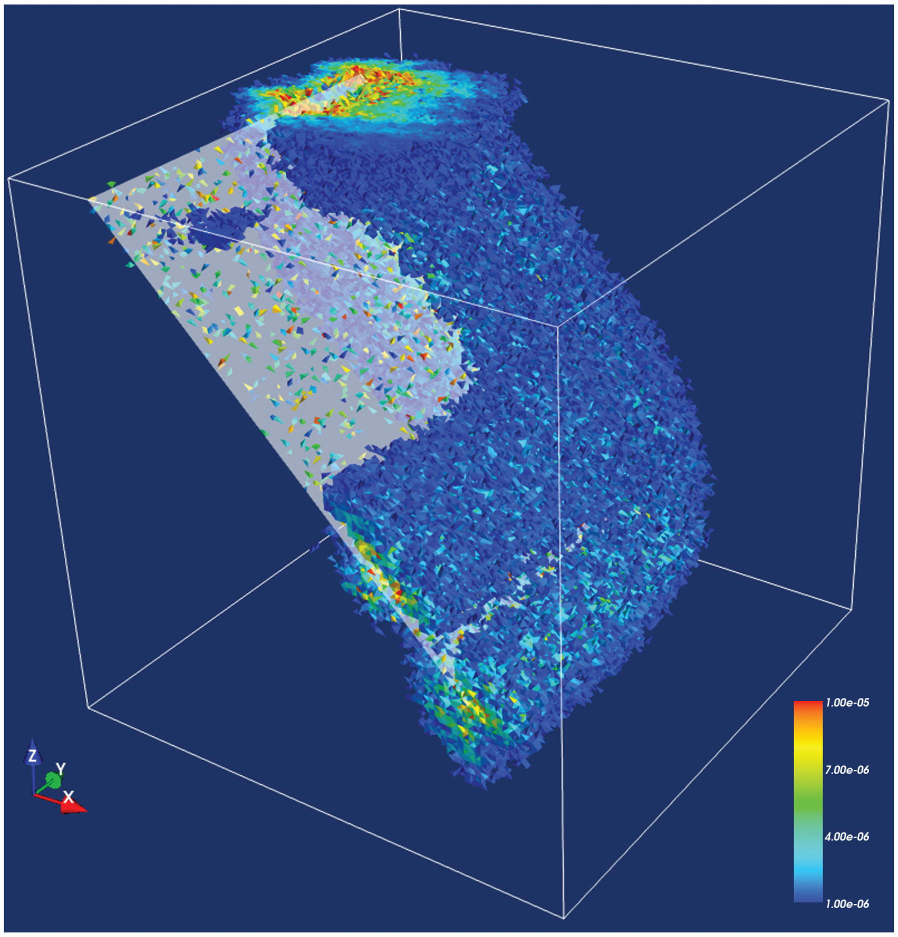
COMPUTATIONAL INFRASTRUCTURE FOR GEODYNAMICS (CIG)

Cigma

User Manual
Version 1.0.0



Luis Armendariz
Susan Kientz

www.geodynamics.org

Cigma

Version 1.0.0

April 30, 2009

About the cover: The cover image depicts Cigma's 3D comparison between two different PyLith solutions to the reverse-slip benchmark problem. PyLith is a finite element code for the solution of dynamic and quasi-static tectonic deformation problems.

This benchmark problem solves for the viscoelastic (Maxwell) relaxation of stresses from a single, finite, reverse-slip earthquake in 3D without gravity, with imposed displacement boundary conditions on a cube with sides of 24 km in length. On the boundary, we impose displacements obtained via the analytic elastic solutions. Additionally, symmetry boundary conditions are imposed on the $y = 0$ plane, so the solution is equivalent to that for a domain with a 48 km length in the y direction. We are visualizing the error in the displacement field between the 500m linear hexahedral and 250m tetrahedral solutions.

Contents

List of Figures

Chapter 1

Introduction

1.1 About Cigma

The CIG Model Analyzer (Cigma) is a program designed to compare general numerical models. In particular, Cigma will calculate the L_2-norm of the difference between finite element models, resulting in approximate global and local error metrics that may be used for both code verification and benchmarking purposes.

CIG developed Cigma in response to demand from the short-term tectonics community for an automated tool that can perform rigorous error analysis on their finite element codes. For example, on the cover of this manual is a 3D comparison for two resolutions of PyLith's reverse-slip benchmark problem (`geodynamics.org/cig/workinggroups/short/workarea/benchmarks/benchmark-rs-nog/description-rs-nog`).

By facilitating the process of data comparisons, independent of discretization, Cigma will better assist the Geodynamics community in their benchmarking activities. We also seek to increase confidence in the output of the CIG software by providing a standardized testing process. Such a process is vital to detecting critical errors during code development. For this purpose, Cigma is designed to be easily extensible.

1.2 Citation

Computational Infrastructure for Geodynamics (CIG) is making this source code available to you in the hope that the software will enhance your research in geophysics. This is a brand-new code and at present no papers are published or in press. Please cite this manual as follows:

- Armendariz, L., and S. Kientz. *Cigma User Manual.* Pasadena, CA: Computational Infrastructure of Geodynamics, 2008. URL: geodynamics.org/cig/software/cs/cigma/cigma.pdf

CIG requests that in your oral presentations and in your papers that you indicate your use of this code and acknowledge the author of the code and CIG (`geodynamics.org`).

1.3 Support

Cigma development is based upon work supported by the National Science Foundation under Grant No. EAR-0406751. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. The code is being released under the GNU General Public License v2.

Chapter 2

Installation and Getting Help

2.1 Getting Help and Reporting Bugs

For help, send an e-mail to the CIG Computational Science Mailing List (`cig-cs@geodynamics.org`). You can subscribe to the `cig-cs` mailing list and view archived discussions at the CIG Mail Lists web page (`geodynamics.org/cig/lists`). If you encounter any bugs or have problems installing Cigma, please submit a report to the CIG Bug Tracker (`geodynamics.org/bugs`).

2.2 Installing from Source

In this section we discuss how to install each of the software libraries necessary for building Cigma. We recommend that you obtain binaries for these dependencies from your package manager of choice, or when available from the corresponding websites. When using a package manager, you will have to make sure to install any associated development packages, as these tend to be distributed separately from the library packages themselves.

2.2.1 Quick Summary

After installing the necessary dependencies described below, download the source package from the CIG Cigma web page (`geodynamics.org/cig/software/packages/cs/cigma`). You will need the GNU C++ compiler for this step. Unpack the source tar file and issue the following commands

```
$ tar xvfz cigma-1.0.0.tar.gz
$ cd cigma-1.0.0
$ ./configure \
    --with-boost=$BOOST_PREFIX       \
    --with-hdf5=$HDF5_PREFIX         \
    --with-vtk=$VTK_PREFIX           \
    --with-vtk-version=$VTK_VERSION  \
    --with-netcdf=$NETCDF_PREFIX     \
    --with-cppunit-prefix=$CPPUNIT_PREFIX
$ make
$ make install
```

In general, the installation prefixes given to the configure options should be used if you have installed the corresponding library in a custom location, but may otherwise be omitted if the prefix is one of the default system directories, such as `/usr` or `/usr/local`. For demonstration purposes, the instructions in the next few sections use a subdirectory of `$HOME/opt` as the installation prefix for each of the dependencies.

The only required options in this step are the ones for the Boost and HDF5 libraries. If you wish to enable VTK library support, you will also have to select a specific version number by providing the configure flag `--with-vtk-version`, whose value defaults to "`5.0`". The NetCDF option is disabled by default, but if you wish to use ExodusII mesh files, you can provide an installation prefix to the corresponding configure option. Lastly, the CppUnit configure option will allow you to use `make check` to compile and run a basic set of unit tests which verify that Cigma is giving internally consistent results.

After all these steps are complete, you should be able to run the `cigma` binary and proceed to compare your data files.

2.2.2 Boost library

The Boost C++ libraries are a collection of peer-reviewed and open source libraries that extend the functionality of C++. Cigma uses Boost for its smart pointer facilities, lexical conversions, as well as for parsing command-line options, and for providing cross-platform filesystem operations. If Boost is not available for your platform as a binary package that can be readily installed, you may choose to install Boost with only the components that Cigma will need,

```
$ tar xvfz boost_1_37_0.tar.gz
$ cd boost_1_37_0
$ ./configure --prefix=$HOME/opt/boost \
              --with-libraries=system,filesystem,program_options
$ make
$ make install
```

Just as a reference, the full list of Boost components can be obtained with the command

```
$ ./configure --show-libraries
```

Enabling all components will certainly increase the compilation time of the Boost code, since some of those components will require compilation.

2.2.3 HDF5 library

The Hierarchical Data Format is a library and multi-object file format for the transfer of numerical data between computers. Since Cigma depends on the C++ API of the HDF5 library, it is important to enable C++ support when running the configure script below. We recommend you install the library version 1.8 or higher, due to the improved metadata API in those versions. The latest source can be obtained from The HDF Group (`hdfgroup.org/HDF5`), and it uses the following sequence of commands to build the library.

```
$ tar xvfz hdf5-1.8.2.tar.gz
$ cd hdf5-1.8.2
$ ./configure --prefix=$HOME/opt/hdf5-1.8.2 \
              --with-zlib=$ZLIB_PREFIX      \
              --enable-cxx
$ make
$ make install
```

In the rare case that the zlib compression library is not already available on your system, you must install it before running the installation procedure given above,

```
$ tar xvfz zlib-1.2.3.tar.gz
$ cd zlib-1.2.3
$ ./configure --prefix=$HOME/opt/hdf5-1.8.2
$ make
$ make install
```

In practice, you only need to do this if your system is missing the zlib development headers and you are unable to install them otherwise.

Alternatively, compiled binaries for the HDF5 library can be obtained at (`hdfgroup.org/HDF5/release/obtain5.html`).

2.2.4 VTK library (optional)

The Visualization Toolkit (VTK) library is a popular open source graphics library for scientific visualizations. Cigma will need to be configured with the VTK library if you plan to directly specify your functions by using the VTK file format. The source for the VTK library is available from Kitware, Inc. (`www.vtk.org/get-software.php`). You will also need the CMake build environment, available from CMake (`cmake.org`). After downloading the source package for the VTK library, you can issue the following commands

```
$ tar xvfz vtk-5.2.0.tar.gz
$ cd VTK
$ mkdir build
$ cd build
$ ccmake ..
$ make
$ make install
```

The installation prefix and other settings can be changed during the `ccmake` step above. By default, the installation prefix will point to the `/usr/local` directory, but you may want to change it to a location like `$HOME/opt/vtk-5.2` in case you would like to manage multiple versions of the VTK library in the same system.

2.2.5 NetCDF library (optional)

NetCDF (Network Common Data Form) is a set of software libraries and machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data. The source for NetCDF can be obtained from Unidata (`www.unidata.ucar.edu/software/netcdf/`).

```
$ tar xvfz netcdf-4.0
$ cd netcdf-4.0
$ ./configure --prefix=$HOME/opt/netcdf-4.0    \
              --with-hdf5=$HOME/opt/hdf5-1.8.2 \
              --enable-netcdf-4

$ make
$ make install
```

Using this library is optional, so it is not automatically detected at configure time. NetCDF support in Cigma is still experimental, and is currently only intended for reading ExodusII mesh files created through the CUBIT mesh generator. Only the first element block will be used, so that the corresponding Cigma mesh object will consist of a single element type.

2.2.6 CppUnit library (optional)

CppUnit is a C++ unit testing framework used by Cigma for automatically running various internal consistency checks during every build. You will need this library if you want to modify the Cigma source and want to make certain guarantees about your additions to the code. After obtaining the latest source release, you may build CppUnit by using the following sequence of steps:

```
$ tar xvfz cppunit-1.21.1.tar.gz
$ cd cppunit-1.21.1
$ ./configure --prefix=$HOME/opt/cppunit-1.21.1
$ make
$ make install
```

CppUnit is available for download from its SourceForge project site (`sourceforge.net/projects/cppunit/`).

