SPHERLS User Guide 1.0

Chris Geroux

March 21, 2018

Chapter 1

Introduction

This manual is primarily designed to be used to get a user up and running quickly. It also includes some information on how to go about modifying and developing SPHERLS. The reference manuals are useful for determining specific information about variables, functions and classes.

1.1 Overview

SPHERLS stands for Stellar Pulsation with a Horizontal Eulerian Radial Lagrangian Scheme. There are three components to SPHERLS: SPHERLS itself which does the hydrodynamics calculations, SPHERLSgen which creates starting models, and SPHERLSanal which is able to manipulate the output files. All three components have their own reference manuals where details of the various classes, functions, and variables are described, however the details on how to use these three components is described here. For the most part SPHERLS and SPHERLSgen are used directly, while SPHERLSanal is primarly used indirectly through the use of python scripts.

SPHERLS calculates the radial pulsation motions together with the horizontal convective flow of stars. The radial pulsation can be described by a radial grid velocity, moving the grid inward and outward with the pulsation. The movement of the grid is defined by the motion required to maintaining the mass in a spherical shell through out the calculation. This motion is determined so that it will change the volume of the shell so the newly calculated density when multiplied with the new volume will produce the same shell mass for all times. The total motion of the stellar material is simply the combination of the three velocity components and the grid velocity. The convective motion is the radial velocity minus the grid velocity, combined with the theta and phi velocities. This is because the grid velocity describes the

bulk motion of the pulsation so subtracting it out leaves only the convective motions.

SPHERLS solves the normal hydrodynamic equations of, mass, momentum, and energy conservation. The form of the mass equation, momentum conservation, and energy conservation are:

$$\frac{dM}{dt} + \oint_{\mathbb{S}} (\rho\vec{v}) \cdot \hat{n} d\sigma = 0 \tag{1.1}$$

$$\frac{\partial\vec{v}}{\partial t} + (\vec{v}\cdot\nabla)\vec{v} = -\frac{1}{\rho}\nabla P + \nabla\cdot\tau - \nabla\phi \tag{1.2}$$

$$\frac{\partial E}{\partial t} + (\vec{v}\cdot\nabla)E + P\frac{d\mathbb{V}}{dt} = \epsilon - \frac{1}{\rho}\nabla\cdot\ F + \frac{D_t\mathcal{K}^{3/2}}{L} + \frac{1}{\rho}\nabla\cdot\left(\frac{\mu_t}{\rho P_{rt}}\nabla E\right) \tag{1.3}$$

where τ is the stress tensor for zero bulk viscosity, E is the specific internal energy, $\mathbb{V}$ is the specific volume, and F is the radiative flux. In addition to these conservation equations an equation of state is needed, in this case the OPAL equation of state and opacities, and the Alaxander opacities at low temperatures are used. The equation of state tables are functions of density and temperature, and produce the energy, pressure, opacity, and adiabatic index of the gas for a given temperature and density. In adiabatic calculations, it is also possible to use a γ-law gas equation of state and in that case an initial energy profile must be specified.

The simulation grid is broken up into two main sections, the 1D region towards the center of the star, the multi-dimensional region towards the surface. The inner part of the multi-dimensional region solves all the conservation equations explicitly, in that the new values for the conserved quantities are directly calculated from the information in the previous time step. In the outer parts of the multi-dimensional region the energy conservation equation is calculated semi-implicitly, which means that the new values are dependent on the new values averaged with the old values to correctly time center the equation. This semi-implicit energy conservation equation can be perturbed and linearised producing a set of linear equations the size of the region being solved implicitly. The solution of these linear equations provide corrections for the temperature. The corrected temperature can then be used to solve for a new set of corrections this processes is repeated until the value of the new temperature converges (the size of the corrections is smaller than some specified amount). The equation of state is a function of temperature and not energy which is why the temperature is perturbed and not the energy. This set of equations for the temperature corrections are solved using the PETSC library.

More info to be included here:

- mention that SPHERLS is parallelized
- Different ways in which SPHERLS can be used, 1D,2D,3D, Adiabatic,Non-adiabatic, implicit, debugging options/test
- mention very quickly how SPHERLS can be used (e.g. give very quick example of running SPHERLS)

1.2 Program Flow

This should give a rough idea of how SPHERLS works, but should be kept quite high level

- Describe the grids
- The order of calculation
- When parts of the grid are updated
- when models are dumped

Chapter 2

Installing SPHERLS

There are a number of optional and required third party libraries used by SPHERLS. This section will describe how to install these third party libraries as well as SPHERLS. However, as the third party libraries often contain their own installation documentation the following section should be thought of as the minimal set of options and steps required to install those third party libraries for use with SPHERLS and not a complete documentation of how those third party libraries should be installed in general. One should consult the third party library's documentation for the full details of their installation.

There are three flavours of the SPHERLS distribution package available: one which includes all third party libraries, one which includes only required third party libraries, and one which does not include any third party libraries. It is recommended to use the package including all third party libraries. This version helps to ensure the greatest functionality of SPHERLS and also that these libraries are compatible both with SPHERLS and the below installation instructions. Other version of the distribution package can be desirable to reduce download size, if the third party libraries are already installed, or if you wish to try out newer versions of the third party libraries and do not want to download the additional data. The included third party libraries can be found under the `lib/` directory under the root package directory.

A few words on installation before we get into the details of the specific third party package installations. In order for the SPHERLS configuration script to find the required libraries and include files they need to be installed in standard locations or have their location added to standard environment variables. Generally nonstandard include paths should be added to the `CPATH` environment variable and library paths should be added to the `LIBRARY_PATH` variable. Multiple paths in these variables should be separated by a ":". In the bash shell this can be done with a command like

`export CPATH="<path to package include director>:$CPATH"` or similarly for the `LIBRARY_PATH` variable.

2.1 Requirements

SPHERLS requires gcc/g++, OpenMPI, and PETSc to compile and run at all. It also makes heavy use of Python scripts some of which depend on Cython. So while you technically can run the basic binaries without Python and Cython in practise you really won't want to use SPHERLS without these.

The below set of instructions do not explain how to install the GCC (GNU Compiler Collection) but assume that your system already has this installed. If you are on a large HPC cluster you likely also have OpenMPI already available to you. Feel free to use the site OpenMPI installation, but if you find later tests produce unexpected results or you have difficulty building SPHERLS you may wish to try using a per-user install of an OpenMPI version known to work with SPHERLS and building with that.

