate are discontinuous across element boundaries.

Recovery methods smooth these *raw* fields to create continuous *recovered* fields that exhibit better convergence behaviour than the *raw* fields, especially if the model flow is continuous. Models with thin, discontinuous flow can suffer from the effective smoothing of recovery methods.

All recovery methods in Underworld produce FeVariable fields as output that store the recovered values. These fields can be used in any other functionality that takes an *FeVariable* as input like *checkpointing* and *visualisation*.

5.13.1 Recovery by Equilibrium in Patches (REP)

Used for recovering total strain rate, total and deviatoric stress including pressure. This algorithm is enabled by including the *REP_Setup_Compressible.xml* in the XML input file.

For example:

```
<include>Underworld/REP_Setup_Compressible.xml</include>
```

This will create the following recovered fields:

- *recoveredStrainRateField*
- *recoveredStressField*
- *recoveredTauField* - recovered deviatoric stress field.
- *recoveredPressureField*
- *recoveredStrainRateInvariantField*
- *recoveredStressInvariantField*
- *recoveredTauInvariantField* - invariant of recovered deviatoric stress field.

5.13.2 Superconvergent Patch Recovery (SPR)

Used for recovering total strain rate. This algorithm is enabled by including the *SPR_Setup.xml* file in the XML input file.

For example:

```
<include>Underworld/SPR_Setup.xml</include>
```

This will create a recovered strain rate field, called *recoveredStrainRateField*.

5.14 HDF5 Initial Condition

It is possible to define the initial condition for an FeVariable using checkpointed data. Only HDF5 checkpoint data is supported. This functionality is enabled by using the *HDF5ConditionFunction* plugin.

For example:

```
<list name="plugins" mergeType="merge">
   <struct>
      <param name="Type">Underworld_HDF5ConditionFunction</param>
      <param name="FeVariableHDF5Filename">VelocityField.input.h5</param>
      <param name="MeshHDF5Filename">Mesh.linearMesh.input.h5</param>
      <param name="TargetFeVariable">VelocityField</param>
   </struct>
</list>
<struct name="velocityICs">
   <param name="type">CompositeVC</param>
   <list name="vcList">
      <struct>
         <param name="type"> AllNodesVC </param>
         <list name="variables">
            <struct>
               <param name="name">velocity</param>
               <param name="type">func</param>
               <param name="value">HDF5ConditionFunction</param>
            </struct>
         </list>
      </struct>
   </list>
</struct>
```

This first section creates an instance of the *HDF5ConditionFunction* plugin. The parameters *FeVariableHDF5Filename* and *MeshHDF5Filename* respectively define the filename (and path) for the checkpointed FeVariable and FeMesh, both of which are required. It is not necessary for the checkpointed resolution to be identical to the current simulation resolution. The file paths are specified with respect to the simulation execution directory. It will be generally safest to specify absolute paths to the data files.

The final parameter *TargetFeVariable* is required to determine auxiliary information for the checkpoint data reload process. It tells the simulation which *FeVariable* should have it's initial condition set, as this is handled in (for example) the *velocityICs* XML struct.

The final section configures the initial condition for the *FeVariable*. It sets the standard velocity *FeVariable* initial condition. For Stokes flow, this will have no effect as the solution is entirely determined by the boundary conditions and body forces. This definition follows the standard procedure for variable conditions.

AllNodesVC is used to ensure all nodes have their value set. The struct within the *variables* list instructs that the initial condition well be calculated using the *HDF5ConditionFunction* plugin. The *variables * struct contains the following parameters:

- *name* - Determines which variable to set.
- *type* - Configures the variable to be determined via *func*.
- *value* - Sets the function to *HDF5ConditionFunction*.

A complete example can be found in *Underworld/plugins/VariableConditions/HDF5ConditionFunction* inside the *exampleHDF5Condition.xml* file.

5.15 Viscosity Field

By default the viscosity of the fluid is not produced as output of an Underworld run. The field must be calculated first before outputting it.

5.15.1 Calculate

To define the creation and calculation of the viscosity field as a type *FeVariable*, the user must includes the *ViscosityField* template XML file.

For example:

```
<include>Underworld/ViscosityField.xml</include>
```

This defines a component, called *ViscosityField* that gets (or if needed calculates) the assumed isotropic viscosity on a particle. The value of this point-based quantity is turned into a field by a smoothing algorithm and, as such, this field is recommended for output only and not as a parameter to feedback into the calculations.

An important parameter of the *ViscosityField* is the swarm where the viscosity is calculated and subsequently integrated for the smoothing algorithm. In models with PIC particles this should be *picIntegrationPoints* swarm. This is the default setting of the *ViscosityField*. But in models with only FEM meshes and not PIC, the swarm should be *gaussSwarm*.

For example:

```
<struct name="components" mergeType="merge">
   <struct name="ViscosityField" mergeType="merge">
      <param name="Swarm">gaussSwarm</param>
   </struct>
</struct>
```

5.15.2 Output

As with any other field in Underworld, output can be to disk via checkpointing (see *Checkpointing* (page 43)) as well as visualised by gLucifer (see *Visualisation* (page 63)). To visualise the *ViscosityField*, a template XML is provided that setups up a subsequent viewport, called *ViscosityVP*. This template must be included in the XML input file.

For example:

```
<include>Underworld/Viewports/ViscosityVP.xml</include>
```

With this viewport defined, the *ViscosityVP* needs only to be passed to a window for rendering.

5.16 Multigrid

Mutigrid can be used in the following scenarios:

- 3D or large 2D problem that can't be solved by a direct solver on a single node.
- There are over 1,000,000 unknowns.
- If there are large variations in viscosities.

See *Timing example using different solvers for an extension model with an imposed weak zone (no yielding).* (page 53) for an example performance comparison when using multigrid.

5.16.1 Amount of Levels

The amount of levels of multigrid can be adjusted using the *mgLevels* parameter. This parameter is dependent on the configuration and assumptions in the model. Geometric multigrid is based on the mesh used to solve a problem. It needs to coarsen the mesh evenly in each dimension a number of times equal to the number of levels requested minus 1.

For example, if solving on a 64 x 64 mesh and the level is set to 3, the sequence of grids will be 64 x 64, 32 x 32, 16 x 16.

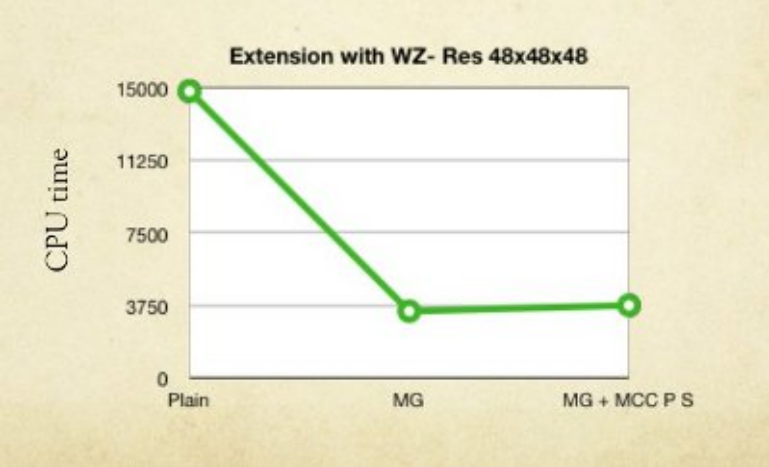


Figure 5.16: Timing example using different solvers for an extension model with an imposed weak zone (no yielding).

The finest level is considered a part of the three requested levels. So, if for the same example the level is set to 8, the sequence of grids will be 64 x 64, 32 x 32, 16 x 16, 8 x 8, 4 x 4, 2 x 2, 1 x 1, *–error*.

The mesh can't suppor the number of levels. The number of levels must be selectes in accordance with the size of the mesh. This can be expressed by

$$\frac{M}{2^{N-1}} = k$$

Where M is the mesh size, N is the number of multigrid levels, and $k \in \mathbb{N}$.

5.17 Passive Tracers

Passive tracers are particles that can be added to a model which move passively within the domain without influencing the model. They can be used to record x, y and z positions and also field values from user-defined points or regions in a model at each timestep.

The key difference between this form of output as compared to outputting fields for the whole domain, is the choice of the user-defined *passiveTracerSwarm*, as opposed to the whole *materialSwarm*. If field value output are required see *Images and Movies* (page 67).

5.17.1 Requirements

To make use of passive tracers, the following aspects must be covered:

- **ParticleMovementHandler** - This is an essential component to be added to the XML input file.

  For example:

```
<struct name="passiveSwarmMovementHandler">
   <param name="Type"> ParticleMovementHandler </param>
</struct>
```

- **Particle Layout** - Defines the layout of the particles. This first layout to be defined is the *ElementCellLayout*.

  For example:

```
<struct name="ElementCellLayoutTracer">
   <param name="Type">ElementCellLayout</param>
```

```
    <param name="Mesh">linearMesh</param>
</struct>
```

Then one can define the required particle layout. Available layouts are as follows:

- *FileParticleLayout* - Loads a global coordinate layout from a file, in either ASCII or HDF5 format. Used to define a particle swarm layout. It is also used in checkpointing data.
- *LineParticleLayout* - Lays out equally spaced particles on a continuous set of line segment.

  For example:

```
<struct name="passiveTracerLayoutExample1">
   <param name="Type">LineParticleLayout</param>
   <param name="totalInitialParticles">6</param>
   <list name="vertices">
      <asciidata>
         <columnDefinition name = "x" type="double"/>
         <columnDefinition name = "y" type="double"/>
         <columnDefinition name = "z" type="double"/>
            0.5 0.1 0.0
            0.1 1.0 0.0
            0.3 1.0 2.0
      </asciidata>
   </list>
</struct>
```

- *ManualParticleLayout* - Lays out a particle in each of the positions manually specified.

  For example:

```
<struct name="passiveTracerLayoutExample2">
   <param name="Type">ManualParticleLayout</param>
   <list name="manualParticlePositions">
      <asciidata>
         <columnDefinition name = "x" type="double"/>
         <columnDefinition name = "y" type="double"/>
         <columnDefinition name = "z" type="double"/>
            0.5 0.1 0.0
            0.1 1.0 0.0
            0.3 1.0 2.0
      </asciidata>
   </list>
</struct>
```

  This creates three particles: (0.5, 0.1, 0.0)(0.1, 1.0, 0.0)(0.3, 1.0, 2.0).

- *SpaceFillerParticleLayout* - Uses the SobolGenerator class to quasi-randomly fill particles throughout the whole global domain, by setting either *totalInitialParticles* or *averageInitialParticles*.

  For example:

```
<struct name="passiveTracerLayoutExample3">
   <param name="Type">SpaceFillerParticleLayout</param>
   <param name="totalInitialParticles">6000</param>
</struct>

<struct name="passiveTracerLayoutExample4">
   <param name="Type">SpaceFillerParticleLayout</param>
   <param name="averageInitialParticlesPerCell">20</param>
</struct>
```

- *PlaneParticleLayout* - Uses the *SpaceFillerParticleLayout* class to choose quasi-random particle positions and then projects them onto a plane.

  For example:

```
<struct name="passiveTracerLayoutExample5">
   <param name="Type">PlaneParticleLayout</param>
   <param name="totalInitialParticles">6000</param>
   <param name="planeAxis">y</param>
   <param name="planeCoord">1.0</param>
</struct>
```

- *WithinShapeParticleLayout* - Uses the *SpaceFillerParticleLayout* class to choose quasi-random particle positions, but it only places a particle there if it is within a particular shape.

  For example:

```
<struct name="passiveTracerLayoutExample6">
   <param name="Type">WithinShapeParticleLayout</param>
   <param name="totalInitialParticles">6000</param>
   <param name="shape">boxShape</param>
</struct>
```

- *UnionParticleLayout* - This combines several particle layouts together into one. when one particle layout finishes initialising all its particles, then the next one begins.

  For example:

```
<struct name="passiveTracerLayoutExample7">
   <param name="Type">UnionParticleLayout</param>
   <list name="ParticleLayoutList">
      <param>passiveTracerLayoutExample1</param>
      <param>passiveTracerLayoutExample2</param>
      <param>passiveTracerLayoutExample3</param>
   </list>
</struct>
```

- **Passive Tracer Swarms** - The *SwarmAdvector* component advects the particles of the *passiveTracerSwarm*, so that they move with the model. A swarm using a tracer layout neds to be created.

  For example:

```
<struct name="passiveSwarmMovementHandler">
   <param name="Type">ParticleMovementHandler</param>
</struct>
<struct name="ElementCellLayoutTracer">
   <param name="Type">ElementCellLayout</param>
   <param name="Mesh">linearMesh</param>
</struct>
<struct name="passiveTracerSwarm">
   <param name="Type">MaterialPointsSwarm</param>
   <param name="CellLayout">ElementCellLayoutTracer</param>
   <param name="ParticleLayout">passiveTracerLayoutExample1</param>
   <param name="FiniteElement_Mesh">linearMesh</param>
   <param name="FeMesh">elementMesh</param>
   <list name="ParticleCommHandlers">
      <param>passiveSwarmMovementHandler</param>
   </list>
</struct>
```

  The tracer swarm can be avected using the *SwarmAdvector*.

  For example:

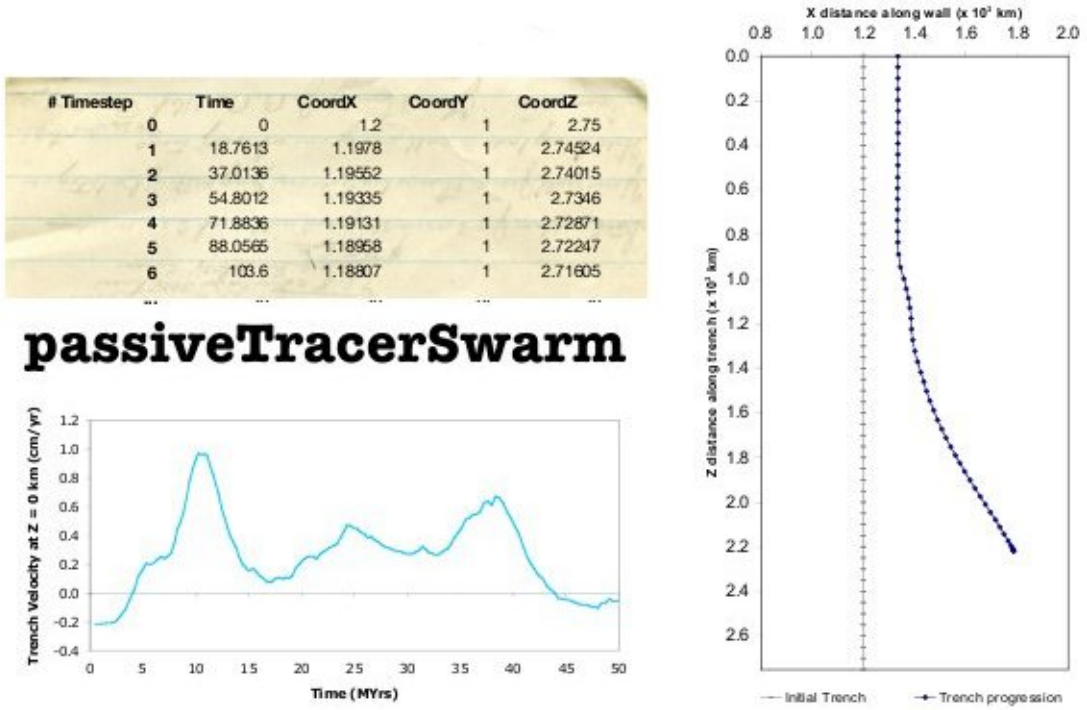
```
<struct name="passiveTracerAdvect">
   <param name="Type">SwarmAdvector</param>
   <param name="Swarm">passiveTracerSwarm</param>
   <param name="TimeIntegrator">timeIntegrator</param>
   <param name="VelocityField">VelocityField</param>
</struct>
```

- **Output Format** - There are three output swarm to choose from: *SwarmOutput*, *PressureTemperatureOutput* and *Underworld_SwarmOutput*.
  - *SwarmOutput* - This is the parent component to *PressureTemperatureOutput*. This component outputs x, y and z positions only, one file per particle in the swarm, appended at each timestep. The filename is set either by default to *<nameOfSwarmOutput>.<timeStep>.dat* or using the *baseFilename* tag.

