# GETTING STARTED

## 1 Introduction
1.1 License and terms of use ........................................... 4

## 2 Getting started
2.1 Pre-built binaries .................................................. 5
2.2 Obtaining BOUT++ .................................................. 6
2.3 Installing dependencies .......................................... 6
2.4 Configuring BOUT++ ............................................... 8
2.5 CMake ............................................................... 9
2.6 Natural Language Support ....................................... 11
2.7 Configuring analysis routines .................................. 11
2.8 Compiling BOUT++ ................................................ 13
2.9 Running the test suite ........................................... 13
2.10 Installing BOUT++ (experimental) .............................. 14

## 3 Advanced installation options
3.1 Optimisation and run-time checking ............................ 17
3.2 Manually set compilation flags ................................. 17
3.3 Machine-specific installation .................................. 18
3.4 File formats ....................................................... 22
3.5 OpenMP ........................................................... 23
3.6 SUNDIALS .......................................................... 24
3.7 PETSc ............................................................... 25
3.8 LAPACK ............................................................ 26
3.9 MUMPS .............................................................. 27
3.10 MPI compilers .................................................... 27
3.11 Installing FFTW from source ................................ 28
3.12 Compiling and running under AIX ............................ 28
3.13 Issues ............................................................... 31

## 4 Running BOUT++
4.1 Quick start .......................................................... 33
4.2 Natural language support ....................................... 35
4.3 When things go wrong .......................................... 36
4.4 Startup output ..................................................... 36
4.5 Per-timestep output .............................................. 40
4.6 Restarting runs ..................................................... 40
4.7 Stopping simulations ............................................ 41
4.8 Manipulating restart files ..................................... 42
11.4 From ELITE and GATO files .......................................................... 109
11.5 Generating equilibria ..................................................................... 109
11.6 Zoidberg grid generator ................................................................. 111

12 Post-processing ................................................................. 121
12.1 Python routines ........................................................................... 121
12.2 Python analysis routines .............................................................. 124
12.3 Reading BOUT++ output into IDL ................................................ 125
12.4 Summary of IDL file routines ...................................................... 126
12.5 IDL analysis routines .................................................................. 127
12.6 Matlab routines ........................................................................... 128
12.7 Mathematica routines .................................................................. 128
12.8 Octave routines ........................................................................... 129

13 Reproducibility and provenance tracking ........................................... 131

14 The python boutcore module ........................................................... 133
14.1 Installing ...................................................................................... 133
14.2 Purpose ....................................................................................... 134
14.3 State .......................................................................................... 134
14.4 Functions ................................................................................... 134
14.5 Examples ................................................................................... 134
14.6 Functions - undocumented ......................................................... 136
14.7 Functions - special and inherited ................................................ 136

15 Time integration ................................................................. 137
15.1 Options ....................................................................................... 137
15.2 CVODE ..................................................................................... 138
15.3 IMEX-BDF2 ............................................................................... 139
15.4 Split-RK ..................................................................................... 140
15.5 Backward Euler - SNES ............................................................. 141
15.6 ODE integration .......................................................................... 141
15.7 Preconditioning ......................................................................... 143
15.8 Jacobian function ....................................................................... 145
15.9 DAE constraint equations ........................................................... 145
15.10 IMEX-BDF2 ............................................................................ 146
15.11 Monitoring the simulation output ............................................... 146
15.12 Implementation internals ............................................................ 148

16 Parallel Transforms ................................................................. 151
16.1 Field-aligned grid ...................................................................... 152
16.2 Shifted metric ........................................................................... 152
16.3 FCI method ................................................................................ 152

17 Laplacian inversion ...................................................................... 155
17.1 Usage of the laplacian inversion .................................................. 155
17.2 Numerical implementation ......................................................... 159
17.3 Implementation internals ............................................................. 164
17.4 LaplaceXY ............................................................................... 167
17.5 LaplaceXZ ............................................................................... 168

18 Differential operators ................................................................. 173
18.1 Differencing methods .................................................................. 173
18.2 User registered methods ............................................................. 175
18.3 Mixed second-derivative operators .............................................. 176
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.4</td>
<td>Non-uniform meshes</td>
<td>177</td>
</tr>
<tr>
<td>18.5</td>
<td>General operators</td>
<td>177</td>
</tr>
<tr>
<td>18.6</td>
<td>Clebsch operators</td>
<td>178</td>
</tr>
<tr>
<td>18.7</td>
<td>The bracket operators</td>
<td>178</td>
</tr>
<tr>
<td>18.8</td>
<td>Finite volume, conservative finite difference methods</td>
<td>179</td>
</tr>
<tr>
<td>18.9</td>
<td>Derivatives of the Fourier transform</td>
<td>182</td>
</tr>
<tr>
<td>19</td>
<td>Algebraic operators</td>
<td>185</td>
</tr>
<tr>
<td>20</td>
<td>Staggered grids</td>
<td>187</td>
</tr>
<tr>
<td>21</td>
<td>Eigenvalue solver</td>
<td>191</td>
</tr>
<tr>
<td>21.1</td>
<td>Configuring with SLEPc</td>
<td>191</td>
</tr>
<tr>
<td>21.2</td>
<td>SLEPc options</td>
<td>191</td>
</tr>
<tr>
<td>21.3</td>
<td>Examples</td>
<td>191</td>
</tr>
<tr>
<td>22</td>
<td>Nonlocal heat flux models</td>
<td>193</td>
</tr>
<tr>
<td>22.1</td>
<td>Spitzer-Harm heat flux</td>
<td>193</td>
</tr>
<tr>
<td>22.2</td>
<td>SNB model</td>
<td>193</td>
</tr>
<tr>
<td>23</td>
<td>Invertable operators</td>
<td>199</td>
</tr>
<tr>
<td>24</td>
<td>PETSc solvers</td>
<td>203</td>
</tr>
<tr>
<td>25</td>
<td>Field-aligned coordinates</td>
<td>205</td>
</tr>
<tr>
<td>25.1</td>
<td>Introduction</td>
<td>205</td>
</tr>
<tr>
<td>25.2</td>
<td>Orthogonal toroidal coordinates</td>
<td>205</td>
</tr>
<tr>
<td>25.3</td>
<td>Field-aligned coordinates</td>
<td>206</td>
</tr>
<tr>
<td>25.4</td>
<td>Shifted radial derivatives</td>
<td>216</td>
</tr>
<tr>
<td>25.5</td>
<td>Useful identities</td>
<td>218</td>
</tr>
<tr>
<td>.1</td>
<td>Differential geometry</td>
<td>219</td>
</tr>
<tr>
<td>.2</td>
<td>Derivation of operators in the BOUT++ Clebsch system</td>
<td>220</td>
</tr>
<tr>
<td>.3</td>
<td>Divergence of ExB velocity</td>
<td>225</td>
</tr>
<tr>
<td>A</td>
<td>BOUT++ preconditioning</td>
<td>227</td>
</tr>
<tr>
<td>A.1</td>
<td>Introduction</td>
<td>227</td>
</tr>
<tr>
<td>A.2</td>
<td>Physics problems</td>
<td>228</td>
</tr>
<tr>
<td>A.3</td>
<td>Jacobian-vector multiply</td>
<td>230</td>
</tr>
<tr>
<td>A.4</td>
<td>Preconditioner-vector multiply</td>
<td>230</td>
</tr>
<tr>
<td>A.5</td>
<td>Stencils</td>
<td>231</td>
</tr>
<tr>
<td>A.6</td>
<td>Jacobian calculation</td>
<td>231</td>
</tr>
<tr>
<td>B</td>
<td>Geometry and Differential Operator</td>
<td>233</td>
</tr>
<tr>
<td>B.1</td>
<td>Geometry</td>
<td>233</td>
</tr>
<tr>
<td>B.2</td>
<td>Geometry and Differential Operators</td>
<td>233</td>
</tr>
<tr>
<td>C</td>
<td>Contributing to BOUT++</td>
<td>239</td>
</tr>
<tr>
<td>C.1</td>
<td>House rules</td>
<td>239</td>
</tr>
<tr>
<td>C.2</td>
<td>Development workflow using Git</td>
<td>240</td>
</tr>
<tr>
<td>C.3</td>
<td>Coding Style</td>
<td>241</td>
</tr>
<tr>
<td>D</td>
<td>Git crash course</td>
<td>245</td>
</tr>
<tr>
<td>D.1</td>
<td>Accessing github from behind a firewall</td>
<td>246</td>
</tr>
<tr>
<td>D.2</td>
<td>Creating a private repository</td>
<td>246</td>
</tr>
</tbody>
</table>
The documentation is divided into the following sections:

- User documentation
  - Getting started
  - BOUT++ models
  - Model inputs
  - Model outputs
  - BOUT++ interfaces

- Developer Documentation
BOUT++ is a C++ framework for writing plasma fluid simulations with an arbitrary number of equations in 3D curvilinear coordinates. More specifically, it is a multiblock structured finite difference (/volume) code in curvilinear coordinates, with some features to support unusual coordinate systems used in fusion plasma physics. It has been developed from the original BOUndary Turbulence 3D 2-fluid edge simulation code written by X.Xu and M.Umansky at LLNL.

The aim of BOUT++ is to automate the common tasks needed for simulation codes, and to separate the complicated (and error-prone) details such as differential geometry, parallel communication, and file input/output from the user-specified equations to be solved. Thus the equations being solved are made clear, and can be easily changed with only minimal knowledge of the inner workings of the code. As far as possible, this allows the user to concentrate on the physics, rather than worrying about the numerics. This doesn’t mean that users don’t have to think about numerical methods, and so selecting differencing schemes and boundary conditions is discussed in this manual. The generality of BOUT++ of course also comes with a limitation: although there is a large class of problems which can be tackled by this code, there are many more problems which require a more specialised solver and which BOUT++ will not be able to handle. Hopefully this manual will enable you to test whether BOUT++ is suitable for your problem as quickly and painlessly as possible.

BOUT++ treats time integration and spatial operators separately, an approach called the Method of Lines (MOL). This means that BOUT++ consists of two main parts:

1. A set of Ordinary Differential Equation (ODE) integrators, including implicit, explicit and IMEX schemes, such as Runge-Kutta and the CVODE solver from SUNDIALS. These don’t “know” anything about the equations being solved, only requiring the time derivative of the system state. For example they make no distinction between the different evolving fields, or the number of dimensions in the simulation. This kind of problem-specific information can be used to improve efficiency, and is usually supplied in the form of user-supplied preconditioners. See section Options for more details.

2. A set of operators and data types for calculating time derivatives, given the system state. These calculate things like algebraic operations (+,-,*,/ etc), spatial derivatives, and some integral operators.

Each of these two parts treats the other as a black box (mostly), and they communicate by exchanging arrays of data: The ODE integrator finds the system state at a given time and passes it to the problem-dependent code, which uses a combination of operators to calculate the time derivative. This time derivative is passed back to the ODE integrator, which updates the state and the cycle continues. This scheme has some advantages in terms of flexibility: Each part of the code doesn’t depend on the details of the other, so can be changed without requiring modifications to the other. Unfortunately for many problems the details can make a big difference, so ways to provide problem-specific information to time integrators, such as preconditioners, are also provided.

Though designed to simulate tokamak edge plasmas, the methods used are very general and almost any metric tensor can be specified, allowing the code to be used to perform simulations in (for example) slab, sheared slab, and cylindrical coordinates. The restrictions on the simulation domain are that the equilibrium must be axisymmetric (in the z coordinate), and that the parallelisation is done in the x and y (parallel to B) directions.

After describing how to install BOUT++ (section Getting started), run the test suite (section Running the test suite) and a few examples (section Running BOUT++, more detail in section More examples), increasingly sophisticated ways to
modify the problem being solved are introduced. The simplest way to modify a simulation case is by altering the input options, described in section BOUT++ options. Checking that the options are doing what you think they should be by looking at the output logs is described in section Running BOUT++, and an overview of the IDL analysis routines for data post-processing and visualisation is given in section Post-processing. Generating new grid files, particularly for tokamak equilibria, is described in section Generating input grids.

Up to this point, little programming experience has been assumed, but performing more drastic alterations to the physics model requires modifying C++ code. Section BOUT++ physics models describes how to write a new physics model specifying the equations to be solved, using ideal MHD as an example. The remaining sections describe in more detail aspects of using BOUT++: section Differential operators describes the differential operators and methods available; section Staggered grids covers the experimental staggered grid system.

Various sources of documentation are:

- This manual
- Most directories in the BOUT++ distribution contain a README file. This should describe briefly what the contents of the directory are and how to use them.
- Most of the code contains Doxygen comment tags (which are slowly getting better). Running doxygen on these files should therefore generate an HTML reference. This is probably going to be the most up-to-date documentation.

1.1 License and terms of use

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see <https://www.gnu.org/licenses/>.

A copy of the LGPL license is in COPYING.LESSER. Since this is based on (and refers to) the GPL, this is included in COPYING.

BOUT++ is free software, but since it is a scientific code we also ask that you show professional courtesy when using this code:

1. Since you are benefiting from work on BOUT++, we ask that you submit any improvements you make to the code to us by emailing Ben Dudson at bd512@york.ac.uk

2. If you use BOUT++ results in a paper or professional publication, we ask that you send your results to one of the BOUT++ authors first so that we can check them. It is understood that in most cases if one or more of the BOUT++ team are involved in preparing results then they should appear as co-authors.

3. Publications or figures made with the BOUT++ code should acknowledge the BOUT++ code by citing B.Dudson et. al. Comp.Phys.Comm 2009 and/or other BOUT++ papers. See the file CITATION for details.
CHAPTER TWO

GETTING STARTED

This section goes through the process of getting, installing, and starting to run BOUT++.

The quickest way to get started is to use a pre-built binary. These take care of all dependencies, configuration and compilation. See section Docker image.

The remainder of this section will go through the following steps to manually install BOUT++. Only the basic functionality needed to use BOUT++ is described here; the next section (Advanced installation options) goes through more advanced options, configurations for particular machines, and how to fix some common problems.

1. Obtaining a copy of BOUT++
2. Installing dependencies
3. Configuring BOUT++
4. Configuring BOUT++ analysis codes
   1. Python
   2. IDL
5. Compiling BOUT++
6. Running the test suite
7. Installing BOUT++ (experimental)

Note: In this manual commands to run in a BASH shell will begin with ‘$’, and commands specific to CSH with a ‘%’.

2.1 Pre-built binaries

2.1.1 Docker image

Docker is a widely used container system, which packages together the operating system environment, libraries and other dependencies into an image. This image can be downloaded and run reproducibly on a wide range of hosts, including Windows, Linux and OS X. Here is the starting page for instructions on installing Docker.

The BOUT++ docker images are hosted on dockerhub for some releases and snapshots. Check the list of BOUT-next tags if you want a recent version of BOUT++ “next” (development) branch. First download the image:

$ sudo docker pull boutproject/boutproject/bout-next:9f4c663-petsc

then run:

$ sudo docker run --rm -it boutproject/bout-next:9f4c663-petsc
This should give a terminal in a “boutuser” home directory, in which there is “BOUT-next”, containing BOUT++ configured and compiled with NetCDF, HDF5, SUNDIALS, PETSc and SLEPc. Python 3 is also installed, with ipython, NumPy, Scipy and Matplotlib libraries. To plot to screen an X11 display is needed. Alternatively a shared directory can be created to pass files between the docker image and host. The following commands both enable X11 and create a shared directory:

```
$ mkdir shared
$ sudo docker run --rm -it
   -e DISPLAY -v $HOME/.Xauthority:/home/boutuser/.Xauthority --net=host \
   -v $PWD/shared:/home/boutuser/bout-img-shared \
   boutproject/bout-next:9f4c663-petsc
```

This should enable plotting from python, and files in the docker image put in “/home/boutuser/bout-img-shared” should be visible on the host in the “shared” directory.

If this is successful, then you can skip to section *Running BOUT++.*

### 2.2 Obtaining BOUT++

BOUT++ is hosted publicly on github at [https://github.com/boutproject/BOUT-dev](https://github.com/boutproject/BOUT-dev). You can the latest stable version from [https://github.com/boutproject/BOUT-dev/releases](https://github.com/boutproject/BOUT-dev/releases). If you want to develop BOUT++, you should use git to clone the repository. To obtain a copy of the latest version, run:

```
$ git clone git://github.com/boutproject/BOUT-dev.git
```

which will create a directory `BOUT-dev` containing the code. To get the latest changes later, go into the `BOUT-dev` directory and run:

```
$ git pull
```

Development is done on the “next” branch, which you can checkout with:

```
$ git checkout next
```

### 2.3 Installing dependencies

The bare-minimum requirements for compiling and running BOUT++ are:

1. A C++ compiler that supports C++11

The FFTW-3 library ([http://www.fftw.org/](http://www.fftw.org/)) is also strongly recommended. Fourier transforms are used for some derivative methods, as well as the *ShiftedMetric* parallel transform which is used in the majority of BOUT++ tokamak simulations. Without FFTW-3, these options will not be available.

*Note:* Only GCC versions >= 4.9 are supported. This is due to a bug in previous versions.
Note: If you use an Intel compiler, you must also make sure that you have a version of GCC that supports C++11 (GCC 4.8+).

On supercomputers, or in other environments that use a module system, you may need to load modules for both Intel and GCC.

### 2.3.1 On a cluster or supercomputer

If you are installing on a cluster or supercomputer then the MPI C++ compilers will already be installed, and on Cray or IBM machines will probably be called CC and x1C respectively.

On large facilities (e.g NERSC or Archer), the compilers and libraries needed should already be installed, but you may need to load them to use them. It is common to organise libraries using the modules system, so try typing:

```
modules avail
```

to get a list of available modules. Some instructions for specific machines can be found in Machine-specific installation. See your system’s documentation on modules and which ones to load. If you don’t know, or modules don’t work, you can still install libraries in your home directory by following the instructions below for FFTW and NetCDF.

### 2.3.2 Ubuntu / Debian

On Ubuntu or Debian distributions if you have administrator rights then you can install MPICH2 and the needed libraries by running:

```bash
$ sudo apt-get install mpich2 libmpich2-dev
$ sudo apt-get install libfftw3-dev libnetcdf-dev libnetcdf-cxx-legacy-dev
```

On Ubuntu 16.04:

```bash
$ sudo apt-get libmpich-dev libfftw3-dev libnetcdf-dev libnetcdf-cxx-legacy-dev
```

On Ubuntu 18.04:

```bash
$ sudo apt-get install mpich libmpich-dev libfftw3-dev libnetcdf-dev libnetcdf-cxx-legacy-dev
$ sudo apt-get install python3 python3-distutils python3-pip python3-numpy python3-netcdf4 python3-scipy
$ pip3 install --user Cython
```

The first line should be sufficient to install BOUT++, while the 2nd and 3rd line make sure that the tests work, and that the python interface can be build. Further, the encoding for python needs to be utf8 - it may be required to set `export LC_CTYPE=C.utf8`.

If you do not have administrator rights, so can’t install packages, then you need to install these libraries from source into your home directory. See sections on installing MPI, installing FFTW and installing NetCDF.

2.3. Installing dependencies
2.3.3 Arch Linux

$ pacman -S openmpi fftw netcdf-cxx make gcc

2.3.4 Fedora

On Fedora the required libraries can be installed by running:

$ sudo dnf build-dep bout++

This will install all the dependencies that are used to install BOUT++ for fedora. Feel free to install only a subset of the suggested packages. For example, only mpich or openmpi is required. To load an mpi implementation type:

$ module load mpi

After that the mpi library is loaded. Precompiled binaries are available for fedora as well. To get precompiled BOUT++ run:

$ # install the mpich version - openmpi is available as well
$ sudo dnf install bout++-mpich-devel
$ # get the python3 modules - python2 is available as well
$ sudo dnf install python3-bout++

2.4 Configuring BOUT++

To compile BOUT++, you first need to configure it. Go into the BOUT-dev directory and run:

$ ./configure

If this finishes by printing a summary, and paths for IDL, Python, and Octave, then the libraries are set up and you can skip to the next section. If you see a message "ERROR: FFTW not found. Required by BOUT++" then make sure FFTW-3 is installed (See the previous section on installing dependencies).

If FFTW-3 is installed in a non-standard location, you can specify the directory with the --with-fftw= option e.g:

$ ./configure --with-fftw=$HOME/local

Configure should now find FFTW, and search for the NetCDF library. If configure finishes successfully, then skip to the next section, but if you see a message NetCDF support disabled then configure couldn't find the NetCDF library. Unless you have another file format (like HDF5) installed, this will be followed by a message ERROR: At least one file format must be supported. Check that you have NetCDF installed (See the previous section on installing dependencies).

Like the FFTW-3 library, if NetCDF is installed in a non-standard location then you can specify the directory with the --with-netcdf= option e.g:

$ ./configure --with-fftw=$HOME/local --with-netcdf=$HOME/local

which should now finish successfully, printing a summary of the configuration:
Configuration summary

PETSc support: no
SLEPc support: no
IDA support: yes
CVODE support: yes
ARKODE support: yes
NetCDF support: yes (parallel: no)
HDF5 support: yes (parallel: no)
MUMPS support: no

If not, see Advanced installation options for some things you can try to resolve common problems.

2.5 CMake

There is now (experimental) support for CMake. You will need CMake > 3.9. CMake supports out-of-source builds by default, which are A Good Idea. Basic configuration with CMake looks like:

```
$ cmake . -B build
```

which creates a new directory `build`, which you can then compile with:

```
$ cmake --build build
```

You can see what build options are available with:

```
$ cmake . -B build -LH
...
// Enable backtrace
BOUT_ENABLE_BACKTRACE:BOOL=ON

// Output coloring
BOUT_ENABLE_COLOR:BOOL=ON

// Enable OpenMP support
BOUT_ENABLE_OPENMP:BOOL=OFF

// Enable support for PETSc time solvers and inversions
BOUT_USE_PETSC:BOOL=OFF
...
```

CMake uses the `-D<variable>=<choice>` syntax to control these variables. You can set `<package>_ROOT` to guide CMake in finding the various optional third-party packages (except for PETSc/SLEPc, which use `_DIR`). Note that some packages have funny capitalisation, for example `NetCDF_ROOT`! Use `-LH` to see the form that each package expects.

CMake understands the usual environment variables for setting the compiler, compiler/linking flags, as well as having built-in options to control them and things like static vs shared libraries, etc. See the CMake documentation for more information.

A more complicated CMake configuration command might look like:

```
$ CC=mpicc CXX=mpic++ cmake . -B build \
   -DBOUT_USE_PETSC=ON -DPETSC_DIR=/path/to/petsc/ \
```

(continues on next page)
If you wish to change the configuration after having built BOUT++, it's wise to delete the CMakeCache.txt file in the build directory. The equivalent of make distclean with CMake is to just delete the entire build directory and reconfigure.

### 2.5.1 Downloading Dependencies

If you don't have some dependencies installed, CMake can be used to download, configure and compile them alongside BOUT++.

For NetCDF, use `-DBOUT_DOWNLOAD_NETCDF_CXX4=ON`

For SUNDIALS, use `-DBOUT_DOWNLOAD_SUNDIALS=ON`. If using ccmake this option may not appear initially. This automatically sets BOUT_USE_SUNDIALS=ON, and configures SUNDIALS to use MPI.

### Bundled Dependencies

BOUT++ bundles some dependencies, currently mpark.variant, fmt and googletest. If you wish to use an existing installation of mpark.variant, you can set `-DBOUT_USE_SYSTEM_MPARK_VARIANT=ON`, and supply the installation path using mpark_variant_ROOT via the command line or environment variable if it is installed in a non standard location. You can also set `-DBOUT_USE_GIT_SUBMODULE=OFF`.

The recommended way to use googletest is to compile it at the same time as your project, therefore there is no option to use an external installation for that.

### Using CMake with your physics model

You can write a CMake configuration file (CMakeLists.txt) for your physics model in only four lines:

```cmake
project(blob2d LANGUAGES CXX)
find_package(bout++ REQUIRED)
add_executable(blob2d blob2d.cxx)
target_link_libraries(blob2d PRIVATE bout++::bout++)
```

You just need to give CMake the location where you built or installed BOUT++ via the CMAKE_PREFIX_PATH variable:

```
$ cmake . -B build -DCMAKE_PREFIX_PATH=/path/to/built/BOUT++
```

If you want to modify BOUT++ along with developing your model, you may instead wish to place the BOUT++ as a subdirectory of your model and use add_subdirectory instead of find_package above:

```cmake
project(blob2d LANGUAGES CXX)
add_subdirectory(BOUT++/source)
```

(continues on next page)
add_executable(blob2d blob2d.cxx)
target_link_libraries(blob2d PRIVATE bout++:bout++)

where BOUT++/source is the subdirectory containing the BOUT++ source. Doing this has the advantage that any changes you make to BOUT++ source files will trigger a rebuild of both the BOUT++ library and your model when you next build your code.

2.6 Natural Language Support

BOUT++ has support for languages other than English, using GNU gettext. If you are planning on installing BOUT++ (see sec-install-bout) then this should work automatically, but if you will be running BOUT++ from the directory you downloaded it into, then configure with the option:

```
./configure --localedir=$PWD/locale
```

This will enable BOUT++ to find the translations. When configure finishes, the configuration summary should contain a line like:

```
configure: Natural language support: yes (path: /home/user/BOUT-dev/locale)
```

where the path is the directory containing the translations.

See Natural language support for details of how to switch language when running BOUT++ simulations.

2.7 Configuring analysis routines

The BOUT++ installation comes with a set of useful routines which can be used to prepare inputs and analyse outputs. Most of this code is now in Python, though IDL was used for many years. Python is useful in particular because the test suite scripts and examples use Python, so to run these you’ll need python configured.

When the configure script finishes, it prints out the paths you need to get IDL, Python, and Octave analysis routines working. If you just want to compile BOUT++ then you can skip to the next section, but make a note of what configure printed out.

2.7.1 Python configuration

To use Python, you will need the dependencies of the boututils and boutdata libraries. The simplest way to get these is to install the packages with pip:

```
$ pip install --user boutdata
```

or conda:

```
$ conda install boutdata
```

You can also install all the packages directly (see the documentation in the boututils and boutdata repos for the most up to date list) using pip:

```
$ pip install --user numpy scipy matplotlib sympy netCDF4 h5py future importlib-metadata
```

or conda:
$ conda install numpy scipy matplotlib sympy netcdf4 h5py future importlib-metadata

They may also be available from your Linux system’s package manager.

To use the versions of boututils and boutdata provided by BOUT++, the path to tools/pylib should be added to the PYTHONPATH environment variable. This is not necessary if you have installed the boututils and boutdata packages. Instructions for doing this are printed at the end of the configure script, for example:

Make sure that the tools/pylib directory is in your PYTHONPATH e.g. by adding to your ~/.bashrc file

    export PYTHONPATH=/home/ben/BOUT/tools/pylib/:$PYTHONPATH

To test if this command has worked, try running:

$ python -c "import boutdata"

If this doesn’t produce any error messages then Python is configured correctly.

Note that boututils and boutdata are provided by BOUT++ as submodules, so versions compatible with the checked out version of BOUT++ are downloaded into the externalpackages directory. These are the versions used by the tests run by make check even if you have installed boututils and boutdata on your system.

### 2.7.2 IDL configuration

If you want to use IDL to analyse BOUT++ outputs, then the IDL_PATH environment variable should include the tools/idllib/ subdirectory included with BOUT++. The required command (for Bash) is printed at the end of the BOUT++ configuration:

$ export IDL_PATH=...

After running that command, check that idl can find the analysis routines by running:

$ idl
IDL> .r collect
IDL> help, /source

You should see the function COLLECT in the BOUT/tools/idllib directory. If not, something is wrong with your IDL_PATH variable. On some machines, modifying IDL_PATH causes problems, in which case you can try modifying the path inside IDL by running:

IDL> !path = !path + ":/path/to/BOUT-dev/tools/idllib"

where you should use the full path. You can get this by going to the tools/idllib directory and typing pwd. Once this is done you should be able to use collect and other routines.
## 2.8 Compiling BOUT++

Once BOUT++ has been configured, you can compile the bulk of the code by going to the BOUT-dev directory (same as configure) and running:

```
$ make
```

(on OS-X, FreeBSD, and AIX this should be `gmake`). This should print something like:

```
----- Compiling BOUT++ -----
CXX    = mpicxx
CFLAGS = -O -DHECK=2 -DSIGHANDLE \
-DREVISION=13571f760cec446d907e1beb1d7a3b1c6e0212a \
-DNCF -DBOUT_HAS_PVODE
CHECKSUM = ff3fb702b13acc092613cfe3869b875
INCLUDE = -I../include
Compiling field.cxx
Compiling field2d.cxx
```

At the end of this, you should see a file `libbout++.a` in the `lib/` subdirectory of the BOUT++ distribution. If you get an error, please create an issue on Github including:

- Which machine you're compiling on
- The output from make, including full error message
- The `make.config` file in the BOUT++ root directory

## 2.9 Running the test suite

BOUT++ comes with three sets of test suites: unit tests, integrated tests and method of manufactured solutions (MMS) tests. The easiest way to run all of them is to simply do:

```
$ make check
```

from the top-level directory. Alternatively, if you just want to run one them individually, you can do:

```
$ make check-unit-tests
$ make check-integrated-tests
$ make check-mms-tests
```

**Note:** The integrated test suite currently uses the `mpirun` command to launch the runs, so won't work on machines which use a job submission system like PBS or SGE.

These tests should all pass, but if not please create an issue on Github containing:

- Which machine you’re running on
- The `make.config` file in the BOUT++ root directory
- The `run.log.*` files in the directory of the test which failed

If the tests pass, congratulations! You have now got a working installation of BOUT++. Unless you want to use some experimental features of BOUT++, skip to section [sec-running] to start running the code.
2.10 Installing BOUT++ (experimental)

Most BOUT++ users install and develop their own copies in their home directory, so do not need to install BOUT++ to a system directory. As of version 4.1 (August 2017), it is possible to install BOUT++ but this is not widely used and so should be considered experimental.

After configuring and compiling BOUT++ as above, BOUT++ can be installed to system directories by running as superuser or sudo:

```
$ sudo make install
```

**Danger:** Do not do this unless you know what you’re doing!

This will install the following files under `/usr/local/`:

- `/usr/local/bin/bout-config` A script which can be used to query BOUT++ configuration and compile codes with BOUT++.
- `/usr/local/include/bout++/*` header files for BOUT++
- `/usr/local/lib/libbout++.a` The main BOUT++ library
- `/usr/local/lib/libpvode.a` and `/usr/local/lib/libpvpre.a`, the PVODE library
- `/usr/local/share/bout++/*` Python analysis routines
- `/usr/local/share/bout++/*` IDL analysis routines
- `/usr/local/share/bout++/make.config` A makefile configuration, used to compile many BOUT++ examples

To install BOUT++ under a different directory, use the `--prefix=` flag e.g. to install in your home directory:

```
$ make install prefix=$HOME/local/
```

You can also specify this prefix when configuring, in the usual way (see Configuring BOUT++):

```
$ ./configure --prefix=$HOME/local/
$ make
$ make install
```

More control over where files are installed is possible by passing options to `configure`, following the GNU conventions:

- `--bindir=` sets where `bout-config` will be installed (default `/usr/local/bin`)
- `--includedir=` sets where the `bout++/*.hxx` header files will be installed (default `/usr/local/include`)
- `--libdir=` sets where the `libbout++.a`, `libpvode.a` and `libpvpre.a` libraries are installed (default `/usr/local/lib`)
- `--datadir=` sets where `idllib`, `pylib` and `make.config` are installed (default `/usr/local/share/`)

After installing, that you can run `bout-config` e.g:

```
$ bout-config --all
```

which should print out the list of configuration settings which `bout-config` can provide. If this doesn’t work, check that the directory containing `bout-config` is in your PATH.
The python and IDL analysis scripts can be configured using `bout-config` rather than manually setting paths as in *Configuring analysis routines*. Add this line to your startup file (e.g. `$HOME/.bashrc`):

```
export PYTHONPATH=`bout-config --python`:$PYTHONPATH
```

note the back ticks around `bout-config --python` not quotes. Similarly for IDL:

```
export IDL_PATH=`bout-config --idl`:`<IDL_DEFAULT>`:$IDL_PATH
```

More details on using `bout-config` are in the section on makefiles.
This section describes some common issues encountered when configuring and compiling BOUT++, how to manually install dependencies if they are not available, and how to configure optional libraries like SUNDIALS and PETSc.

### 3.1 Optimisation and run-time checking

Configure with `--enable-checks=3` enables a lot of checks of operations performed by the field objects. This is very useful for debugging a code, and can be omitted once bugs have been removed. `--enable-checks=2` enables less checking, especially the computationally rather expensive ones, while `--enable-checks=0` disables most checks.

To get most checking, both from BOUT++ and from the compiler `--enable-debug` can be used. That enables checks of level 3, as well as debug flags, e.g. `-g` for gcc.

For (sometimes) more useful error messages, there is the `--enable-track` option. This keeps track of the names of variables and includes these in error messages.

To enable optimization, configure with `--enable-optimize=3`. This will try to set appropriate flags, but may not set the best ones. This should work well for gcc. Similar to checks, different levels can be specified, where 3 is high, and 0 means disabling all optimization. `--enable-optimize=fast` will set the `-Ofast` flag for gcc which enables optimizations that are not standard conforming, so proceed at own risk.

### 3.2 Manually set compilation flags

You can set the following environment variables if you need more control over how BOUT++ is built:

- **LDFLAGS**: extra flags for linking, e.g. `-L<library dir>`
- **LIBS**: extra libraries for linking, e.g. `-l<library>`
- **CPPFLAGS**: preprocessor flags, e.g. `-I<include dir>`
- **CXXFLAGS**: compiler flags, e.g. `-Wall`
- **SUNDIALS_EXTRA_LIBS** specifies additional libraries for linking to SUNDIALS, which are put at the end of the link command.

It is possible to change flags for BOUT++ after running configure, by editing the `make.config` file. Note that this is not recommended, as e.g. PVODE will not be built with these flags.
3.3 Machine-specific installation

These are some configurations which have been found to work on particular machines.

3.3.1 Archer

As of 20th April 2018, the following configuration should work

```bash
$ module swap PrgEnv-cray PrgEnv-gnu/5.1.29
$ module load fftw
$ module load archer-netcdf/4.1.3
```

When using CMake on Cray systems like Archer, you need to pass `-DCMAKE_SYSTEM_NAME=CrayLinuxEnvironment` so that the Cray compiler wrappers are detected properly.

3.3.2 KNL @ Archer

To use the KNL system, configure BOUT++ as follows:

```bash
./configure MPICXX=CC --host=knl --with-netcdf --with-pnetcdf=no --with-hypre=no
   -CXXFLAGS="-xMIC-AVX512 -D_GLIBCXX_USE_CXX11_ABI=0"
```

3.3.3 Atlas

```bash
./configure --with-netcdf=/usr/local/tools/hdf5-gnu-serial-1.8.1/lib --with-fftw=/usr/local --with-pdb=/usr/gapps/pact/new/lnx-2.5-ib/gnu
```

3.3.4 Cab

```bash
./configure --with-netcdf=/usr/local/tools/hdf5-gnu-serial-1.8.1/lib --with-fftw=/usr/local/tools/fftw3-3.2 --with-pdb=/usr/gapps/pact/new/lnx-2.5-ib/gnu
```

3.3.5 Edison

```bash
module swap PrgEnv-intel PrgEnv-gnu
module load fftw
./configure MPICC=cc MPICXX=CC --with-netcdf=/global/u2/c/chma/PUBLIC/netcdf_edison/netcdf --with-fftw=/opt/fftw/3.3.0.1/x86_64
```
3.3.6 Hoffman2

```
˓ current --with-cvode=/u/local/apps/sundials/2.4.0 --with-lapack=/u/local/apps/lapack/
˓ current
```

3.3.7 Hopper

```
module swap PrgEnv-pgi PrgEnv-gnu
module load netcdf
module swap netcdf netcdf/4.1.3
module swap gcc gcc/4.6.3
./configure MPICC=cc MPICXX=CC --with-fftw=/opt/fftw/3.2.2.1 --with-pdb=/global/homes/u/
˓ umansky/PUBLIC/PACT_HOPP2/pact
```

3.3.8 Hyperion

With the bash shell use

```
export PETSC_DIR=~farley9/projects/petsc/petsc-3.2-p1
export PETSC_ARCH=arch-c
./configure --with-netcdf=/usr/local/tools/netcdf-gnu-4.1 --with-fftw=/usr/local_
˓ MPICXX=mpicc EXTRA_LIBS=-lcurl --with-petsc --with-cvode=~farley9/local --with-ida=~
˓ farley9/local
```

With the tcsh shell use

```
setenv PETSC_DIR ~farley9/projects/petsc/petsc-3.2-p1
setenv PETSC_ARCH arch-c
./configure --with-netcdf=/usr/local/tools/netcdf-gnu-4.1 --with-fftw=/usr/local_
˓ MPICXX=mpicc EXTRA_LIBS=-lcurl --with-petsc --with-cvode=~farley9/local --with-ida=~
˓ farley9/local
```

3.3.9 Marconi

```
module load intel intelmpi fftw lapack
module load zlip zlib/1.2.8--gnu--6.1.0
module load hdf5/1.8.17--intel--pe-xe-2017--binary
module load netcdf-cxx4
module load python
```

To compile for the SKL partition, configure with

```
./configure --enable-checks=0 CPPFLAGS="-Ofast -funroll-loops -xCORE-AVX512 -
˓ -mtune=skylake" --host skl
```

to enable AVX512 vectorization.

3.3. Machine-specific installation
Note: As of 20/04/2018, an issue with the netcdf and netcdf-cxx4 modules means that you will need to remove `-lnetcdf` from EXTRA_LIBS in make.config after running `./configure` and before running `make`. `-lnetcdf` needs also to be removed from bin/bout-config to allow a successful build of the python interface. Recreation of `boutcore.pyx` needs to be manually triggered, if `boutcore.pyx` has already been created.

Marconi with gnu compilers

It is also possible to configure on Marconi using gnu compilers, which may give better performance. A set of modules which work as of 30/9/2020 is

```plaintext
module load env-skl
module load profile/advanced
module load intel/pe-xe-2018--binary  # note need to keep the 'intel' module loaded in order for shared libraries needed by numpy/scipy to be available
module load gnu/7.3.0
module load openmpi/4.0.1--gnu--7.3.0
module load mkl/2017--binary
module load python/3.6.4
module load szip/2.1--gnu--6.1.0 zlib/1.2.8--gnu--6.1.0
```

Then download source code for hdf5-1.12.0 (hdf5 is available in a module on Marconi, but has issues linking OpenMPI), netCDF-c-4.7.4, netCDF-cxx4-4.3.1, and FFTW-3.3.8. Optionally also SUNDIALS-5.1.0 or PETSc-3.13.0. Configure and compile all of the downloaded packages. Make sure to install netCDF and netCDF-cxx4 into the same directory (this is assumed by netCDF’s linking strategy, and makes netCDF configuration simpler).

The following configuration commands have been used successfully:

- hdf5-1.12.0:
  ```
  ./configure --prefix /directory/to/install/hdf5 --enable-build-mode=production
  make
  make install
  ```

- netCDF-4.7.4:
  ```
  mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/directory/to/install/netcdf -DCMAKE_BUILD_TYPE=Release ..
  make
  make install
  ```

- netCDF-cxx4-4.3.1:
  ```
  mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/directory/to/install/netcdf -DCMAKE_BUILD_TYPE=Release ..
  make
  make install
  ```

- FFTW-3.3.8:
BOUT++ Documentation, Release 4.4.0

```bash
./configure --prefix /directory/to/install/fftw --enable-shared --enable-sse2 --
  --enable-avx --enable-avx2 --enable-avx512 --enable-avx-128-fma
make
make install
```

- **SUNDIALS-5.1.0:**

  ```bash
  mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=/directory/to/install/
  -D sundials -DMPI_ENABLE=ON ..
make
make install
```

- **PETSc-3.13.0:**

  ```bash
  unset PETSC_DIR
  ./configure COPTFLAGS="-O3" FOPTFLAGS="-O3" -O3 --with-batch --known-
  --mpi-shared-libraries=1 --with-mpi-dir=$OPENMPI_HOME --download-fblaslapack --
  --known-64-bit-blas-indices=0 --download-hypre --with-debugging=0 --prefix=/
  --directory/to/install/petsc
  ```

  then follow the instructions printed by PETSc at the end of each step to make, install and check the build.

Finally example configurations for BOUT++, where you should replace <...> by appropriate directories that you used to install the libraries:

- for an optimized build (some experimentation with optimisation flags would be welcome, please share the results if you do!):

  ```bash
  ./configure --enable-optimize=3 --enable-checks=no --without-hdf5 --enable-static --
  --with-netcdf=<...> --with-sundials=<...> --with-fftw=<...> --with-petsc=<...>
  ```

- for a debugging build:

  ```bash
  ./configure --enable-debug --without-hdf5 --enable-static --with-netcdf=<...> --
  --with-sundials=<...> --with-fftw=<...> --with-petsc=<...>
  ```

3.3. Machine-specific installation

21
### 3.3.10 Ubgl

```
./configure --with-netcdf CXXFLAGS=-DMPICH_IGNORE_CXX_SEEK CFLAGS=-DMPICH_IGNORE_CXX_SEEK --with-pdb=/usr/gapps/pact/new_s/lnx-2.5-ib --with-netcdf=/usr/local/tools/netcdf/netcdf-4.1_c++
```

### 3.4 File formats

BOUT++ can currently use two different file formats: NetCDF-4, and HDF5 and experimental support for parallel flavours of both. NetCDF is a widely used format and so has many more tools for viewing and manipulating files. HDF5 is another widely used format. If you have multiple libraries installed then BOUT++ can use them simultaneously, for example reading in grid files in NetCDF format, but writing output data in HDF5 format.

BOUT++ will try to use NetCDF by default. It will look for `ncxx4-config` or `nc-config` in your `$PATH`. If it cannot find the libraries, or finds a different version than the one you want, you can point it at the correct version using:

```
./configure --with-netcdf=/path/to/ncxx4-config
```

where `/path/to/ncxx4-config` is the location of the `ncxx4-config` tool (`nc-config` will also work, but `ncxx4-config` is preferred).

To use HDF5, you will need to explicitly enable it:

```
./configure --with-hdf5
```

BOUT++ will look for `h5cc` in your `$PATH`. Similar to NetCDF, you can pass the location of the particular version you wish to use with:

```
./configure --with-hdf5=/path/to/h5cc
```

#### 3.4.1 Installing NetCDF from source

The latest versions of NetCDF have separated out the C++ API from the main C library. As a result, you will need to download and install both. Download the latest versions of the NetCDF-C and NetCDF-4 C++ libraries from [https://www.unidata.ucar.edu/downloads/netcdf](https://www.unidata.ucar.edu/downloads/netcdf). As of September 2020, these are versions 4.7.4 and 4.3.1 respectively.

Untar the file and `cd` into the resulting directory:

```
$ tar -xzvf netcdf-4.7.4.tar.gz
$ cd netcdf-4.7.4
```

Then run `configure`, `make` and `make install`:

```
$ ./configure --prefix=$HOME/local
$ make
$ make install
```

Sometimes `configure` can fail, in which case try disabling Fortran:

```
$ ./configure --prefix=$HOME/local --disable-fortran
$ make
$ make install
```

---

Chapter 3. Advanced installation options
Similarly for the C++ API:

```
$ tar -xzvf netcdf-cxx4-4.3.1.tar.gz
$ cd netcdf-cxx4-4.3.1
$ ./configure --prefix=$HOME/local
$ make
$ make install
```

You may need to set a couple of environment variables as well:

```
$ export PATH=$HOME/local/bin:$PATH
$ export LD_LIBRARY_PATH=$HOME/local/lib:$LD_LIBRARY_PATH
```

You should check where NetCDF actually installed its libraries. On some systems this will be $HOME/local/lib, but on others it may be, e.g. $HOME/local/lib64. Check which it is, and set $LD_LIBRARY_PATH appropriately.

### 3.5 OpenMP

BOUT++ can make use of OpenMP parallelism. To enable OpenMP, use the `--enable-openmp` flag to configure:

```
./configure --enable-openmp
```

OpenMP can be used to parallelise in more directions than can be achieved with MPI alone. For example, it is currently difficult to parallelise in X using pure MPI if FCI is used, and impossible to parallelise at all in Z with pure MPI.

OpenMP is in a large number of places now, such that a decent speed-up can be achieved with OpenMP alone. Hybrid parallelisation with both MPI and OpenMP can lead to more significant speed-ups, but it sometimes requires some fine tuning of numerical parameters in order to achieve this. This greatly depends on the details not just of your system, but also your particular problem. We have tried to choose “sensible” defaults that will work well for the most common cases, but this is not always possible. You may need to perform some testing yourself to find e.g. the optimum split of OpenMP threads and MPI ranks.

One such parameter that can potentially have a significant effect (for some problem sizes on some machines) is setting the OpenMP schedule used in some of the OpenMP loops (specifically those using `BOUT_FOR`). This can be set using:

```
./configure --enable-openmp --with-openmp-schedule=<schedule>
```

with `<schedule>` being one of: `static` (the default), `dynamic`, `guided`, `auto` or `runtime`.

**Note:** If you want to use OpenMP with Clang, you will need Clang 3.7+, and either `libomp` or `libomp-pm`.

You will be able to compile BOUT++ with OpenMP with lower versions of Clang, or using the GNU OpenMP library `libgomp`, but it will only run with a single thread.

**Note:** By default PVODE is built without OpenMP support. To enable this add `--enable-pvode-openmp` to the configure command.

**Note:** OpenMP will attempt to use all available threads by default. This can cause oversubscription problems on certain systems. You can limit the number of threads OpenMP uses with the `OMP_NUM_THREADS` environment variable. See your system documentation for more details.

### 3.5. OpenMP
3.6 SUNDIALS

The BOUT++ distribution includes a 1998 version of CVODE (then called PVODE) by Scott D. Cohen and Alan C. Hindmarsh, which is the default time integration solver. Whilst no serious bugs have been found in this code (as far as the authors are aware of), several features such as user-supplied preconditioners and constraints cannot be used with this solver. Currently, BOUT++ also supports the SUNDIALS solvers CVODE, IDA and ARKODE which are available from https://computation.llnl.gov/casc/sundials/main.html.

Note: BOUT++ currently supports SUNDIALS > 2.6, up to 5.4.0 as of September 2020. It is advisable to use the highest possible version.

The full installation guide is found in the downloaded .tar.gz, but we will provide a step-by-step guide to install it and make it compatible with BOUT++ here:

```
$ tar -xzvf sundials-5.4.0.tar.gz  
$ cd sundials-5.4.0  
$ mkdir build && cd build  
$ cmake ..  
  -DCMAKE_INSTALL_PREFIX=$HOME/local  
  -DLAPACK_ENABLE=ON  
  -DOPENMP_ENABLE=ON  
  -DMPR_ENABLE=ON  
  -DCMAKE_C_COMPILER=$(which mpicc)  
  -DCMAKE_CXX_COMPILER=$(which mpicxx)  
$ make  
$ make test  
$ make install
```

The SUNDIALS IDA solver is a Differential-Algebraic Equation (DAE) solver, which evolves a system of the form \( f(u, \dot{u}, t) = 0 \). This allows algebraic constraints on variables to be specified.

Use the `--with-sundials` option to configure BOUT++ with SUNDIALS:

```
$ ./configure --with-sundials=/path/to/sundials/install
```

SUNDIALS will allow you to select at run-time which solver to use. See Options for more details on how to do this.

Notes:

- If compiling SUNDIALS, make sure that it is configured with MPI (MPI_ENABLE=ON)
- If you install SUNDIALS to a non-standard (system) directory, you will probably have to add the lib directory to the LD_LIBRARY_PATH environment variable.
3.7 PETSc

BOUT++ can use PETSc https://www.mcs.anl.gov/petsc/ for time-integration and for solving elliptic problems, such as inverting Poisson and Helmholtz equations.

Currently, BOUT++ supports PETSc versions 3.7 - 3.14. More recent versions may well work, but the PETSc API does sometimes change in backward-incompatible ways, so this is not guaranteed. To install PETSc version 3.13, use the following steps:

```
$ cd ~
$ tar -xzvf petsc-3.13.4.tar.gz
$ cd petsc-3.13.4
```

Use the following configure options to ensure PETSc is compatible with BOUT++:

```
$ ./configure \
   --with-clanguage=cxx \ 
   --with-mpi=yes \ 
   --with-precision=double \ 
   --with-scalar-type=real \ 
   --with-shared-libraries=0
```

You may also wish to add `--with-debugging=yes` to `./configure` in order to allow debugging.

**Note:** If you build BOUT++ using a standalone version of SUNDIALS, it is advisable to not also build PETSc with SUNDIALS.

**Note:** It is also possible to get PETSc to download and install MUMPS (see MUMPS), by adding:

```
--download-mumps \ 
--download-scalapack \ 
--download-blacs \ 
--download-fblas-lapack=1 \ 
--download-parmetis \ 
--download-ptscotch \ 
--download-metis
```

to `./configure`.

To make PETSc, type:

```
$ make PETSC_DIR=$HOME/petsc-3.13.4 PETSC_ARCH=arch-linux2-cxx-debug all
```

Should BLAS, LAPACK, or any other packages be missing, you will get an error, and a suggestion that you can append `--download-name-of-package` to the `./configure` line. You may want to test that everything is configured properly. To do this, type:

```
$ make PETSC_DIR=$HOME/petsc-3.13.4 PETSC_ARCH=arch-linux2-cxx-debug test
```

To use PETSc, you have to define the PETSC_DIR and PETSC_ARCH environment variables to match how PETSc was built:
$ export PETSC_DIR=$HOME/petsc-3.13.4
$ export PETSC_ARCH=arch-linux2-cxx-debug

and add to your startup file $HOME/.bashrc:

export PETSC_DIR=$HOME/petsc-3.13.4
export PETSC_ARCH=arch-linux2-cxx-debug

To configure BOUT++ with PETSc, go to the BOUT++ root directory, and type:

$ ./configure --with-petsc

You can configure BOUT++ against different PETSc installations either through the PETSC_DIR/ARCH variables as above, or by specifying them on the command line:

$ ./configure --with-petsc PETSC_DIR=/path/to/other/petsc PETSC_ARCH=other-arch

**Note:** Unfortunately, there are a variety of ways PETSc can be installed on a system, and it is hard to automatically work out how to compile against a particular installation. In particular, there are two PETSc-supported ways of installing PETSc that are subtly different.

The first way is as above, using PETSC_DIR and PETSC_ARCH. A second way is to use the --prefix argument to configure (much like the traditional GNU configure scripts) when building PETSc. In this case, PETSC_DIR will be the path passed to --prefix and PETSC_ARCH will be empty. When configuring BOUT++, one can use --with-petsc=$PETSC_DIR as a shortcut in this case. This will NOT work if PETSc was installed with a PETSC_ARCH.

However, there are at least some Linux distributions that install PETSc in yet another way and you may need to set PETSC_DIR/ARCH differently. For example, for Fedora, as of May 2018, you will need to configure and build BOUT++ like so:

$ ./configure --with-petsc=/usr/lib64/openmpi
$ PETSC_DIR=/usr make

Replace openmpi with the correct MPI implementation that you have installed.

### 3.8 LAPACK

BOUT++ comes with linear solvers for tridiagonal and band-diagonal systems. Some implementations of these solvers (for example Laplacian inversion, section *Laplacian inversion*) use LAPACK for efficient serial performance. This does not add new features, but may be faster in some cases. LAPACK is however written in FORTRAN 77, which can cause linking headaches. To enable these routines use:

$ ./configure --with-lapack

and to specify a non-standard path:

$ ./configure --with-lapack=/path/to/lapack
### 3.9 MUMPS

This is still experimental, but does work on at least some systems at York. The PETSc library can be used to call MUMPS for directly solving matrices (e.g. for Laplacian inversions), or MUMPS can be used directly. To enable MUMPS, configure with:

```
$ ./configure --with-mumps
```

MUMPS has many dependencies, including ScaLapack and ParMetis. Unfortunately, the exact dependencies and configuration of MUMPS varies a lot from system to system. The easiest way to get MUMPS installed is to install PETSc with MUMPS, or supply the `CPPFLAGS`, `LDFLAGS` and `LIBS` environment variables to configure:

```
$ ./configure --with-mumps CPPFLAGS=-I/path/to/mumps/includes \
    LDFLAGS=-L/path/to/mumps/libs \n    LIBS="-ldmumps -lmumps_common -lother_libs_needed_for_mumps"
```

### 3.10 MPI compilers

These are usually called something like mpicc and mpiCC (or mpicxx), and the configure script will look for several common names. If your compilers aren’t recognised then set them using:

```
$ ./configure MPICC=<your C compiler> MPICXX=<your C++ compiler>
```

**NOTES:**

- On LLNL’s Grendel, mpicxx is broken. Use mpiCC instead by passing “MPICXX=mpiCC” to configure. Also need to specify this to NetCDF library by passing “CXX=mpiCC” to NetCDF configure.

#### 3.10.1 Installing MPICH from source

In your home directory, create two subdirectories: One called “install” where we’ll put the source code, and one called “local” where we’ll install the MPI compiler:

```
$ cd
$ mkdir install
$ mkdir local
```

Download the latest stable version of MPICH from [https://www.mpich.org/](https://www.mpich.org/) and put the file in the “install” subdirectory created above. At the time of writing (January 2018), the file was called `mpich-3.2.1.tar.gz`. Untar the file:

```
$ tar -xzvf mpich-3.2.1.tar.gz
```

which will create a directory containing the source code. `cd` into this directory and run:

```
$ ./configure --prefix=$HOME/local
$ make
$ make install
```

Each of which might take a while. This is the standard way of installing software from source, and will also be used for installing libraries later. The `-prefix=` option specifies where the software should be installed. Since we don’t have permission to write in the system directories (e.g. `/usr/bin`), we just use a subdirectory of our home directory. The `configure` command configures the install, finding the libraries and commands it needs. `make` compiles everything.
using the options found by configure. The final make install step copies the compiled code into the correct places under $HOME/local.

To be able to use the MPI compiler, you need to modify the PATH environment variable. To do this, run:

```bash
$ export PATH=$PATH:$HOME/local/bin
```

and add this to the end of your startup file $HOME/.bashrc. If you’re using CSH rather than BASH, the command is:

```bash
% setenv PATH ${PATH}:${HOME}/local/bin
```

and the startup file is $HOME/.cshrc. You should now be able to run mpicc and so have a working MPI compiler.

### 3.11 Installing FFTW from source

If you haven’t already, create directories “install” and “local” in your home directory:

```bash
$ cd
$ mkdir install
$ mkdir local
```

Download the latest stable version from http://www.fftw.org/download.html into the “install” directory. At the time of writing, this was called fftw-3.3.2.tar.gz. Untar this file, and 'cd' into the resulting directory. As with the MPI compiler, configure and install the FFTW library into $HOME/local by running:

```bash
$ ./configure --prefix=$HOME/local
$ make
$ make install
```

### 3.12 Compiling and running under AIX

Most development and running of BOUT++ is done under Linux, with the occasional FreeBSD and OSX. The configuration scripts are therefore heavily tested on these architectures. IBM’s POWER architecture however runs AIX, which has some crucial differences which make compiling a pain.

- Under Linux/BSD, it’s usual for a Fortran routine foo to appear under C as foo, whilst under AIX the name is unchanged
- MPI compiler scripts are usually given the names mpicc and either mpiCC or mpiCXX. AIX uses mpcc and mpCC.
- Like BSD, the make command isn’t compatible with GNU make, so you have to run gmake to compile everything.
- The POWER architecture is big-endian, different to the little endian Intel and AMD chips. This can cause problems with binary file formats.
3.12.1 SUNDIALS under AIX

To compile SUNDIALS, use:

```
export CC=cc
export CXX=xlc
export F77=xlf
export OBJECT_MODE=64
./configure --prefix=$HOME/local/ --with-mpicc=mpcc --with-mpif77=mpxlf CFLAGS=-maix64
```

You may get an error message like

```
make: Not a recognized flag: w
```

This is because the AIX `make` is being used, rather than `gmake`. The easiest way to fix this is to make a link to `gmake` in your local bin directory

```
ln -s /usr/bin/gmake $HOME/local/bin/make
```

Running `which make` should now point to this `local/bin/make`, and if not then you need to make sure that your bin directory appears first in the `PATH`

```
export PATH=$HOME/local/bin:$PATH
```

If you see an error like this

```
ar: 0707-126 ../../src/sundials/sundials_math.o is not valid with the current object file mode.
        Use the -X option to specify the desired object mode.
```

then you need to set the environment variable `OBJECT_MODE`

```
export OBJECT_MODE=64
```

Configuring BOUT++, you may get the error

```
configure: error: C compiler cannot create executables
```

In that case, you can try using:

```
./configure CFLAGS="-maix64"
```

When compiling, you may see warnings:

```
xlc_r: 1501-216 (W) command option -64 is not recognized - passed to ld
```

At this point, the main BOUT++ library should compile, and you can try compiling one of the examples.

```
lc: 0711-317 ERROR: Undefined symbol: .NcError::NcError(NcError::Behavior)
lc: 0711-317 ERROR: Undefined symbol: .NcFile::is_valid() const
lc: 0711-317 ERROR: Undefined symbol: .NcError::~NcError()
lc: 0711-317 ERROR: Undefined symbol: .NcFile::get_dim(const char*) const
```

This is probably because the NetCDF libraries are 32-bit, whilst BOUT++ has been compiled as 64-bit. You can try compiling BOUT++ as 32-bit
export OBJECT_MODE=32
./configure CFLAGS="-maix32"
gmake

If you still get undefined symbols, then go back to 64-bit, and edit make.config, replacing -lnetcdf_c++ with -lnetcdf64_c++, and -lnetcdf with -lnetcdf64. This can be done by running

```
    sed 's/netcdf/netcdf64/g' make.config > make.config.new
    mv make.config.new make.config
```

### 3.12.2 Compiling on Windows

It is possible to compile BOUT++ on Windows using the CMake interface. Support is currently very experimental, and some features do not work. Testing has been done with MSVC 19.24 and Visual Studio 16.4, although previous versions may still work.

The main difficulty of using BOUT++ on Windows is getting the dependencies sorted. The easiest way to install dependencies on Windows is using vcpkg. You may need to set the CMake toolchain file if calling cmake from PowerShell, or on older versions of Visual Studio. This will be a file somewhere like `C:/vcpkg/scripts/buildsystems/vcpkg.cmake`

The minimal required CMake options are as follows:

```
-DBUILD_SHARED_LIBS=OFF
-DCMAKE_CXX_FLAGS="/permissive- /EHsc /bigobj"
-DBUILD_SHARED_LIBS=OFF
```

`ENABLE_BACKTRACE` must be turned off due to the currently required `addr2line` executable not being available on Windows.

The following flags for the MSVC compiler are required:

- `/permissive-` for standards compliance, such as treating the binary operator alternative tokens (and, or, etc) as tokens
- `/EHsc` for standard C++ exception handling, and to assume that `extern "C"` functions never throw
- `/bigobj` to increase the number of sections in the .obj file, required for the template-heavy derivatives machinery

No modification to the source has been done to export the correct symbols for shared libraries on Windows, so you must either specify `-DBUILD_SHARED_LIBS=OFF` to only build static libraries, or, if you really want shared libraries, `-DCMAKE_WINDOWS_EXPORT_ALL_SYMBOLS=ON`. The latter is untested, use at your own risk!

The unit tests should all pass, but most of the integrated tests will not run work out of the box yet as Windows doesn’t understand shabangs. That is, without a file extension, it doesn’t know what program to use to run `runtest`. The majority of the tests can be run manually with `python.exe runtest`. You will still need to set PYTHONPATH and have a suitable Python environment.
3.13 Issues

3.13.1 Wrong install script

Before installing, make sure the correct version of `install` is being used by running:

```
$ which install
```

This should point to a system directory like `/usr/bin/install`. Sometimes when IDL has been installed, this points to the IDL install (e.g. something like `/usr/common/usg/idl/idl70/bin/install` on Franklin). A quick way to fix this is to create a link from your local bin to the system install:

```
$ ln -s /usr/bin/install $HOME/local/bin/
```

“which install” should now print the install in your local bin directory.

3.13.2 Compiling cvode.cxx fails

Occasionally compiling the CVODE solver interface will fail with an error similar to:

```
cvode.cxx: In member function 'virtual int CvodeSolver::init(rhsfunc, bool, int, BoutR...
cvode.cxx:234:56: error: invalid conversion from 'int (*)(CVINT...
```

This is caused by different sizes of ints used in different versions of the CVODE library. The configure script tries to determine the correct type to use, but may fail in unusual circumstances. To fix, edit `src/solver/impls/cvode/cvode.cxx`, and change line 48 from

```
typedef int CVODEINT;
```

to

```
typedef long CVODEINT;
```

3.13.3 Compiling with IBM xlC compiler fails

When using the xlC compiler, an error may occur:

```
variant.hpp(1568) parameter pack "Ts" was referenced but not expanded
```

The workaround is to change line 428 of `externalpackages/mpark.variant/include/mpark/lib.hpp` from:

```
#ifdef MPARK_TYPE_PACK_ELEMENT
```

to:

```
#ifdef CAUSES_ERROR // MPARK_TYPE_PACK_ELEMENT
```

This will force an alternate implementation of type_pack_element to be defined. See also https://software.intel.com/en-us/forums/intel-c-compiler/topic/501502
3.13.4 Compiling fails after changing branch

If compiling fails after changing branch, for example from master to next, with an error like the following:

```
$ make
Downloading mpark.variant
You need to run this command from the toplevel of the working tree.
make[2]: *** [BOUT-dev/externalpackages/mpark.variant/include/mpark/variant.hpp] Error 1
make[1]: *** [field] Error 2
make: *** [src] Error 2
```

It's possible something has gone wrong with the submodules. To fix, just run `make submodules`:

```
$ make submodules
Downloading mpark.variant
git submodule update --init --recursive /home/peter/Codes/BOUT-dev/externalpackages/
  → mpark.variant
Submodule path 'externalpackages/mpark.variant': checked out
  → '0b488da9bebac980e7ba0e158a959c956a449676'
```

If you regularly work on two different branches and need to run `make submodules` a lot, you may consider telling git to automatically update the submodules:

```
git config submodule.recurse=true
```

This requires git >= 2.14.
4.1 Quick start

The examples/ directory contains some example physics models for a variety of fluid models. There are also some under tests/integrated/, which often just run a part of the code rather than a complete simulation. The simplest example to start with is examples/conduction/. This solves a single equation for a 3D scalar field $T$:

$$\frac{\partial T}{\partial t} = \nabla ||(\chi \partial ||T)$$

There are several files involved:

- **conduction.cxx** contains the source code which specifies the equation to solve. See Heat conduction for a line-by-line walkthrough of this file.
- **conduct_grid.nc** is the grid file, which in this case just specifies the number of grid points in $X$ and $Y$ (nx & ny) with everything else being left as the default (e.g. grid spacings dx and dy are 1, the metric tensor is the identity matrix). For details of the grid file format, see Generating input grids.
- **generate.py** is a Python script to create the grid file. In this case it just writes nx and ny.
- **data/BOUT.inp** is the settings file, specifying how many output timesteps to take, differencing schemes to use, and many other things. In this case it’s mostly empty so the defaults are used.

First you need to compile the example:

```
$ gmake
```

which should print out something along the lines of:

Compiling conduction.cxx
Linking conduction

If you get an error, most likely during the linking stage, you may need to go back and make sure the libraries are all set up correctly. A common problem is mixing MPI implementations, for example compiling NetCDF using Open MPI and then BOUT++ with MPICH2. Unfortunately the solution is to recompile everything with the same compiler.

Then try running the example. If you’re running on a standalone server, desktop or laptop then try:

```
$ mpirun -np 2 ./conduction
```

If you’re running on a cluster or supercomputer, you should find out how to submit jobs. This varies, but usually on these bigger machines there will be a queueing system and you’ll need to use qsub, msub, llsubmit or similar to submit jobs.
When the example runs, it should print a lot of output. This is recording all the settings being used by the code, and is also written to log files for future reference. The test should take a few seconds to run, and produce a bunch of files in the data/ subdirectory.

- `BOUT.log.*` contains a log from each process, so because we ran with “-np 2” there should be 2 logs. The one from processor 0 will be the same as what was printed to the screen. This is mainly useful because if one process crashes it may only put an error message into its own log.

- `BOUT.settings` contains all the options used in the code, including options which were not set and used the default values. It's in the same format as BOUT.inp, so can be renamed and used to re-run simulations if needed. In some cases the options used have documentation, with a brief explanation of how they are used. In most cases the type the option is used as (e.g. int, BoutReal or bool) is given.

- `BOUT.restart.*.nc` are the restart files for the last time point. Currently each processor saves its own state in a separate file, but there is experimental support for parallel I/O. For the settings, see Input and Output.

- `BOUT.dmp.*.nc` contain the output data, including time history. As with the restart files, each processor currently outputs a separate file.

Restart files allow the run to be restarted from where they left off:

```
$ mpirun -np 2 ./conduction restart
```

This will delete the output data `BOUT.dmp.*.nc` files, and start again. If you want to keep the output from the first run, add “append”:

```
$ mpirun -np 2 ./conduction restart append
```

which will then append any new outputs to the end of the old data files. For more information on restarting, see Restarting runs.

To see some of the other command-line options try “-h”:

```
$ ./conduction -h
```

and see the section on options (BOUT++ options).

To analyse the output of the simulation, cd into the data subdirectory and start Python.

### 4.1.1 Analysing the output Using python

**Note:** We now recommend using xBOUT to analyse BOUT++ simulations.

In order to analyse the output of the simulation using Python, you will first need to have set up python to use the BOUT++ libraries boutdata and boututils; see section Python configuration for how to do this. The analysis routines have some requirements such as SciPy; see section Requirements for details.

To print a list of variables in the output files, one way is to use the `DataFile` class. This is a wrapper around the various NetCDF and HDF5 libraries for python:

```python
>>> from boututils.datafile import DataFile
>>> DataFile("BOUT.dmp.0.nc").list()
```

To collect a variable, reading in the data as a NumPy array:
>>> from boutdata.collect import collect
>>> T = collect("T")
>>> T.shape

Note that the order of the indices is different in Python and IDL: In Python, 4D variables are arranged as [t, x, y, z]. To show an animation

```python
>>> from boututils.showdata import showdata
>>> showdata(T[:,0,:,0])
```

The first index of the array passed to `showdata` is assumed to be time, and the remaining indices are plotted. In this example we pass a 2D array [t,y], so `showdata` will animate a line plot.

## 4.2 Natural language support

If you have locales installed, and configured the `locale` path correctly (see `Natural Language Support`), then the `LANG` environment variable selects the language to use. Currently BOUT++ only has support for `fr`, `de`, `es`, `zh_TW` and `zh_CN` locales e.g.

```
LANG=zh_TW.utf8 ./conduction
```

which should produce an output like:

```
BOUT++ ? 4.3.0
?: 667c19c136fc3e72fcd7c7b2109d44886fd818d
MD5 checksum: 2263dc17fa414179c7ad87c3972f624b
??? Nov 21 2019 17:26:55 ??
...
```

or

```
LANG=es_ES.utf8 ./conduction
```

which should produce:

```
Versión de BOUT++ 4.3.0
Revisión: 667c19c136fc3e72fcd7c7b2109d44886fd818d
MD5 checksum: 2263dc17fa414179c7ad87c3972f624b
Código compilado en Nov 21 2019 en 17:26:55
...
```

The name of the locale (`zh_TW.utf8` or `es_ES.utf8` above) can be different on different machines. To see a list of available locales on your system try running:

```
locale -a
```

If you are missing a locale you need, see your distribution's help, or try this Arch wiki page on locale.
4.3 When things go wrong

BOUT++ is still under development, and so occasionally you may be lucky enough to discover a new bug. This is particularly likely if you’re modifying the physics module source code (see BOUT++ physics models) when you need a way to debug your code too.

- Check the end of each processor’s log file (tail data/BOUT.log.*). When BOUT++ exits before it should, what is printed to screen is just the output from processor 0. If an error occurred on another processor then the error message will be written to it’s log file instead.

- By default when an error occurs a kind of stack trace is printed which shows which functions were being run (most recent first). This should give a good indication of where an error occurred. If this stack isn’t printed, make sure checking is set to level 2 or higher (.configure --enable-checks=2).

- If the error is due to non-finite numbers, increase the checking level (.configure --enable-checks=3) to perform more checking of values and (hopefully) find an error as soon as possible after it occurs.

- If the error is a segmentation fault, you can try a debugger such as gdb or totalview. You will likely need to compile with some debugging flags (.configure --enable-debug).

- You can also enable exceptions on floating point errors (.configure --enable-sigfpe), though the majority of these types of errors should be caught with checking level set to 3.

- Expert users can try AddressSanitizer, which is a tool that comes with recent versions of GCC and Clang. To enable AddressSanitizer, include -fsanitize=leak -fsanitize=address -fsanitize=undefined in CXXFLAGS when configuring BOUT++, or add them to BOUT_FLAGS.

4.4 Startup output

When BOUT++ is run, it produces a lot of output initially, mainly listing the options which have been used so you can check that it’s doing what you think it should be. It’s generally a good idea to scan over this see if there are any important warnings or errors. Each processor outputs its own log file BOUT.log.# and the log from processor 0 is also sent to the screen. This output may look a little different if it’s out of date, but the general layout will probably be the same. The exact order that options are printed in may also vary between versions and models.

First comes the introductory blurb:

```
BOUT++ version 4.4.0
Revision: 7cfbc6890a82cb6b3b6c81870d8a8fca723de542
Code compiled on Dec 7 2021 at 15:14:05

B.Dudson (University of York), M.Umansky (LLNL) 2007
Based on BOUT by Xueqiao Xu, 1999
```

The version number (4.4.0 here) gets increased occasionally after some major feature has been added. To help match simulations to code versions, the Git revision of the core BOUT++ code and the date and time it was compiled is recorded. This information makes it possible to verify precisely which version of the code was used for any given run.

The processor number comes next:

```
Processor number: 0 of 1
```

This will always be processor number ‘0’ on screen as only the output from processor ‘0’ is sent to the terminal.

The process ID (pid) is also printed:
pid: 17835

which is useful for distinguishing multiple simulations running at the same time and, for example, to stop one run if it starts misbehaving.

Next comes the compile-time options, which depend on how BOUT++ was configured (see \textit{Compiling BOUT++}):

<table>
<thead>
<tr>
<th>Compile-time options:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checking enabled, level 2</td>
</tr>
<tr>
<td>Signal handling enabled</td>
</tr>
<tr>
<td>netCDF support enabled</td>
</tr>
<tr>
<td>Parallel NetCDF support disabled</td>
</tr>
<tr>
<td>OpenMP parallelisation disabled</td>
</tr>
<tr>
<td>Compiled with flags : &quot;-Wall -Wextra ...&quot;</td>
</tr>
</tbody>
</table>

This says that some run-time checking of values is enabled, that the code will try to catch segmentation faults to print a useful error, that NetCDF files are supported, but that the parallel flavour isn’t. The compilation flags are printed, which can be useful for checking if a run was built with optimisation or debugging enabled. These flags can be quite long, so we’ve truncated them in the snippet above.

The complete command line is printed (excluding any MPI options):

| Command line options for this run : ./conduction nout=1 |

After this the core BOUT++ code reads some options:

<table>
<thead>
<tr>
<th>Reading options file data/BOUT.inp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option nout = 100 (data/BOUT.inp) overwritten with:</td>
</tr>
<tr>
<td>nout = 1 (Command line)</td>
</tr>
<tr>
<td>Writing options to file data/BOUT.settings</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Getting grid data from options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option mesh:type = bout (default)</td>
</tr>
<tr>
<td>Option mesh:StaggerGrids = 0 (default)</td>
</tr>
<tr>
<td>Option mesh:maxregionblocksize = 64 (default)</td>
</tr>
<tr>
<td>Option mesh:calcParallelSlices_on_communicate = 1 (default)</td>
</tr>
<tr>
<td>Option mesh:ddx:fft_filter = 0 (default)</td>
</tr>
<tr>
<td>Option mesh:ddx:first = c2 (data/BOUT.inp)</td>
</tr>
<tr>
<td>Option mesh:ddx:second = c2 (data/BOUT.inp)</td>
</tr>
<tr>
<td>Option mesh:ddx:upwind = w3 (data/BOUT.inp)</td>
</tr>
<tr>
<td>Option mesh:ddy:first = c2 (data/BOUT.inp)</td>
</tr>
<tr>
<td>Option mesh:ddy:second = c2 (data/BOUT.inp)</td>
</tr>
<tr>
<td>Option mesh:ddy:upwind = w3 (data/BOUT.inp)</td>
</tr>
<tr>
<td>Option mesh:ddz:first = fft (data/BOUT.inp)</td>
</tr>
<tr>
<td>Option mesh:ddz:second = fft (data/BOUT.inp)</td>
</tr>
<tr>
<td>Option mesh:ddz:upwind = w3 (data/BOUT.inp)</td>
</tr>
</tbody>
</table>

This lists each option and the value it has been assigned. For every option the source of the value being used is also given. If a value had been given on the command line then (command line) would appear after the option.:

| Option mesh:ddx:first = c2 (data/BOUT.inp) |
| Option mesh:ddx:second = c2 (data/BOUT.inp) |
| Option mesh:ddx:upwind = w3 (data/BOUT.inp) |
| Option mesh:ddy:first = c2 (data/BOUT.inp) |
| Option mesh:ddy:second = c2 (data/BOUT.inp) |
| Option mesh:ddy:upwind = w3 (data/BOUT.inp) |
| Option mesh:ddz:first = fft (data/BOUT.inp) |
| Option mesh:ddz:second = fft (data/BOUT.inp) |
| Option mesh:ddz:upwind = w3 (data/BOUT.inp) |

This is a list of the differential methods for each direction. These are set in the BOUT.inp file ([mesh:ddx], [mesh:ddy] and [mesh:ddz] sections), but can be overridden for individual operators. For each direction, numerical
methods can be specified for first and second central difference terms, upwinding terms of the form \( \frac{\partial f}{\partial t} = v \cdot \nabla f \), and flux terms of the form \( \frac{\partial f}{\partial t} = \nabla \cdot (v f) \). By default the flux terms are just split into a central and an upwinding term. A list of available methods is given in *Differencing methods*.

Loading mesh

```plaintext
Option input:transform_from_field_aligned = 1 (default)
Option mesh:nx = 1 (data/BOUT.inp)
Option mesh:ny = 100 (data/BOUT.inp)
Option mesh:nz = 1 (data/BOUT.inp)
Read nz from input grid file
Grid size: 1 x 100 x 1
Variable 'MXG' not in mesh options. Setting to 0
Option mxg = 0 (data/BOUT.inp)
Variable 'MYG' not in mesh options. Setting to 0
Option MYG = 2 (default)
Guard cells (x,y,z): 0, 2, 0
Option mesh:ixseps1 = -1 (data/BOUT.inp)
Option mesh:ixseps2 = -1 (data/BOUT.inp)
```

Optional quantities (such as MXG/MYG in this case) which are not specified are given a default (best-guess) value, and a warning is printed.

```
EQUILIBRIUM IS SINGLE NULL (SND)
MYPE_IN_CORE = 0
DXS = 0, DIN = -1, DOUT = -1
UXS = 0, UIN = -1, UOUT = -1
XIN = -1, XOUT = -1
Twist-shift:
```

At this point, BOUT++ reads the grid file, and works out the topology of the grid, and connections between processors. BOUT++ then tries to read the metric coefficients from the grid file:

```
Variable 'g11' not in mesh options. Setting to 1.000000e+00
Variable 'g22' not in mesh options. Setting to 1.000000e+00
Variable 'g33' not in mesh options. Setting to 1.000000e+00
Variable 'g12' not in mesh options. Setting to 0.000000e+00
Variable 'g13' not in mesh options. Setting to 0.000000e+00
Variable 'g23' not in mesh options. Setting to 0.000000e+00
```

These warnings are printed because the coefficients have not been specified in the grid file, and so the metric tensor is set to the default identity matrix. For this particular example we don’t need to do anything special in the direction parallel to the magnetic field, so we set the parallel transform to be the identity (see *Parallel Transforms*):

```
Option mesh:paralleltransform = identity (default)
```

If only the contravariant components (g11 etc.) of the metric tensor are specified, the covariant components (g_11 etc.) are calculated by inverting the metric tensor matrix. Error estimates are then calculated by calculating \( g_{ij} g^{jk} \) as a check. Since no metrics were specified in the input, the metric tensor was set to the identity matrix, making inversion easy and the error tiny.

```
Variable 'J' not in mesh options. Setting to 0.000000e+00
  WARNING: Jacobian 'J' not found. Calculating from metric tensor
Variable 'Bxy' not in mesh options. Setting to 0.000000e+00
  WARNING: Magnitude of B field 'Bxy' not found. Calculating from metric tensor
```

(continues on next page)
Calculating differential geometry terms
Communicating connection terms
Boundary regions in this processor: upper_target, lower_target,
Constructing default regions

The Laplacian inversion (see *Laplacian inversion*) code is initialised, and prints out the options used.:

```plaintext
Initialising Laplacian inversion routines
  Option phiboussinesq:async = 1 (default)
  Option phiboussinesq:filter = 0 (default)
  Option phiboussinesq:maxmode = 128 (default)
  Option phiboussinesq:low_mem = 0 (default)
  Option phiboussinesq:nonuniform = 1 (default)
  Option phiboussinesq:all_terms = 1 (default)
  Option phiboussinesq:flags = 0 (delta_1/BOUT.inp)
```

After this comes the physics module-specific output:

```plaintext
Initialising physics module
  Option solver:type = cvode (default)
```

This typically lists the options used, useful/important normalisation factors, and so on.

Finally, once the physics module has been initialised, and the current values loaded, the solver can be started:

```plaintext
Initialising solver
  Option datadir = delta_1 ()
  Option dump_format = nc (default)
  Option restart_format = nc (default)
  Using NetCDF4 format for file 'delta_1/BOUT.restart.nc'

Constructing default regions
  Boundary region inner X
  Boundary region outer X
  3d fields = 2, 2d fields = 0 neq=100, local_N=100
```

This last line gives the number of equations being evolved (in this case 100), and the number of these on this processor (here 100).:

The absolute and relative tolerances come next:

```plaintext
Option solver:atol = 1e-12 (default)
Option solver:rtol = 1e-05 (default)
```

This next option specifies the maximum number of internal timesteps that CVODE will take between outputs.:

```plaintext
Option solver:mxstep = 500 (default)
```

After (almost!) all of the options are read in, the simulation proper starts:

```plaintext
Running simulation
Run ID: 332467c7-1210-401a-b44c-f8a3a3415827
Run started at : Tue 07 Dec 2021 17:50:39 GMT
```
The Run ID here is a universally unique identifier (UUID) which is a random 128-bit label unique to this current simulation. This makes it easier to identify all of the associated outputs of a simulation, and record the data for future reference.

A few more options may appear between these last progress messages and the per-timestep output discussed in the next section.

### 4.5 Per-timestep output

At the beginning of a run, just after the last line in the previous section, a header is printed out as a guide:

```
<table>
<thead>
<tr>
<th>Sim Time</th>
<th>RHS evals</th>
<th>Wall Time</th>
<th>Calc</th>
<th>Inv</th>
<th>Comm</th>
<th>I/O</th>
<th>SOLVER</th>
</tr>
</thead>
</table>
```

Each timestep (the one specified in BOUT.inp, not the internal timestep), BOUT++ prints out something like:

```
1.001e+02 76 2.27e+02 87.1 5.3 1.0 0.0 6.6
```

This gives the simulation time; the number of times the time-derivatives (RHS) were evaluated; the wall-time this took to run, and percentages for the time spent in different parts of the code.

- **Calc** is the time spent doing calculations such as multiplications, derivatives etc
- **Inv** is the time spent in inversion code (i.e. inverting Laplacians), including any communication which may be needed to do the inversion.
- **Comm** is the time spent communicating variables (outside the inversion routine)
- **I/O** is the time spent writing dump and restart files to disk. Most of the time this should not be an issue
- **SOLVER** is the time spent in the implicit solver code.

The output sent to the terminal (not the log files) also includes a run time, and estimated remaining time.

### 4.6 Restarting runs

Every output timestep, BOUT++ writes a set of files named “BOUT.restart.#.nc” where ‘#’ is the processor number (for parallel output, a single file “BOUT.restart.nc” is used). To restart from where the previous run finished, just add the keyword **restart** to the end of the command, for example:

```
$ mpirun -np 2 ./conduction restart
```

Equivalently, put “restart=true” near the top of the BOUT.inp input file. Note that this will overwrite the existing data in the BOUT.dmp./*.nc files. If you want to append to them instead then add the keyword **append** to the command, for example:

```
$ mpirun -np 2 ./conduction restart append
```

or also put **append=true** near the top of the BOUT.inp input file.

When restarting simulations BOUT++ will by default output the initial state, unless appending to existing data files when it will not output until the first timestep is completed. To override this behaviour, you can specify the option **dump_on_restart** manually. If **dump_on_restart** is true then the initial state will always be written out, if false then it never will be (regardless of the values of **restart** and **append**).

If you need to restart from a different point in your simulation, or the BOUT.restart files become corrupted, you can use xBOUT to create new restart files from any time-point in your output files. Use the **.to_restart()** method:

40 Chapter 4. Running BOUT++
>>> import xbout
>>> df = xbout.open_boutdataset("data/BOUT.dmp.*.nc")
>>> df.bout.to_restart(tind=10)

The above will take time point 10 from the BOUT.dmp.*.nc files in the data directory. For each one, it will output a BOUT.restart.*.nc file in the output directory.

4.7 Stopping simulations

If you need to stop a simulation early this can be done by Ctrl-C in a terminal, but this will stop the simulation immediately without shutting down cleanly. Most of the time this will be fine, but interrupting a simulation while it is writing data to file could result in inconsistent or corrupted data.

4.7.1 Stop file

Note This method needs to be enabled before the simulation starts by setting stopCheck=true on the command line or input options:

$ mpirun -np 4 ./conduction stopCheck=true

or in the top section of BOUT.inp set stopCheck=true.

At every output time, the monitor checks for the existence of a file, by default called BOUT.stop, in the same directory as the output data. If the file exists then the monitor signals the time integration solver to quit. This should result in a clean shutdown.

To stop a simulation using this method, just create an empty file in the output directory:

$ mpirun -np 4 ./conduction stopCheck=true
...$ touch data/BOUT.stop

just remember to delete the file afterwards.

4.7.2 Send signal USR1

Another option is to send signal user defined signal 1:

$ mpirun -np 4 ./conduction &
...$ killall -s USR1 conduction

Note that this will stop all conduction simulation on this node. Many HPC systems provide tools to send signals to the simulation nodes, such as qsig on archer.

To just stop one simulation, the bout-stop-script can send a signal based on the path of the simulation data dir:

$ mpirun -np 4 ./conduction &
...$ bout-stop-script data

This will stop the simulation cleanly, and:
$ mpirun -np 4 ./conduction &
...
$ bout-stop-script data -force

will kill the simulation immediately.

## 4.8 Manipulating restart files

It is sometimes useful to change the number of processors used in a simulation, or to modify restart files in various ways. For example, a 3D turbulence simulation might start with a quick 2D simulation with diffusive transport to reach a steady-state. The restart files can then be extended into 3D, noise added to seed instabilities, and the files split over a more processors.

Routines to modify restart files are in `tools/pylib/boutdata/restart.py`:

```python
>>> from boutdata import restart
>>> help(restart)
```

### 4.8.1 Changing number of processors

To change the number of processors use the `redistribute` function:

```python
>>> from boutdata import restart
>>> restart.redistribute(32, path="../oldrun", output=".")
```

where in this example 32 is the number of processors desired; path sets the path to the existing restart files, and output is the path where the new restart files should go. **Note** Make sure that path and output are different.

If your simulation is divided in X and Y directions then you should also specify the number of processors in the X direction, `NXPE`:

```python
>>> restart.redistribute(32, path="../oldrun", output=".", nxpe=8)
```

**Note** Currently this routine doesn’t check that this split is consistent with branch cuts, e.g. for X-point tokamak simulations. If an inconsistent choice is made then the BOUT++ restart will fail.

**Note** It is a good idea to set `nxpe` in the `BOUT.inp` file to be consistent with what you set here. If it is inconsistent then the restart will fail, but the error message may not be particularly enlightening.
Once you have tried some example codes, and generally got the hang of running BOUT++ and analysing the results, there will probably come a time when you want to change the equations being solved. This section demonstrates how a BOUT++ physics model is put together. It assumes you have a working knowledge of C or C++, but you don’t need to be an expert - most of the messy code is hidden away from the physics model. There are several good books on C and C++, but I’d recommend online tutorials over books because there are a lot more of them, they’re quicker to scan through, and they’re cheaper.

Many of the examples which come with BOUT++ are physics models, and can be used as a starting point. Some relatively simple examples are blob2d (2D plasma filament/blob propagation), hasegawa-wakatani (2D turbulence), finite-volume/fluid (1D compressible fluid) and gas-compress (up to 3D compressible fluid). Some of the integrated tests (under tests/integrated) use either physics models (e.g. test-delp2 and test-drift-instability), or define their own main function (e.g. test-io and test-cyclic).

5.1 Heat conduction

The conduction example solves 1D heat conduction

$$\frac{\partial T}{\partial t} = \nabla |\chi \nabla T|$$

The source code to solve this is in conduction.cxx, which we show here:

```cpp
#include <bout/physicsmodel.hxx>

class Conduction : public PhysicsModel {
private:
    Field3D T; // Evolving temperature equation only
    BoutReal chi; // Parallel conduction coefficient
protected:
    // This is called once at the start
    int init(bool UNUSED(restarting)) override {
        // Get the options
        auto& options = Options::root()"conduction";
        // Read from BOUT.inp, setting default to 1.0
```
Let’s go through it line-by-line. First, we include the header that defines the `PhysicsModel` class:

```cpp
#include <bout/physicsmodel.hxx>
```

This also brings in the header files that we need for the rest of the code. Next, we need to define a new class, `Conduction`, that inherits from `PhysicsModel` (line 8):

```cpp
class Conduction : public PhysicsModel {
```

The `PhysicsModel` contains both the physical variables we want to evolve, like the temperature:

```cpp
Field3D T; // Evolving temperature equation only
```

as well as any physical or numerical coefficients. In this case, we only have the parallel conduction coefficient, `chi`:

```cpp
BoutReal chi; // Parallel conduction coefficient
```

A `Field3D` represents a 3D scalar quantity, while a `BoutReal` represents a single number. See the later section on `Variables` for more information.

After declaring our model variables, we need to define two functions: an initialisation function, `init`, that is called to set up the simulation and specify which variables are evolving in time; and a “right-hand side” function, `rhs`, that calculates the time derivatives of our evolving variables. These are defined in lines 18 and 21 respectively above:

```cpp
int init(bool restarting) override {
    ...
}  // The doc() provides some documentation in BOUT.settings
int rhs(BoutReal time) override {
    return 0;
}
```

`PhysicsModel::init()` takes as input a `bool (true or false)` that tells it whether or not the model is being restarted, which can be useful if something only needs to be done once before the simulation starts properly. The simulation
(physical) time is passed to \texttt{PhyiscsModel::rhs} as a \textit{BoutReal}.

The \texttt{override} keyword is just to let the compiler know we’re overriding a method in the base class and is not important to understand.

### 5.1.1 Initialisation

During initialisation (the \texttt{init} function), the conduction example first reads an option (lines 21 and 24) from the input settings file (\texttt{data/BOUT.inp} by default):

```cpp
auto options = Options::root()[:"conduction"];
OPTION(options, chi, 1.0);
```

This first gets a section called “conduction”, then requests an option called “chi” inside this section. If this setting is not found, then the default value of 1.0 will be used. To set this value the \texttt{BOUT.inp} file contains:

```
[conduction]
chi = 1.0
```

which defines a section called “conduction”, and within that section a variable called “chi”. This value can also be overridden by specifying the setting on the command line:

```
$ ./conduction conduction:chi=2
```

where \texttt{conduction:chi} means the variable “chi” in the section “conduction”. When this option is read, a message is printed to the \texttt{BOUT.log} files, giving the value used and the source of that value:

```
Option conduction:chi = 1 (data/BOUT.inp)
```

For more information on options and input files, see \textit{BOUT++ options}, as well as the documentation for the \texttt{Options} class.

After reading the chi option, the \texttt{init} method then specifies which variables to evolve using the \texttt{SOLVE_FOR} macro:

```cpp
// Tell BOUT++ to solve T
SOLVE_FOR(T);
```

This tells the BOUT++ time integration solver to set the variable T using values from the input settings. It looks in a section with the same name as the variable (T here) for variables “scale” and “function”:

```
[T] # Settings for the T variable
scale = 1.0 # Size of the initial perturbation
function = gauss(y-pi, 0.2) # The form of the initial perturbation. y from 0 to 2*pi
```

The function is evaluated using expressions which can involve x, y and z coordinates. More details are given in section \textit{Initialisation of time evolved variables}.

Finally an error code is returned, here 0 indicates no error. If \texttt{init} returns non-zero then the simulation will stop.
5.1.2 Time evolution

During time evolution, the time integration method (ODE integrator) calculates the system state (here \( T \)) at a given time. It then calls the `PhysicsModel::rhs()` function, which should calculate the time derivative of all the evolving variables. In this case, the job of the `rhs` function is to calculate \( \frac{dT}{dt} \), the partial derivative of the variable \( T \) with respect to time, given the value of \( T \):

\[
\frac{\partial T}{\partial t} = \nabla_{||}\left(\chi \frac{\partial}{\partial t} T\right)
\]

The first thing the `rhs` function does is communicate the guard (halo) cells using `Mesh::communicate()` on line 33:

```cpp
mesh->communicate(T);
```

This is because BOUT++ does not (generally) do communications, but leaves it up to the user to decide when the most efficient or convenient time to do them is. Before we can take derivatives of a variable (here \( T \)), the values of the function must be known in the boundaries and guard cells, which requires communication between processors. By default, the values in the guard cells are set to \( NaN \), so if they are accidentally used without first communicating then the code should crash fairly quickly with a non-finite number error.

Once the guard cells have been communicated, we calculate the right hand side (RHS) of the equation above (line 35):

```cpp
ddt(T) = Div_par_K_Grad_par(chi, T);
```

The function `Div_par_K_Grad_par()` is a function in the BOUT++ library which calculates the divergence in the parallel (y) direction of a constant multiplied by the gradient of a function in the parallel direction.

As with the `init` code, a non-zero return value indicates an error and will stop the simulation.

5.1.3 Running the model

The very last thing we need to do in our physics model is to define a `main` function. Here, we do it with the `BOUTMAIN` macro:

```cpp
BOUTMAIN(Conduction);
```

You can define your own `main()` function, but for most cases this is enough. The macro expands to something like:

```cpp
int main(int argc, char **argv) {
    BoutInitialise(argc, argv); // Initialise BOUT++

    Conduction *model = new Conduction(); // Create a model

    Solver *solver = Solver::create(); // Create a solver
    solver->setModel(model); // Specify the model to solve
    solver->addMonitor(bout_monitor); // Monitor the solver

    solver->solve(); // Run the solver

    delete model;
    delete solver;
    BoutFinalise(); // Finished with BOUT++
}
```

(continues on next page)
This initialises the main BOUT++ library, creates the `PhysicsModel` and `Solver`, runs the solver, and finally cleans up the model, solver and library.

## 5.2 Magnetohydrodynamics (MHD)

When going through this section, it may help to refer to the finished code, which is given in the file `mhd.cxx` in the BOUT++ examples directory under `orszag-tang`. The equations to be solved are:

\[
\begin{align*}
\frac{\partial \rho}{\partial t} &= -\mathbf{v} \cdot \nabla \rho - \rho \nabla \cdot \mathbf{v} \\
\frac{\partial p}{\partial t} &= -\mathbf{v} \cdot \nabla p - \gamma p \nabla \cdot \mathbf{v} \\
\frac{\partial \mathbf{v}}{\partial t} &= -\mathbf{v} \cdot \nabla \mathbf{v} + \frac{1}{\rho} (\nabla p + (\nabla \times \mathbf{B}) \times \mathbf{B}) \\
\frac{\partial \mathbf{B}}{\partial t} &= \nabla \times (\mathbf{v} \times \mathbf{B})
\end{align*}
\]

As in the `heat conduction example`, a class is created which inherits from `PhysicsModel` and defines `init` and `rhs` functions:

```cpp
class MHD : public PhysicsModel {
    private:
        int init(bool restarting) override {
            ...
        }
        int rhs(BoutReal t) override {
            ...
        }
};
```

The `init` function is called once at the start of the simulation, and should set up the problem, specifying which variables are to be evolved. The argument `restarting` is false the first time a problem is run, and true if loading the state from a restart file.

The `rhs` function is called every time-step, and should calculate the time-derivatives for a given state. In both cases returning non-zero tells BOUT++ that an error occurred.

### 5.2.1 Variables

We need to define the variables to evolve as member variables (so they can be used in `init` and `rhs`).

For ideal MHD, we need two 3D scalar fields density $\rho$ and pressure $p$, and two 3D vector fields velocity $\mathbf{v}$, and magnetic field $\mathbf{B}$:

```cpp
class MHD : public PhysicsModel {
    private:
        Field3D rho, p; // 3D scalar fields
        Vector3D v, B; // 3D vector fields
};
```
Scalar and vector fields behave much as you would expect: Field3D objects can be added, subtracted, multiplied and divided, so the following examples are all valid operations:

```cpp
Field3D a, b, c;
BoutReal r;

a = b + c; a = b - c;
a = b * c; a = r * b;
a = b / c; a = b / r; a = r / b;
```

Similarly, vector objects can be added/subtracted from each other, multiplied/divided by scalar fields and real numbers, for example:

```cpp
Vector3D a, b, c;
Field3D f;
BoutReal r;

a = b + c; a = b - c;
a = b * f; a = b * r;
a = b / f; a = b / r;
```

In addition the dot and cross products are represented by * and ∧ symbols:

```cpp
Vector3D a, b, c;
Field3D f;

f = a * b // Dot-product
a = b ^ c // Cross-product
```

For both scalar and vector field operations, so long as the result of an operation is of the correct type, the usual C/C++ shorthand notation can be used:

```cpp
Field3D a, b;
Vector3D v, w;

a += b; v *= a; v -= w; v ^= w; // valid
v *= w; // NOT valid: result of dot-product is a scalar
```

**Note:** The operator precedence for ∧ is lower than +, * and / so it is recommended to surround a ^ b with braces.

### 5.2.2 Evolution equations

At this point we can tell BOUT++ which variables to evolve, and where the state and time-derivatives will be stored. This is done using the `bout_solve(variable, name)` function in your physics model `init`:

```cpp
int init(bool restarting) {
    bout_solve(rho, "density");
    bout_solve(p, "pressure");
    v.covariant = true; // evolve covariant components
}
```

bout_solve(v, "v");
B.covariant = false; // evolve contravariant components
bout_solve(B, "B");

return 0;
}

The name given to this function will be used in the output and restart data files. These will be automatically read and written depending on input options (see \textit{BOUT++ options}). Input options based on these names are also used to initialise the variables.

You can add a description of the variable which will be saved as an attribute in the output files by adding a third argument to \texttt{bout\_solve()} e.g.:

\begin{verbatim}
bout_solve(rho, "density", "electron density");
bout_solve(B, "B", "total magnetic field strength");
\end{verbatim}

If the name of the variable in the output file is the same as the variable name, you can use a shorthand macro. In this case, we could use this shorthand for \texttt{v} and \texttt{B}:

\begin{verbatim}
SOLVE\_FOR(v);
SOLVE\_FOR(B);
\end{verbatim}

To make this even shorter, multiple fields can be passed to \texttt{SOLVE\_FOR} (up to 10 at the time of writing). We can also use macros \texttt{SOLVE\_FOR2}, \texttt{SOLVE\_FOR3}, \ldots, \texttt{SOLVE\_FOR6} which are used in many models. Our initialisation code becomes:

\begin{verbatim}
int \textbf{init} (\textbf{bool} restarting) \textbf{override} {
  ...
  bout_solve(rho, "density");
  bout_solve(p, "pressure");
  v.covariant = \textbf{true}; // evolve covariant components
  B.covariant = \textbf{false}; // evolve contravariant components
  SOLVE\_FOR(v, B);
  ...
  return 0;
}
\end{verbatim}

Vector quantities can be stored in either covariant or contravariant form. The value of the \texttt{Vector3D::covariant} property when \texttt{PhysicsModel::bout\_solve()} (or \texttt{SOLVE\_FOR}) is called is the form which is evolved in time and saved to the output file.

The equations to be solved can now be written in the \texttt{rhs} function. The value passed to the function (\texttt{BoutReal t}) is the simulation time - only needed if your equations contain time-dependent sources or similar terms. To refer to the time-derivative of a variable \texttt{var}, use \texttt{ddt(var)}. The ideal MHD equations can be written as:

\begin{verbatim}
int \textbf{rhs} (\textbf{BoutReal} t) \textbf{override} {
  ddt(rho) = -V\_dot\_Grad(v, rho) - rho\_Div(v);
  ddt(p) = -V\_dot\_Grad(v, p) - g\_p\_Div(v);
  ddt(v) = -V\_dot\_Grad(v, v) + (\texttt{Curl(B)}\_B) - Grad(p) \over rho;
  ddt(B) = \texttt{Curl(v}\_B);
}
\end{verbatim}

Where the differential operators \texttt{vector = Grad(scalar)}, \texttt{scalar = Div(vector)}, and \texttt{vector = Curl(vector)} are used. For the density and pressure equations, the \(v \cdot \nabla \rho\) term could be written as \(v^\ast \texttt{Grad}(\rho)\),
but this would then use central differencing in the Grad operator. Instead, the function \( \nabla \cdot \text{Grad}() \) uses upwinding methods for these advection terms. In addition, the Grad() function will not operate on vector objects (since result is neither scalar nor vector), so the \( \mathbf{v} \cdot \nabla \mathbf{v} \) term CANNOT be written as \( \mathbf{v} \times \text{Grad}() \).

### 5.2.3 Input options

Note that in the above equations the extra parameter \( g \) has been used for the ratio of specific heats. To enable this to be set in the input options file (see BOUT++ options), we use the Options object in the initialisation function:

```cpp
class MHD : public PhysicsModel {
    private:
        BoutReal gamma;

    int init(bool restarting) override {
        auto globalOptions = Options::root();
        auto options = globalOptions["mhd"];

        OPTION(options, g, 5.0 / 3.0);
    ...

This specifies that an option called “g” in a section called “mhd” should be put into the variable \( g \). If the option could not be found, or was of the wrong type, the variable should be set to a default value of \( 5/3 \). The value used will be printed to the output file, so if \( g \) is not set in the input file the following line will appear:

Option mhd: \( g = 1.66667 \) (default)

This function can be used to get integers and booleans. To get strings, there is the function (char* options. getString(section, name). To separate options specific to the physics model, these options should be put in a separate section, for example here the “mhd” section has been specified.

Most of the time, the name of the variable (e.g. \( g \)) will be the same as the identifier in the options file (“g”). In this case, there is the macro:

```cpp
OPTION(options, g, 5.0/3.0);
```

which is equivalent to:

```cpp
g = options["g"].withDefault( 5.0/3.0 );
```

See BOUT++ options for more details of how to use the input options.

### 5.2.4 Communication

If you plan to run BOUT++ on more than one processor, any operations involving derivatives will require knowledge of data stored on other processors. To handle the necessary parallel communication, there is the `mesh->communicate` function. This takes care of where the data needs to go to/from, and only needs to be told which variables to transfer.

If you only need to communicate a small number (up to 5 currently) of variables then just call the `Mesh::communicate()` function directly. For the MHD code, we need to communicate the variables \( \rho, p, \mathbf{v}, B \) at the beginning of the `PhysicsModel::rhs()` function before any derivatives are calculated:

```cpp
int rhs(BoutReal t) override {
    mesh->communicate(rho, p, v, B);
}
If you need to communicate lots of variables, or want to change at run-time which variables are evolved (e.g. depending on input options), then you can create a group of variables and communicate them later. To do this, first create a `FieldGroup` object, in this case called `comms`, then use the add method. This method does no communication, but records which variables to transfer when the communication is done later:

```cpp
class MHD : public PhysicsModel {
    private:
    FieldGroup comms;

    int init(bool restarting) override {
        ... 
        comms.add(rho);
        comms.add(p);
        comms.add(v);
        comms.add(B);
        ...
    }
```

The `comms.add()` routine can be given any number of variables at once (there’s no practical limit on the total number of variables which are added to a `FieldGroup`), so this can be shortened to:

```cpp
comms.add(rho, p, v, B);
```

To perform the actual communication, call the `mesh->communicate` function with the group. In this case we need to communicate all these variables before performing any calculations, so call this function at the start of the `rhs` routine:

```cpp
int rhs(BoutReal t) override {
    mesh->communicate(comms);
    ...
}
```

In many situations there may be several groups of variables which can be communicated at different times. The function `mesh->communicate` consists of a call to `Mesh::send()` followed by `Mesh::wait()` which can be done separately to interleave calculations and communications. This will speed up the code if parallel communication bandwidth is a problem for your simulation.