The below libraries can be optionally installed but will greatly enhance the usefulness of SPHERLS and are greatly recommended. The full list of dependencies of these third party libraries is not included, but some library dependencies have been noted when I have specifically had to install these dependencies on various machines.

python python scripts are supplied for various analysis and convenience operations. The included python scripts were used and written for Python 2.X series so will not work with Python 3.X series, though with a little effort these scripts could be updated to work with both, however this has not yet been done so it is best to stick with the Python 2.X series. Having python is highly recommended and will make using SPHERLS far more enjoyable. Many SPHERLS Python scripts use the following Python libraries:

- **numpy** numerical python, fast array operations
- **matplotlib** for creating plots
- **Cython** needed to build evtk, and also to allow Python scripts access to the equation of state and opacity C++ classes used in SPHERLS.
- **evtk** a python library to make writing vtk files easier, the SPHERLS build process actually makes this for you if Cython support is enabled.

scipy for interpolating in equation of state and opacity tables when making new rectangular equation of state and opacity files. This is not something that is often required so skipping installing `scipy` might not be a problem. In my experience it often turns out to be one of the more difficult libraries to install as it has many dependencies due to the large amount of available functionality from this module.

fftw3 used for frequency analysis by SPHERLSanal. Not strictly required.

hdf4 for converting model dumps to hdf4 file format for visualizing, used by SPHERLSanal and some Python scripts (e.g. `make_hdf2.py`). Not required unless you want to make hdf4 files.

jpeg needed by hdf4 library.

zlib needed by hdf4 library.

szlib possibly needed by hdf4 library.

pdflatex used to create documentation from src/doc/ latex files. Use `make docs` to build documentation. Note that the `make docs` command requires an additional option to be set during the configuration process `--enable-make-docs`. Also, the documentation does not need to be made unless it has been modified as all the documentation has already been made and included in the repository and distribution packages.

2.2 Installing OpenMPI

- Download OpenMPI from the open mpi site or use the version in `libs/` directory. Versions 1.6.3, 1.6.5, and 1.8.8 have been tested and known to work with SPHERLS. In later versions of OpenMPI the C++ bindings have been deprecated but in 2.X versions they are still mentioned in the documentation until version 3.X. However, attempts to build with 2.1.1 OpenMPI have failed because of a lack of the MPI namespace.
- `./configure --prefix= <path-to-final-location-of-install>`
- `make`
- `install`
- finally the bin, man, and lib paths will need environment variables set:

```
export OPENMPI=<path-to-final-location-of-install>
export PATH=<path-to-final-location-of-install>/bin:${PATH}
export MANPATH=<path-to-final-location-of-install>/man:${MANPATH}
export LD_LIBRARY_PATH=<path-to-final-location-of-install>/lib-
:${LD_LIBRARY_PATH}
```

2.3 Installing PETSC Library

Version `petsc-lite-3.1-p8`, has been tested to work with SPHERLS. `petsc--lite-3.2-p7` is known to be incompatible, which as of this writing is the current version of the PETSc library. At some point in the future support for the newer version of the library maybe added. The below commands will install PETSc into your home directory.

- Download PETSc library, from the PETSc website or get it from the `libs/` directory in the spherls package
- Then untar and unzip it with `tar -xzf petsc-3.1-p8.tar`
- To install the library change into the directory made when you extracted the archive and type the following commands:

  1. `./configure PETSC_DIR=$PWD --prefix=<path-to-final-location-of-insta --with-c++-support --with-c-support --with-shared --download-f-blas-lapack=1 --with-x11=no`

     where `$USER` is the environment variable corresponding to your username. Note that the `PETSC_DIR=$PWD` needs to come first. Also `--download-f-blas-lapack=1` only works if there is a fortran compiler present, which isn't strictly needed otherwise. Using `--download-c-blas-lapack=1` seems to work when a Fortran compiler isn't available.
  2. `make all` Often at the end of the configuration stage the configuration script will give the command to make the library. One should use this over the above, if given.
  3. `make install` as with the `make all` the makeFile will also likely tell you the command needed for the installation, which should be used over the one provided here.

4. `make PETSC_DIR=<path-to-final-location-of-install>/lib test` will test the code. Again the install process will give you a more specific command that you should use over the one given here.
5. `export CPATH=<path-to-final-location-of-install>/include:$CPATH` to add the PETSc include path to your CPATH environment variable.
6. `export LIBRARY_PATH=<path-to-final-location-of-install>/lib:$LIBRARY_PATH` to allow compilers to find the PETSc libraries during compilation.
7. `export LD_LIBRARY_PATH=<path-to-final-location-of-install>/lib:$LD_LIBRARY_P` so that runtime executables can find the PETSc shared libraries.

2.4 Installing Cython

Cython allows you to build some extra python modules to use in creating visualization output such as vtk files for use in VisIt, and for making the equation of state routines available to the python analysis scripts. Finally there are dependencies on cython in scripts which run some simple tests such as checking that results are reproduced and that restarting a calculation works correctly. Therefore it is highly recommended that cython is installed. There is a version tested to work under the `libs/` directory.

- `tar -xzf Cython-0.19.1.tar.gz`
- `cd Cython-0.19.1`
- `python setup.py install --prefix=<path-to-final-location-of-install>`
- This will require you adding the line `export PYTHONPATH=<path-to-final-location-of-inst` `lib/python2.7/site-packages/:$PYTHONPATH` to `.bashrc` so that python can find your installation of Cython, the exact path may depend on your version of python and of course where you install Cython.

2.5 Installing Python Modules

In this section I will just briefly give some suggests about how to install some of the additional python modules I mentioned above. One can often install python modules into home directory locations with the `pip` tool. For example

```
pip install --install-option="--prefix
=<path-to-install-directory>" numpy
```

If the specified module, in this case `numpy` is already present on the current machine, an upgraded version can be installed in the users home directory if desired with the `--upgrade` option. Another helpful option is the `--user` option which was required when installing the `matplotlib` module, which appears to make `pip` aware that you do not have root access. In order to have python pick up these home directory installs one must add to the `PYTHONPATH` environment variable with something like

`export PYTHONPATH=<path-to-install-directory>`
`/lib/python2.7/site-packages:${PYTHONPATH}`.

The exact details of this path to add will depend on the version of python you have available to you on your system, so check the actual paths were things have been installed.