    For example:

```
<struct name="passiveTracerSwarmOutput">
   <param name="Type">SwarmOutput</param>
   <param name="Swarm">passiveTracerSwarm</param>
   <param name="columnWidth">15</param>
   <param name="decimalLength">10</param>
   <param name="borderString">|</param>
</struct>
```

    Formatting of the output can be controlled via the *columnWidth*, *decimalLength* and *borderString* tags. See *Passive Tracer Swarm Output Data.* (page 56) for example output.



# Timestep	Time	CoordX	CoordY	CoordZ
0	0	1.2	1	2.75
1	18.7613	1.1978	1	2.74524
2	37.0136	1.19552	1	2.74015
3	54.8012	1.19335	1	2.7346
4	71.8836	1.19131	1	2.72871
5	88.0565	1.18958	1	2.72247
6	103.6	1.18807	1	2.71605

Figure 5.17: Passive Tracer Swarm Output Data.

  - *PressureTemperatureOutput* - This component is a child of *SwarmOutput*. It outputs x, y and z positions, as well as pressure and temperature field values on a swarm.

    For example:

```
<struct name="passiveTracerSwarmPressureTemperatureOutput">
   <param name="Type">PressureTemperatureOutput</param>
   <param name="Swarm">passiveTracerSwarm</param>
```

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

- *Underworld SwarmOutput* - This component allows for a given list of *FeVariable(s)* to be calculated on a selected swarm (a passive tracer or the whole materialSwarm). It outputs material id, x, y, z positions and the value of the specified field(s) to a file called *<nameOfField>.<nameOfSwarm>.<timeStep>.dat*, per timestep, per *FeVariable*.

  To output a list of *FeVariables* that are used in the model, the *StgFEM_FeVariableList* plugin must be added in the XML input file.

  For example:

```
    <list name="plugins" mergeType="merge">
        <struct>
            <param name="Type">StgFEM_FeVariableList</param>
            <param name="Context">context</param>
        </struct>
    </list>
```

  The file *FeVariables.list* will be created in the output directory at the zero timestep. This can be enabled by incuding the *Underworld_SwarmOutput* component in the XML file.

  For example:

```
    <struct name="swarmOutput">
        <param name="Type">Underworld_SwarmOutput</param>
        <param name="Swarm">passiveTracerSwarm</param>
        <list name="FeVariables">
            <param>Field1</param>
            <param>Field2</param>
        </list>
    </struct>
```

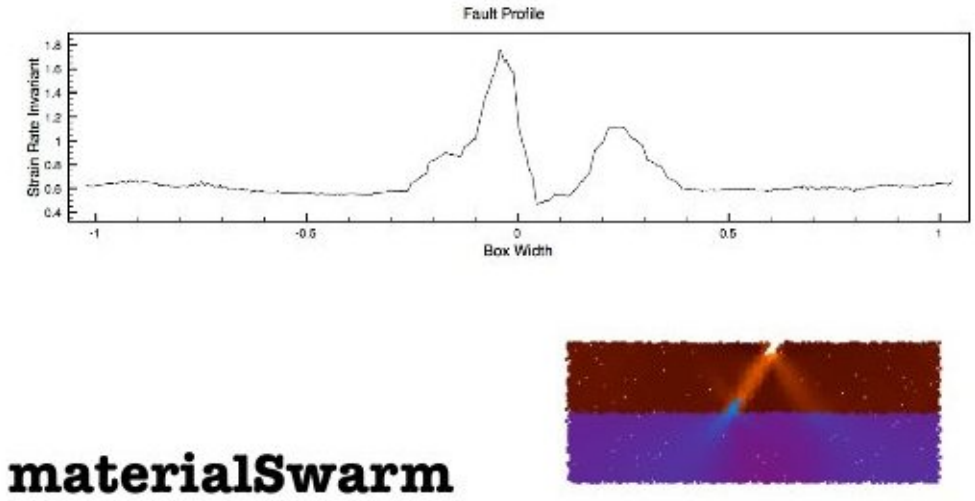


Figure 5.18: Fault Profile and Material Swarm.

- **Visualising Passive Tracer Particles** - The passive tracer particles do not have to be visualised to output field values. To visualise the tracer particles or verify their locations the following options are available:
  - *lucSwarmViewerBase*
    - lucSwarmViewer

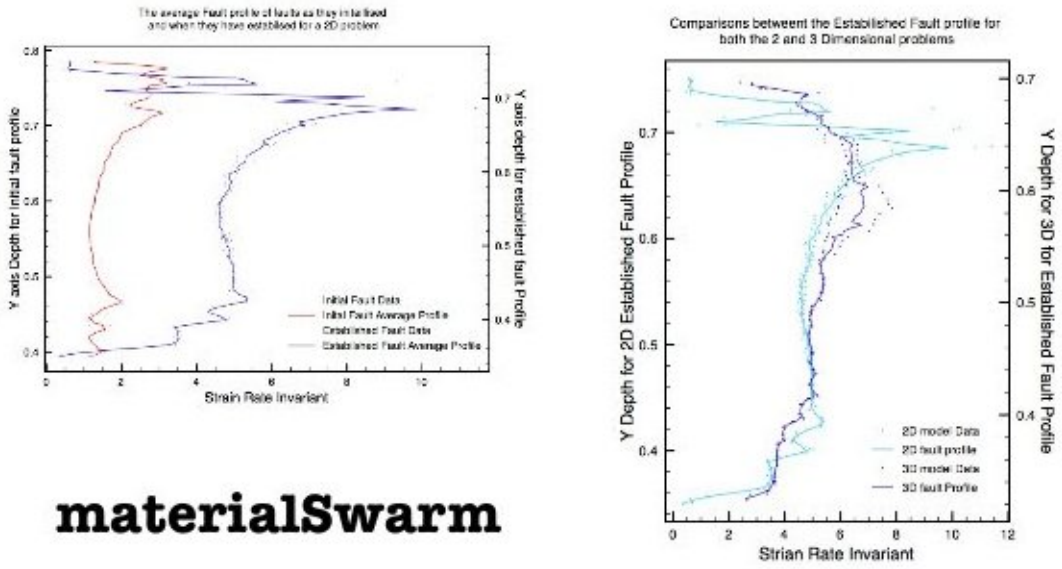


Figure 5.19: Strain Rate Invariant Comparisons using output from *Underworld_SwarmOutput*.

- *lucSwarmRGBColourViewer*
- *lucHistoricalSwarmTrajectory*
- *lucDrawingObject* - maskValue, maskTolerance, maskType.
  - *lucOpenGLDrawingObject* - No extra options to be added.
    - *lucSwarmViewerBase* - Extra options include *colour*, *ColourVariable*, *OpacityVariable*, *MaskVariable*, *drawParticleNumber*, *sameParticleColour*, *subSetEvery*, *positionRange*, *minPositionX*, *minPositionY*, *minPositionZ*, *maxPositionX*, *maxPositionY*, *maxPositionZ*.
    - *lucSwarmViewer* - Extra options include *pointSize*, *pointSmoothing*.
    - *lucSwarmRGBColourViewer* - Extra options include *ColourRedVariable*, *ColourGreenVariable*, *ColourBlueVariable*.
    - *lucHistoricalSwarmTrajectory* - Extra options include *colour*, *historySteps*, *historyTime*, *lineWidth*.

5.17.2 Examples

The following shows examples on the use of passive tracers in gLucifer.

- **lucSwarmViewer** - For visualising and verifying particle locations, the *lucSwarmViewer* component must be included in the XML file.

  For example:

```
<struct name="passiveTracerDots">
   <param name="Type">lucSwarmViewer</param>
   <param name="Swarm">passiveTracerSwarm</param>
   <param name="colour">Black</param>
   <param name="pointSize">5.0</param>
</struct>
```

  Where the *colour* and *pointSize* parameters can be varied. See *Passive Tracer Output using lucSwarmViewer.* (page 59) for sample output image. This example so that a variable can be assigned on the particles.

  For example:

```
<struct name="BCAStrainRateParticleVariable">
   <param name="Type">FeSwarmVariable</param>
   <param name="FeVariable">StrainRateInvariantField</param>
   <param name="Swarm">materialSwarm</param>
</struct>
```

The following example masks everything but the material and 3D space selected.

```
<struct name="Block">
   <param name="Type">lucSwarmViewer</param>
   <param name="Swarm">materialSwarm</param>
   <param name="ColourVariable">BCAStrainRateParticleVariable</param>
   <param name="ColourMap">LowerMantleColourMap</param>
   <param name="MaskVariable">materialSwarm-MaterialIndex</param>
   <param name="maskType">EqualTo</param>
   <param name="maskValue">2.0</param>
   <param name="pointSize">1.0</param>
   <param name="positionRange">True</param>
   <param name="minPositionX">1.0</param>
   <param name="maxPositionX">1.5</param>
   <param name="minPositionY">-10</param>
   <param name="maxPositionY">10</param>
   <param name="minPositionZ">1.5</param>
   <param name="maxPositionZ">10</param>
</struct>
```

See *Tiramisu 3D Cutout from lucSwarmViewer.* (page 60) for sample image output.

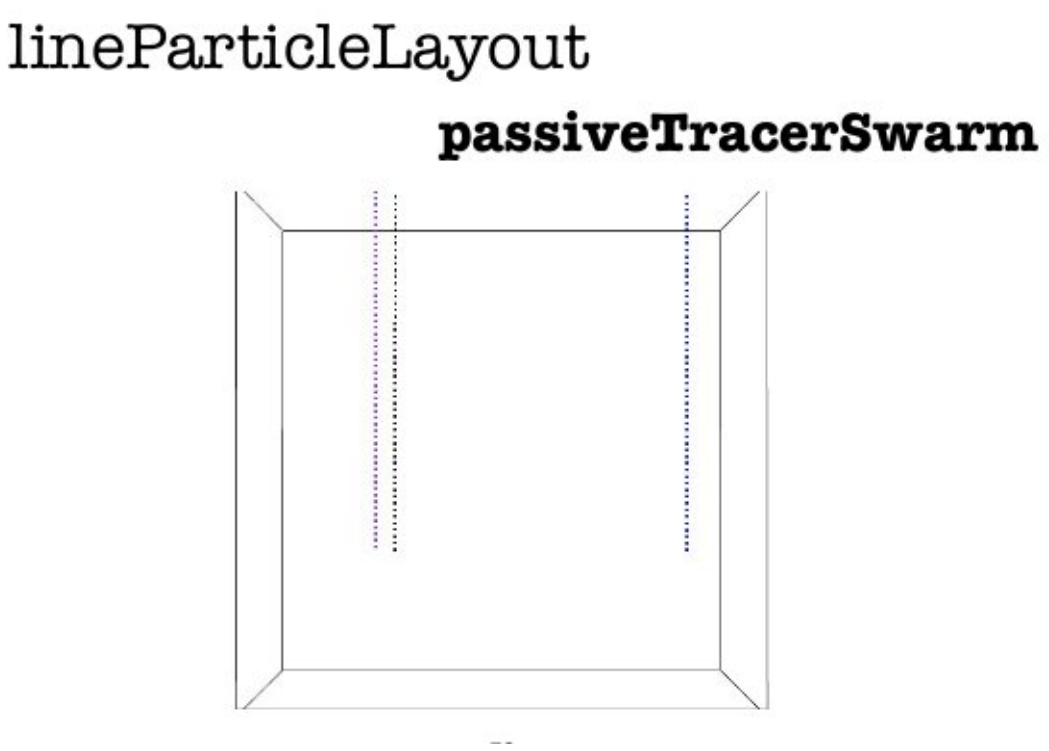


Figure 5.20: Passive Tracer Output using *lucSwarmViewer*.

- **lucSwarmRGBColourViewer** - This is equivalent to the lucSwarmViewer, but one can apply a different variable to each of the R (*red*), G (*green*) and B (*blue*) channels individually. If the channel is not attached to a swarm variable, then it assumes the value *0.0*
- **lucHistoricalSwarmTrajectory** - This allows trajectories to be viewed as part of the passive tracer particles. The *lucHistoricalSwarmTrajectory* must be included in the XML input file.

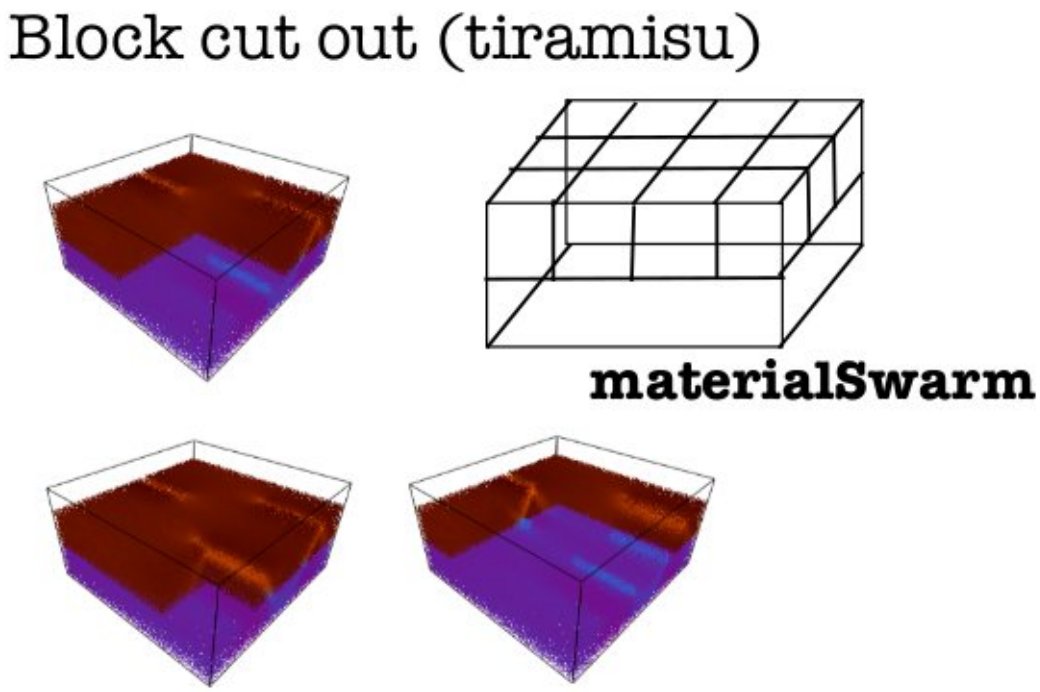


Figure 5.21: Tiramisu 3D Cutout from *lucSwarmViewer*.

