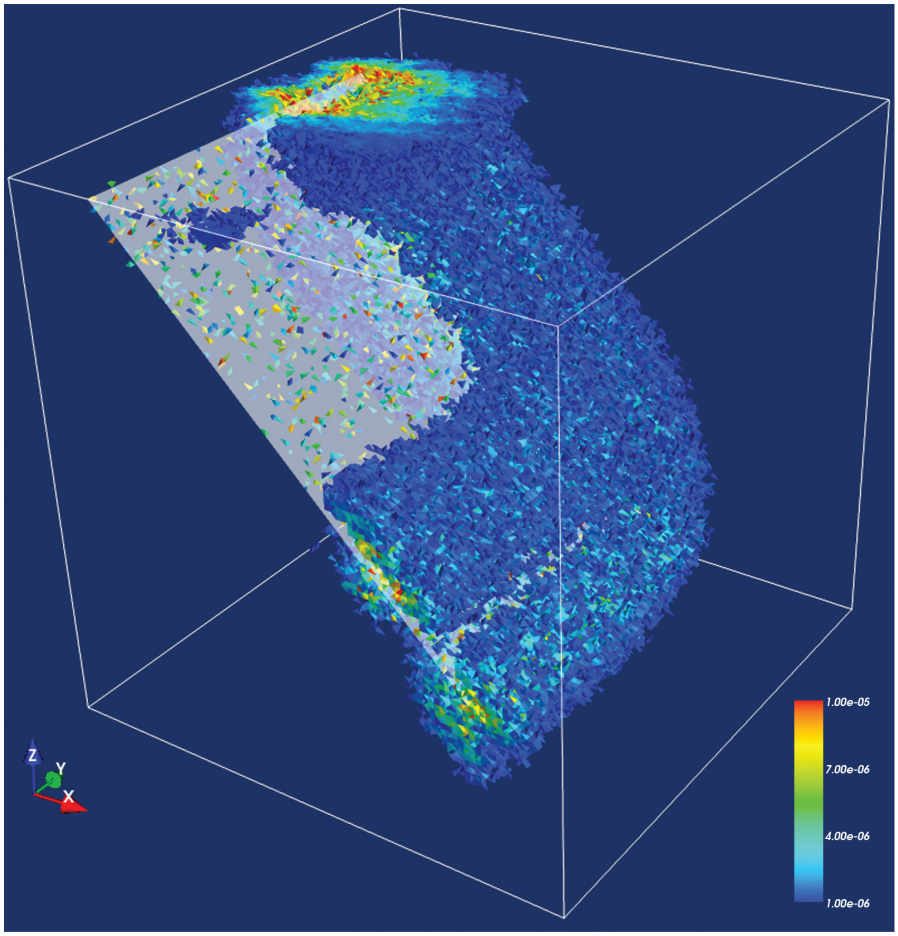
COMPUTATIONAL INFRASTRUCTURE FOR GEODYNAMICS (CIG)

Cigma

User Manual
Version 1.0.0



Luis Armendariz
Susan Kientz

www.geodynamics.org

Cigma

Version 1.0.0

March 2, 2009

Contents

List of Figures

Chapter 1

Introduction

1.1 About Cigma

The CIG Model Analyzer (Cigma) consists of a general suite of tools for comparing numerical models. In particular, this program is intended for the calculation of L_2 residuals for finite element models and published benchmark datasets.

CIG has 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. In the long term, Cigma aims to be general enough for use in other geodynamics modeling codes.

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 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.

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 dpendencies from your package manager of choice, or from the corresponding website. When using a package manager, make sure to install the associated development headers along with the library, as these tend to be distributed separately from the library package itself.