2.6 Installing FFTW Library

- Download the FFTW Library from the FFTW website. Or there is a version available in the `lib/` directory. Version fftw-3.2.2, fftw-3.3.1, and fftw-3.3.3 have been tested to work with SPHERLS.

- Then untar and unzip the FFTW tar ball with something like `tar -xzf fftw-3.2.2.tar.gz`

- To install the library change into the directory made when you extracted the archive and type the following commands:

  1. `./configure --prefix=<path-to-final-location-of-install>`
  2. `make`
  3. `make install`
  4. `export CPATH=<path-to-final-location-of-install>/include:$CPATH` to add the FFTW include path to your CPATH environment variable.
  5. `export LIBRARY_PATH=<path-to-final-location-of-install>/lib:$LIBRARY` to allow compilers to find the FFTW library while linking.
  6. `export LD_LIBRARY_PATH=<path-to-final-location-of-install>/lib:$LD_LI` to allow executables to find the FFTW libraries at run time.

2.7 Installing HDF4 Library

Building these libraries requires gfortran and the jpeg library. If using the SPHERLS distribution package including third party optional libraries, version 4.2.8 of the HDF4 library is available in the `libs` subdirectory. Otherwise this library can be downloaded from the hdfgroup website. The most recent version tested to work with SPHERLS is version 4.2.8, version 4.2.7 has also been tested to work. To build the version included with the SPHERLS distribution package use the following set of commands:

- `cd <SPHERLS package directory>/libs`
- `tar -xzf hdf-4.2.8.tar.gz`
- `cd hdf-4.2.8`
- Run `./configure --prefix=<path-to-final-location-of-library> CFLAGS="-fPIC" CXXFLAGS="-fPIC"`.

  Note that for the hdf library the `<path-to-final-location-of-library>` should be set if doing a global install as well as doing a per user install as the default install directory is inside the build directory, which is not what is usually wanted. So if doing a global install it will probably want to be set to `/usr/local` or for a per-user install `/home/$USER`.

- `make`
- `make install`
- `export CPATH=<path-to-final-location-of-install>/include:$CPATH` to add the HDF include path to your CPATH environment variable.
- `export LIBRARY_PATH=<path-to-final-location-of-install>/lib:$LIBRARY_PATH` to allow compilers to find the HDF libraries when linking.
- `export LD_LIBRARY_PATH=<path-to-final-location-of-install>/lib:$LIBRARY_PATH` to allow executables to find the HDF shared libraries at run time.

The above commands have only been tested with the version including with the SPHERLS distribution package other versions may require slight modifications to the above commands but should be reasonably similar.

2.8 Installing evtk

The SPHERLS build process actually builds evtk for you, so you probably do not need to do these steps unless for some reason you wish to install evtk yourself. As mentioned previously, evtk is a python module which allows one to create vtk files.

- `tar -xzf evtk.tar.gz`
- `cd evtk`
- `python setup.py install --prefix=<path-to-final-location-of-install>`
- This will require you adding the line `export PYTHONPATH=/home/-$USER/lib/python2.7/site-packages/:$PYTHONPATH` to `.bashrc` so that python can find your installation of evtk, the exact path may depend on your version of python also you only need to have this line in your `.bashrc` file once

2.9 Installing SPHERLS

After all the required and optional third party libraries are installed and the appropriate environment variables have been set building and installing SPHERLS should involve running these commands:

- `./configure --prefix=<path-to-final-location-of-install>`
- `make`
- `make install`

If you have the raw git repository you may do the following to recreate the `configure` file. However there is a version of this file included in the repository, so the above steps should also work.

- `autoreconf --install`
  provided you have the gnu build tools installed, this makes the configuration script
- `./configure --prefix=<path-to-final-location-of-install>`
- `make`
- `make install`

- `export PATH=<path-to-final-location-of-install>/bin:$PATH` to add the SPHERLS bin path to your `PATH` environment variable.
- `export PYTHONPYTH=<path-to-final-location-of-install>/bin:$PATH` to add the SPHERLS bin path to your `PYTHONPYTH` environment variable so that the python scripts can find needed python modules.

Common problems which result during the execution of the above commands are: not finding the installed third party libraries. This often results from not having the correct settings for things like `LIBRARY_PATH`, `CPATH`, and `LD_LIBRARY_PATH`. It is also possible to specify more options to the linker by setting the `LDFLAGS` environment variable by including the location of library search paths and or names. There are also options for turning on/off features. Run the command `./configure -h` to get a list of the options. For example if you don't have hdf4 libraries and don't want to use that functionality including the option `--disable-hdf` will build without hdf4 capabilities.

Chapter 3

Using SPHERLS

Once you have built SPHERLS and set up the necessary environment variables the first thing to do is run some simple tests to make sure it will run as expected. There are python scripts which you can use to run these tests and are found in the `bin` directory where SPHERLS was installed. Note that you will need to have built with Cython support enabled to do this step, so it is highly recommended that you have installed Cython and built SPHERLS with it enabled. There are two test scripts which can be run `test_restart.py` and `test_calculation.py`. If you have not built with Cython support you will likely get the error `No module named evtk.hl` when running either of the test scripts. These will be found under your SPHERLS install directory in the `bin` directory. They should also be findable from your path as they are executable and we have added the `bin` directory to your `PATH` environment variable. Python scripts are distinguished from compiled executables with their `.py` extension. Lets examine what the `test_restart.py` script does in detail first.

This script and many other scripts have many optional and required arguments which can be specified. Nearly all scripts can be executed with the `-h` option to display a brief description of the script and how to use it, as well as listing available options. I would suggest running the `test_restart.py` script with the `-h` option to see these available options. The script help says that it tests restarts of the application to ensure that it produces the same output as if there had not been a restart. Also the `-k` option allows us to keep all the temporary output this script creates while running the tests. This script actually runs SPHERLS several times and compares output of the different runs. We will examine the steps this script takes in order to illustrate some aspects of working with SPHERLS. The output that is generated when you run `test_restart.py -k` should look something like that below, but with different paths reflecting where SPHERLS was installed on