For example:

```
<struct name="TracerTrajectoriesColourMap">
   <param name="Type">lucColourMap</param>
   <param name="colours">Red Orange Green</param>
   <param name="dynamicRange">true</param>
</struct>
<struct name="passiveTracerLines">
   <param name="Type">lucHistoricalSwarmTrajectory</param>
   <param name="Swarm">passiveTracerSwarm</param>
   <param name="ColourMap">TracerTrajectoriesColourMap</param>
   <param name="historyTime">50</param>
   <param name="historySteps">20</param>
   <param name="lineSize">2.0</param>
</struct>
```

See *Plane Particle View of lucHistoricalSwarmTrajectory.* (page 61) for sample image output.

5.17.3 Other Options

The following list outlines additional options that can be set when using passive tracers:

- The length of the trajectory lines can be controlled using the *historySteps* and *historyTime* parameters.
- *historySteps* controls the number of timesteps for which the *passiveTracer* particle coordinates are stored. The default is set to *500*. This can be set to match the value passed to the *maxTimeSteps* parameter.
- *historyTime* is not essential. If included, this limits which of the stored coordinates are displayed based on the total time. If using *historyTime*, make sure that historySteps is set to a value large enough to store all the required coordinates.
- Trajectory lines can be plotted using one colour by creating a *ColourMap* that uses that particular colour as the variable *Colour* cannot be used in place of the *ColourMap* component.

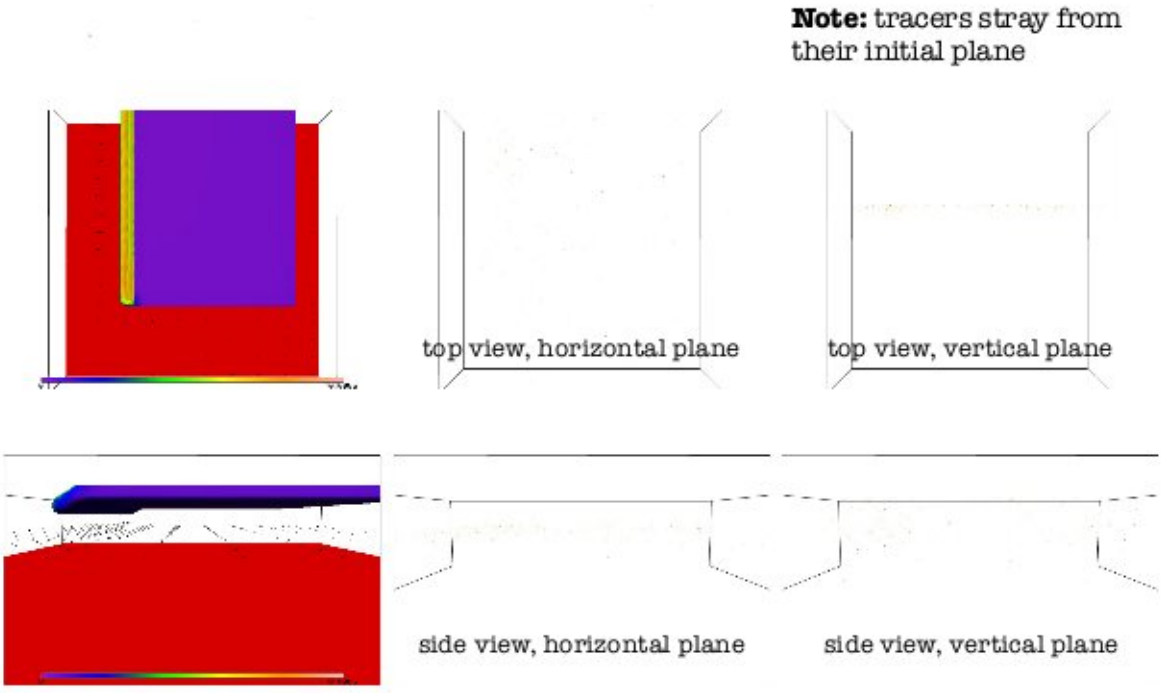


Figure 5.22: Plane Particle View of *lucHistoricalSwarmTrajectory*.

CHAPTER

SIX

VISUALISATION

6.1 Introduction

The gLucifer visualisation framework provides Underworld with a built in set of tools customised to plot, render and explore your model data with results available immediately as the simulation progresses. As a general visualisation analysis tool kit it can be used with any grid based computational models using the StGermain framework.

Instead of doing all the visualisation and analysis stages of your workflow as a post-processing step using large amounts of data gathered from preceding simulation runs, gLucifer allows you to specify visualisation elements in the XML scripts that define your model.

During the initial construction of a model, it will likely be more useful to set drawing elements to plot large areas of data or visualising areas where it is guessed interesting features may turn up.

The visualisation can then be refined in an iterative process to produce more exacting results and hone in on areas of interest, while reducing the amount of unnecessary output data produced.

6.2 Changes in gLucifer 2

This section is only relevant if you are interested in details of the differences between the previous gLucifer version and reasoning behind the changes, if not please skip ahead.

gLucifer 2.0 provides a new visualisation system for Underworld which attempts to be completely backward compatible with previous gLucifer XML scripts and while providing an expansion of same orignal set of features it has a very different design and approach. This backwards compatibility has been extensively tested but if you have any issues with visualising a previously working model, please let us help by posting a question in the Visualisation sub-forum of the Underworld Project forum: https://underworldproject.org/forums/viewforum.php?f=26.

The gLucifer toolbox provides a set of StGermain components to define windows, viewports, cameras, various "drawing objects" that control visualisation elements along with input and output components. All these components previously interacted directly with a parallel OpenGL rendering engine that could produce image and video output while the simulation ran, on machines ranging from supercomputers to multi-core workstations and single processor notebooks.

This approach had some limitations.

- The vast array of combinations of machine architecture, operating system and graphics processing capability that required support in the OpenGL rendering engine.
- Trade-off between supporting so many target systems: using any modern and advanced OpenGL rendering features was impossible.
- Large number of dependencies required for building Underworld.
- Inability to use transparency with parallel compositing.
- 3d model data discarded after image/video output, no viewing/exploration of the visualised data except from the predefined points of view.

- All decisions about rendering defined before the run, even the slightest visual change (eg: colours) could not be adjusted without re-running the simulation steps that needed re-visualising.
- Entanglement of areas of functionality that should be separate: data analysis processing combined with the 3d rendering of results.

In order to solve these problems, a highly modular approach has been taken with gLucifer 2.

- **Data Analysis** provides the familiar gLucifer/StGermain component toolbox interface.
  - Data persistance API - for creating 3D model data, collating the data efficiently in parallel and storing it in a database. Contains the original gLucifer 1 code.
  - No OpenGL calls, all output is replaced with calls to the Data API. All output from Underworld is to database files - images and video can be produced but this is done by calling another program (the gLucifer viewer/renderer), thus Underworld has no OpenGL, imaging or window system dependencies.
  - InputFormat, OutputFormat, Windowing, RenderingEngine and WindowInteraction components/modules are all removed and replaced by the viewer/renderer.
  - Window, Viewport and Camera components remain but simply define a structure which is written to the database and used by the renderer module.
- **Rendering and Output**
  - *Rendering library* load 3D model data from the database and producing OpenGL renderings
  - *Output* Write image & video output of the render to files.
  - *Interactive viewer* Additional features to provide a user interface to interact with and modify the model.

Both interactive and background rendering are still possible, via an intermediate database to store the visualisation data. Background rendering is done in a separate process to render 3d vis data to image frames from the database file(s). Interactive rendering is available by loading the database into the interactive viewer.

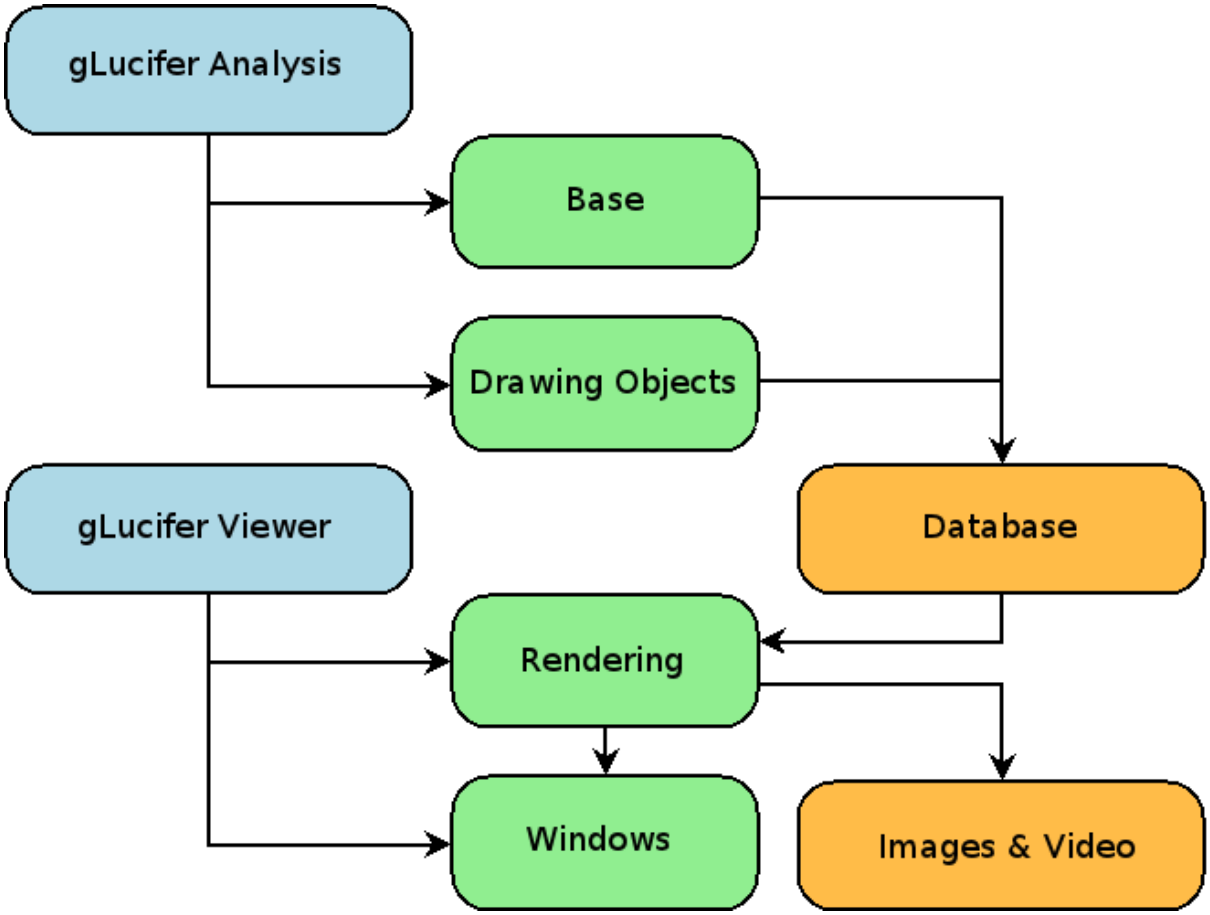
6.3 Framework

The gLucifer analysis modules are flexible and extensible parallel modules which produce can visualisation data from any StGermain-based code as it is generated, they fall into two main categories:

- **Analysis: Base** - Contains the foundation of the gLucifer Analysis module (Camera, Colourmap, DrawingObject, Viewport, Window) and the Data API (Database).
- **Analysis: Drawing Objects** - The functional components of the gLucifer Analysis module. Contains implementations of the *lucDrawingObject* class which provide ways to visualise particular sets of data in specific ways. The results produced by these components are passed to the Data API to be written to the render database.

The gLucifer Viewer provides a general purpose 3d renderer and a set of cross-platform data interaction and output capabilities:

- **Viewer: Renderer** - Contains Rendering library and the Output / Interaction modules. Input data comes from database files produced by the Data API, producing output as images, movies, or interactive exploration of the data.
- **Viewer: Windows** - Contains platform specific code and modules for using the viewer on different target rendering platforms, off-screen: (AGLViewer - Mac only, OSMesaViewer - all except windows) and interactive: (GlutViewer - Mac/Linux, SDLViewer - all platforms including windows, X11Viewer - all except windows).



6.4 gLucifer Build

Underworld pre-packaged binaries are available from the Underworld Project website for Mac OS X and Ubuntu Linux. gLucifer Viewer pre-packaged binary installer packages are also available for Mac OS X, Ubuntu Linux and Windows. http://www.underworldproject.org/downloads.html

If you are using the pre-packaged binaries you will not need to build Underworld from source code so please skip this section.

The gLucifer modules are built by default with Underworld, in fact it is much simpler to get up and running than previous versions as the analysis module has no additional dependencies.

This means that gLucifer can be used for analysis on any system that runs Underworld, without requiring an OpenGL implementation or various image/video output libraries. Even the SQLite3 database component is provided as part of the library so requires no external dependency installation.

If no OpenGL implementation is available, the output will be database files only which must be run through the output or interactive viewer modules to view the results. This can easily be done on another machine where OpenGL drivers are available.

If OpenGL capability is present, two additional executables are produced with Underworld:

- **build/bin/gLucifer** the gLucifer viewer, provides various command line options to produce images and videos from a provided visualisation database file. Requires OpenGL and either X11, SDL, Glut, AGL or OSMesa for output and optionally libpng and libavcodec. If an interactive environment is available (X11,SDL,Glut) allows full user interation with the model data.
- **build/bin/gLuciferOS** A non-interactive, off-screen only version of the gLucifer viewer. Will always use OSMesa or AGL if available, otherwise is the same as the standard viewer. Called by Underworld when set to automatically produce output images.

6.4.1 gLucifer Renderer & Viewer Dependencies

OpenGL capable libraries, to render images at least one of the following must be available

- SDL - Cross-platform display library, works on all platforms when required SDL library package is installed.
- X11 - Available on Linux and other Unix based systems including Mac OS X (if installed), is usually installed on clusters but only usable without access to a display if the Xvfb (X virtual frame-buffer) is installed.
- Glut - Cross-platform display library, available on Mac OS X and Linux if installed.
- AGL - Off-screen rendering for Mac OS X only.
- OSMesa - Off-screen rendering using the Mesa software renderer, used when no display is available such as on HPC clusters.