In our MHD example, the calculation of `ddt(rho)` and `ddt(p)` does not require B, so we could first communicate `rho`, `p`, and `v`, send `B` and do some calculations whilst communications are performed:

```cpp
int rhs(BoutReal t) override {
    mesh->communicate(rho, p, v); // sends and receives rho, p and v
    comm_handle ch = mesh->send(B); // only send B

    ddt(rho) = ...
    ddt(p) = ...

    mesh->wait(ch); // now wait for B to arrive

    ddt(v) = ...
    ddt(B) = ...

    return 0;
}
```

This scheme is not used in `mhd.cxx`, partly for clarity, and partly because currently communications are not a significant bottleneck (too much inefficiency elsewhere!).

When a differential is calculated, points on neighbouring cells are assumed to be in the guard cells. There is no way to
calculate the result of the differential in the guard cells, and so after every differential operator the values in the guard
cells are invalid. Therefore, if you take the output of one differential operator and use it as input to another differential
operator, you must perform communications (and set boundary conditions) first. See Differential operators.

5.2.5 Error handling

Finding where bugs have occurred in a (fairly large) parallel code is a difficult problem. This is more of a concern for
developers of BOUT++ (see the developers manual), but it is still useful for the user to be able to hunt down bug in
their own code, or help narrow down where a bug could be occurring.

If you have a bug which is easily reproducible i.e. it occurs almost immediately every time you run the code, then
the easiest way to hunt down the bug is to insert lots of output.write statements (see Logging output). Things get
harder when a bug only occurs after a long time of running, and/or only occasionally. For this type of problem, a useful
tool can be the message stack. An easy way to use this message stack is to use the TRACE macro:

```{ TRACE("Some message here"); // message pushed }
```

This will push the message, then pop the message when the current scope ends (except when an exception occurs).
The error message will also have the file name and line number appended, to help find where an error occurred. The
run-time overhead of this should be small, but can be removed entirely if the compile-time flag -DCHECK is not defined
or set to 0. This turns off checking, and TRACE becomes an empty macro. It is possible to use standard printf like
formatting with the trace macro, for example:

```{ TRACE("The value of i is %d and this is an arbitrary %s", i, "string"); // message push }
```

In the mhd.cxx example each part of the rhs function is trace’d. If an error occurs then at least the equation where it
happened will be printed:

```{ TRACE("ddt(rho)"); 
    ddt(rho) = -V.dot_Grad(v, rho) - rho*Div(v); }
```

5.2.6 Boundary conditions

All evolving variables have boundary conditions applied automatically before the rhs function is called (or afterwards if
the boundaries are being evolved in time). Which condition is applied depends on the options file settings (see Boundary
conditions). If you want to disable this and apply your own boundary conditions then set boundary condition to none
in the BOUT.inp options file.

In addition to evolving variables, it’s sometimes necessary to impose boundary conditions on other quantities which
are not explicitly evolved.

The simplest way to set a boundary condition is to specify it as text, so to apply a Dirichlet boundary condition:

```Field3D var; 
... 
var.applyBoundary("dirichlet");
```
The format is exactly the same as in the options file. Each time this is called it must parse the text, create and destroy boundary objects. To avoid this overhead and have different boundary conditions for each region, it’s better to set the boundary conditions you want to use first in `init`, then just apply them every time:

```cpp
class MHD : public PhysicsModel {
  Field3D var;

  int init(bool restarting) override {
    ...
    var.setBoundary("myVar");
    ...
  }

  int rhs(BoutReal t) override {
    ...
    var.applyBoundary();
    ...
  }
}
```

This will look in the options file for a section called `[myvar]` (upper or lower case doesn’t matter) in the same way that evolving variables are handled. In fact this is precisely what is done: inside `PhysicsModel::bout_solve()` (or `SOLVE_FOR`) the `Field3D::setBoundary` method is called, and then after `rhs` the `Field3D::applyBoundary()` method is called on each evolving variable. This method also gives you the flexibility to apply different boundary conditions on different boundary regions (e.g. radial boundaries and target plates); the first method just applies the same boundary condition to all boundaries.

Another way to set the boundaries is to copy them from another variable:

```cpp
Field3D a, b;
...
a.setBoundaryTo(b); // Copy b's boundaries into a
...
```

Note that this will copy the value at the boundary, which is half-way between mesh points. This is not the same as copying the guard cells from field `b` to field `a`. The value at the boundary cell is calculated using second-order central difference. For example if there is one boundary cell, so that `a(0,y,z)` is the boundary cell, and `a(1,y,z)` is in the domain, then the boundary would be set so that:

\[
a(0,y,z) + a(1,y,z) = b(0,y,z) + b(1,y,z)
\]

rearranged as:

\[
a(0,y,z) = - a(1,y,z) + b(0,y,z) + b(1,y,z)
\]

To copy the boundary cells (and communication guard cells), iterate over them:

```cpp
BOUT_FOR(i, a.getRegion("RGN_GUARDS")) {
  a[i] = b[i];
}
```

See *Iterating over fields* for more details on iterating over custom regions.

---

5.2. Magnetohydrodynamics (MHD)
5.2.7 Custom boundary conditions

The boundary conditions supplied with the BOUT++ library cover the most common situations, but cannot cover all of them. If the boundary condition you need isn’t available, then it’s quite straightforward to write your own. First you need to make sure that your boundary condition isn’t going to be overwritten. To do this, set the boundary condition to “none” in the BOUT:inp options file, and BOUT++ will leave that boundary alone. For example:

```plaintext
[P]
bndry_all = dirichlet
bndry_xin = none
bndry_xout = none
```

would set all boundaries for the variable “P” to zero value, except for the X inner and outer boundaries which will be left alone for you to modify.

To set an X boundary condition, it’s necessary to test if the processor is at the left boundary (first in X), or right boundary (last in X). Note that it might be both if `NXPE = 1`, or neither if `NXPE > 2`.

```plaintext
Field3D f;
...
if(mesh->firstX()) {
    // At the left of the X domain
    // set f[0:1][*][*] i.e. first two points in X, all Y and all Z
    for(int x=0; x < 2; x++)
        for(int y=0; y < mesh->LocalNy; y++)
            for(int z=0; z < mesh->LocalNz; z++) {
                f(x,y,z) = ...
            }
}
if(mesh->lastX()) {
    // At the right of the X domain
    // Set last two points in X
    for(int x=mesh->LocalNx-2; x < mesh->LocalNx; x++)
        for(int y=0; y < mesh->LocalNy; y++)
            for(int z=0; z < mesh->LocalNz; z++) {
                f(x,y,z) = ...
            }
}
```

note the size of the local mesh including guard cells is given by `Mesh::LocalNx`, `Mesh::LocalNy`, and `Mesh::LocalNz`. The functions `Mesh::firstX()` and `Mesh::lastX()` return true only if the current processor is on the left or right of the X domain respectively.

Setting custom Y boundaries is slightly more complicated than X boundaries, because target or limiter plates could cover only part of the domain. Rather than use a for loop to iterate over the points in the boundary, we need to use a more general iterator:

```plaintext
Field3D f;
...
RangeIterator it = mesh->iterateBndryLowerY();
for(it.first(); !it.isDone(); it++) {
    // it.ind contains the x index
    for(int y=-2; y>=0; y--) // Boundary width 3 points
        for(int z=0; z < mesh->LocalNz; z++) {
            ddt(f)(it.ind, y, z) = 0.; // Set time-derivative to zero in boundary
        }
}
```

(continues on next page)
This would set the time-derivative of \( f \) to zero in a boundary of width 3 in \( Y \) (from 0 to 2 inclusive). In the same way `mesh->iterateBndryUpperY()` can be used to iterate over the upper boundary:

```c++
RangeIterator it = mesh->iterateBndryUpperY();
for(it.first(); !it.isDone(); it++) {
    // it.ind contains the x index
    for(int y=mesh->LocalNy-3;y<mesh->LocalNy;y--) // Boundary width 3 points
        for(int z=0;z<mesh->LocalNz;z++) {
            ddt(f)(it.ind,y,z) = 0.; // Set time-derivative to zero in boundary
        }
}
```

### 5.2.8 Initial profiles

Up to this point the code is evolving total density, pressure etc. This has advantages for clarity, but has problems numerically: For small perturbations, rounding error and tolerances in the time-integration mean that linear dispersion relations are not calculated correctly. The solution to this is to write all equations in terms of an initial “background” quantity and a time-evolving perturbation, for example \( \rho(t) \rightarrow \rho_0 + \tilde{\rho}(t) \). For this reason, the initialisation of all variables passed to the `PhysicsModel::bout_solve` function is a combination of small-amplitude gaussians and waves; the user is expected to have performed this separation into background and perturbed quantities.

To read in a quantity from a grid file, there is the `mesh->get` function:

```c++
Field2D Ni0; // Background density
int init(bool restarting) override {
    ...
    mesh->get(Ni0, "Ni0");
    ...
}
```

As with the input options, most of the time the name of the variable in the physics code will be the same as the name in the grid file to avoid confusion. In this case, you can just use:

```c++
GRID_LOAD(Ni0);
```

which is equivalent to:

```c++
mesh->get(Ni0, "Ni0");
```

(see `Mesh::get()`).
5.2.9 Output variables

BOUT++ always writes the evolving variables to file, but often it’s useful to add other variables to the output. For convenience you might want to write the normalised starting profiles or other non-evolving values to file. For example:

```plaintext
Field2D Ni0;
...
GRID_LOAD(Ni0);
dump.add(Ni0, "Ni0", false);
```

where the ‘false’ at the end means the variable should only be written to file once at the start of the simulation. For convenience there are some macros e.g.:

```plaintext
SAVE_ONCE(Ni0);
```

is equivalent to:

```plaintext
dump.add(Ni0, "Ni0", false);
```

Optionally, you can add a description to document what the variable represents, which will be saved as an attribute of the variable in the output file, e.g.:

```plaintext
dump.add(Ni0, "Ni0", false, "background density profile");
```

(see `Datafile::add()`). In some situations you might also want to write some data to a different file. To do this, create a `Datafile` object:

```plaintext
Datafile mydata;
```

in `init`, you then:

1. (optional) Initialise the file, passing it the options to use. If you skip this step, default (sane) options will be used. This just allows you to enable/disable, use parallel I/O, set whether files are opened and closed every time etc.:

   ```plaintext
   mydata = Datafile(Options::getRoot()->getSection("mydata"));
   ```

   which would use options in a section `[mydata]` in BOUT.inp

2. Open the file for writing:

   ```plaintext
   mydata.openw("mydata.nc")
   ```

   (see `Datafile::openw()`). By default this only specifies the file name; actual opening of the file happens later when the data is written. If you are not using parallel I/O, the processor number is also inserted into the file name before the last ‘.’, so mydata.nc becomes “mydata.0.nc”, “mydata.1.nc” etc. The file format used depends on the extension, so “.nc” will open NetCDF, and “.hdf5” or “.h5” an HDF5 file.

   (see e.g. src/fileio/datafile.cxx line 139, which calls src/fileio/dataformat.cxx line 23, which then calls the file format interface e.g. src/fileio/impls/netcdf/nc_format.cxx line 172).

3. Add variables to the file

   ```plaintext
   // Not evolving. Every time the file is written, this will be overwritten
   mydata.add(variable, "name");
   // Evolving. Will output a sequence of values
   mydata.add(variable2, "name2", true);
   ```

Whenever you want to write values to the file, for example in `rhs` or a monitor, just call:
mydata.write();

(see Datafile::write()). To collect the data afterwards, you can specify the prefix to collect. In Python (see collect()):

```python
>>> var = collect("name", prefix="mydata")
```

By default the prefix is “BOUT.dmp”.

### 5.2.10 Variable attributes

An experimental feature is the ability to add attributes to output variables. Do this using with Datafile::setAttribute():

```
dump.setAttribute(variable, attribute, value);
```

where `variable` is the name of the variable; `attribute` is the name of the attribute, and `value` can be either a string or an integer. For example:

```
dump.setAttribute("Ni\theta", "units", "m^{-3}");
```

### 5.3 Reduced MHD

The MHD example presented previously covered some of the functions available in BOUT++, which can be used for a wide variety of models. There are however several other significant functions and classes which are commonly used, which will be illustrated using the reconnect-2field example. This is solving equations for $\mathbf{A}$ and vorticity $\mathbf{U}$

$$
\frac{\partial \mathbf{U}}{\partial t} = -\frac{1}{B} \mathbf{b}_0 \times \nabla \phi \cdot \nabla \mathbf{U} + B^2 \nabla ||(j_||/B)
$$

$$
\frac{\partial A_||}{\partial t} = -\frac{1}{\beta} \nabla ||(\phi - \eta \frac{1}{\beta} j_||)
$$

with $\phi$ and $j_||$ given by

$$
U = \frac{1}{B} \nabla^2 \phi
$$

$$
\dot{j}_|| = -\nabla^2 A_||
$$

First create the variables which are going to be evolved, ensure they’re communicated:

```
class TwoField : public PhysicsModel {
    private:
    Field3D U, Apar; // Evolving variables

    int init(bool restarting) override {
        SOLVE_FOR(U, Apar);
    }

    int rhs(BoutReal t) override {
        mesh->communicate(U, Apar);
    }
};
```
In order to calculate the time derivatives, we need the auxiliary variables $\phi$ and $j_{||}$. Calculating $j_{||}$ from $A_{||}$ is a straightforward differential operation, but getting $\phi$ from $U$ means inverting a Laplacian.

```cpp
Field3D U, Apar;
Field3D phi, jpar; // Auxilliary variables

int init(bool restarting) override {
    SOLVE_FOR(U, Apar);
    SAVE_REPEAT(phi, jpar); // Save variables in output file
    return 0;
}

int rhs(BoutReal t) override {
    phi = invert_laplace(mesh->Bxy*U, phi_flags); // Solve for phi
    mesh->communicate(U, Apar, phi); // Communicate phi
    jpar = -Delp2(Apar); // Calculate jpar
    mesh->communicate(jpar); // Communicate jpar
    return 0;
}
```

Note that the Laplacian inversion code takes care of boundary regions, so $U$ doesn’t need to be communicated first. The differential operator $Delp2$, like all differential operators, needs the values in the guard cells and so $A_{||}$ needs to be communicated before calculating $j_{||}$. Since we will need to take derivatives of $j_{||}$ later, this needs to be communicated as well.

```cpp
int rhs(BoutReal t) override {
    ... mesh->communicate(jpar);

    ddt(U) = -b0xGrad_dot_Grad(phi, U) + SQ(mesh->Bxy)*Grad_par(Jpar / mesh->Bxy)
    ddt(Apar) = -Grad_par(phi) / beta_hat - eta*jpar / beta_hat;
}
```

### 5.4 Logging output

Logging should be used to report simulation progress, record information, and warn about potential problems. BOUT++ includes a simple logging facility which supports both C printf and C++ iostream styles. For example:

```cpp
output.write("This is an integer: %d, and this a real: %e\n", 5, 2.0)
output << "This is an integer: " << 5 << ", and this a real: " << 2.0 << endl;
```

Messages sent to output on processor 0 will be printed to console and saved to `BOUT.log.0`. Messages from all other processors will only go to their log files, `BOUT.log.#` where # is the processor number.

**Note:** If an error occurs on a processor other than processor 0, then the error message will usually only be in the log file, not printed to console. If BOUT++ crashes but no error message is printed, try looking at the ends of all log files:

```bash
$ tail BOUT.log.*
```

For finer control over which messages are printed, several outputs are available, listed in the table below.
### 5.4.1 Controlling logging level

By default all of the outputs except `output_debug` are saved to log and printed to console (processor 0 only).

To reduce the volume of outputs the command line argument `-q` (quiet) reduces the output level by one, and `-v` (verbose) increases it by one. Running with `-q` in the command line arguments suppresses the `output_info` messages, so that they will not appear in the console or log file. Running with `-q -q` suppresses everything except `output_warn` and `output_error`.

To enable the `output_debug` messages, configure BOUT++ with a `CHECK` level $\geq$ 3. To enable it at lower check levels, configure BOUT++ with `--enable-debug-output` (for `./configure`) or `-DBOUT_ENABLE_OUTPUT_DEBUG` (for CMake). When running BOUT++ add a `-v -v` flag to see `output_debug` messages.

### 5.5 Updating Physics Models from v3 to v4

Version 4.0.0 of BOUT++ introduced several features which break backwards compatibility. If you already have physics models, you will most likely need to update them to work with version 4. The main breaking changes which you are likely to come across are:

- Using round brackets `()` instead of square brackets `[]` for indexing fields
- Moving components of `Mesh` related to the metric tensor and “real space” out into a new object, `Coordinates`
- Changed some `Field3D` member functions into non-member functions
- The shifted metric method has changed in version 4, so that fields are stored in orthogonal X-Z coordinates rather than field aligned coordinates. This has implications for boundary conditions and post-processing. See `Parallel Transforms` for more information.

A new tool is provided, `bin/bout_3to4.py`, which can identify these changes, and fix most of them automatically. Simply run this program on your physic model to see how to update it to work with version 4:

```
$ ${BOUT_TOP}/bin/bout_3to4.py my_model.cxx
```

The output of this command will show you how to fix each problem it identifies. To automatically apply them, you can use the `--replace` option:

```
$ ${BOUT_TOP}/bin/bout_3to4.py --replace my_model.cxx
```

Also in version 4 is a new syntax for looping over each point in a field. See `Iterating over fields` for more information.
5.6 More examples

The code and input files in the examples/ subdirectory are for research, demonstrating BOUT++, and to check for broken functionality. Some proper unit tests have been implemented, but this is something which needs improving. The examples which were published in [Dudson2009] were drift-instability, interchange-instability and orszag-tang.

5.6.1 advect1d

The model in gas_compress.cxx solves the compressible gas dynamics equations for the density $n$, velocity $V$, and pressure $P$:

5.6.2 drift-instability

The physics code 2fluid.cxx implements a set of reduced Braginskii 2-fluid equations, similar to those solved by the original BOUT code. This evolves 6 variables: Density, electron and ion temperatures, parallel ion velocity, parallel current density and vorticity.

Input grid files are the same as the original BOUT code, but the output format is different.

5.6.3 interchange-instability

5.6.4 sod-shock
Fig. 5.1: Interchange instability test. Solid lines are from analytic theory, symbols from BOUT++ simulations, and the RMS density is averaged over $z$. Vertical dashed line marks the reference point, where analytic and simulation results are set equal.

Fig. 5.2: Sod shock-tube problem for testing shock-handling methods.
MAKFILES AND COMPILING BOUT++

BOUT++ has its own makefile system. These can be used to

1. Write an example or executable
2. Add a feature to BOUT++

In all makefiles, BOUT_TOP is required!

These makefiles are sufficient for most uses, but for more complicated, an executable script `bout-config` can be used to get the compilation flags (see `bout-config` script).

### 6.1 Executables example

If writing an example (or physics module that executes) then the makefile is very simple:

```
BOUT_TOP = ../..

SOURCEC = <filename>.cxx

include $(BOUT_TOP)/make.config
```

where BOUT_TOP refers to the relative (or absolute) location of the BOUT directory (the one that includes `/lib` and `/src`) and SOURCEC is the name of your file, e.g. `gas_compress.cxx`.

Optionally, it is possible to specify TARGET which defines what the executable should be called (e.g. if you have multiple source files). That’s it!

#### 6.1.1 Multiple subdirectories

Large physics modules can have many files, and it can be helpful to organise these into subdirectories. An example of how to do this is in `examples/make_subdir`.

In the top level, list the directories

```
DIRS = fuu bar
```

In the makefile in each subdirectory, specify

```
TARGET = sub
```

then specify the path to the top-level directory
6.2 Modules example

If you are writing a new module (or concrete implementation) to go into the BOUT++ library, then it is again pretty simple.

```plaintext
BOUT_TOP = ../..
SOURCEC = communicator.cxx difops.cxx geometry.cxx grid.cxx \
           interpolation.cxx topology.cxx
SOURCEH = $(SOURCEC:%.cxx=%.h)
TARGET = lib

include $(BOUT_TOP)/make.config
```

TARGET - must be lib to signify you are adding to libbout++.a.

The other variables should be pretty self explanatory.

6.2.1 Adding a new subdirectory to 'src'

No worries, just make sure to edit src/makefile to add it to the DIRS variable.

6.3 bout-config script

The bout-config script is in the bin subdirectory of the BOUT++ distribution, and is generated by configure. This script can be used to get the compilers, flags and settings to compile BOUT++. To get a list of available options:

```plaintext
$ bout-config --help
```

so to get the library linking flags, for example

```plaintext
$ bout-config --libs
```

This script can be used in makefiles to compile BOUT++ alongside other libraries. The easiest way is to use bout-config to find the make.config file which contains the settings. For example the heat conduction example can be compiled with the following makefile:

```plaintext
SOURCEC = conduction.cxx
include $(shell bout-config --config-file)
```

This includes the make.config file installed with bout-config, rather than using the BOUT_TOP variable.

A different way to use bout-config is to get the compiler and linker flags, and use them in your own makefile, for example:
CXX=`bout-config --cxx`
CFLAGS=`bout-config --cflags`
LD=`bout-config --ld`
LDFLAGS=`bout-config --libs`

conduction: conduction.cxx
  $(CXX) $(CFLAGS) -c conduction.cxx -o conduction.o
  $(LD) -o conduction conduction.o $(LDFLAGS)

A more general example is in examples/make-script.
Variables in BOUT++ are not initialised automatically, but must be explicitly given a value. For example the following code declares a Field3D variable then attempts to access a particular element:

```cpp
Field3D f; // Declare a variable
f(0,0,0) = 1.0; // Error!
```

This results in an error because the data array to store values in \( \mathbf{f} \) has not been allocated. Allocating data can be done in several ways:

1. Initialise with a value:

   ```cpp
   Field3D f = 0.0; // Allocates memory, fills with zeros
   f(0,0,0) = 1.0; // ok
   ```

   This cannot be done at a global scope, since it requires the mesh to already exist and have a defined size.

2. Set to a scalar value:

   ```cpp
   Field3D f;
   f = 0.0; // Allocates memory, fills with zeros
   f(0,0,0) = 1.0; // ok
   ```

   Note that setting a field equal to another field has the effect of making both fields share the same underlying data. This behaviour is similar to how NumPy arrays behave in Python.

   ```cpp
   Field3D g = 0.0; // Allocates memory, fills with zeros
   Field3D f = g; // f now shares memory with g
   f(0,0,0) = 1.0; // g also modified
   ```

   To ensure that a field has a unique underlying memory array call the `Field3D::allocate()` method before writing to individual indices.

3. Use `Field3D::allocate()` to allocate memory:

   ```cpp
   Field3D f;
   f.allocate(); // Allocates memory, values undefined
   f(0,0,0) = 1.0; // ok
   ```

In a BOUT++ simulation some variables are typically evolved in time. The initialisation of these variables is handled by the time integration solver.
7.1 Initialisation of time evolved variables

Each variable being evolved has its own section, with the same name as the output data. For example, the high-β model has variables “P”, “jpar”, and “U”, and so has sections [P], [jpar], [U] (not case sensitive).

7.1.1 Expressions

The recommended way to initialise a variable is to use the function option for each variable:

```
[p]
function = 1 + gauss(x-0.5)*gauss(y)*sin(z)
```

This evaluates an analytic expression to initialise the \( P \) variable. Expressions can include the usual operators (+, -, *, /), including ^ for exponents. The following values are also already defined:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( x ) position between 0 and 1</td>
</tr>
<tr>
<td>( y )</td>
<td>( y ) position between 0 and ( 2\pi ) (excluding the last point)</td>
</tr>
<tr>
<td>( z )</td>
<td>( z ) position between 0 and ( 2\pi ) (excluding the last point)</td>
</tr>
<tr>
<td>( \pi )</td>
<td>3.1415...</td>
</tr>
</tbody>
</table>

Table: Initialisation expression values

By default, \( x \) is defined as \( i \) / \((nx - 2*MXG)\), where MXG is the width of the boundary region, by default 2. Hence \( x \) actually goes from 0 on the leftmost point to \((nx-1)/(nx-4)\) on the rightmost point. This is not a particularly good definition, but for most cases its sufficient to create some initial profiles. For some problems like island reconnection simulations, it’s useful to define \( x \) in a particular way which is more symmetric than the default. To do this, set in BOUT.inp

```
[mesh]
symmetricGlobalX = true
```

This will change the definition of \( x \) to \( i \) / \((nx - 1)\), so \( x \) is then between 0 and 1 everywhere.

By default the expressions are evaluated in a field-aligned coordinate system, i.e. if you are using the [mesh] option \( \text{paralleltransform} = \text{shifted} \), the input \( f \) will have \( f = \text{fromFieldAligned}(f) \) applied before being returned. To switch off this behaviour and evaluate the input expressions in coordinates with orthogonal \( x-z \) (i.e. toroidal \( \{\psi, \theta, \phi\} \) coordinates when using \( \text{paralleltransform} = \text{shifted} \)), set in BOUT.inp

```
[input]
transform_from_field_aligned = false
```

The functions in Table 7.1 are also available in expressions.
Table 7.1: Initialisation expression functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(x)</td>
<td>Absolute value $</td>
</tr>
<tr>
<td>asin(x), acos(x), atan(x), atan(y,x)</td>
<td>Inverse trigonometric functions</td>
</tr>
<tr>
<td>ballooning(x)</td>
<td>Ballooning transform ((7.1), Fig. 7.1)</td>
</tr>
<tr>
<td>ballooning(x,n)</td>
<td>Ballooning transform, using $n$ terms (default 3)</td>
</tr>
<tr>
<td>cos(x)</td>
<td>Cosine</td>
</tr>
<tr>
<td>cosh(x)</td>
<td>Hyperbolic cosine</td>
</tr>
<tr>
<td>exp(x)</td>
<td>Exponential</td>
</tr>
<tr>
<td>tanh(x)</td>
<td>Hyperbolic tangent</td>
</tr>
<tr>
<td>gauss(x)</td>
<td>Gaussian $\exp(-x^2/2)/\sqrt{2\pi}$</td>
</tr>
<tr>
<td>gauss(x, w)</td>
<td>Gaussian $\exp(-x^2/(2w^2))/w/\sqrt{2\pi}$</td>
</tr>
<tr>
<td>H(x)</td>
<td>Heaviside function: 1 if $x &gt; 0$ otherwise 0</td>
</tr>
<tr>
<td>log(x)</td>
<td>Natural logarithm</td>
</tr>
<tr>
<td>max(x,y,...)</td>
<td>Maximum (variable arguments)</td>
</tr>
<tr>
<td>min(x,y,...)</td>
<td>Minimum (variable arguments)</td>
</tr>
<tr>
<td>mixmode(x)</td>
<td>A mixture of Fourier modes</td>
</tr>
<tr>
<td>mixmode(x, seed)</td>
<td>seed determines random phase (default 0.5)</td>
</tr>
<tr>
<td>power(x,y)</td>
<td>Exponent $x^y$</td>
</tr>
<tr>
<td>sin(x)</td>
<td>Sine</td>
</tr>
<tr>
<td>sinh(x)</td>
<td>Hyperbolic sine</td>
</tr>
<tr>
<td>sqrt(x)</td>
<td>$\sqrt{x}$</td>
</tr>
<tr>
<td>tan(x)</td>
<td>Tangent</td>
</tr>
<tr>
<td>erf(x)</td>
<td>The error function</td>
</tr>
<tr>
<td>TanhHat(x, width, centre, steepness)</td>
<td>The hat function $\frac{1}{2}(\tanh[s(x-c-w/2)] - \tanh[s(x+c+w/2)])$</td>
</tr>
<tr>
<td>fmod(x)</td>
<td>The modulo operator, returns floating point remainder</td>
</tr>
</tbody>
</table>

For field-aligned tokamak simulations, the Y direction is along the field and in the core this will have a discontinuity at the twist-shift location where field-lines are matched onto each other. To handle this, the ballooning function applies a truncated Ballooning transformation to construct a smooth initial perturbation:

$$U_0^{\text{balloon}} = \sum_{i=-N}^{N} F(x)G(y+2\pi i)H(z+q2\pi i)$$  \hspace{1cm} (7.1)

There is an example code test-ballooning which compares methods of setting initial conditions with the ballooning transform.

The mixmode(x) function is a mixture of Fourier modes of the form:

$$\text{mixmode}(x) = \sum_{i=1}^{14} \frac{1}{(1+|i-4|)^2} \cos[ix + \phi(i, \text{seed})]$$

where $\phi$ is a random phase between $-\pi$ and $+\pi$, which depends on the seed. The factor in front of each term is chosen so that the 4th harmonic ($i = 4$) has the highest amplitude. This is useful mainly for initialising turbulence simulations, where a mixture of mode numbers is desired.
7.2 Initialising variables with the FieldFactory class

This class provides a way to generate a field with a specified form. For example to create a variable var from options we could write

```
FieldFactory f(mesh);
Field2D var = f.create2D("var");
```

This will look for an option called “var”, and use that expression to initialise the variable var. This could then be set in the BOUT.inp file or on the command line.

```
var = gauss(x-0.5,0.2)*gauss(y)*sin(3*z)
```

To do this, FieldFactory implements a recursive descent parser to turn a string containing something like "gauss(x-0.5,0.2)*gauss(y)*sin(3*z)" into values in a Field3D or Field2D object. Examples are given in the test-fieldfactory example:

```
FieldFactory f(mesh);
Field2D b = f.create2D("1 - x");
Field3D d = f.create3D("gauss(x-0.5,0.2)*gauss(y)*sin(z)");
```

This is done by creating a tree of FieldGenerator objects which then generate the field values:

```
class FieldGenerator {
    public:
        virtual ~FieldGenerator() {}
        virtual FieldGenerator* clone(const list<FieldGenerator*>& args) {return NULL;}
        virtual BoutReal generate(int x, int y, int z) = 0;
};
```

All classes inheriting from FieldGenerator must implement a FieldGenerator::generate() function, which returns the value at the given (x,y,z) position. Classes should also implement a FieldGenerator::clone() function, which takes a list of arguments and creates a new instance of its class. This takes as input a list of other FieldGenerator objects, allowing a variable number of arguments.

The simplest generator is a fixed numerical value, which is represented by a FieldValue object:
To add a new function to the FieldFactory, a new `FieldGenerator` class must be defined. Here we will use the example of the `sinh` function, implemented using a class `FieldSinh`. This takes a single argument as input, but `FieldPI` takes no arguments, and `FieldGaussian` takes either one or two. Study these after reading this to see how these are handled.

First, edit `src/field/fieldgenerators.hxx` and add a class definition:

```cpp
class FieldSinh : public FieldGenerator {
public:
    FieldSinh(FieldGenerator* g) : gen(g) {}
    ~FieldSinh() {if(gen) delete gen;}
    FieldGenerator* clone(const list<FieldGenerator*> args);
    BoutReal generate(int x, int y, int z);
private:
    FieldGenerator *gen;
};
```

The `gen` member is used to store the input argument, and to make sure it's deleted properly we add some code to the destructor. The constructor takes a single input, the `FieldGenerator` argument to the `sinh` function, which is stored in the member `gen`.

Next edit `src/field/fieldgenerators.hxx` and add the implementation of the `clone` and `generate` functions:

```cpp
FieldGenerator* FieldSinh::clone(const list<FieldGenerator*> args) {
    if(args.size() != 1) {
        throw ParseException("Incorrect number of arguments to sinh function. Expecting 1, got %d", args.size());
    }
    return new FieldSinh(args.front());
}
BoutReal FieldSinh::generate(double x, double y, double z, double t) {
    return sinh(gen->generate(x,y,z,t));
}
```

The `clone` function first checks the number of arguments using `args.size()`. This is used in `FieldGaussian` to handle different numbers of input, but in this case we throw a `ParseException` if the number of inputs isn’t one. `clone` then creates a new `FieldSinh` object, passing the first argument (`args.front()`) to the constructor (which then gets stored in the `gen` member variable).

The `generate` function for `sinh` just gets the value of the input by calling `gen->generate(x,y,z)`, calculates `sinh` of it and returns the result.
The clone function means that the parsing code can make copies of any FieldGenerator class if it's given a single instance to start with. The final step is therefore to give the FieldFactory class an instance of this new generator. Edit the FieldFactory constructor FieldFactory::FieldFactory() in src/field/field_factory.cxx and add the line:

```cpp
addGenerator("sinh", new FieldSinh(NULL));
```

That's it! This line associates the string "sinh" with a FieldGenerator. Even though FieldFactory doesn't know what type of FieldGenerator it is, it can make more copies by calling the clone member function. This is a useful technique for polymorphic objects in C++ called the “Virtual Constructor” idiom.

### 7.4 Parser internals

When a FieldGenerator is added using the addGenerator function, it is entered into a std::map which maps strings to FieldGenerator objects (include/field_factory.hxx):

```cpp
map<string, FieldGenerator*> gen;
```

Parsing a string into a tree of FieldGenerator objects is done by first splitting the string up into separate tokens like operators like '*', brackets '(', names like 'sinh' and so on, then recognising patterns in the stream of tokens. Recognising tokens is done in src/field/field_factory.cxx:

```cpp
char FieldFactory::nextToken() {
  ...
```

This returns the next token, and setting the variable char curtok to the same value. This can be one of:

- -1 if the next token is a number. The variable BoutReal curval is set to the value of the token
- -2 for a string (e.g. “sinh”, “x” or “pi”). This includes anything which starts with a letter, and contains only letters, numbers, and underscores. The string is stored in the variable string curident.
- 0 to mean end of input
- The character if none of the above. Since letters and numbers are taken care of (see above), this includes brackets and operators like '+-' and '-'.

The parsing stage turns these tokens into a tree of FieldGenerator objects, starting with the parse() function:

```cpp
FieldGenerator* FieldFactory::parse(const string &input) {
  ...
```

which puts the input string into a stream so that nextToken() can use it, then calls the parseExpression() function to do the actual parsing:

```cpp
FieldGenerator* FieldFactory::parseExpression() {
  ...
```

This breaks down expressions in stages, starting with writing every expression as:

```cpp
expression := primary [ op primary ]
```

i.e. a primary expression, and optionally an operator and another primary expression. Primary expressions are handled by the parsePrimary() function, so first parsePrimary() is called, and then parseBinOpRHS which checks if there is an operator, and if so calls parsePrimary() to parse it. This code also takes care of operator precedence by keeping
track of the precedence of the current operator. Primary expressions are then further broken down and can consist of either a number, a name (identifier), a minus sign and a primary expression, or brackets around an expression:

```
primary := number 
  := identifier 
  := '-' primary 
  := '(' expression ')' 
  := '[' expression ']'
```

The minus sign case is needed to handle the unary minus e.g. "-x". Identifiers are handled in `parseIdentifierExpr()` which handles either variable names, or functions

```
identifier := name 
  := name '(' expression [ ',' expression [ ',' ... ] ] ')' 
```

i.e. a name, optionally followed by brackets containing one or more expressions separated by commas. names without brackets are treated the same as those with empty brackets, so "x" is the same as "x()". A list of inputs `list<FieldGenerator*> args;` is created, the gen map is searched to find the `FieldGenerator` object corresponding to the name, and the list of inputs is passed to the object’s `clone` function.
BOUNDARY CONDITIONS

Like the variable initialisation, boundary conditions can be set for each variable in individual sections, with default values in a section [All]. Boundary conditions are specified for each variable, being applied to variable itself during initialisation, and the time-derivatives at each timestep. They are a combination of a basic boundary condition, and optional modifiers.

When finding the boundary condition for a variable \texttt{var} on a boundary region, the options are checked in order from most to least specific:

- 
  Section \texttt{var}, \texttt{bndry_} + region name. Depending on the mesh file, regions of the grid are given labels. Currently these are \texttt{core}, \texttt{sol}, \texttt{pf} and \texttt{target} which are intended for tokamak edge simulations. Hence the variables checked are \texttt{bndry_core}, \texttt{bndry_pf} etc.

- Section \texttt{var}, \texttt{bndry_} + boundary side. These names are \texttt{xin}, \texttt{xout}, \texttt{yup} and \texttt{ydown}.

- Section \texttt{var}, variable \texttt{bndry_all}
  - The same settings again except in section \texttt{All}.

The default setting for everything is therefore \texttt{bndry_all} in the \texttt{All} section.

Boundary conditions are given names, with optional arguments in brackets. Currently implemented boundary conditions are:

- \texttt{dirichlet} - Set to zero
- \texttt{dirichlet(<number>)} - Set to some number e.g. \texttt{dirichlet(1)} sets the boundary to 1.0
- \texttt{neumann} - Zero gradient
- \texttt{robin} - A combination of zero-gradient and zero-value \( af + b \frac{\partial f}{\partial x} = g \) where the syntax is \texttt{robin(a, b, g)}.
- \texttt{constgradient} - Constant gradient across boundary
- \texttt{zerolaplace} - Laplacian = 0, decaying solution (X boundaries only)
- \texttt{zerolaplace2} - Laplacian = 0, using coefficients from the Laplacian inversion and Delp2 operator.
- \texttt{constlaplace} - Laplacian = const, decaying solution (X boundaries only)

The zero- or constant-Laplacian boundary conditions works as follows:

\[
\nabla^2 f = 0 \approx g^{xx} \frac{\partial^2 f}{\partial x^2} + g^{zz} \frac{\partial^2 f}{\partial z^2}
\]

which when Fourier transformed in \( z \) becomes:

\[
g^{xx} \frac{\partial^2 \hat{f}}{\partial x^2} - g^{zz} k_z^2 \hat{f} = 0
\]

75
which has the solution

\[ \dot{f} = Ae^{xk_z\sqrt{g_z/g_x}} + Be^{-xk_z\sqrt{g_z/g_x}} \]

Assuming that the solution should decay away from the domain, on the inner \( x \) boundary \( B = 0 \), and on the outer boundary \( A = 0 \). Boundary modifiers change the behaviour of boundary conditions, and more than one modifier can be used. Currently the following are available:

- **relax** - Relaxing boundaries. Evolve the variable towards the given boundary condition at a given rate
- **shifted** - Apply boundary conditions in orthogonal X-Z coordinates, rather than field-aligned
- **width** - Modifies the width of the region over which the boundary condition is applied

These are described in the following subsections.

### 8.1 Relaxing boundaries

All boundaries can be modified to be “relaxing” which are a combination of zero-gradient time-derivative, and whatever boundary condition they are applied to. The idea is that this prevents sharp discontinuities at boundaries during transients, whilst maintaining the desired boundary condition on longer time-scales. In some cases this can improve the numerical stability and timestep.

For example, `relax(dirichlet)` will make a field \( f \) at point \( i \) in the boundary follow a point \( i - 1 \) in the domain:

\[ \frac{\partial f}{\partial t} \big|_i = \frac{\partial f}{\partial t} \big|_{i-1} - \frac{f_i}{\tau} \]

where \( \tau \) is a time-scale for the boundary (currently set to 0.1, but will be a global option). When the time-derivatives are slow close to the boundary, the boundary relaxes to the desired condition (Dirichlet in this case), but when the time-derivatives are large then the boundary approaches Neumann to reduce discontinuities.

By default, the relaxation rate is set to 10 (i.e. a time-scale of \( \tau = 0.1 \)). To change this, give the rate as the second argument e.g. `relax(dirichlet, 2)` would relax to a Dirichlet boundary condition at a rate of 2.

### 8.2 Shifted boundaries

By default boundary conditions are applied in field-aligned coordinates, where \( y \) is along field-lines but \( x \) has a discontinuity at the twist-shift location. If radial derivatives are being done in shifted coordinates where \( x \) and \( z \) are orthogonal, then boundary conditions should also be applied in shifted coordinates. To do this, the **shifted** boundary modifier applies a \( z \) shift, applies the boundary condition, then shifts back. For example:

```plaintext
bndry_core = shifted( neumann )
```

would ensure that radial derivatives were zero in shifted coordinates on the core boundary.
8.3 Changing the width of boundaries

To change the width of a boundary region, the width modifier changes the width of a boundary region before applying the boundary condition, then changes the width back afterwards. To use, specify the boundary condition and the width, for example

```plaintext
bndry_core = width( neumann , 4 )
```

would apply a Neumann boundary condition on the innermost 4 cells in the core, rather than the usual 2. When combining with other boundary modifiers, this should be applied first e.g.

```plaintext
bndry_sol = width( relax( dirichlet ), 3)
```

would relax the last 3 cells towards zero, whereas

```plaintext
bndry_sol = relax( width( dirichlet, 3 ) )
```

would only apply to the usual 2, since relax didn’t use the updated width.

Limitations:

1. Because it modifies then restores a globally-used BoundaryRegion, this code is not thread safe.
2. Boundary conditions can’t be applied across processors, and no checks are done that the width asked for fits within a single processor.

8.4 Examples

This example is taken from the UEDGE benchmark test (in examples/uedge-benchmark):

```plaintext
[All]
bndry_all = neumann # Default for all variables, boundaries

[Ni]
bndry_target = neumann
bndry_core = relax(dirichlet(1.)) # 1e13 cm^-3 on core boundary
bndry_all = relax(dirichlet(0.1)) # 1e12 cm^-3 on other boundaries

[Vi]
bndry_ydown = relax(dirichlet(-1.41648)) # -3.095e4/Vi_x
bndry_yup = relax(dirichlet( 1.41648))
```

The variable $\text{Ni}$ (density) is set to a Neumann boundary condition on the targets (yup and ydown), relaxes towards 1 on the core boundary, and relaxes to 0.1 on all other boundaries. Note that the `bndry_target = neumann` needs to be in the $\text{Ni}$ section: If we just had

```plaintext
[All]
bndry_all = neumann # Default for all variables, boundaries

[Ni]
bndry_core = relax(dirichlet(1.)) # 1e13 cm^-3 on core boundary
bndry_all = relax(dirichlet(0.1)) # 1e12 cm^-3 on other boundaries
```

8.3. Changing the width of boundaries
then the “target” boundary condition for \( \text{Ni} \) would first search in the [Ni] section for \textit{bndry\_target}, then for \textit{bndry\_all} in the [Ni] section. This is set to \textit{relax(dirichlet(0.1))}, not the Neumann condition desired.

### 8.5 Boundary regions

The boundary condition code needs ways to loop over the boundary regions, without needing to know the details of the mesh.

At the moment two mechanisms are provided: A RangeIterator over upper and lower Y boundaries, and a vector of BoundaryRegion objects.

```cpp
// Boundary region iteration
virtual const RangeIterator iterateBndryLowerY() const = 0;
virtual const RangeIterator iterateBndryUpperY() const = 0;

bool hasBndryLowerY();
bool hasBndryUpperY();

bool BoundaryOnCell; // NB: DOESN'T REALLY BELONG HERE
```

The \textit{RangeIterator} class is an iterator which allows looping over a set of indices. For example, in \texttt{src/solver/solver.cxx} to loop over the upper Y boundary of a 2D variable \textit{var}:

```cpp
for(RangeIterator xi = mesh->iterateBndryUpperY(); !xi.isDone(); xi++) {
    ...
}
```

The \textit{BoundaryRegion} class is defined in \texttt{include/boundary_region.hxx}

### 8.6 Boundary regions

Different regions of the boundary such as “core”, “sol” etc. are labelled by the \textit{Mesh} class (i.e. \textit{BoutMesh}), which implements a member function defined in \texttt{mesh.hxx}:

```cpp
// Boundary regions
virtual vector<BoundaryRegion*> getBoundaries() = 0;
```

This returns a vector of pointers to \textit{BoundaryRegion} objects, each of which describes a boundary region with a label, a BndryLoc location (i.e. inner x, outer x, lower y, upper y or all), and iterator functions for looping over the points. This class is defined in \texttt{boundary_region.hxx}:

```cpp
/// Describes a region of the boundary, and a means of iterating over it
class BoundaryRegion {
    public:
        BoundaryRegion();
        BoundaryRegion(const string &name, int xd, int yd);
        virtual ~BoundaryRegion();

        string label; // Label for this boundary region
        BndryLoc location; // Which side of the domain is it on?

    (continues on next page)
```
int x,y; // Indices of the point in the boundary
int bx, by; // Direction of the boundary [x+dx][y+dy] is going outwards

virtual void first() = 0;
virtual void next() = 0; // Loop over every element from inside out (in X or Y first)
virtual void nextX() = 0; // Just loop over X
virtual void nextY() = 0; // Just loop over Y
virtual bool isDone() = 0; // Returns true if outside domain. Can use this with nested nextX, nextY
};

Example: To loop over all points in BoundaryRegion *bndry, use

for(bndry->first(); !bndry->isDone(); bndry->next()) {
    ...
}

Inside the loop, bndry->x and bndry->y are the indices of the point, whilst bndry->bx and bndry->by are unit vectors out of the domain. The loop is over all the points from the domain outwards i.e. the point [bndry->x - bndry->bx][bndry->y - bndry->by] will always be defined.

Sometimes it’s useful to be able to loop over just one direction along the boundary. To do this, it is possible to use nextX() or nextY() rather than next(). It is also possible to loop over both dimensions using:

for(bndry->first(); !bndry->isDone(); bndry->nextX())
for(; !bndry->isDone(); bndry->nextY()) {
    ...
}

8.7 Boundary operations

On each boundary, conditions must be specified for each variable. The different conditions are imposed by BoundaryOp objects. These set the values in the boundary region such that they obey e.g. Dirichlet or Neumann conditions. The BoundaryOp class is defined in boundary_op.hxx:

/// An operation on a boundary
class BoundaryOp {
    public:
        BoundaryOp() {bndry = NULL;}
        BoundaryOp(BoundaryRegion *region)

        // Note: All methods must implement clone, except for modifiers (see below)
        virtual BoundaryOp* clone(BoundaryRegion *region, const list<string> &args);

        // Apply a boundary condition on field f
        virtual void apply(Field2D &f) = 0;
        virtual void apply(Field3D &f) = 0;
        virtual void apply(Vector2D &f);
    }
virtual void apply(Vector3D &f);

/// Apply a boundary condition on ddt(f)
virtual void apply_ddt(Field2D &f);
virtual void apply_ddt(Field3D &f);
virtual void apply_ddt(Vector2D &f);
virtual void apply_ddt(Vector3D &f);

BoundaryRegion *bndry;
};

(where the implementations have been removed for clarity). Which has a pointer to a BoundaryRegion object specifying which region this boundary is operating on.

Boundary conditions need to be imposed on the initial conditions (after PhysicsModel::init()), and on the time-derivatives (after PhysicsModel::rhs()). The apply() functions are therefore called during initialisation and given the evolving variables, whilst the apply_ddt functions are passed the time-derivatives.

To implement a boundary operation, as a minimum the apply(Field2D), apply(Field2D) and clone() need to be implemented: By default the apply(Vector) will call the apply(Field) functions on each component individually, and the apply_ddt() functions just call the apply() functions.

Example: Neumann boundary conditions are defined in boundary_standard.hxx:

/// Neumann (zero-gradient) boundary condition
class BoundaryNeumann : public BoundaryOp {
public:
  BoundaryNeumann() {}  
  BoundaryNeumann(BoundaryRegion *region):BoundaryOp(region) {}  
  BoundaryOp* clone(BoundaryRegion *region, const list<string> &args);
  void apply(Field2D &f);
  void apply(Field3D &f);
};

and implemented in boundary_standard.cxx

void BoundaryNeumann::apply(Field2D &f) {
  // Loop over all elements and set equal to the next point in
  for(bndry->first(); !bndry->isDone(); bndry->next())
    f[bndry->x][bndry->y] = f[bndry->x - bndry->bx][bndry->y - bndry->by];
}

void BoundaryNeumann::apply(Field3D &f) {
  for(int z=0;z<mesh->LocalNz;z++)
    f[bndry->x][bndry->y][z] = f[bndry->x - bndry->bx][bndry->y - bndry->by][z];
}

This is all that’s needed in this case since there’s no difference between applying Neumann conditions to a variable and to its time-derivative, and Neumann conditions for vectors are just Neumann conditions on each vector component.

To create a boundary condition, we need to give it a boundary region to operate over:
BoundaryRegion *bndry = ...  
BoundaryOp op = new BoundaryOp(bndry);

The clone function is used to create boundary operations given a single object as a template in BoundaryFactory. This can take additional arguments as a vector of strings - see explanation in Boundary factory.

### 8.8 Boundary modifiers

To create more complicated boundary conditions from simple ones (such as Neumann conditions above), boundary operations can be modified by wrapping them up in a BoundaryModifier object, defined in boundary_op.hxx:

```cpp
class BoundaryModifier : public BoundaryOp {
public:
  virtual BoundaryOp* clone(BoundaryOp *op, const list<string> &args) = 0;
protected:
  BoundaryOp *op;
};
```

Since BoundaryModifier inherits from BoundaryOp, modified boundary operations are just a different boundary operation and can be treated the same (Decorator pattern). Boundary modifiers could also be nested inside each other to create even more complicated boundary operations. Note that the clone function is different to the BoundaryOp one: instead of a BoundaryRegion to operate on, modifiers are passed a BoundaryOp to modify.

Currently the only modifier is BoundaryRelax, defined in boundary_standard.hxx:

```cpp
/// Convert a boundary condition to a relaxing one
class BoundaryRelax : public BoundaryModifier {
public:
  BoundaryRelax(BoutReal rate) {r = fabs(rate);}  
  BoundaryOp* clone(BoundaryOp *op, const list<string> &args);

  void apply(Field2D &f);
  void apply(Field3D &f);

  void apply_ddt(Field2D &f);
  void apply_ddt(Field3D &f);
private:
  BoundaryRelax() {} // Must be initialised with a rate  
  BoutReal r;
};
```

### 8.9 Boundary factory

The boundary factory creates new boundary operations from input strings, for example turning “relax(dirichlet)” into a relaxing Dirichlet boundary operation on a given region. It is defined in boundary_factory.hxx as a Singleton, so to get a pointer to the boundary factory use

```cpp
BoundaryFactory *bfact = BoundaryFactory::getInstance();
```

and to delete this singleton, free memory and clean-up at the end use:
Because users should be able to add new boundary conditions during `PhysicsModel::init()`, boundary conditions are not hard-wired into `BoundaryFactory`. Instead, boundary conditions must be registered with the factory, passing an instance which can later be cloned. This is done in `bout++.cxx` for the standard boundary conditions:

```cpp
BoundaryFactory::cleanup();

BoundaryFactory* bndry = BoundaryFactory::getInstance();
bndry->add(new BoundaryDirichlet(), "dirichlet");
...
bndry->addMod(new BoundaryRelax(10.), "relax");
```

where the `add` function adds `BoundaryOp` objects, whereas `addMod` adds `BoundaryModifier` objects. **Note:** The objects passed to `BoundaryFactory` will be deleted when `cleanup()` is called.

When a boundary operation is added, it is given a name such as “dirichlet”, and similarly for the modifiers (“relax” above). These labels and object pointers are stored internally in `BoundaryFactory` in maps defined in `boundary_factory.hxx`:

```cpp
// Database of available boundary conditions and modifiers
map<string, BoundaryOp*> opmap;
map<string, BoundaryModifier*> modmap;
```

These are then used by `BoundaryFactory::create()`:

```cpp
/// Create a boundary operation object
BoundaryOp* create(const string &name, BoundaryRegion *region);
BoundaryOp* create(const char* name, BoundaryRegion *region);
```

to turn a string such as “relax(dirichlet)” and a `BoundaryRegion` pointer into a `BoundaryOp` object. These functions are implemented in `boundary_factory.cxx`, starting around line 42. The parsing is done recursively by matching the input string to one of:

- modifier(<expression>, arg1, ...)
- modifier(<expression>)
- operation(arg1, ...)
- operation

the `<expression>` variable is then resolved into a `BoundaryOp` object by calling `create(<expression>, region)`. When an operator or modifier is found, it is created from the pointer stored in the `opmap` or `modmap` maps using the `clone` method, passing a `list<string>` reference containing any arguments. It’s up to the operation implementation to ensure that the correct number of arguments are passed, and to parse them into floats or other types.

**Example:** The Dirichlet boundary condition can take an optional argument to change the value the boundary’s set to. In `boundary_standard.cxx`:

```cpp
BoundaryOp* BoundaryDirichlet::clone(BoundaryRegion *region, const list<string> &args) {
   if(!args.empty()) {
      // First argument should be a value
      stringstream ss;
      ss << args.front();
      BoutReal val;
```
If no arguments are passed i.e. the string was “dirichlet” or “dirichlet()” then the args list is empty, and the default value (0.0) is used. If one or more arguments is used then the first argument is parsed into a BoutReal type and used to create a new BoundaryDirichlet object. If more arguments are passed then these are just ignored; probably a warning should be printed.

To set boundary conditions on a field, FieldData methods are defined in field_data.hxx:

```cpp
void setBoundary(const string &name); ///< Set the boundary conditions
void setBoundary(const string &region, BoundaryOp *op); ///< Manually set
virtual void applyBoundary() {};
virtual void applyTDerivBoundary() {};\nprotected:
vector<BoundaryOp*> bndry_op; // Boundary conditions
```

The FieldData::setBoundary() method is implemented in field_data.cxx. It first gets a vector of pointers to BoundaryRegions from the mesh, then loops over these calling BoundaryFactory::createFromOptions() for each one and adding the resulting boundary operator to the FieldData::bndry_op vector.
There are three types of test used in BOUT++, in order of complexity: unit tests, integrated tests, and “method of manufactured solutions” (MMS) tests. Unit tests are very short, quick tests that test a single “unit” – usually a single function or method. Integrated tests are longer tests that range from tests that need a lot of set up and check multiple conditions, to full physics model tests. MMS tests check the numerical properties of operators, such as the error scaling of derivatives.

There is a test suite that runs through all of the unit tests, and selected integrated and MMS tests. The easiest way to run this is with:

```
$ make check
```

We expect that any new feature or function implemented in BOUT++ also has some corresponding tests, and strongly prefer unit tests.

### 9.1 Automated tests and code coverage

BOUT++ uses Travis CI to automatically run the test suite on every push to the GitHub repository, as well as on every submitted Pull Request. The Travis settings are in `.travis.yml`. Pull requests that fail the tests will not be merged.

We also gather information from how well the unit tests cover the library using CodeCov, the settings for which are stored in `.codecov.yml`.

### 9.2 Unit tests

The unit test suits aims to be a comprehensive set of tests that run very fast and ensure the basic functionality of BOUT++ is correct. At the time of writing, we have around 500 tests that run in less than a second. Because these tests run very quickly, they should be run on every commit (or even more often!). For more information on the unit tests, see `tests/unit/README.md`.

You can run the unit tests with:

```
$ make check-unit-tests
```
9.3 Integrated tests

This set of tests are designed to test that different components of the BOUT++ library work together. These tests are more expensive than the unit tests, but are expected to be run on at least every pull request, and the majority on every commit.

You can run the integrated tests with:

```
$ make check-integrated-tests
```

The test suite is in the `tests/integrated` directory, and is run using the `test_suite` python script. `tests/integrated/test_suite_list` contains a list of the subdirectories to run (e.g. `test-io, test-laplace, interchange-instability`). In each of those subdirectories the script `runtest` is executed, and the return value used to determine if the test passed or failed.

All tests should be short, otherwise it discourages people from running the tests before committing changes. A few minutes or less on a typical desktop, and ideally only a few seconds. If you have a large simulation which you want to stop anyone breaking, find starting parameters which are as sensitive as possible so that the simulation can be run quickly.

9.3.1 Custom test requirements

Some tests require particular libraries or environments, so should be skipped if these are not available. To do this, each `runtest` script can contain a line starting with `#requires`, followed by a python expression which evaluates to `True` or `False`. For example, a test which doesn’t work if both ARKODE and PETSc are used:

```
#requires not (arkode and petsc)
```

or if there were a test which required PETSc to be available, it could specify

```
#requires petsc
```

Currently the requirements which can be combined are `travis, netcdf, pnetcdf, hdf5, pvode, cvode, ida, lapack, petsc, slepc, mumps, arkode, openmp` and `make`. The `make` requirement is set to `True` when the tests are being compiled (but not run), and `False` when the scripts are run. It’s used for tests which do not have a compilation stage.

9.4 Method of Manufactured Solutions

The Method of Manufactured solutions (MMS) is a rigorous way to check that a numerical algorithm is implemented correctly. A known solution is specified (manufactured), and it is possible to check that the code output converges to this solution at the expected rate.

To enable testing by MMS, switch an input option “mms” to true:

```
[solver]
mms = true
```

This will have the following effect:

1. For each evolving variable, the solution will be used to initialise and to calculate the error
2. For each evolving variable, a source function will be read from the input file and added to the time derivative.
Note: The convergence behaviour of derivatives using FFTs is quite different to the finite difference methods: once the highest frequency in the manufactured solution is resolved, the accuracy will jump enormously, and after that, finer grids will not increase the accuracy. Whereas with finite difference methods, accuracy varies smoothly as the grid is refined.

9.4.1 Choosing manufactured solutions

Manufactured solutions must be continuous and have continuous derivatives. Common mistakes:

• Don’t use terms multiplying coordinates together e.g. $x \times z$ or $y \times z$. These are not periodic in $y$ and/or $z$, so will give strange answers and usually no convergence. Instead use $x \times \sin(z)$ or similar, which are periodic.

9.5 Timing

To time parts of the code, and calculate the percentage of time spent in communications, file I/O, etc. there is the Timer class defined in include/bout/sys/timer.hxx. To use it, just create a Timer object at the beginning of the function you want to time:

```cpp
#include <bout/sys/timer.hxx>

void someFunction() {
    Timer timer("test")
    ...
}
```

Creating the object starts the timer, and since the object is destroyed when the function returns (since it goes out of scope) the destructor stops the timer.

```cpp
class Timer {
public:
    Timer();
    Timer(const std::string &label);
    ~Timer();

    double getTime();
    double resetTime();
};
```

The empty constructor is equivalent to setting `label = ""`. Constructors call a private function `getInfo()` which looks up the `timer_info` structure corresponding to the label in a `map<string, timer_info*>`. If no such structure exists, then one is created. This structure is defined as:

```cpp
struct timer_info {
    double time;  // Total time
    bool running; // Is the timer currently running?
    double started; // Start time
};
```

Since each timer can only have one entry in the map, creating two timers with the same label at the same time will lead to trouble. Hence this code is not thread-safe.

9.5. Timing

87
The member functions `getTime()` and `resetTime()` both return the current time. Whereas `getTime()` only returns the time without modifying the timer, `resetTime()` also resets the timer to zero.

If you don’t have the object, you can still get and reset the time using static methods:

```cpp
double Timer::getTime(const std::string &label);
double Timer::resetTime(const std::string &label);
```

These look up the `timer_info` structure, and perform the same task as their non-static namesakes. These functions are used by the monitor function in `bout++.cxx` to print the percentage timing information.
BOUT++ OPTIONS

The inputs to BOUT++ are a text file containing options, command-line options, and for complex grids a binary grid file in NetCDF or HDF5 format. Generating input grids for tokamaks is described in *Generating input grids*. The grid file describes the size and topology of the X-Y domain, metric tensor components and usually some initial profiles. The option file specifies the size of the domain in the symmetric direction (Z), and controls how the equations are evolved e.g. differencing schemes to use, and boundary conditions. In most situations, the grid file will be used in many different simulations, but the options may be changed frequently.

All options used in a simulation are saved to a `BOUT.settings` file. This includes values which are not explicitly set in `BOUT.inp`.

### 10.1 BOUT.inp input file

The text input file `BOUT.inp` is always in a subdirectory called `data` for all examples. The files include comments (starting with either `;` or `#`) and should be fairly self-explanatory. The format is the same as a windows INI file, consisting of `name = value` pairs. Any type which can be read from a stream using the `>>` operator can be stored in an option (see later for the implementation details). Supported value types include:

- Integers
- Real values
- Booleans
- Strings

Options are also divided into sections, which start with the section name in square brackets.

```
[section1]
something = 132   # an integer
another = 5.131   # a real value
??? = true       # a boolean
????? = "some text"  # a string
```

Option names can contain almost any character except `=` and `:`, including unicode. If they start with a number or `.` contain arithmetic symbols (+-*/*), brackets ( () {} []), whitespace or comma `,`, then these will need to be escaped in expressions. See below for how this is done.

Subsections can also be used, separated by colons `:` e.g.

```
[section:subsection]
```

Numerical quantities can be plain numbers or expressions:
Variables can even reference other variables:

\[
\text{pressure} = \text{temperature} \times \text{density} \\
\text{temperature} = 12 \\
\text{density} = 3
\]

Note that variables can be used before their definition; all variables are first read, and then processed afterwards. The value \(\pi\) is already defined, as is \(?\), and can be used in expressions.

Uses for expressions include initialising variables Expressions and input sources, defining grids Generating input grids and MMS convergence tests Method of Manufactured Solutions.

Expressions can include addition (+), subtraction (-), multiplication (*), division (/) and exponentiation (^) operators, with the usual precedence rules. In addition to ?, expressions can use predefined variables \(x\), \(y\), \(z\) and \(t\) to refer to the spatial and time coordinates. A number of functions are defined, listed in table Table 7.1. One slightly unusual feature is that if a number comes before a symbol or an opening bracket (()) then a multiplication is assumed: \(2x+3y^2\) is the same as \(2\times x + 3\times y\times y\), which with the usual precedence rules is the same as \((2\times x) + (3\times (y^2))\).

All expressions are calculated in floating point and then converted to an integer when read inside BOUT++. The conversion is done by rounding to the nearest integer, but throws an error if the floating point value is not within 1e − 3 of an integer. This is to minimise unexpected behaviour. If you want to round any result to an integer, use the \texttt{round} function:

\[
\text{bad\_integer} = 256.4 \\
\text{ok\_integer} = \text{round}(256.4)
\]

Note that it is still possible to read \texttt{bad\_integer} as a real number, since the type is determined by how it is used.

Have a look through the examples to see how the options are used.

### 10.1 Special symbols in Option names

If option names start with numbers or . or contain symbols such as + and - then these symbols need to be escaped in expressions or they will be treated as arithmetic operators like addition or subtraction. To escape a single character \(\texttt{\textbackslash}\) (backslash) can be used, for example \texttt{plasma\textbackslash\text{-density} * 10} would read the option \texttt{plasma\_density} and multiply it by 10 e.g

\[
\text{plasma\_density} = 1e19 \\
\text{2ndvalue} = 10 \\
\text{value} = \text{plasma\textbackslash\text{-density} * \2ndvalue}
\]

To escape multiple characters, \(\texttt{\textbackslash}\) (backquote) can be used:

\[
\text{plasma\_density} = 1e19 \\
\text{2ndvalue} = 10 \\
\text{value} = \text{`plasma\_density` * `2ndvalue`}
\]

The character \(\texttt:\) cannot be part of an option or section name, and cannot be escaped, as it is always used to separate sections.
10.2 Command line options

Command-line switches are:

<table>
<thead>
<tr>
<th>Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h, --help</td>
<td>Prints a help message and quits</td>
</tr>
<tr>
<td>-v, --verbose</td>
<td>Outputs more messages to BOUT.log files</td>
</tr>
<tr>
<td>-q, --quiet</td>
<td>Outputs fewer messages to log files</td>
</tr>
<tr>
<td>-d &lt;directory&gt;</td>
<td>Look in &lt;directory&gt; for input/output files (default “data”)</td>
</tr>
<tr>
<td>-f &lt;file&gt;</td>
<td>Use OPTIONS given in &lt;file&gt;</td>
</tr>
<tr>
<td>-o &lt;file&gt;</td>
<td>Save used OPTIONS given to &lt;file&gt; (default BOUT.settings)</td>
</tr>
</tbody>
</table>

In addition all options in the BOUT.inp file can be set on the command line, and will override those set in BOUT.inp. The most commonly used are “restart” and “append”, described in Running BOUT++. If values are not given for command-line arguments, then the value is set to true, so putting restart is equivalent to restart=true.

Values can be specified on the command line for other settings, such as the fraction of a torus to simulate (ZPERIOD):

```
./command zperiod=10
```

Remember no spaces around the ‘=’ sign. Like the BOUT.inp file, setting names are not case sensitive.

Sections are separated by colons ‘:’, so to set the solver type (Options) you can either put this in BOUT.inp:

```
[solver]
type = rk4
```

or put solver:type=rk4 on the command line. This capability is used in many test suite cases to change the parameters for each run.

10.3 General options

At the top of the BOUT.inp file (before any section headers), options which affect the core code are listed. These are common to all physics models, and the most useful of them are:

```
NOUT = 100 # number of time-points output
TIMESTEP = 1.0 # time between outputs
```

which set the number of outputs, and the time step between them. Note that this has nothing to do with the internal timestep used to advance the equations, which is adjusted automatically. What time-step to use depends on many factors, but for high-\(\beta\) reduced MHD ELM simulations reasonable choices are 1.0 for the first part of a run (to handle initial transients), then around 10.0 for the linear phase. Once non-linear effects become important, you will have to reduce the timestep to around 0.1.

Most large clusters or supercomputers have a limit on how long a job can run for called “wall time”, because it’s the time taken according to a clock on the wall, as opposed to the CPU time actually used. If this is the case, you can use the option

```
wall_limit = 10 # wall clock limit (in hours)
```

BOUT++ will then try to quit cleanly before this time runs out. Setting a negative value (default is -1) means no limit.

Often it’s useful to be able to restart a simulation from a chosen point, either to reproduce a previous run, or to modify the settings and re-run. A restart file is output every timestep, but this is overwritten each time, and so the simulation
can only be continued from the end of the last simulation. Whilst it is possible to create a restart file from the output data afterwards, it’s much easier if you have the restart files. Using the option

```
archive = 20
```

saves a copy of the restart files every 20 timesteps, which can then be used as a starting point.

### 10.3.1 Grids

You can set the size of the computational grid in the mesh section of the input file (see *Generating input grids* for more information):

```
[mesh]
nx = 20  # Number of points in X
ny = 16  # Number of points in Y
nz = 32  # Number of points in Z
```

Due to historical reasons, `nx` is defined differently to `ny` and `nz`:

- `nx` is the number of points in X **including** the boundaries
- `ny` and `nz` are the number of points in Y and Z **not including** the boundaries

The default number of boundary points in X is 2, so taking into account the boundary at each end of the domain, `nx` usually means “the number of interior grid points in X plus four”. In the example above, both X and Y have 16 interior grid points.

It is recommended, but not necessary, that this be `nz = 2^n`, that is 1, 2, 4, 8, … This is because FFTs are usually slightly faster with power-of-two length arrays, and FFTs are used quite frequently in many models.

**Note:** In previous versions of BOUT++, `nz` was constrained to be a power-of-two, and had to be specified as a power-of-two plus one (i.e. a number of the form $2^n + 1$ like 2, 3, 5, 9, …) in order to account for an additional, unused, point in Z. Both of these conditions were relaxed in BOUT++ 4.0. If you use an input file from a previous version, check that this superfluous point is not included in `nz`.

Since the Z dimension is periodic, the domain size is specified as multiples or fractions of $2\pi$. To specify a fraction of $2\pi$, use

```
ZPERIOD = 10
```

This specifies a Z range from 0 to $2\pi / \text{ZPERIOD}$, and is useful for simulation of tokamaks to make sure that the domain is an integer fraction of a torus. If instead you want to specify the Z range directly (for example if Z is not an angle), there are the options

```
ZMIN = 0.0
ZMAX = 0.1
```

which specify the range in multiples of $2\pi$.

In BOUT++, grids can be split between processors in both X and Y directions. By default BOUT++ automatically divides the grid in both X and Y, finding the decomposition with domains closest to square, whilst satisfying constraints. These constraints are:

- Every processor must have the same size and shape domain
- Branch cuts, mostly at X-points, must be on processor boundaries. This is because the connection between grid points is modified in BOUT++ by changing which processors communicate.
To specify a splitting manually, the number of processors in the X direction can be specified:

```
NXPE = 1  # Set number of X processors
```

Alternatively, the number in the Y direction can be specified (if both are given, NXPE takes precedence and NYPE is ignored):

```
NYPE = 1  # Set number of Y processors
```

When choosing NXPE or NYPE, they must also obey some constraints:

- **NXPE** must be a factor of the number of grid points in the x-direction
  - That is, \((nx - 4) / NXPE\) must be an integer, assuming the usual two boundary points
- **NYPE** must be a factor of the number of grid points in the y-direction
  - That is, \(ny \div NYPE\) must be an integer
- For more general topologies, the number of points per processor \(ny / NYPE\) must also be a factor of the number of points in each region. For example, in the usual tokamak topologies:
  - in single-null there are two divertor leg and one core regions
  - in double-null there are four divertor leg, one inner core and one outer core regions

Please note that here "core" means “core and adjacent SOL”. See *BOUT++ Topology* for a more detailed explanation of these regions.

When BOUT++ automatically chooses NXPE and NYPE it finds all valid pairs which give total number of processors \(= NPES = NXPE \times NYPE\) and also satisfy the constraints above. It then chooses the pair that makes the grid on each processor as close to square as possible (technically it chooses the pair that minimises \(\text{abs}(\sqrt{NPES \times (nx - 4) / ny} - NXPE)\)).

If you need to specify complex input values, e.g. numerical values from experiment, you may want to use a grid file. The grid file to use is specified relative to the root directory where the simulation is run (i.e. running “ls ./data/ Bout.inp” gives the options file). You can use the global option grid, or mesh:file:

```
grid = "data/cbm18_8_y064_x260.nc"
# Alternatively:
[mesh]
file = "data/cbm18_8_y064_x260.nc"
```

### 10.4 Communications

The communication system has a section [comms], with a true/false option async. This determines whether asynchronous MPI sends are used; which method is faster varies (though not by much) with machine and problem.
10.5 Differencing methods

Differencing methods are specified in the section ([mesh:ddx], [mesh:ddy], [mesh:ddz] and [mesh:diff]), one for each dimension. The [mesh:diff] section is only used if the section for the dimension does not contain an option for the differencing method. Note that [mesh] is the name of the section passed to the mesh constructor, which is most often mesh - but could have another name, e.g. if multiple meshes are used.

- **first**, the method used for first derivatives
- **second**, method for second derivatives
- **fourth**, method for fourth derivatives
- **upwind**, method for upwinding terms
- **flux**, for conservation law terms

The methods which can be specified include U1, U4, C2, C4, W2, W3, FFT Apart from FFT, the first letter gives the type of method (U = upwind, C = central, W = WENO), and the number gives the order.

The staggered derivatives can be specified as `FirstStag` or if the value is not set, then `First` is checked. Note that for the staggered quantities, if the staggered quantity in a dimension is not set, first the staggered quantity in the [mesh:diff] section is checked. This is useful, as the staggered quantities are more restricted in the available choices than the non-staggered differencing operators.

10.6 Model-specific options

The options which affect a specific physics model vary, since they are defined in the physics module itself (see Input options). They should have a separate section, for example the high-\(\beta\) reduced MHD code uses options in a section called [highbeta].

There are three places to look for these options: the BOUT.inp file; the physics model C++ code, and the output logs. The physics module author should ideally have an example input file, with commented options explaining what they do; alternately they may have put comments in the C++ code for the module. Another way is to look at the output logs: when BOUT++ is run, (nearly) all options used are printed out with their default values. This won’t provide much explanation of what they do, but may be useful anyway. See Post-processing for more details.

10.7 Input and Output

The format of the output (dump) files can be controlled, if support for more than one output format has been configured, by setting the top-level option `dump_format` to one of the recognised file extensions: ‘nc’ for NetCDF; ‘hdf5’, ‘ hdf’ or ‘h5’ for HDF5. For example to select HDF5 instead of the default NetCDF format put

```
dump_format = hdf5
```

before any section headers. The output (dump) files with time-history are controlled by settings in a section called “output”. Restart files contain a single time-slice, and are controlled by a section called “restart”. The options available are listed in table Table 10.1.
**Table 10.1: Output file options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabled</td>
<td>Writing is enabled</td>
<td>true</td>
</tr>
<tr>
<td>floats</td>
<td>Write floats rather than doubles</td>
<td>false</td>
</tr>
<tr>
<td>flush</td>
<td>Flush the file to disk after each write</td>
<td>true</td>
</tr>
<tr>
<td>guards</td>
<td>Output guard cells</td>
<td>true</td>
</tr>
<tr>
<td>openclose</td>
<td>Re-open the file for each write, and close after</td>
<td>true</td>
</tr>
<tr>
<td>parallel</td>
<td>Use parallel I/O</td>
<td>false</td>
</tr>
</tbody>
</table>

**enabled** is useful mainly for doing performance or scaling tests, where you want to exclude I/O from the timings. **floats** can be used to reduce the size of the output files: files are stored as double by default, but setting **floats = true** changes the output to single-precision floats.

To enable parallel I/O for either output or restart files, set

```plaintext
parallel = true
```

in the output or restart section. If you have compiled BOUT++ with a parallel I/O library such as pnetcdf (see Advanced installation options), then rather than outputting one file per processor, all processors will output to the same file. For restart files this is particularly useful, as it means that you can restart a job with a different number of processors. Note that this feature is still experimental, and incomplete: output dump files are not yet supported by the collect routines.

## 10.8 Implementation

To control the behaviour of BOUT++ a set of options is used, with options organised into sections which can be nested. To represent this tree structure there is the `Options` class defined in `bout++/include/options.hxx`.

To access the options, there is a static function (singleton):

```plaintext
auto& options = Options::root();
```

which returns a reference (type `Options&`). Note that without the & the options tree will be copied, so any changes made will not be retained in the global tree. Options can be set by assigning, treating options as a map or dictionary:

```plaintext
options["nout"] = 10;     // Integer
options["restart"] = true;  // bool
```

Internally these values are stored in a variant type, which supports commonly used types including strings, integers, real numbers and fields (2D and 3D). Since strings can be stored, any type can be assigned, so long as it can be streamed to a string (using `<<` operator and a `std::stringstream`).

Often it's useful to see where an option setting has come from e.g. the name of the options file or “command line”. To specify a source, use the `assign` function to assign values:

```plaintext
options["nout"].assign(10, "manual");
```

A value cannot be assigned more than once with different values and the same source (“manual” in this example). This is to catch a common error in which a setting is inconsistently specified in an input file. To force a value to change, overwriting the existing value (if any):
options["nout"].force(20, "manual");

Sub-sections are created as they are accessed, so a value in a sub-section could be set using:

```cpp
auto& section = options["mysection"];
section["myswitch"] = true;
```

or just:

```cpp
options["mysection"]["myswitch"] = true;
```

Names including sections, subsections, etc. can be specified using ":" as a separator, e.g.:

```cpp
options["mysection:mysubsection:myswitch"] = true;
```

To get options, they can be assigned to a variable:

```cpp
int nout = options["nout"];
```

If the option is not found then a BoutException will be thrown. A default value can be given, which will be used if the option has not been set:

```cpp
int nout = options["nout"].withDefault(1);
```

If options is not const, then the given default value will be cached. If a default value has already been cached for this option, then the default values must be consistent: A BoutException is thrown if inconsistent default values are detected.

The default can also be set from another option. This may be useful if two or more options should usually be changed together:

```cpp
BoutReal value2 = options["value2"].withDefault(options["value1"]);
```

Note that if the result should be a real number (e.g. BoutReal) then withDefault should be given a real. Otherwise it will convert the number to an integer:

```cpp
BoutReal value = options["value"].withDefault(42); // Convert to integer
BoutReal value = options["value"].withDefault(42.0); // ok
auto value = options["value"].withDefault<BoutReal>(42); // ok
```

It is common for BOUT++ models to read in many settings which have the same variable name as option setting (e.g. “nout” here). A convenient macro reads options into an already-defined variable:

```cpp
int nout;
OPTION(options, nout, 1);
```

where the first argument is a section, second argument is the variable whose name will also be used as the option string, and third argument is the default value.

Every time an option is accessed, a message is written to output_info. This message includes the value used and the source of that value. By default this message is printed to the terminal and saved in the log files, but this can be disabled by changing the logging level: Add -q to the command line to reduce logging level. See section Logging output for more details about logging.

The type to be returned can also be specified as a template argument:
Any type can be used which can be streamed (operator `>>`) from a `stringstream`. There are special implementations for `bool`, `int` and `BoutReal` which enable use of expressions in the input file. The type can also be specified to `withDefault`, or will be inferred from the argument:

```cpp
BoutReal nout = options["nout"].as<BoutReal>();
```

### 10.8.1 Documentation

Options can be given a `doc` attribute describing what they do. This documentation will then be written to the `BOUT` settings file at the end of a run:

```cpp
Te0 = options["Te0"].doc("Temperature in eV").withDefault(30.0);
```

The `.doc()` function returns a reference `Options&` so can be chained with `withDefault` or as functions, or as part of an assignment:

```cpp
options["value"].doc("Useful setting info") = 42;
```

This string is stored in the attributes of the option:

```cpp
std::string docstring = options["value"].attributes["doc"];```

### 10.8.2 Overriding library defaults

`BOUT++` sets defaults for options controlling the mesh, etc. A physics model (or other user code) can override these defaults by using the convenience macro `BOUT_OVERRIDE_DEFAULT_OPTION`, for example if you want to change the default value of `mesh::staggergrids` from false to true, put (outside any class/function body):

```cpp
BOUT_OVERRIDE_DEFAULT_OPTION("mesh:staggergrids", true);
```

### 10.8.3 Older interface

Some code in `BOUT++` currently uses an older interface to `Options` which uses pointers rather than references. Both interfaces are currently supported, but use of the newer interface above is encouraged.

To access the options, there is a static function (singleton):

```cpp
Options *options = Options::getRoot();
```

which gives the top-level (root) options class. Setting options is done using the `set()` methods which are currently defined for `int`, `BoutReal`, `bool` and `string`. For example:

```cpp
options->set("nout", 10);   // Set an integer
options->set("restart", true); // A bool
```

Often it’s useful to see where an option setting has come from e.g. the name of the options file or “command line”. To specify a source, pass it as a third argument:
options->set("nout", 10, "manual");

To create a section, just use `getSection`: if it doesn’t exist it will be created:

```cpp
Options *section = options->getSection("mysection");
section->set("myswitch", true);
```

To get options, use the `get()` method which take the name of the option, the variable to set, and the default value:

```cpp
int nout;
options->get("nout", nout, 1);
```

Internally, `Options` converts all types to strings and does type conversion when needed, so the following code would work:

```cpp
Options *options = Options::getRoot();
options->set("test", "123");
int val;
options->get("test", val, 1);
```

This is because often the type of the option is not known at the time when it’s set, but only when it’s requested.

## 10.9 Reading options

To allow different input file formats, each file parser implements the `OptionParser` interface defined in `bout++/src/sys/options/optionparser.hxx`:

```cpp
class OptionParser {
    public:
        virtual void read(Options *options, const string &filename) = 0;
    private:
};
```

and so just needs to implement a single function which reads a given file name and inserts the options into the given `Options` object.

To use these parsers and read in a file, there is the `OptionsReader` class defined in `bout++/include/optionsreader.hxx`:

```cpp
class OptionsReader {
    public:
        void read(Options *options, const char *file, ...);
        void parseCommandLine(Options *options, int argc, char **argv);
    }
```

This is a singleton object which is accessed using:

```cpp
OptionsReader *reader = OptionsReader::getInstance();
```

so to read a file `BOUT.inp` in a directory given in a variable `data_dir` the following code is used in `bout++.cxx`:
Options *options = Options::getRoot();
OptionsReader *reader = OptionsReader::getInstance();
reader->read(options, "%s/BOUT.inp", data_dir);

To parse command line arguments as options, the OptionsReader class has a method:

reader->parseCommandLine(options, argc, argv);

This is currently quite rudimentary and needs improving.

## 10.10 Reading and writing to NetCDF

If NetCDF4 support is enabled, then the OptionsNetCDF class provides an experimental way to read and write options. To use this class:

```cpp
#include "options_netcdf.hxx"
using bout::experimental::OptionsNetCDF;
```

Examples are in integrated test tests/integrated/test-options-netcdf/

To write the current Options tree (e.g. from BOUT.inp) to a NetCDF file:

```cpp
OptionsNetCDF("settings.nc").write(Options::root());
```

and to read it in again:

```cpp
Options data = OptionsNetCDF("settings.nc").read();
```

Fields can also be stored and written:

```cpp
Options fields;
fields["f2d"] = Field2D(1.0);
fields["f3d"] = Field3D(2.0);
OptionsNetCDF("fields.nc").write(fields);
```

This should allow the input settings and evolving variables to be combined into a single tree (see above on joining trees) and written to the output dump or restart files.

Reading fields is a bit more difficult. Currently 1D data is read as an `Array<BoutReal>`, 2D as `Matrix<BoutReal>` and 3D as `Tensor<BoutReal>`. These can be extracted directly from the Options tree, or converted to a Field:

```cpp
Options fields_in = OptionsNetCDF("fields.nc").read();
Field2D f2d = fields_in["f2d"].as<Field2D>();
Field3D f3d = fields_in["f3d"].as<Field3D>();
```

Note that by default reading as Field2D or Field3D will use the global `bout::globals::mesh`. To use a different mesh, or different cell location, pass a field which the result should be similar to:

```cpp
Field3D example = ... // Some existing field
Field3D f3d = fields_in["f3d"].as<Field3D>(example);
```

Meta data like `Mesh` pointer, will be taken from `example`. 
Currently converting from Matrix or Tensor types only works if the data in the Matrix or Tensor is the same size as the Field. In the case of grid files, the fields only needs a part of the global values. Some kind of mapping from the global index to local index is needed, probably defined by Mesh. For now it should be possible to be compatible with the current system, so that all quantities from the grid file are accessed through Mesh::get.

### 10.10.1 Time dependence

When writing NetCDF files, some variables should have a time dimension added, and then be added to each time they are written. This has been implemented using an attribute: If variables in the Options tree have an attribute “time_dimension” then that is used as the name of the time dimension in the output file. This allows multiple time dimensions e.g. high frequency diagnostics and low frequency outputs, to exist in the same file:

```cpp
Options data;
data["scalar"] = 1.0;
data["scalar"].attributes["time_dimension"] = "t";

data["field"] = Field3D(2.0);
data["field"].attributes["time_dimension"] = "t";

OptionsNetCDF("time.nc").write(data);

// Update time-dependent values. This can be done without `force` if the time_dimension
// attribute is set
data["scalar"] = 2.0;
data["field"] = Field3D(3.0);

// Append data to file
OptionsNetCDF("time.nc", OptionsNetCDF::FileMode::append).write(data);
```

Some issues:

- Currently all variables in the Options tree are written when passed to OptionsNetCDF::write. This means that the variables with different time dimensions should be stored in different Options trees, so they can be written at different times. One possibility is to have an optional argument to write, so that only variables with one specified time dimension are updated.

### 10.11 FFT

There is one global option for Fourier transforms, fft_measure (default: false). Setting this to true enables the FFTW_MEASURE mode when performing FFTs, otherwise FFTW_ESTIMATE is used:

```cpp
[fft]
fft_measure = true
```

In FFTW_MEASURE mode, FFTW runs and measures how long several FFTs take, and tries to find the optimal method.

**Note:** Technically, FFTW_MEASURE is non-deterministic and enabling fft_measure may result in slightly different answers from run to run, or be dependent on the number of MPI processes. This may be important if you are trying to benchmark or measure performance of your code.

See the FFTW FAQ for more information.
BOUT++ Documentation, Release 4.4.0

10.12 Types for multi-valued options
An enum class can be a useful construct for options in a physics model. It can have an arbitrary number of userdefined, named values (although the code in include/bout/bout_enum_class.hxx needs extending for more than
10 values). The advantage over using a std::string for an option is that a typo cannot produce an unexpected
value: in C++ code it is a compile-time error and reading from BOUT.inp it is a run-time exception. We provide a
utility macro BOUT_ENUM_CLASS to define an enum class with some extra convenience methods. For example, after
defining myoption like:
BOUT_ENUM_TYPE(myoption, foo, bar, baz);
it is possible not only to test for a value, e.g.:
myoption x = <something>;
...
if (x == myoption::foo) {
do a foo thing
}
but also to convert the option to a string:
std::string s = toString(x);
pass it to a stream:
output << x;
or get an option like myinput=baz from an input file or the command line as a myoption:
myoption y = Options::root()["myinput"].as<myoption>();
or with a default value:
myoption y = Options::root()["myinput"].withDefault(myoption::bar);
Only strings exactly (but case-insensitively) matching the name of one of the defined myoption values are allowed,
anything else results in an exception being thrown.

10.12. Types for multi-valued options

101


GENERATING INPUT GRIDS

The simulation mesh describes the number and topology of grid points, the spacing between them, and the coordinate system. For many problems, a simple mesh can be created using options.

```
[mesh]
nx = 260  # X grid size
ny = 256  # Y grid size
dx = 0.1  # X mesh spacing
dy = 0.1  # Y mesh spacing
```

The above options will create a $260 \times 256$ mesh in X and Y (MZ option sets Z resolution), with mesh spacing of 0.1 in both directions. By default the coordinate system is Cartesian (metric tensor is the identity matrix), but this can be changed by specifying the metric tensor components.

Integer quantities such as \texttt{nx} can be numbers (like “260”), or expressions (like “256 + 2*mxg”). A common use is to make \texttt{x} and \texttt{z} dimensions have the same number of points, when \texttt{x} has \texttt{mxg} boundary cells on each boundary but \texttt{z} does not (since it is usually periodic):

```
[mesh]
nx = nz + 2*mxg  # X grid size
nz = 256          # Z grid size
mxg = 2
```

Note that the variable \texttt{nz} can be used before its definition; all variables are first read, and then processed afterwards.

All expressions are calculated in floating point and then converted to an integer. The conversion is done by rounding to the nearest integer, but throws an error if the floating point value is not within 1e-3 of an integer. This is to minimise unexpected behaviour. If you want to round any result to an integer, use the \texttt{round} function:

```
[mesh]
nx = 256.4  # Error!
x = round(256.4)  # ok
```

Real (floating-point) values can also be expressions, allowing quite complicated analytic inputs. For example in the example \texttt{test-griddata}:

```
# Screw pinch
rwidth = 0.4
Rxy = 0.1 + rwidth*x  # Radius from axis  [m]
L = 10                 # Length of the device [m]
```

(continues on next page)
These expressions use the same mechanism as used for variable initialisation \((Expressions)\): \(x\) is a variable from 0 to 1 in the domain which is uniform in index space; \(y\) and \(z\) go from 0 to \(2\pi\). As with variable initialisation, common trigonometric and mathematical functions can be used. In the above example, some variables depend on each other, for example \(dy\) depends on \(L\) and \(ny\). The order in which these variables are defined doesn’t matter, so \(L\) could be defined below \(dy\), but circular dependencies are not allowed. If the variables are defined in the same section (as \(dy\) and \(L\)) then no section prefix is required. To refer to a variable in a different section, prefix the variable with the section name e.g. “\(section:variable\)”.

More complex meshes can be created by supplying an input grid file to describe the grid points, geometry, and starting profiles. Currently BOUT++ supports either NetCDF, HDF5 format binary files. During startup, BOUT++ looks in the grid file for the following variables. If any are not found, a warning will be printed and the default values used.

- X and Y grid sizes (integers) \(nx\) and \(ny\) \textbf{REQUIRED}
- Differencing quantities in 2D arrays \(dx[nx][ny]\) and \(dy[nx][ny]\). If these are not found they will be set to 1.
- Diagonal terms of the metric tensor \(g^{ij}\) \(g_{11}[nx][ny]\), \(g_{22}[nx][ny]\), and \(g_{33}[nx][ny]\). If not found, these will be set to 1.
- Off-diagonal metric tensor \(g^{ij}\) elements \(g_{12}[nx][ny]\), \(g_{13}[nx][ny]\), and \(g_{23}[nx][ny]\). If not found, these will be set to 0.
- \(Z\) shift for interpolation between field-aligned coordinates and shifted coordinates (see manual/coordinates.pdf). Perpendicular differential operators are calculated in shifted coordinates when \texttt{ShiftXderivs} in \texttt{mesh/mesh.hxx} is enabled. \texttt{ShiftXderivs} can be set in the root section of \texttt{BOUT.inp} as \texttt{ShiftXderivs = true}.

The shifts must be provided in the gridfile in a field \texttt{zshift[nx][ny]}. If not found, \texttt{zshift} is set to zero.

The remaining quantities determine the topology of the grid. These are based on tokamak single/double-null configurations, but can be adapted to many other situations.

- Separatrix locations \texttt{ixseps1} and \texttt{ixseps2} If neither is given, both are set to \(nx\) (i.e. all points in closed “core” region). If only \texttt{ixseps1} is found, \texttt{ixseps2} is set to \(nx\), and if only \texttt{ixseps2} is found, \texttt{ixseps1} is set to -1.
- Branch-cut locations \texttt{jayseps1_1}, \texttt{jayseps1_2}, \texttt{jayseps2_1}, and \texttt{jayseps2_2}
- Twist-shift matching condition \texttt{ShiftAngle[nx]} for field aligned coordinates. This is applied in the “core” region between indices \texttt{jayseps2_2}, and \texttt{jayseps1_1 + 1}, if either \texttt{TwistShift = True} enabled in the options file or in general the \texttt{TwistShift} flag in \texttt{mesh/impls/bout/boutmesh.hxx} is enabled by other means. BOUT++ automatically reads the twist shifts in the gridfile if the shifts are stored in a field \texttt{ShiftAngle[nx]}; \texttt{ShiftAngle} must be given in the gridfile or grid-options if \texttt{TwistShift = True}.

The only quantities which are required are the sizes of the grid. If these are the only quantities specified, then the coordinates revert to Cartesian.

This section describes how to generate inputs for tokamak equilibria. If you’re not interested in tokamaks then you can skip to the next section.
The directory `tokamak_grids` contains code to generate input grid files for tokamaks. These can be used by the `2fluid` and `highbeta_reduced` modules, and are (mostly) compatible with inputs to the BOUT-06 code.

### 11.1 BOUT++ Topology

#### 11.1.1 Basic

In order to handle tokamak geometry BOUT++ contains an internal topology which is determined by the branch-cut locations (`jyseps1_1, jyseps1_2, jyseps2_1, and jyseps2_2`) and separatrix locations (`ixseps1` and `ixseps2`).

The separatrix locations, `ixseps1` and `ixseps2`, give the indices in the x domain where the first and second separatrices are located.

If `ixseps1 == ixseps2` then there is a single separatrix representing the boundary between the core region and the SOL region and the grid is a connected double null configuration. If `ixseps1 > ixseps2` then there are two separatrices and the inner separatrix is `ixseps2` so the tokamak is an upper double null. If `ixseps1 < ixseps2` then there are two separatrices and the inner separatrix is `ixseps1` so the tokamak is a lower double null.

In other words: Let us for illustrative purposes say that `ixseps1 > ixseps2` (see Fig. 11.1). Let us say that we have a field `f(x, y, z)` with a global x-index which includes ghost points. `f(x<=xseps1, y, z)` will then be periodic in the y-direction, `f(xseps1<x<=xseps2, y, z)` will have boundary condition in the y-direction set by the lowermost `ydown` and `yup`. If `f(xseps2<x, y, z)` the boundary condition in the y-direction will be set by the uppermost `ydown` and `yup`. As for now, there is no difference between the two sets of upper and lower `ydown` and `yup` boundary conditions (unless manually specified, see `Custom boundary conditions`).

These values are set either in the grid file or in `BOUT.inp`. Fig. 11.1 shows schematically how `ixseps` is used.

The branch cut locations, `jyseps1_1, jyseps1_2, jyseps2_1, and jyseps2_2`, split the y domain into logical regions defining the SOL, the PFR (private flux region) and the core of the tokamak. This is illustrated also in Fig. 11.1. If `jyseps1_2 == jyseps2_1` then the grid is a single null configuration, otherwise the grid is a double null configuration.

#### 11.1.2 Advanced

The internal domain in BOUT++ is deconstructed into a series of logically rectangular sub-domains with boundaries determined by the `ixseps` and `jyseps` parameters. The boundaries coincide with processor boundaries so the number of grid points within each sub-domain must be an integer multiple of `ny/nypes` where `ny` is the number of grid points in y and `nypes` is the number of processors used to split the y domain. Processor communication across the domain boundaries is then handled internally. Fig. 11.2 shows schematically how the different regions of a double null tokamak with `ixseps1 = ixseps2` are connected together via communications.

**Note:** To ensure that each subdomain follows logically, the `jyseps` indices must adhere to the following conditions:

- `jyseps1_1 > -1`
- `jyseps2_1 >= jyseps1_1 + 1`
- `jyseps2_2 >= jyseps1_2`
- `jyseps2_2 >= jyseps1_2`
- `jyseps2_2 <= ny - 1`

To ensure that communications work branch cuts must align with processor boundaries.
Fig. 11.1: Deconstruction of a poloidal tokamak cross-section into logical domains using the parameters $ixseps1$, $ixseps2$, $jyseps1_1$, $jyseps1_2$, $jyseps2_1$, and $jyseps2_2$
The topology of BOUT++ connects these sections together to give

Fig. 11.2: Schematic illustration of domain decomposition and communication in BOUT++ with \( \text{ixseps1} = \text{ixseps2} \)
11.1.3 Implementations

In BOUT++ each processor has a logically rectangular domain, so any branch cuts needed for X-point geometry (see Fig. 11.2) must be at processor boundaries.

In the standard “bout” mesh (src/mesh/impls/bout/), the communication is controlled by the variables

```c
int UDATA_INDEST, UDATA_OUTDEST, UDATA_XSPLIT;
int DDATA_INDEST, DDATA_OUTDEST, DDATA_XSPLIT;
int IDATA_DEST, ODATA_DEST;
```

These control the behavior of the communications as shown in Fig. 11.3.

![Communication of guard cells in BOUT++](image)

Fig. 11.3: Communication of guard cells in BOUT++. Boundaries in X have only one neighbour each, but boundaries in Y can be split into two, allowing branch cuts.

In the Y direction, each boundary region (Up and Down in Y) can be split into two, with 0 \( \leq x < \) UDATA_XSPLIT going to the processor index UDATA_INDEST, and UDATA_INDEST \( \leq x < \) LocalN\( x \) going to UDATA_OUTDEST. Similarly for the Down boundary. Since there are no branch-cuts in the X direction, there is just one destination for the Inner and Outer boundaries. In all cases a negative processor number means that there’s a domain boundary so no communication is needed.

The communication control variables are set in the `topology()` function, in `src/mesh/impls/bout/boutmesh.cxx` starting around line 2056. First the function `default_connections()` sets the topology to be a rectangle.

To change the topology, the function `set_connection()` checks that the requested branch cut is on a processor boundary, and changes the communications consistently so that communications are two-way and there are no “dangling” communications.
11.2 3D variables

BOUT++ was originally designed for tokamak simulations where the input equilibrium varies only in X-Y, and Z is used as the axisymmetric toroidal angle direction. In those cases, it is often convenient to have input grids which are only 2D, and allow the Z dimension to be specified independently, such as in the options file. The problem then is how to store 3D variables in the grid file?

Two representations are now supported for 3D variables:

1. A Fourier representation. If the size of the toroidal domain is not specified in the grid file (\(nz\) is not defined), then 3D fields are stored as Fourier components. In the Z dimension the coefficients must be stored as:

\[
[n = 0, n = 1(\text{real}), n = 1(\text{imag}), n = 2(\text{real}), n = 2(\text{imag}), \ldots]
\]

where \(n\) is the toroidal mode number. The size of the array must therefore be odd in the Z dimension, to contain a constant \((n = 0)\) component followed by real/imaginary pairs for the non-axisymmetric components.

If you are using IDL to create a grid file, there is a routine in \texttt{tools/idllib/bout3dvar.pro} for converting between BOUT++'s real and Fourier representation.

2. Real space, as values on grid points. If \(nz\) is set in the grid file, then 3D variables in the grid file must have size \(nx \times ny \times nz\). These are then read in directly into \texttt{Field3D} variables as required.

11.3 From EFIT files

An IDL code called “Hypnotoad” has been developed to create BOUT++ input files from R-Z equilibria. This can read EFIT ’g’ files, find flux surfaces, and calculate metric coefficients. The code is in \texttt{tools/tokamak_grids/gridgen}, and has its own manual under the \texttt{doc} subdirectory.

11.4 From ELITE and GATO files

Currently conversions exist for ELITE \(\texttt{.eqin}\) and GATO \(\texttt{dskgato}\) equilibrium files. Conversion of these into BOUT++ input grids is in two stages: In the first, both these input files are converted into a common NetCDF format which describes the Grad-Shafranov equilibrium. These intermediate files are then converted to BOUT++ grids using an interactive IDL script.

11.5 Generating equilibria

The directory \texttt{tokamak_grids/shifted_circle} contains IDL code to generate shifted circle (large aspect ratio) Grad-Shafranov equilibria.
Fig. 11.4: IDL routines and file formats used in taking output from different codes and converting into input to BOUT++.
11.6 Zoidberg grid generator

The Zoidberg grid generator creates inputs for the Flux Coordinate Independent (FCI) parallel transform (section Parallel Transforms). The domain is divided into a set of 2D grids in the X-Z coordinates, and the magnetic field is followed along the Y coordinate from each 2D grid to where it either intersects the forward and backward grid, or hits a boundary.

The simplest code which creates an output file is:

```python
import zoidberg

# Define the magnetic field
field = zoidberg.field.Slab()

# Define the grid points
grid = zoidberg.grid.rectangular_grid(10, 10, 10)

# Follow magnetic fields from each point
maps = zoidberg.make_maps(grid, field)

# Write everything to file
zoidberg.write_maps(grid, field, maps, gridfile="grid.fci.nc")
```

As in the above code, creating an output file consists of the following steps:

1. Define a magnetic field
2. Define the grid points. This can be broken down into:
   a) Define 2D “poloidal” grids
   b) Form a 3D grid by putting 2D grids together along the Y direction
3. Create maps from each 2D grid to its neighbours
4. Save grids, fields and maps to file

Each of these stages can be customised to handle more complicated magnetic fields, more complicated grids, and particular output formats. Details of the functionality available are described in sections below, and there are several examples in the examples/zoidberg directory.

11.6.1 Rectangular grids

An important input to Zoidberg is the size of the domain in Y, and whether the domain is periodic in Y. By default rectangular_grid makes a non-periodic rectangular box which is of length 10 in the Y direction. This means that there are boundaries at \( y = 0 \) and at \( y = 10 \). rectangular_grid puts the y slices at equally spaced intervals, and puts the first and last points half an interval away from boundaries in y. In this case with 10 points in y (second argument to rectangular_grid(nx, ny, nz)) the y locations are \( 0.5, 1.5, 2.5, \ldots, 9.5 \).

At each of these y locations rectangular_grid defines a rectangular 2D poloidal grid in the X-Z coordinates, by default with a length of 1 in each direction and centred on \( x = 0, z = 0 \). These 2D poloidal grids are then put together into a 3D Grid. This process can be customised by separating step 2 (the rectangular_grid call) into stages 2a) and 2b). For example, to create a periodic rectangular grid we could use the following:

```python
import numpy as np

# Create a 10x10 grid in X-Z with sides of length 1
poloidal_grid = zoidberg.poloidal_grid.RectangularPoloidalGrid(10, 10, 1.0, 1.0)

# Define the length of the domain in y
```
In the above code the length of the domain in the y direction needs to be given to Grid so that it knows where to put boundaries (if not periodic), or where to wrap the domain (if periodic). The array of y locations ycoords can be arbitrary, but note that finite difference methods (like FCI) work best if grid point spacing varies smoothly.

A more realistic example is creating a grid for a MAST tokamak equilibrium from a G-Eqdsk input file (this is in examples/zoidberg/tokamak.py):

```python
import numpy as np
import zoidberg

field = zoidberg.field.GEQDSK("g014220.00200")  # Read magnetic field

grid = zoidberg.grid.rectangular_grid(100, 10, 100, 1.5-0.1, # Range in R (max - min)
                                  2*np.pi, # Toroidal angle
                                  3., # Range in Z
                                  xcentre=(1.5+0.1)/2, # Middle of grid in R
                                  yperiodic=True)  # Periodic in toroidal angle

# Create the forward and backward maps
maps = zoidberg.make_maps(grid, field)

# Save to file
zoidberg.write_maps(grid, field, maps, gridfile="grid.fci.nc")

# Plot grid points and the points they map to in the forward direction
zoidberg.plot.plot_forward_map(grid, maps)
```

In the last example only one poloidal grid was created (a RectangularPoloidalGrid) and then re-used for each y slice. We can instead define a different grid for each y position. For example, to define a grid which expands along y (for some reason) we could do:

```python
ylength = 10.0
ycoords = np.linspace(0.0, ylength, 10, endpoint=False)
# Create a list of poloidal grids, one for each y location
poloidal_grids = [ RectangularPoloidalGrid(10, 10, 1.0 + y/10., 1.0 + y/10.)
                  for y in ycoords ]
# Create the 3D grid by putting together 2D poloidal grids
grid = zoidberg.grid.Grid(poloidal_grids, ycoords, ylength, yperiodic=True)
```

Note: Currently there is an assumption that the number of X and Z points is the same on every poloidal grid. The shape of the grid can however be completely different. The construction of a 3D Grid is the same in all cases, so for now we will concentrate on producing different poloidal grids.
11.6.2 More general grids

The FCI technique is not restricted to rectangular grids, and in particular Zoidberg can handle structured grids in an annulus with quite complicated shapes. The StructuredPoloidalGrid class handles quite general geometries, but still assumes that the grid is structured and logically rectangular. Currently it also assumes that the z index is periodic.

One way to create this grid is to define the grid points manually e.g.:

```python
import numpy as np
import zoidberg

# First argument is minor radius, second is angle
r,theta = np.meshgrid(np.linspace(1,2,10),
                      np.linspace(0,2*np.pi, 10),
                      indexing="ij")

R = r * np.sin(theta)
Z = r * np.cos(theta)

poloidal_grid = zoidberg.poloidal_grid.StructuredPoloidalGrid(R,Z)
```

For more complicated shapes than circles, Zoidberg comes with an elliptic grid generator which needs to be given only the inner and outer boundaries:

```python
import zoidberg

inner = zoidberg.rzline.shaped_line(R0=3.0, a=0.5,
                                    elong=1.0, triang=0.0, indent=1.0,
                                    n=50)

outer = zoidberg.rzline.shaped_line(R0=2.8, a=1.5,
                                    elong=1.0, triang=0.0, indent=0.2,
                                    n=50)

poloidal_grid = zoidberg.poloidal_grid.grid_elliptic(inner, outer,
                                                      100, 100, show=True)
```

which should produce the figure below:

11.6.3 Grids aligned to flux surfaces

The elliptic grid generator can be used to generate grids whose inner and/or outer boundaries align with magnetic flux surfaces. All it needs is two RZline objects as generated by zoidberg.rzline.shaped_line, one for the inner boundary and one for the outer boundary. RZline objects represent periodic lines in R-Z (X-Z coordinates), with interpolation using splines.

To create an RZline object for a flux surface we first need to find where the flux surface is. To do this we can use a Poincare plot: Start at a point and follow the magnetic field a number of times around the periodic y direction (e.g. toroidal angle). Every time the field line reaches a y location of interest, mark the position to build up a scattered set of points which all lie on the same flux surface.

At the moment this will not work correctly for slab geometries, but expects closed flux surfaces such as in a stellarator or tokamak. A simple test case is a straight stellarator:
Fig. 11.5: A grid produced by `grid_elliptic` from shaped inner and outer lines
import zoidberg
field = zoidberg.field.StraightStellarator(I_coil=0.4, yperiod=10)

By default StraightStellarator calculates the magnetic field due to four coils which spiral around the axis at a distance $r = 0.8$ in a classical stellarator configuration. The yperiod argument is the period in $y$ after which the coils return to their starting locations.

To visualise the Poincare plot for this stellarator field, pass the MagneticField object to zoidberg.plot.plot_poincare, together with start location(s) and periodicity information:

```python
zoidberg.plot.plot_poincare(field, 0.4, 0.0, 10.0)
```

which should produce the following figure:

![Fig. 11.6: Poincare map of straight stellarator showing a single flux surface. Each colour corresponds to a different x-z plane in the y direction.](image)

The inputs here are the starting location $(x, z) = (0.4, 0.0)$, and the periodicity in the $y$ direction (10.0). By default this will integrate from this given starting location 40 times (revs option) around the $y$ domain (0 to 10).

To create an RZline from these Poincare plots we need a list of points in order around the line. Since the points on a flux surface in a Poincare will not generally be in order we need to find the best fit i.e. the shortest path which passes through all the points without crossing itself. In general this is a known hard problem but fortunately in this case the nearest neighbour algorithm seems to be quite robust provided there are enough points.

An example of calculating a Poincare plot on a single $y$ slice ($y=0$) and producing an RZline is:
from zoidberg.fieldtracer import trace_poincare
rzcoord, ycoords = trace_poincare(field, 0.4, 0.0, 10.0,
    y_slices=[0])

R = rzcoord[:,0,0]
Z = rzcoord[:,0,1]

line = zoidberg.rzline.line_from_points(R, Z)
line.plot()

Note: Currently there is no checking that the line created is a good solution. The line could cross itself, but this has to be diagnosed manually at the moment. If the line is not a good approximation to the flux surface, increase the number of points by setting the revs keyword (y revolutions) in the trace_poincare call.

In general the points along this line are not evenly distributed, but tend to cluster together in some regions and have large gaps in others. The elliptic grid generator places grid points on the boundaries which are uniform in the index of the RZline it is given. Passing a very uneven set of points will therefore result in a poor quality mesh. To avoid this, define a new RZline by placing points at equal distances along the line:

line = line.equallySpaced()

The example zoidberg/straight-stellarator-curvilinear.py puts the above methods together to create a grid file for a straight stellarator.

Sections below now describe each part of Zoidberg in more detail. Further documentation of the API can be found in the docstrings and unit tests.

11.6.4 Magnetic fields

The magnetic field is represented by a MagneticField class, in zoidberg.field. Magnetic fields can be defined in either cylindrical or Cartesian coordinates:

- In Cartesian coordinates all (x,y,z) directions have the same units of length
- In cylindrical coordinates the y coordinate is assumed to be an angle, so that the distance in y is given by $ds = R dy$ where $R$ is the major radius.

Which coordinate is used is controlled by the Rfunc method, which should return the major radius if using a cylindrical coordinate system. Should return None for a Cartesian coordinate system (the default).

Several implementations inherit from MagneticField, and provide: Bxfunc, Byfunc, Bzfunc which give the components of the magnetic field in the x,y and z directions respectively. These should be in the same units (e.g. Tesla) for both Cartesian and cylindrical coordinates, but the way they are integrated changes depending on the coordinate system.

Using these functions the MagneticField class provides a Bmag method and field_direction method, which are called by the field line tracer routines (in zoidberg.field_tracer).
**Slabs and curved slabs**

The simplest magnetic field is a straight slab geometry:

```python
import zoidberg
field = zoidberg.field.Slab()
```

By default this has a magnetic field \( \mathbf{B} = (0, 1, 0.1 + x) \).

A variant is a curved slab, which is defined in cylindrical coordinates and has a given major radius (default 1):

```python
import zoidberg
field = zoidberg.field.CurvedSlab()
```

Note that this uses a large aspect-ratio approximation, so the major radius is constant across the domain (independent of \( x \)).

**Straight stellarator**

This is generated by four coils with alternating currents arranged on the edge of a circle, which spiral around the axis:

```python
import zoidberg
field = zoidberg.field.StraightStellarator()
```

**Note:** This requires Sympy to generate the magnetic field, so if unavailable an exception will be raised.

**G-Eqdsk files**

This format is commonly used for axisymmetric tokamak equilibria, for example output from EFIT equilibrium reconstruction. It consists of the poloidal flux psi, describing the magnetic field in \( R \) and \( Z \), with the toroidal magnetic field \( B_t \) given by a 1D function \( f(\psi) = R*B_t \) which depends only on \( \psi \):

```python
import zoidberg
field = zoidberg.field.GEQDSK("gfile.eqdsk")
```

**VMEC files**

The VMEC format describes 3D magnetic fields in toroidal geometry, but only includes closed flux surfaces:

```python
import zoidberg
field = zoidberg.field.VMEC("w7x.wout")
```
11.6.5 Plotting the magnetic field

Routines to plot the magnetic field are in `zoidberg.plot`. They include Poincare plots and 3D field line plots.

For example, to make a Poincare plot from a MAST equilibrium:

```python
import numpy as np
import zoidberg
field = zoidberg.field.GEQDSK("g014220.00200")
zoidberg.plot.plot_poincare(field, 1.4, 0.0, 2*np.pi, interactive=True)
```

This creates a flux surface starting at $R = 1.4$ and $Z = 0.0$. The fourth input ($2\pi$) is the periodicity in the $y$ direction. Since this magnetic field is symmetric in $y$ (toroidal angle), this parameter only affects the toroidal planes where the points are plotted.

The `interactive=True` argument to `plot_poincare` generates a new set of points for every click on the plot window.

11.6.6 Creating poloidal grids

The FCI technique is used for derivatives along the magnetic field (in $Y$), and doesn’t restrict the form of the grid in the $X$-$Z$ poloidal planes. A 3D grid created by Zoidberg is a collection of 2D planes (poloidal grids), connected together by interpolations along the magnetic field. To define a 3D grid we first need to define the 2D poloidal grids.

Two types of poloidal grids can currently be created: Rectangular grids, and curvilinear structured grids. All poloidal grids have the following methods:

- `getCoordinate()` which returns the real space $(R,Z)$ coordinates of a given $(x,z)$ index, or derivatives thereof
- `findIndex()` which returns the $(x,z)$ index of a given $(R,Z)$ coordinate which in general is floating point
- `metric()` which returns the 2D metric tensor
- `plot()` which plots the grid

Rectangular grids

To create a rectangular grid, pass the number of points and lengths in the $x$ and $z$ directions to `RectangularPoloidalGrid`:

```python
import zoidberg
rect = zoidberg.poloidal_grid.RectangularPoloidalGrid( nx, nz, Lx, Lz )
```

By default the middle of the rectangle is at $(R, Z) = (0, 0)$ but this can be changed with the `Rcentre` and `Zcentre` options.
Curvilinear structured grids

To create the structured curvilinear grids inner and outer lines are needed (two RZline objects). The shaped_line function creates RZline shapes with the following formula:

\[ R = R_0 - b + (a + b \cos(\theta) \cos(\theta + \delta \sin(\theta))) \]
\[ Z = (1 + \epsilon) a \sin(\theta) \]

where \(R_0\) is the major radius, \(a\) is the minor radius, \(\epsilon\) is the elongation (elong), \(\delta\) the triangularity (triang), and \(b\) the indentation (indent).
The majority of the existing analysis and post-processing code is written in Python. Routines to read BOUT++ output data, usually called “collect” because it collects data from multiple files, are also available in IDL, Matlab, Mathematica and Octave. All these post-processing routines are in the tools directory, with Python modules in tools/pylib. A summary of available routines is in Python routines; see below for how to install the requirements.

12.1 Python routines

12.1.1 Requirements

The Python tools provided with BOUT++ make heavy use of numpy and scipy, as well as matplotlib for the plotting routines. In order to read BOUT++ output in Python, you will need either netcdf4 or h5py.

While we try to ensure that the Python tools are compatible with both Python 2 and 3, we officially only support Python 3.

If you are developing BOUT++, you may also need Jinja2 to edit some of the generated code (see Field2D/Field3D Arithmetic Operators for more information).

You can install most of the required Python modules by running

```
$ pip3 install --user --requirement requirements.txt
```

in the directory where you have unpacked BOUT++. This will install supported versions of numpy, scipy, netcdf4, matplotlib and jinja2.

Note: If you have difficulties installing SciPy, please see their installation instructions.

12.1.2 Reading BOUT++ data

To read data from a BOUT++ simulation into Python, there is a collect routine. This gathers together the data from multiple processors, taking care of the correct layout.