your system.

```
 1:making a temporary directory, "./test3DNARestarts" to store test data ... SUCCESS
 2:changing into directory "./test3DNARestarts" ...
 3:making "SPHERLS.xml" ...
 4:running "/home/cgeroux/apps/spherls/bin/SPHERLS" for 2 time steps ... SUCCESS
 5:remaking "SPHERLS.xml" ...
 6:combining binary dump "./RestartTest2_t[0-*]" for restart ... SUCCESS
 7:restarting "/home/cgeroux/apps/spherls/bin/SPHERLS" for 1 time step from dump "RestartTest1_t00172487" ... SUCCESS
 8:combining binary files for dump "./RestartTest2_t[0-*]" ... SUCCESS
 9:diffing last dumps ... SUCCESS
10:making a temporary directory, "./test2DNARestarts" to store test data ... SUCCESS
11:changing into directory "./test2DNARestarts" ...
12:making "SPHERLS.xml" ...
13:running "/home/cgeroux/apps/spherls/bin/SPHERLS" for 2 time steps ... SUCCESS
14:remaking "SPHERLS.xml" ...
15:combining binary dump "./RestartTest2_t[0-*]" for restart ... SUCCESS
16:restarting "/home/cgeroux/apps/spherls/bin/SPHERLS" for 1 time step from dump "RestartTest1_t00137887" ... SUCCESS
17:combining binary files for dump "./RestartTest2_t[0-*]" ... SUCCESS
18:diffing last dumps ... SUCCESS
19:making a temporary directory, "./test1DNARestarts" to store test data ... SUCCESS
20:changing into directory "./test1DNARestarts" ...
21:making "SPHERLS.xml" ...
22:running "/home/cgeroux/apps/spherls/bin/SPHERLS" for 2 time steps ... SUCCESS
23:remaking "SPHERLS.xml" ...
24:combining binary dump "./RestartTest2_t[0-*]" for restart ... SUCCESS
25:restarting "/home/cgeroux/apps/spherls/bin/SPHERLS" for 1 time step from dump "RestartTest1_t00000001" ... SUCCESS
26:combining binary files for dump "./RestartTest2_t[0-*]" ... SUCCESS
27:diffing last dumps ... SUCCESS
```

Lets go through the steps the script executed. After the script creates a new directory to run a test in and changing into it (lines 1-2), it creates a `SPHERLS.xml` file (line 3). This file contains all the settings to perform a calculation with SPHERLS. This file has settings such as where the input file is located, where the output should go, and how many time-steps SPHERLS should take. There are also many more settings contained in this file and it will be discussed in more detail in section 3.2. The script then runs SPHERLS for 2 time-steps (line 4), and then remakes the configuration file (line 5) so that the calculation will resume at the second time step and run for one time step. This then allows the output from these two runs to be compared, as they should be at the same time, the output should be identical.

Before SPHERLS is run the second time, the output from the first run is combined into one file (line 6). SPHERLS uses domain decomposition to split up the computational work across multiple separate cpus, this means the physical grid representing the star is split into local grids on each cpu. Each cpu is responsible for computations only on that local grid. As a result the simplest and most efficient way to write out the grid state is to have each cpu write out its local portion of the grid to a file. To have a single file containing the whole grid then requires these separate files to be combined. This combing step is required before a restart occurs. SPHERLS only reads these combined files as input and only generates the distributed files as output. If you look at the directory `test3DNARestarts` created by the script you will see the configuration files `SPHERLS.xml` and `SPHERLS_first_run.xml`, which correspond the first and second run of SPHERLS. You will also see a `log.txt` file which contains output from the `test_restart.py` script as

well as standard output actually generated from the SPHERLS executable. Blocks (or even single lines) of output from SPHERLS are preceded by the file and line number in the source code where that output was generated (e.g. `src/SPHERLS/main.cpp:main:144:0:`) this helps to distinguish it form the output generated by the test script. It is also handy to locate the code which generated a given output. You can also see output from python scripts which combine the output files, which have lines starting with `__main__:combine_bin_files:`. Note that this output is actually generated from the function `combine_bin_files` in the Python script `combine_bins.py`. The `test_restart.py` script executes the `combine_bins.py` script through the command line to combine the files.

Once the output files from the first run are combined a second run of one time step is initiated (line 7). Finally the output from the second run is combined (line 8) and compared to the output from the first run (line 9). This whole process is repeated to test restarts for runs of 2D and 1D models, since the test for 3D models has just been done.

For the most part, the `test_restart.py` script is simply issuing commands on the command line to perform these tests. These commands are a combination of usual linux commands such as `mkdir` and `cd`, and SPHERLS specific commands. The command it uses to actually run the SPHERLS executable is `mpirun -np <number-of-cores> SPHERLS`. This is just the usual `mpirun` command with the `SPHERLS` executable as the argument. The SPHERLS executable expects to read settings from the `SPHERLS.xml` configuration file, and will fail with an error message such as

```
XML Parsing error inside file 'SPHERLS.xml'.
Error: File not found
At line 0, column 0.
src/SPHERLS/main.cpp:main:275:0:src/xmlFunctions.cpp:openXMLFile:268: error parsing the file "SPHERLS.xml" possibly no global "data" node
```

if it doesn't find that file in the current working directory.

The `test_restart.py` script also runs a command `combine_bins.py ./RestartTest1_t[0-*]` in the test directory to combine the distributed files into combined files. The distributed files have a `-#` appended where `#` is the processor rank. In the case of this script SPHERLS has been run with 4 processors and thus you see files with `-0`, `-1`, `-2`, and `-3` suffixes. The files without this suffix are the combined files. If you look at the file sizes you will see that these combined files are actually slightly smaller than the combined size of distributed files. This is because the distributed files contain some extra information to describe which portions of the grid that processor worked on and some duplicated meta data. The number to the left of the suffix (of eight digits preceded by a `_t`) is the number of time steps from some zero initial time. These tests actually run from models which have been run for

a significant time. In the case of 2D and 3D calculations this ensures that the tests include well developed convection, which will exercise all the terms in the conservation equations in the tests. The file name argument specified for `combine_bins.py` script given above does not match a specific file name, that is because it matches all the files names. The `[0-*]` specifies that the script should run over all distributed files with time step count between 0 and "*". "*" is a special case indicating that it should loop until no more files are found with the same matching path. This way of specifying input files can be used to run a command on a subset of output files in an output directory and can be useful if you only want to process some part of the output.

Finally to make the comparison the `test_restart.py` script calls the `diffDumps` function from the `diffDumps.py` script. The same result can be obtained by manually running this function on the command line with something like

`diffDumps.py RestartTest2_t00172488 RestartTest1_t00172488`.

This script compares cell-by-cell each variable in the dump file to check for differences. If the relative difference of the variable is larger than is allowed (the default is 5×10^{-14}) it will declare the two files as different and report the difference. It also uses a threshold value that variables must be larger than in order to be compared (the default is 10^{-17}). This avoids division by zero in the case of the velocities, as other variables are typically several orders of magnitude larger than the 10^{-17} threshold.