Additional libraries, all of which are optional

- libPNG - allows PNG image output with optional transparent backgrounds. If not available, JPEG images will be produced with the included JPEG encoder (libJPEG is not required).
- libavcodec - video encoder, allows movie output, available for all systems often as part of the ffmpeg package.

6.4.2 Output files produced by the Config system

Configuring Underworld before building (running configure.py) should be a automatic process but if problems occur it can be useful to know that there are now up to 4 separate configurations saved depending on packages located:

- config.cfg - the base Underworld configuration, also used to build the gLucifer analysis and data output modules.
- output.cfg - OpenGL and image/video output capabilities available for gLucifer rendering (used to build rendering library)
- offscreen.cfg - Window output capabilities available for off-screen rendering (used to build gLuciferOS)
- viewer.cfg - Window output capabilities available for interactive/on-screen rendering (used to build gLuciferOS)

The content of the last three will define how the gLucifer Viewer and Renderer are built and what capabilities are available for rendering and producing images in the build.

6.5 Required XML Components

The list below outlines the minimum required components to be provided in the XML file to be able to display visualisation output in gLucifer:

- gLucifer toolbox
- A *Window* (page 83)
- At least one *Viewport* (page 84) with * A *Camera* (page 84) * And *Drawing Object* (page 85)s

These are the essential requirements for any rendering. They can be defined by hand coding XML or by including template XML files.

6.6 Quickstart

First you will require the gLucifer toolbox, this is specified by adding it to the *import* list at the start of one of your XML input files.

For example:

```
<list name="import">
   <param> gLucifer </param>
</list>
```

The easiest way to get started is to include one of the predefined visualisation Viewport template files containing a viewport and set of drawing objects which can be included in the XML input file.

For example:

```
<include>Underworld/Viewports/StrainRateInvariantVP.xml</include>

<struct name="components">
   <struct name="window">
      <param name="Type">lucWindow</param>
      <param name="Viewport">StrainRateInvariantVP</param>
   </struct>
</struct>
```

The *Window* (page 83) component is the basic component used to create a visualisation, it must contain at least one *Viewport* (page 84). By default it uses png or jpeg image output and creates a output window large enough for each viewport sized to 400x400 pixels. *StrainRateInvariantVP.xml* contains a pre-packaged set of drawing objects, a camera, colour-map and viewport designed to visualise the strain rate invariant field.

6.6.1 Templates

The Underworld/Viewports directory contains a series of visualisation files that can be used as templates. Users can:

- Call a template visualisation file directly from within the XML file, or
- Call their own visualisation file which could contain modifications to a called template visualisation file.

Some of these are already used in the template input files, so if a template XML file is already being included in the user input file, another visualisation template doesn't need to be included again (see *Creating your own XML file* (page 21) and *XML Examples* (page 25)). Any template visualisation files which is included in Underworld can be used and be a basis customising the model's appearance and outputs. As per other XML file parameters, parameters used in the visualisation template can also be overridden (see *XML Examples* (page 25)).

6.6.2 Images and Movies

gLucifer will render a window at each dump timestep in the calculation, specified by the parameter *dumpEvery*:

```
<element type="param" name="dumpEvery">1</element>
```

Each time it is rendered, the window will be saved to the database. This output can be read by the viewer to produce images or view the model interactively at any time.

There is also the option to produce images on the fly, this will incurr a slight overhead but is enabled by default for historical reasons. By default, Underworld calls the *gLuciferOS* binary to do this, so it must have been built or no images will be produced. You can also define the $GLUCIFER_PATH environment variable to point Underworld to the renderer executable you want to use.

Images are saved in PNG format if libpng is available, otherwise the built in JPEG encoder is used (which has no support for transparent backgrounds).

To produce an images of the first 5 time steps when passed a visualisation database as a command line parameter.

build/bin/gLucifer output/gLucifer.gldb -w -0 -4

Similarly, to produce a video file from a previously created database of time steps 5 to 10.

build/bin/gLucifer output/gLucifer.gldb -m -5 -10

To output a movie file using gLucifer, a video encoding library must be installed when Underworld is configured (see *Additional Software* (page 7)).

Currently only the libavcodec/libavformat (ffmpeg) library is supported. The pre-built viewer binaries have built in video encoding support.

For more details on command line options and the gLucifer Viewer/Renderer see *Interactive visualisation* (page 85).

6.7 Database Output

As well as exporting image and video files, gLucifer writes a database file containing the 3d model objects created during visualisation. The database component is now integral to gLucifer and is required, so if none is specified in the input XML then each window uses a default database file, named *gLucifer.gldb* in the default output directory.

If you wish to change the default options to control the database output, you'll need to manually create the database component:

```
<struct name="components">
 <struct name="db">
   <param name="Type">lucDatabase</param>
   <param name="deleteAfter">0</param>
   <param name="writeimage">True</param>
   <param name="splitTransactions">False</param>
   <param name="transparent">False</param>
   <param name="compressed">True</param>
   <param name="singleFile">True</param>
   <param name="filename">gLucifer</param>
   <param name="vfs"></param>
   <param name="timeUnits"></param>
 </struct>

 <struct name="window">
   <param name="Type">lucWindow</param>
   <param name="Database">db</param>
   ...
 </struct>
</struct>
```

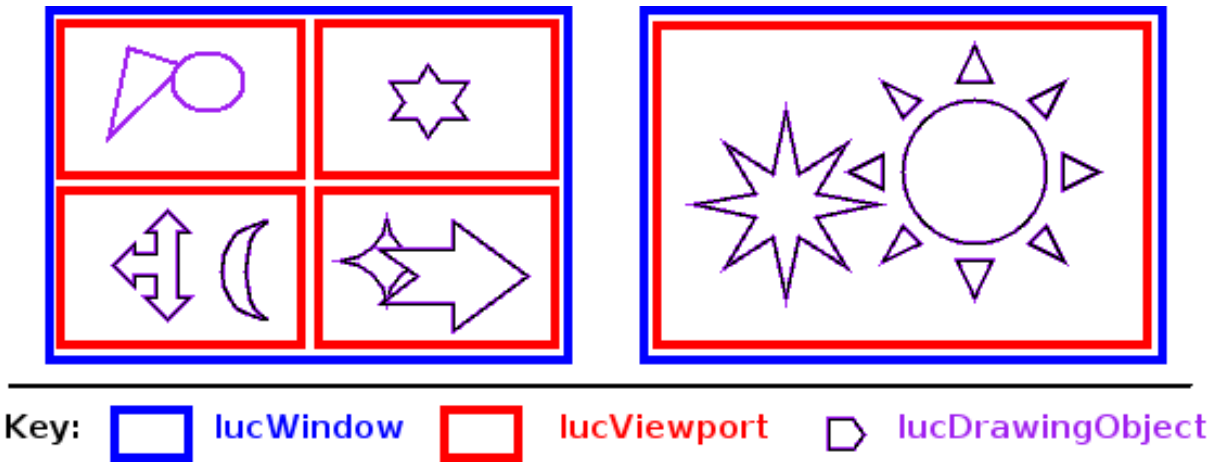
For descriptions of the database parameters see *Database* (page 83)

6.8 Windows and Viewports

Visualisation is done by opening at least one rendering window. This window can then be divided into different viewports. The viewports are really *sub-windows* allowing display of different objects and views of a model inside the single parent window:

The orientation of the Viewports in one window is decided by the way they are listed in the visualisation file *Viewport* (page 84) list. The number of *param* lines determines the number of vertical divisions of the window, the number of named viewports listed inside each param line decides the horizontal divisions of the window.

For example:




```
<struct name="components" mergeType="merge">
   <struct name="window">
      <param name="Type">lucWindow</param>
      <list name="Viewport">
         <param>ParticleDensityVP</param>
         <param>StrainRateInvariantVP</param>
      </list>
   </struct>
</struct>
```

This will display the two viewports on top of each other in one window: Vertical viewport layout

Another example:

```
<struct name="components" mergeType="merge">
   <struct name="window">
      <param name="Type">lucWindow</param>
      <list name="Viewport">
         <param>ParticleDensityVP StrainRateInvariantVP</param>
      </list>
   </struct>
</struct>
```

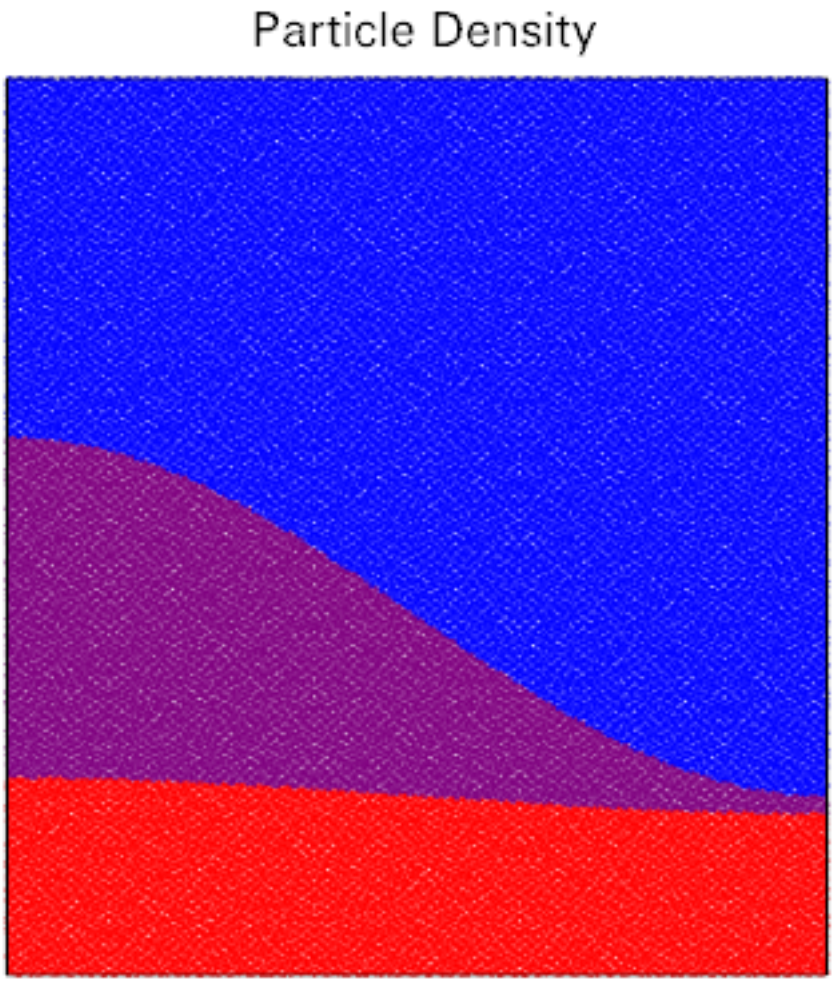
This will display the two viewports side by side: Horizontal viewport layout

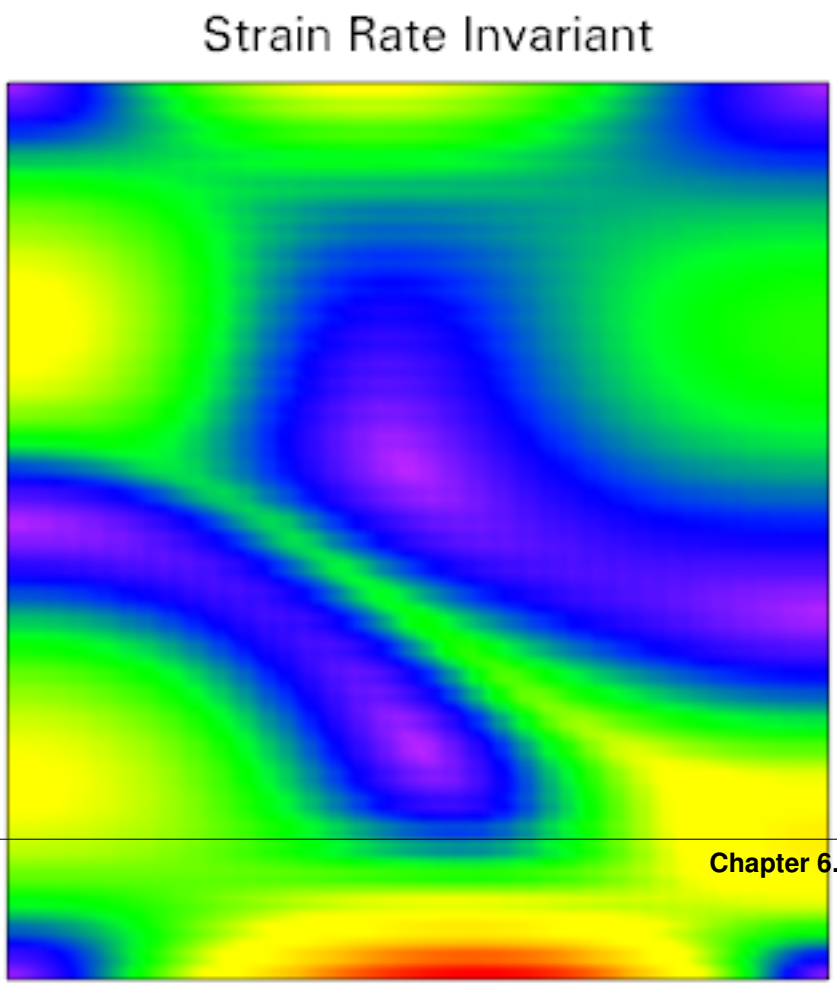
You can combine these methods too: Horizontal and Vertical viewport layout

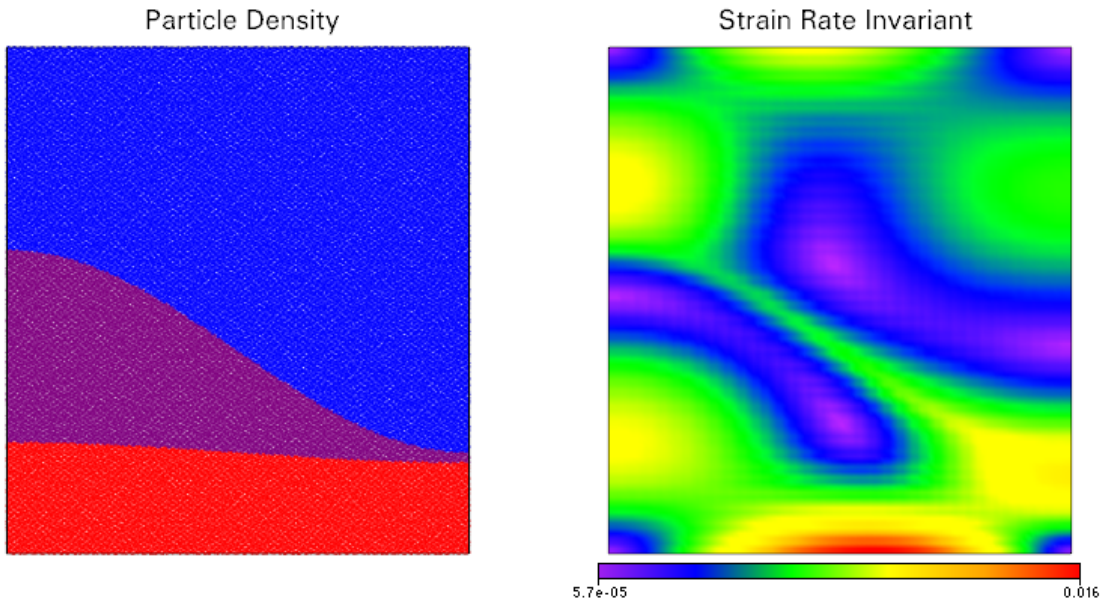
```
<struct name="components" mergeType="merge">
   <struct name="window">
      <param name="Type">lucWindow</param>
      <list name="Viewport">
         <param>ParticleDensityVP StrainRateInvariantVP</param>
         <param>StrainRateInvariantVP</param>
      </list>
   </struct>
</struct>
```