2.2.7 Compiling Cigma

After carefully following the instructions above, we should be able to install Cigma using this set of commands:

```
$ tar xvfz cigma-1.0.0.tar.gz
$ cd cigma-1.0.0
$ ./configure --prefix=$HOME/opt/cigma-1.0.0 \
      --with-boost=$HOME/opt/boost \
      --with-hdf5=$HOME/opt/hdf5-1.8.2 \
      --with-vtk=$HOME/opt/vtk-5.2.0 \
      --with-vtk-version=5.2 \
      --with-cppunit-prefix=$HOME/opt/cppunit-1.21.1
$ make
$ make check
$ make install
```

These instructions have used custom installation locations, so it is also necessary to update the `PATH` and `LD_LIBRARY_PATH` environment variables. Preferably you should update these environment variables in your shell login file, but you can also update them on the command line itself. Here we show the appropriate set of commands for a bash shell session.

```
$ PATH=''$PATH:$HOME/opt/cigma-1.0.0/bin''
$ PATH=''$PATH:$HOME/opt/hdf5-1.8.2/bin''
$ export PATH
$ LD_LIBRARY_PATH=''$HOME/opt/boost/lib:$LD_LIBRARY_PATH''
$ LD_LIBRARY_PATH=''$HOME/opt/hdf5-1.8.2/lib:$LD_LIBRARY_PATH''
$ LD_LIBRARY_PATH=''$HOME/opt/vtk-5.2.0/lib:$LD_LIBRARY_PATH''
$ export LD_LIBRARY_PATH
```

On Mac OS X, you will also need to update the `DYLD_LIBRARY_PATH` environment variable,

```
$ DYLD_LIBRARY_PATH=''$HOME/opt/boost/lib:$DYLD_LIBRARY_PATH''
$ DYLD_LIBRARY_PATH=''$HOME/opt/hdf5-1.8.2/lib:$DYLD_LIBRARY_PATH''
$ DYLD_LIBRARY_PATH=''$HOME/opt/vtk-5.2.0/lib:$DYLD_LIBRARY_PATH''
$ export DYLD_LIBRARY_PATH
```

No environment variables need to be set if all installation prefixes use a system directory.

2.3 Installing from the Software Repository

The Cigma source code is available via the CIG Subversion Repository (`geodynamics.org/cig/software/Repository`). For this section, you will need a Subversion client, as well as the GNU tools Autoconf, Automake, and Libtool. To check if you have a Subversion client installed, type `svn` and look for a usage message

```
$ svn
Type 'svn help' for usage.
```

To check for the presence of the GNU Autotools, use the following commands:

```
$ autoconf --version
$ automake --version
$ libtoolize --version
```

Using the source repository directly is recommended for users who want to extend Cigma.

2.3.1 Downloading the Cigma Source

You may check out the latest version of Cigma by using the `svn checkout` command:

```
$ svn checkout http://geodynamics.org/svn/cig/cs/cigma/trunk cigma
```

This will create the local directory `cigma` and fill it with the latest Cigma source from the CIG software repository. This new directory is called a *working copy*. To merge the latest changes into your working copy, use the `svn update` command:

```
$ cd cigma
$ svn update
```

This will preserve any local changes you have made to your working copy.

2.3.2 Using the GNU Build System

Once you have obtained a working copy of the Cigma source, you will need to use the GNU Autotools to generate the appropriate build files. This can be accomplished by running the command `autoreconf -fi`:

```
$ cd cigma
$ autoreconf -fi
```

The `autoreconf` tool generates the `configure` script from the `configure.ac` file. It also runs Automake to generate `Makefile.in` from `Makefile.am` in each source directory. After running `autoreconf`, you may configure, build, and install Cigma as described in Section 2.2.

Chapter 3

Error Analysis

3.1 Error Analysis

When solving differential equations representing physical systems of interest, we are often able to obtain a family of solutions by applying a variety of solution techniques. Sometimes an analytic method can be found, but most of the time we end up resorting to a numerical algorithm, such as the finite element method. Assessing the quality of these solutions is an important task, so we would like to develop a quantitative measure for indicating just how close our solutions approach the exact answer.

The simplest possible quantitative measure of the difference between two distinct functions consists of taking the pointwise difference at a common set of points. While no finite sample of points can perfectly represent a continuum of values, valuable information can be inferred from the statistics of the resulting set of differences. However, the functions we want to compare may not be defined at a common set of points. Unless we are able to interpolate the two functions on an intermediate set of points, this simple pointwise measure becomes inapplicable.

A very useful distance measure we can use is the L_2 norm, defined by the following integral

$$\varepsilon = ||u - v||_{L_2} = \sqrt{\int_\Omega ||u(\vec{x}) - v(\vec{x})||^2 d\vec{x}} \tag{3.1}$$

where u and v are the two functions defined on a global coordinate system. This gives us a single global estimate ε representing the distance between the two functions $u(\vec{x})$ and $v(\vec{x})$. Alternatively, you may think of this as the size, or norm, of the ***error field*** $\rho(\vec{x}) = u(\vec{x}) - v(\vec{x})$. This measure is useful because it is well known that solutions obtained with the finite element method converge only in a weak sense, as an average over a geometric region. In the rest of this chapter we will discuss the details involved in evaluating the above integral.

Another quantitative measure, which we won't discuss in this version of Cigma, is the energy norm. This norm is typically employed in *a posteriori* error analysis and is problem dependent. In general, the error field used under this norm is defined in terms of the linearized equation from which the solution was obtained, instead of directly subtracting the corresponding function values.

3.2 Functions

The functions that we discuss in this manual are assumed to be defined in a common region of space which we denote by Ω. Elements of Ω are written as $\vec{x}$, and the number of components in the corresponding value of $f(\vec{x})$, is said to be the ***rank*** of the function f. Thus, *scalar functions* have a rank of 1, while *vector functions* usually have a rank of either 2 or 3.

3.3 Integral Approximations

In order to evaluate the integral for our L_2 norm, we will need to define an integration mesh which partitions the common domain Ω on which our two functions are defined. We can then use a numerical approximation, or ***integration rule***, on each of the discrete cell elements Ω_i of our partition. Each of these cell integrals of the continuous error norm $||\rho(\vec{x})||^2$ can then be replaced by a weighed sum evaluated at a finite set of points. This integral will be valid up to a certain accuracy. In later chapters, we will discuss in more detail how to choose the location and values for the integration points and weights which give optimal accuracy.

In general, an integration rule with Q points and weights that allows us to integrate the scalar function $F(\vec{x})$ is given by the equation

$$\int_{\Omega_i} F(\vec{x})\ d\vec{x} \approx \sum_{q=1}^{Q} F(\vec{X}_q) W_q \tag{3.2}$$

where the integration points $\vec{X}_q$ and integration weights W_q depend on the integration region Ω_i. One possibility is to pre-calculate these points and weights explicitly on a specific discretization. We will generally want to define a reference cell $\hat{\Omega}$ which acts as a template for all cells with the same geometric shape. We can achieve this by defining a ***reference map*** $\chi_i : \hat{\Omega} \to \Omega_i$ that describes how points in a specific cell Ω_i originate from the reference cell $\hat{\Omega}$. In this sense, the reference cell $\hat{\Omega}$ is said to define a ***local coordinate system*** for the i-th cell Ω_i.

Note that a given discretization may contain cells of different shapes, which would require us to define an appropriate number of reference cells for each type of cell. However, we may restrict the discussions in this manual to discretizations consisting of a single geometric shape without any loss of generality.

The advantage of this approach is readily apparent when one realizes that the integration points and weights can be calculated once and for all on the reference cell, and then reused for the other cells through the application of the corresponding reference map. In other words,

$$\vec{X}_q = \chi_i(\vec{\xi}_q) \tag{3.3}$$

$$W_q = w_q J_i(\vec{\xi}_q) \tag{3.4}$$

where $\vec{\xi}_q$ and w_q are the optimal integration points and weights for the integration rule defined on the reference domain $\hat{\Omega}$. Here, the factor $J_i(\vec{\xi}) = \det \left|\frac{d\chi_i}{d\vec{\xi}}\right|$ is the Jacobian determinant of the transformation χ_i. Recall that the index i corresponds to the i-th cell Ω_i, and the index q ranges over the usual $q = 1, \ldots, Q$.

3.4 Interpolation Functions

Our two functions $u(\vec{x})$ and $v(\vec{x})$ are given in terms of the point $\vec{x} \in \Omega$, and are said to be defined on a ***global coordinate system***. Based on our discussion in Section 3.1, we see that we only need to know the values of u and v at a finite set of global points in order to calculate an approximation to the L_2 norm of the error field $\rho = u - v$.

Specifically, given a function $f(\vec{x})$ we must be able to calculate its values at exactly those integration points. If we know a formula or algorithm for f, then our task is easy. Alternatively, these values can be given explicitly as a finite list. In all other cases, we will need an interpolation scheme that allows us to calculate f on any intermediate points. This ability is also important because we may want to increase the accuracy of our norm by using more integration points, and thus we will need to be able to evaluate our function f at the new points.

In general, an interpolation scheme involves a set of known functions $\phi_j(\vec{x})$ that we can compute anywhere and a set of parameters c_j that define how these known functions are combined. Usually, the relation between the parameters and the known functions is a linear combination,

$$f(\vec{x}) = d_1\phi_1(\vec{x}) + d_2\phi_2(\vec{x}) + \cdots + d_m\phi_m(\vec{x}) \tag{3.5}$$

where the parameters d_j, with $j = 1, \ldots, m$, are also sometimes referred to as the ***degrees of freedom*** of the function f. As described in Chapter 4, the functions ϕ_j, also known as ***shape functions***, must satisfy a number of conditions. Because the choice of shape functions affects how well the true solution is represented, the convergence of the numerical method used to obtain the solution, the optimal choice of shape functions is very problem dependent.

If we know the shape functions ϕ_j directly in terms of the global points $\vec{x} \in \Omega$, they are said to form a ***global interpolation scheme***, and we may use Eq. 3.3 directly to find the values of $f(\vec{x})$, and we may refer to the ϕ_j as *global shape functions.* Note that since we are working in the global coordinate system, we don't need the discretization of the domain Ω. An example of global shape functions would be the spherical harmonic functions.

Perhaps a more typical case is when $f(\vec{x})$ is defined piecewise, on each discretization element Ω_i. Because these cells partition the original domain by definition, a given point in our domain will be found in one and only one cell, which will have a local definition. That is, for a particular $\vec{x} \in \Omega$ we will be able to find a unique cell Ω_e, for some index e. We can refer to this as a ***local interpolation scheme***.

Refer to the Appendix for more details on the specific interpolation schemes available in Cigma.

3.5 Global Error Measure

In this section we discuss the underlying formula used by Cigma to compute the L_2 norm of the error field ρ.

Suppose the domain Ω is partitioned into an appropriate set of cells $\Omega_1, \Omega_2, \ldots, \Omega_{n_{el}}$. We can then compute the global error ε^2 as a sum over localized cell contributions, $\varepsilon^2 = \sum_e \varepsilon_e^2$, where each cell error ε_e^2 is given by

$$\varepsilon_e^2 \quad = \quad \int_{\Omega_e} ||u(\vec{x}) - v(\vec{x})||^2 d\vec{x} \tag{3.6}$$

In general, we won't be able to integrate each cell error ε_e^2 exactly since either of the functions u and v may have an incompatible representation relative to the finite element space on each domain Ω_e. However, we can still calculate an approximation of each cell error by applying an appropriate quadrature rule with a tolerable truncation error [5].

To obtain an approximation to the integral of a function $F(\vec{x})$ over a cell Ω_e, we simply apply the quadrature rule with weights $w_{e,1}, w_{e,2}, \ldots, w_{e,n_Q}$ and integration points $\vec{x}_{e,1}, \vec{x}_{e,2}, \ldots, \vec{x}_{e,n_Q}$ appropriate for the physical element Ω_e:

$$\int_{\Omega_e} F(\vec{x})\ d\vec{x} = \sum_{q=1}^{n_Q} w_{e,q} F(\vec{x}_{e,q}) \tag{3.7}$$

Applying this quadrature rule directly over the entire physical domain $\Omega = \cup \Omega_e$ gives us

$$\int_{\Omega} F(\vec{x})\ d\vec{x} = \sum_{e=1}^{n_{el}} \sum_{q=1}^{n_Q} w_{e,q} F(\vec{x}_{e,q}) \tag{3.8}$$

For efficiency reasons, it is undesirable in finite element applications to perform calculations in a global coordinate system. To avoid duplication of work, shape function evaluations may be performed once on a reference cell $\hat{\Omega}$ and then transformed back into the corresponding physical cell Ω_e as needed.