The `test_calculation.py` script can be run in a similar way to the `test_restart.py` script. This time instead of checking that restarts work as expected it compares the results of the calculation to a set of reference calculations. This is useful for ensuring a new installation produces the same results as a previous installation. It is also useful when making changes to the code to verify that changes not expected to affect the results do not. If changes are made to the code that one expects to change the results of the calculation these reference calculations can be re-created by passing this script the `-r` option. The precision at which the calculations are compared can be adjusted with the `-p` option and specifying a relative error between calculations. Also the `-t` option can be used to specify a minimum threshold size which values must be above before they are compared. This is particularly useful for avoiding comparison of very small velocities. It is possible to reproduce the reference calculations to a high degree of precision but one must take special care to ensure that all the libraries are built in the same way as the original reference calculations and similar compilers used. It is

often difficult to ensure all these conditions are met so it shouldn't be too surprising if running the `test_calculation.py` fails, especially on the 3D calculations. If this is the case I would recommend inspecting the reported differences and ensuring that they aren't too big, maybe say around 1e-7. At this point it would probably be good to create new reference calculations by re-running `test_calculation.py` with the `-r` option.

By examining these test scripts you have been introduced to the basic work flow of SPHERLS including how SPHERLS reads settings from a configuration file, how it outputs distributed files and only reads combined files, how to combine distributed files, and how to compare combined output files. You have also been introduced to how to get more information about specific Python scripts, and how to tell Python scripts output file ranges to process. In the next section the generation of the initial starting model will be described.

3.1 Generating a Starting Model

As was mentioned in the Overview (§ 1.1) the executable `SPHERLSgen` generates an initial model to be used as input for SPHERLS. It is used in a similar way to SPHERLS in that it requires an input XML file describing the starting models to create. As a starting point to creating a configuration file for `SPHERLSgen` look in the `templateXML` folder under the `docs` folder. The `templateXML` folder contains various XML settings files used by different scripts and executables. Note that there is a `SPHERLS_reference.xml` file, which is the reference file for the settings required to run the `SPHERLS` hydrodynamics executable. We will describe the settings to run `SPHERLS` later in more detail in § 3.2. For now let us focus on the `SPHERLSgen_reference.xml` file.

If you are not familiar with XML files we will give a very brief overview of their basic structure. An XML file is composed of various elements also sometimes referred to as nodes. An element is defined with a construct like `<exampleTag> </exampleTag>` where `<exampleTag>` and `</exampleTag>` are each separately referred to as tags and together define the start and end of the `exampleTag` element respectively. In general the XML format allows you to call the elements what ever you like, however the names of the elements have meaning to the programs which parses them telling the program what information contained within that element. Between the element tags there can be text (e.g. `<exampleTag>Some text</exampleTag>`) which can be interpreted as text, numbers, or logical flags. Elements can also contain other nested elements (e.g. `<exampleTag><innerTag></innerTag></exampleTag>`.

Elements can also have attributes for example `<tag name="Bob">`, where `name` is an attribute of this element and has the value "Bob". XML files can also contain comments with the format `<!-- This is a comment -->` with `<!--` starting the comment and `-->` ending the comment. These comments can span multiple lines. Finally XML files must have a single element at the top level, often refereed to as the root element or element.

The basic structure of a SPHERLSgen XML file is as follows:

```
<data>

  <!-- constants -->
  <G>6.67259e-08</G><!-- gravitational constant [cm^3 g^-1 s^-2]-->
  <R-sun>6.958e+10</R-sun><!--[cm]-->
  <M-sun>1.9891e+33</M-sun><!--[g]-->
  <L-sun>3.839e33</L-sun><!--[ergs s^-1]-->
  <sigma>5.6704e-5</sigma><!--Stefan-Boltzman constant
    [ergs s^-1 cm^-2 K-4]-->

  <model type="stellar">
    <output>... </output>
    <EOS>... </EOS>
    <periodic>... </periodic>
    <dimensions>... </dimensions>
    <velocityDist>... </velocityDist>
  </model>

  <model>... </model>
</data>
```

Above only the highest level elements have been included from a `SPHERLSgen.xml` file in order to briefly describe the overall structure of the file. The root element is the `data` element. Inside this element can be one or more `model` elements. There are also elements inside the `data` element to set the values of various physical constants. Each `model` element requires a `type` attribute describing what type of model should be created. The elements inside the model give more specific details about the model and we will briefly go over the information contained in them. The `output` element contains information about how the model should be output, for example its file name and whether it should be binary or ascii. The `EOS` element contains information about the equation of state to use. The `periodic` element indicates which directions of the grid should have periodic boundaries, in nearly all cases

models are periodic in the two angular directions and not periodic in the radial direction. SPHERLSgen is smart enough to know about these defaults and does not require the user to specify periodicity of the grid so this element can be omitted without issue. The `dimensions` element describes the grid, by specifying how the radial independent variable M_r should vary, as well as number of angular cells and their spacing. Finally the `velocityDist` element describes what the initial velocity distribution should be. This element allows one to set what the radial velocity profile should be, for example the linear fundamental mode or the linear first overtone mode (note that the linear modes are not exactly the same as the non-linear modes but are often reasonable starting points). This element also allows one to add perturbations on top of the radial velocity profile in the angular directions with the basic idea being to speed up the onset of convection.

The `SPHERLSgen_reference.xml` template file contains all available settings and options, even ones which are not required. Many of which have some brief comments giving some rough idea of what the settings are for. Lets walk through the steps to create a new stellar model, most of which just deals with creating the `SPHERLSgen.xml` file and the settings which are required. We will also discuss what values and settings make the most sense and try to give an idea of the sort of settings that one might like to use.