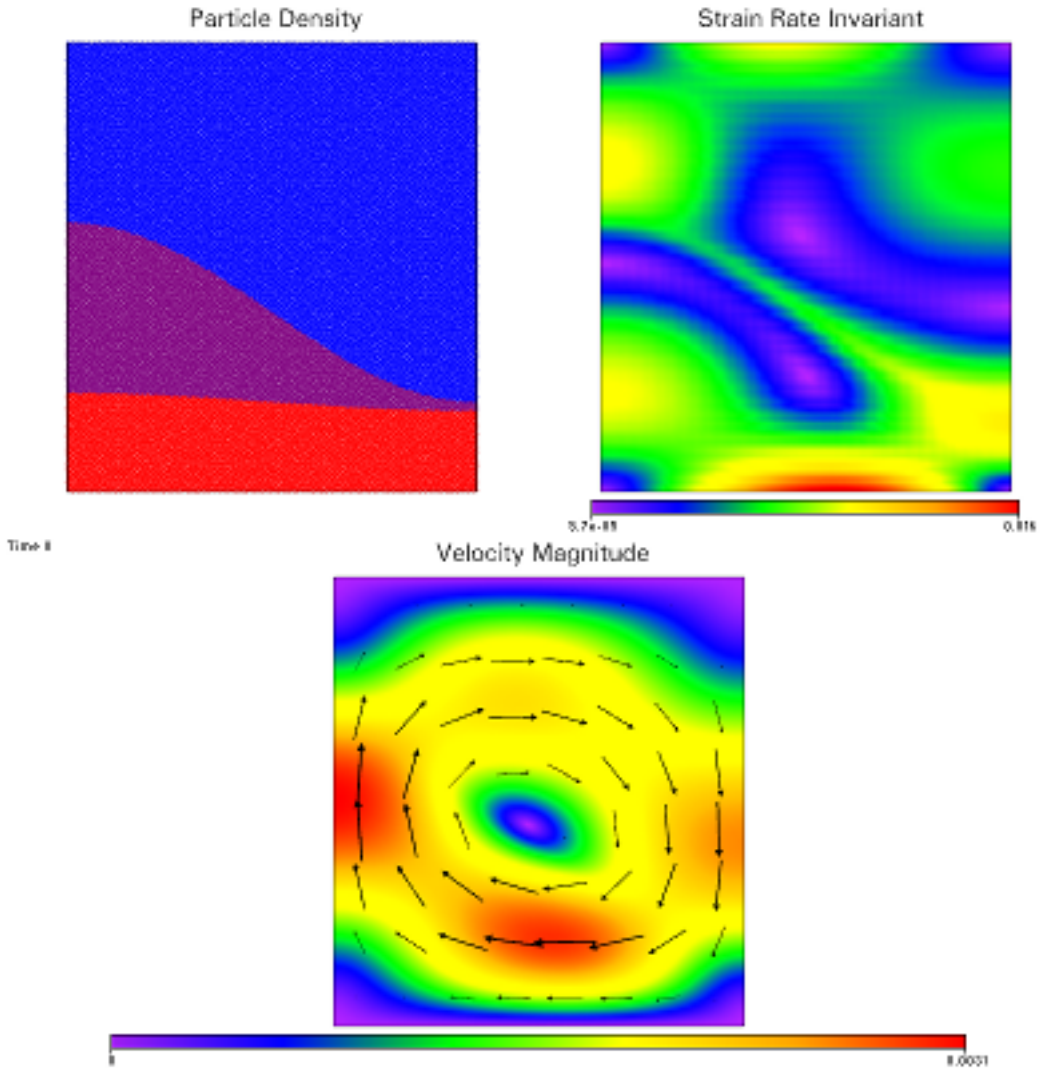
Note: If you don't specify the window width and height, the size of the window will be automatically calculated to allow at least 400 x 400 pixels for each viewport. If you do specify the window size the space will be divided equally between each viewport in the horizontal and vertical directions.

For more details of window and viewport parameters see *Window* (page 83) and *Viewport* (page 84).


Particle Density


Strain Rate Invariant


Particle Density
Strain Rate Invariant
5.7e-05
0.016


Particle Density
Strain Rate Invariant
Velocity Magnitude

6.9 Camera Adjustment

The easiest way to set the camera is supply it with a field variable to centre on and allow the default settings for the camera to automatically zoom to fit the model in the viewport.

If zooming in or out further than the default range, the model position can be set using the *translateX*, *translateY*, *translateZ* parameters. Adjusting the *translateZ* parameter will give the effect of zooming in or out by moving the model closer (negative) or further away (positive).

For example:

```
<struct name="camera" mergeType="replace">
   <param name="Type">lucCamera</param>
   <param name="CentreFieldVariable">PressureField</param>
   <param name="translateZ">-1.5</param>
</struct>
```

Similarly the translateX parameter will move the model to the left (negative) and right (positive) and translateY will move it up (positive) and down (positive). To rotate the model, the following rotation parameters are available:

- **rotateX** - rotate about X axis, tilt forward and back (*pitch*).
- **rotateY** - rotate about Y axis, turn left and right (*yaw*).
- **rotateZ** - rotate about Z axis, twist clockwise and anti-clockwise (*roll*).

For example:

```
<struct name="camera" mergeType="replace">
   <param name="Type">lucCamera</param>
   <param name="rotateX">90</param>
</struct>
```

This will rotate the model about the x-axis by 90-degrees so the model will be viewed from above.

If the simulation bounding area will change as the model runs (expanding or compressing) then setting the *autoZoomTimestep* parameter to a higher value than the default is a good idea.

The default is 0 which will automatically zoom the camera once only when the model is first displayed. To automatically adjust the zoom every timestep set it to 1.

For example:

```
<struct name="camera" mergeType="replace">
   <param name="Type">lucCamera</param>
   <param name="autoZoomTimestep">1</param>
</struct>
```

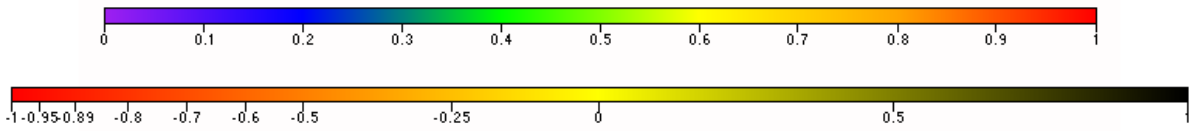
To disable all automatic camera zoom set autoZoomTimestep to -1.

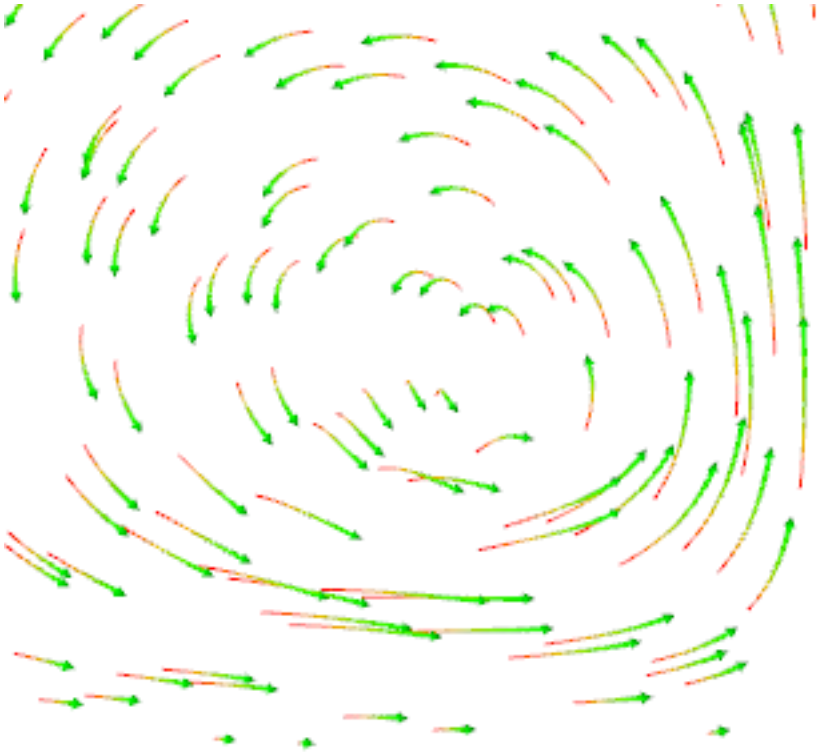
For more details of the camera object parameters see *Camera* (page 84).

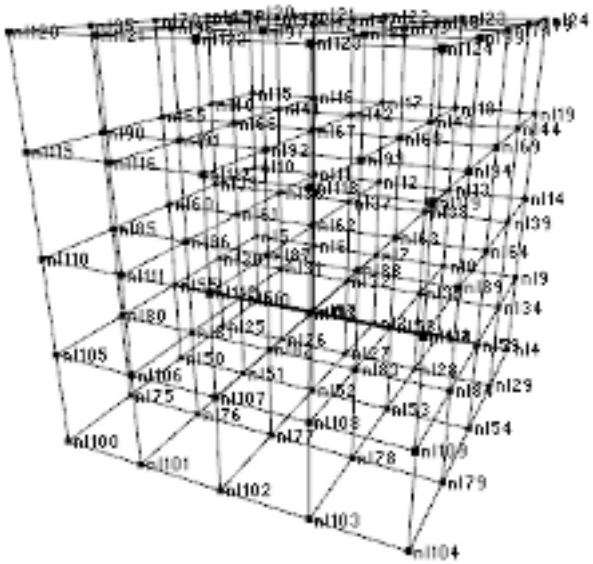
6.10 Drawing Objects

The list below outlines the available Drawing Objects in gLucifer which provide means to visualise particular sets of data in specific ways.

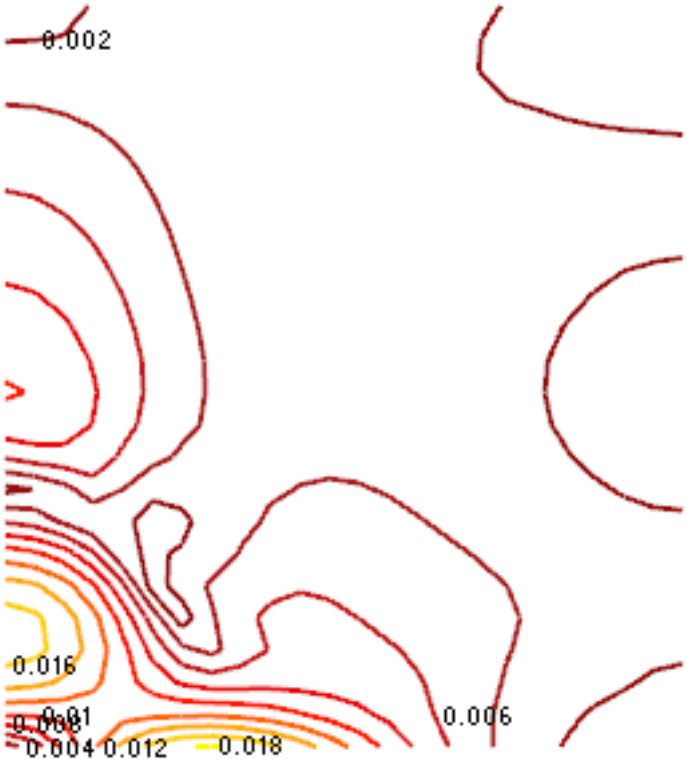
- **General**
  - *luColourBar* Draws a colour bar at the bottom of a *Viewport* (page 84) with labels showing the range of colours mapped to plotted values on the scale of a particular *Colour Map* (page 85).
  - *lucHistoricalSwarmTrajectory* Plot lines following the path of particles in a swarm.
  - *lucMeshViewer* Draws a grid to represent a StgFEM mesh.


0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
-1
-0.95
-0.89
-0.8
-0.7
-0.6
-0.5
-0.25
0
0.5
1



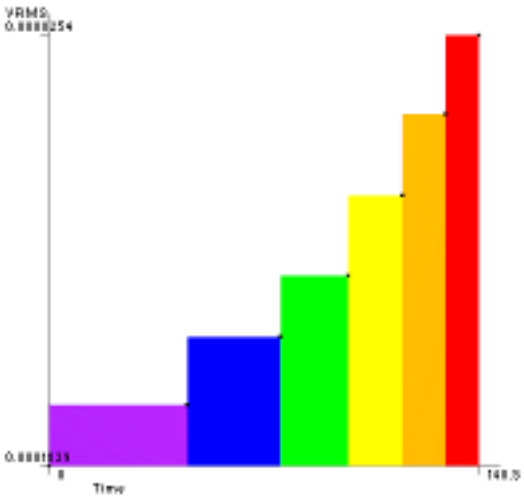


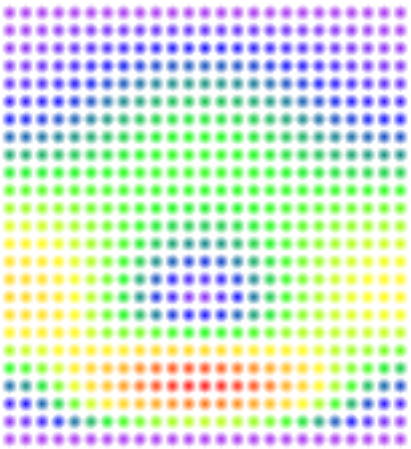
- *lucContour* - Visualises a scalar field by the use of contour lines at specific intervals. It gets the value of the field on a grid and uses a 2D form of the marching cubes algorithm to draw the contours.
- *lucContourCrossSection* - Draws contour lines at specific intervals over a specified 2d cross-section plane.