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-netcdf=$NETCDF_PREFIX    \
    --with-vtk=$VTK_PREFIX          \
    --with-vtk-version=$VTK_VERSION \
    --with-cppunit-prefix=$CPPUNIT_PREFIX
$ make
$ make install
```

Only the Boost and HDF5 libraries are required components for this step. The additional configure flag `--with-vtk-version` is required if you wish to enable VTK support.

In general, the installation prefixes specified above should be used if you have installed the corresponding library in a custom location. In the instructions below, we use a subdirectory of `$HOME/opt` as the installation prefix, but you may change it to something else. After all these steps are complete, you should be able to run the `cigma` binary and start comparing arbitrary functions.

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, you may choose to avoid long compilation times by building Boost with only the necessary components that Cigma needs,

```
$ 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
```

Other components of Boost that need to be built can be chosen from the list generated by the command

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

and modifying the comma-delimited list passed to the `--with-libraries` option when repeating the build procedure above.

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. Cigma depends on the C++ API of the HDF5 library, so 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 use 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
```

If it is not already available on your system, you must build the zlib compression library before building HDF5 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

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. Enabling NetCDF support in Cigma will currently only allow you to read the first element block from any ExodusII file created by the CUBIT mesh generator.

2.2.6 CppUnit library

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 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 (`sourceforge.net/projects/cppunit/`).

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 an 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 on 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 ***residual 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 residual field used under this norm is defined in terms of the linearized equation from which the solution was obtained.

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* would 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 residual 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 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,

$$\begin{aligned} \vec{X}_q &= \chi_i(\vec{\xi}_q) \\ W_q &= w_q J_i(\vec{\xi}_q) \end{aligned}$$

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 residual 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,

$$f(\vec{x}) = d_1\phi_1(\vec{x}) + d_2\phi_2(\vec{x}) + \cdots + d_m\phi_m(\vec{x}) \tag{3.3}$$

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 directly to find the values of $f(\vec{x})$, and we may refer to the $\phi_j$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 residual 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 residual ε_e^2 is given by

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

In general, we won't be able to integrate each cell residual 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 residual by applying an appropriate quadrature rule with a tolerable truncation error **[?]**.

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:

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

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

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

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:

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

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 residual field:

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

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

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

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

$$\varepsilon = \sqrt{\sum_{e=1}^{\mathrm{nel}} \sum_{q=1}^{\mathrm{nQ}} w_q ||u(\vec{x}_e(\vec{\xi}_q)) - v(\vec{x}_e(\vec{\xi}_q))||^2 J_e(\vec{\xi}_q)}$$

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

3.6 Comparing Global Residuals

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 the maximum element size h of the underlying discretization, and an exact solution $u(\vec{x})$, this normalized error is given by

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

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

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

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. 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 you are comparing against.

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 the be estimated from the two approximate bounds

$$\begin{aligned} \varepsilon_1 &\sim Ch_1^\alpha \\ \varepsilon_2 &\sim Ch_2^\alpha \end{aligned}$$

which can be combined into

$$\left(\frac{\varepsilon_2}{\varepsilon_1}\right) \sim \left(\frac{h_2}{h_1}\right)^\alpha$$

or, solving for α, we arrive at the simple equation

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

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

$$\log \varepsilon_i = \log C + \alpha \log h_i$$

with regression variables $X_i = \log h_i$ and $Y_i = \log \varepsilon_i$. The slope of the regression line will result in an estimate of α. For this purpose, a short script called `power-plot.py`, based on the SciPy and Matplotlib Python modules, is provided in the Cigma source code. You can use it for both calculating and plotting the regression line associated with the points (X_i, Y_i), given an input file containing the points (h_i, ε_i).

Chapter 4

Running Cigma

Cigma is primarily designed for calculating error estimates between arbitrary functions, where the primary operation has the form

```
$ cigma compare FunctionA FunctionB -m IntegrationMesh -o LocalResiduals
```

This command will compute local and global error metrics for the difference between two functions specified on the command line. Typically, you will also want to specify a particular discretization to use as the integration mesh. The output file will be used to store the results of the comparison, which are defined relative to the integration mesh.

In this chapter, we give a general overview of the options used in Cigma, and show actual usage examples in the next Chapter. To access the full list of options simply run 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 local and global residuals over two fields
   eval           Evaluate function at specified quadrature points
   extract        Extract quadrature points from a mesh
   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 querying & analyzing numerical models.
For additional information, visit http://geodynamics.org/
```

For the remainder of this chapter, we will focus on the `compare` command, and the information

4.1 Selecting a Dataset

All datasets in Cigma are assumed to be simple two dimensional array of values. As 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. There are a number of different file formats available for specifying and storing your data, according to how Cigma was configured at build time. The file format will be derived directly from the *filename* extension.

4.2 Mesh Options

A mesh block is associated with three items of information: (1) geometrical information for a number of nodes defined on a global coordinate system, and (2) topological information describing how those nodes are

connected to each other to form elements, and (3) an element type associated with the cell. In Cigma, these items are determined by the following command line options, arranged in tabular format for easy reference.

Option	MeshA	MeshB	IntegrationMesh	Items Read
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, 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 Cigma assumes every element in the mesh uses the same basis functions.

4.2.1 Mesh Files

When using an HDF5 file to store the mesh information, the *location* 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.