Start by copy the `SPHERLSgen_reference.xml` template file to a working directory and rename it to `SPHERLSgen.xml`. As mentioned previously, the `SPHERLSgen` executable will look for a configuration file named `SPHERLSgen.xml` in the current working directory for settings to create new models. There are two models in the newly created `SPHERLSgen.xml` file, delete the `sedov` type model. This model type creates a Sedov blast wave initial model, used for testing the results of the code, to run such a model certain pre-processor directives in the code must be defined. Running a Sedov test problem is beyond the scope of this section, instead we will focus on the `stellar` model type. The first setting element is the `output` element. The first element within the `output` element is the `timeStepFactor` element. It is a simple multiplier used on the Courant time step. A value of 0.25 is a good default since SPHERLS handles some terms in the conservation equations explicitly and others implicitly, the more stringent constraint on the explicit conservation equations of the Courant condition should be obeyed, requiring a Courant number, or `timeStepFactor` less than 1.0. From trial and error testing a value of 0.25 has been found to work well. The next element is the `fileName` element, as you might have guessed, it contains the name of the output file. This element can also include paths, both relative (starting with `./`) and absolute (starting with `/`), as well as the output file name. The format of the output file name is unconstrained, meaning there is no special meaning

associated with the output file name. However, as the hydrodynamics program SPHERLS appends a `_t########` to the end of each output file name where the `#` indicate the 8 digits representing the number of time steps taken from the starting model, it may be preferred to follow this convention here also. Next the `binary` element can be either `true` or `fales` and indicates that the output should be binary (`true`) or ascii (`false`). For the most part one will wish to output in binary format as it is more compact, however in some cases (e.g. debugging) it may be helpful to output the data in a human readable format. The `writeToScreen` element is another `true/false` option which indicates whether the model should be written to the screen. If `true` SPHERLSgen will output the 1D zoning as the other dimensions, when present, are just copies and thus all the same, except if there are velocity perturbations. This can be handy for quickly checking that a model has been created with the necessary number of zones in the hydrogen ionization region ($T = 10^4$ K). The final element inside the `output` element is the `precision` element, which indicates the amount of precision written out in the ascii file. The `precision` setting has no impact on binary output.

The next main element is the `EOS` element which describes the equation-of-state and the opacity input as well as the luminosity, effective temperature and thus radius of the star. There are two types of equation of state that can be used `gammaLaw` and `table`. The `gammaLaw` eos uses a simple gamma-law gas equation of state and was used primarily during testing with the adiabatic version of the code. In the adiabatic version of the code the temperature is not tracked instead the energy is used as the radiative terms in the energy equation are omitted and also with it the dependent on temperature, thus there is also no need for an opacity as again it only turns up in the radiative terms in the energy equation. The energy in this case comes from a 1D energy profile in mass from the stellar evolution code ROTORC. Lets use the `table` eos type so remove the `gammaLaw EOS` element. The first element in the `table EOS` element type is `T-eff` which indicates what the effective temperature of the new model should be, for RR Lyrae stars reasonable values between about 5700–6900 K. For this walk through lets pick 6800 K. RR Lyrae variable stars are a good starting point to try and simulate and have already been explored with this code and it is very likely you should have success simulating those types of stars. Other stars should be possible to some extent though one has to think carefully about potential issues which could arise, such as resolving the necessary phenomena well enough with the grid. Then `L` element indicates the luminosity of the star for an RR Lyrae star 50.0 $L_\odot$ is a good value to start with. The next element is `eosTable` this is a path to a combined eos and opacity table. These tables are derived from the OPAL eos and opacity tables combined with the Alexander & Ferguson

low temperature opacity tables. The file name already present is a good place to start. The number after the characters `Y` and `Z` indicate the helium and metal mass fractions of the table respectively. These tables are finely and smoothly interpolations of the original tables onto a rectangular grid to allow for fast linear interpolation.

The next two elements `tolerance` and `maxIterations` indicate how accurately and how many iterations are allowed to solve for the temperature and density of a shell, using the Newton-Raphson method to solve the two couple partial differential equations resulting from the momentum and energy equations when imposing hydrostatic equilibrium. In the case of the gamma-law gas no such iterative method is needed as there is only a single momentum equation to solve, thus these variables are dependent on which equation of state is chosen.

The `periodic` element tells the code which directions are periodic, however, in almost every case one wants to have periodic boundaries in the two angular directions, and non-periodic boundaries in the radial direction. Here (`x0`, `x1`, `x2`) correspond to the $(\hat{r}, \hat{\theta}, \hat{\phi})$ directions and `0` indicates non-periodic boundary conditions, and `1` indicates periodic boundary conditions. As a single setting for the periodic boundary conditions is so common, the code will actually set the values you want automatically, so the `periodic` element can be deleted.

The `dimensions` element contains all the information needed to define the grid. The `radIndepVar` includes information about how to zone the radial independent variable M_r. `M-total` gives the total mass of the star in solar masses. `M-delta-init` indicates the step in mass of the first zone at the surface in solar masses. The node `M-delta-picking` describes how the zoning should change moving inward from the surface. It was allowed to have different methods for picking the zoning to perhaps allow for the code to choose the zoning based on the steepness of the temperature gradient for example. However, only a `manual` method has actually been implemented, thus the `type` should always be set to `manual`. With this method one can set the rate of change of the mass spacing. For example adding an `M-delta-delta` element with a value of 0.1 indicates that the mass spacing of the next zone in should be 10% larger than the previous zone. `M-delta-delta` have two attributes `stopType` and `stopValue`. The `stopType` attribute indicates if the value to stop at, indicated by `stopValue`, should be either a radius value or a temperature value respectively. The stop values are in units of K and cm for temperature and radius stop types respectively. Once a stop value is reached the rate of increase of the mass zones is changed to the next deeper stop value. The ordering of the `M-delta-delta` elements is important as later lines should correspond to deeper parts of the star. Once the last `stopValue`

is reached the model is completed. Choosing the zoning in this way allows one to adjust the zoning to resolve the ionization zones while not using too many zones deeper in the star where it is not necessary.

The `alpha` parameter indicates the amount of extra mass to include above the outer boundary as a boundary condition. The density at the surface of the star is some non-zero value and actuality the star should extend out until the density is zero however simulating extremely low density regions is difficult, for one reason because the equation of state and opacity tables do not go down to such extreme low densities, and it is also numerically difficult to ensure that such small values (on the order of 10^{-15}) never go below zero. Thus to simulate this extra mass we add a bit of mass at the surface as a boundary condition.

The final element describing the radial independent variable is `num-1D`. This indicates the number of zones at the center of the star which should be treated in 1D. RR Lyrae stars have no convection in their cores, simulating well below the convective region is only important for obtaining the correct periods. In addition, simulating to greater depths in 2D and 3D means that the angular zoning will shrink considerably at smaller radii as the sound speed increases giving rise to increasingly strict Courant condition leading to very small time steps. The number of 1D zones near the centre should be as large as possible while ensuring it is deep enough to not interfere with the convective motions in and below the ionization regions. The multi-D and 1D regions are split onto different processors. The boundary condition for the 1D processors grid is the averaged values from the adjacent multi-D region, and the boundary condition for the multi-D region is a copy of the 1D regions values.