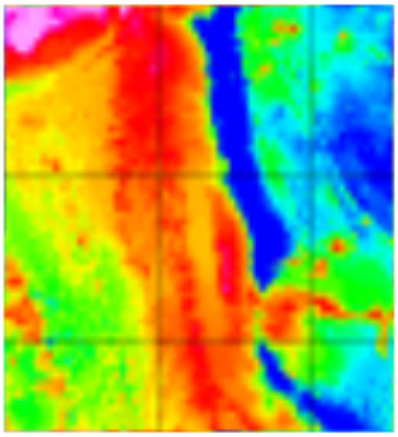


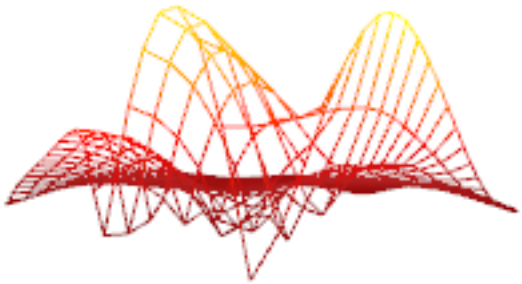
- *lucCrossSection* - Draws a 2d cross-section plane.
- *lucCapture* - Capture drawing commands, for use in plugins.
- *lucPlot* - Draw line and bar charts from data in text files.
- *lucFieldSampler* - Sample a field in a regular grid, display as points.
- *lucTextureMap* - Draws an image from a source file on disk into the rendered scene.

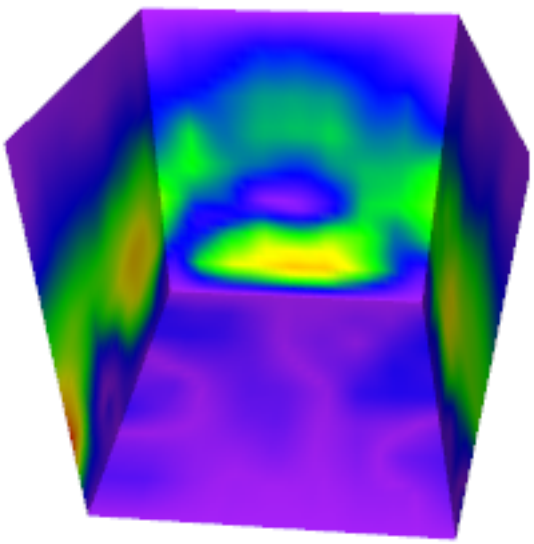
- **Surfaces**
  - *lucFeVariableSurface* - Plots a 3D surface showing the values of an *FeVariable* as a height map.
  - *lucScalarField* - Tiles the side walls of the model with colours derived from the value of a scalar field at the vertex of each tile.
  - *lucScalarFieldOnMesh* - As above, but plots on the mesh nodes rather than arbitrary points.
  - *lucScalarFieldCrossSection* - Tiles a plane with colours derived from the value of a scalar field at the vertex of each tile.
  - *lucScalarFieldOnMeshCrossSection* - As above, but plots on the mesh nodes rather than arbitrary points.
  - *lucIsosurface* - Visualises a scalar field in 3D by the use of a surface over which all the values of the field are constant (an isosurface). It gets the value of the field on a 3D grid and uses the marching Cubes algorithm to construct the surface.
  - *lucIsosurfaceCrossSection* - 2d cross section of an isosurface.
- **Particles**
  - *lucSwarmViewer* - Plots particles in a swarm as small dots.

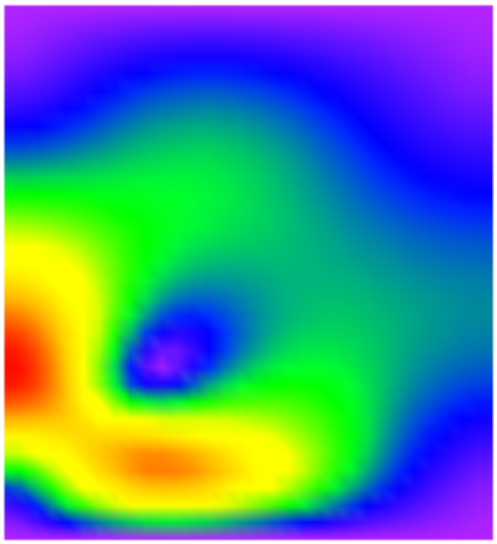

VRMS
Time

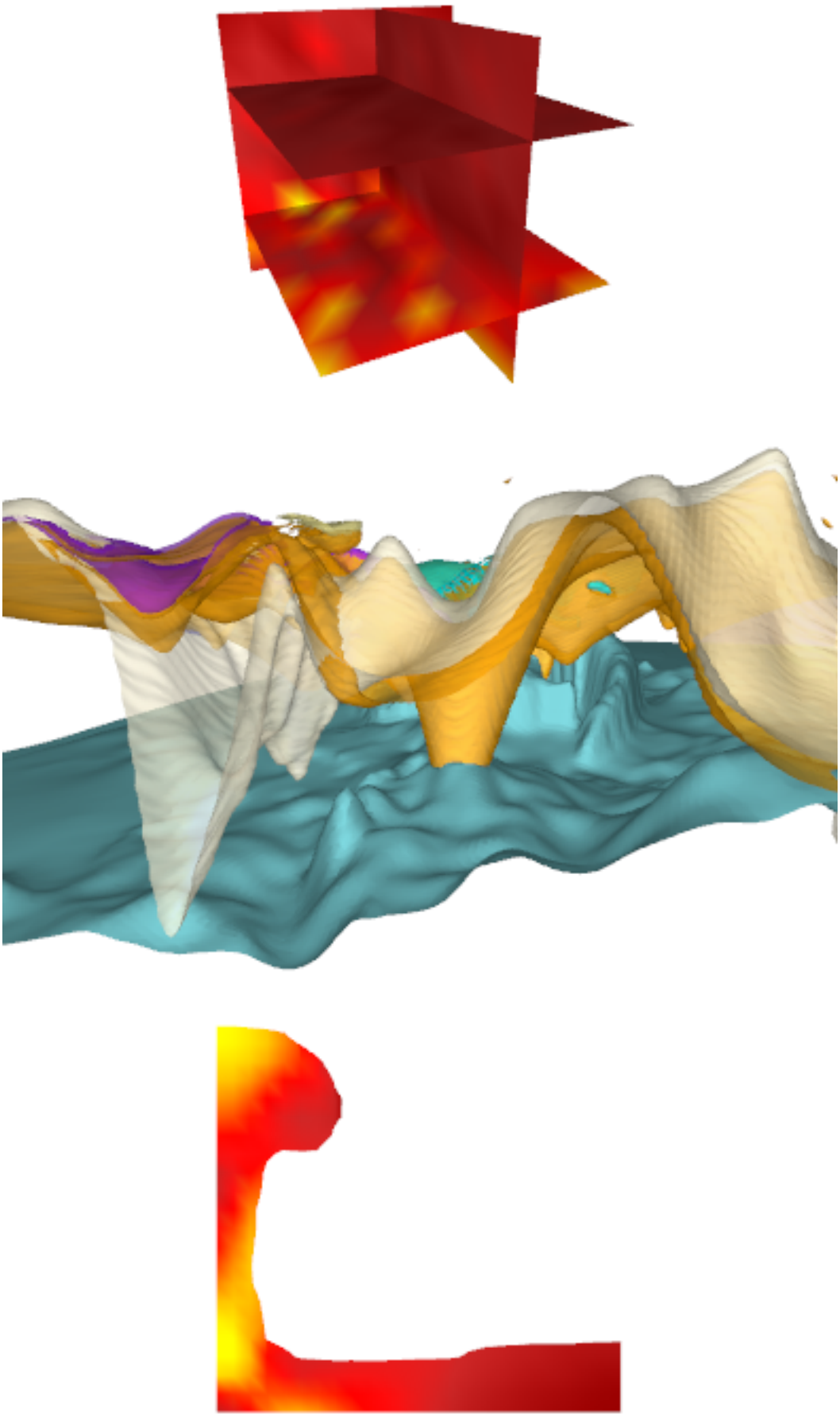


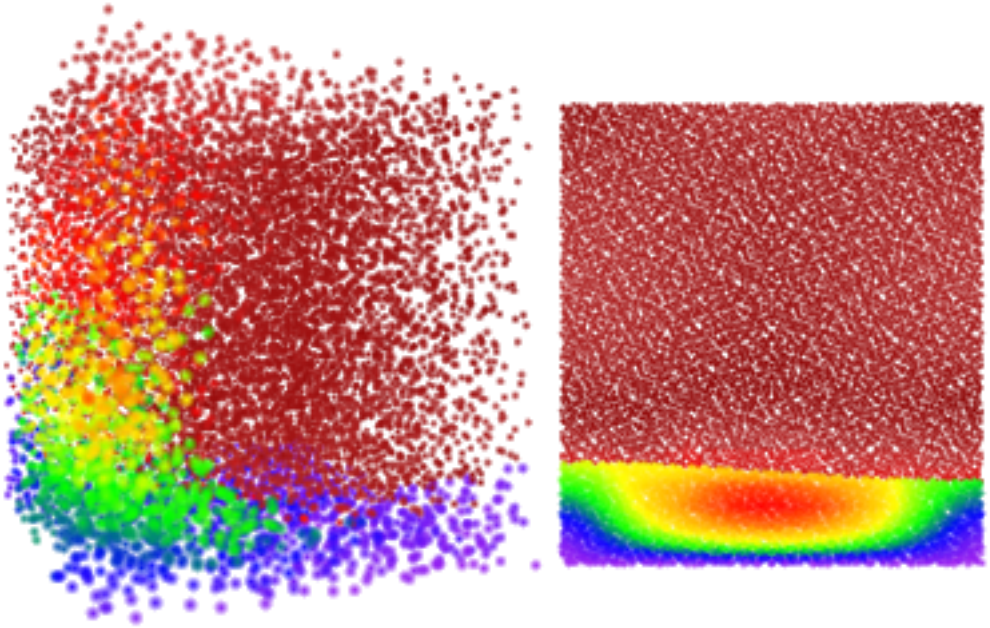




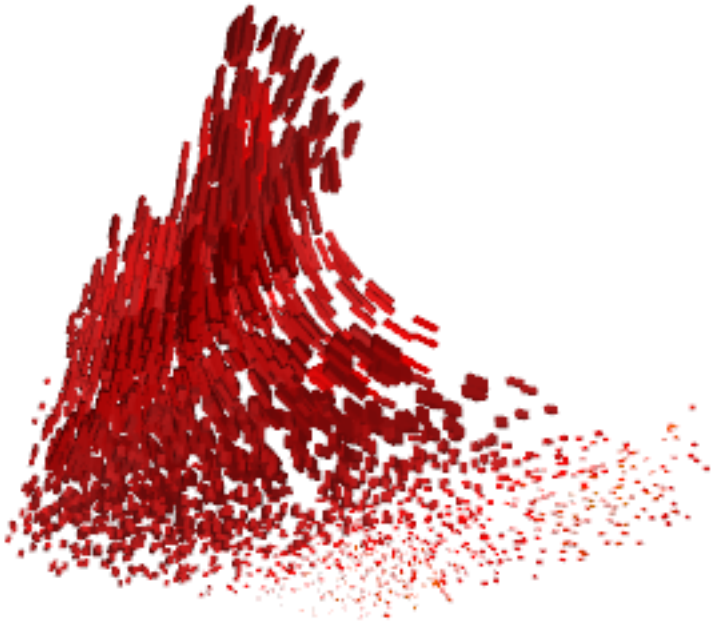






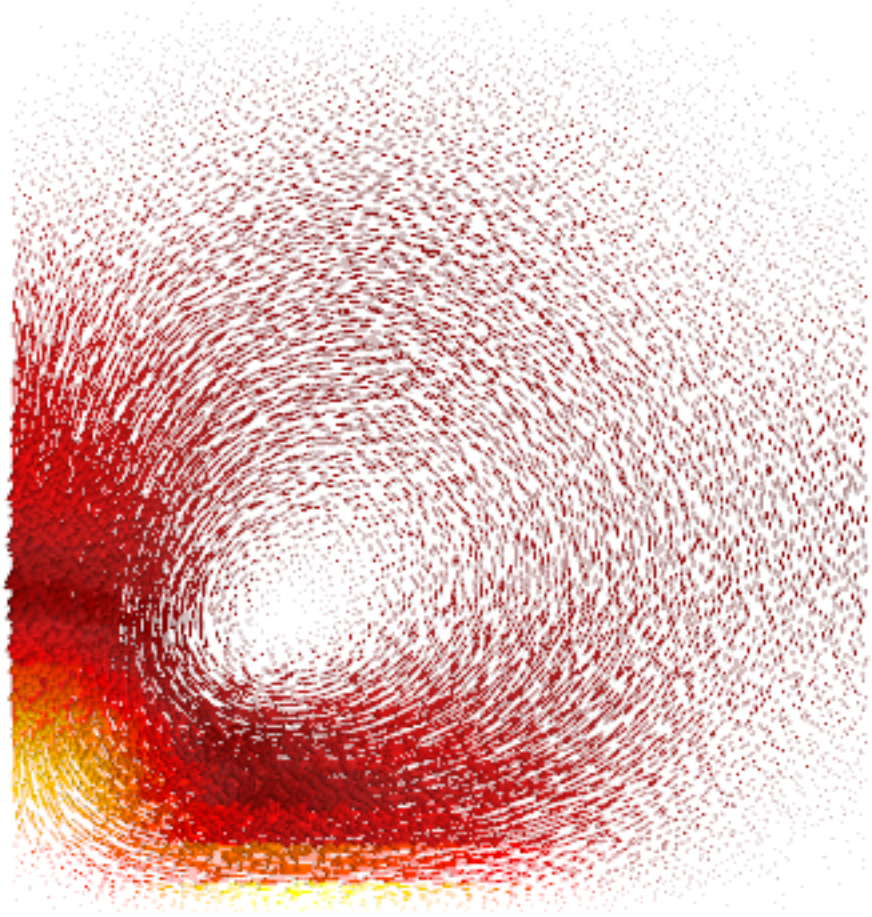


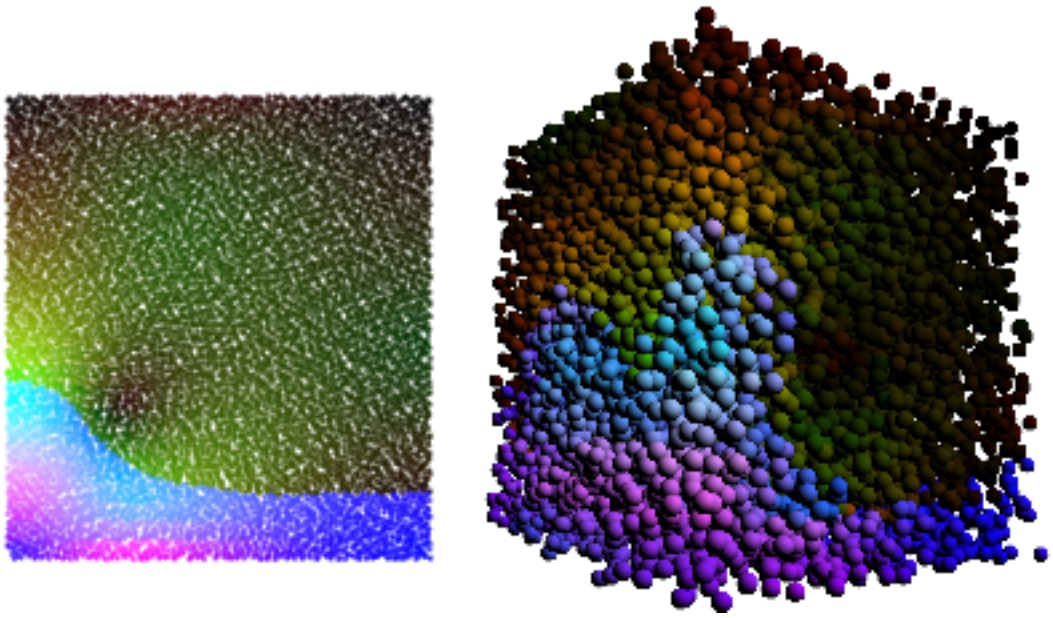
- *lucSwarmShapes* - Plots a 3d shape for each of the particles in a swarm, can be scaled in each direction using a variable.