```
File filename
  Group location
  |-- Attribute CellType
  |-- Array coordinates
  `-- Array connectivity
```

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 can be read implicitly from the `POINTS`, and `CELLS` , which are required by the VTK file format. 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.

4.2.2 Integration Mesh

An integration mesh can be specified by the command line options in Section 3.2, subject to the following rules

1. At least one of MeshA, MeshB, or IntegrationMesh must be given.

2. If IntegrationMesh is specified, then it is always used.

3. If IntegrationMesh is missing, then MeshA is copied to IntegrationMesh.

4. If only MeshB is given, then it is copied to 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 residuals 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. Both of the TetGen and CUBIT mesh generators can be used for this purpose.

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`

Various quadrature rules for different cell types are available in the top level 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, although this will come at a cost of an increasing number of function evaluations due to the higher number of quadrature points for the high-order rules.

The default integration rule used on each cell type can be obtained by running the `cigma element-info` command, as shown in Section 4.5.3.

4.3 Function Options

Cigma will accept two kinds of function arguments: (1) an analytic function chosen from a pre-defined list, or (2) an array of coefficients describing a finite element field. Both of these

4.3.1 By an Analytic Function

Sometimes you will be able to express a function in terms of a general formula or algorithm, in which case 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 for analytic functions that other people who have downloaded Cigma can use to benchmark their own codes. Two such examples can be found in the Cigma source code. A simple analytic benchmark is available in the `fn_gale2.h` and `fn_gale2.cpp` source files, while a more complex benchmark which calls external procedures is available in `fn_disloc3d.h` and `fn_disloc3d.cpp`. For detailed help on the exact steps involved in registering your own functions, you may refer to Appendix B.

4.3.2 By a Finite Element Field

A finite element description of a function is associated with three items of information: (1) a finite set of local basis functions, (2) a discretization over which the function is locally approximated by those basis functions, and finally (3) a global list of shape function coefficients 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 4.2, 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.

```
File filename
  Group variables
  |-- 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 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 unstructured datasets must not mix more than one element in the same file. Otherwise, you may use any of the formats supported by the VTK library (structured, rectilinear, etc.) to define your field.

4.3.3 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 proceding 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.

4.4 Other Options

There are a few other notable options that enable us to monitor various things, such as 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 A B --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 A B --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 A B --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 A B --quiet --global-threshold=0.001
```

In this case, the exit code will indicate failure (return a non-zero value), whenever the specified threshold condition is not met. 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.

4.5 Visualizing the Local Errors

The command `cigma compare` will always output the local residuals in the "raw" form described in Section 3.5. However, for visualization purposes, you may wish to renormalize the integrated errors ε_e by the corresponding integration-cell volumes v_e, so that errors accumulated over smaller cells have more visual influence than errors of the same magnitude accumulated over larger cells. It may be also be useful visually, to output the logarithm of the residual values. Using a logarithmic scale will accentuate the contrast betwen the orders of magnitude in the local residuals. For these reasons, we include in Cigma a post-processing utility called `vtk-residuals` that can take residuals stored in HDF5 format and create a simple legacy VTK file, which can then be conveniently visualized using any number of visualization packages. The utility `vtk-residuals` can be used as follows,

```
$ vtk-residuals [...]            \
      -m MeshFile                \
      -i InputResiduals.h5:/path \
      -o OutputResiduals.vtk
```

By default, this command will only copy the residual values from the input HDF5 file into the output VTK file. Further processing can be performed by specifying two additional options. The option `--divide-by-cell-volumes` will normalize each local residual by its corresponding cell volume in the MeshFile. The other option is `--output-log-values`, which will take the logarithms (base 10) of each residual before writing the VTK file.

4.6 Verifying the Results

The rest of the Cigma commands are there to help you query your model, allowing you to determine whether the input files are being interpreted properly. A common problem in specifying a finite element mesh would be using the wrong node numbering for a particular Cigma element, in which case you might encounter cells with negative or zero volumes, incorrect results for the inverse reference map χ_e^{-1}. Also, if the degrees of freedom are specified using a different node ordering than the mesh, the interpolation will yield different results than expected.

4.6.1 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]''
```

which you can use to verify that the underlying spatial index that maps points to cells is working properly.