```python
from boutdata.collect import collect

Ni = collect("Ni")  # Collect the variable "Ni"
```

The result is an up to 4D array, Ni in this case. The array is a BoutArray object: BoutArray is a wrapper class for Numpy’s ndarray which adds an ‘attributes’ member variable containing a dictionary of attributes. The array is ordered [t,x,y,z]:
so Ni would have 10 time slices, 1 point in x, 2 in y, and 3 in z. This should correspond to the grid size used in the simulation. Since the collected data is a NumPy array, all the useful routines in NumPy, SciPy and Matplotlib can be used for further analysis.

The attributes of the data give:

- the *bout_type* of the variable
  - {'Field3D_t', 'Field2D_t', 'scalar_t'} for time-evolving variables
  - {'Field3D', 'Field2D', 'scalar'} for time-independent variables
- its location, one of {'CELL_CENTRE', 'CELL_XLOW', 'CELL_YLOW', 'CELL_ZLOW'}. See *Staggered grids*.

Attributes can also be read using the *attributes* routine:

```python
>>> Ni.attributes("bout_type")
'Field3D_t'
```

```python
>>> Ni.attributes("location")
'CELL_CENTRE'
```

The result is a dictionary (map) of attribute name to attribute value.

If the data has less then 4 dimension, it can be checked with *dimension* what dimensions are available:

```python
from boutdata.collect import dimension
print(dimension("Ni"))
print(dimension("dx"))
```

The first will print as expected [t, x, y, z] - while the second will print [x, y] as dx is nether evolved in time, nor does it has a z dependency.

To access both the input options (in the BOUT.inp file) and output data, there is the *BoutData* class.

```python
>>> from boutdata.data import BoutData
>>> d = BoutData(path=".")
```

where the path is optional, and should point to the directory containing the BOUT.inp (input) and BOUT.dmp.* (output) files. This will return a dictionary with keys “path” (the given path to the data), “options” (the input options) and “outputs” (the output data). The tree of options can be printed:

```python
>>> print d["options"]
obturions
| - timestep = 50
| - myg = 0
| - nout = 50
| - mxg = 2
| - all
| | - bndry_all = neumann
```

(continues on next page)
and accessed as a tree of dictionaries:

```python
>>> print d["options"]["phisolver"]["fourth_order"]
true
```

Currently the values are either integers, floats, or strings, so in the above example "true" is a string, not a Boolean.

In a similar way the outputs are available as dictionary keys:

```python
>>> print d["outputs"]
ZMAX
rho_s
zperiod
BOUT_VERSION
...
>>> d["outputs"]["rho_s"]
0.00092165524660235405
```

There are several modules available for reading NetCDF files, so to provide a consistent interface, file access is wrapped into a class DataFile. This provides a simple interface for reading and writing files from any of the following modules: netCDF4, Scientific.IO.NetCDF, and scipy.io.netcdf. The DataFile class also provides allows access to HDF5 files through the same interface, using the h5py module. To open a file using DataFile:

```python
from boututils.datafile import DataFile
def = DataFile("file.nc")  # Open the file
def.read("variable")  # Read a variable from the file
def.close()  # Close the file
```

or similarly for an HDF5 file

```python
from boututils.datafile import DataFile
def = DataFile("file.hdf5")  # Open the file
def.read("variable")  # Read a variable from the file
def.close()  # Close the file
```

A more robust way to read from DataFiles is to use the context manager syntax:

```python
from boututils.datafile import DataFile
with DataFile("file.hdf5") as f:  # Open the file
    def.read("variable")  # Read a variable from the file
```

This way the DataFile is automatically closed at the end of the with block, even if there is an error in `f.read`. To list the variables in a file e.g.
>>> f = DataFile("test_io.grd.nc")
>>> print(f.list())
['f3d', 'f2d', 'nx', 'ny', 'rvar', 'ivar']

and to list the names of the dimensions

>>> print(f.dimensions("f3d"))
('x', 'y', 'z')

or to get the sizes of the dimensions

>>> print(f.size("f3d"))
[12, 12, 5]

or the dictionary of attributes

>>> print(f.attributes("f3d"))
{}

To read in all variables in a file into a dictionary there is the file_import

from boututils.file_import import file_import

grid = file_import("grid.nc")

12.2 Python analysis routines

The analysis and postprocessing routines are currently divided into two Python modules: boutdata, which contains BOUT++ specific things like collect, and boututils which contains more generic useful routines.

To plot data, a convenient wrapper around matplotlib is plotdata

from boutdata import collect
n = collect("n")  # Read data as NumPy array [t,x,y,z]

from boututils.plotdata import plotdata
plotdata(n[-1,:,0,:])

If given a 2D array as in the above example, plotdata produces a contour plot (using matplotlib pyplot.contourf) with colour bar. If given a 1D array then it will plot a line plot (using pyplot.plot).

It is sometimes useful to see an animation of a simulation. To do this there is showdata, which again is a wrapper around matplotlib:

from boutdata import collect
n = collect("n")  # Read data as NumPy array [t,x,y,z]

from boututils.showdata import showdata
showdata(n[:,:,0,:])

This always assumes that the first index is time and will be animated over. The above example animates the variable n in time, at each time point plotting a contour plot in x and z dimensions. The colour range is kept constant by default. If a 2D array is given to showdata then a line plot will be drawn at each time, with the scale being kept constant.
12.3 Reading BOUT++ output into IDL

There are several routines provided for reading data from BOUT++ output into IDL. In the directory containing the BOUT++ output files (usually `data/`), you can list the variables available using

```
IDL> print, file_list("BOUT.dmp.0.nc")
Ajpar Apar BOUT_VERSION MXG MXSUB MYG MYSUB MZ NXPE NYPE Ni Ni0 Ni_x Te0 Te_x Ti0 Ti_x ZMAX ZMIN iteration jpar phi rho rho_s t_array wci
```

The `file_list` procedure just returns an array, listing all the variables in a given file.

One thing new users can find confusing is that different simulations may have very different outputs. This is because **BOUT++ is not a single physics model**: the variables evolved and written to file are determined by the model, and will be very different between (for example) full MHD and reduced Braginskii models. There are however some variables which all BOUT++ output files contain:

- **BOUT_VERSION**, which gives the version number of BOUT++ which produced the file. This is mainly to help output processing codes handle changes to the output file format. For example, BOUT++ version 0.30 introduced 2D domain decomposition which needs to be handled when collecting data.
- **MXG, MYG**. These are the sizes of the X and Y guard cells
- **MXSUB**, the number of X grid points in each processor. This does not include the guard cells, so the total X size of each field will be `MXSUB + 2*MXG`.
- **MYSUB**, the number of Y grid points per processor (like MXSUB)
- **MZ**, the number of Z points
- **NXPE, NYPE**, the number of processors in the X and Y directions. `NXPE * MXSUB + 2*MXG = NX, NYPE * MYSUB = NY`
- **ZMIN, ZMAX**, the range of Z in fractions of $2\pi$.
- **iteration**, the last timestep in the file
- **t_array**, an array of times

Most of these - particularly those concerned with grid size and processor layout - are used by post-processing routines such as `collect`, and are seldom needed directly. To read a single variable from a file, there is the `file_read` function:

```
IDL> wci = file_read("BOUT.dmp.0.nc", "wci")
IDL> print, wci
  9.58000e+06
```

To read in all the variables in a file into a structure, use the `file_import` function:

```
IDL> d = file_import("BOUT.dmp.0.nc")
IDL> print, d.wci
  9.58000e+06
```

This is often used to read in the entire grid file at once. Doing this for output data files can take a long time and use a lot of memory.

Reading from individual files is fine for scalar quantities and time arrays, but reading arrays which are spread across processors (i.e. evolving variables) is tedious to do manually. Instead, there is the `collect` function to automate this:

```
IDL> ni = collect(var="ni")
Variable 'ni' not found
```

(continues on next page)
Variables are case-sensitive: Using 'Ni'

Reading from ./BOUT.dmp.0.nc: [0-35][2-6] -> [0-35][0-4]

This function takes care of the case, so that reading “ni” is automatically corrected to “Ni”. The result is a 4D variable:

```
IDL> help, ni
NI FLOAT = Array[36, 5, 64, 400]
```

with the indices [X, Y, Z, T]. Note that in the output files, these variables are stored in [T, X, Y, Z] format instead but this is changed by `collect`. Sometimes you don’t want to read in the entire array (which may be very large). To read in only a subset, there are several optional keywords with [min,max] ranges:

```
IDL> ni = collect(var="Ni", xind=[10,20], yind=[2,2], zind=[0,31],
tind=[300,399])
```

```
IDL> help, ni
NI FLOAT = Array[11, 1, 32, 100]
```

### 12.4 Summary of IDL file routines

Functions `file_` can currently only read/write NetCDF files. HDF5 is not supported yet.

Open a NetCDF file:

```
handle = file_open("filename", /write, /create)
```

Array of variable names:

```
list = file_list(handle)
list = file_list("filename")
```

Number of dimensions:

```
d = file_ndims(handle, "variable")
d = file_ndims("filename", "variable")
```

Read a variable from file. Inds = [xmin, xmax, ymin, ymax, ...]

```
data = file_read(handle, "variable", inds=inds)
data = file_read("filename", "variable", inds=inds)
```

Write a variable to file. For NetCDF it tries to match up dimensions, and defines new dimensions when needed

```
status = file_write(handle, "variable", data)
```

Close a file after use

```
file_close, handle
```

To read in all the data in a file into a structure:

```
data = file_import("filename")
```
status = file_export("filename", data)

12.5 IDL analysis routines

Now that the BOUT++ results have been read into IDL, all the usual analysis and plotting routines can be used. In addition, there are many useful routines included in the idllib subdirectory. There is a README file which describes what each of these routines, but some of the most useful ones are listed here. All these examples assume there is a variable P which has been read into IDL as a 4D [x,y,z,t] variable:

- `fft_deriv` and `fft_integrate` which differentiate and integrate periodic functions.
- `get_integer`, `get_float`, and `get_yesno` request integers, floats and a yes/no answer from the user respectively.
- `showdata` animates 1 or 2-dimensional variables. Useful for quickly displaying results in different ways. This is useful for taking a quick look at the data, but can also produce bitmap outputs for turning into a movie for presentation. To show an animated surface plot at a particular poloidal location (32 here):

  IDL> showdata, p[*,32,*,*]

To turn this into a contour plot,

  IDL> showdata, p[*,32,*,*], /cont

To show a slice through this at a particular toroidal location (0 here):

  IDL> showdata, p[*,32,0,*]

There are a few other options, and ways to show data using this code; see the README file, or comments in showdata.pro. Instead of plotting to screen, showdata can produce a series of numbered bitmap images by using the `bmp` option

  IDL> showdata, p[*,32,*,*], /cont, bmp="result_

which will produce images called `result_0000.bmp`, `result_0001.bmp` and so on. Note that the plotting should not be obscured or minimised, since this works by plotting to screen, then grabbing an image of the resulting plot.

- `moment_xyzt` takes a 4D variable (such as those from `collect`), and calculates RMS, DC and AC components in the Z direction.
- `safe_colors` A general routine for IDL which arranges the color table so that colors are numbered 1 (black), 2 (red), 3 (green), 4 (blue). Useful for plotting, and used by many other routines in this library.

There are many other useful routines in the idllib directory. See the idllib/README file for a short description of each one.
### 12.6 Matlab routines

These are Matlab routines for collecting data, showing animation and performing some basic analysis. To use these routines, either you may copy these routines (from `tools/matlablib`) directly to your present working directory or a path to `tools/matlablib` should be added before analysis.

```matlab
addpath '/full_path_BOUT_directory>/tools/matlablib/
```

Now, the first routine to collect data and import it to Matlab for further analysis is

```matlab
var = import_dmp(path,var_name);
```

Here, `path` is the path where the output data in netcdf format has been dumped. `var_name` is the name of variable which user want to load for further analysis. For example, to load “P” variable from present working directory:

```matlab
P = import_dmp('.', 'P');
```

Variable “P” can be any of [X,Y,Z,T]/[X,Y,Z]/[X,Y]/Constant formats. If we are going to Import a large data set with [X,Y,Z,T] format. Normally such data files are of very big size and Matlab goes out of memory/ or may take too much time to load data for all time steps. To resolve this limitation of above routine `import_dmp`, another routine `import_data_netcdf` is being provided. It serves all purposes the routine `import_dmp` does but also gives user freedom to import data at only few/specific time steps.

```matlab
var = import_data_netcdf(path,var_name,nt,ntsp);
```

Here, `path` and `var_name` are same variables as described before. `nt` is the number of time steps user wish to load data. `ntsp` is the steps at which one wish to write data of of total simulation times the data written.

```matlab
P = import_data_netcdf('.', 'P',5,100);
```

Variable “P” has been imported from present working directory for 5 time steps. As the original netcdf data contains time information of 500 steps (assume NT=500 in BOUT++ simulations), user will pick only 5 time steps at steps of `ntsp` i.e. 100 here. Details of other Matlab routines provided with BOUT++ package can be looked into to README.txt of `tools/matlablib` directory. The Matlab users can develop their own routines using `*ncread, ncinfo, ncrewrite, ncdisp, netcdf etc.*` functions provided in Matlab package.

### 12.7 Mathematica routines

A package to read BOUT++ output data into Mathematica is in `tools/mathematicalib`. To read data into Mathematica, first add this directory to Mathematica’s path by putting

```mathematica
AppendTo[$Path,"/full/path/to/BOUT/tools/mathematicalib"]
```

in your Mathematica startup file (usually `$HOME/.Mathematica/Kernel/init.m`). To use the package, call

```mathematica
Import["BoutCollect.m"]
```

from inside Mathematica. Then you can use e.g.

```mathematica
f=BoutCollect[variable, path->"data"]
```

or
'bc' is a shorthand for 'BoutCollect'. All options supported by the Python collect() function are included, though Info does nothing yet.

### 12.8 Octave routines

There is minimal support for reading data into Octave, which has been tested on Octave 3.2. It requires the octcdf library to access NetCDF files.

```plaintext
f = bcollect()  # optional path argument is "." by default
f = bsetxrange(f, 1, 10)  # Set ranges
# Same for y, z, and t (NOTE: indexing from 1!)
u = bread(f, "U")  # Finally read the variable
```
Reproducibility and Provenance Tracking

To help with reproducibility of simulations and provenance tracking of data, BOUT++ saves some metadata into output files.

**Note:** Most of this is only saved when using NetCDF for output; HDF5 output has a known bug writing string variables and will be removed in v5.

<table>
<thead>
<tr>
<th>File attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOUT_REVISION</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>run_id</td>
</tr>
<tr>
<td>run_restart_from</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Grid-related variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid_id</td>
</tr>
<tr>
<td>hypnoaad_version</td>
</tr>
<tr>
<td>hypnoaad_git_hash</td>
</tr>
<tr>
<td>hypnoaad_git_diff</td>
</tr>
<tr>
<td>hypnoaad_geqdsk_filename</td>
</tr>
</tbody>
</table>
14.1 Installing

Installing boutcore can be tricky. Ideally it should be just

```
./configure --enable-shared
make -j 4 python
```

but getting all the dependencies can be difficult. `make python` creates the python3 module.

If problems arise, it might be worth checking a copy of the bout module out, to reduce the risk of causing issues with the old bout installation. This is especially true if you are trying to run boutcore not on compute nodes of a super computer but rather on post-processing/login/... nodes.

To use boutcore on the login node, a self compiled version of mpi may be required, as the provided one may be only for the compute nodes. Further, numpy header files are required, therefore numpy needs to be compiled as well. Further, the header files need to be exposed to the boutcore cython compilation, e.g. by adding them to `_boutcore_build/boutcore.pyx.in`. It seems both `NUMPY/numpy/core/include` and `NUMPY/build/src.linux-x86_64-2.7/numpy/core/include/numpy` need to be added, where `NUMPY` is the path of the numpy directory. For running boutcore on the post processing nodes, fftw3 needs to be compiled as well, if certain fftw routines are used. Note, fftw needs to be configured with `--enable-shared`.

After installing mpi e.g. in `~/local/mpich`, bout needs to be configured with something like:

```
./configure --enable-shared MPICC=~/local/mpich/bin/mpicc MPICXX=~/local/mpich/bin/mpicxx
--with-fftw=~/local/fftw/
```

`--enable-shared` is required, so that pvode etc. is compiles as position independent code.

If you are running fedora - you can install pre-build binaries:

```
sudo dnf copr enable davidsch/bout
sudo dnf install python3-bout++-mpich
module load mpi/mpich-$\$(arch)
```
14.2 Purpose

The boutcore module exposes (part) of the BOUT++ C++ library to python. It allows to calculate e.g. BOUT++ derivatives in python.

14.3 State

Field3D and Field2D are working. If other fields are needed, please open an issue. Fields can be accessed directly using the [] operators, and give a list of slice objects. The get all data, f3d.getAll() is equivalent to f3d[:,:,] and returns a numpy array. This array can be addressed with e.g. [] operators, and then the field can be set again with f3d.setAll(numpyarray). It is also possible to set a part of an Field3D with the [] operators. Addition, multiplication etc. are all available. The derivatives should all be working, if find a missing one, please open an issue. Vectors are not exposed yet.

14.4 Functions

14.5 Examples

Some trivial post processing:

```python
import boutcore
import numpy as np
args="-d data -f BOUT.settings -o BOUT.post".split(" ")
boutcore.init(args)
dens=boutcore.Field3D.fromCollect("n",path="data")
temp=boutcore.Field3D.fromCollect("T",path="data")
pres=dens*temp
dpdz=boutcore.DDZ(pres,outloc="CELL_ZLOW")
```

A simple MMS test:

```python
import boutcore
import numpy as np
boutcore.init("-d data -f BOUT.settings -o BOUT.post")
for nz in [64,128,256]:
    boutcore.setOption("meshz:nz","%d"%nz)
    mesh=boutcore.Mesh(OptionSection="meshz")
    f=boutcore.create3D("sin(z)",mesh)
    sim=boutcore.DDZ(f)
    ana=boutcore.create3D("cos(z)",mesh)
    err=sim-ana
    err=boutcore.max(boutcore.abs(err))
    errors.append(err)
```

A real example - unstagger data:

```python
import boutcore
boutcore.init("-d data -f BOUT.settings -o BOUT.post")
# uses location from dump - is already staggered
```

(continues on next page)
upar=boutcore.Field3D.fromCollect("Upar")
upar=boutcore.interp_to(upar,"CELL_CENTRE")
# convert to numpy array
upar=upar.getAll()

A real example - check derivative contributions:

```python
#!/usr/bin/env python
from boutcore import *
import numpy as np
from netCDF4 import Dataset
import sys
if len(sys.argv)> 1:
    path=sys.argv[1]
else:
    path="data"
times=collect("t_array",path=path)
boutcore.init("-d data -f BOUT.settings -o BOUT.post")
with Dataset(path+"/vort.nc", format='NETCDF4') as outdmp:
    phiSolver=Laplacian()
    phi=Field3D.fromCollect("n",path=path,tind=0,info=False)
    zeros=phi.getAll()*0
    phi.setAll(zeros)
    outdmp.createDimension('x',zeros.shape[0])
    outdmp.createDimension('y',zeros.shape[1])
    outdmp.createDimension('z',zeros.shape[2])
    outdmp.createDimension('t',None)
t_array_=outdmp.createVariable('t_array','f4',('t',))
t_array_[:]=times
    ExB = outdmp.createVariable('ExB','f4',('t','x','y','z'))
    par_adv = outdmp.createVariable('par_adv','f4',('t','x','y','z'))
def setXGuards(phi,phi_arr):
    for z in range(tmp.shape[2]):
        phi[0,:,z]=phi_arr
        phi[1,:,z]=phi_arr
        phi[-2,:,z]=phi_arr
        phi[-1,:,z]=phi_arr
    with open(path+"/equilibrium/phi_eq.dat","rb") as inf:
        phi_arr=np.fromfile(inf,dtype=np.double)
bm="BRACKET_ARAKAWA_OLD"
    for tind in range(len(times)):
        vort = Field3D.fromCollect("vort",path=path,tind=tind,info=False)
        U = Field3D.fromCollect("U",path=path,tind=tind,info=False)
        setXGuards(phi,phi_arr)
        phi=phiSolver.solve(vort,phi)
        ExB[tind,:,:,:]=(-bracket(phi, vort, bm, "CELL_CENTRE")).getAll()
        par_adv[tind,:,:,:]=(- Vpar_Grad_par(U, vort)).getAll()
```
14.6 Functions - undocumented

14.7 Functions - special and inherited
15.1 Options

BOUT++ can be compiled with several different time-integration solvers, and at minimum should have Runge-Kutta (RK4) and PVODE (BDF/Adams) solvers available.

The solver library used is set using the `solver:type` option, so either in BOUT.inp:

```
[solver]
type = rk4  # Set the solver to use
```

or on the command line by adding `solver:type=pvode` for example:

```
mpirun -np 4 ./2fluid solver:type=rk4
```

**NB:** Make sure there are no spaces around the “=” sign: `solver:type =pvode` won’t work (probably). Table Table 15.1 gives a list of time integration solvers, along with any compile-time options needed to make the solver available.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Compile options</th>
</tr>
</thead>
<tbody>
<tr>
<td>euler</td>
<td>Euler explicit method (example only)</td>
<td>Always available</td>
</tr>
<tr>
<td>rk4</td>
<td>Runge-Kutta 4th-order explicit method</td>
<td>Always available</td>
</tr>
<tr>
<td>rkgeneric</td>
<td>Generic Runge Kutta explicit methods</td>
<td>Always available</td>
</tr>
<tr>
<td>karniadakis</td>
<td>Karniadakis explicit method</td>
<td>Always available</td>
</tr>
<tr>
<td>rk3ssp</td>
<td>3rd-order Strong Stability Preserving</td>
<td>Always available</td>
</tr>
<tr>
<td>splitrk</td>
<td>Split RK3-SSP and RK-Legendre</td>
<td>Always available</td>
</tr>
<tr>
<td>pvode</td>
<td>1998 PVODE with BDF method</td>
<td>Always available</td>
</tr>
<tr>
<td>cvode</td>
<td>SUNDIALS CVODE, BDF and Adams methods</td>
<td>–with-cvode</td>
</tr>
<tr>
<td>ida</td>
<td>SUNDIALS IDA, DAE solver</td>
<td>–with-ida</td>
</tr>
<tr>
<td>arkode</td>
<td>SUNDIALS ARKODE IMEX solver</td>
<td>–with-arkode</td>
</tr>
<tr>
<td>petsc</td>
<td>PETSc TS methods</td>
<td>–with-petsc</td>
</tr>
<tr>
<td>imexbdf2</td>
<td>IMEX-BDF2 scheme</td>
<td>–with-petsc</td>
</tr>
<tr>
<td>beuler / snes</td>
<td>Backward Euler with SNES solvers</td>
<td>–with-petsc</td>
</tr>
</tbody>
</table>

Each solver can have its own settings which work in slightly different ways, but some common settings and which solvers they are used in are given in table Table 15.2.
### Table 15.2: Time integration solver options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Solvers used</th>
</tr>
</thead>
<tbody>
<tr>
<td>atol</td>
<td>Absolute tolerance</td>
<td>rk4, pvode, cvode, ida, imexbdf2, beuler</td>
</tr>
<tr>
<td>rtol</td>
<td>Relative tolerance</td>
<td>rk4, pvode, cvode, ida, imexbdf2, beuler</td>
</tr>
<tr>
<td>mxstep</td>
<td>Maximum internal steps per output step</td>
<td>rk4, imexbdf2</td>
</tr>
<tr>
<td>max_timestep</td>
<td>Maximum timestep</td>
<td>rk4, cvode</td>
</tr>
<tr>
<td>timestep</td>
<td>Starting timestep</td>
<td>rk4, euler, imexbdf2, beuler</td>
</tr>
<tr>
<td>adaptive</td>
<td>Adapt timestep? (Y/N)</td>
<td>rk4, imexbdf2</td>
</tr>
<tr>
<td>use_precon</td>
<td>Use a preconditioner? (Y/N)</td>
<td>pvode, cvode, ida, imexbdf2</td>
</tr>
<tr>
<td>mudq, mldq</td>
<td>BBD preconditioner settings</td>
<td>pvode, cvode, ida</td>
</tr>
<tr>
<td>mukeep, mlkeep</td>
<td></td>
<td></td>
</tr>
<tr>
<td>maxl</td>
<td>Maximum number of linear iterations</td>
<td>cvode, imexbdf2</td>
</tr>
<tr>
<td>max_nonlinear_iterations</td>
<td>Maximum number of nonlinear iterations</td>
<td>cvode, imexbdf2, beuler</td>
</tr>
<tr>
<td>use_jacobian</td>
<td>Use user-supplied Jacobian? (Y/N)</td>
<td>cvode</td>
</tr>
<tr>
<td>adams_moulton</td>
<td>Use Adams-Moulton method rather than BDF</td>
<td>cvode</td>
</tr>
<tr>
<td>diagnose</td>
<td>Collect and print additional diagnostics</td>
<td>cvode, imexbdf2, beuler</td>
</tr>
</tbody>
</table>

The most commonly changed options are the absolute and relative solver tolerances, ATOL and RTOL which should be varied to check convergence.

### 15.2 CVODE

The most commonly used time integration solver is CVODE, or its older version PVODE. CVODE has several advantages over PVODE, including better support for preconditioning and diagnostics.

Enabling diagnostics output using `solver:diagnose=true` will print a set of outputs for each timestep similar to:

```
CVODE: nsteps 51, nfevals 69, nniters 65, npevals 126, nliters 79
    -> Newton iterations per step:  1.274510e+00
    -> Linear iterations per Newton iteration: 1.215385e+00
    -> Preconditioner evaluations per Newton: 1.938462e+00
    -> Last step size:  1.026792e+00, order: 5
    -> Local error fails: 0, nonlinear convergence fails: 0
    -> Stability limit order reductions: 0

1.000e+01  149  2.07e+01  78.3  0.0  10.0  0.9  10.8
```

When diagnosing slow performance, key quantities to look for are nonlinear convergence failures, and the number of linear iterations per Newton iteration. A large number of failures, and close to 5 linear iterations per Newton iteration are a sign that the linear solver is not converging quickly enough, and hitting the default limit of 5 iterations. This limit can be modified using the `solver:maxl` setting. Giving it a large value e.g. `solver:maxl=1000` will show how many iterations are needed to solve the linear system. If the number of iterations becomes large, this may be an indication that the system is poorly conditioned, and a preconditioner might help improve performance. See *Preconditioning*.

CVODE can set constraints to keep some quantities positive, non-negative, negative or non-positive. These constraints can be activated by setting the option `solver:apply_positivity_constraints=true`, and then in the section for
a certain variable (e.g. \([n]\)), setting the option \texttt{positivity_constraint} to one of \texttt{positive}, \texttt{non_negative}, \texttt{negative}, or \texttt{non_positive}.

15.3 IMEX-BDF2

This is an IMplicit-EXplicit time integration solver, which allows the evolving function to be split into two parts: one which has relatively long timescales and can be integrated using explicit methods, and a part which has short timescales and must be integrated implicitly. The order of accuracy is variable (up to 4th-order currently), and an adaptive timestep can be used.

To use the IMEX-BDF2 solver, set the solver type to \texttt{imexbdf2}, e.g. on the command-line add \texttt{solver:type=imexbdf2} or in the options file:

```
[solver]
type = imexbdf2
```

The order of the method is set to 2 by default, but can be increased up to a maximum of 4:

```
[solver]
type = imexbdf2
maxOrder = 3
```

This is a multistep method, so the state from previous steps are used to construct the next one. This means that at the start, when there are no previous steps, the order is limited to 1 (backwards Euler method). Similarly, the second step is limited to order 2, and so on. At the moment the order is not adapted, so just increases until reaching \texttt{maxOrder}.

At each step the explicit (non-stiff) part of the function is called, and combined with previous timestep values. The implicit part of the function is then solved using PETSc’s SNES, which consists of a nonlinear solver (usually modified Newton iteration), each iteration of which requires a linear solve (usually GMRES). Settings which affect this implicit part of the solve are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atol</td>
<td>1e-16</td>
<td>Absolute tolerance on SNES solver</td>
</tr>
<tr>
<td>rtol</td>
<td>1e-10</td>
<td>Relative tolerance on SNES solver</td>
</tr>
<tr>
<td>max_nonlinear</td>
<td>5</td>
<td>Maximum number of nonlinear iterations If adaptive timestepping is used then failure will cause timestep reduction</td>
</tr>
<tr>
<td>maxl</td>
<td>20</td>
<td>Maximum number of linear iterations If adaptive, failure will cause timestep reduction</td>
</tr>
<tr>
<td>predictor</td>
<td>1</td>
<td>Starting guess for the nonlinear solve Specifies order of extrapolating polynomial</td>
</tr>
<tr>
<td>use_precon</td>
<td>false</td>
<td>Use user-supplied preconditioner?</td>
</tr>
<tr>
<td>matrix_free</td>
<td>true</td>
<td>Use Jacobian-free methods? If false, calculates the Jacobian matrix using finite difference</td>
</tr>
<tr>
<td>use_coloring</td>
<td>true</td>
<td>If not matrix free, use coloring to speed up calculation of the Jacobian</td>
</tr>
</tbody>
</table>

Note that the SNES tolerances \texttt{atol} and \texttt{rtol} are set very conservatively by default. More reasonable values might be 1e-10 and 1e-5, but this must be explicitly asked for in the input options.

The predictor extrapolates from previous timesteps to get a starting estimate for the value at the next timestep. This estimate is then used to initialise the SNES nonlinear solve. The value is the order of the extrapolating polynomial, so 1 (the default) is a linear extrapolation from the last two steps, 0 is the same as the last step. A value of -1 uses the explicit update to the state as the starting guess, i.e. assuming that the implicit part of the problem is small. This is usually not a good guess.

To diagnose what is happening in the time integration, for example to see why it is failing to converge or why timesteps are small, there are two settings which can be set to \texttt{true} to enable:
• `diagnose` outputs a summary at each output time, similar to CVODE. This contains information like the last timestep, average number of iterations and number of convergence failures.

• `verbose` prints information at every internal step, with more information on the values used to modify timesteps, and the reasons for solver failures.

By default adaptive timestepping is turned on, using several factors to modify the timestep:

1. If the nonlinear solver (SNES) fails to converge, either because it diverges or exceeds the iteration limits `max_nonlinear_its` or `maxl`. Reduces the timestep by 2 and tries again, giving up after 10 failures.

2. Every `nadapt` internal timesteps (default 4), the error is checked by taking the timestep twice: Once with the current order of accuracy, and once with one order of accuracy lower. The difference between the solutions is then used to estimate the timestep required to achieve the required tolerances. If this is much larger or smaller than the current timestep, then the timestep is modified.

3. The timestep is kept within user-specified maximum and minimum ranges.

The options which control this behaviour are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adaptive</td>
<td>true</td>
<td>Turns on adaptive timestepping</td>
</tr>
<tr>
<td>timestep</td>
<td>output</td>
<td>If adaptive sets the starting timestep. If not adaptive, timestep fixed at this value</td>
</tr>
<tr>
<td>dtMin</td>
<td>1e-10</td>
<td>Minimum timestep</td>
</tr>
<tr>
<td>dtMax</td>
<td>output</td>
<td>Maximum timestep</td>
</tr>
<tr>
<td>mxstep</td>
<td>1e5</td>
<td>Maximum number of internal steps between outputs</td>
</tr>
<tr>
<td>adaptRtol</td>
<td>1e-3</td>
<td>Target relative tolerance for adaptive timestep</td>
</tr>
<tr>
<td>scale-Down</td>
<td>1.0</td>
<td>Timestep scale factor below which the timestep is modified. By default the timestep is always reduced</td>
</tr>
<tr>
<td>scale-Up</td>
<td>1.5</td>
<td>Minimum timestep scale factor based on <code>adaptRtol</code> above which the timestep will be modified. Currently the timestep increase is limited to 25%</td>
</tr>
</tbody>
</table>

### 15.4 Split-RK

The `splitrk` solver type uses Strang splitting to combine two explicit Runge Kutta schemes:

1. **2nd order Runge-Kutta-Legendre method** for the diffusion (parabolic) part. These schemes use multiple stages to increase stability, rather than accuracy; this is always 2nd order, but the stable timestep for diffusion problems increases as the square of the number of stages. The number of stages is an input option, and can be arbitrarily large.


Each timestep consists of

1. A half timestep of the diffusion part
2. A full timestep of the advection part
3. A half timestep of the diffusion part

Options to control the behaviour of the solver are:
And the adaptive timestepping options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adaptive</td>
<td>true</td>
<td>Turn on adaptive timestepping</td>
</tr>
<tr>
<td>atol</td>
<td>1e-10</td>
<td>Absolute tolerance</td>
</tr>
<tr>
<td>rtol</td>
<td>1e-5</td>
<td>Relative tolerance</td>
</tr>
<tr>
<td>max_timestep</td>
<td>output</td>
<td>Maximum internal timestep</td>
</tr>
<tr>
<td>max_timestep_change</td>
<td>2</td>
<td>Maximum factor by which the timestep can be changed at each step</td>
</tr>
<tr>
<td>mxstep</td>
<td>1000</td>
<td>Maximum number of internal steps before output</td>
</tr>
<tr>
<td>adapt_period</td>
<td>1</td>
<td>Number of internal steps between tolerance checks</td>
</tr>
</tbody>
</table>

### 15.5 Backward Euler - SNES

The beuler or snes solver type (either name can be used) is intended mainly for solving steady-state problems, so integrates in time using a stable but low accuracy method (Backward Euler). It uses PETSc’s SNES solvers to solve the nonlinear system at each timestep, and adjusts the internal timestep to keep the number of SNES iterations within a given range.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_nonlinear_it</td>
<td>50</td>
<td>If exceeded, solve restarts with timestep / 2</td>
</tr>
<tr>
<td>upper_its</td>
<td>80%</td>
<td>If exceeded, next timestep reduced by 10%</td>
</tr>
<tr>
<td>lower_its</td>
<td>50%</td>
<td>If under this, next timestep increased by 10%</td>
</tr>
</tbody>
</table>

The predictor is linear extrapolation from the last two timesteps. It seems to be effective, but can be disabled by setting predictor = false.

The SNES type can be set through PETSc command-line options, or in the BOUT++ options as setting snes_type. Good choices for unpreconditioned problems seem to be anderson (the default) and qn (quasinewton).

### 15.6 ODE integration

The Solver class can be used to solve systems of ODEs inside a physics model: Multiple Solver objects can exist besides the main one used for time integration. Example code is in examples/test-integrate.

To use this feature, systems of ODEs must be represented by a class derived from PhysicsModel.

```c++
class MyFunction : public PhysicsModel {
public:
    int init(bool restarting) {
        // Initialise ODE
        // Add variables to solver as usual
        solver->add(result, "result");
    }
}
```

(continues on next page)
To solve this ODE, create a new `Solver` object:

```cpp
Solver* ode = Solver::create(Options::getRoot()->getSection("ode"));
```

This will look in the section `[ode]` in the options file. **Important:** To prevent this solver overwriting the main restart files with its own restart files, either disable restart files:

```cpp
[ode]
enablerestart = false
```

or specify a different directory to put the restart files:

```cpp
[ode]
restartdir = ode  # Restart files ode/BOUT_restart.0.nc, ...
```

Create a model object, and pass it to the solver:

```cpp
MyFunction* model = new MyFunction();
ode->setModel(model);
```

Finally tell the solver to perform the integration:

```cpp
ode->solve(5, 0.1);
```

The first argument is the number of steps to take, and the second is the size of each step. These can also be specified in the options, so calling

```cpp
ode->solve();
```

will cause ode to look in the input for `nout` and `timestep` options:

```cpp
[ode]
nout = 5
timestep = 0.1
```

Finally, delete the model and solver when finished:

```cpp
delete model;
delete solver;
```

**Note:** If an ODE needs to be solved multiple times, at the moment it is recommended to delete the solver, and create a new one each time.
15.7 Preconditioning

At every time step, an implicit scheme such as BDF has to solve a non-linear problem to find the next solution. This is usually done using Newton’s method, each step of which involves solving a linear (matrix) problem. For $N$ evolving variables is an $N \times N$ matrix and so can be very large. By default matrix-free methods are used, in which the Jacobian $J$ is approximated by finite differences (see next subsection), and so this matrix never needs to be explicitly calculated. Finding a solution to this matrix can still be difficult, particularly as $\delta t$ gets large compared with some time-scales in the system (i.e. a stiff problem).

A preconditioner is a function which quickly finds an approximate solution to this matrix, speeding up convergence to a solution. A preconditioner does not need to include all the terms in the problem being solved, as the preconditioner only affects the convergence rate and not the final solution. A good preconditioner can therefore concentrate on solving the parts of the problem with the fastest time-scales.

A simple example\(^1\) is a coupled wave equation, solved in the test-precon example code:

$$\frac{\partial u}{\partial t} = \partial_1 v \quad \frac{\partial v}{\partial t} = \partial_1 u$$

First, calculate the Jacobian of this set of equations by taking partial derivatives of the time-derivatives with respect to each of the evolving variables

$$J = \left( \begin{array}{cc} \frac{\partial}{\partial u} \frac{\partial u}{\partial t} & \frac{\partial}{\partial v} \frac{\partial u}{\partial t} \\ \frac{\partial}{\partial u} \frac{\partial v}{\partial t} & \frac{\partial}{\partial v} \frac{\partial v}{\partial t} \end{array} \right) = \left( \begin{array}{cc} 0 & \partial_1 \\ \partial_1 & 0 \end{array} \right)$$

In this case $\frac{\partial u}{\partial t}$ doesn’t depend on $u$ nor $\frac{\partial v}{\partial t}$ on $v$, so the diagonal is empty. Since the equations are linear, the Jacobian doesn’t depend on $u$ or $v$ and so

$$\frac{\partial}{\partial t} \begin{pmatrix} u \\ v \end{pmatrix} = J \begin{pmatrix} u \\ v \end{pmatrix}$$

In general for non-linear functions $J$ gives the change in time-derivatives in response to changes in the state variables $u$ and $v$.

In implicit time stepping, the preconditioner needs to solve an equation

$$I - \gamma J$$

where $I$ is the identity matrix, and $\gamma$ depends on the time step and method (e.g. $\gamma = \delta t$ for backwards Euler method). For the simple wave equation problem, this is

$$I - \gamma J = \begin{pmatrix} 1 & -\gamma \partial_1 \\ -\gamma \partial_1 & 1 \end{pmatrix}$$

This matrix can be block inverted using Schur factorisation\(^2\)

$$\begin{pmatrix} E & U \\ L & D \end{pmatrix}^{-1} = \begin{pmatrix} I & -E^{-1}U \\ 0 & I \end{pmatrix}(E^{-1}0 \begin{pmatrix} 0 & -1 \\ \gamma \partial_1 & 1 \end{pmatrix})$$

where $P_{Schur} = D - LE^{-1}U$ Using this, the wave problem becomes:

$$\begin{pmatrix} 1 & -\gamma \partial_1 \\ -\gamma \partial_1 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & \gamma \partial_1 \\ 0 & (1 - \gamma^2 \partial_1^2)^{-1} \gamma \partial_1 \\ \gamma \partial_1 & 1 \end{pmatrix}$$ (15.1)

The preconditioner is implemented by defining a function of the form

---

\(^1\) Taken from a talk by L.Chacon available here https://bout2011.llnl.gov/pdf/talks/Chacon_bout2011.pdf

\(^2\) See paper https://arxiv.org/abs/1209.2054 for an application to 2-fluid equations
which takes as input the current time, the \( \gamma \) factor appearing above, and \( \delta \) which is only important for constrained problems (not discussed here... yet). The current state of the system is stored in the state variables (here \( u \) and \( v \)), whilst the vector to be preconditioned is stored in the time derivatives (here \( \text{ddt}(u) \) and \( \text{ddt}(v) \)). At the end of the preconditioner the result should be in the time derivatives. A preconditioner which is just the identity matrix and so does nothing is therefore:

\[
\text{int precon}(\text{BoutReal} \ t, \ \text{BoutReal} \ \gamma, \ \text{BoutReal} \ \delta) \{
\dots
\}
\]

To implement the preconditioner in equation (15.1), first apply the rightmost matrix to the given vector:

\[
\begin{pmatrix}
\text{ddt}(u) \\
\text{ddt}(v)
\end{pmatrix} =
\begin{pmatrix}
1 & 0 \\
\gamma \frac{\partial}{\partial \parallel} & 1
\end{pmatrix}
\begin{pmatrix}
\text{ddt}(u) \\
\text{ddt}(v)
\end{pmatrix}
\]

\[
\text{int precon}(\text{BoutReal} \ t, \ \text{BoutReal} \ \gamma, \ \text{BoutReal} \ \delta) \{
\quad \text{mesh->communicate(ddt(u));}
\quad //\text{ddt}(u) = \text{ddt}(u);
\quad \text{ddt}(v) = \gamma \text{Grad}_\parallel(\text{ddt}(u)) + \text{ddt}(v);
\}
\]

note that since the preconditioner is linear, it doesn’t depend on \( u \) or \( v \). As in the RHS function, since we are taking a differential of \( \text{ddt}(u) \), it first needs to be communicated to exchange guard cell values.

The second matrix

\[
\begin{pmatrix}
\text{ddt}(u) \\
\text{ddt}(v)
\end{pmatrix} \rightarrow
\begin{pmatrix}
1 & 0 \\
0 & (1 - \gamma^2 \frac{\partial^2}{\partial \parallel^2})^{-1}
\end{pmatrix}
\begin{pmatrix}
\text{ddt}(u) \\
\text{ddt}(v)
\end{pmatrix}
\]

doesn’t alter \( u \), but solves a parabolic equation in the parallel direction. There is a solver class to do this called \texttt{InvertPar} which solves the equation \( (A + B \frac{\partial^2}{\partial \parallel^2})x = b \) where \( A \) and \( B \) are \texttt{Field2D} or constants\(^3\). In \texttt{PhysicsModel::init()} we create one of these solvers:

\[
\texttt{InvertPar} *\text{inv}; // \text{Parallel inversion class}
\]

\[
\text{int init(\texttt{bool} \ restarting)} \{
\quad \dots
\quad \text{inv = InvertPar::Create();}
\quad \text{inv->setCoefA(1.0);}
\quad \dots
\}
\]

In the preconditioner we then use this solver to update \( v \):

\[
\text{inv->setCoefB(-}\sqrt{\gamma}(\gamma));
\]

\[
\text{ddt}(v) = \text{inv->solve(ddt(v))};
\]

which solves \( ddt(v) \rightarrow (1 - \gamma^2 \frac{\partial^2}{\partial \parallel^2})^{-1} ddt(v) \). The final matrix just updates \( u \) using this new solution for \( v \)

\[
\begin{pmatrix}
\text{ddt}(u) \\
\text{ddt}(v)
\end{pmatrix} \rightarrow
\begin{pmatrix}
1 & 0 \\
\gamma \frac{\partial}{\partial \parallel} & 1
\end{pmatrix}
\begin{pmatrix}
\text{ddt}(u) \\
\text{ddt}(v)
\end{pmatrix}
\]

\(^3\text{This InvertPar class can handle cases with closed field-lines and twist-shift boundary conditions for tokamak simulations}\)
Finally, boundary conditions need to be imposed, which should be consistent with the conditions used in the RHS:

```cpp
ddt(u).applyBoundary("dirichlet");
ddt(v).applyBoundary("dirichlet");
```

To use the preconditioner, pass the function to the solver in `PhysicsModel::init()`:

```cpp
int init(bool restarting) {
    solver->setPrecon(precon);
    ...}
```

then in the BOUT.inp settings file switch on the preconditioner

```plaintext
[solver]
type = cvode       # Need CVODE or PETSc
use_precon = true  # Use preconditioner
rightprec = false  # Use Right preconditioner (default left)
```

### 15.8 Jacobian function

### 15.9 DAE constraint equations

Using the IDA or IMEX-BDF2 solvers, BOUT++ can solve Differential Algebraic Equations (DAEs), in which algebraic constraints are used for some variables. Examples of how this is used are in the `examples/constraints` subdirectory.

First the variable to be constrained is added to the solver, in a similar way to time integrated variables. For example

```cpp
Field3D phi;
...
solver->constraint(phi, ddt(phi), "phi");
```

The first argument is the variable to be solved for (constrained). The second argument is the field to contain the residual (error). In this example the time derivative field `ddt(phi)` is used, but it could be another `Field3D` variable. The solver will attempt to find a solution to the first argument (`phi` here) such that the second argument (`ddt(phi)`) is zero to within tolerances.

In the RHS function the residual should be calculated. In this example (`examples/constraints/drift-wave-constraint`) we have:

```cpp
ddt(phi) = Delp2(phi) - Vort;
```

so the time integration solver includes the algebraic constraint $\nabla^2 \phi = \omega$.  

---

**15.8. Jacobian function**

**15.9. DAE constraint equations**
15.10 IMEX-BDF2

This is an implicit-explicit multistep method, which uses the PETSc library for the SNES nonlinear solver. To use this solver, BOUT++ must have been configured with PETSc support, and the solver type set to imexbdf2

```
[solver]
type = imexbdf2
```

For examples of using IMEX-BDF2, see the examples/IMEX/ subdirectory, in particular the diffusion-nl, drift-wave and drift-wave-constrain examples.

The time step is currently fixed (not adaptive), and defaults to the output timestep. To set a smaller internal timestep, the `solver:timestep` option can be set. If the timestep is too large, then the explicit part of the problem may become unstable, or the implicit part may fail to converge.

The implicit part of the problem can be solved matrix-free, in which case the Jacobian-vector product is approximated using finite differences. This is currently the default, and can be set on the command-line using the options:

```
solver:matrix_free=true -snes_mf
```

Note the `-snes_mf` flag which is passed to PETSc. When using a matrix free solver, the Jacobian is not calculated and so the amount of memory used is minimal. However, since the Jacobian is not known, many standard preconditioning methods cannot be used, and so in many cases a custom preconditioner is needed to obtain good convergence.

An experimental feature uses PETSc’s ability to calculate the Jacobian using finite differences. This can then speed up the linear solve, and allows more options for preconditioning. To enable this option:

```
solver:matrix_free=false
```

There are two ways to calculate the Jacobian: A brute force method which is set up by this call to PETSc which is generally very slow, and a “coloring” scheme which can be quite fast and is the default. Coloring uses knowledge of where the non-zero values are in the Jacobian, to work out which rows can be calculated simultaneously. The coloring code in IMEX-BDF2 currently assumes that every field is coupled to every other field in a star pattern: one cell on each side, a 7 point stencil for 3D fields. If this is not the case for your problem, then the solver may not converge.

The brute force method can be useful for comparing the Jacobian structure, so to turn off coloring:

```
solver:use_coloring=false
```

Using MatView calls, or the `-mat_view` PETSc options, the non-zero structure of the Jacobian can be plotted or printed.

15.11 Monitoring the simulation output

Monitoring of the solution can be done at two levels: output monitoring, and timestep monitoring. Output monitoring occurs only when data is written to file, whereas timestep monitoring is every timestep and so (usually) much more frequent. Examples of both are in examples/monitor and examples/monitor-newapi.

**Output monitoring:** At every output timestep the solver calls a monitor method of the BoutMonitor class, which writes the output dump file, calculates and prints timing information and estimated time remaining. If you want to run additional code or write data to a different file, you can implement the outputMonitor method of PhysicsModel:

```
int outputMonitor(BoutReal simtime, int iter, int nout)
```
The first input is the current simulation time, the second is the output number, and the last is the total number of outputs requested. This method is called by a monitor object PhysicsModel::modelMonitor, which writes the restart files at the same time. You can change the frequency at which the monitor is called by calling, in PhysicsModel::init:

```cpp
modelMonitor.setTimestep(new_timestep)
```

where `new_timestep` is a BoutReal which is either `timestep*n` or `timestep/n` for an integer `n`. Note that this will change the frequency of writing restarts as well as of calling `outputMonitor()`.

You can also add custom monitor object(s) for more flexibility.

You can call your output monitor class whatever you like, but it must be a subclass of Monitor and provide the method `call` which takes 4 inputs and returns an int:

```cpp
class MyOutputMonitor : public Monitor {
    int call(Solver *solver, BoutReal simtime, int iter, int NOUT) {
        ...
    }
};
```

The first input is the solver object, the second is the current simulation time, the third is the output number, and the last is the total number of outputs requested. To get the solver to call this function every output time, define a `MyOutputMonitor` object as a member of your PhysicsModel:

```cpp
MyOutputMonitor my_output_monitor;
```

and put in your `PhysicsModel::init()` code:

```cpp
solver->addMonitor(my_output_monitor);
```

If you want to later remove a monitor, you can do so with:

```cpp
solver->removeMonitor(my_output_monitor);
```

A simple example using this monitor is:

```cpp
class MyOutputMonitor : public Monitor{
    public:
    MyOutputMonitor(BoutReal timestep=-1):Monitor(timestep){};
    int call(Solver *solver, BoutReal simtime, int iter, int NOUT) override {
        output.write("Output monitor, time = %e, step %d of %d\n",
                      simtime, iter, NOUT);
        return 0;
    }
};
MyOutputMonitor my_monitor;
MyOutputMonitor::call(Solver *solver, BoutReal simtime, int iter, int NOUT) {
    return 0;
}
```

See the monitor example (`examples/monitor`) for full code.

**Timestep monitoring**: This uses functions instead of objects. First define a monitor function:

```cpp
class MyOutputMonitor : public Monitor{
    public:
    MyOutputMonitor(BoutReal timestep=-1):Monitor(timestep){};
    int call(Solver *solver, BoutReal simtime, int iter, int NOUT) override {
        output.write("Output monitor, time = %e, step %d of %d\n",
                      simtime, iter, NOUT);
        return 0;
    }
};
```

See the monitor example (`examples/monitor`) for full code.
where `simtime` will again contain the current simulation time, and `lastdt` the last timestep taken. Add this function to the solver:

```c
solver->addTimestepMonitor(my_timestep_monitor);
```

Timestep monitoring is disabled by default, unlike output monitoring. To enable timestep monitoring, set in the options file (BOUT.inp):

```
[solver]
monitor_timestep = true
```

or put on the command line `solver:monitor_timestep=true`. When this is enabled, it will change how solvers like CVODE and PVODE (the default solvers) are used. Rather than being run in NORMAL mode, they will instead be run in SINGLE_STEP mode (see the SUNDIALS notes here: https://computation.llnl.gov/casc/sundials/support/notes.html). This may in some cases be less efficient.

### 15.12 Implementation internals

The solver is the interface between BOUT++ and the time-integration codes such as SUNDIALS. All solvers implement the `Solver` class interface (see src/solver/generic_solver.hxx).

First all the fields which are to be evolved need to be added to the solver. These are always done in pairs, the first specifying the field, and the second the time-derivative:

```c
void add(Field2D &v, Field2D &F_v, const char* name);
```

This is normally called in the `PhysicsModel::init()` initialisation routine. Some solvers (e.g. IDA) can support constraints, which need to be added in the same way as evolving fields:

```c
bool constraints();
void constraint(Field2D &v, Field2D &C_v, const char* name);
```

The `constraints()` function tests whether or not the current solver supports constraints. The format of `constraint(...)` is the same as `add`, except that now the solver will attempt to make `C_v` zero. If `constraint` is called when the solver doesn’t support them then an error should occur.

If the physics model implements a preconditioner or Jacobian-vector multiplication routine, these can be passed to the solver during initialisation:

```c
typedef int (*PhysicsPrecon)(BoutReal t, BoutReal gamma, BoutReal delta);
void setPrecon(PhysicsPrecon f); // Specify a preconditioner
typedef int (*Jacobian)(BoutReal t);
void setJacobian(Jacobian j); // Specify a Jacobian
```

If the solver doesn’t support these functions then the calls will just be ignored.

Once the problem to be solved has been specified, the solver can be initialised using:

```c
int init(rhsfunc f, int argc, char **argv, bool restarting, int nout, BoutReal tstep);
```
which returns an error code (0 on success). This is currently called in `bout++.cxx`:

```c++
if(solver.init(rhs, argc, argv, restart, NOUT, TIMESTEP)) {
    output.write("Failed to initialise solver. Aborting\n");
    return(1);
}
```

which passes the (physics module) RHS function `PhysicsModel::rhs()` to the solver along with the number and size of the output steps.

```c++
typedef int (*MonitorFunc)(BoutReal simtime, int iter, int NOUT);
int run(MonitorFunc f);
```
In most BOUT++ simulations the Y coordinate is parallel to the magnetic field. In particular if the magnetic field $B$ can be expressed as

$$B = \nabla z \times \nabla x$$

then the Clebsch operators can be used. See section \textit{Differential operators} for more details.

The structure of the magnetic field can be simple, as in a slab geometry, but in many cases it is quite complicated. In a tokamak, for example, the magnetic shear causes deformation of grid cells and numerical issues. One way to overcome this is to transform between local coordinate systems, interpolating in the toroidal ($Z$) direction when calculating gradients along the magnetic field. This is called the \textit{shifted metric} method. In more general geometries such as stellarators, the magnetic field can have a 3D structure and stochastic regions. In this case the interpolation becomes 2D (in $X$ and $Z$), and is known as the Flux Coordinate Independent (FCI) method.

To handle these different cases in the same code, the BOUT++ mesh implements different \textit{ParallelTransform} classes. Each \textit{Field3D} class contains a pointer to the values up and down in the Y direction, called yup and ydown. These values are calculated during communication:

```cpp
Field3D f(0.0); // f allocated, set to zero
f.yup(); // error: f.yup not allocated

mesh->communicate(f);
f.yup(); // ok
f.ydown(0,1,0); // ok
```

In the case of slab geometry, yup and ydown point to the original field ($f$). For this reason the value of $f$ along the magnetic field from $f(x,y,z)$ is given by $f.ydown(x,y-1,z)$ and $f.yup(x,y+1,z)$. To take a second derivative along Y using the Field3D iterators (section \textit{Iterating over fields}):

```cpp
Field3D result;
result.allocate(); // Need to allocate before indexing

for(const auto &i : result.region(RGN_NOBNDRY)) {
    result[i] = f.yup()[i.yp()] - f.ydown()[i.ym()];
}
```

Note the use of yp() and ym() to increase and decrease the Y index.
16.1 Field-aligned grid

The default ParallelTransform is the identity transform, which sets yup() and ydown() to point to the same field. In the input options the setting is

```
[mesh]
paralleltransform = identity
```

This then uses the ParallelTransformIdentity class to calculate the yup and ydown fields.

This is mostly useful for slab geometries, where for a straight magnetic field the grid is either periodic in the y-direction or ends on a y-boundary. By setting the global option TwistShift = true and providing a ShiftAngle in the gridfile or [mesh] options a branch cut can be introduced between the beginning and end of the y-domain.

ParallelTransformIdentity can also be used in non-slab geometries. Then TwistShift = true should be set so that a twist-shift boundary condition is applied on closed field lines, as field-line following coordinates are not periodic in poloidal angle. Note that it is not recommended to use ParallelTransformIdentity with toroidal geometries, as magnetic shear will make the radial derivatives inaccurate away from the outboard midplane (which is normally chosen as the zero point for the integrated shear).

16.2 Shifted metric

The shifted metric method is selected using:

```
[mesh]
paralleltransform = shifted
```

so that mesh uses the ShiftedMetric class to calculate parallel transforms. During initialisation, this class reads a quantity zShift from the input or grid file. If zShift is not found then qinty is read instead. If qinty is not found then the angle is zero, and this method becomes the same as the identity transform. For each X and Z index, the zShift variable should contain the toroidal angle of a magnetic field line at \( z = 0 \) starting at \( \phi = 0 \) at a reference location \( \theta_0 \):

\[
zShift = \int_{\theta_0}^{\theta} \frac{B_\phi h_\theta}{B_\theta R} d\theta
\]

Note that here \( \theta_0 \) does not need to be constant in X (radius), since it is only the relative shifts between Y locations which matters.

16.3 FCI method

To use the FCI method for parallel transforms, set

```
[mesh]
paralleltransform = fci
```

which causes the FCITransform class to be used for parallel transforms. This reads four variables (3D fields) from the input grid: forward_xt_prime, forward_zt_prime, backward_xt_prime, and backward_zt_prime. These give the cell indices, not in general integers, in the forward (yup) and backward (ydown) directions. These are arranged so that forward_xt_prime(x,y,z) is the x index at y+1. Hence f.yup()(x,y+1,z) is calculated using forward_xt_prime(x,y,z) and forward_zt_prime(x,y,z), whilst f.ydown()(x,y-1,z) is calculated using backward_xt_prime(x,y,z) and backward_zt_prime(x,y,z).
Tools for calculating these mappings include Zoidberg, a Python tool which carries out field-line tracing and generates FCI inputs.
A common problem in plasma models is to solve an equation of the form

\[ d\nabla^2 \perp x + \frac{1}{c_1}(\nabla \perp c_2) \cdot \nabla \perp x + ax = b \]  

(17.1)

For example,

\[ \nabla^2 \perp x + ax = b \]

appears in reduced MHD for the vorticity inversion and \( j || \).

Alternative formulations and ways to invert equation (17.1) can be found in section LaplaceXY and LaplaceXZ.

Several implementations of the Laplacian solver are available, which are selected by changing the “type” setting. The currently available implementations are listed in table Table 17.1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>cyclic</td>
<td>Serial/parallel. Gathers boundary rows onto one processor.</td>
<td></td>
</tr>
<tr>
<td>petsc</td>
<td>Serial/parallel. Lots of methods, no Boussinesq</td>
<td>PETSc (section PETSc)</td>
</tr>
<tr>
<td>multigrid</td>
<td>Serial/parallel. Geometric multigrid, no Boussinesq</td>
<td></td>
</tr>
<tr>
<td>naulin</td>
<td>Serial/parallel. Iterative treatment of non-Boussinesq terms</td>
<td></td>
</tr>
<tr>
<td>serial_tr</td>
<td>Serial only. Thomas algorithm for tridiagonal system.</td>
<td>Lapack (section LAPACK)</td>
</tr>
<tr>
<td>serial_ban</td>
<td>Serial only. Enables 4th-order accuracy</td>
<td>Lapack (section LAPACK)</td>
</tr>
<tr>
<td>spt</td>
<td>Parallel only (NXPE&gt;1). Thomas algorithm.</td>
<td></td>
</tr>
<tr>
<td>mumps</td>
<td>Serial/parallel. Direct solver</td>
<td>MUMPS (section MUMPS)</td>
</tr>
<tr>
<td>pdd</td>
<td>Parallel Diagonally Dominant algorithm. Experimental</td>
<td></td>
</tr>
<tr>
<td>shoot</td>
<td>Shooting method. Experimental</td>
<td></td>
</tr>
</tbody>
</table>

### 17.1 Usage of the laplacian inversion

In BOUT++, equation (17.1) can be solved in two ways. The first method Fourier transforms in the \( z \)-direction, whilst the other solves the full two-dimensional problem by matrix inversion. The derivation of \( \nabla^2 \perp f \) for a general coordinate system can be found in the Field-aligned coordinates section. What is important, is to note that if \( g_{xy} \) and \( g_{yz} \) are non-zero, BOUT++ neglects the \( y \)-parallel derivatives when using the solvers Laplacian and LaplaceXZ.

By neglecting the \( y \)-derivatives (or if \( g_{xy} = g_{yz} = 0 \)), one can solve equation (17.1) \( y \) plane by \( y \) plane.

The first approach utilizes the fact that it is possible to Fourier transform the equation in \( z \) (using some assumptions described in section Numerical implementation), and solve a tridiagonal system for each mode. These inversion problems are band-diagonal (tri-diagonal in the case of 2nd-order differencing) and so inversions can be very efficient:
$O(n_z \log n_z)$ for the FFTs, $O(n_x)$ for tridiagonal inversion using the Thomas algorithm, where $n_x$ and $n_z$ are the number of grid-points in the $x$ and $z$ directions respectively.

In the second approach, the full 2-D system is solved. The available solvers for this approach are ‘multigrid’ using a multigrid algorithm; ‘naulin’ using an iterative scheme to correct the FFT-based approach; or ‘petsc’ using KSP linear solvers from the PETSc library (this requires PETSc to be built with BOUT++).

The `Laplacian` class is defined in `invert_laplace.hxx` and solves problems formulated like equation (17.1) To use this class, first create an instance of it:

```cpp
Laplacian *lap = Laplacian::create();
```

By default, this will use the options in a section called “laplace”, but can be given a different section as an argument. By default $d = 1$, $a = 0$, and $c_1 = c_2 = 1$. To set the values of these coefficients, there are the `setCoefA()`, `setCoefC1()`, `setCoefC2()`, `setCoefC()` (which sets both $c_1$ and $c_2$ to its argument), and `setCoefD()` methods:

```cpp
Field2D a = ...;
lap->setCoefA(a);
lap->setCoefC(0.5);
```

arguments can be `Field2D`, `Field3D`, or `BoutReal` values. Note that FFT solvers will use only the DC part of `Field3D` arguments.

Settings for the inversion can be set in the input file under the section `laplace` (default) or whichever settings section name was specified when the `Laplacian` class was created. Commonly used settings are listed in tables Table 17.2 to Table 17.5.

In particular boundary conditions on the $x$ boundaries can be set using the and `outer_boundary_flags` variables, as detailed in table Table 17.4. Note that DC (‘direct-current’) refers to $k = 0$ Fourier component, AC (‘alternating-current’) refers to $k \neq 0$ Fourier components. Non-Fourier solvers use AC options (and ignore DC ones). Multiple boundary conditions can be selected by adding together the required boundary condition flag values together. For example, `inner_boundary_flags = 3` will set a Neumann boundary condition on both AC and DC components.

It is pertinent to note here that the boundary in BOUT++ is defined by default to be located half way between the first guard point and first point inside the domain. For example, when a Dirichlet boundary condition is set, using `inner_boundary_flags = 0`, `16`, or `32`, then the first guard point, $f_-$ will be set to $f_- = 2v - f_+$, where $f_+$ is the first grid point inside the domain, and $v$ is the value to which the boundary is being set to.

The `global_flags`, `inner_boundary_flags`, `outer_boundary_flags` and `flags` values can also be set from within the physics module using `setGlobalFlags`, `setInnerBoundaryFlags` , `setOuterBoundaryFlags` and `setFlags`.

```cpp
lap->setGlobalFlags(Global_Flags_Value);
lap->setInnerBoundaryFlags(Inner_Flags_Value);
lap->setOuterBoundaryFlags(Outer_Flags_Value);
lap->setFlags(Flags_Value);
```
### Table 17.2: Laplacian inversion options

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Which implementation to use. See table Table 17.1</td>
<td>cyclic</td>
</tr>
<tr>
<td>filter</td>
<td>Filter out modes above ((1 - \text{filter}) \times k_{\text{max}}), if using Fourier solver</td>
<td>0</td>
</tr>
<tr>
<td>maxmode</td>
<td>Filter modes with (n &gt; \text{maxmode})</td>
<td>MZ/2</td>
</tr>
<tr>
<td>all_terms</td>
<td>Include first derivative terms</td>
<td>true</td>
</tr>
<tr>
<td>nonuniform</td>
<td>Include corrections for non-uniform meshes (dx not constant)</td>
<td>Same as global non_uniform. See here</td>
</tr>
<tr>
<td>global_flags</td>
<td>Sets global inversion options See table Laplace global flags</td>
<td>0</td>
</tr>
<tr>
<td>inner_boundary</td>
<td>Sets boundary conditions on inner boundary. See table Laplace boundary flags</td>
<td>0</td>
</tr>
<tr>
<td>outer_boundary</td>
<td>Sets boundary conditions on outer boundary. See table Laplace boundary flags</td>
<td>0</td>
</tr>
<tr>
<td>flags</td>
<td>DEPRECATED. Sets global solver options and boundary conditions. See Laplace flags or invert_laplace.cxx</td>
<td>0</td>
</tr>
<tr>
<td>include_yguards</td>
<td>Perform inversion in (y)-boundary guard cells</td>
<td>false</td>
</tr>
</tbody>
</table>

### Table 17.3: Laplacian inversion global_flags values: add the required quantities together.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Code variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No global option set</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>zero DC component (Fourier solvers)</td>
<td>INVERT_ZERO_DC</td>
</tr>
<tr>
<td>2</td>
<td>set initial guess to 0 (iterative solvers)</td>
<td>INVERT_START_NEW</td>
</tr>
<tr>
<td>4</td>
<td>equivalent to outer_boundary_flags = 128, inner_boundary_flags = 128</td>
<td>INVERT_BOTH_BNDRY_ONE</td>
</tr>
<tr>
<td>8</td>
<td>Use 4th order differencing (Apparently not actually implemented anywhere!!!)</td>
<td>INVERT_4TH_ORDER</td>
</tr>
<tr>
<td>16</td>
<td>Set constant component ((k_x = k_z = 0)) to zero</td>
<td>INVERT_KX_ZERO</td>
</tr>
</tbody>
</table>
Table 17.4: Laplacian inversion outer_boundary_flags or inner_boundary_flags values: add the required quantities together.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Code variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Dirichlet (Set boundary to 0)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Neumann on DC component (set gradient to 0)</td>
<td>INVERT_DC_GRAD</td>
</tr>
<tr>
<td>2</td>
<td>Neumann on AC component (set gradient to 0)</td>
<td>INVERT_AC_GRAD</td>
</tr>
<tr>
<td>4</td>
<td>Zero or decaying Laplacian on AC components ($\nabla^2 + k^2\psi$ vanishes/decays)</td>
<td>INVERT_AC_LAP</td>
</tr>
<tr>
<td>8</td>
<td>Use symmetry to enforce zero value or gradient (redundant for 2nd order now)</td>
<td>INVERT_SYM</td>
</tr>
<tr>
<td>16</td>
<td>Set boundary condition to values in boundary guard cells of second argument, $x_0$, of Laplacian::solve(const Field3D &amp;b, const Field3D &amp;x0). May be combined with any combination of 0, 1 and 2, i.e. a Dirichlet or Neumann boundary condition set to values which are $\neq 0$ or $f(y)$</td>
<td>INVERT_SET</td>
</tr>
<tr>
<td>32</td>
<td>Set boundary condition to values in boundary guard cells of RHS, $b$ in Laplacian::solve(const Field3D &amp;b, const Field3D &amp;x0). May be combined with any combination of 0, 1 and 2, i.e. a Dirichlet or Neumann boundary condition set to values which are $\neq 0$ or $f(y)$</td>
<td>INVERT_RHS</td>
</tr>
<tr>
<td>64</td>
<td>Zero or decaying Laplacian on DC components ($\nabla^2 + k^2\psi$ vanishes/decays)</td>
<td>INVERT_DC_LAP</td>
</tr>
<tr>
<td>128</td>
<td>Assert that there is only one guard cell in the $x$-boundary</td>
<td>INVERT_BNDRY_ONE</td>
</tr>
<tr>
<td>256</td>
<td>DC value is set to parallel gradient, $\nabla</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>DC value is set to inverse of parallel gradient $1/\nabla</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>Boundary condition for inner ‘boundary’ of cylinder</td>
<td>INVERT_IN_CYLINDER</td>
</tr>
</tbody>
</table>

Table 17.5: Laplacian inversion flags values (DEPRECATED!): add the required quantities together.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Zero-gradient DC on inner (X) boundary. Default is zero-value</td>
</tr>
<tr>
<td>2</td>
<td>Zero-gradient AC on inner boundary</td>
</tr>
<tr>
<td>4</td>
<td>Zero-gradient DC on outer boundary</td>
</tr>
<tr>
<td>8</td>
<td>Zero-gradient AC on outer boundary</td>
</tr>
<tr>
<td>16</td>
<td>Zero DC component everywhere</td>
</tr>
<tr>
<td>32</td>
<td>Not used currently</td>
</tr>
<tr>
<td>64</td>
<td>Set width of boundary to 1 (default is $\text{MXG}$)</td>
</tr>
<tr>
<td>128</td>
<td>Use $4^{th}$-order band solver (default is $2^{nd}$ order tridiagonal)</td>
</tr>
<tr>
<td>256</td>
<td>Attempt to set zero laplacian AC component on inner boundary by combining 2nd and 4th-order differencing at the boundary. Ignored if tridiagonal solver used.</td>
</tr>
<tr>
<td>512</td>
<td>Zero laplacian AC on outer boundary</td>
</tr>
<tr>
<td>1024</td>
<td>Symmetric boundary condition on inner boundary</td>
</tr>
<tr>
<td>2048</td>
<td>Symmetric outer boundary condition</td>
</tr>
</tbody>
</table>

To perform the inversion, there’s the solve method

```c
x = lap->solve(b);
```

There are also functions compatible with older versions of the BOUT++ code, but these are deprecated:
Field2D a, c, d;
invert_laplace(b, x, flags, &a, &c, &d);

and

\[
x = \text{invert}_\text{laplace}(b, \text{flags}, \&a, \&c, \&d);
\]

The input \(b\) and output \(x\) are 3D fields, and the coefficients \(a, c, \) and \(d\) are pointers to 2D fields. To omit any of the three coefficients, set them to NULL.

### 17.2 Numerical implementation

We will here go through the implementation of the laplacian inversion algorithm, as it is performed in BOUT++. We would like to solve the following equation for \(f\)

\[
d\nabla^2 f + \frac{1}{c_1}(\nabla c_2) \cdot \nabla f + af = b
\]  

(17.2)

BOUT++ neglects the \(y\)-parallel derivatives if \(g_{xy}\) and \(g_{yz}\) are non-zero when using the solvers Laplacian and LaplaceXZ. For these two solvers, equation (17.2) becomes (see Field-aligned coordinates for derivation)

\[
d(g^{xz}\partial^2_x + G^x\partial_x + g^{zz}\partial^2_z + G^z\partial_z + 2g^{xz}\partial_x\partial_z)f
\]

\[
+ \frac{1}{c_1}(e^x\partial_x + e^z\partial_z)c_2 \cdot (e^x\partial_x + e^z\partial_z)f
\]

\[
+ af = b
\]  

(17.3)

#### 17.2.1 Using tridiagonal solvers

Since there are no parallel \(y\)-derivatives if \(g_{xy} = g_{yz} = 0\) (or if they are neglected), equation (17.2) will only contain derivatives of \(x\) and \(z\) for the dependent variable. The hope is that the modes in the periodic \(z\) direction will decouple, so that we in the end only have to invert for the \(x\) coordinate.

If the modes decouples when Fourier transforming equation (17.3), we can use a tridiagonal solver to solve the equation for each Fourier mode.

Using the discrete Fourier transform

\[
F(x, y)_k = \frac{1}{N} \sum_{Z=0}^{N-1} f(x, y)_Z \exp\left(-\frac{2\pi ikZ}{N}\right)
\]

we see that the modes will not decouple if a term consist of a product of two terms which depends on \(Z\), as this would give terms like

\[
\frac{1}{N} \sum_{Z=0}^{N-1} a(x, y)_Z f(x, y)_Z \exp\left(-\frac{2\pi ikZ}{N}\right)
\]

Thus, in order to use a tridiagonal solver, \(a, c_1, c_2\) and \(d\) cannot be functions of \(Z\). Because of this, the \(e^z\partial_z c_2\) term in equation (17.3) is zero. Thus the tridiagonal solvers solve equations of the form

\[
d(x, y)(g^{xx}(x, y)\partial^2_x + G^x(x, y)\partial_x + g^{zz}(x, y)\partial^2_z + G^z(x, y)\partial_z + 2g^{xz}(x, y)\partial_x\partial_z)f(x, y, z)
\]

\[
+ \frac{1}{c_1(x, y)}(e^x\partial_x c_2(x, y)) \cdot (e^x\partial_x + e^z\partial_z)f(x, y, z)
\]

\[
+ a(x, y)f(x, y, z) = b(x, y, z)
\]
after using the discrete Fourier transform (see section Derivatives of the Fourier transform), we get
\[
\begin{align*}
&d(g^{xx}\partial_x^2 F_z + G^x\partial_z F_z + g^{zz}[ik]F_z) + \frac{1}{c_1}(e^x\partial_x c_2) \cdot (e^x\partial_x F_z + e^z ik F_z) \\
&+ aF_z = B_z
\end{align*}
\]
which gives
\[
\begin{align*}
&d(g^{xx}\partial_x^2 F_z + G^x\partial_z F_z - k^2 g^{zz} + ikG^z + ik2g^{zz}\partial_x)F_z \\
&+ \frac{1}{c_1}(\partial_x c_2)(g^{xx}\partial_x F_z + g^{zz} ik F_z) \\
&+ aF_z = B_z
\end{align*}
\] (17.4)
As nothing in equation (17.4) couples points in \( y \) together (since we neglected the \( y \)-derivatives if \( g_{xy} \) and \( g_{yz} \) were non-zero) we can solve \( y \)-plane by \( y \)-plane. Also, as the modes are decoupled, we may solve equation (17.4) \( k \) mode by \( k \) mode in addition to \( y \)-plane by \( y \)-plane.

The second order centred approximation of the first and second derivatives in \( x \) reads
\[
\begin{align*}
\partial_x f &\approx \frac{-f_{n-1} + f_{n+1}}{2dx} \\
\partial_x^2 f &\approx \frac{f_{n-1} - f_{n} + f_{n+1}}{dx^2}
\end{align*}
\]
This gives
\[
\begin{align*}
d \left( g^{xx}\frac{F_{z,n-1} - 2F_{z,n} + F_{z,n+1}}{dx^2} + G^x\frac{F_{z,n-1} - F_{z,n+1}}{2dx} - k^2 g^{zz} F_{z,n} \\
+ ikG^z F_{z,n} + ik2g^{zz}\frac{F_{z,n-1} + F_{z,n+1}}{2dx} \right) \\
+ \frac{1}{c_1}\left(-c_{2,n-1} + c_{2,n+1}\right) \left(g^{xx}\frac{F_{z,n-1} - F_{z,n+1}}{2dx} + g^{zz} ik F_{z,n}\right) \\
+ aF_{z,n} = B_{z,n}
\end{align*}
\]
collecting point by point
\[
\begin{align*}
& \left( \frac{dg^{xx}}{dx^2} - \frac{dG^x}{2dx} \right) \frac{g_{c_{1,n}}^{xx} - c_{2,n-1} + c_{2,n+1}}{4dx^2} - \frac{dk2g^{zz}}{2dx} F_{z,n-1} \\
&+ \left( -\frac{dg^{xx}}{dx^2} - dk^2 g^{zz} + a + idkG^z + \frac{g_{c_{1,n}}^{xx} - c_{2,n-1} + c_{2,n+1}k}{2dx} \right) F_{z,n} \\
&+ \left( \frac{dg^{xx}}{dx^2} + \frac{dG^x}{2dx} \right) \frac{g_{c_{1,n}}^{xx} - c_{2,n-1} + c_{2,n+1}}{4dx^2} + i\frac{dk2g^{zz}}{2dx} F_{z,n+1} = B_{z,n}
\end{align*}
\] (17.5)
We now introduce
\[
\begin{align*}
C_1 &= \frac{dg^{xx}}{dx^2} \\
C_2 &= dg^{zz} \\
C_3 &= \frac{2dg^{zz}}{2dx} \\
C_4 &= \frac{dG^x + g_{c_{1,n}}^{xx} - c_{2,n-1} + c_{2,n+1}}{2c_{1,n}dx} \\
C_5 &= \frac{dG^x + g_{c_{1,n}}^{xx} - c_{2,n-1} + c_{2,n+1}}{2dx}
\end{align*}
\]
which inserted in equation (17.5) gives

\[(C_1 - C_4 - ikC_3)F_{z,n-1} + (-2C_1 - k^2C_2 + ikC_5 + a)F_{z,n} + (C_1 + C_4 + ikC_3)F_{z,n+1} = B_{z,n}\]

This can be formulated as the matrix equation

\[AF_z = B_z\]

where the matrix \(A\) is tridiagonal. The boundary conditions are set by setting the first and last rows in \(A\) and \(B_z\).

The tridiagonal solvers previously required \(c_1 = c_2\) in equation (17.2), but from version 4.3 allow \(c_1 \neq c_2\).

### 17.2.2 Using PETSc solvers

When using PETSc, all terms of equation (17.3) are used when inverting to find \(f\). Note that when using PETSc, we do not Fourier decompose in the \(z\)-direction, so it may take substantially longer time to find the solution. As with the tridiagonal solver, the fields are sliced in the \(y\)-direction, and a solution is found for one \(y\) plane at the time.

Before solving, equation (17.3) is rewritten to the form \(Ax = b\) (however, the full \(A\) is not expanded in memory). To do this, a row \(i\) in the matrix \(A\) is indexed from bottom left of the two dimensional field = (0,0) to top right = (meshx - 1, meshz - 1) = meshx \cdot meshz - 1 of the two dimensional field. This is done in such a way that a row \(i\) in \(A\) increments by 1 for an increase of 1 in the \(z\)-direction, and by meshz for an increase of 1 in the \(x\)-direction, where the variables meshx and meshz represents the highest value of the field in the given direction.

Similarly to equation (17.5), the discretised version of equation (17.3) can be written. Doing the same for the full two dimensional case yields:

**Second order approximation**

\[
\begin{align*}
&c_{i,j} f_{i,j} + c_{i-1,j-1} f_{i-1,j-1} + c_{i-1,j} f_{i-1,j} + c_{i,j-1} f_{i,j-1} + c_{i,j} f_{i,j} + c_{i+1,j-1} f_{i+1,j-1} + c_{i+1,j} f_{i+1,j} + c_{i+1,j+1} f_{i+1,j+1} \\
&= b_{i,j}
\end{align*}
\]

**Fourth order approximation**

\[
\begin{align*}
&c_{i,j} f_{i,j} + c_{i-2,j-2} f_{i-2,j-2} + c_{i-2,j-1} f_{i-2,j-1} + c_{i-2,j} f_{i-2,j} + c_{i-2,j+1} f_{i-2,j+1} \\
&+ c_{i-2,j+2} f_{i-2,j+2} + c_{i-1,j-2} f_{i-1,j-2} + c_{i-1,j-1} f_{i-1,j-1} + c_{i-1,j} f_{i-1,j} + c_{i-1,j+1} f_{i-1,j+1} + c_{i-1,j+2} f_{i-1,j+2} \\
&+ c_{i-2,j} f_{i-2,j-2} + c_{i,j} f_{i,j-1} + c_{i,j+1} f_{i,j+1} + c_{i,j+2} f_{i,j+2} + c_{i+1,j-2} f_{i+1,j-2} + c_{i+1,j-1} f_{i+1,j-1} + c_{i+1,j} f_{i+1,j} + c_{i+1,j+1} f_{i+1,j+1} \\
&+ c_{i+1,j+2} f_{i+1,j+2} + c_{i+2,j-2} f_{i+2,j-2} + c_{i+2,j-1} f_{i+2,j-1} + c_{i+2,j} f_{i+2,j} + c_{i+2,j+1} f_{i+2,j+1} + c_{i+2,j+2} f_{i+2,j+2} \\
&= b_{i,j}
\end{align*}
\]
To determine the coefficient for each node point, it is convenient to introduce some quantities

\[ A_0 = a(x, y_{\text{current}}, z) \]
\[ A_1 = dg^{zz} \]
\[ A_2 = dg^{zz} \]
\[ A_3 = 2dg^{zz} \] (17.6)

In addition, we have:

Second order approximation (5-point stencil)

\[
\begin{align*}
\text{ddx}_c &= \frac{c_2_{x+1} - c_2_{x-1}}{2c_1dx} \\
\text{ddz}_c &= \frac{c_2_{z+1} - c_2_{z-1}}{2c_1dz}
\end{align*}
\]

Fourth order approximation (9-point stencil)

\[
\begin{align*}
\text{ddx}_c &= \frac{-c_2_{x+2} + 8c_2_{x+1} - 8c_2_{x-1} + c_2_{x-2}}{12c_1dx} \\
\text{ddz}_c &= \frac{-c_2_{z+2} + 8c_2_{z+1} - 8c_2_{z-1} + c_2_{z-2}}{12c_1dz}
\end{align*}
\]

This gives

\[ A_4 = dG^{xx} + g^{xx}\text{ddx}_c + g^{zz}\text{ddz}_c \]
\[ A_5 = dG^{xx} + g^{zz}\text{ddx}_c + g^{zz}\text{ddz}_c \]

The coefficients \( c_{i+m,j+n} \) are finally being set according to the appropriate order of discretisation. The coefficients can be found in the file `petsc_laplace.cxx`.

### 17.2.3 Example: The 5-point stencil

Let us now consider the 5-point stencil for a mesh with 3 inner points in the \( x \)-direction, and 3 inner points in the \( z \)-direction. The \( z \) direction will be periodic, and the \( x \) direction will have the boundaries half between the grid-point and the first ghost point (see Fig. 17.1).

![Fig. 17.1: The mesh: The inner boundary points in \( x \) are coloured in orange, whilst the outer boundary points in \( z \) are coloured gray. Inner points are coloured blue.](image)

Applying the 5-point stencil to point \( f_{22} \) this mesh will result in Fig. 17.2.

We want to solve a problem on the form \( Ax = b \). We will order \( x \) in a row-major order (so that \( z \) is varying faster than \( x \)). Further, we put the inner \( x \) boundary points first in \( x \), and the outer \( x \) boundary points last in \( x \). The matrix problem for our mesh can then be written like in Fig. 17.3.

As we are using a row-major implementation, the global indices of the matrix will be as in Fig. 17.4.
Fig. 17.2: The mesh with a stencil in point $f_{22}$: The point under consideration is coloured blue. The point located $+1$ in $z$ direction ($zp$) is coloured yellow and the point located $-1$ in $z$ direction ($zm$) is coloured green. The point located $+1$ in $x$ direction ($xp$) is coloured purple and the point located $-1$ in $x$ direction ($xm$) is coloured red.

Fig. 17.3: Matrix problem for our $3 \times 3$ mesh: The colors follow that of figure Fig. 17.1 and Fig. 17.2. The first index of the elements refers to the $x$-position in figure Fig. 17.1, and the last index of the elements refers to the $z$-position in figure Fig. 17.1. $ig$ refers to “inner ghost point”, $og$ refers to “outer ghost point”, and $c$ refers to the point of consideration. Notice the “wrap-around” in $z$-direction when the point of consideration neighbours the first/last $z$-index.

Fig. 17.4: Global indices of the matrix in figure Fig. 17.3

## 17.2. Numerical implementation
17.3 Implementation internals

The Laplacian inversion code solves the equation:

\[
d\nabla^2 \perp x + \frac{1}{c_1} \nabla \perp c_2 \cdot \nabla \perp x + ax = b
\]

where \( x \) and \( b \) are 3D variables, whilst \( a, c_1, c_2 \) and \( d \) are 2D variables for the FFT solvers, or 3D variables otherwise. Several different algorithms are implemented for Laplacian inversion, and they differ between serial and parallel versions. Serial inversion can currently either be done using a tridiagonal solver (Thomas algorithm), or a band-solver (allowing 4\textsuperscript{th}-order differencing).

To support multiple implementations, a base class \textit{Laplacian} is defined in include/invert_laplace.hxx. This defines a set of functions which all implementations must provide:

```cpp
class Laplacian {
public:
    virtual void setCoefA(const Field2D &val) = 0;
    virtual void setCoefC(const Field2D &val) = 0;
    virtual void setCoefD(const Field2D &val) = 0;
    virtual const FieldPerp solve(const FieldPerp &b) = 0;
}
```

At minimum, all implementations must provide a way to set coefficients, and a solve function which operates on a single FieldPerp (X-Y) object at once. Several other functions are also virtual, so default code exists but can be overridden by an implementation.

For convenience, the \textit{Laplacian} base class also defines a function to calculate coefficients in a Tridiagonal matrix:

```cpp
void tridagCoefs(int jx, int jy, int jz, dcomplex &a, dcomplex &b, dcomplex &c, const Field2D *c1coef = nullptr, const Field2D *c2coef = nullptr, const Field2D *d = nullptr);
```

For the user of the class, some static functions are defined:

```cpp
static Laplacian* create(Options *opt = nullptr);
static Laplacian* defaultInstance();
```

The create function allows new Laplacian implementations to be created, based on options. To use the options in the \[laplace\] section, just use the default:

```cpp
Laplacian* lap = Laplacian::create();
```

The code for the \textit{Laplacian} base class is in src/invert/laplace/invert_laplace.cxx. The actual creation of new Laplacian implementations is done in the \textit{LaplaceFactory} class, defined in src/invert/laplace/laplacefactory.cxx. This file includes all the headers for the implementations, and chooses which one to create based on the type setting in the input options. This factory therefore provides a single point of access to the underlying Laplacian inversion implementations.

Each of the implementations is in a subdirectory of src/invert/laplace/impls and is discussed below.
17.3.1 Serial tridiagonal solver

This is the simplest implementation, and is in src/invert/laplace/impls/serial_tri/

17.3.2 Serial band solver

This is band-solver which performs a $4^{th}$-order inversion. Currently this is only available when $\text{NXPE}=1$; when more than one processor is used in $x$, the Laplacian algorithm currently reverts to $3^{rd}$-order.

17.3.3 SPT parallel tridiagonal

This is a reference code which performs the same operations as the serial code. To invert a single XZ slice ($\text{FieldPerp}$ object), data must pass from the innermost processor ($\text{mesh->PE_XIND} = 0$) to the outermost $\text{mesh->PE_XIND} = \text{mesh->NXPE}-1$ and back again.

Some parallelism is achieved by running several inversions simultaneously, so while processor 1 is inverting $Y=0$, processor 0 is starting on $Y=1$. This works ok as long as the number of slices to be inverted is greater than the number of $X$ processors ($\text{MYSUB} > \text{mesh->NXPE}$). If $\text{MYSUB} < \text{mesh->NXPE}$ then not all processors can be busy at once, and so efficiency will fall sharply. Fig. 17.5 shows the usage of 4 processors inverting a set of 3 poloidal slices (i.e. $\text{MYSUB}=3$).

17.3.4 PDD algorithm

This is the Parallel Diagonally Dominant (PDD) algorithm. It’s very fast, but achieves this by neglecting some cross-processor terms. For ELM simulations, it has been found that these terms are important, so this method is not usually used.

17.3.5 Cyclic algorithm

This is now the default solver in both serial and parallel. It is an FFT-based solver using a cyclic reduction algorithm.

17.3.6 Multigrid solver

A solver using a geometric multigrid algorithm was introduced by projects in 2015 and 2016 of CCFE and the EURO-fusion HLST.

17.3.7 Naulin solver

This scheme was introduced for BOUT++ by Michael Løiten in the CELMA code and the iterative algorithm is detailed in his thesis [Løiten2017].

The iteration can be under-relaxed (see naulin_laplace.cxx for more details of the implementation). A factor $0 < \text{underrelax\_factor} <= 1$ is used, with a value of 1 corresponding to no under-relaxation. If the iteration starts to diverge (the error increases on any step) the underrelax\_factor is reduced by a factor of 0.9, and the iteration is restarted from the initial guess. The initial value of underrelax\_factor, which underrelax\_factor is set to at the beginning of each call to solve can be set by the option initial\_underrelax\_factor (default is 1.0) in the appropriate section of the input file ([laplace] by default). Reducing the value of initial\_underrelax\_factor may speed up convergence in some cases. Some statistics from the solver are written to the output files to help in choosing this value. With $<i>$ being the number of the $\text{LaplaceNaulin}$ solver, counting in the order they are created in the physics model:
Fig. 17.5: Parallel Laplacian inversion with MYSUB=3 on 4 processors. Red periods are where a processor is idle - in this case about 40% of the time.
• `naulinsolver<i>_mean_underrelax_counts` gives the mean number of times `underrelax_factor` had to be reduced to get the iteration to converge. If this is much above 0, it is probably worth reducing `initial_underrelax_factor`.

• `naulinsolver<i>_mean_its` is the mean number of iterations taken to converge. Try to minimise when adjusting `initial_underrelax_factor`.

### 17.4 LaplaceXY

Perpendicular Laplacian solver in X-Y.

\[ \nabla \perp f = \nabla f - b (b \cdot \nabla) \]

\[ = (\frac{\partial f}{\partial x} - g_{xy} \frac{\partial f}{\partial y}) \nabla x + \left( \frac{\partial f}{\partial z} - g_{yz} \frac{\partial f}{\partial y} \right) \nabla z \]

In 2D (X-Y), the \( g_{xy} \) component can be dropped since this depends on integrated shear \( I \) which will cancel with the \( g_{xz} \) component. The \( z \) derivative is zero and so this simplifies to

\[ \nabla \perp f = \frac{\partial f}{\partial x} \nabla x - g_{yz} \frac{\partial f}{\partial y} \nabla z \]

The divergence operator in conservative form is

\[ \nabla \cdot A = \frac{1}{J} \frac{\partial}{\partial u^i} \left( J g^{ij} A_j \right) \]

and so the perpendicular Laplacian in X-Y is

\[ \nabla^2 \perp f = \frac{1}{J} \frac{\partial}{\partial x} \left( J g^{xx} \frac{\partial f}{\partial x} \right) - \frac{1}{J} \frac{\partial}{\partial y} \left( J g^{yy} \frac{\partial f}{\partial y} \right) \]

In field-aligned coordinates, the metrics in the \( y \) derivative term become:

\[ g_{yz} = \frac{B_y^2}{B^2} \frac{1}{h_u} \]

In the LaplaceXY operator this is implemented in terms of fluxes at cell faces.

\[ \frac{1}{J} \frac{\partial}{\partial x} \left( J g^{xx} \frac{\partial f}{\partial x} \right) \rightarrow \frac{1}{J_i \Delta x} \left[ J_{i+1/2} g_{xx}^{i+1/2} \left( \frac{f_{i+1} - f_i}{\Delta x_{i+1/2}} \right) - J_{i-1/2} g_{xx}^{i-1/2} \left( \frac{f_i - f_{i-1}}{\Delta x_{i-1/2}} \right) \right] \]

Notes:

- The ShiftedMetric or FCITransform ParallelTransform must be used (i.e. `mesh:paralleltransform = shifted` or `mesh:paralleltransform = fci`) for this to work, since it assumes that \( g^{xx} = 0 \)

- Setting the option `pctype = hypre` seems to work well, if PETSc has been compiled with the algebraic multigrid library hypre; this can be included by passing the option `--download-hypre` to PETSc’s configure script.

- LaplaceXY (with the default finite-volume discretisation) has a slightly different convention for passing non-zero boundary values than the Laplacian solvers. LaplaceXY uses the average of the last grid cell and first boundary cell of the initial guess (second argument to `solve()`) as the value to impose for the boundary condition.

An alternative discretization is available if the option `finite_volume = false` is set. Then a finite-difference discretization very close to the one used when calling \( A^*\text{Laplace}_\text{perp}(f) + \text{Grad}_\text{perp}(A)^*\text{Grad}_\text{perp}(f) + B^*f \) is used. This also supports non-orthogonal grids with \( g^{xy} \neq 0 \). The difference is that when \( g^{xy} \neq 0 \), Laplace_perp calls `D2DXDY(f)` which applies a boundary condition to \( df/dy = DDY(f) \) before calculating `DDX(df/dy)` with a slightly different result than the way boundary conditions are applied in LaplaceXY.
• The finite difference implementation of LaplaceXY passes non-zero values for the boundary conditions in the same way as the Laplacian solvers. The value in the first boundary cell of the initial guess (second argument to solve()) is used as the boundary value. (Note that this value is imposed as a boundary condition on the returned solution at a location half way between the last grid cell and first boundary cell.)

17.5 LaplaceXZ

This is a Laplace inversion code in X-Z, similar to the Laplacian solver described in Laplacian inversion. The difference is in the form of the Laplacian equation solved, and the approach used to derive the finite difference formulae. The equation solved is:

\[ \nabla \cdot (A \nabla \perp f) + B f = b \]

where \( A \) and \( B \) are coefficients, \( b \) is the known RHS vector (e.g. vorticity), and \( f \) is the unknown quantity to be calculated (e.g. potential), and \( \nabla \perp f \) is the same as equation (17.8), but with negligible \( y \)-parallel derivatives if \( g_{xy} \), \( g_{yz} \) and \( g_{zz} \) is non-vanishing. The Laplacian is written in conservative form like the LaplaceXY solver, and discretised in terms of fluxes through cell faces.

\[ \frac{1}{J} \frac{\partial}{\partial x} \left( J A g_{xx} \frac{\partial f}{\partial x} \right) + \frac{1}{J} \frac{\partial}{\partial z} \left( J A g_{zz} \frac{\partial f}{\partial z} \right) + B f = b \]

The header file is include/bout/invert/laplacexz.hxx. The solver is constructed by using the LaplaceXZ::create() function:

```c++
LaplaceXZ *lap = LaplaceXZ::create(mesh);
```

Note that a pointer to a Mesh object must be given, which for now is the global variable mesh. By default the options section laplacexz is used, so to set the type of solver created, set in the options

```
[laplacexz]
type = petsc    # Set LaplaceXZ type
```

or on the command-line laplacexz:type=petsc.

The coefficients must be set using setCoefs. All coefficients must be set at the same time:

```c++
lap->setCoefs(1.0, 0.0);
```

Constants, Field2D or Field3D values can be passed. If the implementation doesn’t support Field3D values then the average over \( z \) will be used as a Field2D value.

To perform the inversion, call the solve function:

```c++
Field3D vort = ...;
Field3D phi = lap->solve(vort, 0.0);
```

The second input to solve is an initial guess for the solution, which can be used by iterative schemes e.g. using PETSc.
17.5.1 Implementations

The currently available implementations are:

- **cyclic**: This implementation assumes coefficients are constant in \( Z \), and uses FFTs in \( z \) and a complex tridiagonal solver in \( x \) for each \( z \) mode (the CyclicReduction solver). Code in \texttt{src/invert/laplacexz/impls/cyclic/}

- **petsc**: This uses the PETSc KSP interface to solve a matrix with coefficients varying in both \( x \) and \( z \). To improve efficiency of direct solves, a different matrix is used for preconditioning. When the coefficients are updated the preconditioner matrix is not usually updated. This means that LU factorisations of the preconditioner can be re-used. Since this factorisation is a large part of the cost of direct solves, this should greatly reduce the run-time.

17.5.2 Test case

The code in \texttt{examples/test-laplacexz} is a simple test case for \textit{LaplaceXZ}. First it creates a \textit{LaplaceXZ} object:

```cpp
LaplaceXZ *inv = LaplaceXZ::create(mesh);
```

For this test the **petsc** implementation is the default:

```cpp
[laplacexz]
type = petsc
ksptype = gmres # Iterative method
pctype = lu # Preconditioner
```

By default the LU preconditioner is used. PETSc’s built-in factorisation only works in serial, so for parallel solves a different package is needed. This is set using:

```cpp
factor_package = superlu_dist
```

This setting can be “petsc” for the built-in (serial) code, or one of “superlu”, “superlu_dist”, “mumps”, or “cusparse”. Then we set the coefficients:

```cpp
inv->setCoefs(Field3D(1.0),Field3D(0.0));
```

Note that the scalars need to be cast to fields (Field2D or Field3D) otherwise the call is ambiguous. Using the PETSc command-line flag \texttt{-mat_view ::ascii_info} information on the assembled matrix is printed:

```
$ mpirun -np 2 ./test-laplacexz -mat_view ::ascii_info
...
Matrix Object: 2 MPI processes
type: mpiaij
rows=1088, cols=1088
total: nonzeros=5248, allocated nonzeros=5248
total number of mallocs used during MatSetValues calls =0
  not using I-node (on process 0) routines
...
```

which confirms that the matrix element pre-allocation is setting the correct number of non-zero elements, since no additional memory allocation was needed.

A field to invert is created using FieldFactory:
Field3D rhs = FieldFactory::get()->create3D("rhs", Options::getRoot(), mesh);

which is currently set to a simple function in the options:

\[ \text{rhs} = \sin(x - z) \]

and then the system is solved:

Field3D x = inv->solve(rhs, 0.0);

Using the PETSc command-line flags -ksp_monitor to monitor the iterative solve, and -mat_superlu_dist_statprint to monitor SuperLU_dist we get:

| Nonzeros in L | 19984 |
| Nonzeros in U | 19984 |
| Nonzeros in L+U | 38880 |
| Nonzeros in LSUB | 11900 |
| NUMfact space (MB) | sum (procs): L\U 0.45 all 0.61 |
| Total highmark (MB): | All 0.62 Avg 0.31 Max 0.36 |
| Mat conversion (PETSc->SuperLU_DIST) time (max/min/avg): | 4.69685e-05 / 4.69685e-05 / 4.69685e-05 |
| EQUIL time | 0.00 |
| ROWPERM time | 0.00 |
| COLPERM time | 0.00 |
| SYMBFACT time | 0.00 |
| DISTRIBUT time | 0.00 |
| FACTOR time | 0.00 |
| Factor flops | 1.073774e+06 Mflops 222.08 |
| SOLVE time | 0.00 |
| SOLVE time | 0.00 |
| Solve flops | 8.245800e+04 Mflops 28.67 |

0 KSP Residual norm 5.16956004406e+02
| SOLVE time | 0.00 |
| Solve flops | 8.245800e+04 Mflops 60.50 |

0 KSP Residual norm 1.359142853145e-12
| SOLVE time | 0.00 |
| Solve flops | 8.245800e+04 Mflops 49.86 |

So after the initial setup and factorisation, the system is solved in one iteration using the LU direct solve.

As a test of re-using the preconditioner, the coefficients are then modified:

inv->setCoefs(Field3D(2.0), Field3D(0.1));

and solved again:

| SOLVE time | 0.00 |
| Solve flops | 8.245800e+04 Mflops 84.15 |

0 KSP Residual norm 5.16960004406e+02
| SOLVE time | 0.00 |
| Solve flops | 8.245800e+04 Mflops 90.42 |

(continues on next page)
Solve flops 8.245800e+04 Mflops 94.88
KSP Residual norm 2.813291076609e+02
SOLVE time 0.00
Solve flops 8.245800e+04 Mflops 87.27
KSP Residual norm 1.68868390433e+02
SOLVE time 0.00
Solve flops 8.245800e+04 Mflops 88.77
KSP Residual norm 7.436784980024e+01
SOLVE time 0.00
Solve flops 8.245800e+04 Mflops 89.55
KSP Residual norm 1.835640800835e+01
SOLVE time 0.00
Solve flops 8.245800e+04 Mflops 88.00
KSP Residual norm 5.386963293959e-01
SOLVE time 0.00
Solve flops 8.245800e+04 Mflops 93.50
KSP Residual norm 2.093714782067e-02
SOLVE time 0.00
Solve flops 8.245800e+04 Mflops 89.44
KSP Residual norm 5.838501185134e-04
SOLVE time 0.00
Solve flops 8.245800e+04 Mflops 81.47

Note that this time there is no factorisation step, but the direct solve is still very effective.

### 17.5.3 Blob2d comparison

The example examples/blob2d-laplacexz is the same as examples/blob2d but with LaplaceXZ rather than Laplacian.

Tests on one processor: Using Boussinesq approximation, so that the matrix elements are not changed, the cyclic solver produces output:

<table>
<thead>
<tr>
<th></th>
<th>1.000e+02</th>
<th>1.000e+02</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>125</td>
<td>146</td>
</tr>
<tr>
<td>v</td>
<td>8.28e-01</td>
<td>1.15e+00</td>
</tr>
<tr>
<td>p</td>
<td>71.8</td>
<td>61.9</td>
</tr>
<tr>
<td>q</td>
<td>8.2</td>
<td>20.5</td>
</tr>
<tr>
<td>r</td>
<td>0.4</td>
<td>0.5</td>
</tr>
<tr>
<td>s</td>
<td>0.6</td>
<td>0.9</td>
</tr>
<tr>
<td>t</td>
<td>18.9</td>
<td>16.2</td>
</tr>
</tbody>
</table>

whilst the PETSc solver with LU preconditioner outputs:

<table>
<thead>
<tr>
<th></th>
<th>1.000e+02</th>
<th>1.000e+02</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>44</td>
<td>42</td>
</tr>
<tr>
<td>v</td>
<td>3.00e-01</td>
<td>3.30e-01</td>
</tr>
<tr>
<td>p</td>
<td>69.4</td>
<td>58.2</td>
</tr>
<tr>
<td>q</td>
<td>8.1</td>
<td>20.2</td>
</tr>
<tr>
<td>r</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>s</td>
<td>2.1</td>
<td>3.7</td>
</tr>
<tr>
<td>t</td>
<td>20.0</td>
<td>17.5</td>
</tr>
</tbody>
</table>

so the PETSc direct solver seems to take only slightly longer than the cyclic solver. For comparison, GMRES with Jacobi preconditioning gives:

<table>
<thead>
<tr>
<th></th>
<th>1.000e+02</th>
<th>1.000e+02</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>130</td>
<td>78</td>
</tr>
<tr>
<td>v</td>
<td>2.66e+00</td>
<td>1.16e+00</td>
</tr>
<tr>
<td>p</td>
<td>24.1</td>
<td>33.8</td>
</tr>
<tr>
<td>q</td>
<td>68.3</td>
<td>54.9</td>
</tr>
<tr>
<td>r</td>
<td>0.2</td>
<td>0.3</td>
</tr>
<tr>
<td>s</td>
<td>0.8</td>
<td>1.1</td>
</tr>
<tr>
<td>t</td>
<td>6.6</td>
<td>9.9</td>
</tr>
</tbody>
</table>

### 17.5. LaplaceXZ
and with SOR preconditioner:

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.00e+02</td>
<td>1.54e+00</td>
<td>38.6</td>
<td>50.2</td>
<td>0.3</td>
<td>0.4</td>
<td>10.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.00e+02</td>
<td>45</td>
<td>4.51e-01</td>
<td>46.8</td>
<td>37.8</td>
<td>0.3</td>
<td>1.7</td>
<td>13.4</td>
</tr>
</tbody>
</table>

When the Boussinesq approximation is not used, the PETSc solver with LU preconditioning, re-setting the preconditioner every 100 solves gives:

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.00e+02</td>
<td>142</td>
<td>3.06e+00</td>
<td>23.0</td>
<td>70.7</td>
<td>0.2</td>
<td>0.2</td>
<td>6.0</td>
</tr>
<tr>
<td></td>
<td>2.00e+02</td>
<td>41</td>
<td>9.47e-01</td>
<td>21.0</td>
<td>72.1</td>
<td>0.3</td>
<td>0.6</td>
<td>6.1</td>
</tr>
</tbody>
</table>

i.e. around three times slower than the Boussinesq case. When using jacobi preconditioner:

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.00e+02</td>
<td>128</td>
<td>2.59e+00</td>
<td>22.9</td>
<td>70.8</td>
<td>0.2</td>
<td>0.2</td>
<td>5.9</td>
</tr>
<tr>
<td></td>
<td>2.00e+02</td>
<td>68</td>
<td>1.18e+00</td>
<td>26.5</td>
<td>64.6</td>
<td>0.2</td>
<td>0.6</td>
<td>8.1</td>
</tr>
</tbody>
</table>

For comparison, the Laplacian solver using the tridiagonal solver as preconditioner gives:

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.00e+02</td>
<td>222</td>
<td>5.70e+00</td>
<td>17.4</td>
<td>77.9</td>
<td>0.1</td>
<td>0.1</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>2.00e+02</td>
<td>172</td>
<td>3.84e+00</td>
<td>20.2</td>
<td>74.2</td>
<td>0.2</td>
<td>0.2</td>
<td>5.2</td>
</tr>
</tbody>
</table>

or with Jacobi preconditioner:

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.00e+02</td>
<td>107</td>
<td>3.13e+00</td>
<td>15.8</td>
<td>79.5</td>
<td>0.1</td>
<td>0.2</td>
<td>4.3</td>
</tr>
<tr>
<td></td>
<td>2.00e+02</td>
<td>110</td>
<td>2.14e+00</td>
<td>23.5</td>
<td>69.2</td>
<td>0.2</td>
<td>0.3</td>
<td>6.7</td>
</tr>
</tbody>
</table>

The LaplaceXZ solver does not appear to be dramatically faster in serial than the Laplacian solver when the matrix coefficients are modified every solve. When matrix elements are not modified then the solve time is competitive with the tridiagonal solver.

As a test, timing only the setCoefs call for the non-Boussinesq case gives:

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.00e+02</td>
<td>142</td>
<td>1.86e+00</td>
<td>83.3</td>
<td>9.5</td>
<td>0.2</td>
<td>0.3</td>
<td>6.7</td>
</tr>
<tr>
<td></td>
<td>2.00e+02</td>
<td>41</td>
<td>5.04e-01</td>
<td>83.1</td>
<td>8.0</td>
<td>0.3</td>
<td>1.2</td>
<td>7.3</td>
</tr>
</tbody>
</table>

so around 9% of the run-time is in setting the coefficients, and the remaining ~ 60% in the solve itself.
CHAPTER EIGHTEEN

DIFFERENTIAL OPERATORS

There are a huge number of possible ways to perform differencing in computational fluid dynamics, and BOUT++ is intended to be able to implement a large number of them. This means that the way differentials are handled internally is quite involved; see the developer’s manual for full gory details. Much of the time this detail is not all that important, and certainly not while learning to use BOUT++. Default options are therefore set which work most of the time, so you can start using the code without getting bogged down in these details.

In order to handle many different differencing methods and operations, many layers are used, each of which handles just part of the problem. The main division is between differencing methods (such as 4th-order central differencing), and differential operators (such as $\nabla|\|_\cdot$).

18.1 Differencing methods

Methods are typically implemented on 5-point stencils (although exceptions are possible) and are divided into three categories:

- Central-differencing methods, for diffusion operators $\frac{df}{dx}, \frac{d^2 f}{dx^2}$. Each method has a short code, and currently include
  - C2: 2$^{nd}$ order $f_{-1} - 2f_0 + f_1$
  - C4: 4$^{th}$ order $(-f_{-2} + 16f_{-1} - 30f_0 + 16f_1 - f_2)/12$
  - S2: 2$^{nd}$ order smoothing derivative
  - W2: 2$^{nd}$ order CWENO
  - W3: 3$^{rd}$ order CWENO

- Upwinding methods for advection operators $v_x \frac{df}{dx}$
  - U1: 1$^{st}$ order upwinding
  - U2: 2$^{nd}$ order upwinding
  - U3: 3$^{rd}$ order upwinding
  - U4: 4$^{th}$ order upwinding
  - C2: 2$^{nd}$ order central
  - C4: 4$^{th}$ order central
  - W3: 3$^{rd}$ order Weighted Essentially Non-Oscillatory (WENO)

- Flux conserving and limiting methods for terms of the form $\frac{d}{dx} (v_x f)$
  - U1: 1$^{st}$ order upwinding
Special methods:

- **FFT**: Classed as a central method, Fourier Transform method in Z (axisymmetric) direction only. Currently available for first and second order central difference

- **SPLIT**: A flux method that splits into upwind and central terms \( \frac{d}{dx}(v_x f) = v_x \frac{df}{dx} + f \frac{dv_x}{dx} \)

WENO methods avoid overshoots (Gibbs phenomena) at sharp gradients such as shocks, but the simple 1st-order method has very large artificial diffusion. WENO schemes are a development of the ENO reconstruction schemes which combine good handling of sharp-gradient regions with high accuracy in smooth regions.

The stencil based methods are based by a kernel that combines the data in a stencil to produce a single BoutReal (note upwind/flux methods take extra information about the flow, either a BoutReal or another stencil). It is not anticipated that the user would wish to apply one of these kernels directly so documentation is not provided here for how to do so. If this is of interest please look at `include/bout/index_derivs.hxx`. Internally, these kernel routines are combined within a functor struct that uses a BOUT_FOR loop over the domain to provide a routine that will apply the kernel to every point, calculating the derivative everywhere. These routines are registered in the appropriate DerivativeStore and identified by the direction of differential, the staggering, the type (central/upwind/flux) and a key such as “C2”.

The typical user does not need to interact with this store, instead one can add the following to the top of your physics module:

```cpp
#include <derivs.hxx>
```

to provide access to the following routines. These take care of selecting the appropriate method from the store and ensuring the input/output field locations are compatible.

<table>
<thead>
<tr>
<th>Function</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDX(f)</td>
<td>( \frac{\partial f}{\partial x} )</td>
</tr>
<tr>
<td>DDY(f)</td>
<td>( \frac{\partial f}{\partial y} )</td>
</tr>
<tr>
<td>DDZ(f)</td>
<td>( \frac{\partial f}{\partial z} )</td>
</tr>
<tr>
<td>D2DX(f)</td>
<td>( \frac{\partial^2 f}{\partial x^2} )</td>
</tr>
<tr>
<td>D2DY(f)</td>
<td>( \frac{\partial^2 f}{\partial y^2} )</td>
</tr>
<tr>
<td>D2DZ(f)</td>
<td>( \frac{\partial^2 f}{\partial z^2} )</td>
</tr>
<tr>
<td>D2DX(f, g)</td>
<td>( f \frac{\partial g}{\partial x} )</td>
</tr>
<tr>
<td>D2DY(f, g)</td>
<td>( f \frac{\partial g}{\partial y} )</td>
</tr>
<tr>
<td>VDDX(f, g)</td>
<td>( f \frac{\partial g}{\partial z} )</td>
</tr>
<tr>
<td>FDDX(f, g)</td>
<td>( \partial f \frac{\partial g}{\partial x} )</td>
</tr>
<tr>
<td>FDDY(f, g)</td>
<td>( \partial f \frac{\partial g}{\partial y} )</td>
</tr>
<tr>
<td>FDDZ(f, g)</td>
<td>( \partial f \frac{\partial g}{\partial z} )</td>
</tr>
</tbody>
</table>

By default the method used will be the one specified in the options input file (see *Differencing methods*), but most of these methods can take an optional std::string argument (or a DIFF_METHOD argument - to be deprecated), specifying exactly which method to use.
18.2 User registered methods

**Note:** The following may be considered advanced usage.

It is possible for the user to define their own differencing routines, either by supplying a stencil using kernel or writing their own functor that calculates the differential everywhere. It is then possible to register these methods with the derivative store (for any direction, staggering etc.). For examples please look at include/bout/index_derivs.hxx to see how these approaches work.

Here is a verbose example showing how the C2 method is implemented.

```
DEFINE_STANDARD_DERIV(DDX_C2, "C2", 1, DERIV::Standard) {
    return 0.5*(f.p - f.m);
}
```

Here DEFINE_STANDARD_DERIV is a macro that acts on the kernel `return 0.5*(f.p - f.m);` and produces the functor that will apply the differencing method over an entire field. The macro takes several arguments:

- the first (DDX_C2) is the name of the generated functor – this needs to be unique and allows advanced users to refer to a specific derivative functor without having to go through the derivative store if desired.
- the second ("C2") is the string key that is used to refer to this specific method when registering/retrieving the method from the derivative store.
- the third (1) is the number of guard cells required to be able to use this method (i.e. here the stencil will consist of three values – the field at the current point and one point either side). This can be 1 or 2.
- the fourth (DERIV::Standard) identifies the type of method - here a central method.

Alongside DEFINE_STANDARD_DERIV there’s also DEFINE_UPWIND_DERIV, DEFINE_FLUX_DERIV and the staggered versions DEFINE_STANDARD_DERIV_STAGGERED, DEFINE_UPWIND_DERIV_STAGGERED and DEFINE_FLUX_DERIV_STAGGERED.

To register this method with the derivative store in X and Z with no staggering for both field types we can then use the following code:

```
produceCombinations<Set<WRAP_ENUM(DIRECTION, X), WRAP_ENUM(DIRECTION, Z)>>,
    Set<WRAP_ENUM(STAGGER, None)>>,
    Set<TypeContainer<Field2D, Field3D>>,
    Set<DDX_C2>>
someUniqueNameForDerivativeRegistration(registerMethod{});
```

For the common case where the user wishes to register the method in X, Y and Z and for both field types we provide the helper macros, REGISTER_DERIVATIVE and REGISTER_STAGGERED_DERIVATIVE which could be used as REGISTER_DERIVATIVE(DDX_C2).

To simplify matters further we provide REGISTER_STANDARD_DERIVATIVE, REGISTER_UPWIND_DERIVATIVE, REGISTER_FLUX_DERIVATIVE, REGISTER_STANDARD_STAGGERED_DERIVATIVE, REGISTER_UPWIND_STAGGERED_DERIVATIVE and REGISTER_FLUX_STAGGERED_DERIVATIVE macros that can define and register a stencil using kernel in a single step. For example:

```
REGISTER_STANDARD_DERIVATIVE(DDX_C2, "C2", 1, DERIV::Standard) { return 0.5*(f.p-f.m);};
```

Will define the DDX_C2 functor and register it with the derivative store using key "C2" for all three directions and both fields with no staggering.
18.3 Mixed second-derivative operators

Coordinate derivatives commute, as long as the coordinates are globally well-defined, i.e.

\[
\frac{\partial}{\partial x} \left( \frac{\partial}{\partial y} f \right) = \frac{\partial}{\partial y} \left( \frac{\partial}{\partial x} f \right) \\
\frac{\partial}{\partial y} \left( \frac{\partial}{\partial z} f \right) = \frac{\partial}{\partial z} \left( \frac{\partial}{\partial y} f \right) \\
\frac{\partial}{\partial z} \left( \frac{\partial}{\partial x} f \right) = \frac{\partial}{\partial x} \left( \frac{\partial}{\partial z} f \right)
\]

When using paralleltransform = shifted or paralleltransform = fci (see Parallel Transforms) we do not have globally well-defined coordinates. In those cases the coordinate systems are field-aligned, but the grid points are at constant toroidal angle. The field-aligned coordinates are defined locally, on planes of constant \(y\). There are different coordinate systems for each plane. However, within each local coordinate system the derivatives do commute. \(y\)-derivatives are taken in the local field-aligned coordinate system, so mixed derivatives are calculated as

\[
\begin{align*}
D2DXDY(f) &= DDX(DDY(f)) \\
D2DYDZ(f) &= DDZ(DDY(f))
\end{align*}
\]

This order is simpler – the alternative is possible. Using second-order central difference operators for the \(y\)-derivatives we could calculate (not worrying about communications or boundary conditions here)

```cpp
Field3D D2DXDY(Field3D f) {
    auto result{emptyFrom(f)};
    auto& coords = *f.getCoordinates();

    auto dfdx_yup = DDX(f.yup());
    auto dfdx_ydown = DDX(f.ydown());

    BOUT_FOR(i, f.getRegion()) {
        result[i] = (dfdx_yup[i.yp()] - dfdx_ydown[i.ym()]) / (2. * coords.dy[i])
    }

    return result;
}
```

This would give equivalent results to the previous form\(^1\) as \(\text{yup} \) and \(\text{ydown} \) give the values of \(f\) one grid point along the magnetic field in the local field-aligned coordinate system.

The \(x-z\) derivative is unaffected as it is taken entirely on a plane of constant \(y\) anyway. It is evaluated as

\[
D2DXDZ(f) = DDZ(DDX(f))
\]

As the \(z\)-direction is periodic and the \(z\)-grid is not split across processors, \(DDZ\) does not require any guard cells. By taking \(DDZ\) second, we do not have to communicate or set boundary conditions on the result of \(DDX\) or \(DDY\) before taking \(DDZ\).

The derivatives in \(D2DXDY(f)\) are applied in two steps. First \(dfdy = DDY(f)\) is calculated; \(dfdy\) is communicated and has a boundary condition applied so that all the \(x\)-guard cells are filled. The boundary condition is free_o3 by default (3rd order extrapolation into the boundary cells), but can be specified with the fifth argument to \(D2DXDY\) (see Boundary conditions for possible options). Second \(DDX(dfdy)\) is calculated, and returned from the function.

---

\(^1\) Equivalent but not exactly the same numerically. Expanding out the derivatives in second-order central-difference form shows that the two differ in the grid points at which they evaluate \(dx\) and \(dy\). As long as the grid spacings are smooth this should not affect the order of accuracy of the scheme (7).
18.4 Non-uniform meshes

event:examples/test-nonuniform seems to not work? Setting non_uniform = true in the BOUT.inp options file enables corrections to second derivatives in X and Y. This correction is given by writing derivatives as:

\[
\frac{\partial f}{\partial x} \approx \frac{1}{\Delta x} \frac{\partial f}{\partial i}
\]

where \( i \) is the cell index number. The second derivative is therefore given by

\[
\frac{\partial^2 f}{\partial x^2} \approx \frac{1}{\Delta x^2} \frac{\partial^2 f}{\partial i^2} + \frac{1}{\Delta x} \frac{\partial}{\partial x} \left( \frac{1}{\Delta x} \right)
\]

The correction factor \( \partial/\partial i(1/\Delta x) \) can be calculated automatically, but you can also specify d2x in the grid file which is

\[
d2x = \frac{\partial \Delta x}{\partial i} = \frac{\partial^2 x}{\partial i^2}
\]

The correction factor is then calculated from d2x using

\[
\frac{\partial}{\partial i} \left( \frac{1}{\Delta x} \right) = -\frac{1}{\Delta x^2} \frac{\partial \Delta x}{\partial i}
\]

Note: There is a separate switch in the Laplacian inversion code, which enables or disables non-uniform mesh corrections.

18.5 General operators

These are differential operators which are for a general coordinate system.

\[
\begin{align*}
\mathbf{v} &= \nabla f & \text{Vector} &= \text{Grad(Field)} \\
f &= \nabla \cdot \mathbf{a} & \text{Field} &= \text{Div(Vector)} \\
\mathbf{v} &= \nabla \times \mathbf{a} & \text{Vector} &= \text{Curl(Vector)} \\
f &= \mathbf{v} \cdot \nabla g & \text{Field} &= \text{V_dot_Grad(Vector, Field)} \\
v &= \mathbf{a} \cdot \nabla c & \text{Vector} &= \text{V_dot_Grad(Vector, Vector)} \\
f &= \nabla^2 f & \text{Field} &= \text{Laplace(Field)}
\end{align*}
\]

\[
\nabla \phi = \frac{\partial \phi}{\partial u^i} \nabla u^i \rightarrow (\nabla \phi)_i = \frac{\partial \phi}{\partial u^i}
\]

\[
\nabla \cdot A = \frac{1}{J} \frac{\partial}{\partial u^i} (J g^{ij} A_j)
\]

\[
\nabla^2 \phi = G^j \frac{\partial \phi}{\partial u^i} + g^{ij} \frac{\partial^2 \phi}{\partial u^i \partial u^j}
\]

where we have defined

\[
G^j = \frac{1}{J} \frac{\partial}{\partial u^i} (J g^{ij})
\]

not to be confused with the Christoffel symbol of the second kind (see the coordinates manual for more details).
18.6 Clebsch operators

Another set of operators assume that the equilibrium magnetic field is written in Clebsch form as

$$B_0 = \nabla z \times \nabla x \quad B_0 = \sqrt{g_{yy}}$$

where

$$B_0 = |B_0| b_0 = B_0 b_0$$

is the background equilibrium magnetic field.

<table>
<thead>
<tr>
<th>Function</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grad_par</td>
<td>$\phi_0 = b_0 \cdot \nabla = \frac{1}{\sqrt{g_{yy}}} \frac{\partial}{\partial y}$</td>
</tr>
<tr>
<td>Div_par</td>
<td>$\nabla \phi_0 = B_0 (\frac{1}{g_{yy}} \frac{\partial}{\partial y} + \frac{1}{g_{yy}} \frac{\partial^2}{\partial y^2})$</td>
</tr>
<tr>
<td>Grad2_par2</td>
<td>$\phi = \nabla \cdot (b_0 b_0 \cdot \nabla \phi = \frac{1}{\sqrt{g_{yy}}} \frac{\partial}{\partial y} + \frac{1}{g_{yy}} \frac{\partial^2}{\partial y^2})$</td>
</tr>
<tr>
<td>Laplace_par</td>
<td>$\nabla^2 \phi = \nabla \cdot (b_0 b_0 \cdot \nabla \phi = \frac{1}{\sqrt{g_{yy}}} \frac{\partial}{\partial y} + \frac{1}{g_{yy}} \frac{\partial^2}{\partial y^2})$</td>
</tr>
<tr>
<td>Laplace_perp</td>
<td>$\nabla^2 = \nabla^2 - \nabla^2 \parallel$</td>
</tr>
<tr>
<td>Delp2</td>
<td>Perpendicular Laplacian, neglecting all $y$ derivatives. The Laplacian solver performs the inverse operation</td>
</tr>
<tr>
<td>brackets</td>
<td>Poisson brackets. The Arakawa option, neglects the parallel $y$ derivatives if $g_{xy}$ and $g_{yz}$ are non-zero</td>
</tr>
</tbody>
</table>

We have that

$$b_0 \cdot \nabla \phi \times \nabla A = \frac{1}{J \sqrt{g_{yy}}} [(g_{yy} \frac{\partial \phi}{\partial z} - g_{yz} \frac{\partial \phi}{\partial y} \frac{\partial A}{\partial x} + (g_{yz} \frac{\partial \phi}{\partial y} - g_{xy} \frac{\partial \phi}{\partial x}) \frac{\partial A}{\partial y} + (g_{xy} \frac{\partial \phi}{\partial x} - g_{yx} \frac{\partial \phi}{\partial y}) \frac{\partial A}{\partial z}]$$

$$\nabla \perp \equiv \nabla - b (b \cdot \nabla)$$

$$b = \frac{1}{JB} e_y = \frac{1}{JB} [g_{xy} \nabla x + g_{yy} \nabla y + g_{yz} \nabla z]$$

In a Clebsch coordinate system $B = \nabla z \times \nabla x = \frac{1}{2} e_y$, $g_{yy} = e_y \cdot e_y = J^2 B^2$, and so the $\nabla y$ term cancels out:

$$\nabla \perp = \nabla x \left( \frac{\partial}{\partial x} - \frac{g_{xy}}{(JB)^2 \partial y} \right) + \nabla z \left( \frac{\partial}{\partial z} - \frac{g_{yz}}{(JB)^2 \partial y} \right)$$

18.7 The bracket operators

The bracket operator $\text{brackets}(\phi, f, \text{method})$ aims to differentiate equations on the form

$$- \frac{\nabla \phi \times b}{B} \cdot \nabla f$$

Notice that when we use the Arakawa scheme, $y$-derivatives are neglected if $g_{xy}$ and $g_{yz}$ are non-zero. An example of usage of the brackets can be found in for example examples/MMS/advection or examples/blob2d.
18.8 Finite volume, conservative finite difference methods

These schemes aim to conserve the integral of the advected quantity over the domain. If \( f \) is being advected, then

\[
\sum_i (fJdxdydz)_i = \text{const}
\]

is conserved, where the index \( i \) refers to cell index. This is done by calculating fluxes between cells: Whatever leaves one cell is added to another. There are several caveats to this:

- Boundary fluxes can still lead to changes in the total, unless no-flow boundary conditions are used
- When using an implicit time integration scheme, such as the default PVODE/CVODE, the total is not guaranteed to be conserved, but may vary depending on the solver tolerances.
- There will always be a small rounding error, even with double precision.

The methods can be used by including the header:

```cpp
#include "bout/fv_ops.hxx"
```

Note The methods are defined in a namespace `FV`. Some methods (those with templates) are defined in the header, but others are defined in `src/mesh/fv_ops.cxx`.

18.8.1 Parallel divergence `Div_par`

This function calculates the divergence of a flow in \( y \) (parallel to the magnetic field) by a given velocity.

```cpp
template<typename CellEdges = MC>
const Field3D Div_par(const Field3D &f_in, const Field3D &v_in, const Field3D &a, bool fixflux=true);
```

where \( f_{\text{in}} \) is the quantity being advected (e.g. density), \( v_{\text{in}} \) is the parallel advection velocity. The third input, \( a \), is the maximum wave speed, which multiplies the dissipation term in the method.

```cpp
ddt(n) = -FV::Div_par(n, v, cs);
```

By default the `MC` slope limiter is used to calculate cell edges, but this can be changed at compile time e.g:

```cpp
ddt(n) = -FV::Div_par<FV::Fromm>(n, v, cs);
```

A list of available limiters is given in section `Slope limiters` below.

Example and convergence test

The example code `examples/finite-volume/fluid/` solves the Euler equations for a 1D adiabatic fluid, using `FV::Div_par()` for the advection terms.

\[
\frac{\partial n}{\partial t} + \nabla_{||} (nv_{||}) = 0
\]

\[
\frac{\partial p}{\partial t} + \nabla_{||} (pv_{||}) = -(\gamma - 1)p
\nabla_{||} v_{||}
\]

\[
\frac{\partial}{\partial t} (nv_{||}) + \nabla_{||} (nv_{||}v_{||}) = -\partial_{||} p
\]
where \( n \) is the density, \( p \) is the pressure, and \( nv_\parallel \) is the momentum in the direction parallel to the magnetic field. The operator \( \nabla_\parallel \) represents the divergence of a parallel flow (Div_par), and \( \partial_\parallel = b \cdot \nabla \) is the gradient in the parallel direction.

There is a convergence test using the Method of Manufactured Solutions (MMS) for this example. See section Method of Manufactured Solutions for details of the testing method. Running the \texttt{runtest} script should produce the graph

![Graph showing \( l^2 \) (RMS) and \( l^\infty \) (maximum) error for the evolving fields \( n \) (density), \( p \) (pressure) and \( nv \) (momentum). All fields are shown to converge at the expected second order accuracy.]

**18.8.2 Parallel diffusion**

The parallel diffusion operator calculates \( \nabla_\parallel \left[ k \partial_\parallel (f) \right] \)

```cpp
const Field3D Div_par_K_Grad_par(const Field3D &k, const Field3D &f, bool bndry_flux=true);
```

This is done by calculating the flux \( k\partial_\parallel (f) \) on cell boundaries using central differencing.
18.8.3 Advection in 3D

This operator calculates $\nabla \cdot (n \mathbf{v})$ where $\mathbf{v}$ is a 3D vector. It is written in flux form by discretising the expression

$$\nabla \cdot (\mathbf{A}) = \frac{1}{J} \partial_i (JA^i)$$

Like the $\text{Div}_\text{par}$ operator, a slope limiter is used to calculate the value of the field $n$ on cell boundaries. By default this is the MC method, but this can be set as a template parameter.

```
template<typename CellEdges = MC>
const Field3D Div_f_v(const Field3D &n, const Vector3D &v, bool bndry_flux)
```

18.8.4 Slope limiters

Here limiters are implemented as slope limiters: The value of a given quantity is calculated at the faces of a cell based on the cell-centre values. Several slope limiters are defined in `fv_ops.hxx`:

- **Upwind** - First order upwinding, in which the left and right edges of the cell are the same as the centre (zero slope).
- **Fromm** - A second-order scheme which is a fixed weighted average of upwinding and central difference schemes.
- **MinMod** - This second order scheme switches between the upwind and downwind gradient, choosing the one with the smallest absolute value. If the gradients have different signs, as at a maximum or minimum, then the method reverts to first order upwinding (zero slope).
- **MC** (Monotonised Central) is a second order scheme which switches between central, upwind and downwind differencing in a similar way to **MinMod**. It has smaller dissipation than **MinMod** so is the default.

18.8.5 Staggered grids

By default, all quantities in BOUT++ are defined at cell centre, and all derivative methods map cell-centred quantities to cell centres. Switching on staggered grid support in `BOUT.inp`:

```
StaggerGrids = true
```

allows quantities to be defined on cell boundaries. Functions such as `DDX` now have to handle all possible combinations of input and output locations, in addition to the possible derivative methods.

Several things are not currently implemented, which probably should be:

- Only 3D fields currently have a cell location attribute. The location (cell centre etc) of 2D fields is ignored at the moment. The rationale for this is that 2D fields are assumed to be slowly-varying equilibrium quantities for which it won’t matter so much. Still, needs to be improved in future

- Twist-shift and X shifting still treat all quantities as cell-centred.

- No boundary condition functions yet account for cell location.

Currently, BOUT++ does not support values at cell corners; values can only be defined at cell centre, or at the lower X,Y, or Z boundaries. This is

Once staggered grids are enabled, two types of stencil are needed: those which map between the same cell location (e.g. cell-centred values to cell-centred values), and those which map to different locations (e.g. cell-centred to lower X).
Central to Central or Lower to Lower

![Staggered stencils](image)

Fig. 18.2: Stencils with cell-centred (solid) and lower shifted values (open). Processor boundaries marked by vertical dashed line

Central differencing using 4-point stencil:

\[
\begin{align*}
\frac{\partial y}{\partial x} &= \frac{9y_{-1/2} + 9y_{1/2} - y_{-3/2} - y_{3/2}}{16} \\
\frac{\partial^2 y}{\partial x^2} &= \frac{(y_{3/2} + y_{-3/2} - y_{1/2} - y_{-1/2})}{2\Delta x^2}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CENTRE</td>
<td>XLOW</td>
<td>Lower staggered stencil</td>
</tr>
<tr>
<td>XLOW</td>
<td>CENTRE</td>
<td>Upper staggered stencil</td>
</tr>
<tr>
<td>XLOW</td>
<td>Any</td>
<td>Staggered stencil to CENTRE, then interpolate</td>
</tr>
<tr>
<td>CENTRE</td>
<td>Any</td>
<td>Central stencil, then interpolate</td>
</tr>
<tr>
<td>Any</td>
<td>Any</td>
<td>Interpolate to centre, use central stencil, then interpolate</td>
</tr>
</tbody>
</table>

Table: DDX actions depending on input and output locations. Uses first match.

### 18.9 Derivatives of the Fourier transform

By using the definition of the Fourier transformed, we have

\[
F(x, y, \xi) = \int_{-\infty}^{\infty} f(x, y, z) \exp(-2\pi i z \xi) \, dz
\]

this gives

\[
\begin{align*}
\int_{-\infty}^{\infty} (\partial_z f[x, y, z]) \exp(-2\pi i z \xi) \, dz &= \int_{-\infty}^{\infty} \partial_z (f[x, y, z] \exp(-2\pi i z \xi)) \, dz - \int_{-\infty}^{\infty} f(x, y, z) \partial_z \exp(-2\pi i z \xi) \, dz \\
&= (f[x, y, z] \exp(-2\pi i z \xi)) \bigg|_{-\infty}^{\infty} - (2\pi i \xi) \int_{-\infty}^{\infty} f(x, y, z) \exp(-2\pi i z \xi) \, dz \\
&= 2\pi i \xi F(x, y, \xi)
\end{align*}
\]
where we have used that \( f(x, y, \pm \infty) = 0 \) in order to have a well defined Fourier transform. This means that

\[
\partial_z^n F(x, y, \xi) = (2\pi i \xi)^n F(x, y, \xi)
\]

In our case, we are dealing with periodic boundary conditions. Strictly speaking, the Fourier transform does not exist in such cases, but it is possible to define a Fourier transform in the limit which in the end lead to the Fourier series\(^2\) By discretising the spatial domain, it is no longer possible to represent the infinite amount of Fourier modes, but only \( N + 1 \) number of modes, where \( N \) is the number of points (this includes the modes with negative frequencies, and the zeroth offset mode). For the discrete Fourier transform, we have

\[
F(x, y)_k = \frac{1}{N} \sum_{Z=0}^{N-1} f(x, y)_Z \exp(-\frac{2\pi ikZ}{N})
\]  

(18.2)

where \( k \) is the mode number, \( N \) is the number of points in \( z \). If we call the sampling points of \( z \) for \( z_Z \), where \( Z = 0, 1 \ldots N - 1 \), we have that \( z_Z = Z \Delta z \). As our domain goes from \([0, 2\pi]\), we have that (since we have one less line segment than point) \( \Delta z(N - 1) = L_z = 2\pi - \Delta z \), which gives \( \Delta z = \frac{2\pi}{N} \). Inserting this in equation (18.2) yields

\[
F(x, y)_k = \frac{1}{N} \sum_{Z=0}^{N-1} f(x, y)_Z \exp(-ikZ\Delta z) = \frac{1}{N} \sum_{Z=0}^{N-1} f(x, y)_Z \exp(-ikz_Z)
\]

The discrete version of equation (18.1)) thus gives

\[
\partial_z^n F(x, y)_k = (ik)^n F(x, y)_k
\]

---

\(^2\) For more detail see Bracewell, R. N. - The Fourier Transform and Its Applications 3rd Edition chapter 10
BOUT++ provides a wide variety of algebraic operators acting on fields.

The algebraic operators are listed in Table 19.1. For a completely up-to-date list, see the Non-member functions part of field2d.hxx, field3d.hxx, fieldperp.hxx.

Table 19.1: Algebraic operators

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>min(f, allpe=true, region)</td>
<td>Minimum (optionally over all processes)</td>
</tr>
<tr>
<td>max(f, allpe=true, region)</td>
<td>Maximum (optionally over all processes)</td>
</tr>
<tr>
<td>pow(lhs, rhs, region)</td>
<td>lhs^{rhs}</td>
</tr>
<tr>
<td>sqrt(f, region)</td>
<td>\sqrt(f)</td>
</tr>
<tr>
<td>abs(f, region)</td>
<td></td>
</tr>
<tr>
<td>exp(f, region)</td>
<td>e^{f}</td>
</tr>
<tr>
<td>log(f, region)</td>
<td>\log(f)</td>
</tr>
<tr>
<td>sin(f, region)</td>
<td>sin(f)</td>
</tr>
<tr>
<td>cos(f, region)</td>
<td>cos(f)</td>
</tr>
<tr>
<td>tan(f, region)</td>
<td>tan(f)</td>
</tr>
<tr>
<td>sinh(f, region)</td>
<td>sinh(f)</td>
</tr>
<tr>
<td>cosh(f, region)</td>
<td>cosh(f)</td>
</tr>
<tr>
<td>tanh(f, region)</td>
<td>tanh(f)</td>
</tr>
<tr>
<td>floor(f, region)</td>
<td>Returns a field with the floor of f at each point</td>
</tr>
<tr>
<td>filter(f, n, region)</td>
<td>Calculate the amplitude of the Fourier mode in the z-direction with mode number n</td>
</tr>
<tr>
<td>lowpass(f, nmax, region)</td>
<td>Remove Fourier modes (in the z-direction) with mode number higher than zmax</td>
</tr>
<tr>
<td>lowpass(f, nmax, nmin, region)</td>
<td>Remove Fourier modes (in the z-direction) with mode number higher than zmax or lower than zmin</td>
</tr>
<tr>
<td>shiftZ(f, angle, region)</td>
<td>Rotate f by angle in the z-direction. angle/2\pi is the fraction of the domain multiplied by 2\pi so angle is in radians if the total size of the domain is 2\pi</td>
</tr>
<tr>
<td>DC(f, region)</td>
<td>The average in the z-direction of f (DC stands for direct current, i.e. the constant part of f as opposed to the AC, alternating current, or fluctuating part)</td>
</tr>
</tbody>
</table>

These operators take a region argument, whose values can be¹ (see Iterating over fields)

¹ More regions may be added in future, for example to act on only subsets of the physical domain.
BOUT++ Documentation, Release 4.4.0

- **RGN_ALL**, which is the whole mesh;
- **RGN_NOBNDRY**, which skips all boundaries;
- **RGN_NOX**, which skips the x boundaries
- **RGN_NOY**, which skips the y boundaries

The default value for the region argument is **RGN_ALL** which should work in all cases. However, the region argument can be used for optimization, to skip calculations in guard cells if it is known that those results will not be needed (for example, if no derivatives of the result will be calculated). Since these operators can be relatively expensive compared to addition, subtraction, multiplication this can be a useful performance improvement.
CHAPTER
TWENTY

STAGGERED GRIDS

Until now all quantities have been cell-centred i.e. both velocities and conserved quantities were defined at the same locations. This is because these methods are simple and this was the scheme used in the original BOUT. This class of methods can however be susceptible to grid-grid oscillations, and so most shock-capturing schemes involve densities and velocities (for example) which are not defined at the same location: their grids are staggered.

By default BOUT++ runs with all quantities at cell centre. To enable staggered grids, set:

```
StaggerGrids = true
```

in the top section of the `BOUT.inp` file. The test-staggered example illustrates how to use staggered grids in BOUT++.

There are four possible locations in a grid cell where a quantity can be defined in BOUT++: centre, lower X, lower Y, and lower Z. These are illustrated in Fig. 20.1.

To specify the location of a variable, use the method `Field3D::setLocation()` with one of the `CELL_LOC` locations `CELL_CENTRE`, `CELL_XLOW`, `CELL_YLOW`, or `CELL_ZLOW`.

The key lines in the staggered_grid example which specify the locations of the evolving variables are:

```
Field3D n, v;

int init(bool restart) {
    v.setLocation(CELL_YLOW);  // Staggered relative to n
    SOLVE_FOR(n, v);
    ...
}
```

which makes the velocity v staggered to the lower side of the cell in Y, whilst the density n remains cell centred.

**Note:** If BOUT++ was configured --with-checks, `Field3D::setLocation()` will throw an exception if you don’t have staggered grids turned on and try to set the location to something other than `CELL_CENTRE`. If you want to be able to run your model with and without staggered grids, you should do something like:

```
if (v.getMesh()->StaggerGrids) {
    v.setLocation(CELL_YLOW);
}
```

Compiling BOUT++ with checks turned off will instead cause `Field3D::setLocation()` to silently set the location to `CELL_CENTRE` if staggered grids are off, regardless of what you pass it.

Arithmetic operations can only be performed between variables with the same location. When performing a calculation at one location, to include a variable from a different location, use the interpolation routines. Include the header file
Fig. 20.1: The four possible cell locations for defining quantities
then use the `interp_to(field, location, region)` function. For example, given a `CELL_CENTRE` field \( n \) and a `CELL_YLOW` field \( v \), to calculate \( n \times v \) at `CELL_YLOW`, call `interp_to(n, CELL_YLOW) \times v` whose result will be `CELL_YLOW` as \( n \) is interpolated.

**Note:** The `region` argument is optional but useful (see *Iterating over fields* for more on regions). The default `RGN_ALL` reproduces the historical behaviour of BOUT++, which communicates before returning the result from `interp_to`. Communication is necessary because the result of interpolation in the guard cells depends on data from another process (except, currently, in the case of interpolation in the z-direction which can be done without communication because all the z-points are on the same process).

Using `RGN_NOBNDRY` no communication is performed (so `interp_to` is faster, potentially significantly faster when using many processes) and all the guard cells are invalid. Whichever region is used, the boundary guard cells are invalid since no boundary condition is applied in `interp_to`. If the guard cells are needed (e.g. to calculate a derivative) a boundary condition must be applied explicitly to the result.

`RGN_NOX` and `RGN_NOY` currently have identical behaviour to `RGN_ALL` because at present BOUT++ has no functions for single-direction communication which could in principle be used in these cases (if the combination of region and direction of interpolation allows it). \( x \)- or \( y \)-interpolation can never be calculated in guard cells without communication because the corner guard cells are never valid.

Differential operators by default return fields which are defined at the same location as their inputs, so here \( \text{Grad}_{\text{par}}(v) \) would be `CELL_YLOW`. If this is not what is wanted, give the location of the result as an additional argument: \( \text{Grad}_{\text{par}}(v, \text{CELL}_\text{CENTRE}) \) uses staggered differencing to produce a result which is defined at the cell centres. It is an error to ask for the result to be staggered in a different direction from the input as the best that could be done would be to calculate output at `CELL_CENTRE` and then interpolate this to the requested location, but the interpolation would in general require boundary conditions to be applied first.

Advection operators which take two arguments return a result which is defined at the location of the field being advected. For example \( \text{Vpar}_{\text{Grad}_{\text{par}}}(v, \ell) \) calculates \( v \nabla_{\parallel} \ell \) and returns a result at the same location as \( \ell \). If \( v \) and \( \ell \) are defined at the same locations then centred differencing is used, if one is centred and the other staggered then staggered differencing is used; it is an error for both to be staggered to different locations. As with other differential operators, the required location of the result can be given as an optional argument, but at least for now it is an error for this to be different from the location of the field being advected (\( \ell \) here).

Laplace solvers (see *Laplacian inversion*) also need a location to be set in order not to operate at `CELL_CENTRE`: this allows the solver to check the locations of coefficients and right-hand-side which are passed to it, and to return a result at the correct location. For example, in an electromagnetic case with staggered grids, the solver for the magnetic vector potential \( A_{\parallel} \) is probably defined on the staggered grid. The location is set by the second optional argument to `Laplacian::create()`, after the options. For example:

```cpp
aparSolver = Laplacian::create(&options["apar_solver"], CELL_YLOW);
```
By using the SLEPc library, BOUT++ can be used as an eigenvalue solver to find the eigenvectors and eigenvalues of sets of equations.

21.1 Configuring with SLEPc

The BOUT++ interface has been tested with SLEPc version 3.4.3, itself compiled with PETSc 3.4.2. SLEPc version 3.4 should work, but other versions will not yet.

21.2 SLEPc options

Time derivatives can be taken directly from the RHS function, or by advancing the simulation in time by a relatively large increment. This second method acts to damp high frequency components

21.3 Examples

21.3.1 Wave in a box

examples/eigen-box
NONLOCAL HEAT FLUX MODELS

22.1 Spitzer-Harm heat flux

The Spitzer-Harm heat flux $q_{SH}$ is calculated using

$$ q_{SH} = - \frac{n_e e T_e}{m_e} \frac{3\sqrt{\pi}}{4} \tau_{e,i} \kappa_0 \frac{Z + 0.24}{Z + 4.2} \partial_|| T_e $$

where $n_e$ is the electron density in $m^{-3}$, $T_e$ is the electron temperature in eV, $\kappaappa = 13.58$, $Z$ is the average ion charge. The resulting expression is in units of $eV/m^2/s$.

The thermal collision time $\tau_{e,i} = \lambda_{e,i} / v_T$ is calculated using the thermal mean free path and thermal velocity:

$$ \lambda_{e,e} = \frac{v_T^4}{Y n_e \ln \Lambda} $$
$$ \lambda_{e,i} = \frac{v_T^4}{Y Z^2 n_i \ln \Lambda} $$
$$ v_T = \sqrt{\frac{2e T_e}{m_e}} $$

where it is assumed that $n_i = n_e$, and the following are used:

$$ Y = 4\pi \left( \frac{e^2}{4\pi\kappaappa m_e} \right)^2 $$
$$ \ln \Lambda = 6.6 - 0.5 \log \left( \frac{n_e}{10^{20}} \right) + 1.5 \log (T_e) $$

Note: If comparing to online notes, $\kappaappa_0 \frac{Z + 0.24}{Z + 4.2} \approx 3.2$, a different definition of collision time $\tau_{e,i}$ is used here, but the other factors are included so that the heat flux $q_{SH}$ is the same here as in those notes.

22.2 SNB model

The SNB model calculates a correction to the Spitzer-Harm heat flux, solving a diffusion equation for each of a set of energy groups with normalised energy $\beta = E_g / e T_e$ where $E_g$ is the energy of the group.

$$ \frac{1}{\lambda'_{g,ee}} - \nabla_|| \left( \frac{\lambda'_{g,ei}}{3} \partial_|| \right) H_g = -\nabla_|| U_g $$

where $\nabla_||$ is the divergence of a parallel flux, and $\partial_||$ is a parallel gradient. $U_g = W_g q_{SH}$ is the contribution to the Spitzer-Harm heat flux from a group:

$$ W_g = \frac{1}{24} \int_{\beta_{g-1}}^{\beta_{g+1}} \beta^4 e^{-\beta} d\beta $$
The modified mean free paths for each group are:

\[ \lambda'_{g,ee} = \beta^2 \lambda_{ee,T} \frac{Z + 0.24}{Z + 4.2} \]

\[ \lambda'_{g,ei} = \beta^2 \lambda_{ei,T} Z + 0.24 \]

From the quantities \( H_g \) for each group, the SNB heat flux is:

\[ q_{SNB} = q_{SH} - \sum_g \frac{\lambda_{g,ei}}{3} \nabla H_g \]

In fluid models we actually want the divergence of the heat flux, rather than the heat flux itself. We therefore rearrange to get:

\[ \nabla \| \left( \frac{\lambda'_{g,ei}}{3} \right) H_g = \nabla \| U_g + H_g/\lambda'_{g,ee} \]

and so calculate the divergence of the heat flux as:

\[ \nabla \| q_{SNB} = \nabla \| q_{SH} - \sum_g \left( \nabla \| U_g + H_g/\lambda'_{g,ee} \right) \]

The Helmholtz type equation along the magnetic field is solved using a tridiagonal solver. The parallel divergence term is currently split into a second derivative term, and a first derivative correction:

\[ \nabla \| (k \partial \| T) = \frac{1}{J} \frac{\partial}{\partial y} \left( \frac{kJ \partial T}{g_{22}} \right) = k \frac{1}{g_{22}^2} \frac{\partial^2 T}{\partial y^2} + \frac{1}{J} \frac{\partial}{\partial y} \left( \frac{kJ}{g_{22}} \right) \frac{\partial T}{\partial y} \]

### 22.2.1 Using the SNB model

To use the SNB model, first include the header:

```
#include <bout/snb.hxx>
```

then create an instance:

```
HeatFluxSNB snb;
```

By default this will use options in a section called “snb”, but if needed a different `Options&` section can be given to the constructor:

```
HeatFluxSNB snb(Options::root()["mysnb"]);
```

The options are listed in table `tab-snb-options`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>beta_max</td>
<td>Maximum energy group to consider (multiple of eT)</td>
<td>10 40 2</td>
</tr>
<tr>
<td>ngroups r</td>
<td>Number of energy groups Scaling down the electron-electron mean free path</td>
<td></td>
</tr>
</tbody>
</table>

The divergence of the heat flux can then be calculated:
Field3D Div_q = snb.divHeatFlux(Te, Ne);

where \( T_e \) is the temperature in eV, and \( N_e \) is the electron density in \( m^{-3} \). The result is in eV per \( m^3 \) per second, so multiplying by \( e = 1.602 \times 10^{-19} \) will give Watts per cubic meter.

To compare to the Spitzer-Harm result, pass in a pointer to a Field3D as the third argument. This field will be set to the Spitzer-Harm value:

Field3D Div_q_SH;
Field3D Div_q = snb.divHeatFlux(Te, Ne, &Div_q_SH);

This is used in the examples discussed below.

### 22.2.2 Example: Linear perturbation

The examples/conduction-snb example calculates the heat flux for a given density and temperature profile, comparing the SNB and Spitzer-Harm fluxes. The sinusoidal.py case uses a periodic domain of length 1 meter and a small (0.01eV) perturbation to the temperature. The temperature is varied from 1eV to 1keV, so that the mean free path varies. This is done for different SNB settings, changing the number of groups and the maximum \( \beta \):

\`
$ python sinusoid.py
\`

This should output a file snb-sinusoidal.png and display the results, shown in figure Fig. 22.1.

### 22.2.3 Example: Nonlinear heat flux

A nonlinear test is also included in examples/conduction-snb, a step function in temperature from around 200eV to 950eV over a distance of around 0.1mm, at an electron density of 5e26 per cubic meter:

\`
$ python step.py
\`

This should output a file snb-step.png, shown in figure Fig. 22.2.
Fig. 22.1: The ratio of SNB heat flux to Spitzer-Harm heat flux, as a function of electron mean free path divided by temperature perturbation wavelength. Note that the difference between SNB and Spitzer-Harm becomes significant (20%) when the mean free path is just 1% of the wavelength.
Fig. 22.2: Temperature profile and heat flux calculated using Spitzer-Harm and the SNB model, for a temperature step profile, at a density of $5\times10^{26}$ per cubic meter. Note the reduction in peak heat flux (flux limit) and higher flux in the cold region (preheat) with the SNB model.
A common problem in physics models is solve a matrix equation of the form

\[ A \cdot x = b \]

for the unknown \( x \). Here \( A \) represents some differential operator subject to boundary conditions. A specific example is the set of Laplacian operators described in *Laplacian inversion*.

Whilst specific tools are provided to deal with Laplacian and parallel Helmholtz like equations these do not capture all possible systems and are typically implemented (at least partially) independently of the finite difference representation of the forward operators provided by the rest of BOUT++. To address this a class **InvertableOperator** has been implemented that allows the user to define a generic differential operator and provides a simple (for the user) method to invert the operator to find \( x \). This class currently relies on PETSc to provide the inversion functionality and hence is not available when configuring without PETSc support. It is available in the namespace **bout::inversion**.

There is an example in `examples/invertable_operator` that uses the class to solve a simple Laplacian operator and compares to the specific Laplacian inversion solvers.

The **InvertableOperator** class is templated on the field type of the operator (essentially defining the domain over which the problem exists). To define the operator that the **InvertableOperator** instances represents one should use the **InvertableOperator::setOperatorFunction** method. This takes a function of signature

\[ \text{std::function<T(const T&)>} \]

This can be a \text{std::function}, compatible function pointer, lambda or a functor. The last of these allows more complicated functions that use a local context. For example the following code snippet demonstrates a functor that stores several auxiliary Field3D variables used in the **operator()** call:

```
struct myLaplacian {
    Field3D D = 1.0, C = 1.0, A = 0.0;

    // Drop C term for now
    Field3D operator(const Field3D &input) {
        TRACE("myLaplacian::operator()");
        Timer timer("invertable_operator_operate");
        Field3D result = A * input + D * Delp2(input);

        // Ensure boundary points are set appropriately as given by the input field.
        result.setBoundaryTo(input);

        return result;
    }
};
```

A more complete example is
```cpp
struct myLaplacian {
    Field3D D = 1.0, C = 1.0, A = 0.0;

    // Drop C term for now
    Field3D operator()(const Field3D &input) {
        TRACE("myLaplacian::operator()");
        Timer timer("invertable_operator_operate");
        Field3D result = A * input + D * Delp2(input);

        // Ensure boundary points are set appropriately as given by the input field.
        result.setBoundaryTo(input);
        return result;
    }
};

bout::inversion::InvertableOperator<Field3D> solver;
myLaplacian laplacianOperator;
laplacianOperator.A = 1.0;
laplacianOperator.B = 2.0;

// Set the function defining the operator
solver.setOperatorFunction(laplacianOperator);

// Perform initial setup
solver.setup();

// Now invert the operator for a given right hand side.
Field3D rhs = 3.0*x;
auto solution = solver.invert(rhs);
```

The PETSc backend solver is an iterative solver. It can be controlled through the usual PETSc command line options. Note we define the options prefix here to be `-invertable`, so instead of `-ksp_type` one would use `-invertable_ksp_type` for example.

By default the solver caches the result to use as the initial guess for the next call to `invert`. There is an overload of `invert` that takes a second field, which is used to set the initial guess to use in that call.

The routine `setOperatorFunction` takes the function by value, and hence subsequent changes to the functor will not be reflected in the operator without a further call to `setOperatorFunction`. For example:

```cpp
using bout::inversion;
InvertableOperator<Field3D> solver;
myLaplacian laplacianOperator;
laplacianOperator.A = 1.0;
laplacianOperator.B = 2.0;

// Set the function defining the operator
solver.setOperatorFunction(laplacianOperator);

// Perform initial setup
solver.setup();

// This does not change the operator represented by
```

(continues on next page)
// solver yet.
laplacianOperator.B = -1.0;

// This call updates the function used by solver 
// and hence the operator is update to reflect the state 
// of laplacianOperator. 
solver.setOperatorFunction(laplacianOperator);

The class provides a reportTime method that reports the time spent in various parts of the class. Note that by including Timer timer("invertable_operator_operate"); in the function representing the operator reportTime will include the time spent actually applying the operator.

The class provides both apply and operator() methods that can be used to apply the operator to a field. For example the following should be equivalent to no operation:

// Here result should == input, at least in the main simulation domain 
auto result = solver(solver.invert(input));

The class provides a verify method that checks that applying the operator to the calculated inverse returns the input field within some tolerance.

It’s also possible to register a function to use as a preconditioner. By default this is the same as the full operator function.
Options for PETSc solvers can be passed in the input file (or on the command line). Global options are set in the [petsc] section. To set options specific to a particular PETSc-based solver, the options can be set in a petsc subsection of the solver’s options, e.g. for a LaplaceXY solver (using the default options section) use the [laplacexy:petsc] section. Note that the global options, including any passed on the command line*(0), will be ignored for that solver if the subsection is created. To set options from the command line, it is recommended to use the BOUT++ options system rather than PETSc’s, e.g. ./mymodel laplacexy:petsc:type=gmres.

Any options that can be passed on the command line to PETSc can be set, with no preceding hyphen. Flags passed with no value can be passed as options with no value. So for example, if the command line options would be:

```
-ksp_monitor -ksp_type gmres
```

to set for the LaplaceXY solver, in the input file you would put:

```
[laplacexy:petsc]
ksp_monitor
ksp_type = gmres
```

*(0) The object-specific options are passed to PETSc by creating an object-specific prefix boutpetsclib_<sectionname>, where <sectionname> is the name of the options section used to create the PetscLib. So an option could in principle be passed to a particular solver if you use the section name, e.g.:

```
-boutpetsclib_laplaceyksptype gmres
```

The PETSc arguments -options_view and -options_left might be helpful for this - they will show what options have been set, so will show the prefixes used.
Chapter Twentyfive

Field-Aligned Coordinates

Author B.Dudson§, M.V.Umansky, L.C.Wang, X.Q.Xu, L.L.Lodestro §Department of Physics, University of York, UK Lawrence Livermore National Laboratory, USA IFTS, China

25.1 Introduction

This manual covers the field-aligned coordinate system used in many BOUT++ tokamak models, and useful derivations and expressions.

25.2 Orthogonal toroidal coordinates

Starting with an orthogonal toroidal coordinates system \((\psi, \theta, \zeta)\), where \(\psi\) is the poloidal flux, \(\theta\) the poloidal angle (from 0 to \(2\pi\)), and \(\zeta\) the toroidal angle (also 0 to \(2\pi\)). We have that the magnetic field \(B\) can be expressed as

\[
B = B_\theta \nabla \theta + B_\zeta \nabla \zeta
\]

\[
= B_\theta \hat{e}_\theta + B_\zeta \hat{e}_\zeta
\]

\[
= B_{pol} h_\theta \hat{e}_\theta + B_{tor} R \hat{e}_\zeta
\]

The magnitudes of the unit vectors are

\[
|\hat{e}_\psi| = \frac{1}{R |B_{pol}|} \quad |\hat{e}_\theta| = h_\theta \quad |\hat{e}_\zeta| = R
\]

where \(h_\theta\) is the poloidal arc length per radian. The coordinate system is right handed, so \(\hat{e}_\psi \times \hat{e}_\theta = \hat{e}_\zeta\), \(\hat{e}_\psi \times \hat{e}_\zeta = -\hat{e}_\theta\) and \(\hat{e}_\theta \times \hat{e}_\zeta = \hat{e}_\psi\). The covariant metric coefficients are

\[
g_{\psi\psi} = \frac{1}{(R |B_{pol}|)^2} \quad g_{\theta\theta} = h_\theta^2 \quad g_{\zeta\zeta} = R^2
\]

and the magnitudes of the reciprocal vectors are therefore

\[
|\nabla \psi| = R |B_{pol}| \quad |\nabla \theta| = \frac{1}{h_\theta} \quad |\nabla \zeta| = \frac{1}{R}
\]

Because the coordinate system is orthogonal, \(g^{ii} = 1/g_{ii}\) and so the cross-products can be calculated as

\[
\nabla \psi \times \nabla \theta = \hat{e}_\psi \times \hat{e}_\theta = g_{\psi\psi} e_\psi \times g_{\theta\theta} e_\theta
\]

\[
= g_{\psi\psi} g_{\theta\theta} h_\psi h_\theta \hat{e}_\psi \times \hat{e}_\theta
\]

\[
= \frac{1}{h_\psi h_\theta} \hat{e}_\zeta = \frac{R |B_{pol}|}{h_\theta} \hat{e}_\zeta
\]
Similarly,
\[ \nabla \psi \times \nabla \zeta = -|B_{\text{pol}}| \hat{e}_{\theta} \quad \nabla \theta \times \nabla \zeta = \frac{1}{R \hat{h}_{\theta}} \hat{e}_{\psi} = \frac{1}{h_{\theta} R^2 |B_{\text{pol}}|} \nabla \psi \]

### 25.3 Field-aligned coordinates

In order to efficiently simulate (predominantly) field-aligned structures, grid-points are placed in a field-aligned coordinate system. We define \( \sigma_{B\theta} \equiv B_{\text{pol}}/|B_{\text{pol}}| \) i.e. the sign of the poloidal field. The new coordinates \((x, y, z)\) are defined by:

\[
\begin{align*}
x &= \sigma_{B\theta} (\psi - \psi_0) \quad y = \theta \quad z = \sigma_{B\theta} \left( \zeta - \int_{\theta_0}^{\theta} \nu (\psi, \theta) \, d\theta \right) \quad (25.1)
\end{align*}
\]

Where \( \nu \) is the local field-line pitch given by

\[ \nu (\psi, \theta) = \frac{B \cdot \nabla \zeta}{B \cdot \nabla \theta} = \frac{B_{\text{tor}} h_{\theta}}{B_{\text{pol}} R} = \frac{(F/R) \, h_{\theta}}{B_{\text{pol}} R} = F/R^2 \]

where \( F = B_{\text{tor}} R \) is a function only of \( \psi \) (sometimes called the poloidal current function).

The coordinate system is chosen so that \( x \) increases radially outwards, from plasma to the wall. The sign of the toroidal field \( B_{\text{tor}} \) can then be either + or -.

The contravariant basis vectors are therefore

\[
\begin{align*}
\nabla x &= \sigma_{B\theta} \nabla \psi \quad \nabla y = \nabla \theta \quad \nabla z = \sigma_{B\theta} \left( \nabla \zeta - \left[ \int_{\theta_0}^{\theta} \frac{\partial \nu (\psi, \theta)}{\partial \psi} \, d\theta \right] \nabla \psi - \nu (\psi, \theta) \nabla \theta \right)
\end{align*}
\]

The term in square brackets is the integrated local shear:

\[ I = \int_{y_0}^{y} \frac{\partial \nu (x, y)}{\partial \psi} \, dy \]

### 25.3.1 Magnetic field

Magnetic field is given in Clebsch form by:

\[ B = \nabla \times x = \frac{1}{J} e_y \]

The contravariant components of this are then

\[ B^y = \frac{B_{\text{pol}}}{h_{\theta}} \quad B^x = B^z = 0 \]

i.e. \( B \) can be written as

\[ B = \frac{B_{\text{pol}}}{h_{\theta}} e_y \]

and the covariant components calculated using \( g_{ij} \) as

\[ B_x = \sigma_{B\theta} B_{\text{tor}} I R \quad B_y = \frac{B^2 h_{\theta}}{B_{\text{pol}}} \quad B_z = \sigma_{B\theta} B_{\text{tor}} R \]

The unit vector in the direction of equilibrium \( B \) is therefore

\[ b = \frac{1}{JB} e_y = \frac{1}{JB} [g_{xy} \nabla x + g_{yy} \nabla y + g_{yz} \nabla z] \]
25.3.2 Jacobian and metric tensors

The Jacobian of this coordinate system is

$$J^{-1} \equiv (\nabla x \times \nabla y) \cdot \nabla z = B_{pol}/h_\theta$$

which can be either positive or negative, depending on the sign of $B_{pol}$. The contravariant metric tensor is given by:

$$g^{ij} \equiv e^i \cdot e^j \equiv \nabla u^i \cdot \nabla u^j = \begin{pmatrix}
(RB_{pol})^2 & 0 & -I(RB_{pol})^2 \\
0 & 1/h_\theta^2 & -\sigma_{B\theta\nu}^\nu/h_\theta^2 \\
-I(RB_{pol})^2 & -\sigma_{B\theta\nu}^\nu/h_\theta^2 & I^2(RB_{pol})^2 + B^2/(RB_{pol})^2
\end{pmatrix}$$

and the covariant metric tensor:

$$g_{ij} \equiv e_i \cdot e_j = \begin{pmatrix}
I^2R^2 + 1/(RB_{pol})^2 & \sigma_{B\theta}B_{tor}R/B_{pol} & IR^2 \\
\sigma_{B\theta}B_{tor}h_\theta IR/B_{pol} & B^2h_\theta^2/B_{pol}^2 & \sigma_{B\theta}B_{tor}h_\theta R/B_{pol} \\
IR^2 & \sigma_{B\theta}B_{tor}R/B_{pol} & R^2
\end{pmatrix}$$

25.3.3 Differential operators

The derivative of a scalar field $f$ along the unperturbed magnetic field $b_0$ is given by

$$\partial_0^\parallel f \equiv b_0 \cdot \nabla f = \frac{1}{\sqrt{g_{yy}}} \frac{\partial f}{\partial y} = \frac{B_{pol}}{B_{h_\theta}} \frac{\partial f}{\partial y}$$

whilst the parallel divergence is given by

$$\nabla_0^\parallel f = B_0 \partial_0^\parallel \left( \frac{f}{B_0} \right)$$

Using equation (11), the Laplacian operator is given by

$$\nabla^2 = \frac{\partial^2}{\partial x^2} |\nabla x|^2 + \frac{\partial^2}{\partial y^2} |\nabla y|^2 + \frac{\partial^2}{\partial z^2} |\nabla z|^2 - 2 \frac{\partial^2}{\partial x \partial z} I(RB_{pol})^2 - 2 \frac{\partial^2}{\partial y \partial z} h_\theta^0
+ \frac{\partial}{\partial x} \nabla^2 x + \frac{\partial}{\partial y} \nabla^2 y + \frac{\partial}{\partial z} \nabla^2 z$$

Using equation (10) for $\nabla^2 x = G^x$ etc, the values are

$$\nabla^2 x = \frac{B_{pol}}{h_\theta} \frac{\partial}{\partial x} (h_\theta R^2 B_{pol}) \quad \nabla^2 y = \frac{B_{pol}}{h_\theta} \frac{\partial}{\partial y} \left( \frac{1}{B_{pol} h_\theta} \right)$$

$$\nabla^2 z = -\frac{B_{pol}}{h_\theta} \left[ \frac{\partial}{\partial x} (IR^2 B_{pol} h_\theta) + \frac{\partial}{\partial y} \left( \frac{\nu}{B_{pol} h_\theta} \right) \right]$$

Neglecting some parallel derivative terms, the perpendicular Laplacian can be written:

$$\nabla^2_\perp = (RB_{pol})^2 \left[ \frac{\partial^2}{\partial x^2} - 2I \frac{\partial^2}{\partial z \partial x} + \left( I^2 + \frac{B^2}{(RB_{pol})^4} \right) \frac{\partial^2}{\partial z^2} \right] + \nabla^2 x \frac{\partial}{\partial x} + \nabla^2 z \frac{\partial}{\partial z}$$

The second derivative along the equilibrium field

$$\partial^2_\parallel \phi = \partial^0_\parallel \left( \partial^0_\parallel \phi \right) = \frac{1}{\sqrt{g_{yy}}} \frac{\partial}{\partial y} \left( \frac{1}{\sqrt{g_{yy}}} \frac{\partial \phi}{\partial y} \right) + \frac{1}{g_{yy}} \frac{\partial^2 \phi}{\partial y^2}$$
A common expression (the Poisson bracket in reduced MHD) is (from equation (14)):
\[ b_0 \cdot \nabla \times \nabla A = \frac{1}{J} \sqrt{g_{yy}} \left[ \left( g_{yy} \frac{\partial \phi}{\partial z} - g_{yz} \frac{\partial \phi}{\partial x} \right) \frac{\partial A}{\partial x} + \left( g_{yz} \frac{\partial \phi}{\partial x} - g_{xy} \frac{\partial \phi}{\partial y} \right) \frac{\partial A}{\partial y} + \left( g_{xy} \frac{\partial \phi}{\partial y} - g_{yy} \frac{\partial \phi}{\partial x} \right) \frac{\partial A}{\partial z} \right] \]
The perpendicular nabla operator:
\[ \nabla_\perp \equiv \nabla - b (b \cdot \nabla) \]
\[ = \nabla_x \left( \frac{\partial}{\partial x} - \frac{g_{xy}}{(JB)^2} \frac{\partial}{\partial y} \right) + \nabla_z \left( \frac{\partial}{\partial z} - \frac{g_{yz}}{(JB)^2} \frac{\partial}{\partial y} \right) \]

**25.3.4 J x B in field-aligned coordinates**

Components of the magnetic field in field-aligned coordinates:
\[ B^y = \frac{B_{pol}}{h_\theta} \quad B^x = B^z = 0 \]
and
\[ B_x = \sigma_{B\theta} B_{\text{tor}} I R \quad B_y = \frac{B^2 h_\theta}{B_{pol}} \quad B_z = \sigma_{B\theta} B_{\text{tor}} R \]

Calculate current \( J = \frac{1}{\mu_0} \nabla \times B \)
\[ (\nabla \times B)^x = \frac{1}{J} \left( \frac{\partial B_z}{\partial y} - \frac{\partial B_y}{\partial z} \right) = 0 \]
since \( B_{\text{tor}} R \) is a flux-surface quantity, and \( B \) is axisymmetric.

\[ (\nabla \times B)^y = - \sigma_{B\theta} \frac{B_{pol}}{h_\theta} \frac{\partial}{\partial x} (B_{\text{tor}} R) \]
\[ (\nabla \times B)^z = \frac{B_{pol}}{h_\theta} \left[ \frac{\partial}{\partial x} \left( \frac{B^2 h_\theta}{B_{pol}} \right) - \sigma_{B\theta} \frac{\partial}{\partial y} (B_{\text{tor}} I R) \right] \]

The second term can be simplified, again using \( B_{\text{tor}} R \) constant on flux-surfaces:
\[ \frac{\partial}{\partial y} (B_{\text{tor}} I R) = \sigma_{B\theta} B_{\text{tor}} R \frac{\partial \nu}{\partial x} \quad \nu = \frac{h_\theta B_{\text{tor}}}{RB_{pol}} \]

From these, calculate covariant components:
\[ (\nabla \times B)_x = - B_{\text{tor}} R \frac{\partial}{\partial x} (B_{\text{tor}} R) + \frac{IR^2 B_{pol}}{h_\theta} \left[ \frac{\partial}{\partial x} \left( \frac{B^2 h_\theta}{B_{pol}} \right) - B_{\text{tor}} R \frac{\partial \nu}{\partial x} \right] \]
\[ (\nabla \times B)_y = - \sigma_{B\theta} \frac{B^2 h_\theta}{B_{pol}} \frac{\partial}{\partial x} (B_{\text{tor}} R) + \sigma_{B\theta} B_{\text{tor}} R \left[ \frac{\partial}{\partial x} \left( \frac{B^2 h_\theta}{B_{pol}} \right) - B_{\text{tor}} R \frac{\partial \nu}{\partial x} \right] \]
\[ (\nabla \times B)_z = - B_{\text{tor}} R \frac{\partial}{\partial x} (B_{\text{tor}} R) + \frac{R^2 B_{pol}}{h_\theta} \left[ \frac{\partial}{\partial x} \left( \frac{B^2 h_\theta}{B_{pol}} \right) - B_{\text{tor}} R \frac{\partial \nu}{\partial x} \right] \]

Calculate \( J \times B \) using
\[ e^i = \frac{1}{J} (e_j \times e_k) \quad e_i = J (e^j \times e^k) \quad i,j,k \text{ cyc } 1,2,3 \]
gives
\[ \mu_0 (J \times B)^x = \frac{1}{J} \left[ (\nabla \times B)_y B_z - (\nabla \times B)_z B_y \right] \]
\[ = - \frac{B_{pol}^3 R^2}{h_\theta} \left[ \frac{\partial}{\partial x} \left( \frac{B^2 h_\theta}{B_{pol}} \right) - B_{\text{tor}} R \frac{\partial \nu}{\partial x} \right] \]
Covariant components of $\nabla P$:

$$(\nabla P)_x = \frac{\partial P}{\partial x}, \quad (\nabla P)_y = (\nabla P)_z = 0$$

and contravariant:

$$(\nabla P)^x = (RB_{pol})^2 \frac{\partial P}{\partial x}, \quad (\nabla P)^y = 0 \quad (\nabla P)^z = -I(RB_{pol})^2 \frac{\partial P}{\partial x}$$

Hence equating contravariant x components of $\mathbf{J} \times \mathbf{B} = \nabla P$,

$$\frac{\partial}{\partial x} \left( B^2 h_\theta \right) - h_\theta B_{pol} \frac{\partial B_{pol}}{\partial x} - B_{tor} R \frac{\partial}{\partial x} \left( \frac{B_{tor} h_\theta}{R} \right) + \mu_0 h_\theta \frac{\partial P}{\partial x} = 0 \quad (25.2)$$

Use this to calculate $h_\theta$ profiles (need to fix $h_\theta$ at one radial location).

Close to x-points, the above expression becomes singular, so a better way to write it is:

$$\frac{\partial}{\partial x} \left( B^2 h_\theta \right) - h_\theta B_{pol} \frac{\partial B_{pol}}{\partial x} - B_{tor} R \frac{\partial}{\partial x} \left( \frac{B_{tor} h_\theta}{R} \right) + \mu_0 h_\theta \frac{\partial P}{\partial x} = 0$$

For solving force-balance by adjusting $P$ and $f$ profiles, the form used is

$$B_{tor} h_\theta \frac{\partial B_{tor}}{\partial x} + B_{tor}^2 h_\theta \frac{\partial R}{\partial x} + \mu_0 h_\theta \frac{\partial P}{\partial x} = -B_{pol} \frac{\partial}{\partial x} (B_{pol} h_\theta)$$

A quick way to calculate $f$ is to rearrange this to:

$$\frac{\partial B_{tor}}{\partial x} = B_{tor} \left[ -1 \frac{\partial R}{R \partial x} \right] + \frac{1}{B_{tor}} \left[ -\mu_0 \frac{\partial P}{\partial x} - \frac{\partial B_{pol}}{\partial h_\theta} \frac{\partial}{\partial x} (B_{pol} h_\theta) \right]$$

and then integrate this using LSODE.

### 25.3.5 Parallel current

$$J_\parallel = \mathbf{b} \cdot \mathbf{J} \quad b^y = \frac{B_{pol}}{B h_\theta}$$

and from equation (25.2):

$$J_y = \frac{\sigma B_\theta}{\mu_0} \left\{ -B_{tor}^2 \frac{\partial}{\partial x} (B_{tor} R) + B_{tor} R \left[ \frac{\partial}{\partial x} \left( \frac{B^2 h_\theta}{B_{pol}} \right) - \sigma B_\theta B_{tor} \frac{\partial}{\partial x} \right] \right\}$$

since $J_\parallel = b^y J_y$,

$$\mu_0 J_\parallel = \frac{\sigma B_\theta B_{pol} B_{tor} R}{B h_\theta} \left[ \frac{\partial}{\partial x} \left( \frac{B^2 h_\theta}{B_{pol}} \right) - B_{tor} R \frac{\partial}{\partial x} \right] - \frac{\sigma B_\theta B}{B_{pol}} \frac{\partial}{\partial x} (B_{tor} R)$$

### 25.3.6 Curvature

For reduced MHD, need to calculate curvature term $\mathbf{b} \times \boldsymbol{\kappa}$, where $\boldsymbol{\kappa} = (\mathbf{b} \cdot \nabla) \mathbf{b} = -\mathbf{b} \times (\nabla \times \mathbf{b})$. Re-arranging, this becomes:

$$\mathbf{b} \times \boldsymbol{\kappa} = \nabla \times \mathbf{b} - \mathbf{b} \cdot (\nabla \times \mathbf{b})$$
Components of $\nabla \times b$ are:

$$(\nabla \times b)^x = \sigma_{B\theta} \frac{B_{pol}}{h_\theta} \frac{\partial}{\partial y} \left( \frac{B_{tor} R}{B} \right)$$

$$(\nabla \times b)^y = -\sigma_{B\theta} \frac{B_{pol}}{h_\theta} \frac{\partial}{\partial x} \left( \frac{B_{tor} R}{B} \right)$$

$$(\nabla \times b)^z = \frac{B_{pol}}{h_\theta} \frac{\partial}{\partial x} \left( \frac{B h_\theta}{B_{pol}} \right) - \sigma_{B\theta} \frac{B_{pol} B_{tor} R \partial \nu}{h_\theta B} - \sigma_{B\theta} I \frac{B_{pol}}{h_\theta} \frac{\partial}{\partial y} \left( \frac{B_{tor} R}{B} \right)$$

giving:

$$\kappa = \frac{B_{pol}}{B h_\theta} \left[ \frac{\partial}{\partial x} \left( \frac{B h_\theta}{B_{pol}} \right) - \sigma_{B\theta} \frac{\partial}{\partial y} \left( \frac{B_{tor} IR}{B} \right) \right] \nabla x$$

$$+ \sigma_{B\theta} \frac{B_{pol}}{B h_\theta} \frac{\partial}{\partial y} \left( \frac{B_{tor} R}{B} \right) \nabla z$$

$$b \cdot (\nabla \times b) = -\sigma_{B\theta} B \frac{\partial}{\partial x} \left( \frac{B_{tor} R}{B} \right) + \sigma_{B\theta} \frac{B_{tor} B_{pol} R}{B h_\theta} \frac{\partial}{\partial x} \left( \frac{B h_\theta}{B_{pol}} \right) - \frac{B_{pol} B_{tor}^2 R^2 \partial \nu}{h_\theta B^2} \frac{\partial}{\partial x}$$

therefore,

$$(b \times \kappa)^x = \sigma_{B\theta} \frac{B_{pol}}{h_\theta} \frac{\partial}{\partial y} \left( \frac{B_{tor} R}{B} \right) = -\sigma_{B\theta} \frac{B_{pol} B_{tor} R \partial B}{h_\theta B^2}$$

$$(b \times \kappa)^y = \frac{B_{pol}^2 B_{tor}^2 R^2 \partial \nu}{B^3 h_\theta^2} - \sigma_{B\theta} \frac{B_{pol}^2 B_{tor} R}{B^2 h_\theta^2} \frac{\partial}{\partial x} \left( \frac{B h_\theta}{B_{pol}} \right)$$

$$(b \times \kappa)^z = \frac{B_{pol}}{h_\theta} \frac{\partial}{\partial x} \left( \frac{B h_\theta}{B_{pol}} \right) - \sigma_{B\theta} \frac{B_{pol} B_{tor} R \partial \nu}{h_\theta B} - \frac{B_{pol} B_{tor}^2 R^2 \partial \nu}{h_\theta B^2} \frac{\partial}{\partial x} - I (b \times \kappa)^x$$

Using equation (25.2):

$$B \frac{\partial}{\partial x} \left( \frac{B h_\theta}{B_{pol}} \right) + B h_\theta \frac{\partial}{\partial x} - \sigma_{B\theta} B_{tor} \frac{\partial}{\partial x} \left( \frac{B_{tor} h_\theta}{R B_{pol}} \right) + \frac{\mu_0 h_\theta}{B_{pol}} \frac{\partial P}{\partial x} = 0$$

we can re-write the above components as:

$$(b \times \kappa)^y = \sigma_{B\theta} \frac{B_{pol} B_{tor} R}{B^3 h_\theta} \left[ \frac{\mu_0 \partial P}{B} \frac{\partial B}{\partial x} + \frac{\partial B}{\partial x} \right]$$

$$(b \times \kappa)^z = -\frac{\mu_0 \partial P}{B} \frac{\partial B}{\partial x} - \frac{\partial B}{\partial x} - I (b \times \kappa)^x$$

### 25.3.7 Curvature from $\text{div} (b/B)$

The vector $b \times \kappa$ is an approximation of

$$\frac{B}{2} \nabla \times \left( \frac{b}{B} \right) \simeq b \times \kappa$$

so can just derive from the original expression. Using the contravariant components of $b$, and the curl operator in curvilinear coordinates (see appendix):

$$\nabla \times \left( \frac{b}{B} \right) = \frac{B_{pol}}{h_\theta} \left[ \left( \frac{\partial}{\partial x} \left( \frac{h_\theta}{B_{pol}} \right) - \frac{\partial}{\partial y} \left( \frac{\sigma_{B\theta} B_{tor} IR}{B^2} \right) \right) e_z \right. \right.$$  

$$+ \frac{\partial}{\partial y} \left( \frac{\sigma_{B\theta} B_{tor} R}{B^2} \right) e_x \right.$$  

$$+ \left. \left. \frac{\partial}{\partial x} \left( \frac{\sigma_{B\theta} B_{tor} R}{B^2} \right) e_y \right] \right.$$
This can be simplified using
\[ \frac{\partial}{\partial y} \left( \frac{\sigma B \partial I R}{B^2} \right) = I \sigma B \partial R \frac{\partial}{\partial y} \left( \frac{1}{B^2} \right) + \frac{B \partial R}{B^2} \frac{\partial \nu}{\partial x} \]
to give
\[
(b \times \kappa)^x = -\sigma B \frac{B_{pol} B \partial}{h_{\theta} B^2} \frac{\partial B}{\partial y} \\
(b \times \kappa)^y = -\sigma B \frac{BB \partial}{2h_{\theta}} \left( \frac{B \partial R}{B^2} \right) \\
(b \times \kappa)^z = \frac{BB}{2h_{\theta}} \frac{\partial}{\partial x} \left( \frac{h_{\theta}}{B_{pol}} \right) - \frac{B_{pol} B \partial R}{2h_{\theta} B} \frac{\partial \nu}{\partial x} - I (b \times \kappa \cdot \nabla)^x
\]

The first and second terms in \((b \times \kappa \cdot \nabla)^z\) almost cancel, so by expanding out \(\nu\) a better expression is
\[
(b \times \kappa)^z = \frac{B_{pol}^3}{2h_{\theta} B} \frac{\partial}{\partial x} \left( \frac{h_{\theta}}{B_{pol}} \right) - \frac{B_{pol} B \partial R}{2B} \frac{\partial \nu}{\partial x} - I (b \times \kappa \cdot \nabla)^z
\]

### 25.3.8 Curvature of a single line

The curvature vector can be calculated from the field-line toroidal coordinates \((R, Z, \phi)\) as follows. The line element is given by
\[
dr = dR \hat{R} + dZ \hat{Z} + Rd\phi \hat{\phi}
\]
Hence the tangent vector is
\[
\hat{T} = \frac{dr}{ds} = \frac{dR}{ds} \hat{R} + \frac{dZ}{ds} \hat{Z} + \frac{d\phi}{ds} \hat{\phi}
\]
where \(s\) is the distance along the field-line. From this, the curvature vector is given by
\[
\kappa \equiv \frac{dT}{ds} = \frac{d^2 R}{ds^2} \hat{R} + \frac{dR}{ds} \frac{d\phi}{ds} \hat{\phi} \\
+ \frac{d^2 Z}{ds^2} \hat{Z} \\
+ \frac{dR}{ds} \frac{d\phi}{ds} \hat{\phi} + R \frac{d^2 \phi}{ds^2} \hat{\phi} - R \left( \frac{d\phi}{ds} \right)^2 \hat{R}
\]
i.e.
\[
\kappa = \left[ \frac{d^2 R}{ds^2} - R \left( \frac{d\phi}{ds} \right)^2 \right] \hat{R} + \frac{d^2 Z}{ds^2} \hat{Z} + \left[ 2 \frac{dR}{ds} \frac{d\phi}{ds} + R \frac{d^2 \phi}{ds^2} \right] \hat{\phi}
\]
(25.3)

Want the components of \(b \times \kappa\), and since the vector \(b\) is just the tangent vector \(T\) above, this can be written using the cross-products
\[
R \times \hat{Z} = -\hat{\phi} \quad \hat{\phi} \times \hat{Z} = \hat{R} \quad R \times \hat{\phi} = \hat{Z}
\]
This vector must then be dotted with \(\nabla \psi\), \(\nabla \theta\), and \(\nabla \phi\). This is done by writing these vectors in cylindrical coordinates:
\[
\nabla \psi = \frac{\partial \psi}{\partial R} \hat{R} + \frac{\partial \psi}{\partial Z} \hat{Z} \\
\nabla \theta = \frac{1}{B_{pol} h_{\theta}} \nabla \phi \times \nabla \psi = \frac{1}{RB_{pol} h_{\theta}} \left( \frac{\partial \psi}{\partial Z} \hat{R} - \frac{\partial \psi}{\partial R} \hat{Z} \right)
\]

### 25.3. Field-aligned coordinates

211
An alternative is to use
\[ b \times \nabla \phi = \frac{\sigma B \theta}{B R^2} \nabla \psi \]
and that the tangent vector \( T = b \). This gives
\[ \nabla \psi = \sigma B \theta B R \left[ \frac{dR}{ds} Z - \frac{dZ}{ds} R \right] \] (25.4)
and so because \( d\phi / ds = B_{\text{tor}} / (RB) \)
\[ \kappa \cdot \nabla \psi = \sigma B \theta B R \left[ \left( \frac{B_{\text{tor}}}{RB^2} - \frac{d^2 R}{ds^2} \right) \frac{dZ}{ds} + \frac{d^2 Z}{ds^2} \frac{dR}{ds} \right] \] (25.5)
Taking the cross-product of the tangent vector with the curvature in equation (25.3) above gives
\[ b \times \kappa = \left[ \frac{B_{\text{tor}}}{B} \frac{d^2 Z}{ds^2} - \frac{dZ}{ds} \left( 2 \frac{dR}{ds} \frac{d\phi}{ds} + R \frac{d^2 \phi}{ds^2} \right) \right] R
+ \left[ \frac{dR}{ds} \left( 2 \frac{dR}{ds} \frac{d\phi}{ds} + R \frac{d^2 \phi}{ds^2} \right) - \frac{B_{\text{tor}}}{B} \left( \frac{d^2 R}{ds^2} - R \left( \frac{d\phi}{ds} \right)^2 \right) \right] Z
+ \left[ \frac{dZ}{ds} \left( \frac{d^2 R}{ds^2} - R \left( \frac{d\phi}{ds} \right)^2 \right) - \frac{dR}{ds} \frac{d^2 Z}{ds^2} \right] \hat{\phi} \]
The components in field-aligned coordinates can then be calculated:
\[ (b \times \kappa)^x = \sigma B \theta (b \times \kappa) \cdot \nabla \psi \]
\[ = \frac{RB_{\text{pol}}^2}{B} \left( 2 \frac{dR}{ds} \frac{d\phi}{ds} + R \frac{d^2 \phi}{ds^2} \right) - RB_{\text{tor}} \left( \frac{dR}{ds} \frac{d^2 R}{ds^2} + \frac{dZ}{ds} \frac{d^2 Z}{ds^2} \right) + \frac{B_{\text{tor}}^3}{B^2} \frac{dR}{ds} \]
25.3.9 Curvature in toroidal coordinates
In toroidal coordinates \((\psi, \theta, \phi)\), the \( b \) vector is
\[ b = \frac{B_{\text{pol}}}{B} \hat{e}_\theta + \frac{B_{\text{tor}}}{B} \hat{e}_\phi \]
\[ = \frac{B_{\text{pol}} h_\theta}{B} \nabla \theta + \frac{RB_{\text{pol}}}{B} \nabla \phi \]
The curl of this vector is
\[ (\nabla \times b)^\psi = \frac{1}{\sqrt{g}} \left( \frac{\partial b_\phi}{\partial \theta} - \frac{\partial b_\theta}{\partial \phi} \right) \]
\[ (\nabla \times b)^\theta = \frac{1}{\sqrt{g}} \left( \frac{\partial b_\psi}{\partial \phi} - \frac{\partial b_\phi}{\partial \psi} \right) \]
\[ (\nabla \times b)^\phi = \frac{1}{\sqrt{g}} \left( \frac{\partial b_\theta}{\partial \psi} - \frac{\partial b_\psi}{\partial \theta} \right) \]
where \( 1/\sqrt{g} = B_{\text{pol}} / h_\theta \). Therefore, in terms of unit vectors:
\[ \nabla \times b = \frac{1}{Rh_\theta} \frac{\partial}{\partial \theta} \left( \frac{RB_{\text{pol}}}{B} \right) \hat{e}_\psi - B_{\text{pol}} \frac{\partial}{\partial \psi} \left( \frac{RB_{\text{tor}}}{B} \right) \hat{e}_\theta + \frac{B_{\text{pol}} R}{h_\theta} \frac{\partial}{\partial \psi} \left( h_\theta B_{\text{pol}} \right) \hat{e}_\phi \]
25.3.10 psi derivative of the B field

Needed to calculate magnetic shear, and one way to get the curvature. The simplest way is to use finite differencing, but there is another way using local derivatives (implemented using DCT).

\[ B_{\text{pol}} = \frac{\left| \nabla \psi \right|}{R} = \frac{1}{R} \sqrt{\left( \frac{\partial \psi}{\partial R} \right)^2 + \left( \frac{\partial \psi}{\partial \theta} \right)^2} \]

Using

\[ \nabla B_{\text{pol}} = \frac{\partial B_{\text{pol}}}{\partial \psi} \nabla \psi + \frac{\partial B_{\text{pol}}}{\partial \theta} \nabla \theta + \frac{\partial B_{\text{pol}}}{\partial \phi} \nabla \phi \]

we get

\[ \nabla B_{\text{pol}} \cdot \nabla \psi = \frac{\partial B_{\text{pol}}}{\partial \psi} \left| \nabla \psi \right|^2 \]

and so

\[ \frac{\partial B_{\text{pol}}}{\partial \psi} = \frac{\nabla B_{\text{pol}} \cdot \nabla \psi}{(RB_{\text{pol}})^2} \]

The derivatives of \( B_{\text{pol}} \) in \( R \) and \( Z \) are:

\[ \frac{\partial B_{\text{pol}}}{\partial R} = -\frac{B_{\text{pol}}}{R} + \frac{1}{B_{\text{pol}} R^2} \left[ \frac{\partial \psi}{\partial R} \frac{\partial^2 \psi}{\partial R^2} + \frac{\partial \psi}{\partial Z} \frac{\partial^2 \psi}{\partial R \partial Z} \right] \]

\[ \frac{\partial B_{\text{pol}}}{\partial Z} = \frac{1}{B_{\text{pol}} R^2} \left[ \frac{\partial \psi}{\partial Z} \frac{\partial^2 \psi}{\partial Z^2} + \frac{\partial \psi}{\partial R} \frac{\partial^2 \psi}{\partial R \partial Z} \right] \]

For the toroidal field, \( B_{\text{tor}} = f/R \)

\[ \frac{\partial B_{\text{tor}}}{\partial \psi} = \frac{1}{R} \frac{\partial f}{\partial \psi} - \frac{f}{R^2} \frac{\partial R}{\partial \psi} \]

As above, \( \frac{\partial R}{\partial \psi} = \nabla R \cdot \nabla \psi / (RB_{\text{pol}})^2 \), and since \( \nabla R \cdot \nabla R = 1 \),

\[ \frac{\partial R}{\partial \psi} = \frac{\partial \psi}{\partial R} / (RB_{\text{pol}})^2 \]

similarly,

\[ \frac{\partial Z}{\partial \psi} = \frac{\partial \psi}{\partial Z} / (RB_{\text{pol}})^2 \]

and so the variation of toroidal field with \( \psi \) is

\[ \frac{\partial B_{\text{tor}}}{\partial \psi} = \frac{1}{R} \frac{\partial f}{\partial \psi} - \frac{B_{\text{tor}}}{R^3 B_{\text{pol}}^2} \frac{\partial \psi}{\partial R} \]

From the definition \( B = \sqrt{B_{\text{tor}}^2 + B_{\text{pol}}^2} \).

\[ \frac{\partial B}{\partial \psi} = \frac{1}{B} \left( B_{\text{tor}} \frac{\partial B_{\text{tor}}}{\partial \psi} + B_{\text{pol}} \frac{\partial B_{\text{pol}}}{\partial \psi} \right) \]
25.3.11 Parallel derivative of the B field

To get the parallel nablaents of the $B$ field components, start with

$$\frac{\partial}{\partial s} (B^2) = \frac{\partial}{\partial s} (B_{\text{tor}}^2) + \frac{\partial}{\partial s} (B_{\text{pol}}^2)$$

Using the fact that $RB_{\text{tor}}$ is constant along $s$,

$$\frac{\partial}{\partial s} (R^2 B_{\text{tor}}^2) = R^2 \frac{\partial}{\partial s} (B_{\text{tor}}^2) + B_{\text{tor}}^2 \frac{\partial}{\partial s} (R^2) = 0$$

which gives

$$\frac{\partial}{\partial s} (B_{\text{tor}}^2) = - \frac{B_{\text{tor}}^2}{R^2} \frac{\partial}{\partial s} (R^2)$$

The poloidal field can be calculated from

$$\frac{\partial}{\partial s} (\nabla \psi \cdot \nabla \psi) = \frac{\partial}{\partial s} (R^2 B_{\text{pol}}^2) = R^2 \frac{\partial}{\partial s} (B_{\text{pol}}^2) + B_{\text{pol}}^2 \frac{\partial}{\partial s} (R^2)$$

Using equation (25.4), $\nabla \psi \cdot \nabla \psi$ can also be written as

$$\nabla \psi \cdot \nabla \psi = B^2 R^2 \left[ \left( \frac{\partial R}{\partial s} \right)^2 + \left( \frac{\partial Z}{\partial s} \right)^2 \right]$$

and so (unsurprisingly)

$$\frac{B_{\text{pol}}^2}{B^2} = \left[ \left( \frac{\partial R}{\partial s} \right)^2 + \left( \frac{\partial Z}{\partial s} \right)^2 \right]$$

Hence

$$\frac{\partial}{\partial s} (B_{\text{pol}}^2) = B^2 \frac{\partial}{\partial s} \left( \left( \frac{\partial R}{\partial s} \right)^2 + \left( \frac{\partial Z}{\partial s} \right)^2 \right) + \frac{B_{\text{pol}}^2}{B^2} \frac{\partial}{\partial s} (B^2)$$

Which gives

$$\frac{\partial}{\partial s} (B^2) = -\frac{B_{\text{pol}}^2}{R^2} \frac{\partial}{\partial s} (R^2) + \frac{B_{\text{pol}}^2}{B_{\text{tor}}^2} \frac{\partial}{\partial s} \left( \left( \frac{\partial R}{\partial s} \right)^2 + \left( \frac{\partial Z}{\partial s} \right)^2 \right)$$

$$\frac{\partial}{\partial s} (B_{\text{pol}}^2) = \left( 1 + \frac{B_{\text{pol}}^2}{B_{\text{tor}}^2} \right) B^2 \frac{\partial}{\partial s} \left( \left( \frac{\partial R}{\partial s} \right)^2 + \left( \frac{\partial Z}{\partial s} \right)^2 \right) - \frac{B_{\text{pol}}^2}{R^2} \frac{\partial}{\partial s} (B^2)$$

25.3.12 Magnetic shear from J x B

Re-arranging the radial force balance equation (25.2) gives

$$\frac{B_{\text{pol}}^2 R}{B_{\text{tor}}} \frac{\partial \nu}{\partial \psi} + \nu \left( \frac{2 RB \partial B}{B_{\text{tor}}} \frac{\partial B}{\partial \psi} + B^2 \frac{\partial R}{B_{\text{tor}}} \frac{\partial B_{\text{tor}}}{\partial \psi} - B^2 R \frac{\partial B_{\text{tor}}}{\partial \psi} \right) + \frac{\mu_0 h \psi}{B_{\text{pol}}} \frac{\partial P}{\partial \psi} = 0$$
25.3.13 Magnetic shear

The field-line pitch is given by

$$\nu = \frac{h_\theta B_{tor}}{B_{pol} R}$$

and so

$$\frac{\partial \nu}{\partial \psi} = \frac{\nu}{h_\theta} \frac{\partial h_\theta}{\partial \psi} + \frac{\nu}{B_{tor}} \frac{\partial B_{tor}}{\partial \psi} - \frac{\nu}{B_{pol}} \frac{\partial B_{pol}}{\partial \psi} - \nu \frac{\partial R}{R} \frac{\partial \psi}{\partial \psi}$$

The last three terms are given in the previous section, but \(\frac{\partial h_\theta}{\partial \psi}\) needs to be evaluated.

25.3.14 psi derivative of \(h\)

From the expression for curvature (equation (25.3)), and using \(\nabla x \cdot \nabla \psi = \sigma_{B\theta} (RB_{pol})^2\) and \(\nabla z \cdot \nabla \psi = -\sigma_{B\theta} I (RB_{pol})^2\)

$$\kappa \cdot \nabla \psi = -\sigma_{B\theta} \frac{B_{pol}}{Bh_\theta} (RB_{pol})^2 \left[ \frac{\partial}{\partial x} \left( \frac{Bh_\theta}{B_{pol}} \right) - \sigma_{B\theta} \frac{\partial}{\partial y} \left( \frac{B_{tor} IR}{B} \right) \right]$$

$$- I (RB_{pol})^2 \frac{B_{pol}}{Bh_\theta} \frac{\partial}{\partial y} \left( \frac{B_{tor} R}{B} \right)$$

The second and third terms partly cancel, and using \(\frac{\partial I}{\partial y} = \sigma_{B\theta} \frac{\partial \nu}{\partial x}\)

$$\frac{\kappa \cdot \nabla \psi}{(RB_{pol})^2} = -\sigma_{B\theta} \frac{B_{pol}}{Bh_\theta} \frac{\partial}{\partial x} \left( \frac{Bh_\theta}{B_{pol}} \right) + \sigma_{B\theta} \frac{B_{pol}}{Bh_\theta} \frac{B_{tor} R \partial \nu}{\partial x}$$

$$= -\sigma_{B\theta} \frac{B_{pol}}{Bh_\theta} \left[ h_\theta \frac{\partial}{\partial x} \left( \frac{B}{B_{pol}} \right) - h_\theta \frac{B_{tor} R \partial}{\partial x} \left( \frac{B_{tor}}{B_{pol} R} \right) + B^2 \frac{\partial h_\theta}{\partial x} \frac{BB_{pol}}{B_{pol} R} \frac{\partial}{\partial x} \left( \frac{B_{tor}}{B_{pol} R} \right) \right]$$

$$= -\sigma_{B\theta} \frac{B_{pol}}{B^2 h_\theta} \frac{\partial h_\theta}{\partial x} - \sigma_{B\theta} \frac{B_{pol}}{B^2} \left[ \frac{\partial}{\partial x} \left( \frac{B_{tor}}{B_{pol}} \right) - B_{tor} \frac{\partial}{\partial x} \left( \frac{B_{tor}}{B_{pol} R} \right) \right]$$

Writing

$$B \frac{\partial}{\partial x} \left( \frac{B}{B_{pol}} \right) = \frac{\partial}{\partial x} \left( \frac{B^2}{B_{pol}} \right) - \frac{B}{B_{pol}} \frac{\partial B}{\partial x}$$

$$B_{tor} R \frac{\partial}{\partial x} \left( \frac{B_{tor}}{B_{pol} R} \right) = \frac{\partial}{\partial x} \left( \frac{B_{tor}^2}{B_{pol}} \right) - \frac{B_{tor}}{B_{pol} R} \frac{\partial}{\partial x} \left( B_{tor} R \right)$$

and using \(B \frac{\partial B}{\partial x} = B_{tor} \frac{\partial B_{tor}}{\partial x} + B_{pol} \frac{\partial B_{pol}}{\partial x}\), this simplifies to give

$$\frac{\kappa \cdot \nabla \psi}{(RB_{pol})^2} = -\sigma_{B\theta} \frac{B_{pol}^2}{B^2 h_\theta} \frac{\partial h_\theta}{\partial x} - \sigma_{B\theta} \frac{B_{tor}^2}{B^2 R} \frac{\partial R}{\partial x}$$

(25.6)

This can be transformed into an expression for \(\frac{\partial h_\theta}{\partial x}\) involving only derivatives along field-lines. Writing \(\nabla R = \frac{\partial R}{\partial \psi} \nabla \psi + \frac{\partial R}{\partial \theta} \nabla \theta\)

$$\nabla R \cdot \nabla \psi = \frac{\partial R}{\partial \psi} (RB_{pol})^2$$
Using (25.4),

\[ \nabla \psi \cdot \nabla R = -\sigma_{B\theta} B R \frac{dZ}{ds} \]

and so

\[ \frac{\partial R}{\partial x} = -\frac{BR}{(RB_{\text{pol}})^2} \frac{dZ}{ds} \]

Substituting this and equation (25.5) for \( \kappa \cdot \nabla \psi \) into equation (25.6) the \( \frac{\partial R}{\partial x} \) term cancels with part of the \( \kappa \cdot \nabla \psi \) term, simplifying to

\[ \frac{\partial h_{\theta}}{\partial x} = -h_{\theta} \frac{B^3 R}{B_{\text{pol}}^2 (RB_{\text{pol}})^2} \left[ \frac{d^2 Z}{ds^2} \frac{dR}{ds} - \frac{d^2 R}{ds^2} \frac{dZ}{ds} \right] \]

### 25.4 Shifted radial derivatives

The coordinate system given by equation (25.1) and used in the above sections has a problem: There is a special poloidal location \( \theta_0 \) where the radial basis vector \( e_x \) is purely in the \( \nabla \psi \) direction. Moving away from this location, the coordinate system becomes sheared in the toroidal direction.

Making the substitution

\[ \frac{\partial}{\partial x} = \frac{\partial}{\partial \psi} + I \frac{\partial}{\partial z} \]

we also get the mixed derivative

\[ \frac{\partial}{\partial z \partial x} = \frac{\partial}{\partial z} \frac{\partial}{\partial \psi} + \frac{\partial I}{\partial z} \frac{\partial}{\partial \psi} + I \frac{\partial^2}{\partial \psi \partial z} = \frac{\partial^2}{\partial z \partial \psi} + I \frac{\partial^2}{\partial \psi \partial z} \]

and second-order \( x \) derivative

\[ \frac{\partial^2}{\partial x^2} = \frac{\partial^2}{\partial \psi^2} + \left( I \frac{\partial}{\partial \psi} + I \frac{\partial^2}{\partial \psi \partial z} \right) \frac{\partial}{\partial \psi} + \left( I \frac{\partial^2}{\partial \psi \partial z} \right) \frac{\partial}{\partial \psi} = \frac{\partial^2}{\partial \psi^2} + I^2 \frac{\partial^2}{\partial z^2} + 2I \frac{\partial^2}{\partial \psi \partial z} + \frac{\partial I}{\partial \psi} \frac{\partial}{\partial z} \]

#### 25.4.1 Perpendicular Laplacian

\[ \nabla^2_{\perp} = (RB_{\text{pol}})^2 \left[ \frac{\partial^2}{\partial x^2} - 2I \frac{\partial^2}{\partial z \partial x} + \left( I^2 + \frac{B^2}{(RB_{\text{pol}})^4} \right) \frac{\partial^2}{\partial z^2} \right] \]

transforms to

\[ \nabla^2_{\perp} = (RB_{\text{pol}})^2 \left[ \frac{\partial^2}{\partial \psi^2} + \frac{\partial I}{\partial \psi} \frac{\partial}{\partial \psi} + \frac{B^2}{(RB_{\text{pol}})^4} \frac{\partial^2}{\partial z^2} \right] \]

(25.7)

The extra term involving \( I \) disappears, but only if both the \( x \) and \( z \) first derivatives are taken into account:

\[ \nabla^2_{\perp} = (RB_{\text{pol}})^2 \left[ \frac{\partial^2}{\partial x^2} - 2I \frac{\partial^2}{\partial z \partial x} + \left( I^2 + \frac{B^2}{(RB_{\text{pol}})^4} \right) \frac{\partial^2}{\partial z^2} \right] + \nabla^2_x \frac{\partial}{\partial x} + \nabla^2_z \frac{\partial}{\partial z} \]
with

\[ \nabla^2 x = \frac{1}{J} \frac{\partial}{\partial x} \left[ J(RB_{pol})^2 \right] \]
\[ \nabla^2 z = \frac{1}{J} \left[ -\frac{\partial}{\partial x} \left( J(I(RB_{pol})^2) \right) - \frac{\partial}{\partial y} \left( \frac{B_{tor}}{B_{pol}^2 R} \right) \right] \]
\[ = \frac{1}{J} \left[ -I \frac{\partial}{\partial x} (J(RB_{pol})^2) - \frac{\partial}{\partial x} J(RB_{pol})^2 - \frac{\partial}{\partial y} \left( \frac{B_{tor}}{B_{pol}^2 R} \right) \right] \]

where \( J = h_\theta / B_{pol} \) is the Jacobian. Transforming into \( \psi \) derivatives, the middle term of equation (25.8) cancels the \( I \) term in equation (25.7), but introduces another \( I \) term (first term in equation (25.8)). This term cancels with the \( \nabla^2 x \) term when \( \frac{\partial}{\partial x} \) is expanded, so the full expression for \( \nabla^2_\perp \) using \( \psi \) derivatives is:

\[ \nabla^2_\perp = (RB_{pol})^2 \left[ \frac{\partial^2}{\partial \psi^2} + \frac{B^2}{(RB_{pol})^4} \frac{\partial^2}{\partial z^2} \right] \]
\[ + \frac{1}{J} \frac{\partial}{\partial \psi} \left[ J(RB_{pol})^2 \right] \frac{\partial A}{\partial \psi} - \frac{1}{J} \frac{\partial}{\partial y} \left( \frac{B_{tor}}{B_{pol}^2 R} \right) \frac{\partial A}{\partial z} \]

**In orthogonal (psi, theta, zeta) flux coordinates**

For comparison, the perpendicular Laplacian can be derived in orthogonal “flux” coordinates

\[ |\nabla \psi| = RB_{pol} \quad |\nabla \theta| = 1/h_\theta \quad |\nabla \zeta| = 1/R \]

The Laplacian operator is given by

\[ \nabla^2 A = (RB_{pol})^2 \frac{\partial^2 A}{\partial \psi^2} + \frac{1}{h_\theta^2} \frac{\partial^2 A}{\partial \theta^2} + \frac{1}{R^2} \frac{\partial^2 A}{\partial \zeta^2} \]
\[ + \frac{1}{J} \frac{\partial}{\partial \psi} \left[ J(RB_{pol})^2 \right] \frac{\partial A}{\partial \psi} + \frac{1}{J} \frac{\partial}{\partial \theta} \left( \frac{J}{h_\theta^2} \right) \frac{\partial A}{\partial \theta} \]

parallel derivative by

\[ \partial || \equiv \mathbf{b} \cdot \nabla = \frac{B_{pol}}{Bh_\theta} \frac{\partial}{\partial \theta} + \frac{B_{tor}}{RB} \frac{\partial}{\partial \zeta} \]

and so

\[ \partial^2 || A \equiv \partial || \left( \partial || A \right) = \left( \frac{B_{pol}}{Bh_\theta} \right)^2 \frac{\partial^2 A}{\partial \theta^2} + \left( \frac{B_{tor}}{RB} \right)^2 \frac{\partial^2 A}{\partial \zeta^2} \]
\[ + 2 \frac{B_{pol}B_{tor}}{B^2 h_\theta R} \frac{\partial^2 A}{\partial \theta \partial \zeta} \]
\[ + \frac{\partial}{\partial \theta} \left( \frac{B_{pol}}{Bh_\theta} \right) \frac{\partial A}{\partial \theta} + \frac{\partial}{\partial \theta} \left( \frac{B_{tor}}{RB} \right) \frac{\partial A}{\partial \zeta} \]

Hence in orthogonal flux coordinates, the perpendicular Laplacian is:

\[ \nabla^2_\perp = \nabla^2 - \partial^2 || \equiv (RB_{pol})^2 \left[ \frac{\partial^2}{\partial \psi^2} + \frac{1}{R^2 B^2} \frac{\partial^2}{\partial \zeta^2} \right] + \frac{B_{tor}^2}{h_\theta^2 B^2} \frac{\partial^2}{\partial \theta^2} + \cdots \]  \hspace{1cm} (25.8)

where the neglected terms are first-order derivatives. The coefficient for the second-order \( z \) derivative differs from equation (25.8), and equation (25.8) still contains a derivative in \( \theta \). This shows that the transformation made to get equation (25.8) doesn’t result in the same answer as orthogonal flux coordinates: equation (25.8) is in field-aligned coordinates.

Note that in the limit of \( B_{pol} = B \), both equations (25.8) and (25.8) are the same, as they should be.
25.4.2 Operator $B \times \nabla \phi \cdot \nabla A$

\[
B \times \nabla \phi \cdot \nabla A = \left( \frac{\partial \phi}{\partial x} \frac{\partial A}{\partial y} - \frac{\partial \phi}{\partial y} \frac{\partial A}{\partial x} \right) \left( -B_{\text{tor}} \frac{RB_{\text{pol}}}{h_\theta} \right) 
+ \left( \frac{\partial \phi}{\partial x} \frac{\partial A}{\partial z} - \frac{\partial \phi}{\partial z} \frac{\partial A}{\partial x} \right) \left( -B^2 \right) 
- \left( \frac{\partial \phi}{\partial y} \frac{\partial A}{\partial z} - \frac{\partial \phi}{\partial z} \frac{\partial A}{\partial y} \right) \left( IB_{\text{tor}} \frac{RB_{\text{pol}}}{h_\theta} \right) 
\]

\[
B \times \nabla \phi \cdot \nabla A = \left( \frac{\partial \phi}{\partial \psi} \frac{\partial A}{\partial y} + I \frac{\partial \phi}{\partial y} - \frac{\partial \phi}{\partial \psi} \frac{\partial A}{\partial \psi} \right) \left( -B_{\text{tor}} \frac{RB_{\text{pol}}}{h_\theta} \right) 
+ \left( \frac{\partial \phi}{\partial \psi} \frac{\partial A}{\partial z} - \frac{\partial \phi}{\partial z} \frac{\partial A}{\partial \psi} \right) \left( IB_{\text{tor}} \frac{RB_{\text{pol}}}{h_\theta} \right) 
\]

\[
B \times \nabla \phi \cdot \nabla A = \left( \frac{\partial \phi}{\partial \psi} \frac{\partial A}{\partial y} - \frac{\partial \phi}{\partial y} \frac{\partial A}{\partial \psi} \right) \left( -B_{\text{tor}} \frac{RB_{\text{pol}}}{h_\theta} \right) 
+ \left( \frac{\partial \phi}{\partial \psi} \frac{\partial A}{\partial z} - \frac{\partial \phi}{\partial z} \frac{\partial A}{\partial \psi} \right) \left( -B^2 \right) 
\]

25.5 Useful identities

25.5.1 $b \times \kappa \cdot \nabla \psi \simeq -RB_\zeta \partial_{||} \ln B$

Using $b \times \kappa \simeq \frac{B}{2} \nabla \times \frac{b}{B}$, and working in orthogonal ($\psi, \theta, \zeta$) coordinates. The magnetic field unit vector is:

\[
b = \frac{B h_\theta}{B} \nabla \theta + \frac{B \zeta}{B} \nabla \zeta
\]

and using the definition of curl (equation (11)) we can write

\[
b \times \kappa \simeq \frac{B}{2} \nabla \times \frac{b}{B} = \frac{B B_\theta \partial}{2 h_\theta} \left( \frac{B \zeta}{B^2} \frac{\partial}{\partial \theta} \left( \frac{B \zeta}{B} \frac{\partial}{\partial \zeta} \left( \frac{B h_\theta}{B^2} \right) \right) \right) e_\psi + [\ ] e_\theta + [\ ] e_\zeta
\]

so that when dotted with $\nabla \psi$, only the first bracket survives. The parallel gradient is

\[
\partial_{||} = b \cdot \nabla = \frac{B_\theta}{B h_\theta} \frac{\partial}{\partial \theta} + \frac{B_\theta}{B R} \frac{\partial}{\partial \zeta}
\]

Neglecting derivatives for axisymmetric equilibrium

\[
\frac{B}{2} \nabla \times \frac{b}{B} \cdot \nabla \psi = \frac{1}{2} B_\zeta R \left( \frac{B \zeta}{B^2} \right)
\]

Since $B_\zeta R$ is a flux function, this can be written as

\[
\frac{B}{2} \nabla \times \frac{b}{B} \cdot \nabla \psi = -B_\zeta R \frac{1}{B} \partial_{||} B
\]

and so

\[
b \times \kappa \cdot \nabla \psi \simeq -RB_\zeta \partial_{||} \ln B
\]


.1 Differential geometry

Warning: Several mistakes have been found (and is now corrected) in this section, so it should be proof read before removing this warning! The following are notes from [haeseler].

Sets of vectors \{A, B, C\} and \{a, b, c\} are reciprocal if

\[
A \cdot a = B \cdot b = C \cdot c = 1
\]

\[
A \cdot b = A \cdot c = B \cdot a = B \cdot c = C \cdot a = C \cdot b = 0
\]

which implies that \{A, B, C\} and \{a, b, c\} are each linearly independent. Equivalently,

\[
a = \frac{B \times C}{A \cdot (B \times C)} \quad b = \frac{C \times A}{B \cdot (C \times A)} \quad c = \frac{A \times B}{C \cdot (A \times B)}
\]

Either of these sets can be used as a basis, and any vector \(w\) can be represented as \(w = (w \cdot a) A + (w \cdot b) B + (w \cdot c) C\) or as \(w = (w \cdot A) a + (w \cdot B) b + (w \cdot C) c\). In the Cartesian coordinate system, the basis vectors are reciprocal to themselves so this distinction is not needed. For a general set of coordinates \(\{u^1, u^2, u^3\}\), tangent basis vectors are:

\[
e_i = \frac{\partial \mathbf{R}}{\partial u^i}
\]

and in general these will vary from point to point. The scale factor or metric coefficient \(h_i = |e_i|\) is the distance moved for a unit change in \(u^i\). The unit vector \(e_i = e_i/h_i\). Definition of nabla operator:

\[\nabla \Phi\] of a function \(\Phi\) is defined so that \(d\Phi = \nabla \Phi \cdot d\mathbf{R}\)

From the chain rule, \(d\mathbf{R} = \frac{\partial \mathbf{R}}{\partial u^i} du^i = e_i du^j\) and substituting \(\Phi = u^i\)

\[du^i = \nabla u^i \cdot e_j du^j\]

which can only be true if \(\nabla u^i \cdot e_j = \delta^i_j\) i.e. if

Sets of vectors \(e^i \equiv \nabla u^i\) and \(e_j\) are reciprocal

Since the sets of vectors \(\{e^i\}\) and \(\{e_j\}\) are reciprocal, any vector \(\mathbf{D}\) can be written as \(\mathbf{D} = D_i e^i = D^i e_i\) where \(D_i = \mathbf{D} \cdot e_i\) are the covariant components and \(D^i = \mathbf{D} \cdot e^i\) are the contravariant components. To convert between covariant and contravariant components, define the metric coefficients \(g_{ij} = e_i \cdot e_j\) and \(g^{ij} = e^i \cdot e^j\) so that \(e_i = g_{ij} e^j\). \(g_{ij}\) and \(g^{ij}\) are symmetric and if the basis is orthogonal then \(g_{ij} = g^{ij} = 0\) for \(i \neq j\) i.e. the metric is diagonal.

\[g_{ij} = h_i h_j e_i \cdot e_j\] and so \(g_{ii} = h_i^2\)

For a general set of coordinates, the nabla operator can be expressed as

\[\nabla = \nabla u^i \frac{\partial}{\partial u^i} = e^i \frac{\partial}{\partial u^i}\]

and for a general set of (differentiable) coordinates \(\{u^i\}\), the Laplacian is given by

\[\nabla^2 \phi = \frac{1}{f} \frac{\partial}{\partial u^i} \left( J g^{ij} \frac{\partial \phi}{\partial u^j} \right)\]

which can be expanded as

\[\nabla^2 \phi = g^{ij} \frac{\partial^2 \phi}{\partial u^i \partial u^j} + \frac{1}{f} \frac{\partial}{\partial u^i} \left( J g^{ij} \right) \frac{\partial \phi}{\partial u^j}\]
where $G^i_j$ must not be mistaken as the so called connection coefficients (i.e. the Christoffel symbols of second kind). Setting $\phi = u^k$ in equation (9) gives $\nabla^2 u^k = G^k$. Expanding (9) and setting $\{u^i\} = \{x, y, z\}$ gives

$$\nabla^2 f = \nabla \cdot \nabla f = \nabla \cdot \left( \frac{\partial}{\partial x} \nabla x + \frac{\partial}{\partial y} \nabla y + \frac{\partial}{\partial z} \nabla z \right)$$

$$= \frac{\partial^2 f}{\partial x^2} |\nabla x|^2 + \frac{\partial^2 f}{\partial y^2} |\nabla y|^2 + \frac{\partial^2 f}{\partial z^2} |\nabla z|^2$$

$$+ 2 \frac{\partial^2 f}{\partial x \partial y} (\nabla x \cdot \nabla y) + 2 \frac{\partial^2 f}{\partial x \partial z} (\nabla x \cdot \nabla z) + 2 \frac{\partial^2 f}{\partial y \partial z} (\nabla y \cdot \nabla z)$$

$$+ \nabla^2 x \frac{\partial f}{\partial x} + \nabla^2 y \frac{\partial f}{\partial y} + \nabla^2 z \frac{\partial f}{\partial z}$$

Curl defined as:

$$\nabla \times A = \frac{1}{\sqrt{g}} \sum_k \left( \frac{\partial A_j}{\partial u_i} - \frac{\partial A_i}{\partial u_j} \right) e_k \quad i, j, k \text{ cyc } 1, 2, 3$$

Cross-product relation between contravariant and covariant vectors:

$$e^i = \frac{1}{J} (e_j \times e_k) \quad e^i = J (e^i \times e^k) \quad i, j, k \text{ cyc } 1, 2, 3$$

### 2 Derivation of operators in the BOUT++ Clebsch system

The Clebsch system in BOUT++ goes like this

$$B = \nabla z \times \nabla x$$

$$= e^z \times e^x$$

$$J^{-1} e_y = e^z \times e^x$$

We have

$$B^\text{def} = B \cdot B = \sqrt{J^{-1} e_y \cdot J^{-1} e_y} = \sqrt{J^{-2} g_{yy}} = J^{-1} \sqrt{g_{yy}}$$

Further on

$$\frac{B}{B^\text{def}} = \frac{B}{J^{-1} g_{yy}} = \frac{e_y}{\sqrt{g_{yy}}}$$

### 2.1 The parallel and perpendicular gradients

We have that

$$\nabla = e^i \partial_i = e^x \partial_x + e^y \partial_y + e^z \partial_z$$

and that

$$\nabla_\parallel = (b \cdot \nabla) b = bb \cdot \nabla = \frac{e_y e_y}{g_{yy}} \cdot \nabla = \frac{e_y e_y}{g_{yy}} \cdot e^i \partial_i = \frac{e_y}{g_{yy}} \partial_y$$
so that

\[ \nabla_\perp = \nabla - \nabla_\parallel \]
\[ = e^x \partial_x + e^y \partial_y + e^z \partial_z - \frac{e_y}{g_{yy}} \partial_y \]
\[ = e^x \partial_x + e^y \partial_y + e^z \partial_z - \frac{g_{yx} e^x + g_{yy} e^y + g_{yz} e^z}{g_{yy}} \partial_y \]
\[ = e^x \left( \partial_x - \frac{g_{yx}}{g_{yy}} \partial_y \right) + e^z \left( \partial_z - \frac{g_{yz}}{g_{yy}} \partial_y \right) \]

**The perpendicular gradients in Laplacian inversion**

In the Laplacian inversion BOUT++ currently neglects the parallel \( y \) derivatives if \( g_{xy} \) and \( g_{yz} \) are non-zero, thus

\[ \nabla_\perp \simeq e^x \partial_x + e^z \partial_z \]  \( (12) \)

**2.2 The Laplacian**

We would here like to find an expression for the Laplacian

\[ \nabla^2 = \nabla \cdot \nabla \]

In general we have (using equation (2.6.39) in D’Haeseleer \[\text{haeseler}\])

\[ \nabla \cdot A = \frac{1}{J} \partial_i \left( J A^i \right) \]  \( (13) \)

and that

\[ A^i = A \cdot e^i \]

In our case \( A \rightarrow \nabla \), so that

\[ \nabla^i = (\nabla) \cdot e^i = e^i \cdot (\nabla) = e^i \cdot (e^j \partial_j) = g^{ij} \partial_j \]

Thus

\[ \nabla^2 = \frac{1}{J} \partial_i \left( J g^{ij} \partial_j \right) \]
\[ = \frac{1}{J} g^{ij} J \partial_i \partial_j + \frac{1}{J} \partial_i \left( J g^{ij} \right) \partial_j \]
\[ = g^{ij} \partial_i \partial_j + G^i \partial_j \]

where we have defined\(^1\)

\[ G^i = \frac{1}{J} \partial_i \left( J g^{ij} \right) \]
\[ = \frac{1}{J} \left( \partial_x \left[ J g^{xj} \right] + \partial_y \left[ J g^{yj} \right] + \partial_z \left[ J g^{zj} \right] \right) \]

\(^1\) Notice that \( G^i \) is not the same as the Christoffel symbols of second kind (also known as the connection coefficients or \( \Gamma^i_{jk} = e^i \cdot \partial_k e_j \)), although the derivation of the two are quite similar. | We find that \( \Gamma^i_{ji} = \epsilon^i \cdot \partial_k e_j = \nabla e_j \), whereas using equation (13) leads to \( G^i = \epsilon^i \cdot \partial_k e_j = \nabla e^i \), since \( g^{ij} = g^{ji} \) due to symmetry.
By writing the terms out, we get
\[ \nabla^2 = g^{ij} \partial_i \partial_j + G^i \partial_j \]
\[ = (g^{xx} \partial_x \partial_x + g^{yy} \partial_y \partial_y + g^{zz} \partial_z \partial_z) + (G^x \partial_x) \]
\[ + (g^{xy} \partial_x \partial_y + g^{yx} \partial_y \partial_x) + (G^y \partial_y) \]
\[ + (g^{xz} \partial_x \partial_z + g^{yz} \partial_y \partial_z + g^{zz} \partial_z \partial_z) + (G^z \partial_z) \]

We now use that the metric tensor is symmetric (by definition), so that \( g^{ij} = g^{ji} \), and \( g_{ij} = g_{ji} \), and that the partial derivatives commutes for smooth functions \( \partial_i \partial_j = \partial_j \partial_i \). This gives
\[ \nabla^2 = (g^{xx} \partial_x \partial_x) + (G^x \partial_x) \]
\[ + (g^{yy} \partial_y \partial_y) + (G^y \partial_y) \]
\[ + (g^{zz} \partial_z \partial_z) + (G^z \partial_z) \]
\[ + 2 (g^{xy} \partial_x \partial_y + g^{yx} \partial_y \partial_x) + (G^z \partial_x \partial_z + g^{zz} \partial_y \partial_z) \]

Notice that \( G^i \) does not operate on \( \partial_i \), but rather that the two are multiplied together.

### 2.3 The parallel Laplacian

We have that
\[ \nabla \parallel = (b \cdot \nabla) b = bb \cdot \nabla = \frac{e_y e_y}{g_{yy}} \cdot \nabla = \frac{e_x e_x}{g_{yy}} \cdot e^i \partial_i = \frac{e_y}{g_{yy}} \partial_y \]

we have that
\[ \nabla^i \parallel = \left( \frac{e_y}{g_{yy}} \partial_y \right) \cdot e^i = e^i \cdot \left( \frac{e_y}{g_{yy}} \partial_y \right) \]

so that by equation (13),
\[ \nabla^2 \parallel = \nabla \cdot (bb \cdot \nabla) \]
\[ = \nabla \cdot \left( \frac{e_y}{g_{yy}} \cdot \partial_y \right) \]
\[ = \frac{1}{J} \partial_x \left( Je^i \cdot \left[ \frac{e_y}{g_{yy}} \partial_y \right] \right) \]
\[ = \frac{1}{J} \partial_y \left( \frac{J}{g_{yy}} \partial_y \right) \]
.2.4 The perpendicular Laplacian

For the perpendicular Laplacian, we have that

\[
\nabla^2_\perp = \nabla^2 - \nabla^2_\parallel
\]

\[
= g^{ij}_x \partial_i \partial_j + G^j \partial_j - \frac{1}{J} \partial_y \left( \frac{J}{g_{yy}} \partial_y \right)
\]

\[
= \left( g^{xx}_x \partial_x^2 + \frac{1}{J} \partial_x \left\{ J g^{xx} \right\} + \partial_y \left\{ J g^{yx} \right\} + \partial_z \left\{ J g^{zx} \right\} \right) \partial_x
\]

\[
+ \left( g^{yy}_y \partial_y^2 + \frac{1}{J} \partial_x \left\{ J g^{yx} \right\} + \partial_y \left\{ J g^{yy} \right\} + \partial_z \left\{ J g^{zy} \right\} \right) \partial_y
\]

\[
+ \left( g^{zz}_z \partial_z^2 + \frac{1}{J} \partial_x \left\{ J g^{xz} \right\} + \partial_y \left\{ J g^{yz} \right\} + \partial_z \left\{ J g^{zz} \right\} \right) \partial_z
\]

\[
+ 2 \left( g^{xy} \partial_x \partial_y + g^{yx} \partial_y \partial_x + g^{yz} \partial_y \partial_z \right)
\]

\[
- \frac{1}{J} \partial_y \left( \frac{J}{g_{yy}} \partial_y \right)
\]

The perpendicular Laplacian in Laplacian inversion

Notice that BOUT++ currently assumes small parallel gradients in the dependent variable in Laplacian inversion if \( g_{xy} \) and \( g_{yz} \) are non-zero (if these are zero, the derivation can be done directly from equation (12) instead), so that

\[
\nabla^2_\perp \simeq \left( g^{xx}_x \partial_x^2 + \frac{1}{J} \partial_x \left\{ J g^{xx} \right\} + \partial_y \left\{ J g^{yx} \right\} + \partial_z \left\{ J g^{zx} \right\} \right) \partial_x
\]

\[
+ \left( g^{yy}_y \partial_y^2 + \frac{1}{J} \partial_x \left\{ J g^{yx} \right\} + \partial_y \left\{ J g^{yy} \right\} + \partial_z \left\{ J g^{zy} \right\} \right) \partial_y
\]

\[
+ \left( g^{zz}_z \partial_z^2 + \frac{1}{J} \partial_x \left\{ J g^{xz} \right\} + \partial_y \left\{ J g^{yz} \right\} + \partial_z \left\{ J g^{zz} \right\} \right) \partial_z
\]

\[
+ 2 \left( g^{xy} \partial_x \partial_y + g^{yx} \partial_y \partial_x + g^{yz} \partial_y \partial_z \right)
\]

\[
= \left( g^{xx}_x \partial_x^2 + G^x \partial_x \right) + \left( g^{yy}_y \partial_y^2 + G^y \partial_y \right) + \left( g^{zz}_z \partial_z^2 + 2 \left( g^{xy} \partial_x \partial_y \right) + G^z \partial_z \right)
\]

.2.5 The Poisson bracket operator

We will here derive the bracket operators, as they are used in BOUT++.
The electrostatic ExB velocity

Under electrostatic conditions, we have that \( v_E = -\frac{\nabla \phi \times b}{B} \), which is similar to \( v = k \times \nabla \psi \) found in incompressible fluid flow.

\[
v_E = -\frac{\nabla \phi \times b}{B} = -\frac{\nabla \phi \times e_y}{\sqrt{g_{yy} J} J^{-1}} = -\frac{J}{g_{yy}} \nabla \phi \times e_y
\]

\[
= \frac{J}{g_{yy}} e_y \times \nabla \phi
= \frac{J}{g_{yy}} (e_x \partial_x + e_y \partial_y + e_z \partial_z) \phi
\]

\[
= \frac{J}{g_{yy}} (g_{yx} e_x^2 + g_{yy} e_y^2 + g_{yz} e_z^2) \times (e_x^2 \partial_x + e_y^2 \partial_y + e_z^2 \partial_z) \phi
= \frac{J}{g_{yy}} (g_{yx} e_x^2 \times e_x^2 \partial_x + g_{yy} e_y^2 \times e_x^2 \partial_x + g_{yz} e_z^2 \times e_x^2 \partial_x
+ g_{yx} e_x^2 \times e_y^2 \partial_y + g_{yy} e_y^2 \times e_y^2 \partial_y + g_{yz} e_z^2 \times e_y^2 \partial_y
+ g_{yx} e_x^2 \times e_z^2 \partial_z + g_{yy} e_y^2 \times e_z^2 \partial_z + g_{yz} e_z^2 \times e_z^2 \partial_z) \phi
= \frac{J}{g_{yy}} (-g_{yy} e_y^2 \times e_x^2 \partial_x + g_{yz} e_z^2 \times e_x^2 \partial_x
+ g_{yx} e_x^2 \times e_y^2 \partial_y - g_{yz} e_z^2 \times e_y^2 \partial_y
- g_{yx} e_x^2 \times e_z^2 \partial_z + g_{yy} e_y^2 \times e_z^2 \partial_z) \phi
= \frac{1}{g_{yy}} (-g_{yy} e_x \partial_x + g_{yx} e_y \partial_y + g_{yz} e_z \partial_z - g_{yx} e_x \partial_y - g_{yy} e_y \partial_z - g_{yz} e_z \partial_x) \phi
\]

The electrostatic ExB advection

The electrostatic \( E \times B \) advection operator thus becomes

\[
v_E \cdot \nabla = -\frac{\nabla \phi \times b}{B} \cdot \nabla
= \frac{1}{g_{yy}} (-g_{yy} e_x \partial_x + g_{yx} e_y \partial_y + g_{yz} e_z \partial_z - g_{yx} e_x \partial_y - g_{yy} e_y \partial_z + g_{yz} e_z \partial_x) \phi \cdot (e_x^2 \partial_x + e_y^2 \partial_y + e_z^2 \partial_z)
= \frac{1}{g_{yy}} (-g_{yy} e_x \partial_x + g_{yx} e_y \partial_y + g_{yz} e_z \partial_z - g_{yx} e_x \partial_y - g_{yy} e_y \partial_z + g_{yz} e_z \partial_x) \phi
= \frac{1}{g_{yy}} (g_{yx} e_x \partial_x - g_{yx} e_y \partial_y) \phi \partial_x + g_{yz} e_z \partial_y \phi \partial_z - g_{yx} e_x \partial_y \phi \partial_y - g_{yy} \partial_z \phi \partial_x
= \frac{1}{g_{yy}} (g_{yx} \phi, x, y \{ \phi, \cdot \}_{y, z} + g_{yy} \{ \phi, \cdot \}_{z, y} + g_{yz} \{ \phi, \cdot \}_{x, y})
\]

Where we have used the definition of the Poisson bracket

\[
\{a, b\}_{i, j} = (\partial_i a) \partial_j b - (\partial_j a) \partial_i b
\]
The pure solenoidal advection is thus

\[ B \mathbf{v}_E \cdot \nabla = -\nabla \phi \times \mathbf{b} \cdot \nabla \]

\[ = b \times \nabla \phi \cdot \nabla \]

\[ = \frac{\sqrt{g_{yy}}}{g_{yy}} (g_{yx} \{ \phi, \cdot \}_y, z + g_{yy} \{ \phi, \cdot \}_z, x + g_{yz} \{ \phi, \cdot \}_x, y) \]  \hspace{1cm} (14)

\[ = \frac{1}{J \sqrt{g_{yy}}} (g_{yx} \{ \phi, \cdot \}_y, z + g_{yy} \{ \phi, \cdot \}_z, x + g_{yz} \{ \phi, \cdot \}_x, y) \]

### The brackets operator in BOUT++

Notice that the \((\phi,f)@\) operators in BOUT++ returns \(-\nabla \phi \times \mathbf{b} \mathbf{v}_E \cdot \nabla \) rather than \(-\nabla \phi \times \mathbf{b} \cdot \nabla f\).

Notice also that the Arakawa brackets neglects the \(\partial_y\) derivative terms (the \(y\)-derivative terms) if \(g_{xy}\) and \(g_{yz}\) are non-zero, so for the Arakawa brackets, BOUT++ returns

\[ \mathbf{v}_E \cdot \nabla = -\frac{\nabla \phi \times \mathbf{b}}{B} \cdot \nabla \]

\[ \simeq \frac{1}{g_{yy}} (g_{yy} \{ \phi, \cdot \}_z, x) \]

\[ = \partial_z \phi \partial_x - \partial_x \phi \partial_z \]

### .3 Divergence of ExB velocity

\[ \mathbf{v}_{E\times B} = \frac{\mathbf{b} \times \nabla \phi}{B} \]

Using

\[ \nabla \cdot (\mathbf{F} \times \mathbf{G}) = (\nabla \times \mathbf{F}) \cdot \mathbf{G} - \mathbf{F} \cdot (\nabla \times \mathbf{G}) \]

the divergence of the \(\mathbf{E} \times \mathbf{B}\) velocity can be written as

\[ \nabla \cdot \left( \frac{1}{B} \mathbf{b} \times \nabla \phi \right) = \left[ \nabla \times \left( \frac{1}{B} \mathbf{b} \right) \right] \cdot \nabla \phi - \frac{1}{B} \mathbf{b} \cdot \nabla \times \nabla \phi \]  \hspace{1cm} (15)

The second term on the right is identically zero (curl of a nablaient). The first term on the right can be expanded as

\[ \left[ \nabla \times \left( \frac{1}{B} \mathbf{b} \right) \right] \cdot \nabla \phi = \left[ \nabla \left( \frac{1}{B} \right) \times \mathbf{b} + \frac{1}{B} \nabla \times \mathbf{b} \right] \cdot \nabla \phi \]

Using

\[ \mathbf{b} \times \kappa = \nabla \times \mathbf{b} - \mathbf{b} [\mathbf{b} \cdot (\nabla \times \mathbf{b})] \]

this becomes:

\[ \nabla \cdot \left( \frac{1}{B} \mathbf{b} \times \nabla \phi \right) = - \mathbf{b} \times \nabla \left( \frac{1}{B} \right) \cdot \nabla \phi \]

\[ + \frac{1}{B} \mathbf{b} \times \kappa \cdot \nabla \phi \]

\[ + [\mathbf{b} \cdot (\nabla \times \mathbf{b})] \mathbf{b} \cdot \nabla \phi \]
Alternatively, equation (15) can be expanded as

\[
\nabla \cdot \left( \frac{1}{B} b \times \nabla \phi \right) = -B b \times \nabla \left( \frac{1}{B^2} \right) \cdot \nabla \phi + \frac{1}{B^2} \nabla \times B \cdot \nabla \phi \\
= -B b \times \nabla \left( \frac{1}{B^2} \right) \cdot \nabla \phi + \frac{1}{B^2} J \cdot \nabla \phi
\]

\[
\nabla \cdot \left( \frac{n \mathbf{b} \times \nabla \phi}{B} \right) = \frac{1}{J} \frac{\partial}{\partial \psi} \left( J_n \frac{\partial \phi}{\partial z} \right) - \frac{1}{J} \frac{\partial}{\partial z} \left( J_n \frac{\partial \phi}{\partial \psi} \right) \\
+ \frac{1}{J} \frac{\partial}{\partial \psi} \left( J_n g^{\psi \psi} g^{\psi z} \frac{\partial \phi}{\partial y} \right) - \frac{1}{J} \frac{\partial}{\partial y} \left( J_n g^{\psi \psi} g^{\psi z} \frac{\partial \phi}{\partial \psi} \right)
\]
BOUT++ PRECONDITIONING

Author B.Dudson, University of York

A.1 Introduction

This manual describes some of the ways BOUT++ could (and in some cases does) support preconditioning, Jacobian calculations and other methods to speed up simulations. This manual assumes that you’re familiar with how BOUT++ works internally.

Some notation: The ODE being solved is of the form

\[ \frac{\partial \mathbf{f}}{\partial t} = \mathbf{F}(\mathbf{f}) \]

Here the state vector \( \mathbf{f} = (f_0, f_1, f_2, \ldots)^T \) is a vector containing the evolving (3D) variables \( f_i(x, y, z) \).

The Jacobian of this system is then

\[ \mathbf{J} = \frac{\partial \mathbf{F}}{\partial \mathbf{f}} \]

The order of the elements in the vector \( \mathbf{f} \) is determined in the solver code and SUNDIALS, so here just assume that there exists a map \( \mathbb{I} \) between a global index \( k \) and (variable, position) i.e. \( (i, x, y, z) \)

\[ \mathbb{I} : (i, x, y, z) \mapsto k \]

and its inverse

\[ \mathbb{I}^{-1} : k \mapsto (i, x, y, z) \]

Some problem-specific operations which can be used to speed up the timestepping

1. Jacobian-vector multiply: Given a vector, multiply it by \( \mathbf{J} \)
2. Preconditioner multiply: Given a vector, multiply by an approximate inverse of \( M = \mathbb{I} - \gamma \mathbf{J} \)
3. Calculate the stencils i.e. non-zero elements in \( \mathbf{J} \)
4. Calculate the non-zero elements of \( \mathbf{J} \)
A.2 Physics problems

Some interesting physics problems of increasing difficulty

A.2.1 Resistive drift-interchange instability

A “simple” test problem of 2 fields, which results in non-trivial turbulent results. Supports resistive drift wave and interchange instabilities.

\[
\frac{\partial N_i}{\partial t} + \mathbf{v}_E \cdot \nabla N_i = 0
\]

\[
\frac{\partial \omega}{\partial t} + \mathbf{v}_E \cdot \nabla \omega = 2 \omega_{ci} \mathbf{b} \times \kappa \cdot \nabla P + N_i Z_i e \frac{4 \pi V^2}{c^2} \nabla ||j||
\]

\[
\nabla^2 \omega / N_i = \phi
\]

\[
0.51 \nu_{ci} ||j|| = \frac{e}{m_e} \partial ||\phi|| + \frac{T_e}{N_i m_e} \partial ||N_i||
\]

A.2.2 Reduced 3-field MHD

This is a 3-field system of pressure \( P \), magnetic flux \( \psi \) and vorticity \( \phi \):

\[
f = \begin{pmatrix} P \\ \psi \\ U \end{pmatrix}
\]

\[
\frac{\partial \psi}{\partial t} = -\frac{1}{B_0} \nabla ||\phi||
\]

\[
= -\frac{1}{B_0} [\mathbf{b}_0 - (\mathbf{b}_0 \times \nabla \psi)] \cdot \nabla \phi
\]

\[
= -\frac{1}{B_0} \mathbf{b}_0 \cdot \nabla \phi - \frac{1}{B_0} (\mathbf{b}_0 \times \nabla \phi) \cdot \nabla \phi
\]

\[
\Rightarrow \frac{d\psi}{dt} = -\frac{1}{B_0} \mathbf{b}_0 \cdot \nabla \phi
\]

The coupled set of equations to be solved are therefore

\[
\frac{1}{B_0} \nabla^2 \phi = U
\]

\[
\left( \frac{\partial}{\partial t} + \mathbf{v}_E \cdot \nabla \right) \psi = -\frac{1}{B_0} \mathbf{b}_0 \cdot \nabla \phi
\]

\[
\left( \frac{\partial}{\partial t} + \mathbf{v}_E \cdot \nabla \right) P = 0
\]

\[
\left( \frac{\partial}{\partial t} + \mathbf{v}_E \cdot \nabla \right) U = \frac{1}{\rho} B_0^2 [\mathbf{b}_0 - (\mathbf{b}_0 \times \nabla \psi)] \cdot \left( \frac{J_{||0}}{B_0} - \frac{1}{\mu_0} \nabla^2 \psi \right)
\]

\[
+ \frac{1}{\rho} \mathbf{b}_0 \times \kappa_0 \cdot \nabla P
\]

\[
\mathbf{v}_E = \frac{1}{B_0} \mathbf{b}_0 \times \nabla \phi
\]
The Jacobian of this system is therefore:

\[
J = \begin{bmatrix}
-\mathbf{v}_E \cdot \nabla & 0 & [\mathbf{b}_0 \times (P_0 + P) \cdot \nabla] \nabla^{-2}_\perp \\
0 & -\mathbf{v}_E \cdot \nabla & (\mathbf{b}_0 \cdot \nabla) \nabla^{-2}_\perp \\
2\mathbf{b}_0 \times \kappa_0 \cdot \nabla & -\frac{B_0^2}{\mu_0 \rho} (\mathbf{b}_0 - \mathbf{b}_0 \times \nabla\psi) \cdot \nabla^2 \nabla^{-2}_\perp + \frac{B_0^2}{\rho} \left[ \mathbf{b}_0 \times \nabla \left( \frac{J_{||}}{2B_0} \right) \right] \cdot \nabla & -\mathbf{v}_E \cdot \nabla \\
+ \frac{B_0^2}{\mu_0 \rho} \nabla \left( \nabla^2 \psi \right) \cdot (\mathbf{b}_0 \times \nabla) & & \\
\end{bmatrix} 
\]  

(1.1)

Where the blue terms are only included in nonlinear simulations.

This Jacobian has large dense blocks because of the Laplacian inversion terms (involving \( \nabla^{-2}_\perp \) which couples together all points in an X-Z plane. The way to make \( J \) sparse is to solve \( \psi \) as a constraint (using e.g. the IDA solver) which moves the Laplacian inversion to the preconditioner.

### A.2.3 Solving \( \psi \) as a constraint

The evolving state vector becomes

\[
f = \begin{pmatrix}
P \\
\psi \\
U \\
\phi
\end{pmatrix}
\]

### A.2.4 UEDGE equations

The UEDGE benchmark is a 4-field model with the following equations:

\[
\begin{align*}
\frac{\partial N_i}{\partial t} + V_{||} \frac{\partial}{\partial t} N_i &= -N_i \nabla_{||} V_{||} + \nabla_{||} (D_{||} \partial_{||} N_i)
\end{align*}
\]

\[
\begin{align*}
\frac{3}{2} \frac{\partial}{\partial t} (N_i T_e) &= \nabla_{||} (\kappa_i \partial_{||} T_e) + \nabla_{||} (N_i \chi_{||} \partial_{||} T_e)
\end{align*}
\]

This set of equations is good in that there is no inversion needed, and so the Jacobian is sparse everywhere. The state vector is

\[
f = \begin{pmatrix}
N_i \\
V_{||} \\
T_e \\
N_i T_i
\end{pmatrix}
\]

The Jacobian is:

\[
J = \begin{bmatrix}
-V_{||} \frac{\partial}{\partial t} - \nabla_{||} V_{||} + \nabla_{||} D_{||} \partial_{||} \psi & -\partial_{||} N_i - N_i \nabla_{||} \\
-\frac{1}{N_i} \frac{\partial N_i}{\partial t} & 0
\end{bmatrix}
\]

If instead the state vector is

\[
f = \begin{pmatrix}
N_i \\
N_i V_{||} \\
N_i T_e \\
N_i T_i
\end{pmatrix}
\]

then the Jacobian is

A.2. Physics problems

229
A.2.5 2-fluid turbulence

A.3 Jacobian-vector multiply

This is currently implemented into the CVODE (SUNDIALS) solver.

A.4 Preconditioner-vector multiply

A.4.1 Reduced 3-field MHD

The matrix $M$ to be inverted can therefore be written

$$
M = \begin{bmatrix}
D & 0 & U_P \\
L_P & D & U_\psi \\
\end{bmatrix}
$$

where

$$
D = I + \gamma v E \cdot \nabla
$$

For small flow velocities, the inverse of $D$ can be approximated using the Binomial theorem:

$$
D^{-1} \simeq I - \gamma v E \cdot \nabla
$$

(1.2)

Following [chacon-2008], [chacon-2002], $M$ can be re-written as

$$
M = \begin{bmatrix}
E & U \\
L & D \\
\end{bmatrix}
\Rightarrow
U = \begin{bmatrix}
U_P \\
U_\psi \\
\end{bmatrix}
\quad L = \begin{bmatrix}
L_P & L_\psi \\
\end{bmatrix}
$$

The Schur factorization of $M$ yields ([chacon-2008])

$$
M^{-1} = \begin{bmatrix}
E & U \\
L & D \\
\end{bmatrix}^{-1} = \begin{bmatrix}
I & -E^{-1}U \\
0 & I \\
\end{bmatrix} \begin{bmatrix}
E^{-1} & 0 \\
0 & F_{Schur}^{-1} \\
\end{bmatrix} \begin{bmatrix}
I & 0 \\
-LE^{-1} & I \\
\end{bmatrix}
$$

Where $F_{Schur} = D - LE^{-1}U$ is the Schur complement. Note that this inversion is exact so far. Since $E$ is block-diagonal, and $D$ can be easily approximated using equation (1.2), this simplifies the problem to inverting $F_{Schur}$, which is much smaller than $M$.

A possible approximation to $F_{Schur}$ is to neglect:

- All drive terms
  - the curvature term $L_P$
  - the $J_{||0}$ term in $L_\psi$
- All nonlinear terms (blue terms in equation (1.1)), including perpendicular terms (so $D = I$)

This gives

$$
F_{Schur} \simeq I + \gamma^2 \frac{\mu_0 B_0^2}{\rho} (b_0 \cdot \nabla) \nabla_\perp^2 (b_0 \cdot \nabla) \nabla_\perp^{-2} \\
\simeq I + \gamma^2 V_A^2 (b_0 \cdot \nabla)^2
$$

Where the commutation of parallel and perpendicular derivatives is also an approximation. This remaining term is just the shear Alfvén wave propagating along field-lines, the fastest wave supported by these equations.
A.5 Stencils

A.6 Jacobian calculation

The (sparse) Jacobian matrix elements can be calculated automatically from the physics code by keeping track of the (linearised) operations going through the RHS function.

For each point, keep the value (as usual), plus the non-zero elements in that row of $J$ and the constant: result = $Ax + b$

Keep track of elements using product rule.

```cpp
class Field3D {
    data[ngx][ngy][ngz]; // The data as now
    int JacIndex; // Variable index in Jacobian
    SparseMatrix *jac; // Set of rows for indices (JacIndex,*,*,*)
};
```

JacIndex is set by the solver, so for the system

$$f = \begin{pmatrix} P \\ \psi \\ U \end{pmatrix}$$

$P$.JacIndex = 0, psi.JacIndex = 1 and U.JacIndex = 2. All other fields are given JacIndex = -1.

SparseMatrix stores the non-zero Jacobian components for the set of rows corresponding to this variable. Evolving variables do not have an associated SparseMatrix object, but any fields which result from operations on evolving fields will have one.
APPENDIX

GEOMETRY AND DIFFERENTIAL OPERATOR

Author X. Q. Xu

B.1 Geometry

In a axisymmetric toroidal system, the magnetic field can be expressed as

\[ \mathbf{B} = \mathbf{I}(\psi) \nabla \zeta + \nabla \zeta \times \nabla \psi, \]

where \( \psi \) is the poloidal flux, \( \theta \) is the poloidal angle-like coordinate, and \( \zeta \) is the toroidal angle. Here, \( \mathbf{I}(\psi) = R B_t \).

The two important geometrical parameters are: the curvature, \( \kappa \), and the local pitch, \( \nu(\psi, \theta) \),

\[ \nu(\psi, \theta) = I(\psi) J / R^2. \]

The local pitch \( \nu(\psi, \theta) \) is related to the MHD safety \( q \) by \( \hat{q}(\psi) = 2\pi^{-1} \oint \nu(\psi, \theta) d\theta \) in the closed flux surface region, and \( \hat{q}(\psi) = 2\pi^{-1} \int_{\text{outboard}}^{\text{inboard}} \nu(\psi, \theta) d\theta \) in the scrape-off-layer. Here \( J = (\nabla \psi \times \nabla \theta \cdot \nabla \zeta)^{-1} \) is the coordinate Jacobian, \( R \) is the major radius, and \( Z \) is the vertical position.

B.2 Geometry and Differential Operators

In a axisymmetric toroidal system, the magnetic field can be expressed as \[ \mathbf{B} = \mathbf{I}(\psi) \nabla \zeta + \nabla \zeta \times \nabla \psi, \] where \( \psi \) is the poloidal flux, \( \theta \) is the poloidal angle-like coordinate, and \( \zeta \) is the toroidal angle. Here, \( \mathbf{I}(\psi) = R B_t \). The two important geometrical parameters are: the curvature, \( \kappa \), and the local pitch, \( \nu(\psi, \theta) \), and \( \nu(\psi, \theta) = I(\psi) J / R^2 \). The local pitch \( \nu(\psi, \theta) \) is related to the MHD safety \( q \) by \( \hat{q}(\psi) = 2\pi^{-1} \oint \nu(\psi, \theta) d\theta \) in the closed flux surface region, and \( \hat{q}(\psi) = 2\pi^{-1} \int_{\text{outboard}}^{\text{inboard}} \nu(\psi, \theta) d\theta \) in the scrape-off-layer. Here \( J = (\nabla \psi \times \nabla \theta \cdot \nabla \zeta)^{-1} \) is the coordinate Jacobian, \( R \) is the major radius, and \( Z \) is the vertical position.
B.2.1 Differential Operators

For such an axisymmetric equilibrium the metric coefficients are only functions of $\psi$ and $\theta$. Three spatial differential operators appear in the equations given as: $\mathbf{v}_E \cdot \nabla_\perp$, $\nabla_\parallel$ and $\nabla_\perp^2$.

\[
\nabla_\parallel = b_0 \cdot \nabla = \frac{1}{\mathcal{J}B} \frac{\partial}{\partial \theta} + \frac{I}{BR^2} \frac{\partial}{\partial \psi} = \frac{B_p}{hB} \frac{\partial}{\partial \theta} + \frac{B_t}{RB} \frac{\partial}{\partial \psi},
\]

\[
\mathcal{J} \nabla^2 = \frac{\partial}{\partial \psi} \left( \mathcal{J} J_{11} \frac{\partial}{\partial \psi} \right) + \frac{\partial}{\partial \psi} \left( \mathcal{J} J_{12} \frac{\partial}{\partial \psi} \right) + \frac{\partial}{\partial \theta} \left( \mathcal{J} J_{12} \frac{\partial}{\partial \theta} \right) + \frac{1}{R^2} \frac{\partial^2}{\partial \psi^2},
\]

\[
\nabla_\perp^2 = b_0 \cdot \nabla(b_0 \cdot \nabla) = \frac{1}{\mathcal{J}B} \frac{\partial}{\partial \theta} \left( \frac{1}{\mathcal{J}B} \frac{\partial}{\partial \theta} \right) + \frac{1}{\mathcal{J}B} \frac{\partial}{\partial \psi} \left( \frac{B_t}{RB} \frac{\partial}{\partial \psi} \right) + \frac{B_t}{\mathcal{J}RB} \frac{\partial^2}{\partial \theta \partial \psi} + \left( \frac{B_t}{\mathcal{J}RB} \right)^2 \frac{\partial^2}{\partial \psi^2},
\]

\[
\nabla_\perp^2 \Phi = -\nabla \cdot [b \times (b \times \nabla \Phi)] = \nabla^2 \Phi - (\nabla \cdot b)(b \cdot \nabla \Phi) - \nabla_\parallel^2 \Phi
\]

where the coordinate Jacobian and metric coefficients are defined as following:

\[
\mathcal{J} = \nabla \psi \times \nabla \theta \cdot \nabla \zeta = \frac{h}{B_p},
\]

\[
h = \sqrt{Z_\theta^2 + R_\theta^2},
\]

\[
J_{11} = |\nabla \psi|^2 = \frac{R^2}{\mathcal{J}^2}(Z_\theta^2 + R_\theta^2),
\]

\[
J_{12} = J_{21} = \nabla \psi \cdot \nabla \theta = -\frac{R^2}{\mathcal{J}^2}(Z_\theta Z_\psi + R_\theta R_\psi),
\]

\[
J_{13} = J_{31} = 0,
\]

\[
J_{22} = |\nabla \theta|^2 = \frac{R^2}{\mathcal{J}^2}(Z_\psi^2 + R_\psi^2),
\]

\[
J_{23} = J_{32} = 0,
\]

\[
J_{33} = |\nabla \zeta|^2 = \frac{1}{R^2}.
\]
B.2.2 Concentric circular cross section inside the separatrix without the SOL

For concentric circular cross section inside the separatrix without the SOL, the differential operators are reduced to:

\[
R = R_0 + r \cos \theta, \\
Z = r \sin \theta, \\
B_t = \frac{B_t R_0}{R}, \\
B_p = \frac{1}{R} \frac{\partial \psi}{\partial r}, \\
R_\psi = \cos \theta \frac{B_p}{R}, \\
R_\theta = -r \sin \theta, \\
Z_\psi = \frac{\sin \theta}{R} \frac{B_p}{R}, \\
Z_\theta = r \cos \theta, \\
\mathcal{J} = \frac{r}{B_p}, \\
h = r, \\
J_{11} = |\nabla \psi|^2 = r^2 B_p^2, \\
J_{12} = J_{21} = \nabla \psi \cdot \nabla \theta = 0, \\
J_{13} = J_{31} = 0, \\
J_{22} = |\nabla \theta|^2 = \frac{1}{r^2}, \\
J_{23} = J_{32} = 0, \\
J_{33} = |\nabla \zeta|^2 = \frac{1}{R^2}, \\
\nabla^2 \approx \frac{1}{r} \frac{\partial}{\partial r} \left( \frac{r}{R} \frac{\partial}{\partial r} \right) + \frac{1}{r^2} \frac{\partial^2}{\partial \theta^2} + \frac{1}{R^2} \frac{\partial^2}{\partial \zeta^2}.
\]

B.2.3 Field-aligned coordinates with \( \theta \) as the coordinate along the field line

A suitable coordinate mapping between field-aligned ballooning coordinates \((x, y, z)\) and the usual flux coordinates \((\psi, \theta, \zeta)\) is

\[
x = \psi - \psi_s, \\
y = \theta, \\
z = \zeta - \int_0^\theta \nu(x, y) dy.
\]

as shown in Fig. 1. The covering area given by the square ABCD in the usual flux coordinates is the same as the parallelogram ABEF in the field-aligned coordinates. The magnetic separatrix is denoted by \( \psi = \psi_s \). In this choice of coordinates, \( x \) is a flux surface label, \( y \), the poloidal angle, is also the coordinate along the field line, and \( z \) is a field line label within the flux surface.
The coordinate Jacobian and metric coefficients are defined as following:

\[
\mathcal{J} = \nabla \psi \times \nabla \theta \cdot \nabla \zeta = \frac{h}{B_p},
\]

\[
h = \sqrt{Z_\theta^2 + R_\theta^2},
\]

\[
\mathcal{J}_{11} = |\nabla x|^2 = \frac{R^2}{\mathcal{J}^2} (Z_\theta Z_\psi + R_\psi R_\theta),
\]

\[
\mathcal{J}_{12} = \mathcal{J}_{21} = \nabla x \cdot \nabla y = -\frac{R^2}{\mathcal{J}^2} (Z_\theta Z_\psi + R_\psi R_\theta),
\]

\[
\mathcal{J}_{22} = |\nabla y|^2 = \frac{R^2}{\mathcal{J}^2} (Z_\psi^2 + R_\psi^2),
\]

\[
\mathcal{J}_{13} = \mathcal{J}_{31} = \nabla x \cdot \nabla z = -\nu \nabla x \cdot \nabla y - |\nabla x|^2 \left( \int_{y_0}^y \frac{\partial \nu(x, y)}{\partial \psi} \, dy \right) = -|\nabla x|^2 I_s,
\]

\[
\mathcal{J}_{23} = \mathcal{J}_{32} = \nabla y \cdot \nabla z = -\nu |\nabla y|^2 - \nu \nabla x \cdot \nabla y \left( \int_{y_0}^y \frac{\partial \nu(x, y)}{\partial \psi} \, dy \right),
\]

\[
\mathcal{J}_{33} = |\nabla z|^2 = \left| \nabla \zeta - \nu \nabla \theta - \nabla \psi \right| \left( \int_{y_0}^y \frac{\partial \nu(x, y)}{\partial \psi} \, dy \right)^2,
\]

\[
I_s = \frac{\mathcal{J}_{12} |\nabla \psi|^2}{} + \left( \int_{y_0}^y \frac{\partial \nu(x, y)}{\partial \psi} \, dy \right).
\]

Here \( h \) is the local minor radius, \( I_s \) is the integrated local shear, and \( y_0 \) is an arbitrary integration parameter, which, depending on the choice of Jacobian, determines the location where \( I_s = 0 \). The disadvantage of this choice of coordinates is that the Jacobian diverges near the X-point as \( B_p \to 0 \) and its effect spreads over the entire flux surfaces near the separatrix as the results of coordinate transform \( z \). Therefore a better set of coordinates is needed for X-point divertor geometry. The derivatives are obtained from the chain rule as follows:

\[
\frac{d}{d\psi} = \frac{\partial}{\partial x} - \left( \int_{y_0}^y \frac{\partial \nu(x, y)}{\partial \psi} \, dy \right) \frac{\partial}{\partial z},
\]

\[
\frac{d}{d\theta} = \frac{\partial}{\partial y} - \nu(x, y) \frac{\partial}{\partial z},
\]

\[
\frac{d}{d\zeta} = \frac{\partial}{\partial z}.
\]

In the field-aligned ballooning coordinates, the parallel differential operator is simple, involving only one coordinate \( y \)

\[
\partial_y \parallel = b_0 \cdot \nabla_\parallel = \left( \frac{B_p}{h B} \right) \frac{\partial}{\partial y},
\]

which requires a few grid points. The total axisymmetric drift operator becomes

The perturbed \( \mathbf{E} \times \mathbf{B} \) drift operator becomes

\[
\delta \mathbf{v}_E \cdot \nabla_\perp = \frac{c}{B B_\parallel} \left\{ \frac{I}{\mathcal{J}} \frac{\partial \langle \delta \phi \rangle}{\partial \theta} + B_p^2 \frac{\partial \langle \delta \phi \rangle}{\partial z} \right\} \frac{\partial}{\partial \psi},
\]

\[
+ \frac{c}{B B_\parallel} \left\{ \frac{I}{\mathcal{J}} \frac{\partial \langle \delta \phi \rangle}{\partial \psi} + \frac{\mathcal{J}_{12}}{R^2} \frac{\partial \langle \delta \phi \rangle}{\partial z} \right\} \frac{\partial}{\partial \theta},
\]

\[
- \frac{c}{B B_\parallel} \left\{ B_p^2 \frac{\partial \langle \delta \phi \rangle}{\partial \psi} + \frac{\mathcal{J}_{12}}{R^2} \frac{\partial \langle \delta \phi \rangle}{\partial \theta} \right\} \frac{\partial}{\partial z},
\]

when the conventional turbulence ordering \( (k_\parallel \ll k_\perp) \) is used, the perturbed \( \mathbf{E} \times \mathbf{B} \) drift operator can be further reduced to a simple form

\[
\delta \mathbf{v}_E \cdot \nabla_\perp = \frac{c B}{B_\parallel} \left( \frac{\partial \langle \delta \phi \rangle}{\partial z} \frac{\partial}{\partial x} - \frac{\partial \langle \delta \phi \rangle}{\partial x} \frac{\partial}{\partial z} \right).
\]
where \( \partial / \partial \theta \simeq -\nu \partial / \partial z \) is used. In the perturbed \( \mathbf{E} \times \mathbf{B} \) drift operator the poloidal and radial derivatives are written in the usual flux \((\psi, \theta, \zeta)\) coordinates in order to have various options for valid discretizations. The general Laplacian operator for potential is

\[
\mathcal{J} \nabla^2 \Phi = \frac{\partial}{\partial x} \left( \mathcal{J} \mathcal{J}_{11} \frac{\partial \Phi}{\partial x} + \mathcal{J} \mathcal{J}_{12} \frac{\partial \Phi}{\partial y} + \mathcal{J} \mathcal{J}_{13} \frac{\partial \Phi}{\partial z} \right) \\
+ \frac{\partial}{\partial y} \left( \mathcal{J} \mathcal{J}_{21} \frac{\partial \Phi}{\partial x} + \mathcal{J} \mathcal{J}_{22} \frac{\partial \Phi}{\partial y} + \mathcal{J} \mathcal{J}_{23} \frac{\partial \Phi}{\partial z} \right) \\
+ \frac{\partial}{\partial z} \left( \mathcal{J} \mathcal{J}_{31} \frac{\partial \Phi}{\partial x} + \mathcal{J} \mathcal{J}_{32} \frac{\partial \Phi}{\partial y} + \mathcal{J} \mathcal{J}_{33} \frac{\partial \Phi}{\partial z} \right) .
\]

The general perpendicular Laplacian operator for potential is

\[
\mathcal{J} \nabla^2 \perp \Phi = \frac{\partial}{\partial x} \left( \mathcal{J} \mathcal{J}_{11} \frac{\partial \Phi}{\partial x} + \mathcal{J} \mathcal{J}_{13} \frac{\partial \Phi}{\partial z} \right) \\
+ \frac{\partial}{\partial z} \left( \mathcal{J} \mathcal{J}_{31} \frac{\partial \Phi}{\partial x} + \mathcal{J} \mathcal{J}_{33} \frac{\partial \Phi}{\partial z} \right) \\
- \left( \frac{B_p}{hB} \right) \frac{\partial}{\partial y} \left( \left( \frac{B_p}{hB} \right) \frac{\partial \Phi}{\partial y} \right) \\
- \left( \frac{B_p}{hB} \right)^2 \frac{\partial \ln B}{\partial y} \frac{\partial \Phi}{\partial y} .
\]

The general perpendicular Laplacian operator for axisymmetric potential \( \Phi_0(x, y) \) is

\[
\mathcal{J} \nabla^2 \perp \Phi_0 = \frac{\partial}{\partial x} \left( \mathcal{J} \mathcal{J}_{11} \frac{\partial \Phi_0}{\partial x} + \mathcal{J} \mathcal{J}_{13} \frac{\partial \Phi_0}{\partial z} \right) \\
+ \frac{\partial}{\partial y} \left( \mathcal{J} \mathcal{J}_{21} \frac{\partial \Phi_0}{\partial x} + \mathcal{J} \mathcal{J}_{23} \frac{\partial \Phi_0}{\partial z} \right) \\
- \left( \frac{B_p}{hB} \right) \frac{\partial}{\partial y} \left( \left( \frac{B_p}{hB} \right) \frac{\partial \Phi_0}{\partial y} \right) \\
- \left( \frac{B_p}{hB} \right)^2 \frac{\partial \ln B}{\partial y} \frac{\partial \Phi_0}{\partial y} .
\]

For the perturbed potential \( \delta \phi \), we can drop the \( \partial / \partial y \) terms in Eq. (69) due to the elongated nature of the turbulence \((k_\parallel / k_\perp \ll 1)\). The general perpendicular Laplacian operator for perturbed potential \( \delta \phi \) reduces to

\[
\mathcal{J} \nabla^2 \perp \delta \phi = \frac{\partial}{\partial x} \left( \mathcal{J} \mathcal{J}_{11} \frac{\partial \delta \phi}{\partial x} + \mathcal{J} \mathcal{J}_{13} \frac{\partial \delta \phi}{\partial z} \right) \\
+ \frac{\partial}{\partial z} \left( \mathcal{J} \mathcal{J}_{31} \frac{\partial \delta \phi}{\partial x} + \mathcal{J} \mathcal{J}_{33} \frac{\partial \delta \phi}{\partial z} \right) .
\]

If the non-split potential \( \Phi \) is a preferred option, the gyrokinetic Poisson equation (18) and the general perpendicular Laplacian operator Eq. (69) have to be used. Then the assumption \( k_\parallel / k_\perp \ll 1 \) is not used to simplify the perpendicular Laplacian operator.
CONTRIBUTING TO BOUT++

If you would like to help contribute to BOUT++ then there are many things you can do which will make a difference. There are projects large and small which a student or researcher could use to get started with BOUT++ and get more familiar with the code. You don’t need to be particularly familiar with BOUT++ or C++ to work on many of these. You can see a current list of outstanding bugs and feature requests on the GitHub issue page.

This and following sections describe the core of BOUT++, and are intended for anyone who wants to work on improving BOUT++. For a general introduction, and instructions for using BOUT++ see the User guide. The user’s guide assumes only minimal knowledge of C++, and provides only those details needed to use BOUT++.

We use doxygen comments to document the code in the source files. This is then built using breathe and sphinx into the document you’re reading now. The API documentation is in BOUT++ routines.

C.1 House rules

As production codes go, BOUT++ is not particularly big, but it is definitely large enough that keeping the code ‘clean’ and understandable is necessary. This is vital if many people are going to work on the code, and also greatly helps code debugging and verification. There are therefore a few house rules to keep in mind when modifying the BOUT++ code.

When modifying the core BOUT++ code, please keep in mind that this portion of the code is intended to be general (i.e. independent of any particular physical system of equations), and to be used by a wide range of users. Making code clear is also more important in this section than the physics model since the number of developers is potentially much greater.

Here are some rules for editing the core BOUT++ code:

- **NO FORTRAN. EVER.** Though it may be tempting for scientific programmers to use a little Fortran now and then, please please don’t put any into BOUT++. Use of Fortran, particularly when mixed with C/C++, is the cause of many problems in porting and modifying codes.

- If a feature is needed to study a particular system, only include it in the core code if it is more generally applicable, or cannot be put into the physics module.

- If you add a new feature, function, class member, etc. you must also include doxygen comments that explain what each new thing does. Similarly, if a change you make would affect e.g. a function’s arguments, please ensure that you keep the documentation up-to-date with the code. See the section on coding style for best practices in this regard. If you submit a pull request that doesn’t add or update documentation where appropriate, we may ask you to do so before it is merged.

- As well as documentation for new features, you must also include a representative test to ensure that it works correctly. Please see the tests README for more information on tests in BOUT++. Prefer to write unit tests that check the feature at the function level, rather than integrated tests that require setting up a whole physics model.
C.2 Development workflow using Git

The workflow we use is essentially “gitflow”.

- **master** should always be stable
- **next** contains bleeding-edge features

All work should be done in feature branches, branched off **next**. When complete, a pull request can be submitted.

At irregular intervals, we will create a **release** branch. No new features go into the **release** branch - only bug fixes and documentation.

1. Create a new branch
2. (Optional) Push it to Github to share and for backup
3. Make changes, commits
4. Submit a pull request into **next** using Github’s Pull Requests system

C.2.1 Creating a feature branch

First get a copy of the BOUT-dev repository (or git pull to update an existing copy):

```bash
git clone git@github.com:boutproject/BOUT-dev.git
cd BOUT-dev
```

Create a new branch **myfeature**, branching from **next**. Choose a descriptive name for **myfeature**, anything except “master” or “next”.

```bash
git checkout next
git pull
git checkout -b myfeature # Switched to a new branch "myfeature"
```

C.2.2 Pushing to Github

Create a fork on Github following the instructions [here](#).

If you want to push your branch to BOUT-dev to share with other developers, run:

```bash
git push -u yourfork myfeature
```

This command pushes **myfeature** to your fork (named **yourfork**) of the BOUT-dev repository, and the -u flag adds it as a remote tracking branch. After setting up the tracking branch, you can call “git push” without any parameters to push updates to **myfeature**.

If another developer wants to try out this branch, they will first need to add your repository as a new remote:

```bash
git remote add yourfork https://github.com/YourUsername/BOUT-dev.git
```

then they will be able to checkout your branch:

```bash
git checkout -b myfeature yourfork/myfeature
```

**Note:** If you have write access to the central BOUT-dev repository, you can push your branches there.
C.2.3 Making changes, commits

Now you would make changes, commit changes and push as usual:

```bash
... make changes ...
git add <files>
git commit
git push  # Pushes to origin/myfeature
```

You can switch between branches using `checkout`:

```bash
git checkout master  # Switch to "master"
git checkout myfeature  # Switch to "myfeature"
```

C.2.4 Merging into next

Once your feature is complete, ask other developers to have a look by creating a Pull Request on the BOUT-dev page. One of the maintainers will review your code and merge it into **next**. They may give you comments to improve the code. You can make additional changes and push them to the same feature branch and they will be automatically added to the pull request.

C.3 Coding Style

Code is read an order of magnitude more times than it is written. It’s also written for **people** and not for the computer! For these reasons, it’s important that we stick to some form of coding standards. The following coding style guidelines broadly follow the LLVM Coding Standards. The LLVM Coding Standards go into more depth, and explain the reasoning behind the guidelines more thoroughly than here. If you just follow the guidelines below, you won’t go far wrong though.

These guidelines are intended to make code easier to read and therefore easier to understand. Being consistent in coding style also helps comprehension by reducing cognitive load.

C.3.1 Comments

Comments in the code are vital to helping understanding. Comments that are embedded in the code should explain **why** something is done, rather than **how**.

For documenting what functions and classes do, we use **Doxygen**.

- Prefer C++ style comments // over C style /* */
- Doxygen comments: use ///</p>

Doxygen done right:

```c++
/// Foo the bar
///
/// Apply the standard foo method to @p bar
///
/// Typical usage:
///
/// foo(bar, "simple", result);
```

(continues on next page)
/// @param[in] quux Number of times to foo
/// @param[out] result filled with quux fooed bars
/// @returns true on success
bool applyFoo(BoutReal bar, int quux, std::vector<int> &result);

The header files are essentially the “public” API, so prefer to put Doxygen comments there, rather than in the implementation. “Private” functions, etc., can be documented in the implementation.

### C.3.2 Naming

Naming things correctly is super important! It is also one of the trickiest parts of coding. Names should be descriptive. Code is read an order of magnitude more often than it is written, so it is vital that it is easy to comprehend.

There are some conventions you should follow when naming things:

- Type or class names should be nouns and be PascalCase – e.g. BoutReal, BoutMesh, Laplacian
- Variable names should be nouns and snake_case – e.g. generator, forward_map, extra_yguards_lower
- Functions should be verbs (i.e. actions) and camelCase – e.g. solve, getSection, parseString

Prefer a longer descriptive name over a shorter abbreviated one: `inner_boundary_flags` rather than `inbndflgs`, `generator` rather than `gen`. It’s much easier to read and comprehend than the abbreviated form.

### C.3.3 Details

- Use spaces instead of tabs. Tabs may be interpreted differently by different editors, making the code look badly indented and difficult to read. The easiest solution is just use spaces everything instead.
- Two spaces for indentation
- Spaces after if, for, etc.

Wrong:

```cpp
if(expr){
    doSomething();
} else{
    doOtherThing();
}
```

Right:

```cpp
if (expr) {
    doSomething();
} else {
    doOtherThing();
}
```

This especially helps readability, making conditional statements stand out over function calls.
• Braces on same line as statement:

Wrong:

```cpp
def doFoo(bool expr)
{
    if (expr)
    {
        doSomething();
    }
    else
    {
        doOtherThing();
    }
}
```

Right:

```cpp
def doFoo(bool expr) {
    if (expr) {
        doSomething();
    } else {
        doOtherThing();
    }
}
```

This one is more style than readability - it’s the style that the majority of BOUT++ already uses.
The BOUT++ distribution is hosted on Github:
https://github.com/boutproject/BOUT-dev

For a full guide to using Git, see the git website or online tutorials. This manual just explains some basic ways to use Git, and the recommended work flow when working with BOUT++.

If you’re just starting with BOUT++, current developers will want to check your changes before submitting them to the repository. In this case you should fork the git repository, make any changes and then submit a pull request on Github. Fortunately Git makes this process quite easy: First get a copy of BOUT++:

```
$ git clone https://github.com/boutproject/BOUT-dev.git
```

The BOUT++ repository will now be in a directory called “BOUT-dev” (sorry - github doesn’t like ‘+’ in project names). To get the latest changes, use:

```
$ git pull
```

To see the status of the repository, commits etc. in a GUI use:

```
$ gitk
```

This is also useful for showing what changes you’ve made which need to be committed, or which haven’t yet been sent to the main repository.

You can make edits as normal, and commit them using:

```
$ git commit -a
```

which is pretty much the equivalent of svn commit in that it commits all changes, though importantly it doesn’t send them to a central server. To see which changes will be committed, use:

```
$ git status
```

To choose which files you want to commit, use:

```
$ git add file1, file2, ...
$ git commit
```

(Git can actually only commit selected parts of files if you want). To make using Git easier, you can create a config file $HOME/.gitconfig containing:
(though obviously you should change the name and email).

Once you’re done making changes, you should first pull the latest changes from the server:

```bash
$ git pull
```

**Read carefully** what git prints out. If there are conflicts then git will try to resolve them, but in some cases you will have to resolve them yourself. To see a list of conflicting changes run `git status` (or `git st` if you’re using the above `.gitconfig` file). Once you’ve finished resolving conflicts, run `git commit -a` to commit the merge.

### D.1 Accessing github from behind a firewall

If you’re working on a machine which can’t access github directly (such as grendel, smaug etc. at LLNL), you can still seamlessly access github by using another machine as a proxy over SSH. To do this, edit your SSH config file `.ssh/config` and add the following lines:

```
Host gh
 HostName github.com
 User git
 ProxyCommand ssh -q -x user@euclid.nersc.gov nc %h %p
```

where `euclid.nersc.gov` can be replaced by any machine you can access which has netcat (nc) installed, and which can access github.com. If you have set up a github account with SSH keys, you should now be able to get a copy of BOUT++ by running:

```bash
$ git clone gh:boutproject/BOUT-dev.git
```

### D.2 Creating a private repository

Whilst we would prefer it if improvements to BOUT++ were shared, sometimes you might want to keep changes private for a while before publishing them. Creating a private repository with Git is very simple, because every clone of a repository is itself a repository. Git doesn’t have the concept of a central repository, which can seem strange coming from the world of SVN and CVS. What it means is that you can create your own private repository anywhere you have access to. Sharing it with only some people means as giving them read or write access to the repository directory.

The following assumes you have a NERSC account and want to create a private repository on Franklin. To apply this to a different machine just replace `franklin.nersc.gov` with the machine you want to put the repository on.

1. SSH to `franklin.nersc.gov`, or wherever you want your repository:
2. Create a “bare” Git repository by cloning a repository with the \texttt{--bare} option:

\begin{verbatim}
$ cd ~
$ git clone --bare git@github.com:boutproject/BOUT-dev.git bout_private
\end{verbatim}

where you can replace \texttt{git@github.com:boutproject/BOUT-dev.git} with any other repository you can access. \texttt{bout_private} will be the name of the directory which will be created. This will make a repository without a working version. This means you can’t modify the code in it directly, but can pull and push changes to it. If you want to work on the code on Franklin, make a clone of your private repository:

\begin{verbatim}
$ git clone bout_private bout
\end{verbatim}

which creates a repository \texttt{bout} from your private repository. Running \texttt{git pull} and \texttt{git push} from within this new repository will exchange patches with your \texttt{bout_private} repository.

3. You can now clone, pull and push changes to your private repository over SSH e.g.:

\begin{verbatim}
$ git clone username@franklin.nersc.gov:bout_private
\end{verbatim}

4. To keep your private repository up to date you may want to pull changes from github into your private repository. To do this, you need to use a third repository. Log into Franklin again:

\begin{verbatim}
$ cd ~
$ git clone bout_private bout_tmp
\end{verbatim}

This creates a repository \texttt{bout_tmp} from your private repository. Now cd to the new directory and pull the latest changes from github:

\begin{verbatim}
$ cd bout_tmp
$ git pull git://github.com/boutproject/BOUT-dev.git
\end{verbatim}

Note: You should be able to access this repository from Franklin, but if not then see the previous subsection for how to access github from behind a firewall.

5. This pull might result in some conflicts which need to be resolved. If so, \texttt{git} will tell you, and running:

\begin{verbatim}
$ git status
\end{verbatim}

will give a list of files which need to be resolved. Edit each of the files listed, and when you’re happy commit the changes:

\begin{verbatim}
$ git commit -a
\end{verbatim}

6. Your \texttt{bout_tmp} directory now contains a merge of your private repository and the repository on github. To update your private repository, just push the changes back:

\begin{verbatim}
$ git push
\end{verbatim}

You can now delete the \texttt{bout_tmp} repository if you want.
BOUT++ is organised into classes and groups of functions which operate on them: It’s not purely object-oriented, but takes advantage of many of C++’s object-oriented features.

Fig. 5.1 shows the most important parts of BOUT++ and how they fit together.

The initialisation process is shown in red: basic information is first read from the grid file (e.g. size of the grid, topology etc.), then the user-supplied initialisation code is called. This code can read other variables from the grid, and makes at least one call to `PhysicsModel::bout_solve()` to specify a variable to be evolved. The main thing `bout_solve` does is to add these variables to the solver.

The process of running a timestep is shown in blue in Fig. 5.1: The main loop calls the solver, which in turn calls PVODE. To evolve the system PVODE makes calls to the RHS function inside solver. This moves data between PVODE and BOUT++, and calls the user-supplied `PhysicsModel::rhs()` code to calculate time-derivatives. Much of the work calculating time-derivatives involves differential operators.
Calculation of the **RHS function**, and handling of data in BOUT++ involves many different components. Fig. 5.2 shows (most) of the classes and functions involved, and the relationships between them. Some thought was put into how this should be organised, but it has also changed over time, so some parts could be cleaner.

**E.1 Directories**

The source code for the core of BOUT++ is divided into include files (which can be used in physics models) in `bout++/include`, and source code and low-level includes in `bout++/src`. Many parts of the code are defined by their interface, and can have multiple different implementations. An example is the time-integration solvers: many different implementations are available, some of which use external libraries, but all have the same interface and can be used interchangeably. This is reflected in the directory structure inside `bout++/src`. A common pattern is to store individual implementations of an interface in a subdirectory called `impls`.

```
include/foo.hxx
src/.../foo.cxx
src/.../foo_factory.hxx
src/.../foo_factory.cxx
src/.../impls/one/one.hxx
src/.../impls/one/one.cxx
```

where `foo.hxx` defines the interface, `foo.cxx` implements common functions used in several implementations. `foo_factory` creates new implementations, and is the only file which includes all the implementations. Individual implementations are stored in their own subdirectories of `impls`. Components which follow this pattern include fileio formats, invert/laplace and invert/parderiv inversion codes, mesh, and solver.
The current source code files are:

- **bout++.cxx**: Main file which initialises, runs and finalises BOUT++. Currently contains a `main()` function, though this is being removed shortly.

- **field**
  - `field2d.cxx` implements the `Field2D` class. This is a scalar field which varies only in $x$ and $y$ and is used for things like metric tensor components and initial profiles. It supplies lots of overloaded operators and functions on these objects.
  - `field3d.cxx` implements the `Field3D` class, which varies in $x, y$ and $z$. Since these handle a lot more memory than Field2D objects, the memory management is more complicated and includes reference counting. See section `Memory management` for more details.
  - `field_data.cxx` Implements some functions in the `FieldData` class. This is a mainly pure virtual interface class which is inherited by `Field2D` and `Field3D`.
  - `fieldperp.cxx` implements a `FieldPerp` class to store slices perpendicular to the magnetic field i.e. they are a function of $x$ and $z$ only. This is mainly used for Laplacian inversion routines, and needs to be integrated with the other fields better.
  - `initialprofiles.cxx` routines to set the initial values of fields when a simulation first starts. Reads settings from the option file based on the name of the variable.
  - `vecops.cxx` a collection of function to operate on vectors. Contains things like Grad, Div and Curl, and uses a combination of field differential operators (in `difops.cxx`) and metric tensor components (in `Mesh`).
  - `vec2d.cxx` implements the `Vector2D` class, which uses a `Field2D` object for each of its 3 components. Overloads operators to supply things like dot and cross products.
  - `vec3d.cxx` implements `Vector3D` by using a `Field3D` object for each component.
  - `where.cxx` supplies functions for choosing between values based on selection criteria.

- **fileio**
  - `datafile.cxx` supplies an abstract `DataFile` interface for data input and output. Handles the conversion of data in fields and vectors into blocks of data which are then sent to a specific file format.
  - `formatfactory.cxx`
  - `formatfactory.hxx`
  - `impls`
    * `emptyformat.hxx`
    * hdf5
      - `h5_format.cxx` implements an interface to the HDF5 library
      - `h5_format.hxx`
    * netcdf
      - `nc_format.cxx` implements an interface to the NetCDF-4 library
      - `nc_format.hxx`
    * netcdf4
      - `ncxx` implements an interface to the NetCDF-4 library using the C++ API
      - `ncxx`
    * `pnetcdf`
BOUT++ Documentation, Release 4.4.0

- `pnetcdf.cxx` Parallel NetCDF interface
  - `pnetcdf.hxx`

- **invert**
  - `fft_fftw.cxx` implements the `fft.hxx` interface by calling the Fastest Fourier Transform in the West (FFTW) library.

- **invert / laplace**
  - `invert_laplace.cxx` uses Fourier decomposition in $z$ combined with tri- and band-diagonal solvers in $x$ to solve Laplacian problems.
  - `laplacefactory.hxx`
  - `laplacefactory.cxx`
  - impls
    * serial_tri
      - `serial_tri.hxx`
      - `serial_tri.cxx`
    * serial_band
      - `serial_band.hxx`
      - `serial_band.cxx`
    * spt
      - `spt.hxx`
      - `spt.cxx`
    * pdd
      - `pdd.hxx`
      - `pdd.cxx`

- **invert / parderiv**
  - `invert_parderiv.cxx` inverts a problem involving only parallel $y$ derivatives. Intended for use in some preconditioners.
  - `parderiv_factory.hxx`
  - `parderiv_factory.cxx`
  - impls
    * serial
      - `serial.cxx`
      - `serial.hxx`
    * cyclic
      - `cyclic.cxx`
      - `cyclic.hxx`

- `lapack_routines.cxx` supplies an interface to the LAPACK linear solvers, which are used by the `invert_laplace` routines.
• mesh
  – *boundary_factory.cxx* creates boundary condition operators which can then be applied to fields. Described in section *Boundary factory*.
  – *boundary_region.cxx* implements a way to describe and iterate over boundary regions. Created by the mesh, and then used by boundary conditions. See section *Boundary regions* for more details.
  – *boundary_standard.cxx* implements some standard boundary operations and modifiers such as Neumann and Dirichlet.
  – *difops.cxx* is a collection of differential operators on scalar fields. It uses the differential methods in *derivs.cxx* and the metric tensor components in *Mesh* to compute operators.
  – *interpolation.cxx* contains functions for interpolating fields
  – *mesh.cxx* is the base class for the *Mesh* object. Contains routines useful for all *Mesh* implementations.
  – impls
    *bout
    · *boutmesh.cxx* implements a mesh interface which is compatible with BOUT grid files.
    · *boutmesh.hxx*

• physics
  – *gyro_average.cxx* gyro-averaging operators
  – *smoothing.cxx* provides smoothing routines on scalar fields
  – *sourcex.cxx* contains some useful routines for creating sources and sinks in physics equations.

• solver
  – *solver.cxx* is the interface for all solvers
  – *solverfactory.cxx* creates solver objects
  – *solverfactory.hxx*
  – impls
    * cvode
      · *cvode.cxx* is the implementation of *Solver* which interfaces with the SUNDIALS CVODE library.
      · *cvode.hxx*
    * ida
      · *ida.cxx* is the implementation which interfaces with the SUNDIALS IDA library
      · *ida.hxx*
    * petsc
      · *petsc.cxx* is the interface to the PETSc time integration routines
      · *petsc.hxx*
    * pvode
      · *pvode.cxx* interfaces with the 1998 (pre-SUNDIALS) version of PVODE (which became CVODE).
      · *pvode.hxx*

• sys
– `boutcomm.cxx`

– `boutexception.cxx` is an exception class which are used for error handling

– `derivs.cxx` contains basic derivative methods such as upwinding, central difference and WENO methods. These are then used by `difops.cxx`. Details are given in section sec-derivatives.

– `msg_stack.cxx` is part of the error handling system. It maintains a stack of messages which can be pushed onto the stack at the start of a function, then removed (popped) at the end. If an error occurs or a segmentation fault is caught then this stack is printed out and can help to find errors.

– `options.cxx` provides an interface to the BOUT.inp option file and the command-line options.

– `optionsreader.cxx`

– `output.cxx`

– `range.cxx` Provides the RangeIterator class, used to iterate over a set of ranges. Described in section Iterating over ranges

– `stencils.cxx` contains methods to operate on stencils which are used by differential methods.

– `timer.cxx` a class for timing parts of the code like communications and file I/O. Described in section Timing

– `utils.cxx` contains miscellaneous small useful routines such as allocating and freeing arrays.

– `options`

  * `optionparser.hxx`
  * `options_ini.cxx`
  * `options_ini.hxx`
ARRAYS, SCALAR AND VECTOR FIELD TYPES

The classes outlines in red in Fig. 5.2 are data types currently implemented in BOUT++.

**F.1 FieldData**

All BOUT++ data types implement a standard interface for accessing their data, which is then used in communication and file I/O code. This interface is in src/field/field_data.hxx. The mandatory (pure virtual) functions are:

```cpp
bool isReal(); // Returns true if field consists of real values
bool is3D() const; // True if variable is 3D
int byteSize() const; // Number of bytes for a single point
int realSize() const; // Number of reals (not implemented if not real)
```

To support file I/O there are also some additional functions which may be implemented. A code can check if they are implemented by calling `ioSupport`. If one of them is implemented then they all should be.

```cpp
bool ioSupport(); // Return true if these functions are implemented
const string getSuffix(int component) const; // For vectors e.g. "_x"
void* getMark() const; // Store current settings (e.g. co/contra-variant)
void setMark(void* setting); // Return to the stored settings
BoutReal* getData(int component);
void zeroComponent(int component); // Set a component to zero
```

For twist-shift conditions, the optional function `shiftZ` is called in the communication routines.