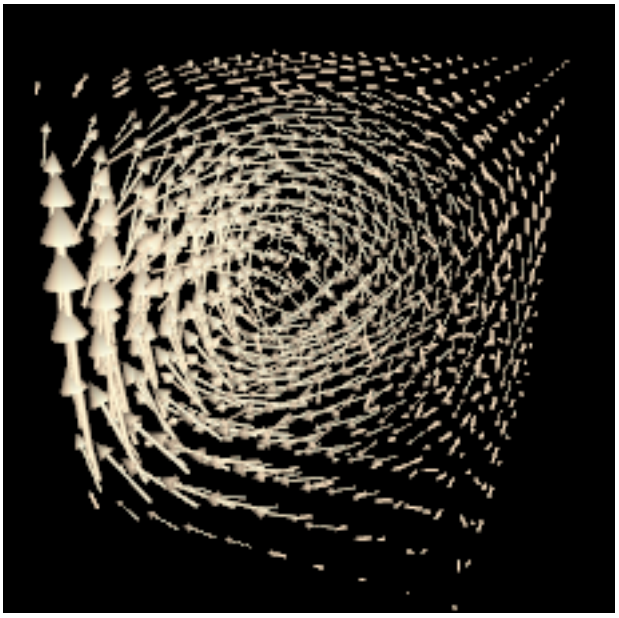


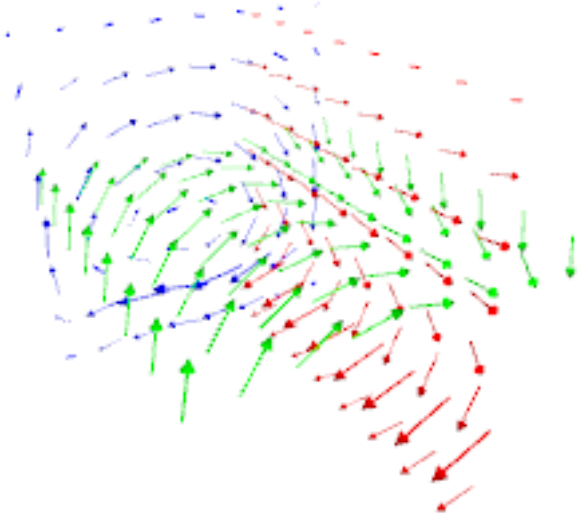
- *lucSwarmVectors* - Plots a vector quantity on particles in a swarm as arrows.
- *lucSwarmRGBColourViewer* - Same as *lucSwarmViewer* but allows mapping a different field to each of the red, green and blue components of the plotted particle colour.

- **Vectors**
  - *lucVectorArrows* - Samples the values of a vector field in a grid and draws arrows to represent the direction, scaled to represent the magnitude.
  - *lucVectorArrowCrossSection* - Samples the values of a vector field in a plane and draws arrows to represent the direction, scaled to represent the magnitude.
  - *lucEigenvectors* - Displays vector arrows of eigenvector values.

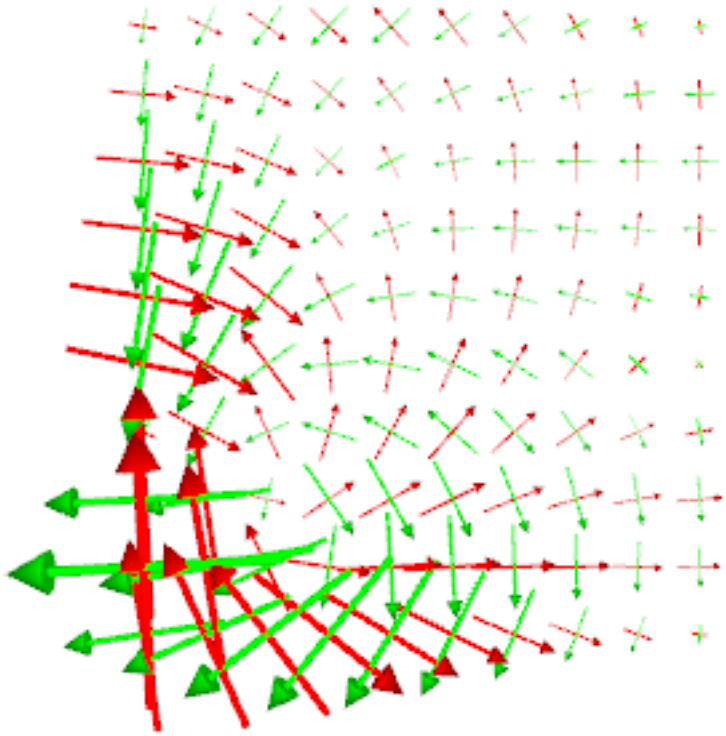








– *lucEigenvectorsCrossSection* - Displays vector arrows of eigenvector values over a specified cross section plane



Note: The title, timestep, border and axis objects have been removed from gLucifer 2 as all these options are now available as parameters on the viewport. The components are still allowed for backwards compatibility but should not be used in new models.

For descriptions of the global drawing object parameters see *Drawing Object* (page 85). For descriptions of parameters for setting up colour maps for drawing objects see *Colour Map* (page 85).

Details of parameters for individual drawing objects in the latest version of Underworld/gLucifer can be found in the component codex: http://underworldproject.org/codex-bleeding-edge/gLucifer.html

6.10.1 Creating 2D Slices

Cross sections of the 3D model can be taken in gLucifer and visualised as 2d slices. To take a slice of a scalar field use the *lucScalarFieldCrossSection* object and set either:

- **crossSectionX**, **crossSectionY** or **crossSectionZ** parameters to fixed coordinate values.
- **crossSection** to *x=Value*, *y=Value* and/or *z=Value*, where *Value* can be a proportion of the range in that direction (eg: x=0.5 for half of the X range) or a percentage (eg: y=75% to place at 3/4 of the way in to the Y axis range) or x/y/z=min/max to place at the minimum or maximum of the range (eg: z=min to place at back of Z range).

The **resolutionA** and **resolutionB** parameters define the sampling resolution of each of the cardinal direction of the cross-section plane. In the example below they are set to the element resolution.

The *lucSwarmViewer* object plots particles and can be set to colour the particles by the particle index number highlighting the different material areas. To show a slice through the particles you can limit the volume over

which particles are plotted by setting **positionRange=true** and setting the **minPositionX** and **maxPositionX** and/or corresponding Y or Z positions as desired.

Examples:

```
<!-- For a top view showing velocity mag field and velocity arrows in horizontal cross sections -
<struct name="velocityMagScalarFieldTop">
   <param name="Type">lucScalarFieldCrossSection</param>
   <param name="FieldVariable">VelocityMagnitudeField</param>
   <param name="ColourMap">velocityMagColourMap</param>
   <param name="crossSectionY">y=0.8</param>
</struct>
<struct name="velocityArrowsTop">
   <param name="Type">lucVectorArrowCrossSection</param>
   <param name="VectorVariable">VelocityField</param>
   <param name="colour">white</param>
   <param name="crossSection">y=85%</param>
   <param name="lengthScale">0.5</param>
   <param name="resolutionA">elementResI</param>
   <param name="resolutionB">elementResK</param>
</struct>
```

```
<!-- For a side view showing materials, velocity arrows and a strain rate invariant cross-section
<struct name="materialParticleDots">
   <param name="Type">lucSwarmViewer</param>
   <param name="Swarm">materialSwarm</param>
   <param name="pointSize">3.0</param>
   <param name="pointSmoothing">true</param>
   <param name="ColourVariable">materialSwarm-MaterialIndex</param>
   <param name="ColourMap">materialColourMap</param>
   <param name="positionRange">true</param>
   <param name="minPositionZ">2.3</param>
   <param name="maxPositionZ">2.5</param>
</struct>

<struct name="velocityArrowsSide">
   <param name="Type">lucVectorArrowCrossSection</param>
   <param name="VectorVariable">VelocityField</param>
   <param name="colour">Bisque</param>
   <param name="crossSection">z=20%</param>
   <param name="resolutionA">elementResI</param>
   <param name="resolutionB">elementResJ</param>
</struct>

<struct name="strainRateInvSide">
   <param name="Type">lucScalarFieldCrossSection</param>
   <param name="FieldVariable">StrainRateInvariantField</param>
   <param name="ColourMap">strainRateColourMap</param>
   <param name="crossSection">z=20%</param>
</struct>
```

6.10.2 Texture Map

A texture mapping draws an image in a rendered scene, using a .ppm, .png or .tga file. To insert an image into the domain, a *lucTextureMap* component must be provided.

For example:

```
<struct name="sampleImage">
   <param name="Type">lucTextureMap</param>
   <param name="image">SampleImage.png</param>
   <param name="bottomLeftX">0.0</param>
   <param name="bottomLeftY">1.0</param>
```

```
    <param name="bottomLeftZ">4.0</param>
    <param name="bottomRightX">4.0</param>
    <param name="bottomRightY">1.0</param>
    <param name="bottomRightZ">4.0</param>
    <param name="topRightX">4.0</param>
    <param name="topRightY">1.0</param>
    <param name="topRightZ">0.0</param>
    <param name="topLeftX">0.0</param>
    <param name="topLeftY">1.0</param>
    <param name="topLeftZ">0.0</param>
</struct>
```

Where *SampleImage.png* is the filename of the image to be used.

6.11 Analysis Component Parameters

Listed here are the base components of the gLucifer Analysis module with the parameters that can be set to control their behaviour.

6.11.1 Window

Create one or more as required. Each contains a visualisation: a set of views and objects that will be output as a single image or shown in a single window when loaded using the viewer. A *Window* requires a list of *Viewport* (page 84)s and a *Database* (page 83). It provides visualisation data to the *Database* to be collated and written to disk for later viewing or output.

- **width** Integer, sets the width of the window in pixels.
- **height** Integer, sets the height of the window in pixels.
- **backgroundColour** Colour, the background colour of the window.
- **useModelBounds** True/False, set to False to prevent the window using the default model size to set the camera, useful for Plot objects.
- **disable** True/False, set to True to prevent this window producing output.
- **Database** Name of the database component, if not provided a default is created.
- **Viewport** A list of viewport components to display in the window. (see *Windows and Viewports* (page 68))

6.11.2 Database

Only one required (but more can be used). Manages the parallel output of visualisation data for all windows and provides some options for controlling the output. Writes the 3D graphics data provided by the analysis components to an SQLite database file on disk for later viewing. Runs the gLucifer renderer for image output automatically when requested.

- **deleteAfter** deletes records from the database after N timesteps have passed, useful if you want to reduce disk usage and don't care about re-visualising the data from the database later.
- **writeimage** calls the renderer to write images after every output timestep, disable if you will be using the database for visualisation later and don't want to waste CPU cycles rendering on the fly.
- **splitTransactions** default is off, this writes all output from each timestep in a single database transaction which is faster but requires more memory. Enable this if memory usage is a problem, will cause gLucifer to write objects out in separate transactions to the database as soon as they become available.
- **transparent** image output will have transparent background (only works if libPNG is available).

- **compressed** zlib compression of database geometry records, very slight overhead to create much smaller database files.
- **singleFile** by default all output is to a single database file, if set to False output will be split into separate files per timestep: filename.gldb + filenameXXXXX.gldb where XXXXX is the current timestep.
- **filename** base filename of database output, default is "*gLucifer*", produces outputdir/gLucifer.gldb.
- **vfs** specify an alternate virtual file system for SQLite3 to control file-locking, eg: "unix-dotfile" (may be necessary on some nfs systems).
- **timeUnits** units to use when writing time step data, eg: "s" for seconds, default is dimensionless (no units).

6.11.3 Viewport

Define as many as required and include them in the *Window* (page 83) viewport lists. A *Viewport* requires a list of *Drawing Object* (page 85)s and a *Camera* (page 84). When a display update is required it will request all its *DrawingObjects* to produce their output.

- **title** text, sets a title to be displayed at the top of the viewport.
- **axis** True/False, if true an axis will be drawn in the viewport.
- **axisLength** Number, sets the length of the axis arrows as a ratio to the viewport size [0,1].
- **antialias** True/False, set to False to disable anti-aliasing.
- **rulers** True/False, draw rulers along model boundaries.
- **timestep** True/False, display a time step printout at the edge of the viewport.
- **border** Integer, 0 to disable border, 1 to enable, 2 for a filled frame.
- **borderColour** Colour, colour to draw the border in.
- **margin** Integer, margin in pixels to leave at the viewport edge when automatically zooming the camera.
- **nearClipPlane** Number, override the OpenGL near clip position.
- **farClipPlane** Number, override the OpenGL far clip position.
- **scaleX** Number, scaling factor to apply to all geometry in X axis (width).
- **scaleY** Number, scaling factor to apply to all geometry in Y axis (height).
- **scaleZ** Number, scaling factor to apply to all geometry in Z axis (depth).
- **disable** True/False, set to True to prevent this viewport producing output.
- **Camera** Name of the camera component to use.
- **DrawingObject** A list of drawing object components to display in the viewport. (see *Windows and Viewports* (page 68))

6.11.4 Camera

Create one for each different viewpoint to be applied on the scene. Include it in the *Viewport* (page 84) definition to apply this camera view to a set of *DrawingObjects* (see *Camera Adjustment* (page 72)).

- **CentreFieldVariable** Name of a field variable used to define region to set the camera focal point automatically.
- **CoordinateSystem** LeftHanded/RightHanded, set to LeftHanded to flip the coordinate system to have the Z-axis pointing into the screen.
- **rotate[X/Y/Z]** Number, rotation in degrees about axis.
- **translate[X/Y/Z]** Number, translate the model view in each axis.

- **focalPoint[X/Y/Z]** Number, override the point the camera is looking at.
- **aperture** Number, override the camera aperture in degrees, default 45.
- **autoZoomTimestep** Integer, when to apply auto zoom (set the camera so the whole model fits within the viewport), set to -1 to disable, 0 for once on setup, or N for every Nth timestep. Default is 0.

6.11.5 Drawing Object

Parent type for all visual elements to be displayed from the available data. Include in viewport's *DrawingObject* lists. Executed by viewports as required. When first called each timestep, the data is re-created from the new results (see *Drawing Objects* (page 72)).