4.6.2 Working with Functions

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

```
$ cigma function-info
...
one
zero
test.square
test.cube
...
```

You can also query any function accepted by the `cigma compare` command, not just builtin ones, 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.

4.6.3 Working with Elements

On a more basic level, you can also verify that the elements basis functions 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 4.2.

```
$ cigma element-info
List of elements available for use in a Field:
   tet4, tet10
   hex8
   tri3, tri6
   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`. For example, querying N_0 for a tet4 reference cell would be accomplished with

```
$ cigma function-info reference-cells.h5:/tet4/dofs/N0 --query-point 0,0,0
$ cigma function-info reference-cells.h5:/tet4/dofs/N0 --query-point 1,0,0
$ cigma function-info reference-cells.h5:/tet4/dofs/N0 --query-point 0,1,0
$ cigma function-info reference-cells.h5:/tet4/dofs/N0 --query-point 0,0,1
```

which serves as a quick check that N_0 evaluates to 1 at one vertex, and to 0 at the other vertices.

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 solutions presented, and discuss a number of visualization techniques.

5.1 Mantle Convection

For this example, we use CitcomCU to solve a thermal convection problem in a three-dimensional domain. The initial temperature field is a linear gradient from the top surface to the bottom surface. The temperature is fixed at 1 at the bottom of the cube, and fixed at 0 at the top of the cube. In this case, we have solved for the velocity field for three different resolutions and stored it in three rectilinear vtk files called `citcomcu.case8.vtr`, `citcomcu.case16.vtr`, and `citcomcu.case32.vtr`.

Now, we run the first comparison, comparing the highest resolution,

```
$ cigma compare \
    -a citcomcu.case32.vtr:velocity \
    -b citcomcu.case8.vtr:velocity  \
    -o citcomcu.h5:/case_32_08
Summary of comparison:
  L2 = 0.0500057391169
  Linf = 0.146940986884
  volume = 1
  L2/volume = 0.0500057391169
  L2/sqrt(volume) = 0.0500057391169
  h1 = 0.0541265877365
  h2 = 0.216506350946
```

Since we did not specify an integration mesh, the mesh from the first field is used for the integration of the L_2 norm. Additionally, the above command creates the HDF5 file `citcomcu.h5`, and stores the results of the comparison into an array called `case32_08`.

```
$ cigma compare \
    -a citcomcu.case32.vtr:velocity \
    -b citcomcu.case16.vtr:velocity \
    -o citcomcu.h5:/case_32_16
Summary of comparison:
  L2 = 0.0100758230674
  Linf = 0.0322452153235
  volume = 1
  L2/volume = 0.0100758230674
  L2/sqrt(volume) = 0.0100758230674
```

```
h1 = 0.0541265877365
h2 = 0.108253175473
```

From these two results, we can estimate how fast we are converging to a common answer between levels a and b by using

$$\begin{aligned} \alpha &\sim \frac{\log(\varepsilon_b/\varepsilon_a)}{\log(h_b/h_a)} \\ &\sim \frac{\log(0.01/0.05)}{\log(0.108/0.217)} \\ &\sim 2.3 \end{aligned}$$

where we have assumed that the highest resolution field is equivalent to the exact solution in our approximation for α.

Using a logarithmic scale to view the residual field for these two comparisons, we generate two vtk files using the comands