```cpp
void shiftZ(int jx, int jy, double zangle);
```

**F.2 Field**

The two main types are *Field2D*, and *Field3D*. Their main functions are to provide an easy way to manipulate data; they take care of all memory management, and most looping over grid-points in algebraic expressions. The 2D field implementation is relatively simple, but more optimisations are used in the 3D field implementation because they are much larger (factor of ∼ 100).

To handle time-derivatives, and enable expressions to be written in the following form:

```cpp
ddt(Ni) = -b0xGrad_dot_Grad(phi, Ni);
```

fields (and vectors, see below) have a function:
Field3D* timeDeriv();

which returns a pointer to the field holding the time-derivative of this variable. This function ensures that this field is unique using a singleton pattern.

A Field has meta-data members, which give:

- location is the location of the field values in a grid cell. May be unstaggered, CELL_CENTRE or staggered to one of the cell faces, CELL_XLOW, CELL_YLOW or CELL_ZLOW.

- directions gives the type of grid that the Field is defined on
  - directions.y is YDirectionType::Standard by default, but can be YDirectionType::Aligned if the Field has been transformed from an ‘orthogonal’ to a ‘field-aligned’ coordinate system.
  - directions.z is ZDirectionType::Standard by default, but can be ZDirectionType::Average if the Field represents a quantity that is averaged or constant in the z-direction (i.e. is a Field2D).

The meta-data members are written to the output files as attributes of the variables.

To create a new Field with meta-data, plus Mesh and Coordinates pointers copied from another one, and data allocated (so that the Field is ready to use) but not initialized, use the function emptyFrom(const T& f) which can act on Field3D, Field2D or FieldPerp. This is often used for example to create a result variable that will be returned from a function from the Field which is given as input, e.g.

Field3D exampleFunction(const Field3D& f) {
  Field3D result{emptyFrom(f)};
  ...
  < do things to calculate result >
  ...
  return result;
}

To zero-initialise the Field as well, use zeroFrom in place of emptyFrom. If a few of the meta-data members need to be changed, you can also chain setter methods to a Field. At the moment the available methods are setLocation(CELL_LOC), setDirectionY(YDirectionType) and setDirectionZ(ZDirectionType); also setIndex(int) for FieldPerp. For example, to set the location of result explicitly you could use

Field3D result{emptyFrom(f).setLocation(CELL_YLOW)};

F.3 Vector

Vector classes build on the field classes, just using a field to represent each component.

To handle time-derivatives of vectors, some care is needed to ensure that the time-derivative of each vector component points to the same field as the corresponding component of the time-derivative of the vector:

ddt(v.x) = ddt(v).x
F.4 dcomplex

Several parts of the BOUT++ code involve FFTs and are therefore much easier to write using complex numbers. Unfortunately, the C++ complex library also tries to define a real type, which is already defined by PVODE. Several work-arounds were tried, some of which worked on some systems, but it was easier in the end to just implement a new class dcomplex to handle complex numbers.

F.5 Memory management

This code has been thoroughly tested/debugged, and should only be altered with great care, since just about every other part of BOUT++ depends on this code working correctly. Two optimisations used in the data objects to speed up code execution are memory recycling, which eliminates allocation and freeing of memory; and copy-on-change, which minimises unnecessary copying of data.

Both of these optimisations are done “behind the scenes”, hidden from the remainder of the code, and are illustrated in Fig. 6.1:

Fig. 6.1: Memory handling in BOUT++. Memory allocation and freeing is eliminated by recycling memory blocks, and assignments without changes (A = B) do not result in copying data, only pointers to the data. Both these optimisations are handled internally, and are invisible to the programmer.

The objects (A,B,C) accessed by the user in operations discussed in the previous section act as an interface to underlying data (a,b). Memory recycling can be used because all the scalar fields are the same size (and vector fields are implemented as a set of 3 scalar fields). Each class implements a global stack of available memory blocks. When an object is assigned a value, it attempts to grab one of these memory blocks, and if none are available then a new block is allocated. When an object is destroyed, its memory block is not freed, but is put onto the stack. Since the evaluation of the time-derivatives involves the same set of operations each time, this system means that memory is only allocated the first time the time-derivatives are calculated, after which the same memory blocks are re-used. This eliminates the often slow system calls needed to allocate and free memory, replacing them with fast pointer manipulation.

Copy-on-change (reference counting) further reduces memory usage and unnecessary copying of data. When one field is set equal to another (e.g. Field3D A = B in Fig. 6.1), no data is copied, only the reference to the underlying data (in this case both A and B point to data block a). Only when one of these objects is modified is a second memory
block used to store the different value. This is particularly useful when returning objects from a routine. Usually this would involve copying data from one object to another, and then destroying the original copy. Using reference counting this copying is eliminated.

## F.6 Global field gather / scatter

In BOUT++ each processor performs calculations on a sub-set of the mesh, and communicates with other processors primarily through exchange of guard cells (the `mesh->communicate` function). If you need to gather data from the entire mesh onto a single processor, then this can be done using either 2D or 3D `GlobalFields`.

First include the header file

```cpp
#include <bout/globalfield.hxx>
```

which defines both `GlobalField2D` and `GlobalField3D`. To create a 3D global field, pass it the mesh pointer:

```cpp
GlobalField3D g3d(mesh);
```

By default all data will be gathered onto processor 0. To change this, specify which processor the data should go to as the second input

```cpp
GlobalField3D g3d(mesh, processor);
```

Gather and scatter methods are defined:

```cpp
Field3D localData;
// Set local data to some value

g3d.gather(localData);  // Gathers all data onto one processor

localData = g3d.scatter();  // Scatter data back
```

**Note:** Boundary guard cells are not handled by the scatter step, as this would mean handling branch-cuts etc. To obtain valid data in the guard and Y boundary cells, you will need to communicate and set Y boundaries.

**Note:** Gather and Scatter are global operations, so all processors must call these functions.

Once data has been gathered, it can be used on one processor. To check if the data is available, call the method `dataIsLocal()`, which will return `true` only on one processor

```cpp
if(g3d.dataIsLocal()) {
    // Data is available on this processor
}
```

The sizes of the global array are available through `xSize()`, `ySize()` and `zSize()` methods. The data itself can be accessed indirectly using `(x,y,z)` operators:

```cpp
for(int x=0; x<g3d.xSize(); x++)
    for(int y=0; y<g3d.ySize(); y++)
        for(int z=0; z<g3d.zSize(); z++)
            output.write("Value at (%d,%d,%d) is %e\n",
                        x,y,z,
                        g3d(x,y,z) );
```
or by getting a pointer to the underlying data, which is stored as a 1D array:

```c
BoutReal *data = g3d.getData();
nx = g3d.xSize();
ny = g3d.ySize();
nz = g3d.zSize();
data[x*ny*nz + y*nz + z]; // Value at g3d(x,y,z)
```

See the example examples/test-globalfield for more examples.

### F.7 Iterating over fields

The recommended way to iterate over a field is to use the `BOUT_FOR` macro:

```c
Field3D f(0.0);
BOUT_FOR(i, f.getMesh()->getRegion3D("RGN_ALL")) {
    f[i] = a[i] + b[i];
}
```

This expands into two nested loops, which have been designed to OpenMP parallelise and vectorise. Some tuning of this is possible, see below for details. It replaces the C-style triple-nested loop:

```c
Field3D f(0.0);
for (int i = mesh->xstart; i < mesh->xend; ++i) {
    for (int j = mesh->ystart; j < mesh->yend; ++j) {
        for (int k = 0; k < mesh->LocalNz; ++k) {
            f(i,j,k) = a(i,j,k) + b(i,j,k)
        }
    }
}
```

The region to iterate over can be over `Field2D`, `Field3D`, or `FieldPerp` domains, obtained by calling functions on `Mesh`: `getRegion2D("name")`, `getRegion3D("name")` and `getRegionPerp("name")` respectively. Currently the available regions include:

- `RGN_ALL`, which is the whole mesh;
- `RGN_NOBNDRY`, which skips all boundaries and guard cells;
- `RGN_GUARDS`, which is only guard cells, both boundary and communication cells;
- `RGN_NOX`, which skips the x boundaries and guard cells
- `RGN_NOY`, which skips the y boundaries and guard cells

New regions can be created and modified, see section below.

A standard C++ range for loop can also be used, but this is unlikely to OpenMP parallelise or vectorise:

```c
Field3D f(0.0);
for (auto i : f) {
    f[i] = a[i] + b[i];
}
```

If you wish to vectorise but can’t use OpenMP then there is a serial version of the macro:
BoutReal max=0.;
BOUT_FOR_SERIAL(i, region) {
    max = f[i] > max ? f[i] : max;
}

For loops inside parallel regions, there is BOUT_FOR_INNER:

Field3D f(0.0);
BOUT_OMP(parallel) {
    BOUT_FOR_INNER(i, f.getMesh()->getRegion3D("RGN_ALL")) {
        f[i] = a[i] + b[i];
    }
    ...}

If a more general OpenMP directive is needed, there is BOUT_FOR_OMP:

BoutReal result=0.;
BOUT_FOR_OMP(i, region, parallel for reduction(max:result)) {
    result = f[i] > result ? f[i] : result;
}

The iterator provides access to the x, y, z indices:

Field3D f(0.0);
BOUT_FOR(i, f.getMesh()->getRegion3D("RGN_ALL")) {
    f[i] = i.x() + i.y() + i.z();
}

Note that calculating these indices involves some overhead: The iterator uses a single index internally, so integer division and modulo operators are needed to calculate individual indices.

To perform finite difference or similar operators, index offsets can be calculated:

Field3D f = ...;
Field3D g(0.0);
BOUT_FOR(i, f.getMesh()->getRegion3D("RGN_NOBNDRY")) {
    g[i] = f[i.xp()] - f[i.xm()];
}

The xp() function by default produces an offset of +1 in X, xm() an offset of -1 in the X direction. These functions can also be given an optional step size argument e.g. xp(2) produces an offset of +2 in the X direction. There are also xpp(), which produces an offset of +2, xmm() an offset of -2, and similar functions exist for Y and Z directions. For other offsets there is a function offset(x,y,z) so that i.offset(1,0,1) is the index at (x+1,y,z+1).

Note that by default no bounds checking is performed. If the checking level is increased to 3 or above then bounds checks will be performed. This will have a significant (bad) impact on performance, so is just for debugging purposes. Configure with --enable-checks=3 option to do this.
F.7.1 Tuning BOUT_FOR loops

The BOUT_FOR macros use two nested loops: The outer loop is OpenMP parallelised, and iterates over contiguous blocks:

```
BOUT_OMP(parallel for schedule(guided))
for (auto block = region.getBlocks().cbegin();
    block < region.getBlocks().cend();
    ++block)
    for (auto index = block->first; index < block->second; ++index)
```

The inner loop iterates over a contiguous range of indices, which enables it to be vectorised by GCC and Intel compilers.

In order to OpenMP parallelise, there must be enough blocks to keep all threads busy. In order to vectorise, each of these blocks must be larger than the processor vector width, preferably several times larger. This can be tuned by setting the maximum block size, set at runtime using the `mesh:maxregionblocksize` option on the command line or in the BOUT.inp input file:

```
[mesh]
maxregionblocksize = 64
```

The default value is set in include/bout/region.hxx:

```
#define MAXREGIONBLOCKSIZE 64
```

By default a value of 64 is used, since this has been found to give good performance on typical x86_64 hardware. Some simple diagnostics are printed at the start of the BOUT++ output which may help. For example the blob2d example prints:

```
Registered region 3D RGN_ALL:
    Total blocks : 1040, min(count)/max(count) : 64 (1040)/ 64 (1040), Max imbalance : 1, Small block count : 0
```

In this case all blocks are the same size, so the Max imbalance (ratio of maximum to minimum block size) is 1. The Small block count is currently defined as the number of blocks with a size less than half the maximum block size. Ideally all blocks should be a similar size, so that work is evenly balanced between threads.

F.7.2 Creating new regions

Regions can be combined in various ways to create new regions. Adding regions together results in a region containing the union of the indices in both regions:

```
auto region = mesh->getRegion2D("RGN_NOBNDRY") + mesh->getRegion2D("RGN_BNDRY");
```

This new region could contain duplicated indices, so if unique points are required then the `unique` function can be used:

```
auto region = unique(mesh->getRegion2D("RGN_NOBNDRY") + mesh->getRegion2D("RGN_BNDRY"));
```

Currently the implementation of `unique` also sorts the indices, but if this changes in future there is also a `sort` function which ensures that indices are in ascending order. This can help improve the division into blocks of contiguous indices. Points can also be removed from regions using the `mask` function. This removes all points in the region which are in the mask (i.e. set subtraction):
auto region = mesh->getRegion2D("RGN_ALL").mask(mesh->getRegion2D("RGN_GUARDS"));

or:

auto region = mask(mesh->getRegion2D("RGN_ALL"), mesh->getRegion2D("RGN_GUARDS"));

The above example would produce a region containing all the indices in RGN_ALL which are not in RGN_GUARDS.

Currently creating new regions is a relatively slow process, so creating new regions should be done in the initialisation stages rather than in inner loops. Some of this overhead could be reduced with caching, but is not done yet.

One way to improve the performance, and make use of custom regions more convenient, is to register a new region in the mesh:

mesh->addRegion3D("Custom region",
                  mesh->getRegion3D("RGN_NOBNDRY") + mesh->getRegion3D("RGN_BNDRY"));

It is advisable, though not required, to register both 2D and 3D regions of the same name.

In the current implementation overwriting a region, by attempting to add a region which already exists, is not allowed, and will result in a BoutException being thrown. This restriction may be removed in future.

F.8 Iterating over ranges

The boundary of a processor’s domain may consist of a set of disjoint ranges, so the mesh needs a clean way to tell any code which depends on the boundary how to iterate over it. The RangeIterator class in include/bout/sys/range.hxx and src/sys/range.cxx provides this.

RangeIterator can represent a single continuous range, constructed by passing the minimum and maximum values.

RangeIterator it(1,4); // Range includes both end points
for(it.first(); !it.isDone(); it.next())
  cout << it.ind; // Prints 1234

A more canonical C++ style is also supported, using overloaded ++, *, and != operators:

for(it.first(); it != RangeIterator::end(); it++)
  cout << *it; // Prints 1234

where it++ is the same as it.next(), and *it the same as it.ind.

To iterate over several ranges, RangeIterator can be constructed with the next range as an argument:

RangeIterator it(1,4, RangeIterator(6,9));
for(it.first(); it != RangeIterator::end(); it++)
  cout << *it; // Prints 12346789

and these can be chained together to an arbitrary depth.

To support statements like:

for(RangeIterator it = mesh->iterateBndryLowerY(); !it.isDone(); it++)
  ...

the initial call to first() is optional, and everything is initialised in the constructor.
F.9 Field2D/Field3D Arithmetic Operators

The arithmetic operators (+, -, /, *) for Field2D and Field3D are generated automatically using the Jinja templating system. This requires Python 3 (2.7 may work, but only 3 is supported).

Because this is fairly low-level code, and we don’t expect it to change very much, the generated code is kept in the git repository. This has the benefit that Python and Jinja are not needed to build BOUT++, only to change the Field operator code.

**Warning:** You should not modify the generated code directly. Instead, modify the template and re-generate the code. If you commit changes to the template and/or driver, make sure to re-generate the code and commit it as well.

The Jinja template is in src/field/gen_fieldops.jinja, and the driver is src/field/gen_fieldops.py. The driver loops over every combination of BoutReal, Field2D, Field3D (collectively just “fields” here) with the arithmetic operators, and uses the template to generate the appropriate code. There is some logic in the template to handle certain combinations of the input fields: for example, for the binary infix operators, only check the two arguments are on identical meshes if neither is BoutReal.

To install Jinja:

```bash
$ pip3 install --user Jinja2
```

To re-generate the code, there is a make target for gen_fieldops.cxx in src/field/makefile. This also tries to apply clang-format in order to keep to a consistent code style.

**Note:** clang-format is bundled with clang. This should be available through your system package manager. If you do not have sufficient privileges on your system, you can install it from the source clang. One of the BOUT++ maintainers can help apply it for you too.
The mesh is used in pretty much all parts of the code, and deals with things like the geometry of the mesh (metric tensors etc.), and how the mesh is divided between processors (communications). The Mesh class defines an interface, and there is currently a single implementation:

- **BoutMesh** (src/mesh/boutmesh.cxx) which is backwards compatible with the BOUT and BOUT-06 codes.
  This is a logically rectangular mesh so the number of radial points (x) can’t change in the poloidal direction (y).

### G.1 Grid data sources

All data sources inherit from GridDataSource. They must supply a method to test if a variable exists, `GridDataSource::hasVar()`:

```cpp
bool hasVar(const string &name);
```

and then use the `get` methods to get integers or reals:

```cpp
bool get(Mesh *m, <type> &variable, const string &name);
```

### G.2 Loading a mesh

The Mesh constructor takes GridDataSource and Options objects. You can also call `Mesh::create()` with just one of these objects, which will call out to the MeshFactory singleton to create a mesh “automatically”. This is the way that it is done in `bout++.cxx`. Once you have instantiated a Mesh object, you can then call `Mesh::load()` to read in all the appropriate variables from the GridDataSource:

```cpp
mesh = Mesh::create();  ///< Create the mesh
mesh->load();           ///< Load from sources. Required for Field initialisation
```

For post-processing of the results, it’s useful to have mesh quantities in the dump files along with the results. To do this, there’s the function `Mesh::outputVars()` (see also Datafile and Options):

```cpp
// Create an output file from an Options object
dump = Datafile(options->getSection("output"));

// Possibly add some other variables to the output file
...
```

(continues on next page)
// Save mesh configuration into output file
mesh->outputVars(dump);

which is called during BOUT++ initialisation.

### G.2.1 Implementation: BoutMesh

*BoutMesh* class uses the BOUT indices (which trace back to UEDGE):

```cpp
int ixseps1, ixseps2, jyseps1_1, jyseps2_1, jyseps1_2, jyseps2_2;
```

*ixseps1* and *ixseps2* give the X location of the separatrices, and are equal in the case of single-null configurations. The indexing is such that all points $0 \leq x < ixseps1$ are inside the separatrix, whilst $ixseps1 \leq x < \text{LocalNx}$ are outside. See *BOUT++ Topology* for more details.

### G.3 Index ranges

The *Mesh* class includes several public members which describe the size of the mesh, and are used all over BOUT++ to loop over variables:

```cpp
/// Size of the mesh on this processor including guard/boundary cells
int LocalNx, LocalNy, LocalNz;

/// Local ranges of data (inclusive), excluding guard cells
int xstart, xend, ystart, yend;
```

### G.4 Getting data

The *Mesh::load()* code above needs to read data for the mesh, and physics codes usually need to read their initial profiles during initialisation. To do this, Mesh provides an overloaded function *Mesh::get()*:

```cpp
int get(var, const char *name); // Request data from mesh file
```

where *var* can be just about any BOUT++ datatype (*Field2D*, *Vector3D* etc.).

### G.4.1 Implementation: BoutMesh

For integers and BoutReals, the implementation is fairly trivial. Uses the Mesh protected functions to find a data source and read data from it:

```cpp
GridDataSource* s = findSource(name); // Find a source of data
s->open(name); // Open the source
bool success = s->fetch(&ival, name); // Get the data
s->close(); // Close the source
```

To read 2D and 3D fields, the branch-cuts need to be taken into account.
G.5 Communications

The most common type of communication is to just exchange all guard cells with neighboring processors. Mesh provides the following commands for doing this:

```
template <typename... Ts>
int communicate(Ts&... ts); // Communicate one or more fields
int communicate(FieldGroup); // Communicate a group of fields
comm_handle send(FieldGroup); // Send data
int wait(comm_handle); // Receive data
```

`Mesh::communicate()` can be used to communicate any number of variables together, and makes the code quite clear. For example in examples/DriftInstability/2fluid.cxx around line 360:

```
// Need to communicate jpar
mesh->communicate(jpar);
```

Since this uses the `FieldData` interface like Datafile, this can be used to communicate all BOUT++ field data types. You can also create a `FieldGroup` object to group fields together, then communicate them all together:

```
FieldGroup comgrp; // Group of variables for communication
Field3D P;
Vector3D V;
comgrp.add(P); // Add the variables
comgrp.add(V); // Usually done in PhysicsModel::init
mesh->communicate(comgrp); // Communicate in PhysicsModel::rhs
```

Internally, this is how the templated `Mesh::communicate()` works.

If you want to overlap communications with calculations then use the `Mesh::send()` and `Mesh::wait()` functions instead of `Mesh::communicate()`:

```
comm_handle ch = mesh->send(comgrp); // Start the communications
// Calculations which don't need variables in comgrp
wait(ch); // Wait for all communications to finish
```

G.5.1 Implementation: BoutMesh

In `BoutMesh`, the communication is controlled by the variables:

```
int UDATA_INDEST, UDATA_OUTDEST, UDATA_XSPLIT;
int DDATA_INDEST, DDATA_OUTDEST, DDATA_XSPLIT;
int IDATA_DEST, ODATA_DEST;
```

In the Y direction, each boundary region (Up and Down in Y) can be split into two, with $0 \leq x < \text{UDATA\_XSPLIT}$ going to the processor index `UDATA\_INDEST`, and `UDATA\_INDEST` $\leq x < \text{LocalNx}$ going to `UDATA\_OUTDEST`. Similarly for the Down boundary. Since there are no branch-cuts in the X direction, there is just one destination for the Inner and Outer boundaries. In all cases a negative processor number means that there’s a domain boundary.
G.6 X communications

For parallel Laplacian inversions, communication is needed in the X direction only, and involves quantities which are not in Fields:

```c
bool firstX(); // True if at the inner X boundary
bool lastX(); // True if at the outer X boundary
int NXPE, PE_XIND; // Number of processors in X, and X processor index
int sendXOut(BoutReal *buffer, int size, int tag);
sendXIn(BoutReal *buffer, int size, int tag);
comm_handle irecvXOut(BoutReal *buffer, int size, int tag);
comm_handle irecvXIn(BoutReal *buffer, int size, int tag);
```

The variables `Mesh::NXPE` and `Mesh::PE_XIND` shouldn’t really be there, but are currently needed because the SPT algorithm in `LaplaceSPT` needs to know when it’s going to be next and so keep track of which processor number is currently working. This logic to pass a problem along a chain in X should really be moved into Mesh.

G.7 Y-Z surface communications

Some operations (like parallel inversions in `bout++/src/invert/invert_parderiv.cxx`) need to be performed on Y-Z surfaces, i.e. slices at constant X. This needs to be able to handle open and closed surfaces, and that closed surfaces may need a shift in the Z direction to match one end onto the other (a twist-shift condition).

The simplest operation is to average a quantity over Y with `averageY()`.

To test if a particular surface is closed, there is the function `periodicY`.

The most general way to access data on surfaces is to use the `SurfaceIter` iterator, which can be created using `SurfaceIter::SurfaceIter()`:

```c
SurfaceIter* surface(mesh);
```

This then allows looping over the surfaces in the usual way:

```c
for(surf->first(); !surf->isDone(); surf->next()) {
    ...
}
```

To test if the surface is closed, there’s the test `SurfaceIter::closed()`:

```c
bool surf->closed(BoutReal &ts)
```

which returns true if the surface is closed, along with the twist-shift angle.
G.8 Initial profiles

The initial profiles code needs to construct a solution which is smooth everywhere, with a form of perturbation specified in the input file for each direction. In order to do this, it needs a continuous function to use as an index. This is supplied by the functions `Mesh::GlobalX()` and `Mesh::GlobalY()`:

```c
BoutReal GlobalX(int jx); // Continuous X index between 0 and 1
BoutReal GlobalY(int jy); // Continuous Y index (0 -> 1)
```

which take a local x or y index and return a globally continuous x or y index.

G.9 Differencing

The mesh spacing is given by the public members `Mesh::dx`, `Mesh::dy` and `Mesh::dx`:

```c
// These used for differential operators
Field2D dx, dy;
Field2D d2x, d2y; // 2nd-order correction for non-uniform meshes
BoutReal zlength, dz; // Derived from options (in radians)
```

G.10 Metrics

While `Mesh` handles the numerical details of the mesh, the “physical” details are handled by `Coordinates`. The contravariant and covariant metric tensor components are public members of `Coordinates`:

```c
// Contravariant metric tensor (g^{ij})
Field2D g11, g22, g33, g12, g13, g23; // These are read in grid.cxx

// Covariant metric tensor
Field2D g_11, g_22, g_33, g_12, g_13, g_23;

int calcCovariant(); // Invert contravariant metric to get covariant
int calcContravariant(); // Invert covariant metric to get contravariant
```

If only one of these sets is modified by an external code, then `Coordinates::calcCovariant()` and `Coordinates::calcContravariant()` can be used to calculate the other (uses Gauss-Jordan currently).

From the metric tensor components, `Coordinates` calculates several other useful quantities:

```c
int jacobian(); // Calculate J and Bxy
Field2D J; // Jacobian
Field2D Bxy; // Magnitude of B = nabla z times nabla x

/// Calculate differential geometry quantities from the metric tensor
int geometry();

// Christoffel symbol of the second kind (connection coefficients)
Field2D G1_11, G1_22, G1_33, G1_12, G1_13;
Field2D G2_11, G2_22, G2_33, G2_12, G2_23;
Field2D G3_11, G3_22, G3_33, G3_13, G3_23;
```

(continues on next page)
Field2D G1, G2, G3;

These quantities are public and accessible everywhere, but this is because they are needed in a lot of the code. They shouldn’t change after initialisation, unless the physics model starts doing fancy things with deforming meshes.

**G.11 Miscellaneous**

There are some public members of `Mesh` which are there for some specific task and don’t really go anywhere else (yet).

To perform radial derivatives in tokamak geometry, interpolation is needed in the Z direction. This is done by shifting in Z by a phase factor, performing the derivatives, then shifting back. The following public variables are currently used for this:

```plaintext
bool ShiftXderivs; // Use shifted X derivatives
int ShiftOrder;   // Order of shifted X derivative interpolation
Field2D zShift;   // Z shift for each point (radians)
Field2D ShiftTorsion; // d <pitch angle> / dx. Needed for vector differentials (Curl)
Field2D IntShiftTorsion; // Integrated shear (I in BOUT notation)
bool IncIntShear; // Include integrated shear (if shifting X)
int TwistOrder;   // Order of twist-shift interpolation
```

This determines what order method to use for the interpolation at the twist-shift location, with 0 meaning FFT during communication. Since this must be 0 at the moment it’s fairly redundant and should be removed.

A (currently experimental) feature is:

```plaintext
bool StaggerGrids; // Enable staggered grids (Centre, Lower). Otherwise all vars are cell centred (default).
```
BOUT++ needs to deal with binary format files to read the grid; read and write restart restart files; and write dump files. The two parts of the code which need to read and write data are therefore the grid routines (*grid.hxx*), and the *Datafile* class (*datafile.hxx* and *datafile.cxx*). All other parts which need to read or write data go through these methods.

Several different file formats are commonly used, such as HDF, HDF5, and netCDF. For historical reasons (inherited from BOUT), BOUT++ originally used the Portable Data Binary (PDB) format developed at LLNL. To separate the basic file format functions from the higher level grid and Datafile classes, these use an abstract class *DataFormat*. Any class which implements the functions listed in *dataformat.hxx* can therefore be passed to grid or datafile. This makes implementing a new file format, and switching between formats at run-time, relatively straightforward.

Access to data in files is provided using a Bridge pattern: The *Datafile* class provides an interface to the rest of the code to read and write variables, whilst file formats implement the *Dataformat* interface.

```cpp
class Datafile {
public:
    Datafile(Options *opt = nullptr, Mesh* mesh_in = nullptr);
    Datafile(Datafile &&other) noexcept;
    ~Datafile(); // need to delete filename
    Datafile& operator=(Datafile &&rhs) noexcept;
    Datafile& operator=(const Datafile &rhs) = delete;
    bool openr(const char *filename, ...);
    bool openw(const char *filename, ...); // Overwrites existing file
    bool opena(const char *filename, ...); // Appends if exists
    bool isValid(); // Checks if the data source is valid
    void close();
    void setLowPrecision(); //< Only output floats
    template <typename t>
    void addRepeat(t &value, std::string name){
        add(value,name.c_str(),true);
    }
    template <typename t>
    void addOnce(t &value, std::string name){
        add(value,name.c_str(),false);
    }
    void add(int &i, const char *name, bool save_repeat = false);
}
```

1 Support for PDB files was removed in BOUT++ 4.0.0
void add(BoutReal &r, const char *name, bool save_repeat = false);
void add(bool &b, const char* name, bool save_repeat = false);
void add(Field2D &f, const char *name, bool save_repeat = false);
void add(Field3D &f, const char *name, bool save_repeat = false);
void add(FieldPerp &f, const char *name, bool save_repeat = false);
void add(Vector2D &f, const char *name, bool save_repeat = false);
void add(Vector3D &f, const char *name, bool save_repeat = false);

bool read(); // Read data into added variables
bool write(); // Write added variables

/// Opens, writes, closes file
bool write(const char* filename, ...) const;

void setAttribute(const std::string &varname, const std::string &attrname, int value);
void setAttribute(const std::string &varname, const std::string &attrname, BoutReal value);

The important bits of the DataFormat interface are:

class DataFormat {
public:
    DataFormat(Mesh* mesh_in = nullptr);
    virtual ~DataFormat() {} 
    // File opening routines
    virtual bool openr(const char *name) = 0;
    virtual bool openr(const std::string &name) {
        return openr(name.c_str());
    }
    virtual bool openr(const std::string &base, int mype);
    virtual bool openw(const char *name, bool append=false) = 0;
    virtual bool openw(const std::string &name, bool append=false) {
        return openw(name.c_str(), append);
    }
    virtual bool openw(const std::string &base, int mype, bool append=false);

    virtual bool is_valid() = 0;
    virtual void close() = 0;
    virtual void flush() = 0;

    virtual const std::vector<int> getSize(const char *var) = 0;
    virtual const std::vector<int> getSize(const std::string &var) = 0;

    // Set the origin for all subsequent calls
    virtual bool setGlobalOrigin(int x = 0, int y = 0, int z = 0) = 0;
    virtual bool setLocalOrigin(int x = 0, int y = 0, int z = 0, int offset_x = 0, int offset_y = 0, int offset_z = 0);
};
virtual bool setRecord(int t) = 0; // negative -> latest

// Add a variable to the file
virtual bool addVarInt(const std::string &name, bool repeat) = 0;
virtual bool addVarBoutReal(const std::string &name, bool repeat) = 0;
virtual bool addVarField2D(const std::string &name, bool repeat) = 0;
virtual bool addVarField3D(const std::string &name, bool repeat) = 0;
virtual bool addVarFieldPerp(const std::string &name, bool repeat) = 0;

// Read / Write simple variables up to 3D
virtual bool read(int *var, const char *name, int lx = 1, int ly = 0, int lz = 0) = 0;
virtual bool read(int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) = 0;
virtual bool read(BoutReal *var, const char *name, int lx = 1, int ly = 0, int lz = 0) = 0;
virtual bool read(BoutReal *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) = 0;
virtual bool read_perp(BoutReal *var, const std::string &name, int lx = 1, int lz = 0) = 0;
virtual bool write(int *var, const char *name, int lx = 0, int ly = 0, int lz = 0) = 0;
virtual bool write(int *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) = 0;
virtual bool write(BoutReal *var, const char *name, int lx = 0, int ly = 0, int lz = 0) = 0;
virtual bool write(BoutReal *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) = 0;
virtual bool write_perp(BoutReal *var, const std::string &name, int lx = 0, int lz = 0) = 0;

// Read / Write record-based variables
virtual bool read_rec(int *var, const char *name, int lx = 1, int ly = 0, int lz = 0) = 0;
virtual bool read_rec(int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) = 0;
virtual bool read_rec(BoutReal *var, const char *name, int lx = 1, int ly = 0, int lz = 0) = 0;
virtual bool read_rec(BoutReal *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) = 0;
virtual bool read_rec_perp(BoutReal *var, const std::string &name, int lx = 1, int lz = 0) = 0;
virtual bool write_rec(int *var, const char *name, int lx = 0, int ly = 0, int lz = 0) = 0;
virtual bool write_rec(int *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) = 0;
virtual bool write_rec(BoutReal *var, const char *name, int lx = 0, int ly = 0, int lz = 0) = 0;
virtual bool write_rec(BoutReal *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) = 0;
virtual bool write_rec_perp(BoutReal *var, const std::string &name, int lx = 0, int lz = 0) = 0;
virtual bool write_rec_perp(BoutReal *var, const std::string &name, int lx = 0, int lz_, ...
˓→ = 0) = 0;

// Optional functions

virtual void setLowPrecision() { } // By default doesn't do anything

// Attributes

/// Sets a string attribute
/// Inputs
/// -----
/// @param[in] varname Variable name. The variable must already exist. If varname is the empty string "" then the attribute will be added to the file instead of to a variable.
/// @param[in] attrname Attribute name
/// @param[in] text A string attribute to attach to the variable
virtual void setAttribute(const std::string &varname, const std::string &attrname, const std::string &text) = 0;

/// Sets an integer attribute
/// Inputs
/// -----
/// @param[in] varname Variable name. The variable must already exist. If varname is the empty string "" then the attribute will be added to the file instead of to a variable.
/// @param[in] attrname Attribute name
/// @param[in] value An int attribute to attach to the variable
virtual void setAttribute(const std::string &varname, const std::string &attrname, int value) = 0;

/// Sets a BoutReal attribute
/// Inputs
/// -----
/// @param[in] varname Variable name. The variable must already exist. If varname is the empty string "" then the attribute will be added to the file instead of to a variable.
/// @param[in] attrname Attribute name
/// @param[in] value A BoutReal attribute to attach to the variable
virtual void setAttribute(const std::string &varname, const std::string &attrname, BoutReal value) = 0;

/// Gets a string attribute
/// Inputs
/// -------
/// @param[in] varname Variable name. The variable must already exist. If
c/// varname is the empty string "" then get the
c/// attribute from the top-level of the file instead
c/// of from a variable.
/// @param[in] attrname Attribute name
c///
/// Returns
c/// -------
/// text A string attribute of the variable
cvirtual bool getAttribute(const std::string &varname, const std::string &attrname,
c    std::string &text) = 0;

/// Gets an integer attribute
///
/// Inputs
/// -------
///
/// @param[in] varname Variable name. The variable must already exist. If
/// varname is the empty string "" then get the
/// attribute from the top-level of the file instead
/// of from a variable.
/// @param[in] attrname Attribute name
///
/// Returns
/// -------
/// value An int attribute of the variable
cvirtual bool getAttribute(const std::string &varname, const std::string &attrname, int &value) = 0;

/// Gets a BoutReal attribute
///
/// Inputs
/// -------
///
/// @param[in] varname Variable name. The variable must already exist. If
/// varname is the empty string "" then get the
/// attribute from the top-level of the file instead
/// of from a variable.
/// @param[in] attrname Attribute name
///
/// Returns
/// -------
/// value A BoutReal attribute of the variable
cvirtual bool getAttribute(const std::string &varname, const std::string &attrname, BoutReal &value) = 0;

/// Write out the meta-data of a field as attributes of the variable
void writeFieldAttributes(const std::string& name, const Field& f);
/// Overload for FieldPerp so we can also write 'yindex'
void writeFieldAttributes(const std::string& name, const FieldPerp& f);

/// Read the attributes of a field
void readFieldAttributes(const std::string& name, Field& f);
/// Overload for FieldPerp so we can also read 'yindex'
void readFieldAttributes(const std::string& name, FieldPerp& f);
}

H.1 FieldPerp I/O

FieldPerp objects can be saved to output files and read from them. The yindex of a FieldPerp is the local y-index on a certain processor, but is saved in output files as a global y-index in the attribute yindex_global. The intention is that a FieldPerp being saved should be a globally well-defined object, e.g. a set of values at one divertor target boundary, that will only be saved from processors holding that global y-index. The expectation is that the other processors would all save an invalid FieldPerp variable, with a yindex_global that is more negative than the lowest y-boundary guard cell\(^2\). The reason for saving the invalid FieldPerp variables is so that all variables are present in every dump file (even if they are not allocated or used); in particular the Python collect routine assumes that any variable will be found in the first output file, which collect uses to get its type and dimensions.

\(^2\) Actually, the C++ I/O code should work fine even if a FieldPerp object is defined with different y-indices on different processors. This may be useful for diagnostic or debugging purposes. However, Python routines like collect and restart.redistribute will fail because they find inconsistent yindex_global values.
NATURAL LANGUAGE SUPPORT

BOUT++ uses GNU gettext to provide translations of output strings. Configuration is described in Natural Language Support and running in Natural language support. Currently only fr, zh_TW, and zh_CN have been added, but it is quite easy to add more. See locale/README.md or below.

I.1 Marking strings for translation

In the code strings are wrapped with `_()` e.g. "hello world" becomes `_("hello world")`. Find a string you want to replace (which can include formatting like `%%d`), surround it with `_()`. Then in the locale directory:

```
make libbout.pot
```

will update the template file libbout.pot under BOUT_TOP/locale. The template file should not be edited, but is used to generate language-specific files (libbout.po). To update these language files see the next section.

I.2 Adding translations

Adding support for a new language, or improving the translations in the existing files can be done by:

1. Going to the locale BOUT++ subdirectory and running:

```
make locale-ll
```

where `ll` is the language code e.g. `make locale-zh_TW` or `make locale-de`. This will create a file libbout.po under a locale/ll subdirectory.

2. Edit the locale/ll/libbout.po file. Edit the .po file in de subdirectory (not the .pot file!), adding the translations. Each msgid entry should have a translated msgstr entry. If you don’t want to translate them all, just delete the ones you don’t translate. Any missing will just revert to the version in the code. If you’re adding UTF-8 characters, change the content line in the .po file to have charset=UTF-8.

3. In the locale directory run make. This should output something like:

```
Building language: fr
Building language: zh_CN
Building language: zh_TW
```

The new language should now be available (no need to recompile BOUT++).
PERFORMANCE PROFILING

Analyzing code behaviour is vital for getting the best performance from BOUT++. This is done by profiling the code, that is, building and running the code using tools that report the amount of time each processor spends in functions, on communications, etc.

This section describes how to compile and run BOUT++ using the Scorep/Scalasca and Extrae/Paraver tool chains. Both are suitable for analyzing code parallelized with MPI and/or OpenMP. Scorep+Scalasca gives timings and call trees for each processor/thread, while Extrae/Paraver produces visualizations showing what each processor/thread is doing at a point in time.

J.1 Scorep/Scalasca profiling

J.1.1 Instrumentation

Scorep automatically reports the time spend in MPI communications and OpenMP loops. However, to obtain information on the time spent in specific functions, it is necessary to instrument the source code. The macros to do this are provided in `scorepwrapper.hxx`.

To include a function in Scorep’s timing, include the scorep wrapper in the source code

```cpp
#include <bout/scorepwrapper.hxx>
```

and then write the macro `SCOREP0()` at the top of the function, e.g.

```cpp
int Field::getNx() const{
    SCOREP0();
    return getMesh()->LocalNx;
}
```

**Caution** Instrumenting a function makes it execute more slowly. This can result in misleading profiling information, particularly if fast-but-frequently-called functions are instrumented. Try to instrument significant functions only.

The profiling overhead in sensibly-instrumented code should be only a few percent of runtime.
J.1.2 Configure and build

Configure with `--with-scorep` to enable Scorep instrumentation, then build as normal. This option can be combined with other options, but it is usually desirable to profile the optimized code, configuring with the flags `--enable-optimize=3 --enable-checks=0`. Build the code with `make` as normal.

With CMake:

```
$ SCOREP_WRAPPER=off cmake \
  -DCMAKE_C_COMPILER=scorep-mpicc \
  -DCMAKE_CXX_COMPILER=scorep-mpicxx \
  <other CMake options>
```

This will turn off the instrumentation during the configure step. Please be aware that if you change `CMakeLists.txt`, CMake will try to automatically reconfigure the build, which the Score-P wrappers interfere with. In this case you will need to restart the configure step from scratch (i.e. remove the build directory and start again).

J.1.3 Run and analysis

When running the code, prepend the run command with `scalasca -analyze`, e.g.

```
$ scalasca -analyze mpirun -np 2 elm_pb
```

The run then produces an "archive" containing profiling data in a directory called `scorep_<exec_name>_<proc_info>_sum`. To view the profiling information with the cube viewer, do

```
$ cube scorep_<exec_name>_<proc_info>_<proc_info>_<proc_info>_{sum}profile.cubex
```

Note that Scorep does not run if doing so would produce an archive with the same name as an existing archive. Therefore to rerun an executable on the same number of processors, it is necessary to move or delete the first archive.

J.1.4 Machine-specific installation

These are some configurations which have been found to work on particular machines.

Archer

As of 23rd January 2019, the following configuration should work

```
$ module swap PrgEnv-cray PrgEnv-gnu
$ module load fftw
$ module load archer-netcdf/4.1.3
$ module load scalasca
```

Note that due to a bug in the CC compiler, it is necessary to modify `make.config` after configuration if profiling OpenMP-parallelized code:

- add the flag `-fopenmp` to `BOUT_FLAGS`
- add the flag `--thread=omp:ancestry` as an argument to `scorep` in `CXX`
J.2 Extrae/Paraver profiling

Extrae is a powerful tool allowing visualization of communication and computation in parallel codes. It requires minimal instrumentation; however the trace files produced can be extremely large.

J.2.1 Instrumentation, configure and build

No changes to the code are necessary. On some systems, environment variables must be set before building. Otherwise, compile and build as normal.

J.2.2 Run

To run, add a trace script into the normal run command, so that for example

```
$ aprun -n 16 blob2d -d delta_1
```

becomes

```
$ aprun -n 16 ./trace.sh blob2d -d delta_1
```

where `trace.sh` is the script file

```bash
#!/bin/bash
export EXTRAE_CONFIG_FILE=./extrae.xml
export LD_PRELOAD=${EXTRAE_HOME}/lib/libmpitrace.so
$*
```

The run directory must also contain the file `extrae.xml`, which configures which data Extrae collects. Example `extrae.xml` files may be found in `${EXTRAE_HOME}/share/example/*/extrae.xml`

Running produces a file called `TRACE.mpits`. To generate the `.prv` trace file that can be read by Paraver, do

```
TRACE_NAME=bout.prv
${EXTRAE_HOME}/bin/mpi2prv -f ${EXTRAE_WORK_DIR}/TRACE.mpits -o ${TRACE_NAME}
```

J.2.3 Analysis

Open the trace file in Paraver with

```
$ wxparaver ${TRACE_NAME}
```

To view time traces, go to File -> Load Configuration. There are many configurations to choose from! Two useful configurations are:

- `mpi/views/MPI_call.cfg` to show when MPI calls are made
- `General/views/useful_duration.cfg` to show continuous bursts of computation
Reducing trace file size

When trace files are very large, Paraver will prompt the user to filter or cut the file to reduce its size. Filtering removes some information from the trace, making it small enough to open and allow the user to select a region of interest. Cutting crops the trace to a region of interest. Both operations create new trace files, and never overwrite the original trace.

The following prescription should work for manipulating large trace files:

1. Open the large trace file in Paraver and click ‘Yes’ to filter it
2. Click on the tick box ‘Filter’
3. **Filter the trace file:**
   a) select box for Events
   b) select box for Communications
   c) in ‘Keep States’ select box for ‘Running’
   d) in ‘Keep States’ select box for ‘IO’
   e) select a min duration of 1000
   f) click ‘Apply’
4. View ‘useful duration’ configuration and locate the region of interest
5. Zoom into the region of interest, and start and end the zoom on equivalent large sections of computation (blue/green)
6. Right click -> Run -> Cutter
7. Change the ‘Input’ trace file to cut from the filtered to the original one.
8. Click cut.

This produces a trace file which has all the original profiling information, but is much smaller as it is limited in time to a region of interest.

### J.2.4 Machine-specific installation

These are some configurations which have been found to work on particular machines.

**Archer**

As of 1st February 2019, the following configuration should work

```
$ module swap PrgEnv-cray PrgEnv-gnu
$ module load fftw
$ module load archer-netcdf/4.1.3
$ module load papi
$ module load bsctools/extrae
$
$ export CRAYPE_LINK_TYPE=dynamic
```

Note that due to a bug in the CC compiler, it is necessary to modify `make.config` after configuration to add the flag `-fopenmp` to `BOUT_FLAGS`, when profiling OpenMP-parallelized code.
K.1 BOUT++ functions (alphabetical)

This is a list of functions which can be called by users writing a physics module. For a full list of functions, see the Reference manual, DOxygen documentation, and source code.

- Field = abs(Field | Vector)
- (Communicator).add(Field | Vector)
  Add a variable to a communicator object.
- apply_boundary(Field, name)
- Field = b0xGrad_dot_Grad(Field, Field, CELL_LOC)
- bout_solve(Field, Field, name)
- bout_solve(Vector, Vector, name)
- (Communicator).clear()
  Remove all variables from a Communicator object
- Field = cos(Field)
- Field = cosh(Field)
- Vector = Curl(Vector)
- Field = Delp2(Field)
  $\nabla^2$ operator
- Field = Div(Vector)
  Divergence of a vector
- Field = Div_par(Field f)
  Parallel divergence $B_0 b \cdot \nabla (f/B_0)$
- dump.add(Field, name, 1/0)
- Field = filter(Field, modenr)
- geometry_derivs()
  Calculates useful quantities from the metric tensor. Call this every time the metric tensor is changed.
- Vector = Grad(Field)
- Field = Grad_par(Field)
- Field = Grad2_par2(Field)
• grid_load(BoutReal, name)
  Load a scalar real from the grid file
• grid_load2d(Field2D, name)
  Load a 2D scalar field from the grid file
• grid_load3d(Field3D, name)
  Load a 3D scalar field from the grid file
• invert_laplace(Field input, Field output, flags, Field2D *A)
  Inverts an equation \( A \cdot x + B \cdot \text{Grad2}_\text{par2}(x) = r \)
• Field = invert_parderiv(Field2D|BoutReal A, Field2D|BoutReal B, Field3D r)
• Field = Laplacian(Field)
• Field3D = low_pass(Field3D, max_modenr)
• BoutReal = max(Field)
• BoutReal = min(Field)
• msg_stack.pop( |int)
  Remove a message from the top of the stack. If a message ID is passed, removes all messages back to that point.
• int = msg_stack.push(format, ...)
  Put a message onto the stack. Works like printf (and output.write).
• options.get(name, variable, default)
  Get an integer, real or boolean value from the options file. If not in the file, the default value is used. The value used is printed to log file.
• options.setSection(name) Set the section name in the input file
• output << values
  Behaves like cout for stream output
• output.write(format, ...)
  Behaves like printf for formatted output
• (Communicator).receive()
  Receive data from other processors. Must be preceded by a send call.
• (Communicator).run()
  Sends and receives data.
• (Communicator).send()
  Sends data to other processors (and posts receives). This must be followed by a call to receive() before calling send again, or adding new variables.
• (Field3D).setLocation(CELL_LOC)
• (Field3D).ShiftZ(bool)
• Field = sin(Field)
• Field = sinh(Field)
• solver.setPrecon(PhysicsPrecon)
  Set a preconditioner function
• Field = sqrt(Field)
• Field = tan(Field)
• Field = tanh(Field)
• Field = V_dot_Grad(Vector v, Field f)
  Calculates an advection term \( v \cdot \nabla f \)
• Vector = V_dot_Grad(Vector v, Vector u)
  Advection term \( v \cdot \nabla u \)
• Field = Vpar_Grad_par(Field v, Field f)
• Field3D = where(Field2D test, Field|BoutReal gt0, Field|BoutReal lt0)
  Chooses between two values, depending on sign of test.

K.2 File list

K.2.1 File adams_bashforth.cxx

K.2.2 File adams_bashforth.hxx

class AdamsBashforthSolver : public Solver

Public Functions

AdamsBashforthSolver(Options *options = nullptr)

~AdamsBashforthSolver() = default

virtual void resetInternalFields() override
  Should wipe out internal field vector and reset from current field object data.

virtual voidsetMaxTimestep(BoutReal dt) override
  Set a maximum internal timestep (only for explicit schemes)

inline virtual BoutReal getCurrentTimestep() override
  Return the current internal timestep.

virtual int init(int nout, BoutReal tstep) override
  Initialise the solver NOTE: nout and tstep should be passed to run, not init. Needed because of how the
  PETSc TS code works

virtual int run() override
  Run the solver, calling monitors nout times, at intervals of tstep. This function is called by solve(), and is
  specific to each solver type
  This should probably be protected, since it shouldn’t be called by users.
Private Functions

\textit{BoutReal} \texttt{take\_step(BoutReal \textit{timeIn}, BoutReal \textit{dt}, int \textit{order}, Array\textless BoutReal\textgreater \ &current, Array\textless BoutReal\textgreater \ &result)}

Private Members

\textit{Array\textless BoutReal\textgreater \ \textit{state}}
\textit{Array\textless BoutReal\textgreater \ \textit{nextState}}
\textit{std\textcolon deq\textless Array\textless BoutReal\textgreater \textgreater \ \textit{history}}
\textit{std\textcolon deq\textless BoutReal\textgreater \ \textit{times}}
\textit{BoutReal \ \textit{atol}}
\textit{BoutReal \ \textit{rtol}}
\textit{BoutReal \ \textit{max\_timestep}}
\textit{int \ \textit{mxstep}}
\textit{bool \ \textit{adaptive}}
\textit{bool \ \textit{adaptive\_order}}
\textit{bool \ \textit{followHighOrder}}
\textit{BoutReal \ \textit{dtFac}}
\textit{int \ \textit{maximum\_order}}
\textit{BoutReal \ \textit{timestep}}
\textit{BoutReal \ \textit{out\_timestep}}
\textit{int \ \textit{current\_order}}
\textit{int \ \textit{nsteps}}
\textit{int \ \textit{nlocal}}
\textit{int \ \textit{neq}}

K.2.3 File arkode.cxx

Defines

\texttt{ZERO}
\texttt{ONE}
**Typedefs**

using ARKODEINT = bout::utils::function_traits<ARKLocalFn>::arg_t<0>

**Functions**

static int arkode_rhs_explicit(BoutReal t, N_Vector u, N_Vector du, void *user_data)

static int arkode_rhs_implicit(BoutReal t, N_Vector u, N_Vector du, void *user_data)

static int arkode_rhs(BoutReal t, N_Vector u, N_Vector du, void *user_data)

static int arkode_bbd_rhs(ARKODEINT Nlocal, BoutReal t, N_Vector u, N_Vector du, void *user_data)

RHS function for BBD preconditioner.

static int arkode_pre(BoutReal t, N_Vector yy, N_Vector yp, N_Vector rvec, N_Vector zvec, BoutReal gamma, BoutReal delta, int lr, void *user_data)

Preconditioner function.

static inline int arkode_pre_shim(BoutReal t, N_Vector yy, N_Vector yp, N_Vector rvec, N_Vector zvec, BoutReal gamma, BoutReal delta, int lr, void *user_data, N_Vector tmp)

static int arkode_jac(N_Vector v, N_Vector Jv, realtype t, N_Vector y, N_Vector fy, void *user_data, N_Vector tmp)

Jacobian-vector multiplication function.

inline int ARKStepSetJacTimes(void *arkode_mem, std::nullptr_t, ARKSpilsJacTimesVecFn jtimes)

void *ARKStepCreate(ARKRhsFn fe, ARKRhsFn fi, BoutReal t0, N_Vector y0)

**Variables**

constexpr auto &ARKStepEvolve = ARKode

constexpr auto &ARKStepFree = ARKodeFree

constexpr auto &ARKStepGetCurrentTime = ARKodeGetCurrentTime

constexpr auto &ARKStepGetDky = ARKodeGetDky

constexpr auto &ARKStepGetLastStep = ARKodeGetLastStep

constexpr auto &ARKStepGetNumLinIters = ARKSpilsGetNumLinIters

constexpr auto &ARKStepGetNumNonlinSolvIters = ARKodeGetNumNonlinSolvIters

constexpr auto &ARKStepGetNumPrecEvals = ARKSpilsGetNumPrecEvals

constexpr auto &ARKStepGetNumRhsEvals = ARKodeGetNumRhsEvals

constexpr auto &ARKStepGetNumSteps = ARKodeGetNumSteps

constexpr auto &ARKStepReInit = ARKodeReInit

constexpr auto &ARKStepSSstolerances = ARKodeSSstolerances

K.2. File list 287
constexpr auto &ARKStepSVtolerances = ARKodeSVtolerances
constexpr auto &ARKStepSetAdaptivityMethod = ARKodeSetAdaptivityMethod
constexpr auto &ARKStepSetCFLFraction = ARKodeSetCFLFraction
constexpr auto &ARKStepSetEpsLin = ARKSpilsSetEpsLin
constexpr auto &ARKStepSetExplicit = ARKodeSetExplicit
constexpr auto &ARKStepSetFixedPoint = ARKodeSetFixedPoint
constexpr auto &ARKStepSetFixedStep = ARKodeSetFixedStep
constexpr auto &ARKStepSetImEx = ARKodeSetImEx
constexpr auto &ARKStepSetImplicit = ARKodeSetImplicit
constexpr auto &ARKStepSetInitStep = ARKodeSetInitStep
constexpr auto &ARKStepSetLinear = ARKodeSetLinear
constexpr auto &ARKStepSetMaxNumSteps = ARKodeSetMaxNumSteps
constexpr auto &ARKStepSetMaxStep = ARKodeSetMaxStep
constexpr auto &ARKStepSetMinStep = ARKodeSetMinStep
constexpr auto &ARKStepSetOptimalParams = ARKodeSetOptimalParams
constexpr auto &ARKStepSetOrder = ARKodeSetOrder
constexpr auto &ARKStepSetPreconditioner = ARKSpilsSetPreconditioner
constexpr auto &ARKStepSetUserData = ARKodeSetUserData

K.2.4 File arkode.hxx

class ArkodeSolver : public Solver

Public Functions

ArkodeSolver(Options *opts = nullptr)

~ArkodeSolver()

inline virtual BoutReal getCurrentTimestep() override
   Return the current internal timestep.

virtual int init(int nout, BoutReal tstep) override
   Initialise the solver NOTE: nout and tstep should be passed to run, not init. Needed because of how the
   PETSc TS code works

virtual int run() override
   Run the solver, calling monitors nout times, at intervals of tstep. This function is called by solve(), and is
   specific to each solver type
   This should probably be protected, since it shouldn’t be called by users.
BoutReal run(BoutReal tout)

void rhs_e(BoutReal t, BoutReal *udata, BoutReal *dudata)

void rhs_i(BoutReal t, BoutReal *udata, BoutReal *dudata)

void rhs(BoutReal t, BoutReal *udata, BoutReal *dudata)

void pre(BoutReal t, BoutReal gamma, BoutReal delta, BoutReal *udata, BoutReal *rvec, BoutReal *zvec)

void jac(BoutReal t, BoutReal *ydata, BoutReal *vdata, BoutReal *Jvdata)

Private Functions

void set_abstol_values(BoutReal *abstolvec_data, std::vector<BoutReal> &f2dtols, std::vector<BoutReal> &f3dtols)

void loop_abstol_values_op(Ind2D i2d, BoutReal *abstolvec_data, int &p, std::vector<BoutReal> &f2dtols, std::vector<BoutReal> &f3dtols, bool bndry)

Private Members

int NOUT

BoutReal TIMESTEP

BoutReal hcur

bool diagnose = {false}

N_Vector uvec = {nullptr}

void *arkode_mem = {nullptr}

BoutReal pre_Wtime = {0.0}

int pre_ncalls = {0}

int nsteps = {0}

int nfe_evals = {0}

int nfi_evals = {0}

int nniters = {0}

int npevals = {0}

int nliters = {0}
K.2.5 File array.hxx

Functions

template<typename T, typename Backing>
Array<T, Backing> copy(const Array<T, Backing> &other)
   Create a copy of an Array, which does not share data

template<typename T>
struct ArrayData
   #include <array.hxx> ArrayData holds the actual data Handles the allocation and deletion of data

Public Functions

inline ArrayData(int size)

inline ~ArrayData() const

inline iterator<T> begin() const

inline iterator<T> end() const

inline int size() const

inline void operator=(ArrayData<T> &in)

inline T &operator[](int ind)

Private Members

int len
   Size of the array.

T *data
   Array of data.

template<typename T, typename Backing = ArrayData<T>>
class Array
   #include <array.hxx> Data array type with automatic memory management
   This implements a container similar to std::vector but with reference counting like a smart pointer and custom memory management to minimise new and delete calls
   This can be used as an alternative to static arrays
   Array<dcomplex> vals(100); // 100 complex numbers
   vals[10] = 1.0; // ok
When an `Array` goes out of scope or is deleted, the underlying memory (dataBlock/Backing) is put into a map, rather than being freed. If the same size arrays are used repeatedly then this avoids the need to use new and delete.

This behaviour can be disabled by calling the static function `useStore`:

```cpp
Array<dcomplex>::useStore(false); // Disables memory store
```

The second template argument determines what type of container to use to store data. This defaults to a custom struct but can be `std::valarray` (provided T is a compatible type), `std::vector` etc. Must provide the following: size, operator=, operator[], begin, end

**Public Types**

```cpp
using data_type = T
using backing_type = Backing
using size_type = int
```

**Public Functions**

```cpp
inline Array() noexcept
Create an empty array
    Array a(); a.empty(); // True

inline Array(size_type len)
Create an array of given length

inline ~Array() noexcept
Destructor. Releases the underlying dataBlock

inline Array(const Array &other) noexcept
Copy constructor

inline Array &operator=(Array other) noexcept
Assignment operator After this both Arrays share the same dataBlock
    Uses copy-and-swap idiom

inline Array(Array &&other) noexcept
Move constructor

inline void reallocate(size_type new_size)
Reallocate the array with size = new_size
    Note that this invalidates the existing data!

inline void clear() noexcept
Release data. After this the Array is empty and any data access will be invalid

inline bool empty() const noexcept
Returns true if the Array is empty

inline size_type size() const noexcept
Return size of the array. Zero if the array is empty.

inline bool unique() const noexcept
Returns true if the data is unique to this Array.
inline void ensureUnique()
Ensures that this *Array* does not share data with another. This should be called before performing any write operations on the data.

inline iterator<T> begin() noexcept

inline iterator<T> end() noexcept

inline const_iterator<T> begin() const noexcept

inline const_iterator<T> end() const noexcept

inline T &operator[](size_type ind)
Access a data element. This will fail if the *Array* is empty (so ptr is null), or if ind is out of bounds. For efficiency no checking is performed, so the user should perform checks.

inline const T &operator[](size_type ind) const

**Public Static Functions**

static inline bool useStore(bool keep_using = true) noexcept
Holds a static variable which controls whether memory blocks (dataBlock) are put into a store or new/deleted each time.

The variable is initialised to true on first use, but can be set to false by passing “false” as input. Once set to false it can’t be changed back to true.

static inline void cleanup()
Delete all data from the store and disable the store

Note: After this is called the store cannot be re-enabled

**Private Types**

using dataBlock = *Backing*
using dataPtrType = std::shared_ptr<dataBlock>*
using storeType = std::map<size_type, std::vector<dataPtrType>>*
using arenaType = std::vector<storeType>*

**Private Functions**

inline dataPtrType get(size_type len)
Returns a pointer to a dataBlock object of size *len* with no references. This is either from the store, or newly allocated

   Expects *len* >= 0
inline void release(dataPtrType &d) noexcept

Release an dataBlock object, reducing its reference count by one. If no more references, then put back into the store. It’s important to pass a reference to the pointer, otherwise we get a copy of the shared_ptr, which therefore increases the use count and doesn’t allow us to free the pass pointer directly.

Note that this is noexcept only because we’ve ensure that both a) store()[<size>] already exists, and b) it has space for at least one data block. Of course, store() could throw in which case we’re doomed anyway, so the only thing we can do is abort.

Private Members

dataPtrType ptr

Pointer to the data container object owned by this Array. May be null.

Private Static Functions

static inline storeType &store(bool cleanup = false)

This maps from array size (size_type) to vectors of pointers to dataBlock objects.

By putting the static store inside a function it is initialised on first use, and doesn’t need to be separately declared for each type T.

Inputs

Parameters cleanup – [in] If set to true, deletes all dataBlock and clears the store.

Friends

inline friend friend void swap (Array< T > &first, Array< T > &second) noexcept

Exchange contents with another Array of the same type. Sizes of the arrays may differ.

K.2.6 File assert.hxx

Defines

CHECKLEVEL

Defines a macro ASSERT which throws a BoutException if a given condition is false. Whether the assertion is tested depends on the checking level, so assertions can be removed for optimised runs.

ASSERT<level>( condition )

level - An integer known at compile time. condition tested if level >= CHECK

collection - The expression to test

e.g. ASSERT2( condition ) will only test condition if CHECK >= 2

ASSERT0(condition)

ASSERT1(condition)
class BoundaryFactory
#include <boundary_factory.hxx> Create BoundaryOp objects on demand.

This implements a simple string parser, used to match boundary condition names like “dirichlet” with a BoundaryOp object.

Modifiers: Simple modifications of boundary conditions can be performed, for example transforming the coordinate system

This is a singleton, so only one instance can exist. This is enforced by making the constructor private, and having a getInstance() method to return a pointer to the only instance.

Example

Boundaries are defined as classes which inherit from BoundaryOp. These define a clone() function which creates a new BoundaryOp, given a list of arguments. See boundary_standard.hxx for examples.

```cpp
class MyBoundary : public BoundaryOp {
public:
    BoundaryOp* clone(BoundaryRegion *region, const list<string> &args) {
        // Decide what to do with arguments
        return new MyBoundary();
    }
    void apply(Field2D &f);
    void apply(Field3D &f);
};
```

The singleton instance of BoundaryFactory from getInstance():

```cpp
BoundaryFactory* bf = BoundaryFactory::getInstance();
```

New boundary types can be added to the BoundaryFactory

```cpp
bf->add(new MyBoundary, "myboundary");
```

Subsequent calls to create() or createFromOptions() can make use of the boundary type “myboundary”.

```cpp
BoundaryOpBase *bndry = bf->create("myboundary()"); new BoundaryRegionXOut("xout", 0, 10, localmesh);
```

where the region is defined in boundary_region.hxx
**Public Functions**

~BoundaryFactory()

*BoundaryOpBase* `create` (const `std::string &name`, `BoundaryRegionBase *region`)  
Create a boundary operation object.

*BoundaryOpBase* `create` (const `char *name`, `BoundaryRegionBase *region`)  

*BoundaryOpBase* `createFromOptions` (const `std::string &varname`, `BoundaryRegionBase *region`)  
Create a boundary object using the options file.

*BoundaryOpBase* `createFromOptions` (const `char *varname`, `BoundaryRegionBase *region`)  

void `add` (`BoundaryOp *bop`, const `std::string &name`)  
Add available boundary conditions and modifiers. Supply an object, and the name to be used.

void `add` (`BoundaryOp *bop`, const `char *name`)  
Add a boundary condition.  
Note: This method should be removed, as the string method is sufficient.

void `addMod` (`BoundaryModifier *bmod`, const `std::string &name`)  
Add a boundary condition modifier.

void `addMod` (`BoundaryModifier *bmod`, const `char *name`)  
Note: This method should be removed, as the string method is sufficient.

void `add` (`BoundaryOpPar *bop`, const `std::string &name`)  

`add` (`BoundaryOpPar *bop`, const `char *name`)  

**Public Static Functions**

static `BoundaryFactory *getInstance()`  
Return a pointer to the only instance.

static void `cleanup()`  
Frees all memory.

**Private Functions**

`BoundaryFactory()`  
Private constructor, preventing instantiation of this class.

`BoundaryOp *findBoundaryOp` (const `std::string &s`)  

`BoundaryModifier *findBoundaryMod` (const `std::string &s`)  

`BoundaryOpPar *findBoundaryOpPar` (const `std::string &s`)  

K.2. File list 295
**Private Members**

```cpp
std::map<std::string, BoundaryOp*> opmap
std::map<std::string, BoundaryModifier*> modmap
std::map<std::string, BoundaryOpPar*> par_opmap
```

**Private Static Attributes**

```cpp
static BoundaryFactory *instance = nullptr
```

The only instance of this class (Singleton)

---

**K.2.10 File boundary_op.hxx**

class **BoundaryOpBase**

Subclassed by `BoundaryOp`, `BoundaryOpPar`

**Public Functions**

```cpp
BoundaryOpBase() = default
```

```cpp
virtual ~BoundaryOpBase() = default
```

```cpp
virtual void apply(Field2D &f) = 0
```

Apply a boundary condition on field f.

```cpp
inline virtual void apply(Field2D &f, BoutReal t)
```

```cpp
virtual void apply(Field3D &f) = 0
```

```cpp
inline virtual void apply(Field3D &f, BoutReal t)
```

```cpp
inline virtual void apply(Vector2D &f)
```

```cpp
inline virtual void apply(Vector3D &f)
```

class **BoundaryOp**: public `BoundaryOpBase`

```
#include <boundary_op.hxx> An operation on a boundary.
```

**Public Functions**

```cpp
inline BoundaryOp() // inline BoundaryOp(BoundaryRegion *region)

~BoundaryOp() override = default

inline virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args)

inline virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args, const std::map<std::string, std::string> &keywords)

Clone using positional args and keywords If not implemented, check if keywords are passed, then call two-argument version

inline virtual void apply_ddt(Field2D &f) // Apply a boundary condition on ddt(f)

inline virtual void apply_ddt(Field3D &f)

inline virtual void apply_ddt(Vector2D &f)

inline virtual void apply_ddt(Vector3D &f)
```

**Public Members**

```
BoundaryRegion *bndry

bool apply_to_ddt

class BoundaryModifier : public BoundaryOp

Subclassed by BoundaryFromFieldAligned, BoundaryRelax, BoundaryToFieldAligned, BoundaryWidth
```

**Public Functions**

```cpp
BoundaryModifier() = default

inline BoundaryModifier(BoundaryOp *operation)

virtual BoundaryOp *cloneMod(BoundaryOp *op, const std::list<std::string> &args) = 0
```
Protected Attributes

\[ \text{BoundaryOp} * \text{op} = \{ \text{nullptr} \} \]

K.2.11 File boundary_region.cxx

K.2.12 File boundary_region.hxx

Enums

```cpp
enum BndryLoc
{
    Location of boundary.
    Values:
    enumerator xin
    enumerator xout
    enumerator ydown
    enumerator yup
    enumerator all
    enumerator par_fwd
    enumerator par_bkwd
};
```

Variables

```cpp
constexpr BndryLoc BNDRY_XIN = BndryLoc::xin
constexpr BndryLoc BNDRY_XOUT = BndryLoc::xout
constexpr BndryLoc BNDRY_YDOWN = BndryLoc::ydown
constexpr BndryLoc BNDRY_YUP = BndryLoc::yup
constexpr BndryLoc BNDRY_ALL = BndryLoc::all
constexpr BndryLoc BNDRY_PAR_FWD = BndryLoc::par_fwd
constexpr BndryLoc BNDRY_PAR_BKWD = BndryLoc::par_bkwd
```

class BoundaryRegionBase

Subclassed by BoundaryRegion, BoundaryRegionPar
Public Functions

BoundaryRegionBase() = delete

inline BoundaryRegionBase(string name, Mesh* passmesh = nullptr)

inline BoundaryRegionBase(string name, BndryLoc loc, Mesh* passmesh = nullptr)

virtual ~BoundaryRegionBase() = default

virtual void first() = 0
    Move the region iterator to the start.

virtual void next() = 0
    Get the next element in the loop over every element from inside out (in X or Y first)

virtual bool isDone() = 0
    Returns true if outside domain. Can use this with nested nextX, nextY.

Public Members

Mesh *localmesh
    Mesh does this boundary region belongs to.

std::string label
    Label for this boundary region.

BndryLoc location
    Which side of the domain is it on?

bool isParallel = false
    Is this a parallel boundary?

class BoundaryRegion: public BoundaryRegionBase
    #include <boundary_region.hxx> Describes a region of the boundary, and a means of iterating over it.

Subclassed by BoundaryRegionXIn, BoundaryRegionXOut, BoundaryRegionYDown, BoundaryRegionYUp

Public Functions

BoundaryRegion() = delete

inline BoundaryRegion(string name, BndryLoc loc, Mesh* passmesh = nullptr)

inline BoundaryRegion(string name, int xd, int yd, Mesh* passmesh = nullptr)

~BoundaryRegion() override = default
virtual void next1d() = 0
   Loop over the innermost elements.

virtual void nextX() = 0
   Just loop over X.

virtual void nextY() = 0
   Just loop over Y.

**Public Members**

int x

int y
   Indices of the point in the boundary.

int bx

int by
   Direction of the boundary [x+dx][y+dy] is going outwards.

int width
   Width of the boundary.

class BoundaryRegionXIn : public BoundaryRegion

**Public Functions**

**BoundaryRegionXIn**(std::string name, int ymin, int ymax, Mesh *passmesh = nullptr)

virtual void first() override
   Move the region iterator to the start.

virtual void next() override
   Get the next element in the loop over every element from inside out (in X or Y first)

virtual void next1d() override
   Loop over the innermost elements.

virtual void nextX() override
   Just loop over X.

virtual void nextY() override
   Just loop over Y.

virtual bool isDone() override
   Returns true if outside domain. Can use this with nested nextX, nextY.
Private Members

int ys
int ye

class BoundaryRegionXOut : public BoundaryRegion

Public Functions

BoundaryRegionXOut (std::string name, int ymin, int ymax, Mesh *passmesh = nullptr)

virtual void first() override
    Move the region iterator to the start.

virtual void next() override
    Get the next element in the loop over every element from inside out (in X or Y first)

virtual void next1d() override
    Loop over the innermost elements.

virtual void nextX() override
    Just loop over X.

virtual void nextY() override
    Just loop over Y.

virtual bool isDone() override
    Returns true if outside domain. Can use this with nested nextX, nextY.

Private Members

int ys
int ye

class BoundaryRegionYDown : public BoundaryRegion

Public Functions

BoundaryRegionYDown (std::string name, int xmin, int xmax, Mesh *passmesh = nullptr)

virtual void first() override
    Move the region iterator to the start.

virtual void next() override
    Get the next element in the loop over every element from inside out (in X or Y first)

virtual void next1d() override
    Loop over the innermost elements.

virtual void nextX() override
    Just loop over X.

virtual void nextY() override
    Just loop over Y.
virtual bool isDone() override
    Returns true if outside domain. Can use this with nested nextX, nextY.

Private Members

int xs
int xe

class BoundaryRegionYUp : public BoundaryRegion

Public Functions

BoundaryRegionYUp(std::string name, int xmin, int xmax, Mesh *passmesh = nullptr)

virtual void first() override
    Move the region iterator to the start.

virtual void next() override
    Get the next element in the loop over every element from inside out (in X or Y first)

virtual void next1d() override
    Loop over the innermost elements.

virtual void nextX() override
    Just loop over X.

virtual void nextY() override
    Just loop over Y.

virtual bool isDone() override
    Returns true if outside domain. Can use this with nested nextX, nextY.

Private Members

int xs
int xe

namespace bout
    SNB model

namespace globals
Variables

*mesh
Global mesh.
The mesh object.

K.2.13 File boundary_standard.cxx

Functions

void verifyNumPoints(BoundaryRegion *region, int ptsRequired)
  Check that there are sufficient non-boundary points for desired B.C.
  Checks both the size of the global grid (i.e. if this B.C. could be ok for some parallel setup or not) and the local grid.
  Note the local grid check is not strictly necessary as this would typically lead to an out of bounds access error later but we add it here to provide a more explanatory message.

K.2.14 File boundary_standard.hxx

Functions

BoutReal default_func(BoutReal t, int x, int y, int z)

class BoundaryDirichlet_2ndOrder : public BoundaryOp
#include <boundary_standard.hxx> Some standard boundary conditions.
  Dirichlet boundary condition set half way between guard cell and grid cell at 2nd order accuracy

Public Functions

inline BoundaryDirichlet_2ndOrder()

inline BoundaryDirichlet_2ndOrder(BoutReal setval)

inline BoundaryDirichlet_2ndOrder(BoundaryRegion *region, BoutReal setval = 0.)

virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args) override

virtual void apply(Field2D &f) override
  Apply a boundary condition on field f.
virtual void apply(Field3D &f) override

virtual void apply_ddt(Field2D &f) override
  Apply a boundary condition on ddt(f)
virtual void apply_ddt(Field3D &f) override

Private Members

BoutReal val

class BoundaryDirichlet : public BoundaryOp
#include <boundary_standard.hxx> Dirichlet (set to zero) boundary condition.

Public Functions

inline BoundaryDirichlet()

inline BoundaryDirichlet(BoundaryRegion *region, std::shared_ptr<FieldGenerator> g)

virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args) override

virtual void apply(Field2D &f) override
    Apply a boundary condition on field f.
virtual void apply(Field2D &f, BoutReal t) override

virtual void apply(Field3D &f) override

virtual void apply(Field3D &f, BoutReal t) override

virtual void apply_ddt(Field2D &f) override
    Apply a boundary condition on ddt(f)
virtual void apply_ddt(Field3D &f) override

Private Members

std::shared_ptr<FieldGenerator> gen

class BoundaryDirichlet_O3 : public BoundaryOp
#include <boundary_standard.hxx> 3rd-order boundary condition
Public Functions

inline BoundaryDirichlet_O3()

inline BoundaryDirichlet_O3(BoundaryRegion *region, std::shared_ptr<FieldGenerator> g)

virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args) override

virtual void apply(Field2D &f) override
  Apply a boundary condition on field f.

virtual void apply(Field2D &f, BoutReal t) override

virtual void apply(Field3D &f) override

virtual void apply(Field3D &f, BoutReal t) override

virtual void apply_ddt(Field2D &f) override
  Apply a boundary condition on ddt(f)

virtual void apply_ddt(Field3D &f) override

Private Members

std::shared_ptr<FieldGenerator> gen

class BoundaryDirichlet_O4 : public BoundaryOp
#include <boundary_standard.hxx> 4th-order boundary condition

Public Functions

inline BoundaryDirichlet_O4()

inline BoundaryDirichlet_O4(BoundaryRegion *region, std::shared_ptr<FieldGenerator> g)

virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args) override

virtual void apply(Field2D &f) override
  Apply a boundary condition on field f.

virtual void apply(Field2D &f, BoutReal t) override

virtual void apply(Field3D &f) override

virtual void apply(Field3D &f, BoutReal t) override
virtual void apply_ddt(Field2D &f) override
    Apply a boundary condition on ddt(f)

virtual void apply_ddt(Field3D &f) override

Private Members

std::shared_ptr<FieldGenerator> gen

class BoundaryDirichlet_4thOrder : public BoundaryOp
    #include <boundary_standard.hxx> Dirichlet boundary condition set half way between guard cell and grid cell at 4th order accuracy.

Public Functions

inline BoundaryDirichlet_4thOrder()

inline BoundaryDirichlet_4thOrder(BoutReal setval)

inline BoundaryDirichlet_4thOrder(BoundaryRegion *region, BoutReal setval = 0.)

virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args) override

virtual void apply(Field2D &f) override
    Apply a boundary condition on field f.

virtual void apply(Field3D &f) override

virtual void apply_ddt(Field2D &f) override
    Apply a boundary condition on ddt(f)

virtual void apply_ddt(Field3D &f) override

Private Members

BoutReal val

class BoundaryNeumann_NonOrthogonal : public BoundaryOp
    #include <boundary_standard.hxx> Neumann (zero-gradient) boundary condition for non-orthogonal meshes.
**Public Functions**

inline `BoundaryNeumann_NonOrthogonal()`

inline `BoundaryNeumann_NonOrthogonal(BoutReal setval)`

inline `BoundaryNeumann_NonOrthogonal(BoundaryRegion *region, BoutReal setval = 0.)`

virtual `BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args)` override

virtual void `apply(Field2D &f)` override  
Apply a boundary condition on field f.

virtual void `apply(Field3D &f)` override

**Private Members**

`BoutReal val`

class `BoundaryNeumann2` : public `BoundaryOp`

#include <boundary_standard.hxx> Neumann (zero-gradient) boundary condition, using 2nd order on boundary.

**Public Functions**

inline `BoundaryNeumann2()`

inline `BoundaryNeumann2(BoundaryRegion *region)`

virtual `BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args)` override

virtual void `apply(Field2D &f)` override  
Apply a boundary condition on field f.

virtual void `apply(Field3D &f)` override

class `BoundaryNeumann_2ndOrder` : public `BoundaryOp`

#include <boundary_standard.hxx> Neumann boundary condition set half way between guard cell and grid cell at 2nd order accuracy.
Public Functions

inline BoundaryNeumann_2ndOrder()

inline BoundaryNeumann_2ndOrder(BoutReal setval)

inline BoundaryNeumann_2ndOrder(BoundaryRegion *region, BoutReal setval = 0.)

virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args) override

virtual void apply(Field2D &f) override
    Apply a boundary condition on field f.

virtual void apply(Field3D &f) override

virtual void apply_ddt(Field2D &f) override
    Apply a boundary condition on ddt(f)

virtual void apply_ddt(Field3D &f) override

Private Members

BoutReal val

class BoundaryNeumann : public BoundaryOp

Public Functions

inline BoundaryNeumann()

inline BoundaryNeumann(BoundaryRegion *region, std::shared_ptr<FieldGenerator> g)

virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args) override

virtual void apply(Field2D &f) override
    Apply a boundary condition on field f.

virtual void apply(Field2D &f, BoutReal t) override

virtual void apply(Field3D &f) override

virtual void apply(Field3D &f, BoutReal t) override

virtual void apply_ddt(Field2D &f) override
    Apply a boundary condition on ddt(f)

virtual void apply_ddt(Field3D &f) override
Private Members

`std::shared_ptr<FieldGenerator> gen`

class `BoundaryNeumann_4thOrder` : public `BoundaryOp`
#include `<boundary_standard.hxx>` Neumann boundary condition set half way between guard cell and grid cell at 4th order accuracy.

Public Functions

inline `BoundaryNeumann_4thOrder()`

inline `BoundaryNeumann_4thOrder(BoutReal setval)`

inline `BoundaryNeumann_4thOrder(BoundaryRegion *region, BoutReal setval = 0.)`

virtual `BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args)` override

virtual void `apply(Field2D &f)` override
   Apply a boundary condition on field f.

virtual void `apply(Field3D &f)` override

virtual void `apply_ddt(Field2D &f)` override
   Apply a boundary condition on ddt(f)

virtual void `apply_ddt(Field3D &f)` override

Private Members

`BoutReal val`

class `BoundaryNeumann_04` : public `BoundaryOp`
#include `<boundary_standard.hxx>` Neumann boundary condition set half way between guard cell and grid cell at 4th order accuracy.

Public Functions

inline `BoundaryNeumann_04()`

inline `BoundaryNeumann_04(BoundaryRegion *region, std::shared_ptr<FieldGenerator> g)`

virtual `BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args)` override

virtual void `apply(Field2D &f)` override
   Apply a boundary condition on field f.
virtual void apply(Field2D &f, BoutReal t) override

virtual void apply(Field3D &f) override

virtual void apply(Field3D &f, BoutReal t) override

virtual void apply_ddt(Field2D &f) override
    Apply a boundary condition on ddt(f)
virtual void apply_ddt(Field3D &f) override

**Private Members**

std::shared_ptr<FieldGenerator> gen

class BoundaryNeumannPar : public BoundaryOp
    #include <boundary_standard.hxx> NeumannPar (zero-gradient) boundary condition on the variable / sqrt(g_22)

**Public Functions**

inline BoundaryNeumannPar()

inline BoundaryNeumannPar(BoundaryRegion *region)

virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args) override

virtual void apply(Field2D &f) override
    Apply a boundary condition on field f.
virtual void apply(Field3D &f) override

class BoundaryRobin : public BoundaryOp
    #include <boundary_standard.hxx> Robin (mix of Dirichlet and Neumann)

**Public Functions**

inline BoundaryRobin()

inline BoundaryRobin(BoundaryRegion *region, BoutReal a, BoutReal b, BoutReal g)

virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args) override

virtual void apply(Field2D &f) override
    Apply a boundary condition on field f.
virtual void apply(Field3D &f) override

Private Members

BoutReal aval
BoutReal bval
BoutReal gval

class BoundaryConstGradient : public BoundaryOp
#include <boundary_standard.hxx> Constant gradient (zero second derivative)

Public Functions

inline BoundaryConstGradient()

inline BoundaryConstGradient(BoundaryRegion *region)

virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args) override

virtual void apply(Field2D &f) override
Apply a boundary condition on field f.

virtual void apply(Field3D &f) override

class BoundaryZeroLaplace : public BoundaryOp
#include <boundary_standard.hxx> Zero Laplacian, decaying solution.

Public Functions

inline BoundaryZeroLaplace()

inline BoundaryZeroLaplace(BoundaryRegion *region)

virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args) override

virtual void apply(Field2D &f) override
Apply a boundary condition on field f.

virtual void apply(Field3D &f) override

class BoundaryZeroLaplace2 : public BoundaryOp
#include <boundary_standard.hxx> Zero Laplacian.
**Public Functions**

```cpp
inline BoundaryZeroLaplace2()

inline BoundaryZeroLaplace2(BoundaryRegion *region)

virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args) override

virtual void apply(Field2D &f) override
    Apply a boundary condition on field f.

virtual void apply(Field3D &f) override
```

**class BoundaryConstLaplace : public BoundaryOp**
```
#include <boundary_standard.hxx> Constant Laplacian, decaying solution.
```

**Public Functions**

```cpp
inline BoundaryConstLaplace()

inline BoundaryConstLaplace(BoundaryRegion *region)

virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args) override

virtual void apply(Field2D &f) override
    Apply a boundary condition on field f.

virtual void apply(Field3D &f) override
```

**class BoundaryDivCurl : public BoundaryOp**
```
#include <boundary_standard.hxx> Vector boundary condition Div(B) = 0, Curl(B) = 0.
```

**Public Functions**

```cpp
inline BoundaryDivCurl()

inline BoundaryDivCurl(BoundaryRegion *region)

virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args) override

inline virtual void apply(Field2D &f) override
    Apply a boundary condition on field f.

inline virtual void apply(Field3D &f) override

virtual void apply(Vector2D &f) override
```
virtual void apply(Vector3D &f) override

class BoundaryFree : public BoundaryOp
#include <boundary_standard.hxx> Free boundary condition (evolve the field in the guard cells, using non-centred derivatives to calculate the ddt)

Public Functions

inline BoundaryFree()

inline BoundaryFree(BoutReal setval)

inline BoundaryFree(BoundaryRegion *region, BoutReal setval = 0.)

virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args) override

virtual void apply(Field2D &f) override
Apply a boundary condition on field f.

virtual void apply(Field3D &f) override

virtual void apply_ddt(Field2D &f) override
Apply a boundary condition on ddt(f)

virtual void apply_ddt(Field3D &f) override

Private Members

BoutReal val

class BoundaryFree_02 : public BoundaryOp
#include <boundary_standard.hxx> Alternative free boundary condition (evolve the field in the guard cells, using non-centred derivatives to calculate the ddt)

Public Functions

inline BoundaryFree_02()

inline BoundaryFree_02(BoundaryRegion *region)

virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args) override

virtual void apply(Field2D &f) override
Apply a boundary condition on field f.

virtual void apply(Field3D &f) override
virtual void apply_ddt(Field2D &f) override  
Apply a boundary condition on ddt(f)

virtual void apply_ddt(Field3D &f) override

class BoundaryFree_O3 : public BoundaryOp

Public Functions

inline BoundaryFree_O3()

inline BoundaryFree_O3(BoundaryRegion *region)

virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args) override

virtual void apply(Field2D &f) override  
Apply a boundary condition on field f.

virtual void apply(Field3D &f) override

virtual void apply_ddt(Field2D &f) override  
Apply a boundary condition on ddt(f)

virtual void apply_ddt(Field3D &f) override

class BoundaryRelax : public BoundaryModifier

#include <boundary_standard.hxx>  Convert a boundary condition to a relaxing one.

Public Functions

inline BoundaryRelax()

inline BoundaryRelax(BoundaryOp *operation, BoutReal rate)

virtual BoundaryOp *cloneMod(BoundaryOp *op, const std::list<std::string> &args) override

inline virtual void apply(Field2D &f) override  
Apply a boundary condition on field f.

virtual void apply(Field2D &f, BoutReal t) override

inline virtual void apply(Field3D &f) override

virtual void apply(Field3D &f, BoutReal t) override

virtual void apply_ddt(Field2D &f) override  
Apply a boundary condition on ddt(f)
virtual void apply_ddt(Field3D &f) override

**Private Members**

*BoutReal* `r`

class **BoundaryWidth** : public **BoundaryModifier**

```
#include <boundary_standard.hxx>
```

Increase the width of a boundary.

**Public Functions**

inline **BoundaryWidth**()

inline **BoundaryWidth**(BoundaryOp *operation, int wid)

virtual BoundaryOp *cloneMod(BoundaryOp *op, const std::list<std::string> &args) override

inline virtual void apply(Field2D &f) override

Apply a boundary condition on field `f`.

virtual void apply(Field2D &f, BoutReal t) override

inline virtual void apply(Field3D &f) override

virtual void apply(Field3D &f, BoutReal t) override

virtual void apply_ddt(Field2D &f) override

Apply a boundary condition on ddt(f)

virtual void apply_ddt(Field3D &f) override

**Private Members**

int `width`

class **BoundaryToFieldAligned** : public **BoundaryModifier**

```
#include <boundary_standard.hxx>
```

Convert input field from `FieldAligned`, apply boundary and then convert back to `FieldAligned`. Equivalent to converting the boundary condition to “Field Aligned” from “orthogonal”
**Public Functions**

inline `BoundaryToFieldAligned()`

inline `BoundaryToFieldAligned(BoundaryOp *operation)`

virtual `BoundaryOp *cloneMod(BoundaryOp *op, const std::list<std::string> &args)` override

inline virtual void `apply(Field2D &f)` override  
Apply a boundary condition on field f.

virtual void `apply(Field2D &f, BoutReal t)` override

inline virtual void `apply(Field3D &f)` override

virtual void `apply(Field3D &f, BoutReal t)` override

virtual void `apply(ddt)(Field2D &f)` override  
Apply a boundary condition on ddt(f)

virtual void `apply(ddt)(Field3D &f)` override

---

class `BoundaryFromFieldAligned` : public `BoundaryModifier`  
`#include <boundary_standard.hxx>` Convert input field toFieldAligned, apply boundary and then convert back fromFieldAligned Equivalent to converting the boundary condition from “Field Aligned” to “orthogonal”

**Public Functions**

inline `BoundaryFromFieldAligned()`

inline `BoundaryFromFieldAligned(BoundaryOp *operation)`

virtual `BoundaryOp *cloneMod(BoundaryOp *op, const std::list<std::string> &args)` override

inline virtual void `apply(Field2D &f)` override  
Apply a boundary condition on field f.

virtual void `apply(Field2D &f, BoutReal t)` override

inline virtual void `apply(Field3D &f)` override

virtual void `apply(Field3D &f, BoutReal t)` override

virtual void `apply_ddt(Field2D &f)` override  
Apply a boundary condition on ddt(f)

virtual void `apply_ddt(Field3D &f)` override
K.2.15 File bout++-time.cxx

Variables

const char *boutcompiledate = __DATE__
const char *boutcompiletime = __TIME__

K.2.16 File bout++-time.hxx

Variables

const char *boutcompiledate
const char *boutcompiletime

K.2.17 File bout++.cxx

Defines

BUILDFLAG1_(x)

BUILDFLAG(x)

GLOBALORIGIN

INDIRECT1_BOUTMAIN(a)

INDIRECT0_BOUTMAIN(...)

STRINGIFY(a)

BOUT_NO_USING_NAMESPACE_BOUTGLOBALS

Functions

void bout_signal_handler(int sig)
    Signal handler - handles all signals.

std::string time_to_hms(BoutReal t)
    Write a time in h:mm:ss.s format.

char get_spin()
    Produce a spinning bar character.

int BoutInitialise(int &argc, char **&argv)
    Initialise BOUT++

Inputs
The command-line arguments argc and argv are passed by reference, and pointers to these will be stored in various places in BOUT++.

**Outputs**

Any non-zero return value should halt the simulation. If the return value is less than zero, the exit status from BOUT++ is 0, otherwise it is the return value of BoutInitialise.

```c
int bout_run(Solver *solver, rhsfunc physics_run)
```

Run the given solver. This function is only used for old-style physics models with standalone C functions. The main() function in boutmain.hxx calls this function to set up the RHS function and add bout_monitor.

```c
int BoutFinalise(bool write_settings)
```

BOUT++ finalisation. This should be called at the end of the program.

Frees memory, flushes buffers, and closes files. If BOUT++ initialised MPI or external libraries, then these are also finalised.

If `write_settings` is true, output the settings, showing which options were used. This overwrites the file written during initialisation (BOUT.settings by default)

```c
Variables
```

```c
const char DEFAULT_DIR[] = "data"
BoutReal simtime = {0.0}
int iteration = {0}
bool user_requested_exit = false
```

```c
namespace bout
```

```c
SNB model
```

```c
namespace experimental
```

### K.2.18 File bout.hxx

#### Functions

```c
int BoutInitialise(int &argc, char **&argv)
```

BOUT++ initialisation. This function must be called first, passing command-line arguments.

This will call MPI_Initialize, and if BOUT++ has been configured with external libraries such as PETSc then these will be initialised as well.

**Example**

A minimal BOUT++ program consists of:

```c
int main(int argc, char** argv) {
    BoutInitialise(argc, argv);

    BoutFinalise();
}
```
Usually this function is called in a standard main() function, either by including boutmain.hxx or by including bout/physicsmodel.hxx and using the BOUTMAIN macro.

Initialise BOUT++

**Inputs**

The command-line arguments argc and argv are passed by reference, and pointers to these will be stored in various places in BOUT++.

**Outputs**

Any non-zero return value should halt the simulation. If the return value is less than zero, the exit status from BOUT++ is 0, otherwise it is the return value of BoutInitialise.

```cpp
int bout_run(Solver *solver, rhsfunc physics_run)
```

Run the given solver. This function is only used for old-style physics models with standalone C functions. The main() function in boutmain.hxx calls this function to set up the RHS function and add bout_monitor.

```cpp
int BoutFinalise(bool write_settings = true)
```

BOUT++ finalisation. This should be called at the end of the program.

Frees memory, flushes buffers, and closes files. If BOUT++ initialised MPI or external libraries, then these are also finalised.

If `write_settings` is true, output the settings, showing which options were used. This overwrites the file written during initialisation (BOUT.settings by default)

**Variables**

```cpp
const BoutReal BOUT_VERSION = BOUT_VERSION_DOUBLE
```

Version number.

```
#include <bout.hxx>
```

Monitor class for output. Called by the solver every output timestep.

This is added to the solver in bout_run (for C-style models) or in bout/physicsmodel.hxx

**Public Functions**

```cpp
inline BoutMonitor(BoutReal timestep = -1)
```

**Private Functions**

```cpp
virtual int call(Solver *solver, BoutReal t, int iter, int NOUT) override
```

SOLUTION MONITOR FUNCTION

Called each timestep by the solver
Private Members

*RunMetrics* run_data

namespace bout

SNB model

namespace experimental

Typedefs

using SignalHandler = void (*)(int)

Function type for handling signals.

Functions

void *setupSignalHandler*(*SignalHandler* signal_handler)*

Set a signal handler for user-requested clean exit, and (optionally) segmentation faults and floating point errors

- For segmentation faults, compile with `--enable-signal`.
- For floating point errors, compile with `--enable-sigfpe`

void *defaultSignalHandler*(*int* sig)*

The default BOUT++ signal handler: throw an exception with an appropriate message

void *setupGetText*()

Set up the i18n environment.

*CommandLineArgs* parseCommandLineArgs(*int* argc, *char**argv)*

Parse the “fixed” command line arguments, like help and -d.

void *checkDataDirectoryIsAccessible*(*const std::string &data_dir)*

Throw an exception if *data_dir* is either not a directory or not accessible. We do not check whether we can write, as it is sufficient that the files we need are writeable

void *setupOutput*(*const std::string &data_dir,* const std::string &log_file, *int* verbosity, *int* MYPE = 0)*

Set up the output: open the log file for each processor, enable or disable the default outputs based on verbosity, disable writing to stdout for MYPE != 0

void *savePIDtoFile*(*const std::string &data_dir,* int MYPE)*

Save the process ID for processor N = MYPE to file “.BOUT.pid.N” in data_dir, so it can be shut down by user signal

Throws if it was not possible to create the file

void *printStartupHeader*(*int* MYPE, *int* NPES)*

Print the initial header.

void *printCompileTimeOptions*()

Print the compile-time options.

void *printCommandLineArguments*(*const std::vector<std::string> &original_argv)*

Print the arguments given on the command line.
bool setupBoutLogColor(bool color_output, int MYPE)
    Setup the pipe etc and run stdout through bout-log-color. Return true if it was successful

void setRunStartInfo(Options &options)
    Set BOUT++ version information, along with current time (as started), into run section of options

void setRunFinishInfo(Options &options)
    Set the current time (as finished) into run section of options

void writeSettingsFile(Options &options, const std::string &data_dir, const std::string &settings_file)
    Write options to settings_file in directory data_dir.

Datafile setupDumpFile(Options &options, Mesh &mesh, const std::string &data_dir)
    Setup the output dump files from options using the mesh. Files are created in the data_dir directory

struct CommandLineArgs
    #include <bout.hxx> Results of parsing the command line arguments.

    Public Members

    int verbosity = {4}
    bool color_output = {false}
    std::string data_dir = {"data"}
        Directory for data input/output.

    std::string opt_file = {"BOUT.inp"}
        Filename for the options file.

    std::string set_file = {"BOUT.settings"}
        Filename for the options file.

    std::string log_file = {"BOUT.log"}
        File name for the log file

    std::vector<std::string> original_argv
        The original set of command line arguments.

K.2.19 File bout_enum_class.hxx

Defines

_ec-expand_1(_call, enumname, x)
    Create some macro magic similar to bout/macro_for_each.hxx, but allowing for the enum class name to be passed through to each _call _ec-expand_x set of macros expand a number of arguments without ‘;’ between them

_ec-expand_2(_call, enumname, x, ...)

_ec-expand_3(_call, enumname, x, ...)
_ec_expand_4 (_call, enumname, x, ...)

_ec_expand_5 (_call, enumname, x, ...)

_ec_expand_6 (_call, enumname, x, ...)

_ec_expand_7 (_call, enumname, x, ...)

_ec_expand_8 (_call, enumname, x, ...)

_ec_expand_9 (_call, enumname, x, ...)

_ec_expand_10 (_call, enumname, x, ...)

BOUT_ENUM_CLASS_MAP_ARGS (mac, enumname, ...)

BOUT_ENUM_CLASS_STR (enumname, val)

BOUT_STR_ENUM_CLASS (enumname, val)

BOUT_MAKE_FROMSTRING_NAME (enumname)

BOUT_ENUM_CLASS (enumname, ...)
    Create an enum class with toString and <enum name>="">FromString functions, and an Options::as<enum> overload to read the enum

K.2.20 File bout_types.cxx

Functions

std::string toString (CELL_LOC location)

CELL_LOC CELL_LOCFromString (const std::string &location_string)

std::string toString (DIFF_METHOD location)

std::string toString (REGION region)

std::string toString (DIRECTION direction)

void swap (DirectionTypes &first, DirectionTypes &second)
bool areDirectionsCompatible(const DirectionTypes &d1, const DirectionTypes &d2)
    Check whether direction types are compatible, so two fields with attributes d1 and d2 respectively can be added, subtracted, etc.

std::string toString(STAGGER stagger)

std::string toString(DERIV deriv)

std::string toString(YDirectionType d)

YDirectionType YDirectionTypeFromString(const std::string &y_direction_string)

std::string toString(ZDirectionType d)

ZDirectionType ZDirectionTypeFromString(const std::string &z_direction_string)

K.2.21 File bout_types.hxx

Defines

ENUMSTR(val)

STRENUM(val)

Typedefs

using BoutReal = double
    Size of real numbers.

using FuncPtr = BoutReal (*)(BoutReal t, BoutReal x, BoutReal y, BoutReal z)
    Boundary condition function.

Enums

enum CELL_LOC
    4 possible variable locations. Default is for passing to functions
    Values:

    enumerator deflt
    enumerator centre
    enumerator xlow
    enumerator ylow
    enumerator zlow
enumerator \texttt{vshift}

enum \texttt{DIFF\_METHOD}
Differential methods. Both central and upwind.

\textit{Values:}

enumerator \texttt{deflt}
enumerator \texttt{u1}
enumerator \texttt{u2}
enumerator \texttt{c2}
enumerator \texttt{w2}
enumerator \texttt{w3}
enumerator \texttt{c4}
enumerator \texttt{u3}
enumerator \texttt{fft}
enumerator \texttt{split}
enumerator \texttt{s2}

enum \texttt{REGION}
Specify grid region for looping.

\textit{Values:}

enumerator \texttt{all}
enumerator \texttt{nobndry}
enumerator \texttt{nox}
enumerator \texttt{noy}
enumerator \texttt{noz}

enum \texttt{DIRECTION}
To identify particular directions (in index space):

\begin{itemize}
  \item X, Y, Z are the coordinate directions
  \item YAligned is a special case of Y, indicating a field-aligned grid, where the x- and z- axes are not necessarily orthogonal
  \item YOrthogonal is a special case of Y, indicating a grid where the x and z axes are orthogonal but the y-direction is not necessarily field-aligned
\end{itemize}

\textit{Values:}

enumerator \texttt{X}
enumerator \texttt{Y}
enumerator \texttt{Z}
enumerator \texttt{YAligned}
enumerator \texttt{YOrthogonal}
enum YDirectionType
   Identify kind of a field’s y-direction
   • Standard is the default for the Mesh/Coordinates/ParallelTransform
   • Aligned indicates that the field has been transformed to field-aligned coordinates

Values:

enumerator Standard
enumerator Aligned

enum ZDirectionType
   Identify kind of a field’s z-direction
   • Standard is the default
   • Average indicates that the field represents an average over the z-direction, rather than having a particular z-position (i.e. is a Field2D)

Values:

enumerator Standard
enumerator Average

enum STAGGER
   To identify valid staggering combinations.

Values:

enumerator None
enumerator C2L
enumerator L2C

enum DERIV
   To identify types of derivative method combinations.

Values:

enumerator Standard
enumerator StandardSecond
enumerator StandardFourth
enumerator Upwind
enumerator Flux
Functions

```cpp
std::string toString(CELL_LOC location)

CELL_LOC CELL_LOCFromString(const std::string &location_string)

inline std::string CELL_LOC_STRING(CELL_LOC location)

std::string toString(DIFF_METHOD location)

inline std::string DIFF_METHOD_STRING(DIFF_METHOD location)

std::string toString(REGION region)

inline std::string REGION_STRING(REGION region)

std::string toString(DIRECTION direction)

inline std::string DIRECTION_STRING(DIRECTION direction)

std::string toString(YDirectionType d)

YDirectionType YDirectionTypeFromString(const std::string &y_direction_string)

std::string toString(ZDirectionType d)

ZDirectionType ZDirectionTypeFromString(const std::string &z_direction_string)

bool areDirectionsCompatible(const DirectionTypes &d1, const DirectionTypes &d2)
    // Check whether direction types are compatible, so two fields with attributes d1 and d2 respectively can be added, subtracted, etc.

void swap(const DirectionTypes &first, const DirectionTypes &second)

std::string toString(STAGGER stagger)

inline std::string STAGGER_STRING(STAGGER stagger)

std::string toString(DERIV deriv)

inline std::string DERIV_STRING(DERIV deriv)
```
Variables

constexpr BoutReal BoutNaN = std::numeric_limits<BoutReal>::quiet_NaN()
    Quiet NaN.
constexpr CELL_LOC CELL_DEFAULT = CELL_LOC::deflt
constexpr CELL_LOC CELL_CENTRE = CELL_LOC::centre
constexpr CELL_LOC CELL_CENTER = CELL_LOC::centre
constexpr CELL_LOC CELL_XLOW = CELL_LOC::zlow
constexpr CELL_LOC CELL_YLOW = CELL_LOC::ylow
constexpr CELL_LOC CELL_ZLOW = CELL_LOC::zlow
constexpr CELL_LOC CELL_VSHIFT = CELL_LOC::vshift
constexpr DIFF_METHOD DIFF_DEFAULT = DIFF_METHOD::deflt
constexpr DIFF_METHOD DIFF_U1 = DIFF_METHOD::u1
constexpr DIFF_METHOD DIFF_U2 = DIFF_METHOD::u2
constexpr DIFF_METHOD DIFF_C2 = DIFF_METHOD::c2
constexpr DIFF_METHOD DIFF_W2 = DIFF_METHOD::w2
constexpr DIFF_METHOD DIFF_W3 = DIFF_METHOD::w3
constexpr DIFF_METHOD DIFF_C4 = DIFF_METHOD::c4
constexpr DIFF_METHOD DIFF_U3 = DIFF_METHOD::u3
constexpr DIFF_METHOD DIFF_FFT = DIFF_METHOD::fft
constexpr DIFF_METHOD DIFF_SPLIT = DIFF_METHOD::split
constexpr DIFF_METHOD DIFF_S2 = DIFF_METHOD::s2
constexpr REGION RGN_ALL = REGION::all
constexpr REGION RGN_NOBNDRY = REGION::nobndry
constexpr REGION RGN_NOX = REGION::nox
constexpr REGION RGN_NOY = REGION::noy
constexpr REGION RGN_NOZ = REGION::noz
struct DirectionTypes
    #include <bout_types.hxx> Container for direction types.
Public Members

YDirectionType y
ZDirectionType z

template<typename T, T val>
struct enumWrapper

Public Types

using type = T

Public Functions

inline T lookup()

Public Static Attributes

static const type value = val

K.2.22 File boutcomm.cxx

K.2.23 File boutcomm.hxx

class BoutComm
#include <boutcomm.hxx> Class to represent the ‘global’ communicator.

Public Functions

~BoutComm()

void setComm(MPI_Comm c)

MPI_Comm getComm()

bool isSet()
**Public Static Functions**

static BoutComm *getInstance()  
Get a pointer to the only instance.

static MPI_Comm get()  
Shortcut method.

static void setArgs(int &c, char **&v)

static void cleanup()

static int rank()  
Rank: my processor number.

static int size()  
Size: number of processors.

**Private Functions**

BoutComm()

**Private Members**

int *pargc = nullptr

char ***pargv = nullptr
Command-line arguments. These can be modified by MPI init, so pointers are used.

bool hasBeenSet = false

MPI_Comm comm

**Private Static Attributes**

static BoutComm *instance = nullptr  
The only instance of this class (Singleton)

**K.2.24 File boutexception.cxx**

**Defines**

INIT_EXCEPTION(s)  
Common set up for exceptions
Formats the message s using C-style printf formatting
Functions

void BoutParallelThrowRhsFail(int status, const char *message)

Throw BoutRhsFail with message if any one process has non-zero status

K.2.25 File boutexception.hxx

Functions

void BoutParallelThrowRhsFail(int status, const char *message)

Throw BoutRhsFail with message if any one process has non-zero status

class BoutException : public std::exception

Subclassed by BoutIterationFail, BoutRhsFail

Public Functions

BoutException(const char*, ...)

inline BoutException(std::string msg)

~BoutException() override

inline const char *what() const noexcept override

inline void Backtrace()

std::string getBacktrace() const

Return the exception message along with the MsgStack and backtrace (if available)

Public Members

const std::string header = {"====== Exception thrown ======
"

Protected Functions

void makeBacktrace()
Protected Attributes

char *buffer = nullptr
int buflen
std::string message
void *trace[TRACE_MAX]
int trace_size
char **messages
std::string backtrace_message = {}
Functions

int physics_init(bool restarting)
   Initialise the model. Called once at the start
   Parameters  restarting – [in] True if the simulation is restarting
   Returns  zero on success, non-zero error code

int physics_run(BoutReal t)
   Calculate the time derivative
   Parameters  t – [in] Simulation time
   Returns  zero on success, non-zero error code

template<class T>
void bout_solve(T &var, const char *name, const std::string &description = "")
   Global functions used by some legacy models.

bool bout_constraint(Field3D &var, Field3D &F_var, const char *name)
   Add a contraint variable

int main(int argc, char **argv)
   Main function.

Variables

Solver *solver
   Need a global Solver pointer, which is the same as the PhysicsModel solver

class LegacyModel : public PhysicsModel
   #include <boutmain.hxx> Class interface to Solvers, which emulates the older standalone function interface

Protected Functions

inline virtual int init(bool restarting) override
   Initialise.

inline virtual int rhs(BoutReal t) override
   Calculate time derivatives.

K.2.27 File boutmesh.cxx

Defines

PVEC_REAL_MPI_TYPE
   MPI type of BoutReal for communications.
Variables

const int IN_SENT_UP = 0
   Data lower in X than branch-cut, at upper boundary in Y.

const int OUT_SENT_UP = 1
   Data higher in X than branch-cut, at upper boundary in Y.

const int IN_SENT_DOWN = 2
   Data lower in X than branch-cut, at lower boundary in Y.

const int OUT_SENT_DOWN = 3
   Data higher in X than branch-cut, at lower boundary in Y.

const int IN_SENT_OUT = 4
   Data going in positive X direction (in to out)

const int OUT_SENT_IN = 5
   Data going in negative X direction (out to in)

K.2.28 File boutmesh.hxx

class BoutMesh : public Mesh

#include <boutmesh.hxx> Implementation of Mesh (mostly) compatible with BOUT

Topology and communications compatible with BOUT conventions.