The `num-ghost-cells` element should always be 2, unless one changes conservation equations and a wider ghost region is needed. Again as this is a common setting the code has a reasonable default set and this element can be deleted. The elements `num-theta` and `delta-theta` indicate the number and width of zones in the $\hat{\theta}$-direction, while `num-phi` and `delta-phi` indicate the number and width of zones in the $\hat{\phi}$-direction. To create a 1D, simply set the number of zones in the ϕ and θ zones to zero, likewise for a 2D model set the number of ϕ zones to zero.

The final top level element to describe the model is the `velocityDist` element which describes the initial velocity values. There are two main types of `velocityDist` elements `POLY` and `PRO`. The `POLY` type allows one to define the radial velocities by using a polynomial function with multiple terms using a `term` element. Each `term` element has a constant `c` and a power `p` which can be used to create quite general radial velocity profiles. The resultant radial velocity profile will be the value of the polynomial function

resulting from the sum of the given terms with the independent variable being the fractional radius (e.g. $r/R_{\text{surf.}}$). The `PRO` type on the other hand, creates the radial velocity profile from interpolating in data from a file given by the `fileName` element. The velocity profile should be as a function of fractional radius and the velocity values should be normalized relative to the surface velocity. The element `uSurf` will multiply the interpolated velocity values from the normalized radial profile. There are various velocity profiles available from 1D linear calculations of various models of both first and fundamental modes located in the `data/velocity_pro` directory under the SPHERLS root installation directory. The file name gives the parameters of the model for which the profile was created. For example a file with the name `T6900_Y24_Z002_M575_L50_fu.dat` was created for a model with an effective temperature of 6900 K, a Helium and metals mass fraction of 0.24 and 0.002 respectively, with a mass of 0.575 $M_\odot$, a luminosity of 50 $L_\odot$ and for the fundamental model.

Below are some reasonably good settings to create a 6300 K RR Lyrae model:

```
<data>
  <G>6.67259e-08</G>
  <R-sun>6.958e+10</R-sun>
  <M-sun>1.9891e+33</M-sun>
  <L-sun>3.839e33</L-sun>
  <sigma>5.6704e-5</sigma>
  <model type="stellar">
    <output>
      <timeStepFactor>0.25</timeStepFactor>
      <fileName>T6300_20x1_t00000000</fileName>
      <binary>true</binary>
      <writeToScreen>false</writeToScreen>
      <precision>16</precision>
    </output>
    <EOS type="table">
      <T-eff>6.3e3</T-eff>
      <L>50.0</L>
      <eosTable>data/eos/eosNewY240Z0005_wider_finer</eosTable>
      <tolerance>5e-15</tolerance>
    </EOS>
    <dimensions>
      <radIndepVar>
        <M-total>0.7</M-total>
```

```
        <M-delta-init>1E-9</M-delta-init>
        <M-delta-picking type="manual">
        <M-delta-delta stopType="T" stopValue="1e4">1e-01</M-delta-delta>
        <M-delta-delta stopType="T" stopValue="3e6">1.1e-01</M-delta-delta>
        </M-delta-picking>
        <alpha>0.2</alpha>
        <num-1D>50</num-1D>
      </radIndepVar>
      <num-ghost-cells>2</num-ghost-cells>
      <num-theta>20</num-theta>
      <delta-theta>0.3</delta-theta>
      <num-phi>1</num-phi>
      <delta-phi>0.3</delta-phi>
    </dimensions>
    <velocityDist type="PRO">
      <fileName>data/velocity_pro/T6300_Y24_Z002_M575_L50_fu.dat</fileName>
      <uSurf>1.0e5</uSurf>
    </velocityDist>
  </model>
</data>
```

3.2 Running SPHERLS

When the tests were executed at the beginning of this chapter, the test scripts executed SPHERLS for you. Running SPHERLS manually isn't much more work, except for the fact that one must choose what settings to use via the `SPHERLS.xml` file. This works in a very similar way to the `SPHERLSgen.xml` settings file we just described but the settings will be different. Lets proceed in a similar way by copying the reference file under the `docs/templateXML` directory named `SPHERLS_reference.xml` to the same directory where you ran SPHERLSgen to create your initial model renaming it to `SPHERLS.xml`. As with SPHERLSgen, SPHERLS looks for a specific settings file named `SPHERLS.xml`.

The basic structure of the `SPHERLS.xml` is:

```
<data>
  <job>... </job>
  <procDims>... </procDims>
  <startModel>... </startModel>
  <outputName>... </outputName>
```

```
  <debugProfileOutput>... </debugProfileOutput>
  <watchZones>... </watchZones>
  <dedm temperature="2.35e4"></dedm>
  <prints type="normal">... </prints>
  <dumps>... </dumps>
  <eos>... </eos>
  <extraAlpha>0.0</extraAlpha>
  <av>1.4</av>
  <av-threshold>0.01</av-threshold>
  <donorMult>1.0</donorMult>
  <time>... </time>
  <adiabatic>false</adiabatic>
  <turbMod>... </turbMod>
  <implicit>... </implicit>
</data>
```

The `job` element contains information used by the `SPHERLS_run.py` script, in the scripts directory, which will create grid engine submission scripts from these settings and submit the jobs to the queue. This script will crawl all sub directories looking for `SPHERLS.xml` files and create job submission scripts and submit the jobs to the scheduler. As these settings are specific to the particular job scheduler, sun grid engine, I will not go into detail about these settings. However, if one is interested in such an automation script, much insight can be gained by looking at the `SPHERLS_run.py` script, it should be a reasonably straight forward task to modify it to work with the job scheduler for your particular cluster. If you are not using and automated submission method, these can be ignored as the SPHERLS program its self does not use them.

The `procDims` element describes the 3D distribution of processors. It was intended to allow domain decomposition in the three directions and much of the code has been developed with this in mind, however there were some short cuts taken with the moving grid algorithm which require only a decomposition in the radial direction. For many of the calculations performed with the code to date the radial direction has had many more cells than the angular directions. However, if greater angular resolutions are desired allowing for proper domain decomposition in all three directions my help with parallel efficiency somewhat. If one is interested in this they should contact the author of SPHERLS with regard to how one might go about adding full 3D domain decomposition. As it stands, only the `x0` element should be set to a value other than 1, and should be equal to the number of processors used when submitting the job. It should also be pointed out

that SPHERLS requires there to be at least two processors, one to handle the inner 1D region and one or more to handle the outer muli-dimensional region.