```
$ vtk-residuals \
    --output-log-values \
    -m citcomcu.case32.vtr:velocity \
    -i citcomcu.h5:/case_32_08 \
    -o log-res-vel-32-08.vtk
$ vtk-residuals \
    --output-log-values \
    -m citcomcucase32.vtr:velocity \
    -i citcomcu.h5:/case_32_16 \
    -o log-res-vel-32-16.vtk
```

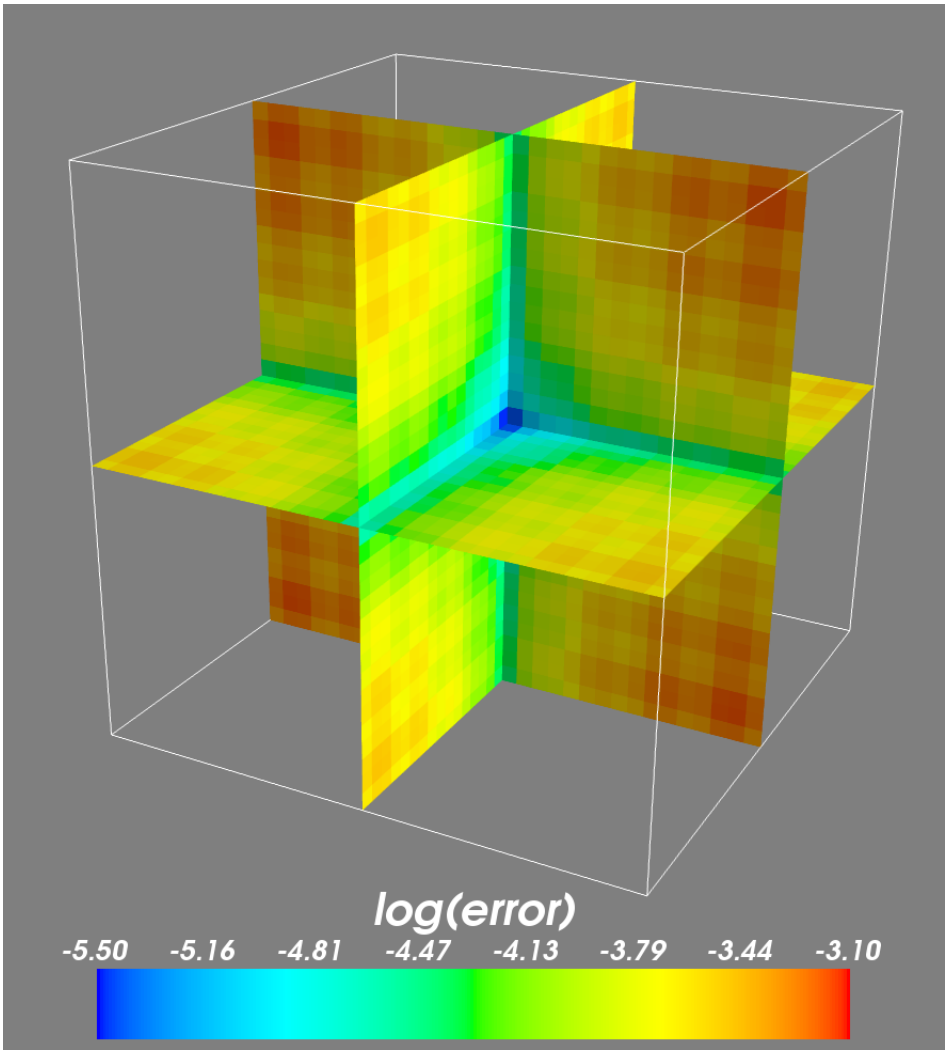


Figure 5.1: Caption for 5.1 here

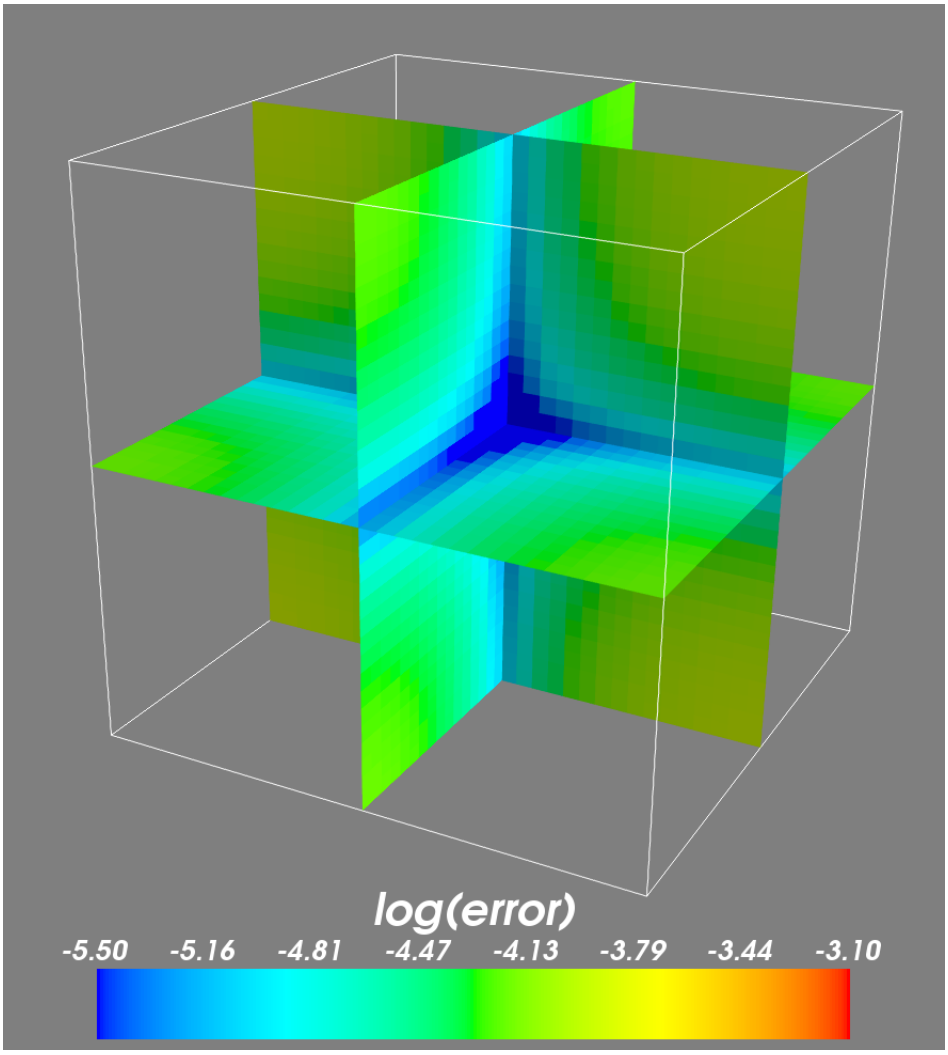


Figure 5.2: Caption for 5.2 here

In the figures above, we show three cross sections of the error in the velocity field. The convergence behavior of these two comparisons can almost be confirmed visually by observing the overall color shift between the two figures, which use the same absolute color scale.

5.2 Circular Inclusion Benchmark

We begin by analyzing a two-dimensional example benchmark problem for which we know an exact analytical solution.