To compute integrals of F in a reference coordinate system, we need to apply a change of variables:

$$\int_{\Omega_e} F(\vec{x})\ d\vec{x} = \int_{\hat{\Omega}} F(\vec{x}_e(\vec{\xi})) J_e(\vec{\xi})\ d\vec{\xi} \tag{3.9}$$

where the additional factor $J_e(\vec{\xi}) = \det \left[\frac{d\vec{x}_e}{d\vec{\xi}} \right]$ is the Jacobian determinant of the reference map $\vec{x}_e(\vec{\xi}) : \hat{\Omega} \to \Omega_e$. This map describes how the physical points $\vec{x} \in \Omega_e$ are transformed from the reference points $\vec{\xi} \in \hat{\Omega}$. Put another way, the inverse reference map $\vec{\xi} = \vec{x}_e^{-1}(\vec{x})$ tells us how the physical domain Ω_e maps into the reference cell $\hat{\Omega}$.

At this point, we can assume without loss of generality that every physical cell Ω_e can be derived from a single reference cell $\hat{\Omega}$, so that our quadrature rule becomes simply a set of weights $w_1, \ldots, w_{n_Q}$ and points $\vec{\xi}_1, \ldots, \vec{\xi}_{n_Q}$ over $\hat{\Omega}$. After changing variables, we end up with the final form of the quadrature rule that we can use to integrate the global error field:

$$\int_{\hat{\Omega}} F(\vec{x}_e(\vec{\xi}))J_e(\vec{\xi})\ d\vec{\xi} = \sum_{q=1}^{n_Q} w_q F(\vec{x}_e(\vec{\xi}_q))J_e(\vec{\xi}_q) \tag{3.10}$$

If we let $F(\vec{x}) = ||u(\vec{x}) - v(\vec{x})||^2$ in the above expression, we can find the cell error contribution over Ω_e from

$$\begin{aligned} \varepsilon_e^2 &= \int_{\Omega_e} ||u(\vec{x}) - v(\vec{x})||^2\ d\vec{x} \\ &= \int_{\hat{\Omega}} ||u(\vec{x}_e(\vec{\xi})) - v(\vec{x}_e(\vec{\xi}))||^2 J_e(\vec{\xi})\ d\vec{\xi} \\ &= \sum_{q=1}^{\mathrm{n_Q}} ||u(\vec{x}_e(\vec{\xi}_q)) - v(\vec{x}_e(\vec{\xi}_q))||^2 w_q J_e(\vec{\xi}_q) \end{aligned} \tag{3.11}$$

The global error $\varepsilon = \sqrt{\sum_e \varepsilon_e^2}$ is then approximated by the expression

$$\varepsilon = \sqrt{\sum_{e=1}^{\mathrm{n_{el}}} \sum_{q=1}^{\mathrm{n_Q}} ||u(\vec{x}_e(\vec{\xi}_q)) - v(\vec{x}_e(\vec{\xi}_q))||^2 w_q J_e(\vec{\xi}_q)} \tag{3.12}$$

We use this final form to calculate the global and localized errors on arbitrary discretizations.

3.6 Comparing Global Error Quantities

Since the absolute magnitude of the L_2-norm is typically not important, you may want to normalize it relative to another computable global quantity, to make comparisons easier. One possibility would be to normalize the global error by the norm of the exact solution. For a family of solutions $u_h(\vec{x})$, parametrized by the maximum element size h of the underlying discretization, and an exact solution $u(\vec{x})$, this normalized error is given by

$$\begin{aligned} \varepsilon_{rel} &= \frac{||u - u_h||_{L_2}}{||u||_{L_2}} \\ &= \frac{\int_\Omega ||u(\vec{x}) - u_h(\vec{x})||^2 d\vec{x}}{\int_\Omega ||u(\vec{x})||^2 d\vec{x}} \end{aligned} \tag{3.13}$$

Alternatively, we can normalize the global error using the total volume of the domain

$$\begin{aligned} \varepsilon_{abs} &= \frac{||u - u_h||_{L_2}}{\sqrt{V}} \\ &= \frac{\sqrt{\int_\Omega ||u(\vec{x}) - u_h(\vec{x})||^2 d\vec{x}}}{\sqrt{\int_\Omega d\vec{x}}} \end{aligned} \tag{3.14}$$

which is the normalization that Cigma uses by default.

This normalized error can be interpreted as the average error in the physical quantity being evaluated, so that a value of 0.01 corresponds to a 1% averaged error, in absolute units. Even if the exact solution is not currently known, this normalized error may be used to test the accuracy between two or more numerical solutions, defined on successively refined meshes.

3.7 Convergence Rates

Once we have calculated a family of solutions $u_h(\vec{x})$ on a family of increasingly refined meshes Ω_h, where h is a parameter denoting the size of the largest element in the corresponding mesh Ω_h, we may estimate how fast we are converging to the analytic solution $u(\vec{x})$ by using the standard error estimate $||u - u_h||_p \leq Ch^\alpha$, where C is a constant independent of both h and u. This error bound holds for a wide range of problems and may be used for estimating the convergence rate α. If the analytic solution u is unknown, one may use in its place the highest-accuracy solution available. In that case, the value of h you should use would correspond to the lower-accuracy solution against which you are comparing.

For a single refinement level we have two discretizations Ω_1 and Ω_2, along with the corresponding solutions u_1 and u_2. The convergence rate can then be estimated from the two approximate bounds

$$\varepsilon_1 \sim Ch_1^\alpha \tag{3.15}$$
$$\varepsilon_2 \sim Ch_2^\alpha \tag{3.16}$$

by taking their ratio and solving for α, giving us the equation

$$\alpha \sim \frac{\log(\varepsilon_2/\varepsilon_1)}{\log(h_2/h_1)} \tag{3.17}$$

If solutions are available for several refinement levels Ω_i, we can estimate α by performing linear regression analysis on the equation

$$\log \varepsilon_i = \log C + \alpha \log h_i \tag{3.18}$$

over the transformed variables $X_i = \log h_i$ and $Y_i = \log \varepsilon_i$. The slope of that regression line gives an estimate of α. You can use the SciPy and Matplotlib Python modules to write two functions that can give you precisely that estimate.

Listing 3.1: Python functions for calculating and plotting the best least-squares power law fit.

```
import numpy
import matplotlib

from scipy.optimize import leastsq
from pylab import loglog, title, show


def power_law_fit(X, Y):

    # verify input arrays have same length
    assert len(X) == len(Y)

    # parametric function representing power-law
    power_law = lambda v,x: v[0] * x ** v[1]

    # error function
    error_fn = lambda v,x,y: (power_law(v,x) - y)

    # initial parameter value
    v0 = (1,1)

    # find optimal parameters v
    (v, success) = leastsq(error_fn, v0, args=(X,Y), maxfev=1000)
    print 'Best fit: err =', v[0], '* h ^', v[1]

    return (v, success)
```

```
def plot_power_law(X,Y, **kw):

    (v, success) = power_law_fit(X,Y)

    x = numpy.array(X)
    y = numpy.array(Y)

    # finally, plot points along with best-fit function.
    plot_fn = kw.get('plot', loglog)
    plot_fn(x, y, 'ro')
    plot_fn(x, power_law(v,x))

    title(r'$\varepsilon\ =\ %f\ h^{%f}$' % (v[0],v[1]))

    show()
```

A similar script called `power-plot.py` is provided in the Cigma source code. It expects a single command line argument: a file name which contains the points (h_i, ε_i). Running that script will plot the regression line associated with the points $(\log h_i, \log \varepsilon_i)$.

Chapter 4

Running Cigma

Cigma is primarily designed for calculating error estimates between arbitrary functions. These functions can be defined as mathematical functions or as fields over finite element meshes. The primary operation of Cigma has the basic form

```
$ cigma compare FunctionA FunctionB
```

which executes a global comparison between two generic functions `FunctionA` and `FunctionB`.

The full list of options for Cigma can be obtained by running the `cigma help` command.

```
$ cigma help
Usage: cigma <subcommand> [options] [args]
Type 'cigma help <subcommand>' for help on a specific subcommand.
Type 'cigma --version' to see the program version
Available subcommands:
   compare        Calculate global L2 error, and local differences between two functions
   list           List file contents (fields, dimensions, ...)
   mesh-info      Show information about specified mesh
   function-info  Show list of pre-defined functions
   element-info   Show information about available element types
Cigma is a tool for calculating L2-norm differences between numerical models.
For additional information, visit http://geodynamics.org/
```

In this Chapter, we give a general overview of each of these subcommands, explain the meaning of each of their command line options, and give hypothetical examples of typical usage patterns. We delay discussion of actual examples until the next Chapter, where we rely on publically available data files that can be downloaded from the Cigma software page at (`geodynamics.org/cig/software/packages/cs/cigma/cigma-data-1.0.0.tar.gz`).

4.1 Selecting a Dataset

In general, you will provide data to Cigma as a simple array of values. Because most scientific formats are capable of storing multiple datasets in the same file, you will need a consistent way to select a specific array on a given data file. Therefore, all option arguments to Cigma that expect a dataset can be specified in the form `Filename:Location`. If the *filename* part unambiguously determines the target array, then the *location* part of the option argument may be omitted. For example, consider two HDF5 files, "`low_res.h5`" and "`high_res.h5`" which each have fields "`density`" and "`pressure`". Then the command

```
cigma compare high_res.h5:density low_res.h5:density -o low_high.h5
```

will compare the density fields between the two files on the mesh defined in `high_res.h5`.

Depending on how it was configured, Cigma understands a number of different file formats, including HDF5 and VTK formats. The file format is determined by the filename extension.

4.2 Comparison Options

We can exercise more control over the region of integration, and obtain a corresponding local error field by using the command

```
$ cigma compare \
        -a FunctionA \
        -b FunctionB \
        -m IntegrationMesh \
        -o L2_Errors_on_IntegrationCells
```

This command will evaluate `FunctionA` and `FunctionB` on the quadrature points of `IntegrationMesh`, and compute the L_2-norm of the difference between these two functions. The output file will be used to store each of the local L_2-differences between the two functions over the cells in the integration mesh.

4.3 Mesh Options

A mesh in Cigma is defined by three items: (1) the nodal Cartesian coordinates, (2) the connectivity (topological information describing how those nodes are connected to each other to form elements), and (3) an element type associated with the cell, e.g., triangles, quadrilaterals, tetrahedra, and hexahedra. Up to three such meshes, **MeshA** (the mesh for `FunctionA`), **MeshB** (for `FunctionB`) and the `IntegrationMesh`, can be passed as arguments in Cigma. These items are determined by the following command line options, arranged in a tabular format for easier reference.

Option	MeshA	MeshB	IntegrationMesh	Item
M	`--first-mesh`	`--second-mesh`	`--mesh`	(1,2,3)
M1	`--first-mesh-coords`	`--second-mesh-coords`	`--mesh-coords`	(1)
M2	`--first-mesh-connect`	`--second-mesh-connect`	`--mesh-connect`	(2)
MC	`--first-cell`	`--second-cell`	`--mesh-cell`	(3)

There are a number of rules determining which of these options are required. For any of the **MeshA**, **MeshB**, and **IntegrationMesh** columns in the table above, the relationship between these options is as follows:

1. If option (M) is given, then options (M1) and (M2) are forbidden.
2. If option (M) is missing, then both options (M1) and (M2) must be specified.
3. Option (MC) can be deduced from option (M).
4. Option (MC) is required if options (M1) and (M2) are given.

It is important to note that currently Cigma assumes that every element in the mesh is defined over the same cell type.

4.3.1 Data Layout in a Mesh File

When using an HDF5 file to store the mesh information, the *location*, or internal path, associated with option (M) must point to an HDF5 group that contains two arrays with the relevant mesh information. Those two arrays must be named `coordinates` and `connectivity`. If the *location* is empty, as is the case when only *filename* is specified, then the HDF5 group is assumed to be the root group (`/`) of the hierarchy. Lastly, the option (MC) can be omitted if the HDF5 group has an attribute called *`CellType`* with the appropriate value. A typical hierarchy that specifies a mesh would look like this:

```
File mesh.h5
 `Group /mesh
  |-- Attribute CellType
  |-- Array coordinates
  `-- Array connectivity
```

One may specify options (M1) and (M2) separately by pointing directly to the arrays corresponding to the coordinates and connectivity information. The node ordering on the connectivity dataset is described in Appendix A.

Alternatively, using a VTK file is very convenient. In this case, no *location* needs to be specified since all the appropriate information are always in the VTK file. The value of the option (MC) is determined from the first entry in the `CELL_TYPES` dataset, since we are limiting ourselves to element blocks consisting of a single cell type.

Another convenient way to prepare a mesh file for use with Cigma is to use the ExodusII format created by the CUBIT mesh generation toolkit. In this case, the mesh can be stored on disk using the NetCDF format. It suffices to give the *filename*, without the *location* part, to option (M). The appropriate mesh information will be read from the file. Only the first element block in the ExodusII file is used. The value of the (MC) option is determined from the NetCDF string attribute `elem_type` that is attached to the NetCDF integer variable called `connect1`.