The `startModel` is a path to the start model, and `outputName` is the base name for the output model. To the base name for the output model is appended with the time step count suffix, e.g `_t00000000`.

The `debugProfileOutput` element is used for setting an output base file name when writing out extra debug output. This debugging output is turned off in the production version but can be enabled in the `src/SPHERLS/global.h` file by setting `DEBUG_EQUATIONS` to 1. One can see how the output to this debugging file is generated by searching in `physEquations.cpp` which adds various equation terms to the file for use when debugging the various terms of the physics equations. The class `profileData` is defined in `profileData.h` and implemented in `profileData.cpp`. This is the class which is used to gather and output this debugging information, and by examining these files one can see the available methods for `profileData` objects such as the `profileDataDebug` object defined as a member of the `parameters` object.

The `watchZones` element allows one to add multiple zones which should have extra data printed out by specifying the cell numbers by adding sub-elements like the following

```
<watchZone x0="121" x1="0" x2="0"></watchZone>
```

which instructs SPHERLS to output extra information about zone (121,0,0). The watch zones add extra functionality beyond simply inspecting the dump files at that zone in that the output from a watch zone is performed every time step, so it will often be of much higher temporal resolution than the dump files. The file `watchzone.h` and `watchzone.cpp` define the data structure for watchzones which is used simply to keep track of zone numbers, but most of the work of managing the watchzones is taken care of in the various functions defined in `dataMonitoring.h` and `dataMonitoring.cpp` which dictate the information that is output and its format.

The `dedm` element should not be used, it is a legacy setting from some exploration of limiting energy gradients, specificly in the ionization region. In the end it was not used in any publications.

The `prints` element defines how SPHERLS should report information about the run to standard output.

I should really just describe the configuration file here.

I have already given the basics of how to run SPHERLS. However, at some point I should deal with the fact that it might be run in different environments and the various helper scripts designed to work the the Sun grid engine might not work if you are using a different queueing system, and

certainly wont do what you want if you are running without any sort of job scheduler. This however might not be quite the right place or title to discuss that topic.

3.3 Working with Output

3.3.1 Creating Profile Files

3.3.2 Plotting Profiles

3.3.3 Creating 2D Slices

3.3.4 Plotting 2D Slices

3.3.5 VTK Files

Useful for VisIt

3.3.6 HDF Files

Useful for various visualization software, specifically vGeo.

Chapter 4

Modifying & Developing SPHERLS

This section should include

- Basic layout/design of the code
- model output
- data monitoring
- watch zones
- peak KE tracking
- internal/versus external variables
- message passing
- grid layout
- ranges of grids
- boundary regions
- grid updating
- How to document SPHERLS
- Premade test for SPHERLS after modification
- reference calculations
- restart test

- calculation test (if not modifying calcluation part of SPHERLS)
- How to modify SPHERLS
- Common changes
- How to add a new internal variable

    1. **Add to the internal variable count:** Decide in what cases the variable will be needed, 1D calculations, 2D calculations, when there is a gamma law gas or a tabulated equation of state, adiabatic or non-adiabatic etc. Then once decided it can be added to the total number of internal variables Grid::nNumIntVars by increasing the value by one in the function modelRead in the section below the comment "set number of internal variables ..." under the appropriate if block. If the specific if block for the situation you need isn't there, you can create your own, and add it there.
    2. **Create a new variable ID:** In the grid.h file under the Grid class are variable ID's. These ID's simply indicate the location of the variable in the array. One must add a new ID for the new variable as an integer. The value of the ID is set in the function modelRead in the same section as the number of internal variables. The value used should be the last integer after the last pre-existing variable ID. This should also be Grid::nNumVars + Grid::nNumIntVars -1. The ID should also be initialized to -1, so that the code knows when it isn't being used. This is done in the grid class constructor, Grid::Grid. Simply add a line in the constructor setting your new ID = -1.
    3. **Set variable infos:** Decide what the dimensions of the new variable will be. It can be cell centered or interface centered. It can also be only 1D, 2D, or 3D. Of course it will be only 1D if the entire calculation is 1D, or 2D if the calculation is 2D, but if the calculation is 3D it could also only be 2D, or 1D, and if 2D it could be only 1D. Also decide if the variable will change with time, dependent variables are only initialized and not updated during the calculations. This information is given to SPHERLS in the setInternalVarInf function in the physEquations.cpp file. The variable that is set is Grid::nVariables. It is a 2D array, the first index corresponds to the particular variable in question, the ID you made in the previous step can be used as the first index of this array. The second index refers to one of the three directions (0-2) or the time dimension (3). If the variable is centered in

the grid in direction 0 (r-direction) then this array element should have a value of 0. If the variable is interface centered in the grid in direction 0, then this array element should have a value of 1. If it isn't defined in direction 0 (for example the theta independent variable isn't defined in the 0 direction) then it should be -1. This is the same for the other 2 directions. The last element (3) should be either 0 not updated every time step, or 1 if updated every timestep. There are various sections here which allows one to set variable information based on which conditions are the variable is defined in. Put these variable infos into the most general case in which the variable is defined. At the end of this function variables are automatically adjusted depending on what the number of dimensions the model uses, so this does not need to be considered unless the variable is not used at all for a specific case of dimensions. For example a variable which is defined at cell center for all three cases for the number of dimensions (1D, 2D, 3D) will be automatically adjusted to be not defined in the 3rd direction when only doing 2D calculations, and similarly for 1D only defined in 1st direction and not defined in the 2nd or 3rd directions.

4. **Add functions:** Finally to do anything usefull with your new internal variable functions must be added to initialize the values of the variables, and to update them with time if needed. Initialization functions are called within the initInternalVars function in the physEquations.cpp file. The details of these functions will depend on what the individual variables are intended for. Functions to be called every timestep must be called from the main program loop in the file main.cpp in the appropriate order.

- How to add a new external variable
- How to add a new physics functions
- Function naming conventions
- Grid variables
- indecies and their ranges
- SPHERLS debugging tips

4.1 The Equations

I will want to give a detailed description of the equations used (probably copied from my notes wiki) so that the reader can easily see a 1-1 correspondence between the equation and the terms in SPHERLS.

4.2 Message Passing

Explain message passing in SPHERLS