```
$ cigma compare \
    -a p256.vts:PressureField \
    -b p128.vts:PressureField \
    -o circ_inc.h5:/pressure_256_128
$ cigma compare \
    -a p256.vts:PressureField \
    -b p64.vts:PressureField \
    -o circ_inc.h5:/pressure_256_064
```

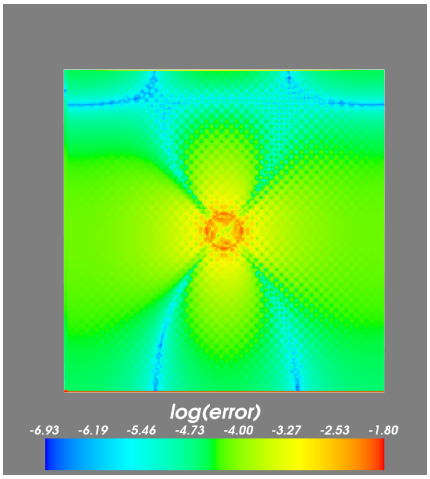


Figure 5.3: Caption for 5.3 here

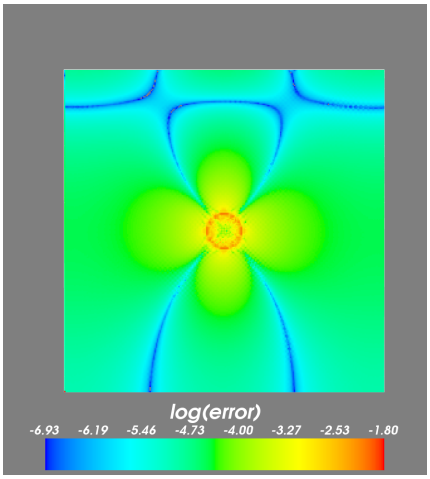


Figure 5.4: Caption for 5.4 here

The analytic solution is registered under the somewhat verbose name