Lastly, it is also possible to use a simple text file, in which case all three options (M1), (M2), and (MC) are required.

4.3.2 Integration Mesh

An integration mesh can be specified by the command line options, subject to the following rules:

1. If `IntegrationMesh` is specified, then it is used.
2. If `IntegrationMesh` is missing, then `MeshA` is used as `IntegrationMesh`.
3. If only `MeshB` is given, then it is used as `IntegrationMesh`.

Deciding if a given mesh is adequate for accurately capturing the error in the comparison will clearly depend on the functions being compared.

One strategy to ensure an adequate mesh would be to take the discretization cells on which the error metric exceeds a certain threshold, mark those cells for refinement, and recalculate the errors over the new discretization, repeating the process if necessary. This strategy could be implemented on top of Cigma as a series of post-processing steps by writing a suitable program or script that repeats this refinement process until the variations in the global error metric are small enough.

A second strategy would be to increase the order of the quadrature rule that we associate with the cells in the integration mesh. This approach has the advantage that we can increase the accuracy of the L_2-norm without performing pre- and post-processing on a sequence of integration meshes.

	IntegrationRule
R	`--rule`
R1	`--rule-points`
R2	`--rule-weights`

A number of common quadrature rules, including high-order quadratures, for different cell types are available by default. They are stored in file `integration-rules.h5`, which was generated using the Jacobi quadrature rules available in the FIAT (finite element automatic tabulator) Python module. Using these higher-order quadrature rules should allow you to increase the accuracy of the L_2-norm integral without having to adaptively refine the integration meshes.

The default integration rule used on each cell type can be obtained by running the `cigma element-info` command, as shown in Section 4.7.3.

4.4 Function Options

Cigma will accept two kinds of function arguments: (1) an analytic function chosen from a pre-defined list, or (2) a finite element field. Of these two, only the second kind is associated with a specific mesh. Therefore, when comparing two analytic functions, an integration mesh must (XXX).

4.4.1 Analytic Functions

Sometimes you will be able to express a function in terms of a general formula or algorithm, in which case you would like to be able to refer to such a function by using a simple name when specifying either of the `FunctionA` or `FunctionB` arguments. Extending Cigma by defining your own functions is ideal (XXX) for analytic functions for benchmarking your own code.

Two such examples can be found in the Cigma source code. A simple analytic function is available in the `fn_inclusion.h` and `fn_inclusion.cpp` source files (used in Example 5.3), while a more complex function which calls external procedures is available in `fn_disloc3d.h` and `fn_disloc3d.cpp`. If you wish to define your own analytic functions, it might be instructive to use either of these two examples as a template.

4.4.2 Finite Element Fields

A finite element description of a function is associated with three items of information: (1) a discretization (nodes and connectivity), (2) a finite set of local basis functions (cell type) defined on the discretization, and finally (3) a global list of coefficients defined on the nodes that yield the closest approximation to the function. How these items are specified depends on the underlying file format used to store the finite element description to our function. Items (1) and (2) are typically deduced from the appropriate mesh options discussed in Section 4.3, while item (3) is determined from the dataset given to either of the `--first` or `--second` command line options.

If the dataset is stored in an HDF5 file, item (3) would be typically specified as a single array containing the shape function coefficients. You can optionally attach an attribute called `MeshLocation` to that dataset, which points to the mesh datagroup.

```
File ‘‘model.h5’’
  Group fields
  |-- Group temperature
  |   |-- Array step000
  |   |   ‘-- Attribute MeshLocation
  |   |-- Array step001
  |   |   ‘-- Attribute MeshLocation
  |   ‘-- ...
  ‘-- Group displacements
      |-- ...
      ‘-- ...
```

Note that in this example, we have grouped all fields by variable name first (e.g., temperature and displacements) and then by time step, but we could have easily grouped everything by time step first and then by variable name. Cigma will only operate on the full path to a dataset, leaving you the freedom to organize your data as you see fit.

If the dataset is stored in a VTK file, the only restrictions are that you can only compare against Point Data arrays, and one cell type per file. Otherwise, you may use any of the formats supported by the VTK library (structured, rectilinear, etc.) to define your field.

4.4.3 Spatial Database Options

Recall from Chapter 3 that computing the local error metric involves approximating a local error integral by a finite weighed sum over a set of quadrature points. Because we allow arbitrary meshes for the integration procedure, that means that in general we must be able to evaluate our functions wherever those quadrature points may lie. This implies that we must be able to evaluate the given functions at arbitrary points. This step is typically inefficient even for a moderate-size finite element mesh, since a sequential scan over a mesh with n elements would take $O(n)$ time per quadrature point evaluation. To make this process efficient in general, we can build a spatial-index database of the geometric locations of every element in the mesh. Using a tree-based partitioning of the element locations, we can ideally reduce the search time to $O(\log n)$ per quadrature point evaluation.

Cigma uses a nearest-neighbor search on a kd-tree structure to find the appropriate cell that contains a given quadrature point. This means that for a given point, a fixed number of nearest neighbors are checked before proceeding with a sequential scan over all elements. If you find a particular comparison is taking too long, it may help to increase the number of nearest elements that are checked, which can be done by passing a positive integer to any of the command line options `--first-mesh-nnk` and `--second-mesh-nnk`.

For meshes with more structure, it would be possible to define an appropriate `cigma::Locator` subclass to provide a search time of $O(1)$ per quadrature point evaluation, but this is not currently implemented in Cigma.

4.5 Other Options

There are a few other notable options that enable us to monitor the timing statistics on the progress of the calculation, as well as debugging information that may prove useful when encountering an unexpected exception.

In order to monitor the progress of the integration, simply add the `--verbose` (or `-v`) flag to the command options,

```
$ cigma compare FunctionA FunctionB --verbose
```

By default, the timer will report its progress every 100 integration cells, but you can change that default by using the option `--timer-frequency` (or `-t`),

```
$ cigma compare FunctionA FunctionB --verbose --timer-frequency=1000
```

Debugging information can be displayed using the `--debug` (or `-D`) flag, which will output an internal trace of which functions have been called and with what arguments.

```
$ cigma compare FunctionA FunctionB --debug
```

On the other hand, suppressing all output can be accomplished with the `--quiet` flag. This flag is probably most useful when used together with the option `--global-threshold` (or `-g`),

```
$ cigma compare FunctionA FunctionB --quiet --global-threshold=0.001
```

In this case, the exit code will indicate failure (i.e., return a non-zero value) whenever the L_2-norm is grearter than the specified threshold condition. This allows you to set up automated regression scripts that can constantly compare output from your numerical codes against a series of known benchmark solutions.

4.6 Visualizing the Local Errors

The command `cigma compare` will always output the local L_2-errors in the "raw" form described in Equation 3.11. However, for visualization purposes, it's more convenient to normalize those integrated errors ε_i by the corresponding integration-cell volumes v_i, so that errors accumulated over smaller cells have more visual influence than errors of the same magnitude accumulated over larger cells. Additionally, it may also be more useful to output the logarithm of the normalized errors instead. Using a logarithmic scale will accentuate the contrast between the orders of magnitude when visualizing the normalized L_2-error field.

For these reasons, we include in Cigma a post-processing utility called `visualize-errors` that can take our error fields stored in HDF5 format and create a simple legacy VTK file, which can then be conveniently visualized using many other visualization packages. The utility `visualize-errors` can be used as follows,

```
$ visualize-errors [--use-logarithmic-scale] \
      -m MeshFile.h5:/meshpath              \
      -i InputErrors.h5:/errorpath          \
      -o OutputErrors.vtk:NormalizedErrors
```

Specifically, this command will take each of the local errors ε_i from the HDF5 file `InputErrors.h5`, normalize them by the factor $\sqrt{v_i}$, where v_i is the corresponding volume of the i-th cell taken from `MeshFile.h5`, and write the result to a VTK file called `OutputErrors.vtk` under an array called `NormalizedErrors`. One may also specify the flag `--use-logarithmic-scale`, in which case the above command will take the logarithm (in base 10) before writing the final results to the VTK file.

4.7 Verifying the Results

The rest of the Cigma commands are there to help you query your model data files, allowing you to determine whether the input files are being interpreted properly. A common problem in specifying a finite element mesh is using the wrong node numbering for a particular Cigma element, in which case you will probably encounter cells with negative or zero volumes, and incorrect results for the inverse reference map $\vec{x}_e^{-1}(\vec{x})$. Also, if the degrees of freedom are specified using a different node ordering than the mesh, the interpolation will yield different results than expected. Querying your model at a number of pre-selected points will probably curtail most of these mistakes when preparing your input files.

4.7.1 Working with a Mesh

Using the command `cigma mesh-info`, you can examine your mesh files and extract basic information such as the number of nodes and cells, the total volume, bounding box, and the maximum cell diameter of your mesh.

```
$ cigma mesh-info MESH
```

You can also extract connectivity information for a specific cell in your mesh, which will help you find out whether the elements that your finite element model use have the same node ordering as the elements in Cigma,

```
$ cigma mesh-info MESH --cell-id=N
```

Lastly, you can also query an arbitrary point within the mesh,

```
$ cigma mesh-info MESH --query-point=”[x,y,z]”
```

This will return the information of the cell that contains the query point, which is useful to verify that the underlying spatial index that maps points to cells is working properly.

4.7.2 Working with Analytic Functions

Finding information on given function can be done with the `cigma function-info` command. Using it without arguments will return the list of analytic functions that have been compiled into Cigma. Two special scalar functions,

```
$ cigma function-info
one
zero
...
```

You can also query any function accepted by the `cigma compare` command, not just built-in analytic functions, at any arbitrary point in its function domain,

```
$ cigma function-info FUNCTION --query-point=”[x,y,z]”
```

This will allow you to verify whether the function is reporting the correct value at the expected point. You can use this to check that the element interpolations are done correctly, and whether a newly defined analytic function returns the appropriate values on a selection of points.

4.7.3 Working with Elements

On a more basic level, you can also verify that the basis functions for your reference element work as expected, especially if you decide to extend Cigma by adding your own element types. Calling the `cigma element-info` command without arguments will generate the list of registered elements. Note that the names of the element types listed here correspond to the acceptable values that you can give to the (MC) argument discussed in Section 4.3.

```
$ cigma element-info
List of elements available for use in a Field:
   tet4, hex8
   tri3, quad4
```

For example, querying the hex8 element would result in the following information being displayed

```
$ cigma element-info hex8
Information on cell 'hex8'
Reference-Cell Nodes:
     0  -1.000000 -1.000000 -1.000000
     1  +1.000000 -1.000000 -1.000000
     2  +1.000000 +1.000000 -1.000000
     3  -1.000000 +1.000000 -1.000000
     4  -1.000000 -1.000000 +1.000000
     5  +1.000000 -1.000000 +1.000000
     6  +1.000000 +1.000000 +1.000000
     7  -1.000000 +1.000000 +1.000000
Shape functions:
   N[0] = 0.125 * (1.0 - u) * (1.0 - v) * (1.0 - w)
   N[1] = 0.125 * (1.0 + u) * (1.0 - v) * (1.0 - w)
   N[2] = 0.125 * (1.0 + u) * (1.0 + v) * (1.0 - w)
   N[3] = 0.125 * (1.0 - u) * (1.0 + v) * (1.0 - w)
   N[4] = 0.125 * (1.0 - u) * (1.0 - v) * (1.0 + w)
   N[5] = 0.125 * (1.0 + u) * (1.0 - v) * (1.0 + w)
   N[6] = 0.125 * (1.0 + u) * (1.0 + v) * (1.0 + w)
   N[7] = 0.125 * (1.0 - u) * (1.0 + v) * (1.0 + w)
Default integration rule (weights & points):
    1.000000   -0.577350 -0.577350 -0.577350
    1.000000   +0.577350 -0.577350 -0.577350
    1.000000   +0.577350 +0.577350 -0.577350
    1.000000   -0.577350 +0.577350 -0.577350
    1.000000   -0.577350 -0.577350 +0.577350
    1.000000   +0.577350 -0.577350 +0.577350
    1.000000   +0.577350 +0.577350 +0.577350
    1.000000   -0.577350 +0.577350 +0.577350
```

Querying the shape function values is possible by using the previously discussed command `cigma function-info` and an appropriately defined mesh containing a single element. These reference meshes are provided in the top level data file `reference-cells.h5`.

Chapter 5

Examples

In this chapter, we show how to use Cigma to run specific comparisons on included datasets, estimate the order of convergence of the numerical methods used in obtaining those solutions, and visualize the resulting error fields.