Global Parameters: (these may be ingnored or interpreted differently by some objects)

- **lit** True/False, set to False to disable lighting of this object.
- **lineWidth** Integer, set the width of lines drawn by this object.
- **colour** Colour, base colour to draw the object in.
- **ColourMap** Name of the colourmap component to use when drawing the object.
- **opacity** Number [0,1], set the transparency level, 1.0 is fully opaque, 0 is invisible.
- **wireframe** True/False, set to True to draw object surfaces as wireframes instead of filled polygons.
- **disable** True/False, set to True to prevent this object producing output.

6.11.6 Colour Map

Required by some *Drawing Object* (page 85)s, and many may be defined or the same one used for various *DrawingObjects*. It maps a colour range to the values of a variable (field or swarm).

- **dynamicRange** True/False, set to False to manually set the minimum/maximum of the colour range, if True the global minimum and maximum of the variable mapped will be used.
- **minimum** Number, set the minimum value to map to the start of the colour range.
- **maximum** Number, set the maximum value to map to the end of the colour range.
- **centreValue** Number, if set and dynamic range is True, this value will be at the centre of the map, the minimum or maximum will be adjusted out in order to keep the scale linear.
- **discrete** True/False, set to True to disable colour interpolation, colours will not be blended but plotted as exact discrete values.
- **logScale** True/False, set to True to use a logarithmic scale on the map.
- **colours** Colour List, list of colours separated by blank space, commas or semi-colons. Each colour can be one of the names listed here http://en.wikipedia.org/wiki/X11_color_names or a hexidecimal RGB value in the format #RRGGBB. An alpha (opacity) value can be specified with a colon, #RRGGBB:A, eg: #AAFF00:0.5. If a colour value is immediately preceded by a numerical value in brackets that colour will be locked to that value. Eg: (990)red, red will represent the value 990 in the map and will be positioned accordingly. Such values need to be selected sensibly, in ascending or descending order.

6.12 Interactive visualisation

The database output file generated from the simulation can be visualised interactively using the gLucifer viewer. This is particularly useful for 3D model runs when a set of 2D image views of the output is too limiting.

The interactive viewer is built with Underworld by default if the required visualisation dependencies (see *Additional Software* (page 7)) are available. The *gLucifer* viewer executable is found together with the *Underworld*

executable, usually in *$PATHTOUNDERWORLD/build/bin*. If not, pre-built binary installs of the gLucifer Viewer for Ubuntu/Mac/Windows can be downloaded from the project website.

The database output can be visualised by simply passing its filename into the *gLucifer* executable.

For example:

```
build/bin/gLucifer output/gLucifer.gldb
```

A window showing the first timestep should appear. The mouse and keyboard can be used to interact with the model. The user interface to set viewing parameters and other actions is keyboard driven, pressing the *F1* key will display a list of available interaction commands in the terminal.

Note: On Macs, holding the apple key with the left mouse button can be used if the mouse has no right button.

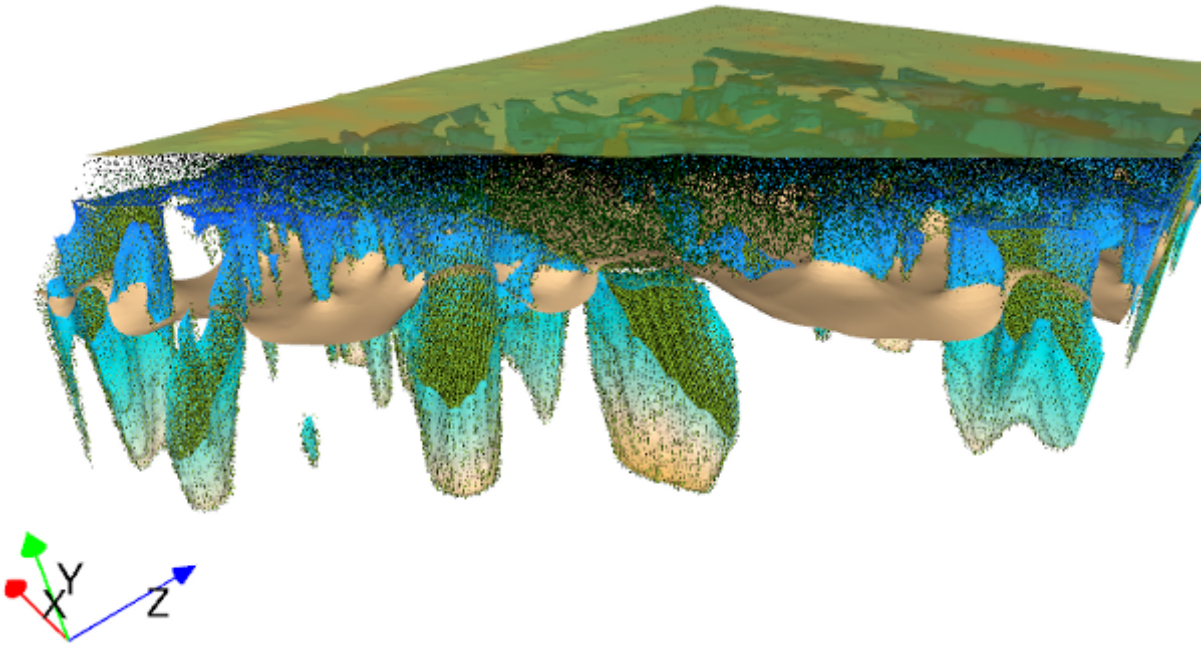


Figure 6.1: A 3D Model loaded in the gLucifer Interactive Viewer

6.12.1 Command Line Options

- **-v** verbose: print debugging output showing details of the viewer processing
- **-o** output: echo entered mouse and keyboard commands to standard output, allowing capture to a script
- **-i** input: read commands from standard input, allows loading a script
- **-x** output width: set a fixed image width for writing output images, the window will be set to this width when writing images, maintaining the original aspect ratio
- **-a** alpha: override the global opacity setting for all objects [0-255]
- **-d** request SDL window, if available SDL will be used to create the viewer window
- **-g** request GLUT window, if available GLUT will be used to create the viewer window
- **-h** hide window, don't display the window
- **-s** request stereo, if available a stereo context will be created allowing 3D stereo viewing (must also be turned on in viewer with the "" key)
- **-f** full screen window, open the viewer in full screen mode

- **-w** write images, writes all selected timesteps as images then quits
- **-W** write images to the database location, as above but saves to the directory containing the database instead of the current directory
- **-m** write movie, writes all selected timesteps to a video file then quits
- **-t** transparent png, png images will have a transparent background
- **-#** where # is a number, the first time this occurs will set the initial timestep, the second time will set the final timestep for batch writing of images and video.

The following switches stop the viewer loading geometry of the specified type from the database:

- **-B** disable label text output
- **-P** disable point output
- **-S** disable surface output
- **-U** disable isosurface output
- **-V** disable vector output
- **-T** disable tracer output
- **-L** disable line output
- **-H** disable shape output

6.12.2 Viewer Controls

Mouse controls

- **Left button (click & drag)** Rotate about the X & Y axes
- **Right button (click & drag)** Pan (left/right/up/down)
- **Middle button (click & drag)** Rotate about the Z axis
- **Scroll wheel** Zoom in and out.
- **[Shift] + Scroll wheel** Move the near clip plane in and out.

Keyboard controls: single key

- **F1** Print help
- **Down** Load next model/window at current time-step if data available
- **Up** Load previous model/window at current time-step if data available
- **Right** Next time-step
- **Left** Previous time-step
- **Page Down** Next viewport if available
- **Page Up** Previous viewport if available
- **Home** View All mode ON/OFF, shows all objects in a single viewport
- **End** ViewPort mode ON/OFF, shows all viewports in window together
- **a** Hide/show axis
- **b** Background colour switch: White/Black
- **B** Background colour switch: Light grey/Dark grey
- **c** Camera info: XML output of current camera parameters
- **d** Draw quad surfaces as triangle strips On/Off

- **f** Frame box mode On/Filled/Off
- **g** Colour map log scales override Default/On/Off
- **j** Experimental: localise colour scales, minimum and maximum calibrated to each object drawn
- **J** Restore original colour scale min & max
- **k** Lock colour scale calibrations to current values On/Off
- **l** Lighting On/Off
- **m** Model bounding box update - resizes based on min & max vertex coords read
- **n** Recalculate surface normals
- **o** Print list of object names with id numbers.
- **r** Reset camera viewpoint
- **s** Take screen-shot and save as png/jpg image file
- **q** or [ESC] Quit program
- **u** Back face Culling On/Off
- **w** Wireframe On/Off
- **‘** Full screen On/Off
- ***** Auto zoom to fit On/Off
- **/** Stereo On/Off
- **“*”** Switch coordinate system Right-handed/Left-handed
- **|** Switch rulers On/Off
- @ Zero camera - set to coord (0,0,0)
- **;** Flat tracer rendering On/Off
- **:** Tracer scaling by time-step On/Off
- **,** Switch to next particle rendering texture
- **+** More particles (reduce sub-sampling)
- **=** Less particles (increase sub-sampling)
- **v** Increase vector size scaling
- **V** Reduce vector size scaling
- **t** Increase tracer size scaling
- **T** Reduce tracer size scaling
- **p** Increase particle size scaling
- **P** Reduce particle size scaling
- **h** Increase shape size scaling
- **H** Reduce shape size scaling

Keyboard controls: key combinations

- **Alt + p** hide/show all particle swarms
- **Alt + v** hide/show all vector arrow fields
- **Alt + t** hide/show all tracer trajectories
- **Alt + s** hide/show all quad surfaces (scalar fields, cross sections etc.)
- **Alt + u** hide/show all triangle surfaces (isosurfaces)

- **Alt + h** hide/show all shapes
- **Alt + i** hide/show all lines

Keyboard controls: type a number and then press:

- **Enter** skip to time-step entered
- **Page Up** skip back entered timesteps relative to current timestep
- **Page Down** skip forward entered timesteps relative to current timestep
- **o** hide/show objects by global id number.
- **p** hide/show particle swarms by id number.
- **v** hide/show vector arrow fields by id
- **t** hide/show tracer trajectories by id
- **s** hide/show quad surfaces by id (scalar fields etc.)
- **u** hide/show triangle surfaces by id (isosurfaces)
- **h** hide/show shapes by id
- **i** hide/show lines by id
- **e** override tracer trajectory steps with entered number
- **E** clears the tracer step override.

CHAPTER

SEVEN

HINTS AND TRICKS

7.1 Introduction

This chapter provides common troubleshooting information for users using Underworld.

7.2 MPI Conflicts

It is important to note that when running Underworld using MPI, the *mpirun* or *mpiexec* executable corresponds to the same MPI library the *Underworld* executable is compiled against. Running *Underworld* with a different MPI library can result in some nasty parallel errors. This issue will also affect CREDO tests that is being ran.

7.2.1 Resolution

On clusters using the *module* system, make sure that the correct modules are being loaded when running the code as well as compiling it. On desktop machines, make sure that the *PATH* environment variable is setup correctly so that *mpirun* will be launched using the correct version of the MPI library used when compiling. It is advised to set the *PATH* environment variable on the user's login script (*~/.bashrc*).

With regards to CREDO, the environment variable *MPI_RUN_COMMAND* can be set to tell the system that the CREDO jobs will be running using a custom *mpirun* or *mpiexec* executables rather than the one specified in the *PATH*.

For example:

```
MPI_RUN_COMMAND="/usr/local/packages/mpich2-1.2.1p1-debug/bin/"
```

7.3 Underworld and StGermain Executables

There is no difference between the *Underworld* and *StGermain* executables found inside the *bin* directory of the Underworld installation. Technically, Underworld is only a library and does not itself have an executable. StGermain is the underlying framework which provides the executable. A renamed version of the *StGermain* executable (*Underworld*) is provided to allay any confusion of new Underworld users. Using symbolic link would be a more natural approach, but there were issues with this approach on certain platforms.

7.4 Estimate Simulation Runtime

The *FrequentOutput.dat* file which is generated by the simulation and resides in the output directory together with other output files contains the *CPU_Time* column. This column tells how long a specific timestep took to finish.

7.5 Simulation's *dt* correspondence to non-dimensional time

While time *dt* is non-dimensional, it is not exactly equal to one non-dimensional time unit. *dt* is computed automatically by Underworld to keep some algorithms inside its stability field. It changes from step to step. The evolution of "dt" can be viewed in the *FrequentOutput.dat* file. If *FrequentOutput.dat* is not generated during the simulation run, the *StgFEM_FrequentOutput* plugin must be included in the XML input file.

For example:

```
<list name="plugins" mergeType="merge">
   <struct>
      <param name="Type">StgFEM_FrequentOutput</param>
      <param name="Context">context</param>
   </struct>
</list>
```

The timestep information is written in the *timeInfo.xxxxx.dat* file. For every checkpointed step, the total non-dimensional time since the models start is acquired.

7.6 Getting *time* and Peak memory usage information

The command:

```
/usr/bin/time -f "user\t\t\t%U \nsystem\t\t\t%S \nelapsed \t\t%E \nMajor
page faults\t%F \nMinor page faults\t%R \nSystem Page size\t%Zbytes"
StGermain
```

will dump out more *time* information, in particular multiplying *Minor page faults* by *System Page size* will give a fairly accurate proxy to the peak memory usage during an Underworld run.

Also, the plugin *StgFEM_PeakMemory* can be used to report memory reports from Petsc.

For example:

```
<list name="plugins" mergeType="merge">
   <struct>
      <param name="Type">StgFEM_PeakMemory</param>
      <param name="Context">context</param>
   </struct>
</list>
```

This will output to *FrequentOutput.dat* information pertaining to StGermain's internal memory usage (*StgPeakMem*), PETSc's maximum malloc'ed memory (*PetscMem*) and the total process memory used (*ProgMem*). in Megabytes.

CHAPTER

EIGHT

MORE HELP

8.1 Introduction

This chapter provides links to additional references useful for Underworld users and developers.

8.2 Links

- Underworld Project - For descriptions of Underworld models, downloading Underworld, documentation options and contact information on the developers.
- Underworld Forums - Forum for Underworld-related discussions and information.

BIBLIOGRAPHY

[MoresiQuenetteLemiale2007] Moresi L., Quenette S., Lemiale V., Meriaux C., Appelbe B., and Mühlhaus H. B. 2007. Computational approaches to studying non-linear dynamics of the crust and mantle. Physics of the Earth and Planetary Interiors 163, 69-82.

[MoresiDufourMuhlhaus2003] L.N. Moresi, F. Dufour and H.B. Mühlhaus, A Lagrangian integration point finite element method for large deformation modelling of viscoelastic geomaterials, J. Comp. Phys., Volume 184, pp. 476-497, 2003.

[VanKeken1997] P.E. van Keken et al. A comparison of methods for the modelling of thermochemical convection. Journal of Geophysical Research 102:B10, 22477-22495, 1997

[MoresiMuhlhaus2006] Moresi, L. and Mühlhaus, H.-B. Anisotropic viscous models of large-deformation Mohr-Coulomb failure. Philosophical Magazine, Volume 86, Numbers 21-22, -22/21 July 1 August 2006, pp. 3287-3305