```
bm.gale.circular_inclusion.pressure
```

which we shorten by storing it in a shell variable,

```
$ export p=PressureField
$ export bm=bm.gale.circular_inclusion
$ cigma compare ${bm}.pressure p256.vts:${p} -o circ_inc.h5:/pressure_256
$ cigma compare ${bm}.pressure p128.vts:${p} -o circ_inc.h5:/pressure_128
$ cigma compare ${bm}.pressure p64.vts:${p}  -o circ_inc.h5:/pressure_064
```

5.3 Strikeslip Benchmark Convergence

This benchmark problem computes the viscoelastic (Maxwell) relaxation of stresses from a single, finite strike-slip earthquake in 3D without gravity. In order to obtain several data points, we use the CUBIT mesh generator to create a sequence of meshes with a 1.2 refinement ratio on which to solve. In this case, we also calculate the displacement field over 100 timesteps, saving every 10th field.

```
$ export a=hex8_0500m
$ export b=hex8_1000m
$ export d=displacements
$ export steps=‘seq -f ’%04g’ 0 10 100‘  # generates list 0000 0010 0020 ... 0100
$ for n in ${steps}; do
    for b in ‘‘hex8_1000m hex8_0833m hex8_0694m hex8_0578m’’; do
      echo ‘‘Calculating ${a}-${b}-${n}’’
      cigma compare \
        -a strikeslip_${a}_t${n}.vtk:${d} \
        -b strikeslip_${b}_t${n}.vtk:${d} \
        -o ‘‘strikeslipnog.h5:/${d}-${a}-${b}-${n}’’ \
        --verbose
    done
  done
```

5.4 Laplace Problem

Here we obtain a sequence of solutions by solving a Laplace problem inside a cubic domain.

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 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 visualize. 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 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.

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	$a_{11}, a_{12}, \ldots, a_{1n}$ $a_{21}, a_{22}, \ldots, a_{2n}$ $\cdots$ $a_{m1}, a_{m2}, \ldots, a_{mn}$

The second index in this array varies the fastest, so that data often referenced together remains together in memory as well.

.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.2.1 Node Coordinates

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

Array Shape	$(n_{no}, 3)$
3-D 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.

3-D 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 2-D elements, we have

2-D 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.

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 Rule

As described in Chapter 4, an integration rule is specified by a list of points and associated weights. The points are given on natural coordinate system of the reference element.

Array Shape	Weights: $(Q, 1)$	Points: (Q, n_{dim})	2-D Points: $(Q, 2)$	3-D 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})	2-D Points: $(n_{pts}, 2)$	3-D 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	$(n_{pts}, 1)$
2-D Vector	$(n_{pts}, 2)$
3-D Vector	$(n_{pts}, 3)$
2-D Tensor	$(n_{pts}, 3)$
3-D Tensor	$(n_{pts}, 6)$

A.2.7 Comparison Residuals

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	$(k, 1)$
Residual Value	ε_1
	ε_2
	$\vdots$
	ε_k

These residual values may be combined in the following form:

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

The final global residual 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 1.1, 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 filesystem, and groups are analagous to folders. One important difference, however, is that you can attach supporting metadata to both kinds of objects. The metadata objects are called *attribute*s.

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.

In Cigma you may always provide an explicit path to every dataset, so you have a fair amount of flexibility for how you organize your datasets inside HDF5 files. For example, a typical Cigma HDF5 file could have the following structure.