5.1 Poisson Problem

Cigma can compare two functions and return the L_2-norm difference between those two functions. For a given numerical problem, if you compute the solutions over a range of resolutions, you can use this global measure to quantify the rate of convergence of your numerical method. To demonstrate how this works, we will use the two Poisson problems:

(1) $\nabla^2\phi(x,y) = 4x^4 + 4y^4$ inside $\Omega = [-1,1]^2$, subject to the Dirichlet boundary condition $\phi(x_0,y_0) = x_0^2 + y_0^2$ on the boundary $\partial\Omega$, and

(2) $\nabla^2\phi(x,y,z) = 4x^4 + 4y^4 + 4z^4$ on $\Omega = [-1,1]^3$ subject to $\phi(x_0,y_0,z_0) = x_0^2 + y_0^2 + z_0^2$ on the boundary $\partial\Omega$.

These differential equations can be easily solved numerically by using a finite element software library called Deal.II, which supports Lagrange finite elements of any order in either 2 or 3 dimensions. In fact, the two equations we have described are already solved in Step 4 in the list of the Deal.II tutorial programs in (`dealii.org/developer/doxygen/tutorial/`).

In the `examples/` subdirectory, we include a version of the Step 4 tutorial program that has been slightly modified to suit our purposes in this section. First, we change the output format to use VTK files, in order to make it very convenient for us to provide our input datasets to Cigma. Next, we solve the 2D problem over meshes of resolution 64×64, 32×32, 16×16, and 8×8 by using linear quadrilateral elements. The solution is saved in file `phi_64x64x64.vtk` for the highest resolution, for example. In a similar fashion, we solve our 3D Poisson problem on meshes of resolution $64 \times 64 \times 64$, $32 \times 32 \times 32$, $16 \times 16 \times 16$, and $8 \times 8 \times 8$, that use linear hexahedral elements. Note that in either case, we don't really know the exact solution to either problem, so we use the highest resolution available as the reference point for our comparisons. The following bash shell command can take care of all the comparisons

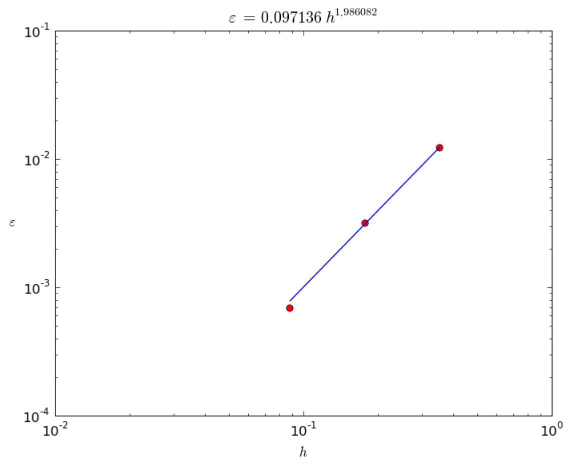
```
$ for r in 32 16 8; do
    cigma compare phi_64x64.vtk     phi_${r}x${r}.vtk
    cigma compare phi_64x64x64.vtk  phi_${r}x${r}x${r}.vtk
  done
```

When we run the above code on the command line, or script, Cigma will generate a summary of each comparison, including the L_2 norm of the difference between the two functions, and other basic mesh information such as its volume V and maximum cell diagonal h. We summarize that information in the following tables:

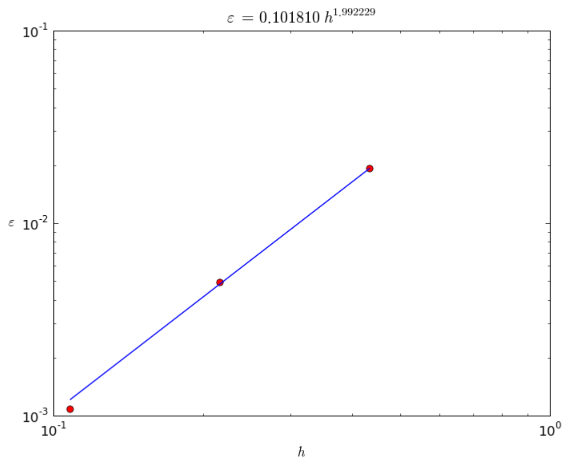
n	h_n	$\|\|\phi_n - \phi_{64\times 64}\|\|_{L_2}/\sqrt{V}$
8×8	0.35355	0.012312
16×16	0.17677	0.003157
32×32	0.08838	0.000689

n	h_n	$\|\|\phi_n - \phi_{64\times 64\times 64}\|\|_{L_2}/\sqrt{V}$
$8\times 8\times 8$	0.43301	0.019205
$16\times 16\times 16$	0.21650	0.004889
$32\times 32\times 32$	0.10825	0.001075

It should become apparent from these results that the global error metric ε does indeed decrease with increasing resolution at the expected rate. To verify the exact order of convergence, you can use regression analysis to fit a power law through the above points, and obtain the following plots:



(a) 2D Poisson Problem



(b) 3D Poisson Problem

Figure 5.1: Convergence of L_2 Global Error for Poisson problems in 2D and 3D.

Based on the slope of the lines in the above log-log plots, we have indeed verified that the global errors behave as $\varepsilon = Ch^2$, as expected from a method that uses linear elements.

5.2 Mantle Convection

In addition to reporting a global error metric, Cigma can also output local errors between two fields, averaged over each of the integration cells used in the calculation of the L_2 norm (or error). You can use this scalar error field to ascertain a number of physical insights by correlating the spatial regions that contribute the most to the global error.

For our next example, we use CitcomCU, a CIG code for mantle convection, to solve a thermal convection problem inside a three-dimensional domain under base heating, stress-free boundary conditions, constant viscosity, and using a Rayleigh number of 10^5 in the domain $\Omega = [0, 1]^3$. Under these conditions, we expect the solution to this problem to eventually converge to a steady state consisting of a single convection cell. We then use Cigma to compare those steady state solutions in order to recover the order of convergence of the CitcomCU numerical code and to examine how the error behaves on a representative slice of the original domain.

Again, as in the previous section, we solve the same problem over four different resolutions, and then use the highest resolution dataset as the reference point for all the subsequent comparisons. The four cases we will use for this example use $64 \times 64 \times 64$, $32 \times 32 \times 32$, $16 \times 16 \times 16$, and $8 \times 8 \times 8$ elements. In the following figure, we display the temperature and velocity fields on the $y = 0.99$ plane, near one of the faces of the domain. We can distinctly identify an upwelling and downwelling cycle, as well as sharp gradients in the temperature field. The variations in the velocity field are smoother throughout the slice.

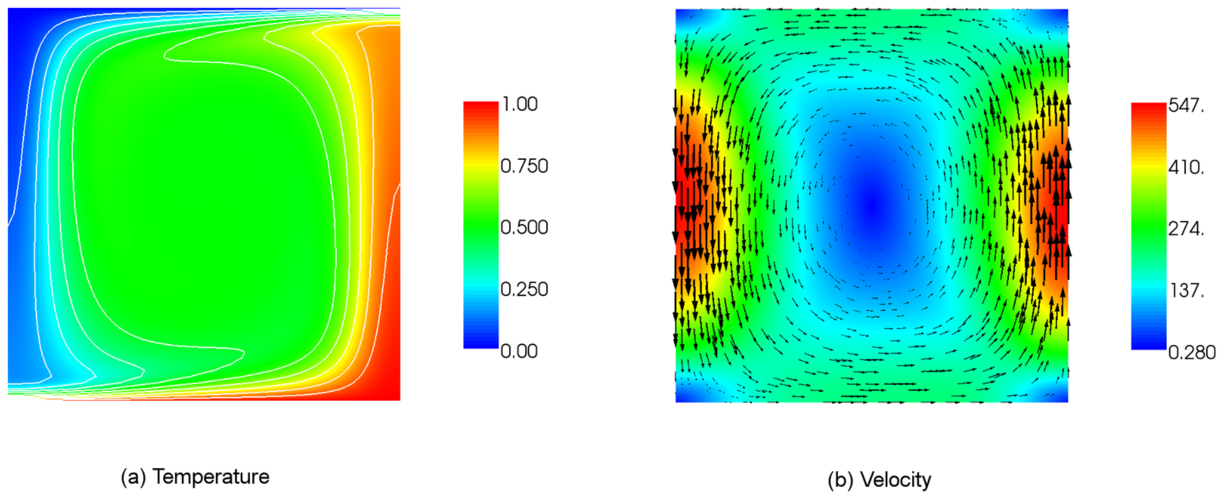


Figure 5.2: Slice of the steady state temperature and velocity fields for our thermal convection problem, on the $y = 0.99$ plane. In the temperature plot (a), we show ten equally spaced contour values. In the velocity plot (b), the vectors represent the xz-plane components of the velocity, while the colors represent the velocity magnitude.

Once we obtain the solutions for our four resolutions over 20000 solution steps, we process the ASCII output data from CitcomCU into a number of rectilinear grids in VTK format (*.vtr files), and a subsequent *.pvtr file for each solution step.

As a preliminary step, we also need to ensure that each of our cases do indeed reach a steady state. We can verify this fact numerically by using Cigma to compare the same field at two different times, and checking that the L_2-error is negligible. However, for this problem, we can also visually verify that the solution has reached steady state by checking that the oscillations in the values of the average heat flux on the top surface have dampened out, as shown by Figure 5.3,

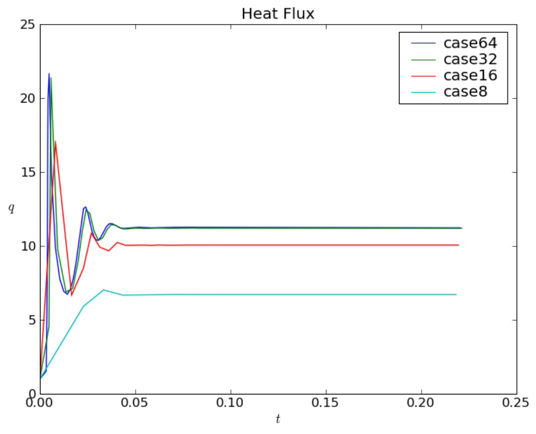


Figure 5.3: Time variation of average heat flux over top surface.

Our first comparison of two steady states is simple enough. Although it is not critical for this example since we have already converged, in general time-dependent problems you must ensure you are comparing solutions at the same absolute time. Thus, even though it appears that the following command is comparing two different solution steps, they actually correspond to roughly the same physical time.

```
$ cigma compare \
    -a case64.20000.pvtr:temperature \
    -b case32.9900.pvtr:temperature  \
    -o steady-state.h5:/error_temperature_64_32
```

Since we did not specify an integration mesh through the `-m` option, Cigma will take the mesh from the first field and use it for the integration of the L_2 error. We have, however, used the `-o` option to specify an output file. This will create the HDF5 file `steady-state.h5` after the comparison is complete, and store the results of that comparison into an array called `error_temperature_64_32` under the HDF5 `/` root group. You may also refer to the same HDF5 file in subsequent runs of Cigma. The target dataset will be appended to the file if it doesn't exist, and overwritten if it does. The commands for other two comparisons are,

```
$ cigma compare \
    -a case64.20000.pvtr:temperature \
    -b case16.4700.pvtr:temperature \
    -o steady-state.h5:/error_temperature_64_16
$ cigma compare \
    -a case64.20000.pvtr:temperature \
    -b case8.2100.pvtr:temperature \
    -o steady-state.h5:/error_temperature_64_8
```

Comparisons involving the velocity can be obtained in a similar manner. Note that even though we are comparing vectors here, the resulting errors will be scalars, since Cigma is integrating the scalar function $||\vec{v}_a(\vec{x}) - \vec{v}_b(\vec{x})||^2$.

```
$ cigma compare \
     -a case64.20000.pvtr:velocity \
     -b case32.9900.pvtr:velocity \
     -o steady-state.h5:/error_velocity_64_32
$ cigma compare \
     -a case64.20000.pvtr:velocity \
     -b case16.4700.pvtr:velocity \
     -o steady-state.h5:/error_velocity_64_16
$ cigma compare \
     -a case64.20000.pvtr:velocity \
     -b case8.2100.pvtr:velocity \
     -o steady-state.h5:/error_velocity_64_8
```

At this point, you have obtained the local L_2-errors for each of the above comparisons, and have stored them in arrays inside the file `steady-state.h5`. In order to visualize these errors, you will need to reattach the original mesh information by using a post-processing utility program called `visualize-errors`. We can accomplish this for our six comparisons, by using the following bash comand line,

```
$ for variable in temperature velocity; do
    for b in 32 16 8; do
      visualize-errors \
        --use-logarithmic-scale \
        -m case64.20000.vtr \
        -i steady-state.h5:/error_${variable}_64_${b} \
        -o log_error_temperature_64_${b}.vtk:log_error_${variable}
    done
  done
```

The first two options we give our post-processing program are important. The first option eliminates the volume scale factor in the definition of the local L_2 error (larger cells contribute more than smaller cells, which would be more obvious if the error were constant). The second option tells the post-processing program to use a logarithmic scale for writing those locally averaged errors. Using a log-10 scale makes it much easier to spot the spatial range of variation in our error fields.