Public Functions

BoutMesh(GridDataSource *s, Options *options = nullptr)

~BoutMesh() override

virtual int load() override
   Read in the mesh from data sources.

virtual comm_handle send(FieldGroup &g) override
   Send data between processors Does not wait for communications to complete, so wait() must be called
   before guard cell values are used

   Example

   comm_handle handle = mesh->send(group); ... mesh->wait(handle);

   Parameters g – [in] A group of fields to communicate

virtual int wait(comm_handle handle) override
   Wait for a send operation to complete

   Parameters handle – [in] The handle returned by send()
virtual MPI_Request sendToProc(int xproc, int yproc, BoutReal *buffer, int size, int tag) override

Low-level communication routine Send a buffer of data from this processor to another This must be matched by a corresponding call to receiveFromProc on the receiving processor

**Parameters**

- **xproc** – [in] X index of processor to send to
- **yproc** – [in] Y index of processor to send to
- **buffer** – [in] A buffer of data to send
- **size** – [in] The length of **buffer**
- **tag** – [in] A label, must be the same at receive

virtual comm_handle receiveFromProc(int xproc, int yproc, BoutReal *buffer, int size, int tag) override

Low-level communication routine Receive a buffer of data from another processor Must be matched by corresponding sendToProc call on the sending processor

**Parameters**

- **xproc** – [in] X index of sending processor
- **yproc** – [in] Y index of sending processor
- **buffer** – [inout] The buffer to fill with data. Must already be allocated of length **size**
- **size** – [in] The length of **buffer**
- **tag** – [in] A label, must be the same as send

virtual int getNXPE() override

The number of processors in the X direction.

virtual int getNYPE() override

The number of processors in the Y direction.

virtual int getXProcIndex() override

This processor’s index in X direction.

virtual int getYProcIndex() override

This processor’s index in Y direction.

virtual bool firstX() override

Is this processor the first in X? i.e. is there a boundary to the left in X?

virtual bool lastX() override

Is this processor last in X? i.e. is there a boundary to the right in X?

virtual int sendXOut(BoutReal *buffer, int size, int tag) override

Send a buffer of data to processor at X index +1

**Parameters**

- **buffer** – [in] The data to send. Must be at least length **size**
- **size** – [in] The number of BoutReals to send
- **tag** – [in] A label for the communication. Must be the same at receive

virtual int sendXIn(BoutReal *buffer, int size, int tag) override

Send a buffer of data to processor at X index -1

**Parameters**

- **buffer** – [in] The data to send. Must be at least length **size**
• size – [in] The number of BoutReals to send
• tag – [in] A label for the communication. Must be the same at receive

virtual comm_handle irecvXOut(BoutReal *buffer, int size, int tag) override
Receive a buffer of data from X index +1

Parameters
• buffer – [in] A buffer to put the data in. Must already be allocated of length size
• size – [in] The number of BoutReals to receive and put in buffer
• tag – [in] A label for the communication. Must be the same as sent

virtual comm_handle irecvXIn(BoutReal *buffer, int size, int tag) override
Receive a buffer of data from X index -1

Parameters
• buffer – [in] A buffer to put the data in. Must already be allocated of length size
• size – [in] The number of BoutReals to receive and put in buffer
• tag – [in] A label for the communication. Must be the same as sent

inline virtual MPI_Comm getXcomm(int jy) const override
Return communicator containing all processors in X.

virtual MPI_Comm getYcomm(int jx) const override
Return communicator containing all processors in Y.

virtual bool periodicY(int jx, BoutReal &ts) const override
Is local X index jx periodic in Y?

Parameters
• jx – [in] The local (on this processor) index in X
• ts – [out] The Twist-Shift angle if periodic

virtual bool periodicY(int jx) const override
Is local X index jx periodic in Y?

Parameters jx – [in] The local (on this processor) index in X

virtual std::pair<bool, BoutReal> hasBranchCutLower(int jx) const override
Is there a branch cut at this processor’s lower boundary?

Parameters jx – [in] The local (on this processor) index in X

Returns pair<bool, BoutReal> - bool is true if there is a branch cut, BoutReal gives the total
zShift for a 2pi poloidal circuit if there is a branch cut

virtual std::pair<bool, BoutReal> hasBranchCutUpper(int jx) const override
Is there a branch cut at this processor’s upper boundary?

Parameters jx – [in] The local (on this processor) index in X

Returns pair<bool, BoutReal> - bool is true if there is a branch cut, BoutReal gives the total
zShift for a 2pi poloidal circuit if there is a branch cut

virtual int ySize(int jx) const override
The number of points in Y at fixed X index jx.

virtual bool firstY() const override
Is this processor first in Y? i.e. is there a boundary at lower Y?
virtual bool lastY() const override
   Is this processor last in Y? i.e. is there a boundary at upper Y?

virtual bool firstY(int xpos) const override
   Is this processor first in Y? i.e. is there a boundary at lower Y?

virtual bool lastY(int xpos) const override
   Is this processor last in Y? i.e. is there a boundary at upper Y?

virtual int UpXSplitIndex() override
   If the upper Y guard cells are split in two, return the X index where the split occurs.

virtual int DownXSplitIndex() override
   If the lower Y guard cells are split in two, return the X index where the split occurs.

virtual int sendYOutIndest(BoutReal *buffer, int size, int tag) override
   Send data.

virtual int sendYOutOutdest(BoutReal *buffer, int size, int tag) override

virtual int sendYInIndest(BoutReal *buffer, int size, int tag) override

virtual int sendYInOutdest(BoutReal *buffer, int size, int tag) override

virtual comm_handle irecvYOutIndest(BoutReal *buffer, int size, int tag) override
   Non-blocking receive. Must be followed by a call to \texttt{wait()}

   Parameters
   \begin{itemize}
      \item buffer -- \texttt{[out]} A buffer of length \texttt{size} which must already be allocated
      \item size -- \texttt{[in]} The number of BoutReals expected
      \item tag -- \texttt{[in]} The tag number of the expected message
   \end{itemize}

virtual comm_handle irecvYOutOutdest(BoutReal *buffer, int size, int tag) override
   Non-blocking receive. Must be followed by a call to \texttt{wait()}

   Parameters
   \begin{itemize}
      \item buffer -- \texttt{[out]} A buffer of length \texttt{size} which must already be allocated
      \item size -- \texttt{[in]} The number of BoutReals expected
      \item tag -- \texttt{[in]} The tag number of the expected message
   \end{itemize}

virtual comm_handle irecvYInIndest(BoutReal *buffer, int size, int tag) override
   Non-blocking receive. Must be followed by a call to \texttt{wait()}

   Parameters
   \begin{itemize}
      \item buffer -- \texttt{[out]} A buffer of length \texttt{size} which must already be allocated
      \item size -- \texttt{[in]} The number of BoutReals expected
      \item tag -- \texttt{[in]} The tag number of the expected message
   \end{itemize}

virtual comm_handle irecvYInOutdest(BoutReal *buffer, int size, int tag) override
   Non-blocking receive. Must be followed by a call to \texttt{wait()}

   Parameters
   \begin{itemize}
      \item buffer -- \texttt{[out]} A buffer of length \texttt{size} which must already be allocated
- **size** – [in] The number of BoutReals expected
- **tag** – [in] The tag number of the expected message

virtual const RangeIterator iterateBndryLowerY() const override
Iterate over the lower Y boundary.

virtual const RangeIterator iterateBndryUpperY() const override
Iterate over the upper Y boundary.

virtual const RangeIterator iterateBndryLowerInnerY() const override

virtual const RangeIterator iterateBndryLowerOuterY() const override

virtual const RangeIterator iterateBndryUpperInnerY() const override

virtual const RangeIterator iterateBndryUpperOuterY() const override

virtual std::vector<BoundaryRegion*> getBoundaries() override
Return a vector containing all the boundary regions on this processor.

virtual std::vector<BoundaryRegionPar*> getBoundariesPar() override
Get all the parallel (Y) boundaries on this processor.

virtual void addBoundaryPar(BoundaryRegionPar *bndry) override
Add a parallel(Y) boundary to this processor.

virtual const Field3D smoothSeparatrix(const Field3D &f) override
Branch-cut special handling (experimental)

inline int getNx() const

inline int getNy() const

virtual BoutReal GlobalX(int jx) const override
Continuous X index between 0 and 1.

virtual BoutReal GlobalY(int jy) const override
Continuous Y index (0 -> 1)

virtual BoutReal GlobalX(BoutReal jx) const override
Continuous X index between 0 and 1.

virtual BoutReal GlobalY(BoutReal jy) const override
Continuous Y index (0 -> 1)

inline BoutReal getIxseps1() const

inline BoutReal getIxseps2() const

virtual void outputVars(Datafile &file) override
Add output variables to a data file These are used for post-processing

virtual int getGlobalXIndex(int xlocal) const override
Returns a global X index given a local index. Global index includes boundary cells, local index includes boundary or guard cells.
virtual int getGlobalXIndexNoBoundaries(int xlocal) const override
    Returns a global X index given a local index. Global index excludes boundary cells, local index includes boundary or guard cells.

virtual int getLocalXIndex(int xglobal) const override
    Returns a local X index given a global index. Global index includes boundary cells, local index includes boundary or guard cells.

virtual int getLocalXIndexNoBoundaries(int xglobal) const override
    Returns a local X index given a global index. Global index excludes boundary cells, local index includes boundary or guard cells.

virtual int getGlobalYIndex(int ylocal) const override
    Returns a global Y index given a local index. Global index includes boundary cells, local index includes boundary or guard cells.

virtual int getGlobalYIndexNoBoundaries(int ylocal) const override
    Returns a global Y index given a local index. Global index excludes boundary cells, local index includes boundary or guard cells.

virtual int getLocalYIndex(int yglobal) const override
    Returns a local Y index given a global index. Global index includes boundary cells, local index includes boundary or guard cells.

virtual int getLocalYIndexNoBoundaries(int yglobal) const override
    Returns a local Y index given a global index. Global index excludes boundary cells, local index includes boundary or guard cells.

virtual int getGlobalZIndex(int zlocal) const override
    Returns a global Z index given a local index. Global index includes boundary cells, local index includes boundary or guard cells.

virtual int getGlobalZIndexNoBoundaries(int zlocal) const override
    Returns a global Z index given a local index. Global index excludes boundary cells, local index includes boundary or guard cells. Note: at the moment z-direction is always periodic, so has zero boundary cells.

virtual int getLocalZIndex(int zglobal) const override
    Returns a local Z index given a global index. Global index includes boundary cells, local index includes boundary or guard cells.

virtual int getLocalZIndexNoBoundaries(int zglobal) const override
    Returns a local Z index given a global index. Global index excludes boundary cells, local index includes boundary or guard cells.

Private Functions

int XGLOBAL(BoutReal xloc, BoutReal &xglo) const
    Returns the global X index given a local index.

int YGLOBAL(BoutReal yloc, BoutReal &yglo) const
    Returns the global Y index given a local index.

int PROC_NUM(int xind, int yind)
    Returns the processor number, given X and Y processor indices.

    If out of range returns -1 (no processor)

int YGLOBAL(int yloc, int yproc) const
    Global Y index given local index and processor.
int \texttt{YLOCAL}(\texttt{int yglo, int yproc}) \texttt{const}

int \texttt{YPROC}(\texttt{int yind})
\hspace{1cm} \text{Return the Y processor number given a global Y index.}

int \texttt{XPROC}(\texttt{int xind})
\hspace{1cm} \text{Return the X processor number given a global X index.}

\textbf{void} \texttt{default\_connections()}
\hspace{1cm} \text{Connection initialisation: Set processors in a simple 2D grid.}

\textbf{void} \texttt{set\_connection}(\texttt{int ypos1, int ypos2, int xge, int xlt, bool ts = false})
\hspace{1cm} \text{Add a topology connection.}
\hspace{1cm} \text{Set ypos1 and ypos2 to be neighbours in the range xge <= x < xlt. Optional argument ts sets whether to use twist-shift condition}

\textbf{void} \texttt{add\_target}(\texttt{int ypos, int xge, int xlt})
\hspace{1cm} \text{Add a divertor target or limiter.}
\hspace{1cm} \text{ypos is the y index which will become an upper target ypos+1 will become a lower target. Target created in the range xge <= x < xlt.}

\textbf{void} \texttt{topology()}

\textbf{void} \texttt{addBoundaryRegions()}
\hspace{1cm} \text{Adds 2D and 3D regions for boundaries.}

\textbf{void} \texttt{free\_handle}(\texttt{CommHandle *h})

\texttt{CommHandle *} \texttt{get\_handle}(\texttt{int xlen, int ylen})

\textbf{void} \texttt{clear\_handles()}

\textbf{void} \texttt{post\_receive}(\texttt{CommHandle &ch})
\hspace{1cm} \text{Create the MPI requests to receive data. Non-blocking call.}

\textbf{int} \texttt{pack\_data}(\texttt{std::vector<\textit{FieldData}*} &\textit{var\_list, int xge, int xlt, int yge, int ylt, BoutReal *buffer})
\hspace{1cm} \text{Take data from objects and put into a buffer.}

\textbf{int} \texttt{unpack\_data}(\texttt{std::vector<\textit{FieldData}*} &\textit{var\_list, int xge, int xlt, int yge, int ylt, BoutReal *buffer})
\hspace{1cm} \text{Copy data from a buffer back into the fields.}

\textbf{Private Members}

\texttt{std::string gridname}

int \texttt{nx}

int \texttt{ny}

int \texttt{nz}
\hspace{1cm} \text{Size of the grid in the input file.}

int \texttt{MX}

int \texttt{MY}
int MZ
    size of the grid excluding boundary regions

int MYSUB
int MXSUB
int MZSUB
    Size of the grid on this processor.

int NPES
    Number of processors.

int MYPE
    Rank of this processor.

int PE_YIND
    Y index of this processor.

int NYPE
int NZPE
int MYPE_IN_CORE
int ixseps1
int ixseps2
int jyseps1_1
int jyseps2_1
int jyseps1_2
int jyseps2_2
int ixseps_inner
int ixseps_outer
int ixseps_upper
int ixseps_lower
int ny_inner

*std::vector<BoutReal>* ShiftAngle
    Angle for twist-shift location.

bool TS_up_in
bool TS_up_out
bool TS_down_in
bool TS_down_out
int UDATA_INDEST
int UDATA_OUTDEST
int UDATA_XSPLIT
int DDATA_INDEST
int DDATA_OUTDEST
int DDATA_XSPLIT
int IDATA_DEST
int ODATA_DEST
bool TwistShift

bool symmetricGlobalX
   Use a symmetric definition in GlobalX() function.

bool symmetricGlobalY

int zperiod

*BoutReal* ZMIN

*BoutReal* ZMAX

int MXG
int MYG
int MZG

*std::string* grid_id = ""

*std::string* hypnotoad_version = ""

*std::string* hypnotoad_git_hash = ""

*std::string* hypnotoad_git_diff = ""

*std::string* hypnotoad_geqdsk_filename = ""

*std::vector<BoundaryRegion*>* boundary

*std::vector<BoundaryRegionPar*>* par_boundary

bool async_send
   Switch to asynchronous sends (ISend, not Send)

*std::list<CommHandle*>* comm_list

MPI_Comm comm_x
   Communicator containing all processors in X.

MPI_Comm comm_inner
   Communicators in Y. Inside both separatrices; between separatrices; and outside both separatrices

MPI_Comm comm_middle

MPI_Comm comm_outer

struct CommHandle
   Communication handle Used to keep track of communications between send and receive
Public Members

MPI_Request request[6]
   Array of receive requests. One for each possible neighbour; one each way in X, two each way in Y

MPI_Request sendreq[6]
   Array of send requests (for non-blocking send). One for each possible neighbour; one each way in X, two each way in Y

int xbufflen
   Length of the buffers used to send/receive (in BoutReals)

int ybufflen
   Array<BoutReal> umsg_sendbuff
      Sending buffers.
   Array<BoutReal> dmsg_sendbuff
   Array<BoutReal> imsg_sendbuff
   Array<BoutReal> omsg_sendbuff
   Array<BoutReal> umsg_recvbuff
      Receiving buffers.
   Array<BoutReal> dmsg_recvbuff
   Array<BoutReal> imsg_recvbuff
   Array<BoutReal> omsg_recvbuff

bool in_progress
   Is the communication still going?

FieldGroup var_list
   List of fields being communicated.

K.2.29 File cashkarp.cxx

K.2.30 File cashkarp.hxx

class CASKARPScheme : public RKScheme
Public Functions

CASHKARPScheme(Options *options)

K.2.31 File constants.hxx

Variables

constexpr BoutReal PI = 3.141592653589793
    Mathematical constant pi.

constexpr BoutReal TWOPI = 2 * PI
    Mathematical constant 2 * pi.

namespace SI

Variables

constexpr BoutReal c = 299792458
    Speed of light in vacuum.

constexpr BoutReal mu0 = 4.e-7 * PI
    Permeability of free space.

constexpr BoutReal e0 = 1 / (c * c * mu0)
    Permittivity of free space.

constexpr BoutReal qe = 1.602176634e-19
    Electron charge.

constexpr BoutReal Me = 9.10938356e-31
    Electron mass.

constexpr BoutReal Mp = 1.672621898e-27
    Proton mass.

constexpr BoutReal kb = 1.38064852e-23
    Boltzmanns constant.

constexpr BoutReal amu = 1.660539040e-27
    Unified atomic mass unit.

constexpr BoutReal M_Hydrogen = 1.008 * amu
    Mass of a Hydrogen atom.

constexpr BoutReal M_Deuterium = 2.01410178 * amu
    Mass of a Deuterium atom.
constexpr BoutReal M_Tritium = 3.0160492 * amu
    Mass of a Tritium atom.

**K.2.32 File coordinates.cxx**

**K.2.33 File coordinates.hxx**

Defines

`DERIV_FUNC_REGION_ENUM_TO_STRING(func, T)`

`GRAD_FUNC_REGION_ENUM_TO_STRING(func, T)`

class `Coordinates`

    `#include <coordinates.hxx>` Represents a coordinate system, and associated operators

    This is a container for a collection of metric tensor components

**Public Functions**

`Coordinates(Mesh *mesh, Options *options = nullptr)`

    Standard constructor from input.

`Coordinates(Mesh *mesh, Options *options, const CELL_LOC loc, const Coordinates *coords_in, bool force_interpolate_from_centre = false)`

    Constructor interpolating from another `Coordinates` object By default attempts to read staggered `Coordinates` from grid data source, interpolating from CELL_CENTRE if not present. Set `force_interpolate_from_centre` argument to true to always interpolate (useful if CELL_CENTRE Coordinates have been changed, so reading from file would not be correct).

`Coordinates(Mesh *mesh, Field2D dx, Field2D dy, BoutReal dz, Field2D J, Field2D Bxy, Field2D g11, Field2D g22, Field2D g33, Field2D g12, Field2D g13, Field2D g_11, Field2D g_22, Field2D g_33, Field2D g_12, Field2D g_13, Field2D g_23, Field2D ShiftTorsion, Field2D IntShiftTorsion, bool calculate_geometry = true)`

    A constructor useful for testing purposes. To use it, inherit from `Coordinates`. If `calculate_geometry` is true (default), calculate the non-uniform variables, Christoffel symbols

    `Coordinates &operator=(Coordinates&&) = default`

    `~Coordinates() = default`

    void `outputVars(Datafile &file)`

    Adds variables to the output file, for post-processing

    Must be a better way so that `Coordinates` doesn’t depend on `Datafile`

    inline `BoutReal zlength() const`

    Length of the Z domain. Used for FFTs.

    int `geometry(bool recalculate_staggered = true, bool force_interpolate_from_centre = false)`

    Calculate differential geometry quantities from the metric tensor.
int calcCovariant(const std::string &region = "RGN_ALL")
   Invert contravariant metric to get covariant components.

int calcContravariant(const std::string &region = "RGN_ALL")
   Invert covariant metric to get contravariant components.

int jacobian()
   Calculate J and Bxy.

inline void setParallelTransform(std::unique_ptr<ParallelTransform> pt)
   Set the parallel (y) transform for this mesh. Mostly useful for tests.

inline ParallelTransform &getParallelTransform()
   Return the parallel transform.

Field2D DDX(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

Field2D DDX(const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)

Field2D DDX(const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region = RGN_NOBNDRY)

Field2D DDY(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

Field2D DDY(const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)

Field2D DDY(const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region = RGN_NOBNDRY)

Field2D DDZ(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

Field2D DDZ(const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)

Field2D DDZ(const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region = RGN_NOBNDRY)

Field2D Grad_par(const Field2D &var, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")
   Gradient along magnetic field b.Grad(f)

Field3D Grad_par(const Field3D &f, CELL_LOC outloc, DIFF_METHOD method)

Field3D Grad_par(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")

Field3D Grad_par(const Field3D &f, CELL_LOC outloc, DIFF_METHOD method)
Field2D Vpar_Grad_par(const Field2D &v, const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")
Advection along magnetic field V*b.Grad(f)

inline Field2D Vpar_Grad_par(const Field2D &v, const Field2D &f, CELL_LOC outloc, DIFF_METHOD method)

Field3D Vpar_Grad_par(const Field3D &v, const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")

inline Field3D Vpar_Grad_par(const Field3D &v, const Field3D &f, CELL_LOC outloc, DIFF_METHOD method)

Field2D Div_par(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")
Divergence along magnetic field Div(b*f) = B.Grad(f/B)

inline Field2D Div_par(const Field2D &f, CELL_LOC outloc, DIFF_METHOD method)

Field3D Div_par(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")

inline Field3D Div_par(const Field3D &f, CELL_LOC outloc, DIFF_METHOD method)

Field2D Grad2_par2(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")

inline Field2D Grad2_par2(const Field2D &f, CELL_LOC outloc, DIFF_METHOD method)

Field3D Grad2_par2(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")

inline Field3D Grad2_par2(const Field3D &f, CELL_LOC outloc, DIFF_METHOD method)

Field2D Delp2(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, bool useFFT = true)

Field3D Delp2(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, bool useFFT = true)

FieldPerp Delp2(const FieldPerp &f, CELL_LOC outloc = CELL_DEFAULT, bool useFFT = true)

Field2D Laplace_par(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT)

Field3D Laplace_par(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT)

Field2D Laplace(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT)
Field3D Laplace(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT)

Public Members

Field2D dx

Field2D dy
    Mesh spacing in x and y.

BoutReal dz
    Mesh spacing in Z.

bool non_uniform
    True if corrections for non-uniform mesh spacing should be included in operators.

Field2D d1_dx

Field2D d1_dy
    2nd-order correction for non-uniform meshes d/di(1/dx) and d/di(1/dy)

Field2D J
    Coordinate system Jacobian, so volume of cell is J*dx*dy*dz.

Field2D Bxy
    Magnitude of B = nabla z times nabla x.

Field2D g11
    Contravariant metric tensor (g^{ij})

Field2D g22
Field2D g33
Field2D g12
Field2D g13
Field2D g23
Field2D g_11
    Covariant metric tensor.

Field2D g_22
Field2D g_33
Field2D g_12
Field2D g_13
Field2D g_23
Field2D G1_11
    Christoffel symbol of the second kind (connection coefficients)

Field2D G1_22
Field2D G1_33
Field2D G1_12
Field2D G1_13
Field2D G1_23
Field2D G2_11
Field2D G2_22
Field2D G2_33
Field2D G2_12
Field2D G2_13
Field2D G2_23
Field2D G3_11
Field2D G3_22
Field2D G3_33
Field2D G3_12
Field2D G3_13
Field2D G3_23
Field2D G1
Field2D G2
Field2D G3

Field2D ShiftTorsion
\[ \text{d pitch angle / dx. Needed for vector differentials (Curl)} \]

Field2D IntShiftTorsion
Integrated shear (I in BOUT notation)

Private Functions

void setParallelTransform(Options *options)
\[ \text{Set the parallel (y) transform from the options file. Used in the constructor to create the transform object.} \]

Private Members

int nz
Mesh *localmesh
CELL_LOC location

std::unique_ptr<ParallelTransform> transform = {nullptr}
\[ \text{Handles calculation of yup and ydown.} \]
K.2.34 File cvode.cxx

Defines

ZERO
ONE

Typedefs

using CVODEINT = bout::utils::function_traits<CVLocalFn>::arg_t<0>

Functions

BOUT_ENUM_CLASS (positivity_constraint, none, positive, non_negative, negative, non_positive)

static int cvode_rhs (BoutReal t, N_Vector u, N_Vector du, void *user_data)

static int cvode_bbd_rhs (CVODEINT Nlocal, BoutReal t, N_Vector u, N_Vector du, void *user_data)
    RHS function for BBD preconditioner.

static int cvode_pre (BoutReal t, N_Vector yy, N_Vector yp, N_Vector rvec, N_Vector zvec, BoutReal gamma, BoutReal delta, int lr, void *user_data)
    Preconditioner function.

static inline int cvode_pre_shim (BoutReal t, N_Vector yy, N_Vector yp, N_Vector rvec, N_Vector zvec, BoutReal gamma, BoutReal delta, int lr, void *user_data, N_Vector tmp)

static int cvode_jac (N_Vector v, N_Vector Jv, realtype t, N_Vector y, N_Vector fy, void *user_data, N_Vector tmp)
    Jacobian-vector multiplication function.

inline int CVSpilsSetJacTimes (void *arkode_mem, std::nullptr_t, CVSpilsJacTimesVecFn jtimes)

K.2.35 File cvode.hxx

class CvodeSolver : public Solver

Public Functions

CvodeSolver (Options *opts = nullptr)

~CvodeSolver ()

inline virtual BoutReal getCurrentTimestep () override
    Return the current internal timestep.
virtual int init(int nout, BoutReal tstep) override
    Initialise the solver NOTE: nout and tstep should be passed to run, not init. Needed because of how the
    PETSc TS code works
virtual int run() override
    Run the solver, calling monitors nout times, at intervals of tstep. This function is called by solve(), and is
    specific to each solver type
    This should probably be protected, since it shouldn’t be called by users.
BoutReal run(BoutReal tout)

virtual void resetInternalFields() override
    Should wipe out internal field vector and reset from current field object data.
void rhs(BoutReal t, BoutReal *udata, BoutReal *dudata)

void pre(BoutReal t, BoutReal gamma, BoutReal delta, BoutReal *udata, BoutReal *rvec, BoutReal *zvec)

void jac(BoutReal t, BoutReal *ydata, BoutReal *vdata, BoutReal *Jvdata)

Private Functions

void set_vector_option_values(BoutReal *option_data, std::vector<BoutReal> &f2dtols,
                             std::vector<BoutReal> &f3dtols)

void loop_vector_option_values_op(Ind2D i2d, BoutReal *option_data, int &p,
                                  std::vector<BoutReal> &f2dtols,
                                  std::vector<BoutReal> &f3dtols, bool bndry)

template<class FieldType>
std::vector<BoutReal> create_constraints(const std::vector<VarStr<FieldType>> &fields)

Private Members

int NOUT

BoutReal TIMESTEP

BoutReal hcur

bool diagnose = {false}
N_Vector uvec = {nullptr}
void *cvode_mem = {nullptr}
BoutReal pre_Wtime = {0.0}
int pre_ncalls = {0}
int nsteps = {0}
int nfevals = {0}
int \texttt{nniters} = \{0\}
int \texttt{npevals} = \{0\}
int \texttt{nliters} = \{0\}

\textit{BoutReal} \ \texttt{last\_step} = \{0.0\}
int \texttt{last\_order} = \{0\}
int \texttt{num\_fails} = \{0\}
int \texttt{nonlin\_fails} = \{0\}
int \texttt{stab\_lims} = \{0\}
bool \texttt{cvode\_initialised} = \text{false}

\textbf{K.2.36 File cyclic.cxx}

FFT + Tridiagonal solver in serial or parallel.
Not particularly optimised: Each y slice is solved sequentially

\textit{CHANGELOG}

Jan 2014: Brendan Shanahan \texttt{bws502@york.ac.uk}
• Added DST option

Copyright 2013 B.D.Dudson
Contact: Ben Dudson, \texttt{benjamin.dudson@york.ac.uk}

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see \texttt{http://www.gnu.org/licenses/}.

\textbf{K.2.37 File cyclic.hxx}

class \texttt{InvertParCR} : public \texttt{InvertPar}
Public Functions

**InvertParCR** (*Options* *opt*, *CELL_LOC* location = *CELL_CENTRE*, *Mesh* *mesh_in* = *bout::globals::mesh*)

virtual const *Field3D* **solve**(const *Field3D* &f) override
Solve the system of equations

This method must be implemented

inline virtual void **setCoefA**(const *Field2D* &f) override
*Set* the constant coefficient A

inline virtual void **setCoefB**(const *Field2D* &f) override
*Set* the Grad2_par2 coefficient B

inline virtual void **setCoefC**(const *Field2D* &f) override
*Set* the D2DYDZ coefficient C

inline virtual void **setCoefD**(const *Field2D* &f) override
*Set* the D2DZ2 coefficient D

inline virtual void **setCoefE**(const *Field2D* &f) override
*Set* the DDY coefficient E

Private Members

*Field2D* **A** = {0.0}

*Field2D* **B** = {0.0}

*Field2D* **C** = {0.0}

*Field2D* **D** = {0.0}

*Field2D* **E** = {0.0}

*Field2D* **sg**

int **nsys**

K.2.38 File cyclic_laplace.cxx

K.2.39 File cyclic_laplace.hxx

class **LaplaceCyclic** : public **Laplacian**
  #include <cyclic_laplace.hxx> Solves the 2D Laplacian equation using the CyclicReduce class.
Public Functions

LaplaceCyclic(*opt = nullptr, const CELL_LOC loc = CELL_CENTRE, Mesh *mesh_in = nullptr)

~LaplaceCyclic()

inline virtual void setCoefA(const Field2D &val) override

inline virtual void setCoefC(const Field2D &val) override

inline virtual void setCoefC1(const Field2D &val) override

inline virtual void setCoefC2(const Field2D &val) override

inline virtual void setCoefD(const Field2D &val) override

inline virtual void setCoefEx(const Field2D &val) override

inline virtual void setCoefEz(const Field2D &val) override

inline virtual FieldPerp solve(const FieldPerp &b) override

virtual FieldPerp solve(const FieldPerp &b, const FieldPerp &x0) override

virtual Field3D solve(const Field3D &b, const Field3D &x0) override

Performs the laplacian inversion y-slice by y-slice

Parameters

- \( b \) – [in] All the y-slices of \( b_{\text{slice}} \), which is the right hand side of the equation \( A \cdot x_{\text{slice}} = b_{\text{slice}} \)
- \( x_0 \) – [in] All the y-slices of the variable eventually used to set BC

Returns \( x \) All the y-slices of \( x_{\text{slice}} \) in the equation \( A \cdot x_{\text{slice}} = b_{\text{slice}} \)

Private Members

\( \text{Field2D } A_{\text{coef}} \)
\( \text{Field2D } C_{1\text{coef}} \)
\( \text{Field2D } C_{2\text{coef}} \)
\( \text{Field2D } D_{\text{coef}} \)

int nmode
int xs
int xe
Matrix<dcomplex> a
Matrix<dcomplex> b
Matrix<dcomplex> c
Matrix<dcomplex> bcmplx
Matrix<dcomplex> xcmplx
bool dst
CyclicReduce<dcomplex> *cr
Tridiagonal solver.

K.2.40 File cyclic_reduction.hxx

template<class T>
class CyclicReduce

Public Functions

CyclicReduce() = default

inline CyclicReduce(MPI_Comm c, int size)

inline void setup(MPI_Comm c, int size)
Set parameters

Parameters

• c – [in] The communicator of all processors involved in the solve
• size – [in] The number of rows on this processor

~CyclicReduce() = default

inline void setPeriodic(bool p = true)
Specify that the tridiagonal system is periodic By default not periodic

inline void setCoefs(const Array<T> &a, const Array<T> &b, const Array<T> &c)

inline void setCoefs(const Matrix<T> &a, const Matrix<T> &b, const Matrix<T> &c)
Set the entries in the matrix to be inverted

Parameters

• a – [in] Left diagonal. Should have size [nsys][N] where N is set in the constructor or setup
• b – [in] Diagonal values. Should have size [nsys][N]
• c – [in] Right diagonal. Should have size [nsys][N]

inline void solve(const Array<T> &rhs, Array<T> &x)
Solve a set of tridiagonal systems
Parameters

- **rhs** – [in] *Array* storing Values of the rhs for a single system
- **x** – [out] *Array* storing the result for a single system

inline void **solve**(const *Matrix*<T>& rhs, *Matrix*<T>& x)
    Solve a set of tridiagonal systems

Parameters

- **rhs** – [in] *Matrix* storing Values of the rhs for each system
- **x** – [out] *Matrix* storing the result for each system

Private Functions

inline void **allocMemory**(int np, int nsys, int n)
    Allocate memory arrays

Parameters

- **np** – [in] Number of processors
- **nsys** – [in] Number of independent systems to solve
- **n** – [in] Size of each system of equations

inline void **reduce**(int ns, int nloc, *Matrix*<T>& co, *Matrix*<T>& ifc)
    Calculate interface equations
    This reduces *ns* separate systems of equations, each consisting of *nloc* rows on this processor, to two inter-
    face rows for each system, which are stored in ifc.
    
    \[(a_1 b_1 c_1) (a_2 b_2 c_2) (A_1 B_1 C_1) (a_3 b_3 c_3) \Rightarrow (A_2 B_2 C_2) (\ldots) (a_n b_n c_n)\]

inline void **back_solve**(int ns, int nloc, const *Matrix*<T>& co, const *Array*<T>& x1, const *Array*<T>& xn, *Matrix*<T>& xa)
    Back-solve from x at ends (x1, xn) to obtain remaining values Coefficients ordered [ns, nloc*(a,b,c,r)]

Private Members

MPI_Comm **comm**
    Communicator.

int **nprocs** = {0}
int **myproc** = {-1}
    Number of processors and ID of my processor.

int **N** = {0}
    Total size of the problem.

int **Nsys** = {0}
    Number of independent systems to solve.

int **myns**
    Number of systems for interface solve on this processor.
int sys0
    Starting system index for interface solve.

bool periodic = {false}
    Is the domain periodic?

Matrix<T> coefs
    Starting coefficients, rhs [Nsys, {3*coef.rhs}*N].

Matrix<T> myif
    Interface equations for this processor.

Matrix<T> recvbuffer
    Buffer for receiving from other processors.

Matrix<T> ifcs
    Coefficients for interface solve.

Matrix<T> if2x2
    2x2 interface equations on this processor

Matrix<T> ifx
    Solution of interface equations.

Array<T> ifp
    Interface equations returned to processor p.

Array<T> x1

Array<T> xn
    Interface solutions for back-solving.

K.2.41 File datafile.cxx

K.2.42 File datafile.hxx

Data file handling object definition.

26th Sep 2009: Modified to use varargs

Author  B.Dudson

Date  April 2009
Defines

SAVE_ONCE1(var)
    Write this variable once to the grid file.

SAVE_ONCE2(var1, var2)

SAVE_ONCE3(var1, var2, var3)

SAVE_ONCE4(var1, var2, var3, var4)

SAVE_ONCE5(var1, var2, var3, var4, var5)

SAVE_ONCE6(var1, var2, var3, var4, var5, var6)

SAVE_ONCE(...)

SAVE_REPEAT1(var)
    Write this variable every timestep.

SAVE_REPEAT2(var1, var2)

SAVE_REPEAT3(var1, var2, var3)

SAVE_REPEAT4(var1, var2, var3, var4)

SAVE_REPEAT5(var1, var2, var3, var4, var5)

SAVE_REPEAT6(var1, var2, var3, var4, var5, var6)

SAVE_REPEAT(...)

class Datafile
    #include <datafile.hxx> Uses a generic interface to file formats (DataFormat) and provides an interface for reading/writing simulation data.

Public Functions

Datafile(Options *opt = nullptr, Mesh *mesh_in = nullptr)

Datafile(Datafile &&other) noexcept

~Datafile()

Datafile &operator=(Datafile &&rhs) noexcept
```cpp
Datafile &operator=(const Datafile &rhs) = delete

bool openr(const char *filename, ...)

bool openw(const char *filename, ...)

bool opena(const char *filename, ...)

bool isValid()

void close()

void setLowPrecision()
    Only output floats.

template<typename T>
inline void addRepeat(T &value, std::string name)

template<typename T>
inline void addOnce(T &value, std::string name)

void add(int &i, const char *name, bool save_repeat = false, const std::string &description = "")

void add(std::vector<int> &ivec, const char *name, bool save_repeat = false, const std::string &description = "")

void add(std::string &s, const char *name, bool save_repeat = false, const std::string &description = "")

void add(BoutReal &r, const char *name, bool save_repeat = false, const std::string &description = "")

void add(bool &b, const char *name, bool save_repeat = false, const std::string &description = "")

void add(Field2D &f, const char *name, bool save_repeat = false, const std::string &description = "")

void add(Field3D &f, const char *name, bool save_repeat = false, const std::string &description = "")

void add(FieldPerp &f, const char *name, bool save_repeat = false, const std::string &description = "")

void add(Vector2D &f, const char *name, bool save_repeat = false, const std::string &description = "")

void add(Vector3D &f, const char *name, bool save_repeat = false, const std::string &description = "")

bool read()
    Read data into added variables.
```
bool write()
    Write added variables.
bool write(const char *filename, ...) const
    Opens, writes, closes file.
void setAttribute(const std::string &varname, const std::string &attrname, const std::string &text)
void setAttribute(const std::string &varname, const std::string &attrname, int value)
void setAttribute(const std::string &varname, const std::string &attrname, BoutReal value)

inline bool can_write_strings()

Private Functions

Datafile(const Datafile &other)
    Shallow copy, not including dataformat, therefore private.
bool read_f2d(const std::string &name, Field2D *f, bool save_repeat)
bool read_f3d(const std::string &name, Field3D *f, bool save_repeat)
bool read_fperp(const std::string &name, FieldPerp *f, bool save_repeat)
bool write_int(const std::string &name, int *f, bool save_repeat)
bool write_int_vec(const std::string &name, std::vector<int> *f, bool save_repeat)
bool write_string(const std::string &name, std::string *f, bool save_repeat)
bool write_real(const std::string &name, BoutReal *f, bool save_repeat)
bool write_f2d(const std::string &name, Field2D *f, bool save_repeat)
bool write_f3d(const std::string &name, Field3D *f, bool save_repeat)
bool write_fperp(const std::string &name, FieldPerp *f, bool save_repeat)
bool varAdded(const std::string &name)
    Check if a variable has already been added.
void *varPtr(const std::string &name)
    Get the pointer to the variable, nullptr if not added. This is used to check if the same variable is being added
Private Members

Mesh *mesh
bool parallel = {false}
bool flush = {true}
bool guards = {true}
bool floats = {false}
bool openclose = {true}
int Lx
int Ly
int Lz
bool enabled = {true}
bool init_missing
bool shiftOutput = {false}
bool shiftInput = {false}
int flushFrequencyCounter = {0}
int flushFrequency = {1}
std::unique_ptr<DataFormat> file
size_t filenamelen
char *filename
bool writable = {false}
bool appending = {false}
bool first_time = {true}
std::vector<VarStr<int>> int_arr
std::vector<VarStr<std::vector<int>>> int_vec_arr
std::vector<VarStr<std::vector<int>>> int_vec_arr
std::vector<VarStr<std::string>> string_arr
std::vector<VarStr<BoutReal>> BoutReal_arr
std::vector<VarStr<bool>> bool_arr
std::vector<VarStr<Field2D>> f2d_arr
std::vector<VarStr<Field3D>> f3d_arr
std::vector<VarStr<FieldPerp>> fperp_arr
std::vector<VarStr<Vector2D>> v2d_arr
std::vector<VarStr<Vector3D>> v3d_arr
Private Static Attributes

static const size_t FILENAMELEN = 512

template<class T>
struct VarStr
  A structure to hold a pointer to a class, and associated name and flags.

Public Members

*T *ptr
  Pointer to the data. Note that this may be a user object, not a copy, so must not be destroyed

std::string name
  Name as it appears in the output file.

bool save_repeat
  If true, has a time dimension and is saved every time step.

bool covar
  For vectors, true if a covariant vector, false if contravariant.

size_t size
  Size of a stored vector or string, to check it does not change after being added.

std::string description = "{""
  Documentation of what the variable is.

K.2.43 File dataformat.cxx

K.2.44 File dataformat.hxx

Generic interface for file formats e.g. netCDF, HDF5.

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Author  B.Dudson

Date  April 2009

Contact Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.
You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

Functions

```cpp
std::unique_ptr<DataFormat> data_format(const char *filename = nullptr)
```

class DataFormat

Subclassed by H5Format, NcFormat, Ncxt4, PncFormat

Public Functions

```cpp
DataFormat(Mesh *mesh_in = nullptr)
```

virtual  ```cpp
~DataFormat() = default
```

virtual bool ```cpp
openr(const char *name) = 0
```

inline virtual bool ```cpp
openr(const std::string &name)
```

virtual bool ```cpp
openr(const std::string &base, int mype)
```

virtual bool ```cpp
openw(const char *name, bool append = false) = 0
```

inline virtual bool ```cpp
openw(const std::string &name, bool append = false)
```

virtual bool ```cpp
openw(const std::string &base, int mype, bool append = false)
```

virtual bool is_valid() = 0

virtual void close() = 0

virtual void flush() = 0

virtual const std::vector<int> getSize(const char *var) = 0

virtual const std::vector<int> getSize(const std::string &var) = 0

virtual bool setGlobalOrigin(int x = 0, int y = 0, int z = 0) = 0

virtual bool setLocalOrigin(int x = 0, int y = 0, int z = 0, int offset_x = 0, int offset_y = 0, int offset_z = 0)

virtual bool setRecord(int t) = 0
virtual bool addVarInt(const std::string &name, bool repeat) = 0

virtual bool addVarIntVec(const std::string &name, bool repeat, size_t size) = 0

virtual bool addVarString(const std::string &name, bool repeat, size_t size) = 0

virtual bool addVarBoutReal(const std::string &name, bool repeat) = 0

virtual bool addVarField2D(const std::string &name, bool repeat) = 0

virtual bool addVarField3D(const std::string &name, bool repeat) = 0

virtual bool addVarFieldPerp(const std::string &name, bool repeat) = 0

virtual bool read(int *var, const char *name, int lx = 1, int ly = 0, int lz = 0) = 0

virtual bool read(int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) = 0

virtual bool read(char *var, const char *name, int n = 1) = 0

virtual bool read(char *var, const std::string &name, int n = 1) = 0

virtual bool read(BoutReal *var, const char *name, int lx = 1, int ly = 0, int lz = 0) = 0

virtual bool read(BoutReal *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) = 0

virtual bool read_perp(BoutReal *var, const std::string &name, int lx = 1, int lz = 0) = 0

virtual bool write(int *var, const char *name, int lx = 0, int ly = 0, int lz = 0) = 0

virtual bool write(int *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) = 0

virtual bool write(char *var, const char *name, int n = 1) = 0

virtual bool write(char *var, const std::string &name, int n = 1) = 0

virtual bool write(BoutReal *var, const char *name, int lx = 0, int ly = 0, int lz = 0) = 0

virtual bool write(BoutReal *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) = 0

virtual bool write_perp(BoutReal *var, const std::string &name, int lx = 0, int lz = 0) = 0

virtual bool write_perp(BoutReal *var, const std::string &name, int lx = 0, int lz = 0) = 0
virtual bool read_rec(int *var, const char *name, int lx = 1, int ly = 0, int lz = 0) = 0

virtual bool read_rec(int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) = 0

virtual bool read_rec(char *var, const char *name, int n = 1) = 0

virtual bool read_rec(char *var, const std::string &name, int n = 1) = 0

virtual bool read_rec(BoutReal *var, const char *name, int lx = 1, int ly = 0, int lz = 0) = 0

virtual bool read_rec(BoutReal *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) = 0

virtual bool read_rec_perp(BoutReal *var, const std::string &name, int lx = 1, int lz = 0) = 0

virtual bool write_rec(int *var, const char *name, int lx = 0, int ly = 0, int lz = 0) = 0

virtual bool write_rec(int *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) = 0

virtual bool write_rec(char *var, const char *name, int n = 1) = 0

virtual bool write_rec(char *var, const std::string &name, int n = 1) = 0

virtual bool write_rec(BoutReal *var, const char *name, int lx = 0, int ly = 0, int lz = 0) = 0

virtual bool write_rec(BoutReal *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) = 0

virtual bool write_rec_perp(BoutReal *var, const std::string &name, int lx = 0, int lz = 0) = 0

inline virtual void setLowPrecision()

virtual void setAttribute(const std::string &varname, const std::string &attrname, const std::string &text) = 0

Sets a string attribute

Inputs

Parameters

• varname – [in] Variable name. The variable must already exist. If varname is the empty string "" then the attribute will be added to the file instead of to a variable.

• attrname – [in] Attribute name

• text – [in] A string attribute to attach to the variable

virtual void setAttribute(const std::string &varname, const std::string &attrname, int value) = 0

Sets an integer attribute

Inputs

Appendix K. API reference
Parameters

- **varname** – [in] Variable name. The variable must already exist. If varname is the empty string """, then the attribute will be added to the file instead of to a variable.

- **attrname** – [in] Attribute name

- **value** – [in] An int attribute to attach to the variable

```cpp
virtual void setAttribute(const std::string &varname, const std::string &attrname, BoutReal value) = 0
```

Sets a BoutReal attribute

**Inputs**

**Parameters**

- **varname** – [in] Variable name. The variable must already exist. If varname is the empty string """, then the attribute will be added to the file instead of to a variable.

- **attrname** – [in] Attribute name

- **value** – [in] A BoutReal attribute to attach to the variable

```cpp
virtual bool getAttribute(const std::string &varname, const std::string &attrname, std::string &text) = 0
```

Gets a string attribute

**Inputs**

**Returns**

- **text** A string attribute of the variable

```cpp
virtual bool getAttribute(const std::string &varname, const std::string &attrname, int &value) = 0
```

Gets an integer attribute

**Inputs**

**Returns**

- **value** An int attribute of the variable

```cpp
virtual bool getAttribute(const std::string &varname, const std::string &attrname, BoutReal &value) = 0
```

Gets a BoutReal attribute

**Inputs**

**Returns**

- **value** A BoutReal attribute of the variable

```cpp
```

K.2. File list 365
• attrname – [in] Attribute name
  
  void writeFieldAttributes(const std::string &name, const Field &f, bool shiftOutput = false)
  Write out the meta-data of a field as attributes of the variable.

  void writeFieldAttributes(const std::string &name, const FieldPerp &f, bool shiftOutput)
  Overload for FieldPerp so we can also write ‘yindex’.

  void readFieldAttributes(const std::string &name, Field &f)
  Read the attributes of a field.

  void readFieldAttributes(const std::string &name, FieldPerp &f)
  Overload for FieldPerp so we can also read ‘yindex’.

Protected Attributes

Mesh *mesh

K.2.45 File dcomplex.hxx

Complex number class definition.

Changelog

2015-03-09 Ben Dudson bd512@york.ac.uk o Removed, redefined in terms of std::complex

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

Typedefs

using dcomplex = std::complex<BoutReal>
Functions

\texttt{const dcomplex Im (0, 1)}

struct \texttt{fcmplx}
\begin{verbatim}
#include <dcomplex.hxx> \end{verbatim} Complex type for passing data to/from FORTRAN.

Public Members

\texttt{BoutReal r}
\texttt{BoutReal i}

K.2.46 File deprecated.hxx

Defines

\texttt{DEPRECATED(func)}
Mark functions for future removal
On gcc, expands to
\begin{verbatim}
func __attribute__((deprecated))
\end{verbatim}

Example

\begin{verbatim}
class SomeClass {
  public:
    DEPRECATED(int someFunction(const string &input));
}
\end{verbatim}

K.2.47 File deriv_store.hxx

Definition of derivative methods storage class
Copyright 2018 D.Dickinson, P.Hill, B.Dudson
Contact: Ben Dudson, bd512@york.ac.uk
This file is part of BOUT++.
BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.
You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see \url{http://www.gnu.org/licenses/}. 
template<typename FieldType>

struct DerivativeStore
#include <deriv_store.hxx> Here we have a templated singleton that is used to store DerivativeFunctions for all types of derivatives. It is templated on the FieldType (2D or 3D) as the function interfaces also depend on this. It provides public routines for registering and fetching derivative methods defined by a string key (e.g. “C2”) a DIRECTION (e.g. DIRECTION::X) and a STAGGER (e.g. STAGGER::None). There is one routine for each class of derivative (standard, standard2nd, standard4th, upwind and flux).

Public Types

using standardFunc = std::function<void(const FieldType&, FieldType&, const std::string&)>  
using flowFunc = std::function<void(const FieldType& , const FieldType& , FieldType& , const std::string& )>  
using upwindFunc = flowFunc 
using fluxFunc = flowFunc 
using storageType = std::unordered_map<K, V>

Public Functions

DerivativeStore(const DerivativeStore &junk) = delete

inline bool isEmpty() const  
  Report if store has any registered methods.

inline bool isEmpty(std::size_t key) const  
  Report if store has any registered methods for specific type determined by key.

inline bool isEmpty(DERIV derivType, DIRECTION direction, STAGGER stagger = STAGGER::None) const  
  Report if store has any registered methods for specific type.

inline std::set<std::string> getAvailableMethods(DERIV derivType, DIRECTION direction, STAGGER stagger = STAGGER::None) const  
  Returns a vector of all registered method names for the specified derivative type, direction and stagger.

inline void listAvailableMethods(DERIV derivType, DIRECTION direction, STAGGER stagger = STAGGER::None) const  
  Outputs a list of all registered method names for the specified derivative type, direction and stagger.

inline void registerDerivative(standardFunc func, DERIV derivType, DIRECTION direction, STAGGER stagger, std::string methodName)  
  Register a function with standardFunc interface. Which map is used depends on the derivType input.

inline void registerDerivative(upwindFunc func, DERIV derivType, DIRECTION direction, STAGGER stagger, std::string methodName)  
  Register a function with upwindFunc/fluxFunc interface. Which map is used depends on the derivType input.

template<typename Direction, typename Stagger, typename Method>
inline void registerDerivative(standardFunc func, Direction direction, Stagger stagger, Method method)  
  Templated versions of the above registration routines.

template<typename Direction, typename Stagger, typename Method>
inline void registerDerivative(upwindFunc func, Direction direction, Stagger stagger, Method method)
inline `standardFunc` `getStandardDerivative`(`std::string` name, `DIRECTION` direction, `STAGGER` stagger = `STAGGER::None`, `DERIV` derivType = `DERIV::Standard`) const

Routines to return a specific differential operator. Note we have to have a separate routine for different methods as they have different return types. As such we choose to use a different name for each of the method-classes so everything is consistently treated

inline `standardFunc` `getStandard2ndDerivative`(`std::string` name, `DIRECTION` direction, `STAGGER` stagger = `STAGGER::None`) const

inline `standardFunc` `getStandard4thDerivative`(`std::string` name, `DIRECTION` direction, `STAGGER` stagger = `STAGGER::None`) const

inline `flowFunc` `getFlowDerivative`(`std::string` name, `DIRECTION` direction, `STAGGER` stagger = `STAGGER::None`, `DERIV` derivType = `DERIV::Upwind`) const

inline `upwindFunc` `getUpwindDerivative`(`std::string` name, `DIRECTION` direction, `STAGGER` stagger = `STAGGER::None`) const

inline `fluxFunc` `getFluxDerivative`(`std::string` name, `DIRECTION` direction, `STAGGER` stagger = `STAGGER::None`) const

inline void `initialise`(`Options` *options)

inline void `forceDefaultMethod`(`std::string` methodName, `DERIV` deriv, `DIRECTION` direction, `STAGGER` stagger = `STAGGER::None`)

Provide a method to override/force a specific default method.

inline void `clear`()

Empty all member storage.

inline void `reset`()

Reset to initial state.

**Public Static Functions**

static inline `DerivativeStore` &`getInstance`()

**Private Functions**

inline `DerivativeStore`()

inline void `setDefaults`()

inline `std::string` `getMethodName`(`std::string` name, `DIRECTION` direction, `STAGGER` stagger = `STAGGER::None`) const

inline `std::string` `nameLookup`(`const std::string` name, `const std::string` defaultName) const
inline `std::size_t getKey(DIRECTION direction, STAGGER stagger, std::string key) const`
Provides a routine to produce a unique key given information about the specific type required. This is templated so requires compile-time information. Need to also supply a non-templated version to account for run-time choices Note: We could include the derivType in the key this would allow us to store all methods with the same function interface in the same map, which might be nice.

```
template<typename Direction, typename Stagger, typename Method>
inline std::size_t getKey() const
```
Provides a routine to produce a unique key given information about the specific type required. This is templated so requires compile-time information. Makes use of a non-templated version that can be used to account for run-time choices

**Private Members**

```
storageType<std::size_t, standardFunc> standard
storageType<std::size_t, standardFunc> standardSecond
storageType<std::size_t, standardFunc> standardFourth
storageType<std::size_t, upwindFunc> upwind
storageType<std::size_t, fluxFunc> flux
storageType<std::size_t, std::set<std::string>> registeredMethods
storageType<std::size_t, std::string> defaultMethods
```
The following stores what actual method to use when DIFF_DEFAULT is passed. The key is determined using the getKey routine here, where the name we pass is determined by the type of method (standard, upwind etc.). Note for now we’ll always use STAGGER::None as we currently assume the default method is independent of staggering it might be useful to relax this assumption!

### K.2.48 File derivs.cxx

**Functions**

**Field3D DDX(const Field3D &f, CELL_LOC outloc, const std::string &method, const std::string &region)**
Calculate first partial derivative in X
\[ \frac{\partial}{\partial x} \]

**Parameters**

- `f` – [in] The field to be differentiated
- `outloc` – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect. If not given, defaults to CELL_DEFAULT
- `method` – [in] Differencing method to use. This overrides the default. If not given, defaults to DIFF_DEFAULT
- `region` – [in] What region is expected to be calculated. If not given, defaults to RGN_NOBNDRY

**Field2D DDX(const Field2D &f, CELL_LOC outloc, const std::string &method, const std::string &region)**
Calculate first partial derivative in X
\[ \frac{\partial}{\partial x} \]

**Parameters**
• \( f \) – [in] The field to be differentiated

• \( \text{outloc} \) – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• \( \text{method} \) – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• \( \text{region} \) – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

\( \text{Field3D DDY} \) (\text{const Field3D &f, CELL_LOC outloc, const std::string &method, const std::string &region})

Calculate first partial derivative in \( Y \)

\( \partial/\partial y \)

Parameters

• \( f \) – [in] The field to be differentiated

• \( \text{outloc} \) – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• \( \text{method} \) – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• \( \text{region} \) – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

\( \text{Field2D DDY} \) (\text{const Field2D &f, CELL_LOC outloc, const std::string &method, const std::string &region})

Calculate first partial derivative in \( Y \)

\( \partial/\partial y \)

Parameters

• \( f \) – [in] The field to be differentiated

• \( \text{outloc} \) – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• \( \text{method} \) – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• \( \text{region} \) – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

\( \text{Field3D DDZ} \) (\text{const Field3D &f, CELL_LOC outloc, const std::string &method, const std::string &region})

Calculate first partial derivative in \( Z \)

\( \partial/\partial z \)

Parameters

• \( f \) – [in] The field to be differentiated

• \( \text{outloc} \) – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• \( \text{method} \) – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• \( \text{region} \) – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY
Field2D DDZ(const Field2D &f, CELL_LOC outloc, const std::string &method, const std::string &region)
Calculate first partial derivative in Z
\[ \frac{\partial}{\partial z} \]
Parameters
- \( f \) – [in] The field to be differentiated
- \( outloc \) – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- \( method \) – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( region \) – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Vector3D DDZ(const Vector3D &v, CELL_LOC outloc, const std::string &method, const std::string &region)
Calculate first partial derivative in Z
\[ \frac{\partial}{\partial z} \]
Parameters
- \( f \) – [in] The field to be differentiated
- \( outloc \) – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- \( method \) – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( region \) – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Vector2D DDZ(const Vector2D &v, CELL_LOC outloc, const std::string &method, const std::string &region)
Calculate first partial derivative in Z
\[ \frac{\partial}{\partial z} \]
Parameters
- \( f \) – [in] The field to be differentiated
- \( outloc \) – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- \( method \) – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( region \) – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field3D D2DX2(const Field3D &f, CELL_LOC outloc, const std::string &method, const std::string &region)
Calculate second partial derivative in X
\[ \frac{\partial^2}{\partial x^2} \]
Parameters
- \( f \) – [in] The field to be differentiated
- \( outloc \) – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to **DIFF_DEFAULT**

• **region** – [in] What region is expected to be calculated If not given, defaults to **RGN_NOBNDRY**

**Field2D D2DX2** (const **Field2D f**, **CELL_LOC outloc**, const **std::string &method**, const **std::string &region**)  
Calculate second partial derivative in X  
\( \frac{\partial^2}{\partial x^2} \)

**Parameters**

• **f** – [in] The field to be differentiated  

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to **CELL_DEFAULT**  

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to **DIFF_DEFAULT**  

• **region** – [in] What region is expected to be calculated If not given, defaults to **RGN_NOBNDRY**

**Field3D D2DY2** (const **Field3D f**, **CELL_LOC outloc**, const **std::string &method**, const **std::string &region**)  
Calculate second partial derivative in Y  
\( \frac{\partial^2}{\partial y^2} \)

**Parameters**

• **f** – [in] The field to be differentiated  

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to **CELL_DEFAULT**  

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to **DIFF_DEFAULT**  

• **region** – [in] What region is expected to be calculated If not given, defaults to **RGN_NOBNDRY**

**Field2D D2DY2** (const **Field2D f**, **CELL_LOC outloc**, const **std::string &method**, const **std::string &region**)  
Calculate second partial derivative in Y  
\( \frac{\partial^2}{\partial y^2} \)

**Parameters**

• **f** – [in] The field to be differentiated  

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to **CELL_DEFAULT**  

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to **DIFF_DEFAULT**  

• **region** – [in] What region is expected to be calculated If not given, defaults to **RGN_NOBNDRY**

**Field3D D2DZ2** (const **Field3D f**, **CELL_LOC outloc**, const **std::string &method**, const **std::string &region**)  
Calculate second partial derivative in Z  
\( \frac{\partial^2}{\partial z^2} \)

**Parameters**
• **f** – [in] The field to be differentiated

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field2D **D2DZ2** (const Field2D &f, CELL_LOC outloc, const std::string &method, const std::string &region)

Calculate second partial derivative in Z

\( \partial^2 / \partial z^2 \)

**Parameters**

• **f** – [in] The field to be differentiated

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field3D **D4DX4** (const Field3D &f, CELL_LOC outloc, const std::string &method, const std::string &region)

Calculate forth partial derivative in X

\( \partial^4 / \partial x^4 \)

**Parameters**

• **f** – [in] The field to be differentiated

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field2D **D4DX4** (const Field2D &f, CELL_LOC outloc, const std::string &method, const std::string &region)

Calculate forth partial derivative in X

\( \partial^4 / \partial x^4 \)

**Parameters**

• **f** – [in] The field to be differentiated

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY
Field3D D4DY4(const Field3D &f, CELL_LOC outloc, const std::string &method, const std::string &region)
Calculate forth partial derivative in Y
\[ \frac{\partial^4}{\partial y^4} \]

Parameters
- f – [in] The field to be differentiated
- outloc – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- method – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- region – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field2D D4DY4(const Field2D &f, CELL_LOC outloc, const std::string &method, const std::string &region)
Calculate forth partial derivative in Y
\[ \frac{\partial^4}{\partial y^4} \]

Parameters
- f – [in] The field to be differentiated
- outloc – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- method – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- region – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field3D D4DZ4(const Field3D &f, CELL_LOC outloc, const std::string &method, const std::string &region)
Calculate forth partial derivative in Z
\[ \frac{\partial^4}{\partial z^4} \]

Parameters
- f – [in] The field to be differentiated
- outloc – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- method – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- region – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field2D D4DZ4(const Field2D &f, CELL_LOC outloc, const std::string &method, const std::string &region)
Calculate forth partial derivative in Z
\[ \frac{\partial^4}{\partial z^4} \]

Parameters
- f – [in] The field to be differentiated
- outloc – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

*Field2D D2DXDY*(const *Field2D* &f, *CELL_LOC* outloc, const *std::string* &method, const *std::string* &region, const *std::string* &dfdy_boundary_condition)

Mixed derivative in X and Y

This first takes derivatives in Y, then in X.

** Communicates and applies boundary in X.

*Field3D D2DXDY*(const *Field3D* &f, *CELL_LOC* outloc, const *std::string* &method, const *std::string* &region, const *std::string* &dfdy_boundary_condition)

Mixed derivative in X and Y

This first takes derivatives in Y, then in X.

** Communicates and applies boundary in X.

*Field2D D2DXDZ*(const *Field2D* &f, *CELL_LOC* outloc, const *std::string* &method, const *std::string* &region)

Calculate mixed partial derivative in x and z

\[ \frac{\partial^2}{\partial x \partial z} \]

**Parameters**

• **f** – [in] The field to be differentiated

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

*Field3D D2DXDZ*(const *Field3D* &f, *CELL_LOC* outloc, const *std::string* &method, const *std::string* &region)

X-Z mixed derivative.

Calculate mixed partial derivative in x and z

\[ \frac{\partial^2}{\partial x \partial z} \]

**Parameters**

• **f** – [in] The field to be differentiated

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

*Field2D D2DYDZ*(const *Field2D* &f, *CELL_LOC* outloc, const *std::string* &method, const *std::string* &region)

Calculate mixed partial derivative in y and z

\[ \frac{\partial^2}{\partial y \partial z} \]

**Parameters**
• \( f \) – [in] The field to be differentiated

• \( \text{outloc} \) – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• \( \text{method} \) – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• \( \text{region} \) – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

\( \text{Field3D D2DYDZ} \) (const Field3D &f, CELL_LOC outloc, MAYBE_UNUSED(const std::string &method), const std::string &region)

\( \text{Field2D VDDX} \) (const Field2D &v, const Field2D &f, CELL_LOC outloc, const std::string &method, const std::string &region)

Special case where both arguments are 2D. Output location ignored for now.

For terms of form \( v \cdot \nabla f \)

\( v \cdot \frac{\partial f}{\partial x} \)

Parameters

• \( v \) – [in] The velocity field

• \( f \) – [in] The field of the advected quantity

• \( \text{outloc} \) – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• \( \text{method} \) – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• \( \text{region} \) – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

\( \text{Field3D VDDX} \) (const Field3D &v, const Field3D &f, CELL_LOC outloc, const std::string &method, const std::string &region)

General version for 2 or 3-D objects.

For terms of form \( v \cdot \nabla f \)

\( v \cdot \frac{\partial f}{\partial x} \)

Parameters

• \( v \) – [in] The velocity field

• \( f \) – [in] The field of the advected quantity

• \( \text{outloc} \) – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• \( \text{method} \) – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• \( \text{region} \) – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

\( \text{Field2D VDDY} \) (const Field2D &v, const Field2D &f, CELL_LOC outloc, const std::string &method, const std::string &region)

For terms of form \( v \cdot \nabla f \)

\( v \cdot \frac{\partial f}{\partial y} \)
Parameters

- `v` – [in] The velocity field
- `f` – [in] The field of the advected quantity
- `outloc` – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect. If not given, defaults to `CELL_DEFAULT`
- `method` – [in] Differencing method to use. This overrides the default. If not given, defaults to `DIFF_DEFAULT`
- `region` – [in] What region is expected to be calculated. If not given, defaults to `RGN_NOBNDRY`

Field3D `VDDY` (const `Field3D` &v, const `Field3D` &f, `CELL_LOC` outloc, const `std::string` &method, const `std::string` &region)

For terms of form \( v \cdot \nabla f \)

\[ v \cdot \frac{\partial f}{\partial y} \]

Parameters

- `v` – [in] The velocity field
- `f` – [in] The field of the advected quantity
- `outloc` – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect. If not given, defaults to `CELL_DEFAULT`
- `method` – [in] Differencing method to use. This overrides the default. If not given, defaults to `DIFF_DEFAULT`
- `region` – [in] What region is expected to be calculated. If not given, defaults to `RGN_NOBNDRY`

Field2D `VDDZ` (const `Field2D` &v, const `Field2D` &f, `CELL_LOC` outloc, const `std::string` &method, const `std::string` &region)

For terms of form \( v \cdot \nabla f \)

\[ v \cdot \frac{\partial f}{\partial z} \]

Parameters

- `v` – [in] The velocity field
- `f` – [in] The field of the advected quantity
- `outloc` – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect. If not given, defaults to `CELL_DEFAULT`
- `method` – [in] Differencing method to use. This overrides the default. If not given, defaults to `DIFF_DEFAULT`
- `region` – [in] What region is expected to be calculated. If not given, defaults to `RGN_NOBNDRY`

Field2D `VDDZ` (const `Field3D` &v, const `Field2D` &f, `CELL_LOC` outloc, const `std::string` &method, const `std::string` &region)

For terms of form \( v \cdot \nabla f \)

\[ v \cdot \frac{\partial f}{\partial z} \]

Parameters

- `v` – [in] The velocity field
• **f** – [in] The field of the advected quantity

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field3D **VDDZ**(const Field3D &v, const Field3D &f, CELL_LOC outloc, const std::string &method, const std::string &region)

For terms of form $v \cdot \frac{\partial f}{\partial z}$

\[
v \cdot \partial f / \partial z\]

**Parameters**

• **v** – [in] The velocity field

• **f** – [in] The field of the advected quantity

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field2D **FDDX**(const Field2D &v, const Field2D &f, CELL_LOC outloc, const std::string &method, const std::string &region)

for terms of form $\frac{\partial (vf)}{\partial x}$

\[
\frac{\partial (vf)}{\partial x}\]

**Parameters**

• **v** – [in] The velocity field

• **f** – [in] The field of the advected quantity

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field3D **FDDX**(const Field3D &v, const Field3D &f, CELL_LOC outloc, const std::string &method, const std::string &region)

for terms of form $\frac{\partial (vf)}{\partial x}$

\[
\frac{\partial (vf)}{\partial x}\]

**Parameters**

• **v** – [in] The velocity field

• **f** – [in] The field of the advected quantity
• `outloc` – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to `CELL_DEFAULT`

• `method` – [in] Differencing method to use. This overrides the default If not given, defaults to `DIFF_DEFAULT`

• `region` – [in] What region is expected to be calculated If not given, defaults to `RGN_NOBNDRY`

`Field2D FDDY` (const `Field2D` &v, const `Field2D` &f, `CELL_LOC` outloc, const `std::string` &method, const `std::string` &region)

for terms of form \( \text{div}(v * f) \)

\( \frac{\partial(vf)}{\partial y} \)

Parameters

• `v` – [in] The velocity field

• `f` – [in] The field of the advected quantity

• `outloc` – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to `CELL_DEFAULT`

• `method` – [in] Differencing method to use. This overrides the default If not given, defaults to `DIFF_DEFAULT`

• `region` – [in] What region is expected to be calculated If not given, defaults to `RGN_NOBNDRY`

`Field3D FDDY` (const `Field3D` &v, const `Field3D` &f, `CELL_LOC` outloc, const `std::string` &method, const `std::string` &region)

for terms of form \( \text{div}(v * f) \)

\( \frac{\partial(vf)}{\partial y} \)

Parameters

• `v` – [in] The velocity field

• `f` – [in] The field of the advected quantity

• `outloc` – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to `CELL_DEFAULT`

• `method` – [in] Differencing method to use. This overrides the default If not given, defaults to `DIFF_DEFAULT`

• `region` – [in] What region is expected to be calculated If not given, defaults to `RGN_NOBNDRY`

`Field2D FDDZ` (const `Field2D` &v, const `Field2D` &f, `CELL_LOC` outloc, const `std::string` &method, const `std::string` &region)

for terms of form \( \text{div}(v * f) \)

\( \frac{\partial(vf)}{\partial z} \)

Parameters

• `v` – [in] The velocity field

• `f` – [in] The field of the advected quantity

• `outloc` – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to `CELL_DEFAULT`
• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field3D **FDDZ**(const Field3D &v, const Field3D &f, CELL_LOC outloc, const std::string &method, const std::string &region)

for terms of form \(\text{div}(v \cdot f)\)

\(\frac{\partial(vf)}{\partial z}\)

**Parameters**

• **v** – [in] The velocity field

• **f** – [in] The field of the advected quantity

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

**K.2.49 File derivs.hxx**

Basic differential functions

Copyright 2010,2017 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu, D. Schwörer

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

**Defines**

**DERIV_FUNC_REGION_ENUM_TO_STRING**(func, T)

**VDERIV_FUNC_REGION_ENUM_TO_STRING**(func, T, T1, T2)
Functions

Field3D DDX(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
Calculate first partial derivative in X
\( \frac{\partial}{\partial x} \)

Parameters

- **f** – [in] The field to be differentiated
- **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline Field3D DDX(const Field3D &f, CELL_LOC outloc, const std::string &method, REGION region)

inline Field3D DDX(const Field3D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field2D DDX(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
Calculate first partial derivative in X
\( \frac{\partial}{\partial x} \)

Parameters

- **f** – [in] The field to be differentiated
- **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline Field2D DDX(const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)

inline Field2D DDX(const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field3D DDY(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
Calculate first partial derivative in Y
\( \frac{\partial}{\partial y} \)

Parameters

- **f** – [in] The field to be differentiated
• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline `Field3D DDY(const Field3D &f, CELL_LOC outloc, const std::string &method, REGION region)`

inline `Field3D DDY(const Field3D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)`

`Field2D DDY(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")`

Calculate first partial derivative in Y

\[ \frac{\partial}{\partial y} \]

**Parameters**

• **f** – [in] The field to be differentiated

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline `Field2D DDY(const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)`

inline `Field2D DDY(const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)`

`Field3D DDZ(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")`

Calculate first partial derivative in Z

\[ \frac{\partial}{\partial z} \]

**Parameters**

• **f** – [in] The field to be differentiated

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline `Field3D DDZ(const Field3D &f, CELL_LOC outloc, const std::string &method, REGION region)`
inline Field3D DDZ(const Field3D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field2D DDZ(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
Calculate first partial derivative in Z
\( \partial / \partial z \)

Parameters

- **f** – [in] The field to be differentiated
- **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect. If not given, defaults to CELL_DEFAULT
- **method** – [in] Differencing method to use. This overrides the default. If not given, defaults to DIFF_DEFAULT
- **region** – [in] What region is expected to be calculated. If not given, defaults to RGN_NOBNDRY

inline Field2D DDZ(const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)

inline Field2D DDZ(const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Vector3D DDZ(const Vector3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
Calculate first partial derivative in Z
\( \partial / \partial z \)

Parameters

- **f** – [in] The field to be differentiated
- **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect. If not given, defaults to CELL_DEFAULT
- **method** – [in] Differencing method to use. This overrides the default. If not given, defaults to DIFF_DEFAULT
- **region** – [in] What region is expected to be calculated. If not given, defaults to RGN_NOBNDRY

inline Vector3D DDZ(const Vector3D &f, CELL_LOC outloc, const std::string &method, REGION region)

inline Vector3D DDZ(const Vector3D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Vector2D DDZ(const Vector2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
Calculate first partial derivative in Z
\( \partial / \partial z \)

Parameters

- **f** – [in] The field to be differentiated
• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

```cpp
inline Vector2D DDZ(const Vector2D &f, CELL_LOC outloc, const std::string &method, REGION region)
```

```cpp
inline Vector2D DDZ(const Vector2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)
```

```cpp
Field3D D2DX2(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
```

Calculate second partial derivative in X

\[ \frac{\partial^2}{\partial x^2} \]

**Parameters**

• **f** – [in] The field to be differentiated

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

```cpp
inline Field3D D2DX2(const Field3D &f, CELL_LOC outloc, const std::string &method, REGION region)
```

```cpp
inline Field3D D2DX2(const Field3D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)
```

```cpp
Field2D D2DX2(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
```

Calculate second partial derivative in X

\[ \frac{\partial^2}{\partial x^2} \]

**Parameters**

• **f** – [in] The field to be differentiated

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

```cpp
inline Field2D D2DX2(const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)
```
inline Field2D D2DX2 (const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field3D D2DY2 (const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
Calculate second partial derivative in Y
\( \partial^2 / \partial y^2 \)

Parameters
- **f** – [in] The field to be differentiated
- **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline Field3D D2DY2 (const Field3D &f, CELL_LOC outloc, const std::string &method, REGION region)

inline Field3D D2DY2 (const Field3D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field2D D2DY2 (const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
Calculate second partial derivative in Y
\( \partial^2 / \partial y^2 \)

Parameters
- **f** – [in] The field to be differentiated
- **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline Field2D D2DY2 (const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)

inline Field2D D2DY2 (const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field3D D2DZ2 (const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
Calculate second partial derivative in Z
\( \partial^2 / \partial z^2 \)

Parameters
- **f** – [in] The field to be differentiated
In BOUT++ Documentation, Release 4.4.0, the following functions are described:

### Field3D D2DZ2

```cpp
inline Field3D D2DZ2(const Field3D &f, CELL_LOC outloc, const std::string &method,
                     REGION region)
```

Calculate second partial derivative in Z

\[ \frac{\partial^2}{\partial z^2} \]

**Parameters**

- **f** – [in] The field to be differentiated
- **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

### Field2D D2DZ2

```cpp
inline Field2D D2DZ2(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method =
                     "DEFAULT", const std::string &region = "RGN_NOBNDRY")
```

Calculate second partial derivative in Z

\[ \frac{\partial^2}{\partial z^2} \]

### Field3D D4DX4

```cpp
inline Field3D D4DX4(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method =
                     "DEFAULT", const std::string &region = "RGN_NOBNDRY")
```

Calculate forth partial derivative in X

\[ \frac{\partial^4}{\partial x^4} \]

**Parameters**

- **f** – [in] The field to be differentiated
- **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY
inline Field3D D4DX4 (const Field3D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field2D D4DX4 (const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
Calculate forth partial derivative in X
\( \frac{\partial^4}{\partial x^4} \)

Parameters
• f – [in] The field to be differentiated
• outloc – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
• method – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
• region – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline Field2D D4DX4 (const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)

inline Field2D D4DX4 (const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field3D D4DY4 (const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
Calculate forth partial derivative in Y
\( \frac{\partial^4}{\partial y^4} \)

Parameters
• f – [in] The field to be differentiated
• outloc – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
• method – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
• region – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline Field3D D4DY4 (const Field3D &f, CELL_LOC outloc, const std::string &method, REGION region)

inline Field3D D4DY4 (const Field3D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field2D D4DY4 (const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
Calculate forth partial derivative in Y
\( \frac{\partial^4}{\partial y^4} \)

Parameters
• f – [in] The field to be differentiated
• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline `Field2D D4DY4` (const `Field2D` &f, `CELL_LOC` outloc, const `std::string` &method, `REGION` region)

```
inline Field2D D4DY4 (const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)
```

```Field3D D4DZ4```
(const `Field3D` &f, `CELLLOC` outloc = `CELL_DEFAULT`, const `std::string` &method = "DEFAULT", const `std::string` &region = "RGN_NOBNDRY")
Calculate forth partial derivative in Z
\[ \partial^4 / \partial z^4 \]

**Parameters**

• **f** – [in] The field to be differentiated

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline `Field3D D4DZ4` (const `Field3D` &f, `CELL_LOC` outloc, const `std::string` &method, `REGION` region)

```
inline Field3D D4DZ4 (const Field3D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)
```

```
Field2D D4DZ4 (const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
```

Calculate forth partial derivative in Z
\[ \partial^4 / \partial z^4 \]

**Parameters**

• **f** – [in] The field to be differentiated

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline `Field2D D4DZ4` (const `Field2D` &f, `CELL_LOC` outloc, const `std::string` &method, `REGION` region)
inline Field2D D4DZ4(const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field3D VDDX(const Field3D &v, const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
General version for 2 or 3-D objects.

For terms of form \( v \cdot \nabla f \)

\[ v \cdot \frac{\partial f}{\partial x} \]

Parameters

- \( v \) – [in] The velocity field
- \( f \) – [in] The field of the advected quantity
- \( outloc \) – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- \( method \) – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( region \) – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline Field3D VDDX(const Field3D &v, const Field3D &f, CELL_LOC outloc, const std::string &method, REGION region)

inline Field3D VDDX(const Field3D &v, const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field2D VDDX(const Field2D &v, const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
Special case where both arguments are 2D. Output location ignored for now.

For terms of form \( v \cdot \nabla f \)

\[ v \cdot \frac{\partial f}{\partial x} \]

Parameters

- \( v \) – [in] The velocity field
- \( f \) – [in] The field of the advected quantity
- \( outloc \) – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- \( method \) – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( region \) – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline Field2D VDDX(const Field2D &v, const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)

inline Field2D VDDX(const Field2D &v, const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)
Field3D VDDY(const Field3D &v, const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

For terms of form v * grad(f)

\[ \mathbf{v} \cdot \nabla f \]

Parameters

- **v** – [in] The velocity field
- **f** – [in] The field of the advected quantity
- **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect. If not given, defaults to CELL_DEFAULT
- **method** – [in] Differencing method to use. This overrides the default. If not given, defaults to DIFF_DEFAULT
- **region** – [in] What region is expected to be calculated. If not given, defaults to RGN_NOBNDRY

inline Field3D VDDY(const Field3D &v, const Field3D &f, CELL_LOC outloc, const std::string &method, REGION region)

Field2D VDDY(const Field2D &v, const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

For terms of form v * grad(f)

\[ \mathbf{v} \cdot \nabla f \]

Parameters

- **v** – [in] The velocity field
- **f** – [in] The field of the advected quantity
- **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect. If not given, defaults to CELL_DEFAULT
- **method** – [in] Differencing method to use. This overrides the default. If not given, defaults to DIFF_DEFAULT
- **region** – [in] What region is expected to be calculated. If not given, defaults to RGN_NOBNDRY

inline Field2D VDDY(const Field2D &v, const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)

Field3D VDDZ(const Field3D &v, const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

For terms of form v * grad(f)

\[ \mathbf{v} \cdot \nabla z \]

Parameters
• **v** – [in] The velocity field

• **f** – [in] The field of the advected quantity

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline `Field3D VDDZ(const Field3D &v, const Field3D &f, CELL_LOC outloc, const std::string &method, REGION region)`

inline `Field3D VDDZ(const Field3D &v, const Field3D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)`

For terms of form $v \cdot \nabla f$

\[v \cdot \frac{\partial f}{\partial z}\]

**Parameters**

• **v** – [in] The velocity field

• **f** – [in] The field of the advected quantity

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline `Field2D VDDZ(const Field2D &v, const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")`

For terms of form $v \cdot \nabla f$

\[v \cdot \frac{\partial f}{\partial z}\]

**Parameters**

• **v** – [in] The velocity field

• **f** – [in] The field of the advected quantity

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

```cpp
inline Field2D VDDZ(const Field3D &v, const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)
```

```cpp
inline Field3D FDDX(const Field3D &v, const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
```

```cpp
for terms of form div(v * f)
\[ \frac{\partial(vf)}{\partial x} \]
```

**Parameters**

• **v** – [in] The velocity field

• **f** – [in] The field of the advected quantity

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

```cpp
inline Field3D FDDX(const Field3D &v, const Field3D &f, CELL_LOC outloc, const std::string &method, REGION region)
```

```cpp
inline Field3D FDDX(const Field3D &v, const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
```

```cpp
for terms of form div(v * f)
\[ \frac{\partial(vf)}{\partial x} \]
```

**Parameters**

• **v** – [in] The velocity field

• **f** – [in] The field of the advected quantity

• **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT

• **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

```cpp
Field2D FDDX(const Field2D &v, const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
```
inline Field2D FDDX(const Field2D &v, const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)

inline Field2D FDDX(const Field2D &v, const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field3D FDDY(const Field3D &v, const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

for terms of form div(v * f)

\[ \frac{\partial (vf)}{\partial y} \]

Parameters

- v – [in] The velocity field
- f – [in] The field of the advected quantity
- outloc – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect. If not given, defaults to CELL_DEFAULT
- method – [in] Differencing method to use. This overrides the default. If not given, defaults to DIFF_DEFAULT
- region – [in] What region is expected to be calculated. If not given, defaults to RGN_NOBNDRY

inline Field3D FDDY(const Field3D &v, const Field3D &f, CELL_LOC outloc, const std::string &method, REGION region)

inline Field3D FDDY(const Field3D &v, const Field3D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field2D FDDY(const Field2D &v, const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

for terms of form div(v * f)

\[ \frac{\partial (vf)}{\partial y} \]

Parameters

- v – [in] The velocity field
- f – [in] The field of the advected quantity
- outloc – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect. If not given, defaults to CELL_DEFAULT
- method – [in] Differencing method to use. This overrides the default. If not given, defaults to DIFF_DEFAULT
- region – [in] What region is expected to be calculated. If not given, defaults to RGN_NOBNDRY

inline Field2D FDDY(const Field2D &v, const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)
inline Field2D FDDY(const Field2D &v, const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field3D FDDZ(const Field3D &v, const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
for terms of form div(v * f)
\[ \partial(vf)/\partial z \]

Parameters
- v – [in] The velocity field
- f – [in] The field of the advected quantity
- outloc – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- method – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- region – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline Field3D FDDZ(const Field3D &v, const Field3D &f, CELL_LOC outloc, const std::string &method, REGION region)

inline Field3D FDDZ(const Field3D &v, const Field3D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field2D FDDZ(const Field2D &v, const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
for terms of form div(v * f)
\[ \partial(vf)/\partial z \]

Parameters
- v – [in] The velocity field
- f – [in] The field of the advected quantity
- outloc – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- method – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- region – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline Field2D FDDZ(const Field2D &v, const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)

inline Field2D FDDZ(const Field2D &v, const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)
Field3D D2DXY(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY", const std::string &dfdy_boundary_condition = "free_o3")

Calculate mixed partial derivative in x and y
\[ \frac{\partial^2}{\partial x \partial y} \]

Mixed derivative in X and Y
This first takes derivatives in Y, then in X.
** Communicates and applies boundary in X.

Parameters
- **f** – [in] The field to be differentiated
- **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY
- **dfdy_boundary_condition** – [in] Boundary condition to use to set the guard cells of df/dy, before calculating the x-derivative.

inline Field3D D2DXY(const Field3D &f, CELL_LOC outloc, const std::string &method, REGION region, const std::string &dfdy_boundary_condition = "free_o3")

inline Field3D D2DXY(const Field3D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY, const std::string &dfdy_boundary_condition = "free_o3")

Field2D D2DXY(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY", const std::string &dfdy_boundary_condition = "free_o3")

Calculate mixed partial derivative in x and y
\[ \frac{\partial^2}{\partial x \partial y} \]

Mixed derivative in X and Y
This first takes derivatives in Y, then in X.
** Communicates and applies boundary in X.

Parameters
- **f** – [in] The field to be differentiated
- **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY
• **dfdy_boundary_condition** – [in] Boundary condition to use to set the guard cells of df/dy, 
before calculating the x-derivative.

```cpp
inline Field2D D2DXDY(const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region, const 
std::string &dfdy_boundary_condition = "free_o3")
```

```cpp
inline Field2D D2DXDY(const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY, const std::string &dfdy_boundary_condition = "free_o3")
```

**Field3D D2DXDZ** (const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

X-Z mixed derivative.

Calculate mixed partial derivative in x and z

\[ \frac{\partial^2}{\partial x \partial z} \]

**Parameters**

- **f** – [in] The field to be differentiated
- **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled 
then this has no effect If not given, defaults to CELL_DEFAULT
- **method** – [in] Differencing method to use. This overrides the default If not given, defaults 
to DIFF_DEFAULT
- **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

```cpp
inline Field3D D2DXDZ(const Field3D &f, CELL_LOC outloc, const std::string &method, REGION region)
```

```cpp
inline Field3D D2DXDZ(const Field3D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)
```

**Field2D D2DXDZ** (const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = 
"DEFAULT", const std::string &region = "RGN_NOBNDRY")

Calculate mixed partial derivative in x and z

\[ \frac{\partial^2}{\partial x \partial z} \]

**Parameters**

- **f** – [in] The field to be differentiated
- **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled 
then this has no effect If not given, defaults to CELL_DEFAULT
- **method** – [in] Differencing method to use. This overrides the default If not given, defaults 
to DIFF_DEFAULT
- **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

```cpp
inline Field2D D2DXDZ(const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)
```

```cpp
inline Field2D D2DXDZ(const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)
```
Field3D D2DYDZ(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

Calculate mixed partial derivative in y and z
\[ \frac{\partial^2}{\partial y \partial z} \]

Parameters

- **f** – [in] The field to be differentiated
- **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline Field3D D2DYDZ(const Field3D &f, CELL_LOC outloc, const std::string &method, REGION region)

inline Field3D D2DYDZ(const Field3D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field2D D2DYDZ(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

Calculate mixed partial derivative in y and z
\[ \frac{\partial^2}{\partial y \partial z} \]

Parameters

- **f** – [in] The field to be differentiated
- **outloc** – [in] The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to CELL_DEFAULT
- **method** – [in] Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- **region** – [in] What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

inline Field2D D2DYDZ(const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)

inline Field2D D2DYDZ(const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)
**K.2.50 File difops.cxx**

**Functions**

```cpp
const Field2D Grad_par(const Field2D &var, CELL_LOC outloc, const std::string &method)
  Parallel derivative (central differencing) in Y along unperturbed field
  
  Parameters
  • var – [in] The field to be differentiated
  • outloc – [in] The cell location where the output is needed (if staggered grids is enabled)
  • method – [in] The method to use. The default is set in the options.
```

```cpp
const Field2D Grad_par(const Field2D &var, const std::string &method, CELL_LOC outloc)
```

```cpp
const Field3D Grad_par(const Field3D &var, CELL_LOC outloc, const std::string &method)
```

```cpp
const Field3D Grad_par(const Field3D &var, const std::string &method, CELL_LOC outloc)
```

```cpp
const Field3D Grad_parP(const Field3D &apar, const Field3D &f)
  Derivative along perturbed magnetic field in Clebsch coordinate system
  b0 dot Grad - (1/B)b0 x Grad(apar) dot Grad
  Combines the parallel and perpendicular calculation to include grid-points at the corners.
```

```cpp
const Field2D Vpar_Grad_par(const Field2D &v, const Field2D &f, CELL_LOC outloc, const std::string &method)
  vpar times parallel derivative along unperturbed B-field (upwinding)
  
  \[ v b_0 \cdot \nabla f \]
  
  Parameters
  • v – [in] The velocity in y direction
  • f – [in] The scalar field to be differentiated
  • outloc – [in] The cell location of the output. By default this is the same as f
  • method – [in] The numerical method to use. The default is set in the options
```

```cpp
const Field2D Vpar_Grad_par(const Field2D &v, const Field2D &f, const std::string &method, CELL_LOC outloc)
```

```cpp
const Field3D Vpar_Grad_par(const Field3D &v, const Field3D &f, CELL_LOC outloc, const std::string &method)
```

```cpp
const Field3D Vpar_Grad_par(const Field3D &v, const Field3D &f, const std::string &method, CELL_LOC outloc)
```

```cpp
const Field2D Div_par(const Field2D &f, CELL_LOC outloc, const std::string &method)
  parallel divergence operator
```
\[ B \partial_t (f/B) = B \nabla \cdot (bf/B) \]

Parameters

- \( f \) – [in] The component of a vector along the magnetic field
- \( outloc \) – [in] The cell location for the result. By default the same as \( f \)
- \( method \) – [in] The numerical method to use

\[
const \ Field2D \ Div\_par(const \ Field2D &f, const \ std::string &method, CELL\_LOC \ outloc) \\
const \ Field3D \ Div\_par(const \ Field3D &f, CELL\_LOC \ outloc, const \ std::string &method) \\
const \ Field3D \ Div\_par(const \ Field3D &f, const \ std::string &method, CELL\_LOC \ outloc) \\
const \ Field3D \ Div\_par(const \ Field3D &f, const \ Field3D &v) \\
const \ Field3D \ Div\_par\_flux(const \ Field3D &v, const \ Field3D &f, CELL\_LOC \ outloc, const \ std::string &method) \\
const \ Field3D \ Div\_par\_flux(const \ Field3D &v, const \ Field3D &f, const \ std::string &method, CELL\_LOC \ outloc) \\
const \ Field2D \ Grad2\_par\_2(const \ Field2D &f, CELL\_LOC \ outloc, const \ std::string &method) \\
second parallel derivative \\
\( (b\cdot\nabla)(b\cdot\nabla) \)

Note: For parallel Laplacian use LaplacePar

Parameters

- \( f \) – [in] The field to be differentiated
- \( outloc \) – [in] The cell location of the result

\[
const \ Field3D \ Grad2\_par\_2(const \ Field3D &f, CELL\_LOC \ outloc, const \ std::string &method) \\
const \ Field2D \ Div\_par\_K\_Grad\_par(BoutReal \ kY, const \ Field2D &f, CELL\_LOC \ outloc) \\
\] Parallel divergence of diffusive flux, \( K \cdot \operatorname{Grad}\_par \)

\[ \nabla \cdot (b_0 kY (b_0 \cdot \nabla) f) \]

Parameters

- \( kY \) – [in] The diffusion coefficient
- \( f \) – [in] The field whose gradient drives a flux

\[
const \ Field3D \ Div\_par\_K\_Grad\_par(BoutReal \ kY, const \ Field3D &f, CELL\_LOC \ outloc) \\
const \ Field2D \ Div\_par\_K\_Grad\_par(const \ Field2D &kY, const \ Field2D &f, CELL\_LOC \ outloc) \\
const \ Field2D \ Div\_par\_K\_Grad\_par(const \ Field2D &kY, const \ Field2D &f, CELL\_LOC \ outloc) \\
\]
const Field3D Div_par_K_Grad_par(const Field2D &kY, const Field3D &f, CELL_LOC outloc)

const Field3D Div_par_K_Grad_par(const Field3D &kY, const Field2D &f, CELL_LOC outloc)

const Field3D Div_par_K_Grad_par(const Field3D &kY, const Field3D &f, CELL_LOC outloc)

const Field2D Delp2(const Field2D &f, CELL_LOC outloc, bool useFFT)
Perpendicular Laplacian operator
This version only includes terms in X and Z, dropping derivatives in Y. This is the inverse operation to the Laplacian inversion class.
For the full perpendicular Laplacian, use Laplace_perp

const Field3D Delp2(const Field3D &f, CELL_LOC outloc, bool useFFT)

const FieldPerp Delp2(const FieldPerp &f, CELL_LOC outloc, bool useFFT)

const Field2D Laplace_perp(const Field2D &f, CELL_LOC outloc)
Perpendicular Laplacian, keeping y derivatives

const Field3D Laplace_perp(const Field3D &f, CELL_LOC outloc)

const Field2D Laplace_par(const Field2D &f, CELL_LOC outloc)
Parallel Laplacian operator

const Field3D Laplace_par(const Field3D &f, CELL_LOC outloc)

const Field2D Laplace(const Field2D &f, CELL_LOC outloc)
Full Laplacian operator (par + perp)

const Field3D Laplace(const Field3D &f, CELL_LOC outloc)

const Field2D b0xGrad_dot_Grad(const Field2D &phi, const Field2D &A, CELL_LOC outloc)
Terms of form b0 x Grad(phi) dot Grad(A)

const Field3D b0xGrad_dot_Grad(const Field2D &phi, const Field3D &A, CELL_LOC outloc)

const Field3D b0xGrad_dot_Grad(const Field3D &p, const Field2D &A, CELL_LOC outloc)
Terms of form

b0 x \nabla \phi \cdot \nabla A

Parameters

- phi – [in] The scalar potential
- A – [in] The field being advected
- outloc – [in] The cell location where the result is defined. By default the same as A.

const Field3D b0xGrad_dot_Grad(const Field3D &phi, const Field3D &A, CELL_LOC outloc)
const Field2D bracket(const Field2D &f, const Field2D &g, BRACKET_METHOD method, CELL_LOC outloc, Solver *solver)

Compute advection operator terms, which can be cast as antisymmetric Poisson brackets

\[ [f, g] = (1/B_0) \mathbf{b}_0 \times \nabla f \cdot \nabla g \]

Parameters

- f – [in] The potential
- g – [in] The field being advected
- method – [in] The method to use
- outloc – [in] The cell location where the result is defined. Default is the same as g
- solver – [in] Pointer to the time integration solver

const Field3D bracket(const Field3D &f, const Field2D &g, BRACKET_METHOD method, CELL_LOC outloc, Solver *solver)

const Field3D bracket(const Field2D &f, const Field3D &g, BRACKET_METHOD method, CELL_LOC outloc, Solver *solver)

const Field3D bracket(const Field3D &f, const Field3D &g, BRACKET_METHOD method, CELL_LOC outloc, Solver *solver)

K.2.51 File difops.hxx

Differential operators

Changelog:

2009-01 Ben Dudson bd512@york.ac.uk

- Added two optional parameters which can be put in any order. These determine the method to use (DIFF_METHOD) and CELL_LOC location of the result. Both of these options are defined in bout_types.hxx

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.
Enums

enum BRACKET_METHOD
  Poisson bracket methods

Values:

enumerator standard
  Use b0xGrad_dot_Grad.

enumerator simple
  Keep only terms in X-Z.

enumerator arakawa
  Arakawa method in X-Z (optimised)

enumerator ctu
  Corner Transport Upwind (CTU) method. Explicit method only, needs the timestep from the solver

enumerator arakawa_old
  Older version, for regression testing of optimised version.

Functions

cost Field2D Grad_par(const Field2D &var, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")
  Parallel derivative (central differencing) in Y along unperturbed field

  Parameters
  • var – [in] The field to be differentiated
  • outloc – [in] The cell location where the output is needed (if staggered grids is enabled)
  • method – [in] The method to use. The default is set in the options.

cost Field2D Grad_par(const Field2D &var, const std::string &method, CELL_LOC outloc = CELL_DEFAULT)

cost Field3D Grad_par(const Field3D &var, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")

cost Field3D Grad_par(const Field3D &var, const std::string &method, CELL_LOC outloc = CELL_DEFAULT)

inline const Field2D Grad_par(const Field2D &var, CELL_LOC outloc, DIFF_METHOD method)

inline const Field2D Grad_par(const Field2D &var, DIFF_METHOD method, CELL_LOC outloc)

inline const Field3D Grad_par(const Field3D &var, DIFF_METHOD method, CELL_LOC outloc)

inline const Field3D Grad_par(const Field3D &var, CELL_LOC outloc, DIFF_METHOD method)

inline const Field3D Grad_par(const Field3D &var, DIFF_METHOD method, CELL_LOC outloc)
const Field3D Grad_parP(const Field3D &apar, const Field3D &f)
Derivative along perturbed magnetic field in Clebsch coordinate system
b0 dot Grad - (1/B)b0 x Grad(apar) dot Grad
Combines the parallel and perpendicular calculation to include grid-points at the corners.

const Field2D Vpar_Grad_par(const Field2D &v, const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")

vpar times parallel derivative along unperturbed B-field (upwinding)

\[ v b_0 \cdot \nabla f \]

Parameters

• v – [in] The velocity in y direction
• f – [in] The scalar field to be differentiated
• outloc – [in] The cell location of the output. By default this is the same as f
• method – [in] The numerical method to use. The default is set in the options

const Field2D Vpar_Grad_par(const Field2D &v, const Field2D &f, const std::string &method, CELL_LOC outloc = CELL_DEFAULT)

inline const Field2D Vpar_Grad_par(const Field2D &v, const Field2D &f, CELL_LOC outloc, DIFF_METHOD method)

inline const Field2D Vpar_Grad_par(const Field2D &v, const Field2D &f, DIFF_METHOD method, CELL_LOC outloc)

const Field3D Vpar_Grad_par(const Field3D &v, const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")

const Field3D Vpar_Grad_par(const Field3D &v, const Field3D &f, const std::string &method, CELL_LOC outloc = CELL_DEFAULT)

inline const Field3D Vpar_Grad_par(const Field3D &v, const Field3D &f, CELL_LOC outloc, DIFF_METHOD method)

inline const Field3D Vpar_Grad_par(const Field3D &v, const Field3D &f, DIFF_METHOD method, CELL_LOC outloc)

const Field2D Div_par(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")

parallel divergence operator

\[ B \partial_i (f/B) = B \nabla \cdot (bf/B) \]

Parameters

• f – [in] The component of a vector along the magnetic field
• **outloc** – [in] The cell location for the result. By default the same as \( f \)

• **method** – [in] The numerical method to use

```cpp
const Field2D Div_par(const Field2D &f, const std::string &method, CELL_LOC outloc = CELL_DEFAULT)
```

```cpp
inline const Field2D Div_par(const Field2D &f, CELL_LOC outloc, DIFF_METHOD method)
```

```cpp
inline const Field2D Div_par(const Field2D &f, DIFF_METHOD method, CELL_LOC outloc)
```

```cpp
const Field3D Div_par(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")
```

```cpp
const Field3D Div_par(const Field3D &f, const std::string &method, CELL_LOC outloc = CELL_DEFAULT)
```

```cpp
inline const Field3D Div_par(const Field3D &f, DIFF_METHOD method, CELL_LOC outloc)
```

```cpp
inline const Field3D Div_par(const Field3D &f, DIFF_METHOD method)
```

```cpp
const Field3D Div_par(const Field3D &f, const Field3D &v)
```

```cpp
const Field3D Div_par_flux(const Field3D &v, const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")
```

```cpp
const Field3D Div_par_flux(const Field3D &v, const Field3D &f, const std::string &method, CELL_LOC outloc = CELL_DEFAULT)
```

```cpp
inline const Field3D Div_par_flux(const Field3D &v, const Field3D &f, DIFF_METHOD method)
```

```cpp
inline const Field3D Div_par_flux(const Field3D &v, const Field3D &f, DIFF_METHOD method, CELL_LOC outloc = CELL_DEFAULT)
```

```cpp
const Field2D Grad2_par2(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")
```

second parallel derivative

\[ (b\cdot\nabla)(b\cdot\nabla) \]

Note: For parallel Laplacian use LaplacePar

**Parameters**

• **f** – [in] The field to be differentiated

• **outloc** – [in] The cell location of the result

```cpp
inline const Field2D Grad2_par2(const Field2D &f, CELL_LOC outloc, DIFF_METHOD method)
```
const Field3D Grad2_par2(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")

inline const Field3D Grad2_par2(const Field3D &f, CELL_LOC outloc, DIFF_METHOD method)

inline const Field3D Grad_par_CtoL(const Field3D &var)

Parallel derivatives, converting between cell-centred and lower cell boundary. These are a simple way to do staggered differencing

inline const Field2D Grad_par_CtoL(const Field2D &var)

inline const Field3D Vpar_Grad_par_LCtoC(const Field3D &v, const Field3D &f, const std::string &region = "RGN_NOBNDRY")

inline const Field3D Vpar_Grad_par_LCtoC(const Field3D &v, const Field3D &f, REGION region = RGN_NOBNDRY)

inline const Field3D Grad_par_LtoC(const Field3D &var)

inline const Field2D Grad_par_LtoC(const Field2D &var)

inline const Field3D Div_par_LtoC(const Field3D &var)

inline const Field2D Div_par_LtoC(const Field2D &var)

inline const Field3D Div_par_CtoL(const Field3D &var)

inline const Field2D Div_par_CtoL(const Field2D &var)

const Field2D Div_par_K_Grad_par(BoutReal kY, const Field2D &f, CELL_LOC outloc = CELL_DEFAULT)

Parallel divergence of diffusive flux, K*Grad_par

\[ \nabla \cdot (b_0 kY (b_0 \cdot \nabla) f) \]

Parameters

- \( kY \) – [in] The diffusion coefficient
- \( f \) – [in] The field whose gradient drives a flux

const Field3D Div_par_K_Grad_par(BoutReal kY, const Field3D &f, CELL_LOC outloc = CELL_DEFAULT)

const Field2D Div_par_K_Grad_par(const Field2D &kY, const Field2D &f, CELL_LOC outloc = CELL_DEFAULT)

const Field3D Div_par_K_Grad_par(const Field2D &kY, const Field3D &f, CELL_LOC outloc = CELL_DEFAULT)
const Field3D Div_par_K_Grad_par(const Field3D &kY, const Field2D &f, CELL_LOC outloc = CELL_DEFAULT)

const Field3D Div_par_K_Grad_par(const Field3D &kY, const Field3D &f, CELL_LOC outloc = CELL_DEFAULT)

const Field2D Delp2(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, bool useFFT = true)

Perpendicular Laplacian operator

This version only includes terms in X and Z, dropping derivatives in Y. This is the inverse operation to the Laplacian inversion class.