```
model.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]
\...
```

Generally, Cigma will only require you to specify the path to a specific field dataset. If a small amount of metadata is present in your field dataset, the rest of the required information, such as which mesh and finite elements to use, will be deduced from that metadata.

MeshLocation points to the HDF5 group which contains the appropriate coordinates and connectivity datasets. This attribute should be attached to the array corresponding to your degrees of freedom (shape function coefficients).

CellType string identifier 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 dataset.

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 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 if 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 Section 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 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 Section 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 Section 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 residual 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 `vtk-residuals` that can convert HDF5 residuals 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.

Appendix A

Data and File Formats

As we've seen in the previous chapters, there are a number of data objects involved in making any particular comparison. Here, we discuss the variety of formats available, as well as the layout within the corresponding data file.

A.1 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	$a_{11}, a_{12}, \ldots, a_{1n}$ $a_{21}, a_{22}, \ldots, a_{2n}$ $\cdots$ $a_{m1}, a_{m2}, \ldots, a_{mn}$

The second index in this array varies the fastest, so that data often referenced together remains together in memory as well.

A.1.1 Node Coordinates

One of the more efficient

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

Array Shape	$(n_{no}, 3)$
3-D 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.1.2 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.

3-D 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 2-D elements, we have

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

A.1.3 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.

A.1.4 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.1.5 Integration Rule

As described in Chapter 4, an integration rule is specified by a list of points and associated weights. The points are given on natural coordinate system of the reference element.

Array Shape	Weights: $(Q, 1)$	Points: (Q, n_{dim})	2-D Points: $(Q, 2)$	3-D 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.1.6 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})	2-D Points: $(n_{pts}, 2)$	3-D 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	$(n_{pts}, 1)$
2-D Vector	$(n_{pts}, 2)$
3-D Vector	$(n_{pts}, 3)$
2-D Tensor	$(n_{pts}, 3)$
3-D Tensor	$(n_{pts}, 6)$

A.1.7 Comparison Residuals

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	$(k, 1)$
Residual Value	ε_1
	ε_2
	$\vdots$
	ε_k

These residual values may be combined in the following form:

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

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

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

A.2 File Formats

A number of file formats are supported for storing and retrieving the above data structures to and from disk. The following file extensions are used for these purposes:

File Format	Recognized Extensions
HDF5	`*.h5`
VTK	`*.vtk`
XML VTK	`*.vtu, *.vts, *.vtr`
Parallel XML VTK	`*.pvtu, *.pvts, *.pvtr`
Text	`*.txt, *.dat`

A.3 Input Files

In Section 1.1, 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 filesystem, and groups are analagous to folders. One important difference, however, is that you can attach supporting metadata to both kinds of objects. The metadata objects are called *attribute*s.

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.

In Cigma you may always provide an explicit path to every dataset, so you have a fair amount of flexibility for how you organize your datasets inside HDF5 files. For example, a typical Cigma HDF5 file could have the following structure.

```
model.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]
            \...
```

Generally, Cigma will only require you to specify the path to a specific field dataset. If a small amount of metadata is present in your field dataset, the rest of the required information, such as which mesh and finite elements to use, will be deduced from that metadata.

MeshID an identifier assigned for use in linking child datasets to their parent mesh.

MeshLocation points to the HDF5 group which contains the appropriate coordinates and connectivity datasets.

FunctionSpace string identifier to determine which shape functions to use for interpolating values inside the element (e.g., tet4, hex8, quad4, tri3, ...).

DatasetType string identifier for classifying the type of data contained in the dataset (e.g., points, connectivity, degrees of freedom, quadrature rules, global quadrature points, global field values).

A.3.2 VTK

Note that while you typically provide a path (or name) for every dataset, this is not necessary when specifying a VTK mesh, since this data is taken from the special Points and Cells arrays, which you cannot rename. However, you will still need to provide a name when referring to the field coefficients, which are assumed to be stored as Point Data in the input VTK file.

For more information, you may want to refer to Visualization ToolKit's File Formats (`www.vtk.org/pdf/file-formats.pdf`) document.

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 Section 1.1. The first line of the file contains the dimensions of the array, just two integers separated by whitespace. The rest of the file, describing the array data, must contain as many numbers as the product of the two dimensions given in the first line. These numbers are formatted as integer or floating point, depending on context, and are all separated by whitespace.

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 Section 1.3.1.

Another point worthy of mention is that arrays must be two-dimensional, so two integers are always expected in the first line. When specifying a one-dimensional array, such as a list of weights corresponding to an integration rule, one of the array dimensions must be 1.

A.4 Output Files

Depending on the particular operation you perform, there are three different kinds of output files. The output format for residuals consists simply of a list of scalars for each element in the discretization, stored using the legacy VTK file format in the Cell Data section of the output file. Note that this output consists of the squared values of the local residuals, so further post-processing will be necessary.

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] R. Cools, *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.