In Figure 5.4, we show two plots representing the errors in the temperature field on the $y = 0.99$ plane for the two $16 \times 16 \times 16$ and $32 \times 32 \times 32$ resolutions, relative to the $64 \times 64 \times 64$ temperature solution. Examining this plot and Figure 5.2a reveals that the largest errors are concentrated in areas where the temperature gradient is also large, near the boundary layers created by activity from the upwelling and downwelling. Similarly, the region where the errors are smallest corresponds to a region where both the temperature gradients and velocity magnitude are at their lowest.

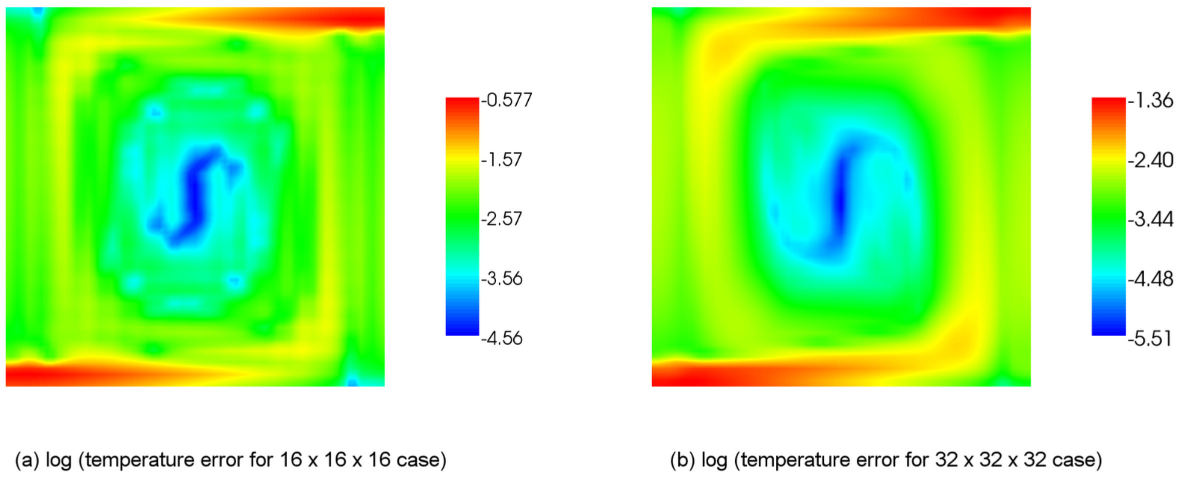


Figure 5.4: Temperature differences on $y = 0.99$ plane, as compared against case with $64 \times 64 \times 64$ elements.

Likewise, in Figure 5.5 we show two plots representing the magnitude of the velocity field on the same plane. This time, the distribution of the largest errors are spread out more evenly over the domain. In this case, the largest errors in the velocity field also correspond to regions where the magnitude of the velocity field is the largest.

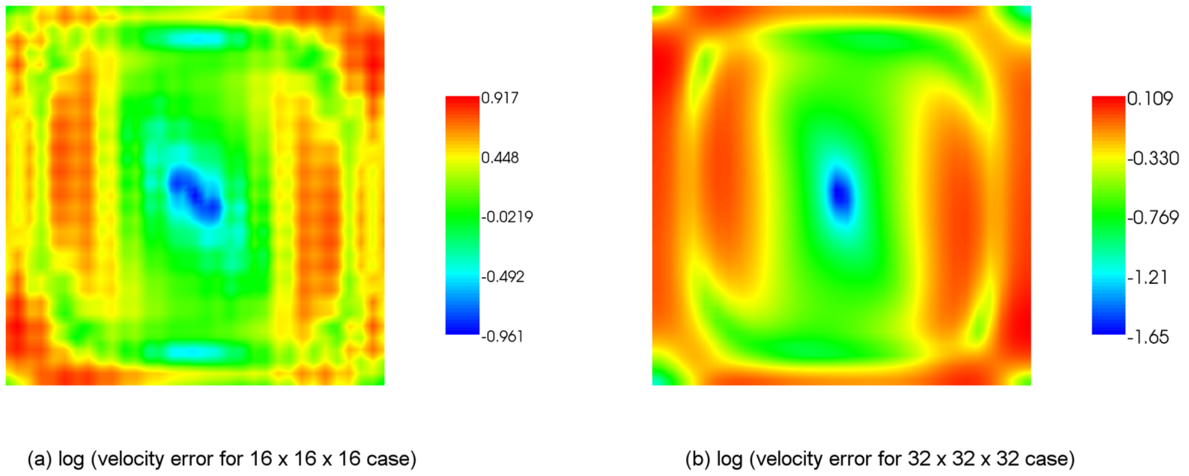


Figure 5.5: Velocity differences on $y = 0.99$ plane, as compared against case with $64 \times 64 \times 64$ elements.

5.3 Circular Inclusion Benchmark

In some cases, you will be able to obtain an exact solution to your differential equations. In these cases, you can extend Cigma by implementing your function directly in C++, and registering a convenient name under which to call it. In this section, we'll show you exactly how to accomplish this by considering a two-dimensional problem for which an exact analytical solution is known. In a 2003 paper [8], Schmid and Podladchikov derived an analytic solution for the pressure and velocity fields of a circular inclusion under simple shear, depicted in Figure 5.6. The viscosity parameters introduced are μ_m for the viscosity of the matrix, and μ_i for the viscosity of the inclusion. The kinematic boundary conditions are given generally in terms of the simple shear strain rate $\dot{\epsilon}$.

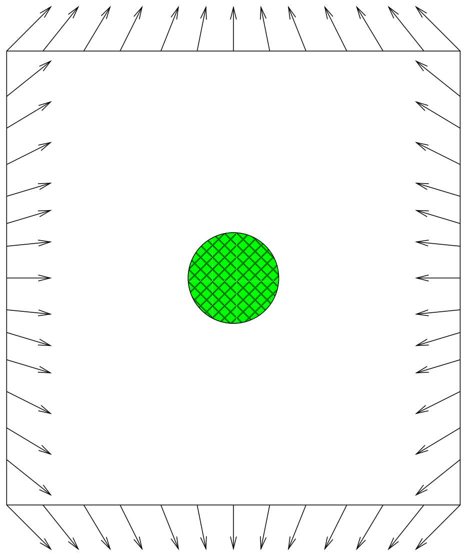


Figure 5.6: Schematic for the circular inclusion benchmark.

In particular, we place the inclusion of radius $r_i = 0.1$ at the origin, and exploit the symmetry in this problem by only solving the field on the top right quarter of the domain.

From [8], we end up with the following analytic formula for the pressure field under the case of simple shear,

$$p = \begin{cases} 4\dot{\epsilon}\frac{\mu_m(\mu_i-\mu_m)}{\mu_i+\mu_m}\left(\frac{r_i^2}{r^2}\right)\cos(2\theta) & r > r_i \\ 0 & r < r_i \end{cases} \tag{5.1}$$

where we use $\mu_i = 2$ for the viscosity of the inclusion, $\mu_m = 1$ for the viscosity of the background media, and $\dot{\epsilon} = 1$ for the magnitude of the shear. In Cigma, you can refer to this specific analytic solution by the somewhat verbose name `bm.circular_inclusion.pressure`, whose definition is summarized in Listing 5.1.

Listing 5.1: Definition of the `bm.circular_inclusion.pressure` analytic function.

```
#include Function.h

namespace benchmark {
  namespace circular_inclusion {
    class Pressure;
  }
}

class benchmark::circular_inclusion::Pressure : public cigma::Function {
public:
    Pressure() {}
    ~Pressure() {}

    virtual void init() { }
    virtual FunctionType getType() const { return Function::GeneralType; }

    virtual int n_dim() { return 2; }
    virtual int n_rank() { return 1; }

    virtual bool eval(double *x, double *y) {
        const double R = 0.1;
        const double mu_m = 1;
        const double mu_i = 2;
        const double mag_shear = 1.0;
        const double C = 4*mag_shear*(mu_m*(mu_i-mu_m)/(mu_i+mu_m))*R*R;
        const double xc = 0.0;
        const double yc = 0.0;
        const double dx = x[0] - xc;
        const double dy = y[0] - yc;
        const double r2 = dx*dx + dy*dy;

        if (r2 < R*R) {
            value[0] = 0.0;
        } else {
            double theta = atan2(dy, dx);
            value[0] = (C/r2) * cos(2*theta);
        }

        return true;
    }
};
```

One additional step must be taken before we can use the name `bm.circular_inclusion.pressure` on the Cigma command line. We must instantiate our new C++ class, and register it under an assigned name in the constructor defined in the `FunctionRegistry.cpp` source file.

Listing 5.2: Assigning the name `bm.circular_inclusion` to our analytic function.

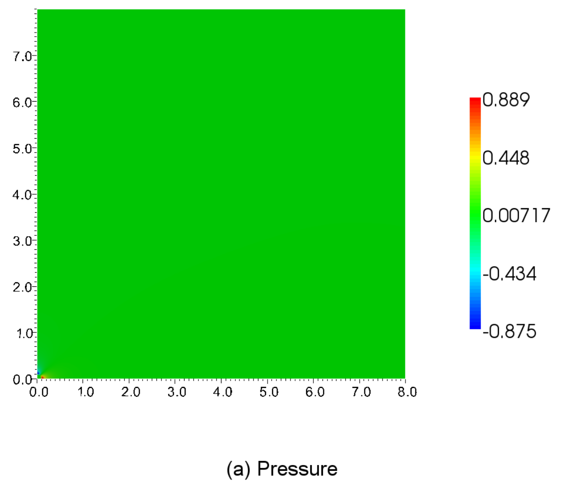
```
FunctionRegistry::FunctionRegistry()
{
    //
    // ... other definitions
    //

    typedef benchmark::circular_inclusion::Pressure PressureFn1;
    shared_ptr<PressureFn1> pressure1(new PressureFn1());
    this->addFunction(bm.circular_inclusion.pressure, pressure1);
}
```

Now that we have an analytic expression for the pressure, we would like to obtain an approximate numerical solution to this circular inclusion problem. For this purpose we can use Gale, a CIG code for long-term crustal dynamics. Since the analytic formula for the velocity field is more complicated than the analytic formula for the pressure, we will simply expand the domain and use the far-field approximation of the velocity field in order to impose the kinematic boundary conditions in Gale,

$$v_x = +\dot{\epsilon}x \qquad (5.2)$$
$$v_y = -\dot{\epsilon}y \qquad (5.3)$$

Since we expect our solution to vary as $p \sim 1/r^2$, we target an error of 0.01% by enlarging the domain under consideration to about 80 times the radius of the inclusion. This results in the final domain $\Omega = [0, 8]^2$.


(a) Pressure

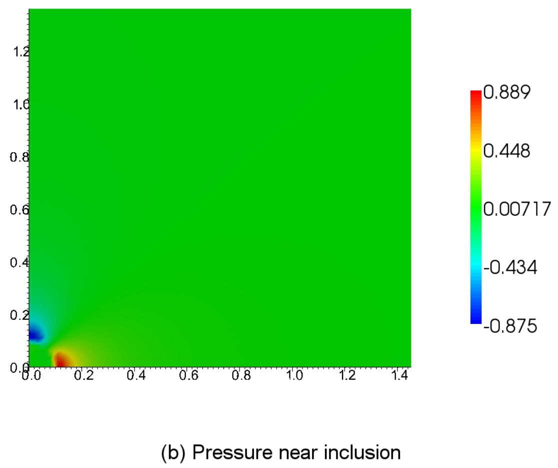

(b) Pressure near inclusion

Figure 5.7: Pressure field for circular inclusion problem, with resolution of 512 × 512 elements.

First, we'd like to see how well the Gale solutions converge to a common answer by comparing each other against the highest resolution field available. Since we only have three solutions, that leaves us with only two meaningful comparisons,

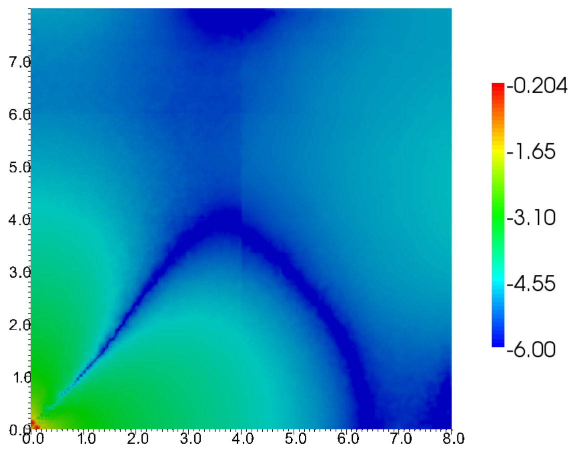
```
$ cigma compare \
    -a 512_8/fields.00000.pvts:PressureField \
    -b 128_8/fields.00000.pvts:PressureField \
    -o inclusion.h5:/error_pressure_512_128
$ cigma compare \
    -a 512_8/fields.00000.pvts:PressureField \
        -b 256_8/fields.00000.pvts:PressureField \
        -o circ_inc.h5:/error_pressure_512_256
```

whose results we summarize in the following Table.

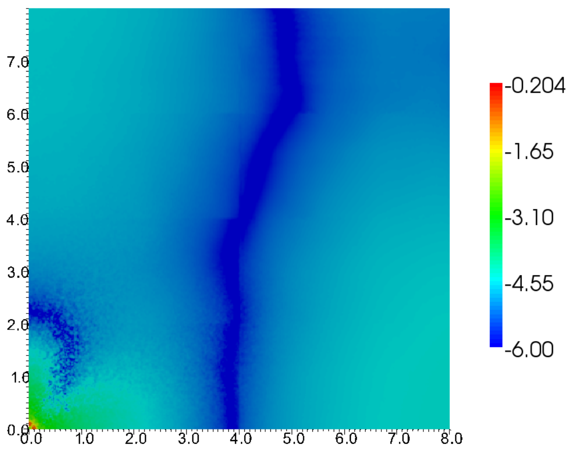
n	h_n	$\|\|p_n - p_{512}\|\|_{L_2}/\sqrt{V}$
128	0.088388	0.0040745
256	0.044194	0.0015786

from which we can estimate the order of convergence $\alpha = \log(0.00158/0.00407)/\log(0.0442/0.0884) = 1.36$.

Processing the local error fields stored in `circ_inc.h5`, we obtain the following plots,

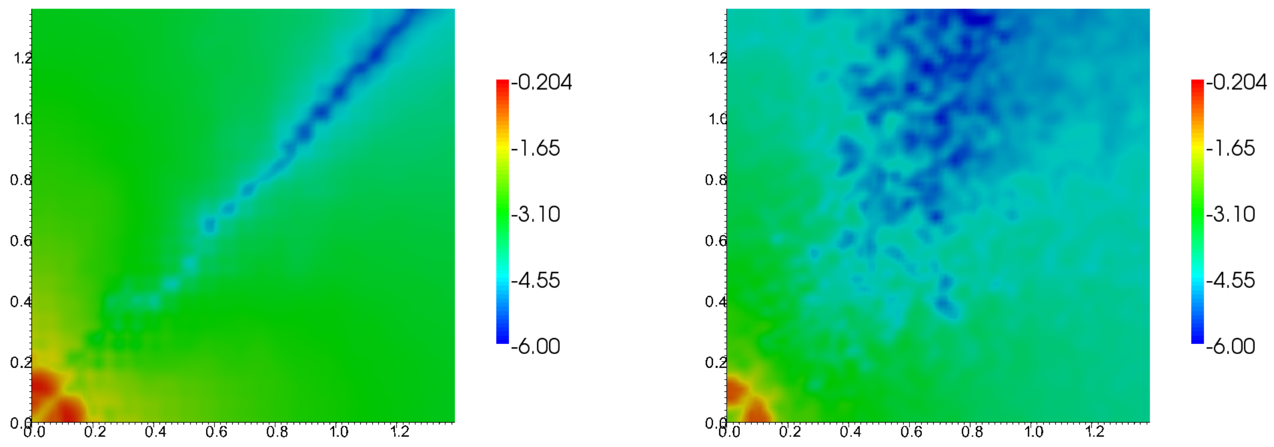


(a) log (pressure error for 128 x 128 case)



(a) log (pressure error for 256 x 256 case)

Figure 5.8: Errors in pressure field for 128×128, and 256×256 cases, relative to the 512×512 case, shown over the entire domain.



(a) log (pressure error for 128 x 128 case)

(a) log (pressure error for 256 x 256 case)

Figure 5.9: Errors in pressure field for 128×128, and 256×256 cases, relative to the 512×512 case, shown near the inclusion.

Next, we compare each of the three pressure fields obtained with Gale against the analytical formula for the pressure given earlier. We accomplish that by invoking the following bash command line,

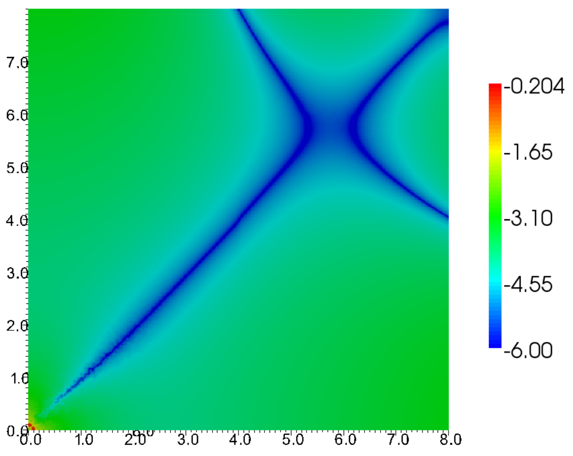
```
$ for r in 512 256 128; do
    cigma compare \
      -a ${r}_8/fields.00000.pvts:PressureField \
      -b bm.circular_inclusion.pressure \
      -o inclusion.h5:/errror_pressure_${r}
  done
```

whose output we summarize in the following table,

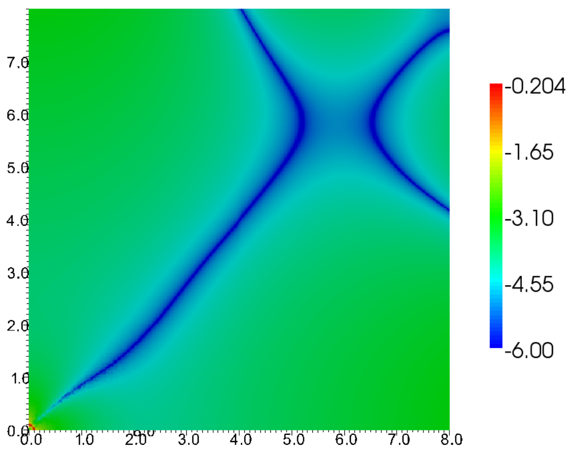
n	h_n	$\|\|p_n - p\|\|_{L_2}/\sqrt{V}$
128	0.088388	0.0050388
256	0.044194	0.0036636
512	0.0022097	0.0024535

from which we can obtain the order of convergence $\alpha = 0.5$, via a power-law regression analysis. Such a low value of α is expected since the pressure function we are solving has a discontinuity across the inclusion boundary.

After processing the errors in `inclusion.h5` with the utility program `visualize-errors`, we obtain the following plots of the pressure field differences,

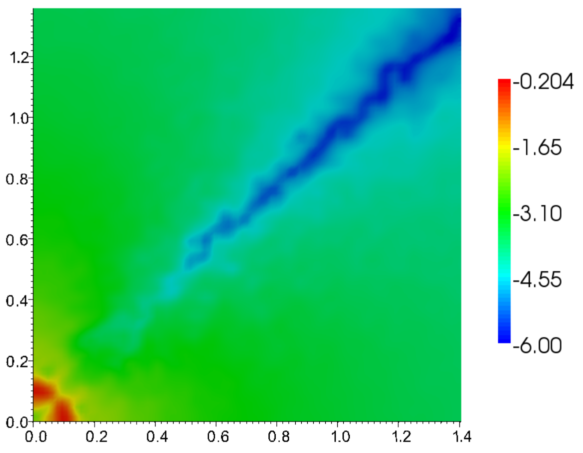


(a) log (pressure error for 256 x 256 case)

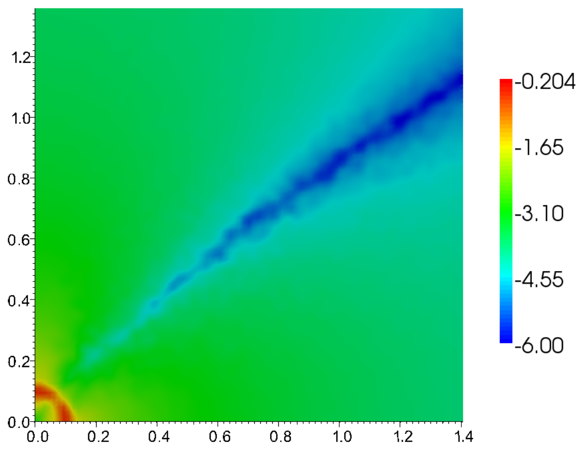


(b) log (pressure error for 512 x 512 case)

Figure 5.10: Errors in pressure field for 256×256 and 512×512 element resolutions, relative to the analytic formula, shown over the entire domain.



(a) log (pressure error for 256 x 256 case)



(b) log (pressure error for 512 x 512 case)

Figure 5.11: Errors in pressure field for 256×256 and 512×512 element resolutions, relative to the analytic formula, shown near the inclusion.

The only appreciable change between these two plots happens in the region near the inclusion boundary, which becomes slightly more resolved spatially. Also, in contrast to the previous two sections, we do not detect much of a color shift between two plots, even though the L_2 global error is indeed decreasing. This is due to the fact that the error remains large near the inclusion with approximately the same maximum value, making it harder to see any variation in the local error field.

In summary, we have observed that for this problem the L_2 global error does indeed decrease as we increase the mesh resolution, but it does not do so at the optimal rate. This indicates that the numerical scheme used to obtain these solutions is not completely accurate. Note that many numerical codes that solve Stokes flow, Gale included, assume that the pressure, velocity, and viscosity fields are continuous, an assumption that is violated for this particular problem.

Appendix A

Input and Output

A.1 Supported File Formats

Cigma can understand a number of file formats, depending on how it is configured (see Chapter 2). By default it will read and write all information in binary form as HDF5 files, which is a format optimal for archiving data with minimal redundancy. VTK files are also sometimes used by finite element software, due to the ease with which the solution fields can be visualized. Lastly, ExodusII mesh files generated by the CUBIT mesh generation package can also be used directly as integration meshes.

The most flexible way to store your arrays will be to use an HDF5 file (with extension `.h5`), which will allow you to organize your arrays in a hierarchy. The `h5attr` allows you to add or modify any scalar metadata attributes to any HDF5 group or array.

You can also use a simple text file (with extension `.dat`) containing a single array in row-column format.

.h5	input/output	HDF5 Format
.dat	input/output	Text Format
.vtk	input/output	Legacy VTK Format
.vtu	input	VTK File for Unstructured Datasets
.vts	input	VTK File for Structured Datasets
.vtr	input	VTK File for Rectilinear Datasets
.exo	input	Exodus Mesh Format

Table A.1: Overview of Cigma's file formats

A.2 Data Formats

The basic data structure is a two-dimensional array of values, stored in a contiguous format as shown below:

Array shape	(m, n)
Array Data	$a_{11}, a_{12}, \ldots, a_{1n}$ $a_{21}, a_{22}, \ldots, a_{2n}$ $\cdots$ $a_{m1}, a_{m2}, \ldots, a_{mn}$

This layout is sufficient for describing both 2- and 3-dimensional data. Note the second index in this array varies the fastest, so that data often referenced together remains together in memory as well.

A.2.1 Node Coordinates

On a mesh with n_{no} node coordinates, which are specified on a global coordinate system, we have,

Array Shape	$(n_{no}, 3)$
3D Coordinates	x_1, y_1, z_1
	x_2, y_2, z_2
	$\vdots$
	$x_{n_{no}}, y_{n_{no}}, z_{n_{no}}$

Array Shape	$(n_{no}, 2)$
2-D Coordinates	x_1, y_1
	x_2, y_2
	$\vdots$
	$x_{n_{no}}, y_{n_{no}}$

A.2.2 Element Connectivity

Mesh connectivity is specified at the element-block level. On a given block, we only consider a single element type, which allows us to store the global node ids into a contiguous array of integers. Multiple blocks are specified separately, and they all reference the same node ids into the node coordinates table.

In general, if our block contains k elements with m-degrees of freedom each, then the connectivity array defining our element block has the form

Array Shape	(k, m)
Connectivity Data	$n_{11}, n_{12}, \ldots, n_{1m}$
	$n_{21}, n_{22}, \ldots, n_{2m}$
	$\vdots$
	$n_{k1}, n_{k2}, \ldots, n_{km}$

For the element types defined in Cigma, the values for m are as follows. Note that the total number of degrees of freedom depends on the rank of the specific function being represented.

3D Element	Scalar	Vector	Tensor
tet4	4	12	24
tet10	10	30	60
hex8	8	24	48
hex20	20	60	120
hex27	27	81	162

Similarly for the 2D elements, we have

2D Element	Scalar	Vector	Tensor
tri3	3	6	9
quad4	4	8	12

A.2.3 Field Variables

A field variable represents one of the functions that we can evaluate, and is typically stored over a series of timesteps. The data for a field variable consists of snapshots through time, which are stored as separate arrays. A value for each degree of freedom is provided on the global list of nodes in the mesh. The ordering must be the same as the ordering of the node coordinates.

A.2.4 Shape Function Values

In certain circumstances you may be able to provide your custom reference element by simply providing the appropriate n shape function values to be used at the expected integration points. This is most useful when you are using the same mesh for both the integration mesh and the array.

Array Shape	Shape Function Values: (n, Q)	Jacobian Determinant: $(Q, 1)$
Integration Point 1	$N_1(\vec{\xi_1}), N_2(\vec{\xi_1}), \ldots, N_n(\vec{\xi_1})$	$J(\vec{\xi_1})$
Integration Point 2	$N_1(\vec{\xi_2}), N_2(\vec{\xi_2}), \ldots, N_n(\vec{\xi_2})$	$J(\vec{\xi_2})$
$\vdots$	$\vdots$	$\vdots$
Integration Point Q	$N_1(\vec{\xi_Q}), N_2(\vec{\xi_Q}), \ldots, N_n(\vec{\xi_Q})$	$J(\vec{\xi_Q})$

A.2.5 Integration Rules

As mentioned in Section 4.3.2, an integration rule is specified by a list of points and associated weights. The points should be specified on the natural coordinate system used by the corresponding reference element.

Array Shape	Weights: $(Q, 1)$	Points: (Q, n_{dim})	2D Points: $(Q, 2)$	3D Points: $(Q, 3)$
Rule Definition	w_1	$\vec{\xi_1}$	ξ_1, η_1	ξ_1, η_1, ζ_1
	w_2	$\vec{\xi_2}$	ξ_2, η_2	ξ_2, η_2, ζ_2
	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	w_Q	$\vec{\xi_Q}$	ξ_Q, η_Q	ξ_Q, η_Q, ζ_Q

A.2.6 Integration Points and Values

Given a mesh and an integration rule, we can map the integration points on each element into a possibly large set of global points.

Array Shape	Points: (n_{pts}, n_{dim})	2D Points: $(n_{pts}, 2)$	3D Points: $(n_{pts}, 3)$
Coordinates	$\vec{x}_1$	x_1, y_1	x_1, y_1, z_1
	$\vec{x}_2$	x_2, y_2	x_2, y_2, z_2
	$\vdots$	$\vdots$	$\vdots$
	$\vec{x}_{n_{pts}}$	$x_{n_{pts}} y_{n_{pts}}$	$x_{n_{pts}}, y_{n_{pts}}, z_{n_{pts}}$

The result of an evaluation will depend on the rank of the function being evaluated, as indicated in the table below. Note that the fastest varying dimension of the array contains the components of the function value at each point.

Evaluated Quantity	Resulting Array Shape
Scalar	$(n_{pts}, 1)$
2D Vector	$(n_{pts}, 2)$
3D Vector	$(n_{pts}, 3)$
2D Tensor	$(n_{pts}, 3)$
3D Tensor	$(n_{pts}, 6)$

A.2.7 Local L_2-errors

Since our norm reduces integrals over cells to a single scalar value, the result of a comparison will result in a simple scalar array for each element block examined. Note that element blocks used in the integration process belong to the integration mesh, which may differ from the associated mesh to either function.

Thus, for each element block containing k cells, we have the output array

Array Shape	$(k, 1)$
Error Value	ε_1
	ε_2
	$\vdots$
	ε_k

These error values may be combined in the following form:

$$\varepsilon_{block} = \sqrt{\varepsilon_1^2 + \varepsilon_2^2 + \cdots + \varepsilon_k^2}$$

The final global error norm maybe obtained by simply summing over all element blocks that partition the domain of interest:

$$\varepsilon_{global} = \sqrt{\sum_{block} \varepsilon_{block}^2}$$

Currently, this last calculation can only be performed using a problem-specific post-processing script, since Cigma will only perform comparisons on a single element block.

A.3 Input Files

In Section A.2, we examined what kinds of objects we will be accessing, so now let's discuss the actual layout in the files in which these objects will be stored.

A.3.1 HDF5

HDF5 files are binary files encoded in a data format designed to store large amounts of scientific data in a portable and self-describing way.

The internal organization of a typical HDF5 file consists of a hierarchical structure similar to a UNIX file system. Two types of primary objects, *groups* and *datasets*, are stored in this structure. A group contains instances of zero or more groups or datasets, while a dataset stores a multi-dimensional array of data elements. In a sense, datasets are analogous to files in a traditional file system, and groups are analagous to folders. One important difference, however, is that you can attach supporting metadata to both kinds of objects. These associated metadata are known as *attributes.*

A dataset is physically stored in two parts: a header and a data array. The header contains miscellaneous metadata describing the dataset as well as information that is needed to interpret the array portion of the dataset. Essentially, it includes the name, datatype, dataspace, and storage layout of the dataset. The name is a text string identifying the dataset. The datatype describes the type of the data array elements. The dataspace defines the dimensionality of the dataset, i.e., the size and shape of the multi-dimensional array.

With Cigma you always provide an explicit path to specific datasets, so you have a fair amount of flexibility in how you organize datasets into groups inside your HDF5 files. For example, a typical file could have the following structure,

```
file.h5
\__ model
    |__ mesh
    |   |__ coordinates    [nno x nsd]
    |   \__ connectivity   [nel x ndof]
    \__ variables
        |__ temperature
        |   |__ step00000  [nno x 1]
        |   |__ step00010  [nno x 1]
        |   \...
        |__ displacement
        |   |__ step00000  [nno x 3]
        |   |__ step00010  [nno x 3]
        |   \...
        \__ velocity
            |__ step00000  [nno x 3]
            |__ step00010  [nno x 3]
            \...
```

where the structure of the mesh does not change from timestep to timestep.

Alternatively, a file which stores data associated with a mesh that changes over time would have a different structure,

```
file.h5
\__ model
    |__ step00000
    |   |__ mesh
    |   |   |__ coordinates
    |   |   \__ connectivity
    |   |__ variables
    |       |__ temperature
    |       |__ displacement
    |       \__ velocity
    |__ step00010
    |   |__ mesh
    |   |   |__ coordinates
    |   |   \__ connectivity
    |   |__ variables
    |       |__ temperature
    |       |__ displacement
    |       \__ velocity
    \__ ...
```

where we have allowed for each.

To summarize, Cigma requires you to specify a full path to a specific dataset. Moreover, by attaching a small amount of metadata to your datasets, it becomes possible to reduce the amount of information that must be provided to the comparison routine, such as what mesh to use in conjunction with a given set of coefficients, and which set of shape functions to use in the respective evaluations.

MeshLocation is a string representing the HDF5 path pointing to the mesh group which contains the coordinates and connectivity datasets. This attribute should be attached to the array corresponding to your degrees of freedom (shape function coefficients).

CellType is a string identifier used to determine which shape functions to use for interpolating values inside the element (e.g., tet4, hex8, quad4, tri3, ...). This attribute should be attached to the mesh group that contains the coordinates and connectivity datasets.

Even if you forgot to set these attributes when creating your HDF5 datasets, setting or changing them can be done easily by using the `h5attr` utility included in Cigma. For example, the following command would let Cigma know that the path `/model/mesh` corresponds to a linear tetrahedral mesh

```
$ h5attr model.h5:/model/mesh CellType tet4
```

Likewise, the following command would associate the mesh `/model/mesh` with the field coefficients `/model/variables/temperature/step00000`

```
$ h5attr model.h5:/model/variables/temperature/step00000 MeshLocation /model/mesh
```

Setting the MeshLocation attribute in this manner has the advantage that corresponding mesh options can be omitted when calling `cigma compare` on the command line.

A.3.2 VTK

For detailed information on this format, you may want to refer to Visualization ToolKit's File Formats (`www.vtk.org/pdf/file-formats.pdf`) document. Here we will only note that using VTK files is convenient due to the fact that the mesh is always required by the file format. You typically only need to provide the file name, and the name of the dataset inside the file that you wish to use in the comparison operation, although you may safely omit the dataset name if your VTK file contains only a single point dataset. Perhaps one of the most important things to keep in mind, when using VTK files with the current version of Cigma, is that unstructured datasets will need to consist of cells of a single type.

A.3.3 Text

The text files we use in Cigma consist of a simple list of numbers in ASCII format whose layout corresponds exactly to the simple array layout discussed earlier in Appendix A.2. The first line of the file contains the dimensions of the array, just two integers separated by whitespace. The rest of the file specifies the array data, and must contain as many numbers as the product of the two dimensions given in the first line, all separated by whitespace. These numbers should be formatted as integer or floating point, depending on whether you are describing coordinates or coefficients, or whether you are referring to a connectivity array.

Some restrictions apply when providing data in text format, mostly because you can only specify a single element block in a text file containing connectivity information. Operations involving a larger number of blocks must use data in the format described in Appendix A.3.1.

Another point worthy of mention is that arrays must be given in two-dimensional form, i.e., two integers are always expected in the first line. Therefore, when specifying a one-dimensional array, such as a list of weights corresponding to an integration rule, the second array dimension must be 1.

A.4 Output Files

For the most part, there are three different kinds of output files you can use, as already summarized in Appendix A.1. Switching output formats is as simple as changing the extension of the output file name.

The result of all comparisons always consists of a one dimensional list of scalars, one for each element in the underlying discretization used to approximate the L_2-norm integral. You can output these error values directly into a VTK file, in which case the array will be stored in the Cell Data section of the output file. Since Cigma includes a post-processing utility called `visualize-errors` that can convert HDF5 errors into VTK files, we recommend that you use HDF5 files to store the result of your comparisons. Note that HDF5 files will not be overwritten when used for output, which allows you to store multiple datasets inside the same HDF5 file without keeping redundant mesh information as is typically the case when using VTK files.

Bibliography

[1] Akin, J.E. (2005), *Finite Element Analysis with Error Estimators,* Butterworth-Heinemann, Oxford, 447 pp.

[2] P. Knupp, K. Salari (2003), *Verification of Computer Codes in Computational Science and Engineering,* Chapman & Hall/CRC, 144 pp.

[3] Hughes, Thomas J.R. (2000), *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*, Dover, 672 pp.

[4] Karniadakis, George E. and Spencer J. Sherwin (2005), *Spectral/*hp *Element Methods for Computational Fluid Dynamics, Second Edition*, Numerical Mathematics and Scientific Computation, Oxford, 657 pp.

[5] Cools, R., *An Encyclopaedia of Cubature Formulas*, J. Complexity, 19: 445-453, 2003; Katholieke Universiteit Leuven, Dept. Computerwetenschappen, www.cs.kuleuven.be/~nines/research/ecf/

[6] Uesu, D., L. Bavoil, S. Fleishman, J. Shepherd, and C. Silva (2005), Simplification of Unstructured Tetrahedral Meshes by Point-Sampling, *Proceedings of the 2005 International Workshop on Volume Graphics,* 157-165, doi: 10.2312/VG/VG05/157-165.

[7] Travis, J., et al (1990), A benchmark comparison of numerical methods for infinite Prandtl number thermal convection in two-dimensional Cartesian geometry, *Geophys. Astro. Fluid 55*(3), 137-160, doi: 10.1080/03091929008204111.

[8] Schmid, D.W., and Y.Y. Podladchikov (2003), Analytical solutions for deformable elliptical inclusions in general shear, *Geophys. J. Int. 155*(1), 269-288, doi: 10.1046/j.1365-246X.2003.02042.x.