For the full perpendicular Laplacian, use Laplace_perp

const Field3D Delp2(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, bool useFFT = true)

const FieldPerp Delp2(const FieldPerp &f, CELL_LOC outloc = CELL_DEFAULT, bool useFFT = true)

const Field2D Laplace_perp(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT)

Perpendicular Laplacian, keeping Y derivatives

const Field3D Laplace_perp(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT)

const Field2D Laplace_par(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT)

Parallel Laplacian operator

const Field3D Laplace_par(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT)

const Field2D Laplace(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT)

Full Laplacian operator (par + perp)

const Field3D Laplace(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT)

const Field2D b0xGrad_dot_Grad(const Field2D &phi, const Field2D &A, CELL_LOC outloc = CELL_DEFAULT)

Terms of form b0 x Grad(phi) dot Grad(A)

const Field3D b0xGrad_dot_Grad(const Field3D &phi, const Field2D &A, CELL_LOC outloc = CELL_DEFAULT)

Terms of form

\[ b_0 \times \nabla \phi \cdot \nabla A \]

Parameters

• phi – [in] The scalar potential
• A – [in] The field being advected
• outloc – [in] The cell location where the result is defined. By default the same as A.

const Field3D b0xGrad_dot_Grad(const Field2D &phi, const Field3D &A, CELL_LOC outloc = CELL_DEFAULT)
const Field3D b0xGrad_dot_Grad(const Field3D &phi, const Field3D &A, CELL_LOC outloc = CELL_DEFAULT)

const Field2D bracket(const Field2D &f, const Field2D &g, BRACKET_METHOD method = BRACKET_STD, CELL_LOC outloc = CELL_DEFAULT, Solver *solver = nullptr)

Compute advection operator terms, which can be cast as antisymmetric Poisson brackets

\[ [f, g] = (1/B) b_0 \times \nabla f \cdot \nabla g \]

Parameters
- \( f \) – [in] The potential
- \( g \) – [in] The field being advected
- \( \text{method} \) – [in] The method to use
- \( \text{outloc} \) – [in] The cell location where the result is defined. Default is the same as \( g \)
- \( \text{solver} \) – [in] Pointer to the time integration solver

const Field3D bracket(const Field2D &f, const Field3D &g, BRACKET_METHOD method = BRACKET_STD, CELL_LOC outloc = CELL_DEFAULT, Solver *solver = nullptr)

const Field3D bracket(const Field3D &f, const Field3D &g, BRACKET_METHOD method = BRACKET_STD, CELL_LOC outloc = CELL_DEFAULT, Solver *solver = nullptr)

const Field3D bracket(const Field3D &f, const Field3D &g, BRACKET_METHOD method = BRACKET_STD, CELL_LOC outloc = CELL_DEFAULT, Solver *solver = nullptr)

Variables

constexpr BRACKET_METHOD BRACKET_STD = BRACKET_METHOD::standard
constexpr BRACKET_METHOD BRACKET_SIMPLE = BRACKET_METHOD::simple
constexpr BRACKET_METHOD BRACKET_ARAKAWA = BRACKET_METHOD::arakawa
constexpr BRACKET_METHOD BRACKET_ARAKAWA_OLD = BRACKET_METHOD::arakawa_old

K.2.52 File emptyformat.hxx

class EmptyFormat
Private Functions

inline EmptyFormat()

inline bool openr(const std::string &name)

inline bool openw(const std::string &name, bool append)

inline bool is_valid()

inline void close()

inline const std::vector<int> getSize(const char *var)

inline const std::vector<int> getSize(const std::string &var)

inline bool setOrigin(int x = 0, int y = 0, int z = 0)

inline bool setRecord(int t)

inline bool read(int *var, const char *name, int lx = 1, int ly = 0, int lz = 0)

inline bool read(int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0)

inline bool read(BoutReal *var, const char *name, int lx = 1, int ly = 0, int lz = 0)

inline bool read(BoutReal *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0)

inline bool write(int *var, const char *name, int lx = 0, int ly = 0, int lz = 0)

inline bool write(int *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0)

inline bool write(BoutReal *var, const char *name, int lx = 0, int ly = 0, int lz = 0)

inline bool write(BoutReal *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0)

inline bool read_rec(int *var, const char *name, int lx = 1, int ly = 0, int lz = 0)

inline bool read_rec(int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0)

inline bool read_rec(BoutReal *var, const char *name, int lx = 1, int ly = 0, int lz = 0)

inline bool read_rec(BoutReal *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0)
inline bool write_rec(int *var, const char *name, int lx = 0, int ly = 0, int lz = 0)

inline bool write_rec(int *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0)

inline bool write_rec(BoutReal *var, const char *name, int lx = 0, int ly = 0, int lz = 0)

inline bool write_rec(BoutReal *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0)

K.2.53 File euler.cxx

K.2.54 File euler.hxx

class EulerSolver : public Solver

Public Functions

inline EulerSolver(Options *options)

inline ~EulerSolver()

virtual void setMaxTimestep(BoutReal dt) override
    Set a maximum internal timestep (only for explicit schemes)

inline virtual BoutReal getCurrentTimestep() override
    Return the current internal timestep.

virtual int init(int nout, BoutReal tstep) override
    Initialise the solver NOTE: nout and tstep should be passed to run, not init. Needed because of how the PETSc TS code works

virtual int run() override
    Run the solver, calling monitors nout times, at intervals of tstep. This function is called by solve(), and is specific to each solver type
    This should probably be protected, since it shouldn’t be called by users.

Private Functions

void take_step(BoutReal curtime, BoutReal dt, Array<BoutReal> &start, Array<BoutReal> &result)
Private Members

int mxstep

BoutReal cfl_factor

Array<BoutReal> f0

Array<BoutReal> f1

BoutReal out_timestep

int nsteps

BoutReal timestep

bool timestep_reduced

int nlocal

K.2.55 File expr.hxx

Defines

DEFINE_BINARY_OP(name, op)
Binary operator classes.

DEFINE_OVERLOAD_FUNC(name, func)
Define functions add, mul which use operator structs.

Functions

template<typename ExprT1, typename ExprT2>
BinaryResult<ExprT1, ExprT2, Add>::type add(const ExprT1 &e1, const ExprT2 &e2)
Addition of two Expressions.

template<typename ExprT1, typename ExprT2>
BinaryResult<ExprT1, ExprT2, Multiply>::type mul(const ExprT1 &e1, const ExprT2 &e2)
Multiplication of two Expressions.

template<typename Expr>
const Field3D eval3D(Expr e)
A function to evaluate expressions.

class Literal
#include <expr.hxx> Literal class to capture BoutReal values in expressions.
Public Types

using type = Literal
Type of this expression.

Public Functions

inline Literal(BoutReal v)

inline ~Literal()

inline BoutReal operator()(int x, int y, int z) const

Private Members

const BoutReal val

class Field3DExpr

Public Types

using type = Field3D

Public Functions

inline Field3DExpr(const Field3D &f)

inline const BoutReal &operator()(int x, int y, int z) const

Private Members

const BoutReal *data

class Field2DExpr
**Public Types**

using type = Field2D

**Public Functions**

inline Field2DExpr(const Field2D &f)

inline const BoutReal &operator()(int x, int y, int z) const

**Private Members**

const BoutReal *data

template<class ExprT>
struct exprTraits
    #include <expr.hxx> Expression traits, to convert doubles etc. to Literal.

**Public Types**

using expr_type = ExprT

template<>
struct exprTraits<double>

**Public Types**

using expr_type = Literal

template<>
struct exprTraits<float>

**Public Types**

using expr_type = Literal

template<>
struct exprTraits<int>
Public Types

using expr_type = Literal
template<typename T>
struct asExpr

Public Types

using type = T

Public Static Functions

static inline const T &getExpr(const T &x)

Public Static Functions

static inline const Literal getExpr(const int &x)

Public Static Functions

static inline const Literal getExpr(const double &x)

Public Static Functions

static inline const Literal getExpr(const float &x)
Public Types

using type = Literal

Public Static Functions

static inline const Literal getExpr(const float &x)

template<>
struct asExpr<Field3D>

Public Types

using type = Field3DExpr

Public Static Functions

static inline const Field3DExpr getExpr(const Field3D &x)

template<typename Lhs, typename Rhs>
struct PromoteType

Public Types

using type = Field3D

template<class ExprT1, class ExprT2, class BinOp>
class BinaryExpr

Public Types

using ltype = typename exprTraits<ExprT1>::expr_type
using rtype = typename exprTraits<ExprT2>::expr_type
using type = typename PromoteType<ltype, rtype>::type
Type of the resulting expression.
Public Functions

inline BinaryExpr(const ExprT1 &e1, const ExprT2 &e2)

inline BoutReal operator()(int x, int y, int z) const

Private Members

ltype const _expr1
rtype const _expr2
template<typename ExprT1, typename ExprT2, class name>
struct BinaryResult

Public Types

using arg1 = typename asExpr<ExprT1>::type
using arg2 = typename asExpr<ExprT2>::type
using type = BinaryExpr<arg1, arg2, name>
struct Add

Public Static Functions

template<typename T>
static inline T apply(T a, T b)

struct Subtract

Public Static Functions

template<typename T>
static inline T apply(T a, T b)

struct Multiply
Public Static Functions

```cpp
template<typename T>
static inline T apply(T a, T b)
```

struct Divide

Public Static Functions

```cpp
template<typename T>
static inline T apply(T a, T b)
```

struct Power

Public Static Functions

```cpp
template<typename T>
static inline T apply(T a, T b)
```

K.2.56 File expressionparser.cxx

K.2.57 File expressionparser.hxx

Parses strings containing expressions, returning a tree of generators

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.
Typedefs

using FieldGeneratorPtr = std::shared_ptr<FieldGenerator>

class FieldGenerator

#include <expressionparser.hxx> Represents an operation which generates a value at a given (x,y,z) location, perhaps using other generators passed to clone()

Subclassed by FieldAbs, FieldATan, FieldBallooning, FieldBinary, FieldCos, FieldCosh, FieldErf, FieldFunction, FieldGaussian, FieldGenOneArg< Op >, FieldGenTwoArg< Op >, FieldHeaviside, FieldMax, FieldMin, FieldMixmode, FieldNull, FieldRound, FieldSin, FieldSinh, FieldSqrt, FieldTanh, FieldTanhHat, FieldValue, FieldValuePtr

Public Functions

virtual ~FieldGenerator() = default

inline virtual FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> &args)

Virtual constructor. Makes a copy of this FieldGenerator, initialised with the given list of arguments. It is up to the implementations to test whether the correct number of arguments is passed.

Parameters

args – [in] A (possibly empty) list of arguments to the generator function

virtual double generate(double x, double y, double z, double t = 0)

Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

inline virtual std::string str() const

Create a string representation of the generator, for debugging output.

class ExpressionParser

#include <expressionparser.hxx> Parses expressions, turning strings into FieldGenerator objects which can be used to perform calculations.

This class is not intended to be used directly, but should be inherited to add additional functionality

Subclassed by FieldFactory

Public Functions

ExpressionParser()

virtual ~ExpressionParser() = default

void addGenerator(const std::string &name, FieldGeneratorPtr g)

Add a generator to the parser, which can then be recognised and used in expressions.

Parameters

name – [in] The name to be recognised in expressions. This should start with a letter and contain no whitespace, only alphanumeric letters and underscores.

g – [in] The class inheriting from FieldGenerator. When recognised in an expression, the clone() function will be called to build a tree of generators
void addBinaryOp(char sym, FieldGeneratorPtr b, int precedence)
    Add a binary operator such as +,-,\ast,^.

Parameters

- sym – [in] The operator symbol. This must be a single character
- b – [in] The FieldGenerator to use. When the symbol is recognised, b->clone() will be called with two input arguments
- precedence – [in] The precedence of the operator, which decides which order operators are performed in. Higher precedence operations are done before low precedence operations. Binary operators already defined are +, - precedence = 10, \ast, / precedence = 20, ^ precedence = 30

Protected Functions

inline virtual FieldGeneratorPtr resolve(std::string &name) const
    This will be called to resolve any unknown symbols.

FieldGeneratorPtr parseString(const std::string &input) const
    Parses a given string into a tree of FieldGenerator objects.

Protected Attributes

std::string reserved_chars = "\+\-\*/[^\[\{\}]{\}"
    Characters which cannot be used in symbols; all other allowed. In addition, whitespace cannot be used.
    Adding a binary operator adds its symbol to this string.

Private Functions

FieldGeneratorPtr parseIdentifierExpr(LexInfo &lex) const

FieldGeneratorPtr parseParenExpr(LexInfo &lex) const

FieldGeneratorPtr parsePrimary(LexInfo &lex) const
    Parse a primary expression, one of:
    - number
    - identifier
    - ( ... )
    - [ ... ]
    - a unary ‘-‘, which is converted to ‘0 -‘. A ParseException is thrown if none of these is found.

FieldGeneratorPtr parseBinOpRHS(LexInfo &lex, int prec, FieldGeneratorPtr lhs) const

FieldGeneratorPtr parseExpression(LexInfo &lex) const
Private Members

`std::map<std::string, FieldGeneratorPtr> gen`
Generators, addressed by name.

`std::map<char, std::pair<FieldGeneratorPtr, int>> bin_op`
Binary operations.

struct LexInfo
Lexing info, used when splitting input into tokens.

Public Functions

`LexInfo(const std::string &input, std::string reserved_chars = "")`

char `nextToken()`
Get the next token in the string.

Public Members

signed char `curtok` = 0
Current token. -1 for number, -2 for string, 0 for “end of input”.

double `curval`
Value if a number.

`std::string curident`
Identifier, variable or function name.

signed char `LastChar`
The last character read from the string.

`std::stringstream ss`
Used to read values from the input string.

`std::string reserved_chars`
Reserved characters, not in symbols.

class FieldBinary : public FieldGenerator
#include <expressionparser.hxx> Binary operators.
Public Functions

inline FieldBinary(FieldGeneratorPtr l, FieldGeneratorPtr r, char o)

virtual FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> args) override
    Virtual constructor. Makes a copy of this FieldGenerator, initialised with the given list of arguments. It is up to the implementations to test whether the correct number of arguments is passed.

Parameters args – [in] A (possibly empty) list of arguments to the generator function

virtual double generate(double x, double y, double z, double t) override
    Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

inline virtual std::string str() const override
    Create a string representation of the generator, for debugging output.

Private Members

FieldGeneratorPtr lhs
FieldGeneratorPtr rhs
char op

class FieldValue : public FieldGenerator
    #include <expressionparser.hxx> Represent fixed values.

Public Functions

inline FieldValue(double val)

inline virtual FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> args) override
    Virtual constructor. Makes a copy of this FieldGenerator, initialised with the given list of arguments. It is up to the implementations to test whether the correct number of arguments is passed.

Parameters args – [in] A (possibly empty) list of arguments to the generator function

inline virtual double generate(double x, double y, double z, double t) override
    Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

inline virtual std::string str() const override
    Create a string representation of the generator, for debugging output.
Private Members

double value

class ParseException : public std::exception

Public Functions

ParseException (const char *, ...) BOUT_FORMAT_ARGS(2
~ParseException() override = default

const char *what() const noexcept override

Protected Attributes

std::string message

K.2.58 File fci.cxx

K.2.59 File fci.hxx

class FCIMap
   #include <fci.hxx> Field line map - contains the coefficients for interpolation.

Public Functions

FCIMap() = delete

FCIMap(Mesh &mesh, int offset, BoundaryRegionPar *boundary, bool zperiodic)

inline Field3D interpolate(Field3D &f) const

Field3D integrate(Field3D &f) const
Public Members

Mesh &map_mesh
const int offset
    Direction of map.

BoutMask boundary_mask
    boundary mask - has the field line left the domain

BoutMask corner_boundary_mask
    If any of the integration area has left the domain.

Private Members

std::unique_ptr<Interpolation> interp
    Interpolation objects.

std::unique_ptr<Interpolation> interp_corner

class FCITransform : public ParallelTransform
    #include <fci.hxx> Flux Coordinate Independent method for parallel derivatives.

Public Functions

FCITransform() = delete

inline FCITransform(Mesh &mesh, bool zperiodic = true)

virtual void calcParallelSlices(Field3D &f) override
    Given a 3D field, calculate and set the Y up down fields.

virtual void integrateParallelSlices(Field3D &f) override
    Calculate Yup and Ydown fields by integrating over mapped points This should be used for parallel divergence operators

inline virtual const Field3D toFieldAligned(const Field3D &f, const std::string &region = "RGN_ALL") override
    Convert a field into field-aligned coordinates so that the y index is along the magnetic field

inline virtual const FieldPerp toFieldAligned(const FieldPerp &f, const std::string &region = "RGN_ALL") override

inline virtual const Field3D fromFieldAligned(const Field3D &f, const std::string &region = "RGN_ALL") override
    Convert back from field-aligned coordinates into standard form

inline virtual const FieldPerp fromFieldAligned(const FieldPerp &f, const std::string &region = "RGN_ALL") override

inline virtual bool canToFromFieldAligned() override
inline bool requiresTwistShift(bool twist_shift_enabled, MAYBE_UNUSED(YDirectionType ytype)) override

Protected Functions

virtual void checkInputGrid() override

This method should be called in the constructor to check that if the grid has a ‘parallel_transform’ variable, it has the correct value

Private Members

std::vector<FCIMap> field_line_maps

FCI maps for each of the parallel slices.

K.2.60 File fft.hxx

FFT routines

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

Enums

enum FFT_MEASUREMENT_FLAG

Values:

enumerator estimate
enumerator measure
enumerator exhaustive
Functions

inline `std::string toString(FFT_MEASUREMENT_FLAG e)`

inline `FFT_MEASUREMENT_FLAG FFT_MEASUREMENT_FLAGFromString(const std::string &s)`

inline `std::ostream &operator<<(std::ostream &out, const FFT_MEASUREMENT_FLAG &e)`

inline void `rfft(const BoutReal *in, int length, dcomplex *out)`

Returns the fft of a real signal using fftw_forward

The fftw_forward returns \( \text{out}_k = \sum_{j=0}^{(\text{length}-1)} \text{in}_j \exp(-2\pi j k \sqrt{-1}/\text{length}) \)

Thus, \( \text{out}_k \) must be divided by ‘length’ in order for \( \text{DFT}[\text{IDFT}[\text{in}]] = \text{in} \) where IDFT is the inverse fourier transform. See the the fftw user manual for details.

Parameters

- `in` – [in] Pointer to the 1D array to take the fourier transform of
- `length` – [in] Number of points in the input array
- `out` – [out] Pointer to the complex 1D array which is the FFT of \( \text{in} \)

inline void `irfft(const dcomplex *in, int length, BoutReal *out)`

Take the inverse fft of signal where the outputs are only reals.

This is done through a call to `fttw_plan_dft_c2r_1d` which is calling fftw_backwards.

That is \( \text{out}_k = \sum_{j=0}^{(\text{length}-1)} \text{in}_j \exp(2\pi j k \sqrt{-1}/\text{length}) \)

See the the fftw user manual for details.

Parameters

- `in` – [in] Pointer to the 1D array to take the inverse fourier transform of
- `length` – [in] Number of points in the input array
- `out` – [out] Pointer to the complex 1D array which is IFFT\( \text{ed} \)

inline void `DST(const BoutReal *in, int length, dcomplex *out)`

Discrete Sine Transform

in and out arrays must both be of the same length

inline void `DST_rev(dcomplex *in, int length, BoutReal *out)`

Inverse Discrete Sine Transform

in and out arrays must both be of the same length

namespace `bout`

SNB model

namespace `fft`
**Functions**

**inline void rfft(const BoutReal *in, int length, dcomplex *out)**

Returns the fft of a real signal using fftw_forward.

The `fftw_forward` returns \( \text{out}_k = \sum_{j=0}^{(\text{length}-1)} \text{in}_j \exp(-2\pi j k \sqrt{-1}/\text{length}) \).

Thus, \( \text{out}_k \) must be divided by ‘\( \text{length} \)’ in order for DFT IDFT [\( \text{in} \)] = in where IDFT is the inverse fourier transform. See the the fftw user manual for details.

**Parameters**

- **in** – [in] Pointer to the 1D array to take the fourier transform of
- **length** – [in] Number of points in the input array
- **out** – [out] Pointer to the complex 1D array which is the FFT of in

**inline void irfft(const dcomplex *in, int length, BoutReal *out)**

Take the inverse fft of signal where the outputs are only reals.

This is done through a call to `fftw_plan_dft_c2r_1d` which is calling `fftw_backwards`.

That is \( \text{out}_k = \sum_{j=0}^{(\text{length}-1)} \text{in}_j \exp(2\pi j k \sqrt{-1}/\text{length}) \)

See the the fftw user manual for details.

**Parameters**

- **in** – [in] Pointer to the 1D array to take the inverse fourier transform of
- **length** – [in] Number of points in the input array
- **out** – [out] Pointer to the complex 1D array which is IFFTed

**inline void DST(const BoutReal *in, int length, dcomplex *out)**

Discrete Sine Transform

\( \text{in} \) and \( \text{out} \) arrays must both be of the same \( \text{length} \)

**inline void DST_rev(const dcomplex *in, int length, BoutReal *out)**

Inverse Discrete Sine Transform

\( \text{in} \) and \( \text{out} \) arrays must both be of the same \( \text{length} \)

**void fft_init(bool fft_measure)**

Should the FFT functions find and use an optimised plan?

**void fft_init(FFT_MEASUREMENT_FLAG fft_flag)**

Should the FFT functions find and use an optimised plan?

**void fft_init(Options *options = nullptr)**

Should the FFT functions find and use an optimised plan?

If \( \text{options} \) is not nullptr, it should contain a bool called “fftw_measure”. If it is nullptr, use the global `Options` root

**Array<dcomplex> rfft(const Array<BoutReal> &in)**

Returns the fft of a real signal \( \text{in} \) using `fftw_forward`.

**Array<BoutReal> irfft(const Array<dcomplex> &in, int length)**

Take the inverse fft of signal \( \text{in} \) where the outputs are only reals. Requires the \( \text{length} \) of the original real signal

\( \text{length} \) is required because input signals to the forward transform of length \( n \) and \( n + 1 \) both produce ffts of length \( (n / 2) + 1 \) i.e. it’s not possible to recover the length of the original signal from the fft alone.

Expects that \( \text{in.size()} = (\text{length} / 2) + 1 \)
K.2.61 File fft_fftw.cxx

FFT routines using external libraries.

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

namespace bout

SNB model

namespace fft

Functions

unsigned int get_measurement_flag (FFT_MEASUREMENT_FLAG fft_measurement_flag)

void rfft (MAYBE_UNUSED(const BoutReal *in), MAYBE_UNUSED(int length),
           MAYBE_UNUSED(dcomplex *out))

void irfft (MAYBE_UNUSED(const dcomplex *in), MAYBE_UNUSED(int length),
            MAYBE_UNUSED(BoutReal *out))

void DST (MAYBE_UNUSED(const BoutReal *in), MAYBE_UNUSED(int length),
          MAYBE_UNUSED(dcomplex *out))

void DST_rev (MAYBE_UNUSED(dcomplex *in), MAYBE_UNUSED(int length),
              MAYBE_UNUSED(BoutReal *out))
Variables

bool fft_initialised = {false}
Have we set fft_measure?

FFT_MEASUREMENT_FLAG fft_measurement_flag = {FFT_MEASUREMENT_FLAG::estimate}
Should FFTW find an optimised plan by measuring various plans?

K.2.62 File field2d.cxx

Class for 2D X-Y profiles
Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu
Contact: Ben Dudson, bd512@york.ac.uk
This file is part of BOUT++.
BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.
You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

Functions

Field2D operator- (const Field2D &f)
Unary minus. Returns the negative of given field, iterates over whole domain including guard/boundary cells.

void checkData (const Field2D &f, const std::string &region)
Check if the data is valid.
Throw an exception if f is not allocated or if any elements are non-finite (for CHECK > 2). Loops over all points including the boundaries by default (can be changed using the rgn argument

void invalidateGuards (Field2D &var)
Force guard cells of passed field var to NaN.

bool operator== (const Field2D &a, const Field2D &b)
Test if two fields are the same, by calculating the minimum absolute difference between them

std::ostream &operator<< (std::ostream &out, const Field2D &value)
K.2.63 File field2d.hxx

Definition of 2D scalar field class.

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

Defines

__FIELD2D_H__

Functions

Field2D operator+(const Field2D &lhs, const Field2D &rhs)

Field2D operator-(const Field2D &lhs, const Field2D &rhs)

Field2D operator*(const Field2D &lhs, const Field2D &rhs)

Field2D operator/(const Field2D &lhs, const Field2D &rhs)

Field3D operator+(const Field2D &lhs, const Field3D &rhs)

Field3D operator-(const Field2D &lhs, const Field3D &rhs)

Field3D operator*(const Field2D &lhs, const Field3D &rhs)

Field3D operator/(const Field2D &lhs, const Field3D &rhs)

Field2D operator+(const Field2D &lhs, BoutReal rhs)

Field2D operator-(const Field2D &lhs, BoutReal rhs)

Field2D operator*(const Field2D &lhs, BoutReal rhs)
**Field2D** operator/(const **Field2D** &lhs, **BoutReal** rhs)

**Field2D** operator+(**BoutReal** lhs, const **Field2D** &rhs)

**Field2D** operator-(**BoutReal** lhs, const **Field2D** &rhs)

**Field2D** operator*(**BoutReal** lhs, const **Field2D** &rhs)

**Field2D** operator/(**BoutReal** lhs, const **Field2D** &rhs)

**Field2D** operator-((const **Field2D** &f))

Unary minus. Returns the negative of given field, iterates over whole domain including guard/boundary cells.

inline **Field2D** toFieldAligned((const **Field2D** &f, const **std**::string &region = "RGN_ALL")

inline **Field2D** toFieldAligned(const **Field2D** &f, **REGION** region)

inline **Field2D** fromFieldAligned(const **Field2D** &f, const **std**::string &region = "RGN_ALL")

inline **Field2D** fromFieldAligned(const **Field2D** &f, **REGION** region)

void checkData((const **Field2D** &f, const **std**::string &region = "RGN_NOBNDRY")

Check if the data is valid.

Throw an exception if f is not allocated or if any elements are non-finite (for CHECK > 2). Loops over all points including the boundaries by default (can be changed using the rgn argument

inline void checkData(const **Field2D** &f, **REGION** region)

void invalidateGuards (**Field2D** &var)

Force guard cells of passed field var to NaN.

inline **Field2D** &ddt (**Field2D** &f)

Returns a reference to the time-derivative of a field f

Wrapper around member function f.timeDeriv()

template<>

inline **std**::string toString(const **Field2D** &val)

toString template specialisation Defined in utils.hxx

bool operator==(const **Field2D** &a, const **Field2D** &b)

Test if two fields are the same, by calculating the minimum absolute difference between them

**std**::ostream &operator<<(**std**::ostream &out, const **Field2D** &value)

class **Field2D** : public **Field**, public **FieldData**

#include <field2d.hxx> 2D X-Y scalar fields

Handles data for axisymmetric quantities. Essentially the same as the **Field3D** class.
Public Types

using \texttt{ind\_type} = \texttt{Ind2D}

using \texttt{value\_type} = \texttt{BoutReal}

Data type.

Public Functions

\texttt{Field2D(Mesh*localmesh = \texttt{nullptr}, CELL\_LOC \texttt{location\_in = \texttt{CELL\_CENTRE}, DirectionTypes}}
\texttt{directions\_in = \{YDirectionType::Standard, ZDirectionType::Average\})}

Constructor, taking an optional mesh pointer. This mesh pointer is not used until the data is allocated, since \texttt{Field2D} objects can be globals, created before a mesh has been created.

By default the global \texttt{Mesh} pointer (mesh) is used.

\textbf{Parameters} \texttt{localmesh} – \texttt{[in]} The mesh which defines the field size.

\texttt{Field2D(const Field2D &f)}

Copy constructor. After this both fields will share the same underlying data.

inline \texttt{Field2D(Field2D &&f) noexcept}

Move constructor

\texttt{Field2D(BoutReal val, Mesh*localmesh = \texttt{nullptr})}

Constructor. This creates a \texttt{Field2D} using the global \texttt{Mesh} pointer (mesh) allocates data, and assigns the value \texttt{val} to all points including boundary cells.

\texttt{Field2D(Array<BoutReal> data, Mesh*localmesh, CELL\_LOC \texttt{location = \texttt{CELL\_CENTRE}, DirectionTypes}}
\texttt{directions\_in = \{YDirectionType::Standard, ZDirectionType::Average\})}

Constructor from \texttt{Array} and \texttt{Mesh}.

\texttt{~Field2D()} override

Destructor

\texttt{Field2D &allocate()}\n
Ensure data is allocated.

inline bool \texttt{isAllocated()} const

Test if data is allocated.

\texttt{Field2D *timeDeriv()}\n
Return a pointer to the time-derivative field.

inline virtual int \texttt{getNx()} const override

Return the number of nx points

inline virtual int \texttt{getNy()} const override

Return the number of ny points

inline virtual int \texttt{getNz()} const override

Return the number of nz points

inline \texttt{Field2D &setLocation(CELL\_LOC new\_location)}

inline \texttt{Field2D &setDirectionY(YDirectionType d)}
inline bool hasParallelSlices() const
    Check if this field has yup and ydown fields.

inline bool hasYupYdown() const

inline Field2D &yup(std::vector<Field2D>::size_type index = 0)

inline Field2D &yup(std::vector<Field2D>::size_type index = 0) const

inline Field2D &ydown(std::vector<Field2D>::size_type index = 0)

inline Field2D &ydown(std::vector<Field2D>::size_type index = 0) const

inline Field2D &ynext(int dir)

inline Field2D &ynext(int dir) const

Field2D &operator=(const Field2D &rhs)
    Assignment from Field2D. After this both fields will share the same underlying data. To make a true copy, call .allocate() after assignment, or use the copy() function.

Field2D &operator=(BoutReal rhs)
    Allocates data if not already allocated, then sets all cells to rhs

const Region<Ind2D> &getRegion(REGION region) const
    Return a Region<Ind2D> reference to use to iterate over this field.

const Region<Ind2D> &getRegion(const std::string &region_name) const

inline Region<Ind2D>::RegionIndices::const_iterator begin() const

inline Region<Ind2D>::RegionIndices::const_iterator end() const

inline BoutReal &operator[](const Ind2D &d)

inline const BoutReal &operator[](const Ind2D &d) const

BoutReal &operator[](const Ind3D &d)

cost BoutReal &operator[](const Ind3D &d) const

inline BoutReal &operator()(int jx, int jy)
    Access to the underlying data array.
    If CHECK <= 2 then no checks are performed
    If CHECK > 2 then both jx and jy are bounds checked. This will significantly reduce performance.

inline const BoutReal &operator()(int jx, int jy) const
inline \texttt{BoutReal \&operator() (int jx, int jy, int jz)}
\begin{itemize}
\item Direct access to underlying array. This version is for compatibility with \texttt{Field3D} objects
\end{itemize}

inline \texttt{const BoutReal \&operator() (int jx, int jy, int jz) const}

\begin{itemize}
\item \texttt{Field2D \&operator+=\texttt{(const Field2D \&rhs)}}
\item In-place addition. Copy-on-write used if data is shared.
\end{itemize}

\begin{itemize}
\item \texttt{Field2D \&operator+=\texttt{(BoutReal rhs)}}
\item In-place addition. Copy-on-write used if data is shared.
\end{itemize}

\begin{itemize}
\item \texttt{Field2D \&operator\-=\texttt{(const Field2D \&rhs)}}
\item In-place subtraction. Copy-on-write used if data is shared.
\end{itemize}

\begin{itemize}
\item \texttt{Field2D \&operator\-=\texttt{(BoutReal rhs)}}
\item In-place subtraction. Copy-on-write used if data is shared.
\end{itemize}

\begin{itemize}
\item \texttt{Field2D \&operator\*=\texttt{(const Field2D \&rhs)}}
\item In-place multiplication. Copy-on-write used if data is shared.
\end{itemize}

\begin{itemize}
\item \texttt{Field2D \&operator\*=\texttt{(BoutReal rhs)}}
\item In-place multiplication. Copy-on-write used if data is shared.
\end{itemize}

\begin{itemize}
\item \texttt{Field2D \&operator/\texttt{=\texttt{(const Field2D \&rhs)}}}
\item In-place division. Copy-on-write used if data is shared.
\end{itemize}

\begin{itemize}
\item \texttt{Field2D \&operator/\texttt{=\texttt{(BoutReal rhs)}}}
\item In-place division. Copy-on-write used if data is shared.
\end{itemize}

\begin{itemize}
\item inline virtual void \texttt{accept\texttt{(FieldVisitor \&v)}}\texttt{override}
\item Visitor pattern support.
\end{itemize}

\begin{itemize}
\item inline virtual bool \texttt{isReal\texttt{() \&\texttt{const override}}}
\item Returns true if field consists of BoutReal values.
\end{itemize}

\begin{itemize}
\item inline virtual bool \texttt{is3D\texttt{() \&\texttt{const override}}}
\item True if variable is 3D.
\end{itemize}

\begin{itemize}
\item inline virtual int \texttt{byteSize\texttt{() \&\texttt{const override}}}
\item Number of bytes for a single point.
\end{itemize}

\begin{itemize}
\item inline virtual int \texttt{BoutRealSize\texttt{() \&\texttt{const override}}}
\item Number of BoutReals (not implemented if not BoutReal).
\end{itemize}

\begin{itemize}
\item inline virtual void \texttt{doneComms\texttt{() \&\texttt{override}}}
\end{itemize}

\begin{itemize}
\item virtual void \texttt{applyBoundary\texttt{(bool init = false) \&\texttt{override}}}
\end{itemize}

\begin{itemize}
\item void \texttt{applyBoundary\texttt{(BoutReal time)}}
\end{itemize}

\begin{itemize}
\item void \texttt{applyBoundary\texttt{(const std::string \&condition)}}
\end{itemize}

\begin{itemize}
\item inline void \texttt{applyBoundary\texttt{(const char \*condition)}}
\end{itemize}

\begin{itemize}
\item void \texttt{applyBoundary\texttt{(const std::string \&region, const std::string \&condition)}}
\end{itemize}
virtual void applyTDerivBoundary() override

void setBoundaryTo(const Field2D &f2d)
    Copy the boundary region.

Private Members

int nx = {-1}
    Array sizes (from fieldmesh). These are valid only if fieldmesh is not null.

int ny = {-1}
    Array<BoutReal> data
    Internal data array. Handles allocation/freeing of memory.

Field2D deriv = {nullptr}
    Time-derivative, can be nullptr.

Friends

friend class Vector2D

inline friend friend void swap (Field2D &first, Field2D &second) noexcept

K.2.64 File field3d.cxx

Class for 3D X-Y-Z scalar fields
Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu
Contact: Ben Dudson, bd512@york.ac.uk
This file is part of BOUT++.
BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.
You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.
Functions

Field3D operator- (const Field3D &f)
Unary minus. Returns the negative of given field, iterates over whole domain including guard/boundary cells.

Field3D pow (const Field3D &lhs, const Field2D &rhs, const std::string &rgn)
Exponent: pow(lhs, rhs) is lhs raised to the power of rhs
Extra overloads not provided by the templates in field.hxx
This loops over the entire domain, including guard/boundary cells by default (can be changed using the rgn argument). If CHECK >= 3 then the result will be checked for non-finite numbers

FieldPerp pow (const Field3D &lhs, const FieldPerp &rhs, const std::string &rgn)

Field3D filter (const Field3D &var, int N0, const std::string &rgn)
Fourier filtering, removes all except one mode

Parameters
• var – [in] Variable to apply filter to
• N0 – [in] The component to keep
• rgn – [in] The region to calculate the result over

Field3D lowPass (const Field3D &var, int zmax, bool keep_zonal, const std::string &rgn)
Fourier low pass filtering. Removes modes higher than zmax and optionally the zonal component

Parameters
• var – [in] Variable to apply filter to
• zmax – [in] Maximum mode in Z
• keep_zonal – [in] Keep the zonal component if true
• rgn – [in] The region to calculate the result over

void shiftZ (Field3D &var, int jx, int jy, double zangle)
Perform a shift by a given angle in Z

Parameters
• var – [inout] The variable to be modified in-place
• jx – [in] X index
• jy – [in] Y index
• zangle – [in] The Z angle to apply

void shiftZ (Field3D &var, double zangle, const std::string &rgn)
Apply a phase shift by a given angle zangle in Z to all points

Parameters
• var – [inout] The variable to modify in-place
• zangle – [in] The angle to shift by in Z
• rgn – [in] The region to calculate the result over

void checkData (const Field3D &f, const std::string &region)
Throw an exception if f is not allocated or if any elements are non-finite (for CHECK > 2). Loops over all points including the boundaries by default (can be changed using the rgn argument)
Field2D DC(const Field3D &f, const std::string &rgn)
Average in the Z direction

Parameters

• **f** – [in] Variable to average
• **rgn** – [in] The region to calculate the result over

void invalidateGuards(Field3D &var)
Force guard cells of passed field var to NaN.

bool operator==(const Field3D &a, const Field3D &b)
Test if two fields are the same, by calculating the minimum absolute difference between them

std::ostream &operator<<(std::ostream &out, const Field3D &value)
Output a string describing a Field3D to a stream.

### K.2.65 File field3d.hxx

**Defines**

```cpp
#define __FIELD3D_H__
```

**Functions**

```cpp
FieldPerp operator+(const Field3D &lhs, const FieldPerp &rhs)
FieldPerp operator-(const Field3D &lhs, const FieldPerp &rhs)
FieldPerp operator*(const Field3D &lhs, const FieldPerp &rhs)
FieldPerp operator/(const Field3D &lhs, const FieldPerp &rhs)
Field3D operator+(const Field3D &lhs, const Field3D &rhs)
Field3D operator-(const Field3D &lhs, const Field3D &rhs)
Field3D operator*(const Field3D &lhs, const Field3D &rhs)
Field3D operator/(const Field3D &lhs, const Field3D &rhs)
Field3D operator+(const Field3D &lhs, const Field2D &rhs)
Field3D operator-(const Field3D &lhs, const Field2D &rhs)
Field3D operator*(const Field3D &lhs, const Field2D &rhs)
Field3D operator/(const Field3D &lhs, const Field2D &rhs)
```
\texttt{Field3D\ operator/}(\texttt{const Field3D} \ &\texttt{lhs}, \texttt{const Field2D} \ &\texttt{rhs})

\texttt{Field3D\ operator+}(\texttt{const Field3D} \ &\texttt{lhs}, \texttt{BoutReal} \ \texttt{rhs})

\texttt{Field3D\ operator-}(\texttt{const Field3D} \ &\texttt{lhs}, \texttt{BoutReal} \ \texttt{rhs})

\texttt{Field3D\ operator*}(\texttt{const Field3D} \ &\texttt{lhs}, \texttt{BoutReal} \ \texttt{rhs})

\texttt{Field3D\ operator/}(\texttt{const Field3D} \ &\texttt{lhs}, \texttt{BoutReal} \ \texttt{rhs})

\texttt{Field3D\ operator+}(\texttt{BoutReal} \ \texttt{lhs}, \texttt{const Field3D} \ &\texttt{rhs})

\texttt{Field3D\ operator-}(\texttt{BoutReal} \ \texttt{lhs}, \texttt{const Field3D} \ &\texttt{rhs})

\texttt{Field3D\ operator*}(\texttt{BoutReal} \ \texttt{lhs}, \texttt{const Field3D} \ &\texttt{rhs})

\texttt{Field3D\ operator/}(\texttt{BoutReal} \ \texttt{lhs}, \texttt{const Field3D} \ &\texttt{rhs})

\texttt{Field3D\ operator-}(\texttt{const Field3D} \ &\texttt{f})

Unary minus. Returns the negative of given field, iterates over whole domain including guard/boundary cells.

\texttt{Field3D\ pow}(\texttt{const Field3D} \ &\texttt{lhs}, \texttt{const Field2D} \ &\texttt{rhs}, \texttt{const std::string} \ &\texttt{rgn} = \texttt{"RGN\_ALL"})

Exponent: \texttt{pow(lhs, rhs) is lhs raised to the power of rhs}

Extra overloads not provided by the templates in field.hxx

This loops over the entire domain, including guard/boundary cells by default (can be changed using the \texttt{rgn} argument). If \texttt{CHECK} \geq 3 then the result will be checked for non-finite numbers

\texttt{inline Field3D\ pow}\ (\texttt{const Field3D} \ &\texttt{lhs}, \texttt{const Field2D} \ &\texttt{rhs}, \texttt{REGION} \ \texttt{rgn})

\texttt{FieldPerp\ pow}(\texttt{const Field3D} \ &\texttt{lhs}, \texttt{const FieldPerp} \ &\texttt{rhs}, \texttt{const std::string} \ &\texttt{rgn} = \texttt{"RGN\_ALL"})

\texttt{inline FieldPerp\ pow}\ (\texttt{const Field3D} \ &\texttt{lhs}, \texttt{const FieldPerp} \ &\texttt{rhs}, \texttt{REGION} \ \texttt{rgn})

\texttt{void checkData}\ (\texttt{const Field3D} \ &\texttt{f}, \texttt{const std::string} \ &\texttt{region} = \texttt{"RGN\_NOBNDRY"})

Throw an exception if \texttt{f} is not allocated or if any elements are non-finite (for \texttt{CHECK} \geq 2). Loops over all points including the boundaries by default (can be changed using the \texttt{rgn} argument

\texttt{inline void checkData}\ (\texttt{const Field3D} \ &\texttt{f}, \texttt{REGION} \ \texttt{region})

\texttt{Field3D\ filter}(\texttt{const Field3D} \ &\texttt{var}, \texttt{int} \texttt{N0}, \texttt{const std::string} \ &\texttt{rgn} = \texttt{"RGN\_ALL"})

Fourier filtering, removes all except one mode

\textbf{Parameters}

- \texttt{var} – \texttt{[in]} Variable to apply filter to
- \texttt{N0} – \texttt{[in]} The component to keep
- \texttt{rgn} – \texttt{[in]} The region to calculate the result over
BOUT++ Documentation, Release 4.4.0

inline `Field3D filter` (const `Field3D` &var, int N0, `REGION` rgn)

`Field3D lowPass` (const `Field3D` &var, int zmax, bool keep_zonal, const `std`::string &rgn = "RGN_ALL")
Fourier low pass filtering. Removes modes higher than zmax and optionally the zonal component

Parameters

- `var` – [in] Variable to apply filter to
- `keep_zonal` – [in] Keep the zonal component if true
- `rgn` – [in] The region to calculate the result over

inline `Field3D lowPass` (const `Field3D` &var, int zmax, bool keep_zonal, `REGION` rgn)

The argument `keep_zonal` used to be integer "zmin" this was a misnomer. Please use the version above which uses a bool instead

inline `Field3D lowPass` (const `Field3D` &var, int zmax, const `std`::string rgn = "RGN_ALL")
Fourier low pass filtering. Removes modes higher than zmax

Parameters

- `var` – [in] Variable to apply filter to
- `rgn` – [in] The region to calculate the result over

void `shiftZ` (`Field3D` &var, int jx, int jy, double zangle)
Perform a shift by a given angle in Z

Parameters

- `var` – [inout] The variable to be modified in-place
- `jx` – [in] X index
- `jy` – [in] Y index
- `zangle` – [in] The Z angle to apply

void `shiftZ` (`Field3D` &var, `BoutReal` zangle, const `std`::string &rgn = "RGN_ALL")
Apply a phase shift by a given angle `zangle` in Z to all points

Parameters

- `var` – [inout] The variable to modify in-place
- `zangle` – [in] The angle to shift by in Z
- `rgn` – [in] The region to calculate the result over

inline void `shiftZ` (`Field3D` &var, `BoutReal` zangle, `REGION` rgn)

`Field2D DC` (const `Field3D` &f, const `std`::string &rgn = "RGN_ALL")
Average in the Z direction

Parameters
• \( f \) – [in] Variable to average
• \( rgn \) – [in] The region to calculate the result over

inline Field2D DC(const Field3D &f, REGION rgn)

void invalidateGuards(Field3D &var)
    Force guard cells of passed field \( var \) to NaN.

inline Field3D &ddt(Field3D &f)
    Returns a reference to the time-derivative of a field \( f \)
    Wrapper around member function \( f.timeDeriv() \)

template<>
inline std::string toString(const Field3D &val)
    toString template specialisation Defined in utils.hxx

bool operator==(const Field3D &a, const Field3D &b)
    Test if two fields are the same, by calculating the minimum absolute difference between them

std::ostream &operator<<(std::ostream &out, const Field3D &value)
    Output a string describing a Field3D to a stream.

class Field3D : public Field, public FieldData
    #include <field3d.hxx> Class for 3D X-Y-Z scalar fields.

This class represents a scalar field defined over the mesh. It handles memory management, and provides overloaded operators for operations on the data, iterators and access methods.

Initialisation

Fields can be declared in any scope (even global), but cannot be accessed by index or used until the data is allocated.

```plaintext
Field3D f;  // Declare variable, no data allocated
f(0,0,0) = 1.0;  // Error !

f = 0.0;  // Allocates memory, fills with value (0.0)

Field3D g(1.0);  // Declares, allocates memory, fills with value (1.0)

Field3D h;  // not allocated
h.allocate();  // Data array allocated, values undefined
f(0,0,0) = 1.0;  // ok
```

Copy-on-Write

A field is a reference to the underlying data array, so setting one field equal to another has the effect of making both fields share the same underlying data.

```plaintext
Field3D f(0.0);
Field3D g = f;  // f and g now share data
f(0,0,0) = 1.0;  // g is also modified
```

Setting the entire field equal to a new value changes the reference:
Field3D f(0.0);
Field3D g = f; // f and g now share data
   g = 1.0;    // g and f are now separate

To ensure that a field is unique, call allocate() which will make a copy of the underlying data if it is shared.

Field3D f(0.0);
Field3D g = f; // f and g now share data
   g.allocate(); // Data copied so g and f don’t share data
   f(0,0,0) = 1.0; // ok

Data access

Individual data indices can be accessed by index using round brackets:

Field3D f;
   f(0,1,2) = 1.0; // Set value
   BoutReal val = f(2,1,3); // Get value

If CHECK is greater than 2, this function will perform bounds checking. This will significantly slow calculations.

Some methods, such as FFT routines, need access to a pointer to memory. For the Z dimension this can be done by passing only the X and Y indices

BoutReal *data = f(0,1);

data now points to f(0,1,0) and can be incremented to move in Z.

Iteration

To loop over all points in a field, a for loop can be used to get the indices:

Field3D f(0.0); // Allocate, set to zero
   for( const auto &i : f ) { // Loop over all points, with index i
      f[i] = 1.0;
   }

There is also more explicit looping over regions:

for( const auto &i : f.region(RGN_ALL) ) { // Loop over all points, with index i
      f[i] = 1.0;
}

Parallel (y) derivatives

In several numerical schemes the mapping along magnetic fields (default y direction) is a relatively complex map. To accommodate this, the values of a field in the positive (up) and negative (down) directions can be stored in separate fields.

Field3D f(0.0); // f allocated, set to zero
   f.yup(); // error; f.yup not allocated
   f.clearParallelSlices(); // f.yup_fields and f.ydown_fields are now empty
   f.yup(); // error; f.yup not allocated

(continues on next page)
To have separate fields for yup and ydown, first call

\[ f.\text{splitParallelSlices;} \quad // \quad f.\text{yup()} \quad \text{and} \quad f.\text{ydown()} \quad \text{separate} \]

\[ f.\text{yup;} \quad // \quad \text{ok} \]
\[ f.\text{yup}();(0,1,0) \quad // \quad \text{error; } f.\text{yup not allocated} \]

\[ f.\text{yup()} = 1.0; \quad // \quad \text{Set } f.\text{yup()} \quad \text{field to } 1.0 \]

\[ f.\text{yup}();(0,1,0) \quad // \quad \text{ok} \]

**Unnamed Group**

Field3D &operator=(const Field3D &rhs)
Assignment operators

Field3D &operator=(const Field2D &rhs)

void operator=(const FieldPerp &rhs)
return void, as only part initialised

Field3D &operator=(BoutReal val)

**Unnamed Group**

Field3D &operator+=(const Field3D &rhs)
Addition operators

Field3D &operator+=(const Field2D &rhs)

Field3D &operator+=(BoutReal rhs)

**Unnamed Group**

Field3D &operator-=(const Field3D &rhs)
Subtraction operators

Field3D &operator-=(const Field2D &rhs)

Field3D &operator-=(BoutReal rhs)
**Unnamed Group**

\[ \text{Field3D} \& \text{operator*=}(\text{const Field3D} \& \text{rhs}) \]
Multiplication operators

\[ \text{Field3D} \& \text{operator*=}(\text{const Field2D} \& \text{rhs}) \]

\[ \text{Field3D} \& \text{operator*=}(\text{BoutReal} \ \text{rhs}) \]

**Unnamed Group**

\[ \text{Field3D} \& \text{operator/=}(\text{const Field3D} \& \text{rhs}) \]
Division operators

\[ \text{Field3D} \& \text{operator/=}(\text{const Field2D} \& \text{rhs}) \]

\[ \text{Field3D} \& \text{operator/=}(\text{BoutReal} \ \text{rhs}) \]

**Public Types**

using \textit{ind\_type} = \textit{Ind3D}

using \textit{value\_type} = \textit{BoutReal}

Data type stored in this field.

**Public Functions**

\textbf{Field3D(\textit{Mesh} \*\textit{localmesh} = \textit{nullptr}, \textit{CELL\_LOC} \textit{location\_in} = \textit{CELL\_CENTRE}, \textit{DirectionTypes} \textit{directions\_in} = \{\textit{YDirectionType::Standard, ZDirectionType::Standard}\})}
Constructor.

Note: the global “mesh” can’t be passed here because fields may be created before the mesh is.

\textbf{Field3D(\textit{const Field3D} \&\textit{f})}
Copy constructor

Doesn’t copy any data, just create a new reference to the same data (copy on change later)

\textbf{inline Field3D(\textit{Field3D} \&\&\textit{f}) noexcept}
Move constructor.

\textbf{Field3D(\textit{const Field2D} \&\textit{f})}
Constructor from 2D field.

\textbf{Field3D(\textit{BoutReal} \textit{val}, \textit{Mesh} \*\textit{localmesh} = \textit{nullptr})}
Constructor from value.

\textbf{Field3D(\textit{Array<\textit{BoutReal}> data, \textit{Mesh} \*\textit{localmesh}, \textit{CELL\_LOC} \textit{location} = \textit{CELL\_CENTRE}, \textit{DirectionTypes} \textit{directions\_in} = \{\textit{YDirectionType::Standard, ZDirectionType::Standard}\})}
Constructor from \textit{Array} and \textit{Mesh}. 
~Field3D() override
Destructor.

Field3D &allocate()
Ensures that memory is allocated and unique

inline bool isAllocated() const
Test if data is allocated

Field3D* timeDeriv()
Return a pointer to the time-derivative field
The first time this is called, a new field will be allocated. Subsequent calls return the same field

inline virtual int getNx() const override
Return the number of nx points

inline virtual int getNy() const override
Return the number of ny points

inline virtual int getNz() const override
Return the number of nz points

inline Field3D &setLocation(CELL_LOC new_location)

inline Field3D &setDirectionY(YDirectionType d)

void splitParallelSlices()
Ensure that this field has separate fields for yup and ydown.

inline void splitYupYdown()

void clearParallelSlices()
Clear the parallel slices, yup and ydown

inline void mergeYupYdown()

inline bool hasParallelSlices() const
Check if this field has yup and ydown fields.

inline bool hasYupYdown() const

inline Field3D &yup(std::vector<Field3D>::size_type index = 0)
Check if this field has yup and ydown fields Return reference to yup field

inline const Field3D &yup(std::vector<Field3D>::size_type index = 0) const
Return const reference to yup field.

inline Field3D &ydown(std::vector<Field3D>::size_type index = 0)
Return reference to ydown field.

inline const Field3D &ydown(std::vector<Field3D>::size_type index = 0) const
Return const reference to ydown field.

Field3D &ynext(int offset)
Return the parallel slice at offset
offset of 0 returns the main field itself
const \textit{Field3D} &\texttt{ynext}(\textit{int offset}) const

bool \texttt{requiresTwistShift}(\texttt{bool twist\_shift\_enabled})

\textit{If twist\_shift\_enabled} is true, does this \textit{Field3D} require a twist-shift at branch cuts on closed field lines?

const \textit{Region<Ind3D>} &\texttt{getRegion}(\textit{REGION region}) const

Return a \textit{Region<Ind3D>} reference to use to iterate over this field

\textit{Example}

This loops over the interior points, not the boundary and inside the loop the index is used to calculate the difference between the point one index up in \textit{x} (\texttt{i.xp()}) and one index down in \textit{x} (\texttt{i.xm()}), putting the result into a different field \textquote{g'}

\begin{verbatim}
for(const auto &i : f.getRegion(RGN_NOBNDRY)) { g[i] = f[i.xp()] - f[i.xm()]; }
\end{verbatim}

const \textit{Region<Ind3D>} &\texttt{getRegion}(\textit{std::string \&region\_name}) const

const \textit{Region<Ind2D>} &\texttt{getRegion2D}(\textit{REGION region}) const

Return a \textit{Region<Ind2D>} reference to use to iterate over the \textit{x-} and \textit{y-}indices of this field

const \textit{Region<Ind2D>} &\texttt{getRegion2D}(\textit{std::string \&region\_name}) const

\begin{verbatim}
inline \textit{Region<Ind3D>::RegionIndices::const\_iterator} \texttt{begin()} const

inline \textit{Region<Ind3D>::RegionIndices::const\_iterator} \texttt{end()} const

inline \textit{BoutReal \&operator[]}(\textit{const Ind3D \&d})

inline const \textit{BoutReal \&operator[]}(\textit{const Ind3D \&d}) const

\textit{BoutReal \&operator()}(\textit{const IndPerp \&d, \textit{int jy}})

const \textit{BoutReal \&operator()}(\textit{const IndPerp \&d, \textit{int jy}}) const

\textit{BoutReal \&operator()}(\textit{const Ind2D \&d, \textit{int jz}})

const \textit{BoutReal \&operator()}(\textit{const Ind2D \&d, \textit{int jz}}) const

inline \textit{BoutReal \&operator()}(\textit{int jx, \textit{int jy, \textit{int jz}}})

\textit{Direct access to the underlying data array}

\textit{If CHECK > 2 then bounds checking is performed}

\textit{If CHECK <= 2 then no checks are performed, to allow inlining and optimisation of inner loops}

inline const \textit{BoutReal \&operator()}(\textit{int jx, \textit{int jy, \textit{int jz}}}) const

\begin{verbatim}
inline const \textit{BoutReal \*operator()}(\textit{int jx, \textit{int jy}}) const

\textit{Direct access to the underlying data array}
\end{verbatim}
This version returns a pointer to a data array, and is intended for use with FFT routines. The data is guaranteed to be contiguous in Z index.

```cpp
inline BoutReal *operator()(int jx, int jy)
```

```cpp
inline virtual bool isReal() const override
Returns true if field consists of BoutReal values.
```

```cpp
inline virtual bool is3D() const override
True if variable is 3D.
```

```cpp
inline virtual int byteSize() const override
Number of bytes for a single point.
```

```cpp
inline virtual int BoutRealSize() const override
Number of BoutReals (not implemented if not BoutReal)
```

```cpp
inline virtual void accept(FieldVisitor &v) override
Visitor pattern support.
```

```cpp
inline virtual void doneComms() override
```

```cpp
Field3D &calcParallelSlices()
```

```cpp
virtual void applyBoundary(bool init = false) override
```

```cpp
void applyBoundary(BoutReal t)
```

```cpp
void applyBoundary(const std::string &condition)
```

```cpp
inline void applyBoundary(const char *condition)
```

```cpp
void applyBoundary(const std::string &region, const std::string &condition)
```

```cpp
virtual void applyTDerivBoundary() override
```

```cpp
void setBoundaryTo(const Field3D &f3d)
Copy the boundary values half-way between cells This uses 2nd order central differences to set the value on the boundary to the value on the boundary in field f3d. Note: does not just copy values in boundary region.
```

```cpp
void applyParallelBoundary()
```

```cpp
void applyParallelBoundary(BoutReal t)
```

```cpp
void applyParallelBoundary(const std::string &condition)
```

```cpp
inline void applyParallelBoundary(const char *condition)
```

```cpp
void applyParallelBoundary(const std::string &region, const std::string &condition)
```
void applyParallelBoundary(const std::string &region, const std::string &condition, Field3D *f)

**Private Members**

const Field2D *background = {nullptr}

Boundary - add a 2D field.

int nx = {-1}

Array sizes (from fieldmesh). These are valid only if fieldmesh is not null.

int ny = {-1}

int nz = {-1}

Array<BoutReal> data

Internal data array. Handles allocation/freeing of memory.

Field3D *deriv = {nullptr}

Time derivative (may be nullptr)

std::vector<Field3D> yup_fields = {}

Fields containing values along Y.

std::vector<Field3D> ydown_fields = {}

**Friends**

friend class Vector3D

inline friend friend void swap (Field3D &first, Field3D &second) noexcept

**K.2.66 File field.cxx**

namespace bout

SNB model

**Functions**

CELL_LOC normaliseLocation(CELL_LOC location, Mesh *mesh)

Make sure location is a sensible value for mesh

Throws if checks are enabled and trying to use a staggered location on a non-staggered mesh
K.2.67 File field.hxx

field base class definition for differencing methods
Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu
Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

Defines

\texttt{FIELD\_FUNC(name, func)}

This macro takes a function \texttt{func}, which is assumed to operate on a single BoutReal and return a single BoutReal, and wraps it up into a function of a \texttt{Field} called \texttt{name}.

If \texttt{CHECK} \geq 1, checks if the \texttt{Field} is allocated

Loops over the entire domain, applies function, and uses checkData() to, if \texttt{CHECK} \geq 3, check result for non-finite numbers

Parameters

- \texttt{name} – The name of the function to define
- \texttt{func} – The function to apply to each value

Functions

\texttt{inline bool areFieldsCompatible(const Field &field1, const Field &field2)}

Check if Fields have compatible meta-data.

\texttt{template\textless T\textgreater \ inline T emptyFrom(const T &f)}

Return an empty shell field of some type derived from \texttt{Field}, with metadata copied and a data array that is allocated but not initialised.

\texttt{template\textless T\textgreater \ inline T zeroFrom(const T &f)}

Return a field of some type derived from \texttt{Field}, with metadata copied from another field and a data array allocated and initialised to zero.

\texttt{template\textless T\textgreater \ inline T filledFrom(const T &f, BoutReal fill_value)}

Return a field of some type derived from \texttt{Field}, with metadata copied from another field and a data array allocated and filled with the given value.
template<typename T, typename = bout::utils::EnableIfField<T>>
T operator+(const T &f)
   Unary + operator. This doesn’t do anything.

template<typename T>
inline T toFieldAligned(const T &f, const std::string &region = "RGN_ALL")

template<typename T>
inline T toFieldAligned(const T &f, REGION region)

template<typename T>
inline T fromFieldAligned(const T &f, const std::string &region = "RGN_ALL")

template<typename T>
inline T fromFieldAligned(const T &f, REGION region)

template<typename T, typename = bout::utils::EnableIfField<T>>
inline BoutReal min(const T &f, bool allpe = false, const std::string &rgn = "RGN_NOBNDRY")

template<typename T, typename = bout::utils::EnableIfField<T>>
inline BoutReal min(const T &f, bool allpe, REGION rgn)

template<typename T, typename = bout::utils::EnableIfField<T>>
inline BoutReal max(const T &f, bool allpe = false, const std::string &rgn = "RGN_NOBNDRY")

template<typename T, typename = bout::utils::EnableIfField<T>>
inline BoutReal max(const T &f, bool allpe, REGION rgn)

template<typename T, typename = bout::utils::EnableIfField<T>>
inline BoutReal mean(const T &f, bool allpe = false, const std::string &rgn = "RGN_NOBNDRY")

template<typename T, typename = bout::utils::EnableIfField<T>>
inline BoutReal mean(const T &f, bool allpe, REGION rgn)

template<typename T, typename = bout::utils::EnableIfField<T>>
T pow(const T &lhs, const T &rhs, const std::string &rgn = "RGN_ALL")
   Exponent: pow(lhs, rhs) is lhs raised to the power of rhs
   This loops over the entire domain, including guard/boundary cells by default (can be changed using the rgn argument) If CHECK >= 3 then the result will be checked for non-finite numbers

template<typename T, typename = bout::utils::EnableIfField<T>>
inline T pow(const T &lhs, const T &rhs, REGION rgn)

template<typename T, typename = bout::utils::EnableIfField<T>>
T pow(const T &lhs, BoutReal rhs, const std::string &rgn = "RGN_ALL")
inline \texttt{T pow}(\texttt{const T &}lhs, \texttt{BoutReal} rhs, \texttt{REGION} rgn)

\begin{verbatim}
template<typename T, typename = bout::utils::EnableIfField<T>>
T pow(BoutReal lhs, const T &rhs, const std::string &rgn = "RGN_ALL")
\end{verbatim}

template<typename T, typename = bout::utils::EnableIfField<T>>
inline T pow(BoutReal lhs, const T &rhs, \texttt{REGION} rgn)

\begin{verbatim}
template<typename T, typename =bout::utils::EnableIfField<T>>
inline T sqrt(const T &f, const std::string &rgn = "RGN_ALL")
\end{verbatim}

Square root of \(f\) over region \texttt{rgn}
This loops over the entire domain, including guard/boundary cells by default (can be changed using the \texttt{rgn} argument). If \texttt{CHECK} \(\geq 3\) then the result will be checked for non-finite numbers

\begin{verbatim}
template<typename T, typename =bout::utils::EnableIfField<T>>
inline T sqrt(const T &f, \texttt{REGION} region)
\end{verbatim}

\begin{verbatim}
template<typename T, typename =bout::utils::EnableIfField<T>>
inline T abs(const T &f, \texttt{REGION} region)
\end{verbatim}

Absolute value (modulus, |\texttt{f}|) of \(f\) over region \texttt{rgn}
This loops over the entire domain, including guard/boundary cells by default (can be changed using the \texttt{rgn} argument). If \texttt{CHECK} \(\geq 3\) then the result will be checked for non-finite numbers

\begin{verbatim}
template<typename T, typename =bout::utils::EnableIfField<T>>
inline T abs(const T &f, \texttt{REGION} region)
\end{verbatim}

\begin{verbatim}
template<typename T, typename =bout::utils::EnableIfField<T>>
inline T exp(const T &f, const std::string &rgn = "RGN_ALL")
\end{verbatim}

Exponential: \(\exp(f)\) is \(e\) to the power of \(f\), over region \texttt{rgn}
This loops over the entire domain, including guard/boundary cells by default (can be changed using the \texttt{rgn} argument). If \texttt{CHECK} \(\geq 3\) then the result will be checked for non-finite numbers

\begin{verbatim}
template<typename T, typename =bout::utils::EnableIfField<T>>
inline T exp(const T &f, \texttt{REGION} region)
\end{verbatim}

\begin{verbatim}
template<typename T, typename =bout::utils::EnableIfField<T>>
inline T log(const T &f, const std::string &rgn = "RGN_ALL")
\end{verbatim}

Natural logarithm of \(f\) over region \texttt{rgn}, inverse of exponential
\(\ln(\exp(f)) = f\)
This loops over the entire domain, including guard/boundary cells by default (can be changed using the \texttt{rgn} argument). If \texttt{CHECK} \(\geq 3\) then the result will be checked for non-finite numbers

\begin{verbatim}
template<typename T, typename =bout::utils::EnableIfField<T>>
inline T log(const T &f, \texttt{REGION} region)
\end{verbatim}

\begin{verbatim}
template<typename T, typename =bout::utils::EnableIfField<T>>
inline T sin(const T &f, const std::string &rgn = "RGN_ALL")
\end{verbatim}

Sine trigonometric function.
This loops over the entire domain, including guard/boundary cells by default (can be changed using the `rgn` argument). If CHECK >= 3 then the result will be checked for non-finite numbers

**Parameters**

- `f` – [in] Angle in radians
- `rgn` – [in] The region to calculate the result over

```cpp
template<typename T, typename = bout::utils::EnableIfField<T>>
inline T sin(const T &f, REGION region)
```

```cpp
template<typename T, typename = bout::utils::EnableIfField<T>>
inline T cos(const T &f, const std::string &rgn = "RGN_ALL")
```

Cosine trigonometric function.

This loops over the entire domain, including guard/boundary cells by default (can be changed using the `rgn` argument). If CHECK >= 3 then the result will be checked for non-finite numbers

**Parameters**

- `f` – [in] Angle in radians
- `rgn` – [in] The region to calculate the result over

```cpp
template<typename T, typename = bout::utils::EnableIfField<T>>
inline T cos(const T &f, REGION region)
```

```cpp
template<typename T, typename = bout::utils::EnableIfField<T>>
inline T tan(const T &f, const std::string &rgn = "RGN_ALL")
```

Tangent trigonometric function.

This loops over the entire domain, including guard/boundary cells by default (can be changed using the `rgn` argument). If CHECK >= 3 then the result will be checked for non-finite numbers

**Parameters**

- `f` – [in] Angle in radians
- `rgn` – [in] The region to calculate the result over

```cpp
template<typename T, typename = bout::utils::EnableIfField<T>>
inline T tan(const T &f, REGION region)
```

```cpp
template<typename T, typename = bout::utils::EnableIfField<T>>
inline T sinh(const T &f, const std::string &rgn = "RGN_ALL")
```

Hyperbolic sine trigonometric function.

This loops over the entire domain, including guard/boundary cells by default (can be changed using the `rgn` argument). If CHECK >= 3 then the result will be checked for non-finite numbers

**Parameters**

- `f` – [in] Angle in radians
- `rgn` – [in] The region to calculate the result over

```cpp
template<typename T, typename = bout::utils::EnableIfField<T>>
inline T sinh(const T &f, REGION region)
```
template<typename T, typename = bout::utils::EnableIfField<T>>
inline T sinh(const T &f, REGION region)

Hyperbolic cosine trigonometric function.
This loops over the entire domain, including guard/boundary cells by default (can be changed using the rgn argument). If CHECK >= 3 then the result will be checked for non-finite numbers

Parameters
- \( f \) – [in] Angle in radians
- \( rgn \) – [in] The region to calculate the result over

Hyperbolic tangent trigonometric function.
This loops over the entire domain, including guard/boundary cells by default (can be changed using the rgn argument). If CHECK >= 3 then the result will be checked for non-finite numbers

Parameters
- \( f \) – [in] Angle in radians
- \( rgn \) – [in] The region to calculate the result over

Check if all values of a field \( var \) are finite. Loops over all points including the boundaries by default (can be changed using the rgn argument)

Parameters
- \( var \) – [in] Variable to apply floor to
- \( f \) – [in] The floor value
• rgn – [in] The region to calculate the result over

template<typename T, typename = bout::utils::EnableIfField<T>>
inline T floor(const T &var, BoutReal f, REGION rgn)

class Field
#include <field.hxx> Base class for fields.
    Defines the virtual function SetStencil, used by differencing methods
    Subclassed by Field2D, Field3D, FieldPerp

Public Functions
Field() = default
Field(const Field &other) = default
Field(Field &&other) = default
Field &operator=(const Field &other) = default
Field &operator=(Field &&other) = default
virtual ~Field() = default
Field(Mesh *localmesh, CELL_LOC location_in, DirectionTypes directions_in)

void setLocation(CELL_LOC new_location)
Set variable location for staggered grids to

Parameters new_location – Throws BoutException if new_location is not CELL_CENTRE and
staggered grids are turned off and checks are on. If checks are off, silently sets location to
CELL_CENTRE instead.

CELL_LOC getLocation() const
Get variable location.

inline DirectionTypes getDirections() const
Getters for DIRECTION types.

inline YDirectionType getDirectionY() const

inline ZDirectionType getDirectionZ() const

inline void setDirectionY(YDirectionType y_type)
Setters for *DirectionType.

inline void setDirectionZ(ZDirectionType z_type)
inline virtual bool bndryValid()

inline Mesh *getMesh() const

Coordinates *getCoordinates() const
    Returns a pointer to the coordinates object at this field’s location from the mesh this field is on.

Coordinates *(CELL_LOC loc) const
    Returns a pointer to the coordinates object at the requested location from the mesh this field is on. If location
    is CELL_DEFAULT then return coordinates at field location

virtual int getNx() const
    Return the number of nx points

virtual int getNy() const
    Return the number of ny points

virtual int getNz() const
    Return the number of nz points

Public Members

std::string name

bool bndry_xin = {true}
    Status of the 4 boundaries.

bool bndry_xout = {true}
bool bndry_yup = {true}
bool bndry_ydown = {true}

Protected Functions

inline void copyFieldMembers(const Field &f)
    Copy the members from another Field.

Protected Attributes

Mesh *fieldmesh = {nullptr}

mutable std::shared_ptr<Coordinates> fieldCoordinates = {nullptr}

CELL_LOC location = {CELL_CENTRE}
    Location of the variable in the cell.

DirectionTypes directions = {YDirectionType::Standard, ZDirectionType::Standard}
    Labels for the type of coordinate system this field is defined over.
Friends

```cpp
inline friend friend void swap (Field &first, Field &second) noexcept
```

namespace bout
SNB model

Functions

```cpp
template<typename T>
inline void checkFinite(const T &f, const std::string &name = "field", const std::string &rgn = "RGN_ALL")
```

Check if all values of a field \emph{var} are finite. Loops over all points including the boundaries by default (can be changed using the \emph{rgn} argument) If any element is not finite, throws an exception that includes the position of the first found.

Note that checkFinite runs the check irrespective of CHECK level. It is intended to be used during initialization, where we always want to check inputs, even for optimized builds.

```cpp
template<typename T>
inline void checkPositive(const T &f, const std::string &name = "field", const std::string &rgn = "RGN_ALL")
```

Check if all values of a field \emph{var} are positive. Loops over all points including the boundaries by default (can be changed using the \emph{rgn} argument) If any element is not finite, throws an exception that includes the position of the first found.

Note that checkPositive runs the check irrespective of CHECK level. It is intended to be used during initialization, where we always want to check inputs, even for optimized builds.

K.2.68 File field_data.cxx

K.2.69 File field_data.hxx

Class inherited by any field wanting to use Communicator or Solver objects.

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.
Defines

__FIELD_DATA_H__

class FieldData
    #include <field_data.hxx> Interface used to access data in field classes.
    Used by communicator, solver and (soon) datafile classes to access internal data in a general way
    Subclassed by Field2D, Field3D, Vector2D, Vector3D

Public Functions

FieldData() = default

virtual ~FieldData()

virtual void accept(FieldVisitor &v) = 0

virtual bool isReal() const = 0
    Returns true if field consists of BoutReal values.

virtual bool is3D() const = 0
    True if variable is 3D.

virtual int byteSize() const = 0
    Number of bytes for a single point.

inline virtual int BoutRealSize() const
    Number of BoutReals (not implemented if not BoutReal)

inline virtual void doneComms()

void setBoundary(const std::string &name)
    Set the boundary conditions.

void copyBoundary(const FieldData &f)
    Copy the boundary conditions from another field.

inline virtual void applyBoundary(bool init = false)

inline virtual void applyTDerivBoundary()

void addBndryFunction(FuncPtr userfunc, BndryLoc location)

void addBndryGenerator(FieldGeneratorPtr gen, BndryLoc location)

FieldGeneratorPtr getBndryGenerator(BndryLoc location)
Protected Attributes

std::vector<BoundaryOp*> bndry_op
Boundary conditions.

bool boundaryIsCopy = {false}
True if bndry_op is a copy.

bool boundaryIsSet = {false}
Set to true when setBoundary called.

std::vector<BoundaryOpPar*> bndry_op_par
Boundary conditions.

std::map<BndryLoc, FieldGeneratorPtr> bndry_generator

K.2.70 File field_factory.cxx

Functions

FieldGeneratorPtr generator(BoutReal value)
Helper function to create a FieldValue generator from a BoutReal.

FieldGeneratorPtr generator(BoutReal *ptr)
Helper function to create a FieldValuePtr from a pointer to BoutReal.

K.2.71 File field_factory.hxx

Functions

FieldGeneratorPtr generator(BoutReal value)
Helper function to create a FieldValue generator from a BoutReal.

FieldGeneratorPtr generator(BoutReal *ptr)
Helper function to create a FieldValuePtr from a pointer to BoutReal.

class FieldFactory : public ExpressionParser

Public Functions

FieldFactory(Mesh *mesh = nullptr, Options *opt = nullptr)

~FieldFactory() override = default

Field2D create2D(const std::string &value, const Options *opt = nullptr, Mesh *m = nullptr, CELL_LOC loc = CELL_CENTRE, BoutReal t = 0.0) const
Create a 2D field by parsing a string and evaluating the expression using the given options opt, over Mesh m at time t. The resulting field is at cell location loc.
Field3D create3D(const std::string &value, const Options *opt = nullptr, Mesh *m = nullptr, CELL_LOC loc = CELL_CENTRE, BoutReal t = 0.0) const
Create a 3D field by parsing a string and evaluating the expression using the given options opt, over Mesh m at time t. The resulting field is at cell location loc.

FieldPerp createPerp(const std::string &value, const Options *opt = nullptr, Mesh *m = nullptr, CELL_LOC loc = CELL_CENTRE, BoutReal t = 0.0) const
Create a FieldPerp by parsing a string and evaluating the expression using the given options opt, over Mesh m at time t. The resulting field is at cell location loc.

FieldGeneratorPtr parse(const std::string &input, const Options *opt = nullptr) const
Parse a string into a tree of generators.

Field2D create2D(FieldGeneratorPtr generator, Mesh *m = nullptr, CELL_LOC loc = CELL_CENTRE, BoutReal t = 0.0) const
Create a 2D field from a generator, over a given mesh at a given cell location and time.

Field3D create3D(FieldGeneratorPtr generator, Mesh *m = nullptr, CELL_LOC loc = CELL_CENTRE, BoutReal t = 0.0) const
Create a 3D field from a generator, over a given mesh at a given cell location and time.

FieldPerp createPerp(FieldGeneratorPtr generator, Mesh *m = nullptr, CELL_LOC loc = CELL_CENTRE, BoutReal t = 0.0) const
Create a FieldPerp from a generator, over a given mesh at a given cell location and time.

void cleanCache()
clean the cache of parsed strings

Public Static Functions

static FieldFactory *get()
Get the Singleton object.

Protected Functions

virtual FieldGeneratorPtr resolve(std::string &name) const override
These functions called by the parser to resolve unknown symbols. This is used to enable options to be referred to in expressions.

Private Functions

const Options *findOption(const Options *opt, const std::string &name, std::string &val) const
Find an Options object which contains the given name.

Private Members

Mesh *fieldmesh
The default mesh for create functions.

bool transform_from_field_aligned = {true}
Should we transform input from field-aligned coordinates (if possible)?
mutable const Options *options
The default options used in resolve(), can be temporarily overridden in parse/create2D/create3D()

mutable std::list<std::string> lookup
Names currently being parsed.

mutable std::map<std::string, FieldGeneratorPtr> cache
Cache parsed strings so repeated evaluations don’t result in allocating more generators.

class FieldFunction : public FieldGenerator

Public Functions
FieldFunction() = delete

inline FieldFunction(FuncPtr userfunc)

inline virtual BoutReal generate(BoutReal x, BoutReal y, BoutReal z, BoutReal t) override
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

Private Members

FuncPtr func

class FieldNull : public FieldGenerator

Public Functions
FieldNull() = default

inline virtual BoutReal generate(BoutReal x, BoutReal y, BoutReal z, BoutReal t) override
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

inline virtual FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> args) override
Virtual constructor. Makes a copy of this FieldGenerator, initialised with the given list of arguments. It is up to the implementations to test whether the correct number of arguments is passed.

Parameters args – [in] A (possibly empty) list of arguments to the generator function
Public Static Functions

static inline FieldGeneratorPtr get() 
Singleton.

K.2.72 File field_visitor.hxx

class FieldVisitor

Public Functions

virtual void accept(Field2D &f) = 0
virtual void accept(FieldPerp &f) = 0
virtual void accept(Field3D &f) = 0
virtual void accept(Vector2D &f) = 0
virtual void accept(Vector3D &f) = 0

K.2.73 File fieldgenerators.cxx

K.2.74 File fieldgenerators.hxx

These classes are used by FieldFactory

Typedefs

using single_arg_op = BoutReal (*)(BoutReal)
Template class to define generators around a C function.

using double_arg_op = BoutReal (*)(BoutReal, BoutReal)
Template for a FieldGenerator with two input arguments.

class FieldValuePtr : public FieldGenerator
#include <fieldgenerators.hxx> Creates a Field Generator using a pointer to value WARNING: The value pointed to must remain in scope until this generator is finished
**Public Functions**

```cpp
inline FieldValuePtr(BoutReal *val)
```

```cpp
inline virtual FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> &args) override
Virtual constructor. Makes a copy of this FieldGenerator, initialised with the given list of arguments. It is up to the implementations to test whether the correct number of arguments is passed.
```

**Parameters**

- `args` – [in] A (possibly empty) list of arguments to the generator function

```cpp
inline virtual BoutReal generate(double x, double y, double z, double t) override
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs
```

**Private Members**

```cpp
BoutReal *ptr
```

**Class**

`FieldSin : public FieldGenerator
#include <fieldgenerators.hxx>` Sine function field generator.

**Public Functions**

```cpp
inline FieldSin(FieldGeneratorPtr g)
```

```cpp
virtual FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> &args) override
Virtual constructor. Makes a copy of this FieldGenerator, initialised with the given list of arguments. It is up to the implementations to test whether the correct number of arguments is passed.
```

**Parameters**

- `args` – [in] A (possibly empty) list of arguments to the generator function

```cpp
virtual BoutReal generate(double x, double y, double z, double t) override
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs
```

```cpp
inline virtual std::string str() const override
Create a string representation of the generator, for debugging output.
```

**Private Members**

```cpp
FieldGeneratorPtr gen
```

**Class**

`FieldCos : public FieldGenerator
#include <fieldgenerators.hxx>` Cosine function field generator.
**Public Functions**

inline `FieldCos(FieldGeneratorPtr g)`

virtual `FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> args)` override  
Virtual constructor. Makes a copy of this `FieldGenerator`, initialised with the given list of arguments. It is up to the implementations to test whether the correct number of arguments is passed.

**Parameters** `args` – [in] A (possibly empty) list of arguments to the generator function

virtual `BoutReal generate(double x, double y, double z, double t)` override  
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

inline virtual `std::string str()` const override  
Create a string representation of the generator, for debugging output.

**Private Members**

`FieldGeneratorPtr gen`

template<`single_arg_op Op>`  

class `FieldGenOneArg` : public `FieldGenerator`

**Public Functions**

inline `FieldGenOneArg(FieldGeneratorPtr g)`  
< Template for single-argument function

inline virtual `FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> args)` override  
Virtual constructor. Makes a copy of this `FieldGenerator`, initialised with the given list of arguments. It is up to the implementations to test whether the correct number of arguments is passed.

**Parameters** `args` – [in] A (possibly empty) list of arguments to the generator function

inline virtual `BoutReal generate(double x, double y, double z, double t)` override  
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

inline virtual `std::string str()` const override  
Create a string representation of the generator, for debugging output.

**Private Members**

`FieldGeneratorPtr gen`

template<`double_arg_op Op>`  

class `FieldGenTwoArg` : public `FieldGenerator`
# Public Functions

```cpp
inline FieldGenTwoArg(FieldGeneratorPtr a, FieldGeneratorPtr b)
< Template for two-argument function
```

```cpp
inline virtual FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> &args) override
Virtual constructor. Makes a copy of this FieldGenerator, initialised with the given list of arguments. It is
top to the implementations to test whether the correct number of arguments is passed.
```

**Parameters**
- `args` – [in] A (possibly empty) list of arguments to the generator function

```cpp
inline virtual BoutReal generate(double x, double y, double z, double t) override
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same
value given the same inputs
```

```cpp
inline virtual std::string str() const override
Create a string representation of the generator, for debugging output.
```

## Private Members

```cpp
FieldGeneratorPtr A
FieldGeneratorPtr B
```

class FieldATan : public FieldGenerator
```
#include <fieldgenerators.hxx> Arc (Inverse) tangent. Either one or two argument versions.
```

# Public Functions

```cpp
inline FieldATan(FieldGeneratorPtr a, FieldGeneratorPtr b = nullptr)
```

```cpp
inline virtual FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> &args) override
Virtual constructor. Makes a copy of this FieldGenerator, initialised with the given list of arguments. It is
top to the implementations to test whether the correct number of arguments is passed.
```

**Parameters**
- `args` – [in] A (possibly empty) list of arguments to the generator function

```cpp
inline virtual BoutReal generate(double x, double y, double z, double t) override
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same
value given the same inputs
```

## Private Members

```cpp
FieldGeneratorPtr A
FieldGeneratorPtr B
```

class FieldSinh : public FieldGenerator
```
#include <fieldgenerators.hxx> Hyperbolic sine function.
```
Public Functions

```
inline FieldSinh(FieldGeneratorPtr g)
```

erlines `FieldSinh` constructs a FieldSinh object.

```
virtual FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> args) override
```

Virtual constructor. Makes a copy of this FieldGenerator, initialised with the given list of arguments. It is up to the implementations to test whether the correct number of arguments is passed.

Parameters `args` – [in] A (possibly empty) list of arguments to the generator function

```
virtual BoutReal generate(double x, double y, double z, double t) override
```

Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

Private Members

```
FieldGeneratorPtr gen
```

class `FieldCosh` : public `FieldGenerator`

```
#include <fieldgenerators.hxx>
```

Hyperbolic cosine.

Public Functions

```
inline FieldCosh(FieldGeneratorPtr g)
```

```
virtual FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> args) override
```

Virtual constructor. Makes a copy of this FieldGenerator, initialised with the given list of arguments. It is up to the implementations to test whether the correct number of arguments is passed.

Parameters `args` – [in] A (possibly empty) list of arguments to the generator function

```
virtual BoutReal generate(double x, double y, double z, double t) override
```

Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

Private Members

```
FieldGeneratorPtr gen
```

class `FieldTanh` : public `FieldGenerator`

```
#include <fieldgenerators.hxx>
```

Hyperbolic tangent.

Public Functions

```
inline FieldTanh(FieldGeneratorPtr g = nullptr)
```

```
virtual FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> args) override
```

Virtual constructor. Makes a copy of this FieldGenerator, initialised with the given list of arguments. It is up to the implementations to test whether the correct number of arguments is passed.

Parameters `args` – [in] A (possibly empty) list of arguments to the generator function
virtual BoutReal generate(double x, double y, double z, double t) override
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

Private Members

FieldGeneratorPtr gen

class FieldGaussian : public FieldGenerator
#include <fieldgenerators.hxx> Gaussian distribution, taking mean and width arguments.

Public Functions

inline FieldGaussian(FieldGeneratorPtr xin, FieldGeneratorPtr sin)

virtual FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> args) override
Virtual constructor. Makes a copy of this FieldGenerator, initialised with the given list of arguments. It is up to the implementations to test whether the correct number of arguments is passed.

Parameters
args – [in] A (possibly empty) list of arguments to the generator function

virtual BoutReal generate(double x, double y, double z, double t) override
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

Private Members

FieldGeneratorPtr X
FieldGeneratorPtr s

class FieldAbs : public FieldGenerator
#include <fieldgenerators.hxx> Absolute value.

Public Functions

inline FieldAbs(FieldGeneratorPtr g)

virtual FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> args) override
Virtual constructor. Makes a copy of this FieldGenerator, initialised with the given list of arguments. It is up to the implementations to test whether the correct number of arguments is passed.

Parameters
args – [in] A (possibly empty) list of arguments to the generator function

virtual BoutReal generate(double x, double y, double z, double t) override
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs
Private Members

FieldGeneratorPtr gen

class FieldSqrt : public FieldGenerator
#include <fieldgenerators.hxx> Square root function.

Public Functions

inline FieldSqrt(FieldGeneratorPtr g)

virtual FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> args) override
Virtual constructor. Makes a copy of this FieldGenerator, initialised with the given list of arguments. It is
up to the implementations to test whether the correct number of arguments is passed.

Parameters args – [in] A (possibly empty) list of arguments to the generator function

virtual BoutReal generate(double x, double y, double z, double t) override
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same
value given the same inputs

Private Members

FieldGeneratorPtr gen

class FieldHeaviside : public FieldGenerator
#include <fieldgenerators.hxx> Heaviside function, switches between 0 and 1.

Public Functions

inline FieldHeaviside(FieldGeneratorPtr g)

virtual FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> args) override
Virtual constructor. Makes a copy of this FieldGenerator, initialised with the given list of arguments. It is
up to the implementations to test whether the correct number of arguments is passed.

Parameters args – [in] A (possibly empty) list of arguments to the generator function

virtual BoutReal generate(double x, double y, double z, double t) override
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same
value given the same inputs

inline virtual std::string str() const override
Create a string representation of the generator, for debugging output.
**Private Members**

`FieldGeneratorPtr gen`

class **FieldErf** : public `FieldGenerator`

```
#include <fieldgenerators.hxx> Generator for the error function erf.
```

**Public Functions**

```
inline `FieldErf` (FieldGeneratorPtr g)
```

virtual `FieldGeneratorPtr clone` (const `std::list<FieldGeneratorPtr>` args) override

Virtual constructor. Makes a copy of this `FieldGenerator`, initialised with the given list of arguments. It is up to the implementations to test whether the correct number of arguments is passed.

**Parameters**

`args` – [in] A (possibly empty) list of arguments to the generator function

virtual `BoutReal generate` (double x, double y, double z, double t) override

Generate a value at the given coordinates (x,y,z,t). This should be deterministic, always returning the same value given the same inputs

**Private Members**

`FieldGeneratorPtr gen`

class **FieldMin** : public `FieldGenerator`

```
#include <fieldgenerators.hxx> Minimum.
```

**Public Functions**

```
`FieldMin()` = default
```

```
inline `FieldMin` (const `std::list<FieldGeneratorPtr>` args)
```

```
inline virtual `FieldGeneratorPtr clone` (const `std::list<FieldGeneratorPtr>` args) override
```

Virtual constructor. Makes a copy of this `FieldGenerator`, initialised with the given list of arguments. It is up to the implementations to test whether the correct number of arguments is passed.

**Parameters**

`args` – [in] A (possibly empty) list of arguments to the generator function

```
inline virtual `BoutReal generate` (double x, double y, double z, double t) override
```

Generate a value at the given coordinates (x,y,z,t). This should be deterministic, always returning the same value given the same inputs
Private Members

`std::list<FieldGeneratorPtr> input`

class FieldMax : public FieldGenerator
#include <fieldgenerators.hxx> Maximum.

Public Functions

FieldMax() = default

`inline FieldMax(const std::list<FieldGeneratorPtr> args)`

inline virtual FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> args) override
Virtual constructor. Makes a copy of this FieldGenerator, initialised with the given list of arguments. It is up to the implementations to test whether the correct number of arguments is passed.

Parameters args – [in] A (possibly empty) list of arguments to the generator function

`inline virtual BoutReal generate(double x, double y, double z, double t)` override
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

Private Members

`std::list<FieldGeneratorPtr> input`

class FieldRound : public FieldGenerator
#include <fieldgenerators.hxx> Generator to round to the nearest integer.

Public Functions

`inline FieldRound(FieldGeneratorPtr g)`

inline virtual FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> args) override
Virtual constructor. Makes a copy of this FieldGenerator, initialised with the given list of arguments. It is up to the implementations to test whether the correct number of arguments is passed.

Parameters args – [in] A (possibly empty) list of arguments to the generator function

inline virtual BoutReal generate(double x, double y, double z, double t) override
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs
Private Members

FieldGeneratorPtr gen

class FieldBallooning : public FieldGenerator

Public Functions

inline FieldBallooning(Mesh *m, FieldGeneratorPtr a = nullptr, int n = 3)

virtual FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> args) override
    Virtual constructor. Makes a copy of this FieldGenerator, initialised with the given list of arguments. It is
    up to the implementations to test whether the correct number of arguments is passed.

    Parameters args – [in] A (possibly empty) list of arguments to the generator function

virtual BoutReal generate(double x, double y, double z, double t) override
    Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same
    value given the same inputs

Private Members

Mesh *mesh

FieldGeneratorPtr arg

int ball_n

class FieldMixmode : public FieldGenerator

Public Functions

FieldMixmode(FieldGeneratorPtr a = nullptr, BoutReal seed = 0.5)

virtual FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> args) override
    Virtual constructor. Makes a copy of this FieldGenerator, initialised with the given list of arguments. It is
    up to the implementations to test whether the correct number of arguments is passed.

    Parameters args – [in] A (possibly empty) list of arguments to the generator function

virtual BoutReal generate(double x, double y, double z, double t) override
    Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same
    value given the same inputs
Private Functions

`BoutReal genRand(BoutReal seed)`
Generate a random number between 0 and 1 (exclusive) given an arbitrary seed value

This PRNG has no memory, i.e. you need to call it with a different seed each time.

Private Members

`FieldGeneratorPtr arg`

`BoutReal phase[14]`

class `FieldTanhHat` : public `FieldGenerator`

Public Functions

inline `FieldTanhHat(FieldGeneratorPtr xin, FieldGeneratorPtr widthin, FieldGeneratorPtr centerin, FieldGeneratorPtr steepnessin)`

virtual `FieldGeneratorPtr clone(const std::list<FieldGeneratorPtr> args)` override
Virtual constructor. Makes a copy of this `FieldGenerator`, initialised with the given list of arguments. It is up to the implementations to test whether the correct number of arguments is passed.

Parameters `args` – [in] A (possibly empty) list of arguments to the generator function

virtual `BoutReal generate(double x, double y, double z, double t)` override
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

Private Members

`FieldGeneratorPtr X`

`FieldGeneratorPtr width`

`FieldGeneratorPtr center`

`FieldGeneratorPtr steepness`

K.2.75 File fieldgroup.cxx

Functions

`FieldGroup operator+(const FieldGroup &lhs, const FieldGroup &rhs)`
Combine two FieldGroups.
K.2.76 File fieldgroup.hxx

Functions

FieldGroup operator+(const FieldGroup &lhs, const FieldGroup &rhs)
    Combine two FieldGroups.

class FieldGroup
    #include <fieldgroup.hxx> Group together fields for easier communication

    Note: The FieldData class is used as a base class, which is inherited by Field2D, Field3D, Vector2D and Vector3D however Vector2D and Vector3D are stored by reference to their components (x,y,z) as Field2D or Field3D objects.

Public Types

    using iterator = std::vector<FieldData*>::iterator
        Iteration over all fields.

    using const_iterator = std::vector<FieldData*>::const_iterator
        Const iteration over all fields.

Public Functions

    FieldGroup() = default

    FieldGroup(const FieldGroup &other) = default

    FieldGroup(FieldGroup &&other) = default

    FieldGroup &operator=(const FieldGroup &other) = default

    FieldGroup &operator=(FieldGroup &&other) = default

    inline FieldGroup(FieldData &f)
        Constructor with a single FieldData f.

    inline FieldGroup(Field3D &f)
        Constructor with a single Field3D f.

    inline FieldGroup(Vector2D &v)
        Constructor with a single Vector2D v
        This is needed so that fvec only contains Field2D or Field3D

    inline FieldGroup(Vector3D &v)
        Constructor with a single Vector3D v
        This is needed so that fvec only contains Field2D or Field3D

    template<typename ...Ts>

inline explicit **FieldGroup**\((Ts\&...\ ts)\)
Variadic constructor. Allows an arbitrary number of **FieldData** arguments

The explicit keyword prevents **FieldGroup** being constructed with arbitrary types. In particular arguments to \(add()\) cannot be implicitly converted to **FieldGroup**, leading to an infinite loop.

inline void **add**\((\text{const } FieldGroup &\text{other})\)
Copy contents of another **FieldGroup** \(\text{other}\) into this group.

inline **FieldGroup &operator+=**\((\text{const } FieldGroup &\text{other})\)
Add the contents of \(\text{other}\) to this.

inline void **add**\((FieldData &f)\)
Add a **FieldData** \(f\) to the group.

A pointer to this field will be stored internally, so the lifetime of this variable should be longer than the lifetime of this group.

inline void **add**\((Field3D &f)\)

inline void **add**\((Vector2D &v)\)
Add a **Vector2D** \(v\) to the group.

Pointers to this vector’s components will be stored internally, so the lifetime of this variable should be longer than the lifetime of this group.

inline void **add**\((Vector3D &v)\)
Add a **Vector3D** \(v\) to the group.

Pointers to this vector’s components will be stored internally, so the lifetime of this variable should be longer than the lifetime of this group.

**template<typename ...Ts>**
inline void **add**\((FieldData &t, Ts\&...\ ts)\)
Add multiple fields to this group

This is a variadic template which allows **Field3D** objects to be treated as a special case. An arbitrary number of fields can be added.

**template<typename ...Ts>**
inline void **add**\((Field3D &t, Ts\&...\ ts)\)

**template<typename ...Ts>**
inline void **add**\((Vector3D &t, Ts\&...\ ts)\)

**template<typename ...Ts>**
inline void **add**\((Vector2D &t, Ts\&...\ ts)\)

**inline int size() const**
Return number of fields.

**inline int size_field3d() const**
Return number of Field3Ds.

**inline bool empty() const**
Test whether this group is empty.

**inline void clear()**
Remove all fields from this group.
inline `iterator` `begin()`

inline `iterator` `end()`

inline `const_iterator` `begin()` const

inline `const_iterator` `end()` const

inline const `std::vector<FieldData*> &get()` const

inline const `std::vector<Field3D*> &field3d()` const

Iteration over 3D fields.

void `makeUnique()`

Ensure that each field appears only once.

**Private Members**

`std::vector<FieldData*> fvec`

`std::vector<Field3D*> f3vec`

---

**K.2.77 File fieldperp.cxx**

**Functions**

`FieldPerp toFieldAligned(const FieldPerp &f, const std::string &region)`

`FieldPerp fromFieldAligned(const FieldPerp &f, const std::string &region)`

`FieldPerp operator-(const FieldPerp &f)`

Unary minus. Returns the negative of given field, iterates over whole domain including guard/boundary cells.

const `FieldPerp sliceXZ(const Field3D &f, int y)`

Create a `FieldPerp` by slicing a 3D field at a given y.

void `checkDataIsFiniteOnRegion(const FieldPerp &f, const std::string &region)`

void `checkData(const FieldPerp &f, const std::string &region)`

Check if the data is valid.

void `invalidateGuards(FieldPerp &var)`

Force guard cells of passed field `var` to NaN.

bool `operator==(const FieldPerp &a, const FieldPerp &b)`

Test if two fields are the same, by checking that they are defined at the same y-index, and if the minimum absolute difference between them is less than 1e-10

`std::ostream &operator<<(std::ostream &out, const FieldPerp &value)`

*Output a string describing a FieldPerp to a stream.*
K.2.78 File fieldperp.hxx

Functions

\texttt{FieldPerp operator+} (const \texttt{FieldPerp} &lhs, const \texttt{FieldPerp} &rhs)

\texttt{FieldPerp operator+} (const \texttt{FieldPerp} &lhs, const \texttt{Field3D} &rhs)

\texttt{FieldPerp operator+} (const \texttt{FieldPerp} &lhs, const \texttt{Field2D} &rhs)

\texttt{FieldPerp operator+} (const \texttt{FieldPerp} &lhs, \texttt{BoutReal} rhs)

\texttt{FieldPerp operator+} (\texttt{BoutReal} lhs, const \texttt{FieldPerp} &rhs)

\texttt{FieldPerp operator-} (const \texttt{FieldPerp} &lhs, const \texttt{FieldPerp} &rhs)

\texttt{FieldPerp operator-} (const \texttt{FieldPerp} &lhs, const \texttt{Field3D} &rhs)

\texttt{FieldPerp operator-} (const \texttt{FieldPerp} &lhs, const \texttt{Field2D} &rhs)

\texttt{FieldPerp operator-} (const \texttt{FieldPerp} &lhs, \texttt{BoutReal} rhs)

\texttt{FieldPerp operator-} (\texttt{BoutReal} lhs, const \texttt{FieldPerp} &rhs)

\texttt{FieldPerp operator*} (const \texttt{FieldPerp} &lhs, const \texttt{FieldPerp} &rhs)

\texttt{FieldPerp operator*} (const \texttt{FieldPerp} &lhs, const \texttt{Field3D} &rhs)

\texttt{FieldPerp operator*} (const \texttt{FieldPerp} &lhs, const \texttt{Field2D} &rhs)

\texttt{FieldPerp operator*} (const \texttt{FieldPerp} &lhs, \texttt{BoutReal} rhs)

\texttt{FieldPerp operator*} (\texttt{BoutReal} lhs, const \texttt{FieldPerp} &rhs)

\texttt{FieldPerp operator/} (const \texttt{FieldPerp} &lhs, const \texttt{FieldPerp} &rhs)

\texttt{FieldPerp operator/} (const \texttt{FieldPerp} &lhs, const \texttt{Field3D} &rhs)

\texttt{FieldPerp operator/} (const \texttt{FieldPerp} &lhs, const \texttt{Field2D} &rhs)

\texttt{FieldPerp operator/} (const \texttt{FieldPerp} &lhs, \texttt{BoutReal} rhs)

\texttt{FieldPerp operator/} (\texttt{BoutReal} lhs, const \texttt{FieldPerp} &rhs)
FieldPerp operator-(const FieldPerp &f)
   Unary minus. Returns the negative of given field, iterates over whole domain including guard/boundary cells.

const FieldPerp sliceXZ(const Field3D &f, int y)
   Create a FieldPerp by slicing a 3D field at a given y.

template<>
inline FieldPerp emptyFrom<FieldPerp>(const FieldPerp &f)
   Return an empty shell field of some type derived from Field, with metadata copied and a data array that is
   allocated but not initialised.

doCheckData(const FieldPerp &f, const std::string &region = "RGN_NOX")
   Check if the data is valid.

inline void doCheckData(const FieldPerp &f, REGION region)

doFvalidateGuards(FieldPerp &var)
   Force guard cells of passed field var to NaN.

template<>
inline std::string toString(const FieldPerp &val)
   toString template specialisation Defined in utils.hxx

bool operator==(const FieldPerp &a, const FieldPerp &b)
   Test if two fields are the same, by checking that they are defined at the same y-index, and if the minimum absolute
   difference between them is less than 1e-10

std::ostream &operator<<(std::ostream &out, const FieldPerp &value)
   Output a string describing a FieldPerp to a stream.

class FieldPerp : public Field
   #include <fieldperp.hxx> Represents a 2D field perpendicular to the magnetic field at a particular index in Y,
   which only varies in X-Z.
   Primarily used inside field solvers

Public Types

using ind_type = IndPerp

Public Functions

FieldPerp(Mesh *fieldmesh = nullptr, CELL_LOC location_in = CELL_CENTRE, int yindex_in = -1, 
   DirectionTypes directions_in = {YDirectionType::Standard, ZDirectionType::Standard})
   Constructor

FieldPerp(const FieldPerp &f) = default
   Copy constructor. After this the data will be shared (non unique)

FieldPerp(FieldPerp &&rhs) = default
   Move constructor

FieldPerp(BoutReal val, Mesh *localmesh = nullptr)
   Constructor. This creates a FieldPerp using the global Mesh pointer (mesh) allocates data, and assigns the
   value val to all points including boundary cells.
`FieldPerp`\( (\text{Array<BoutReal>} \text{ data, Mesh *} \text{fieldmesh, CELL\_LOC location\_in = \text{CELL\_CENTRE}, int yindex\_in = -1, DirectionTypes directions\_in = \{\text{YDirectionType::Standard, ZDirectionType::Standard}\})\)

Constructor from `Array` and `Mesh`

```cpp
~FieldPerp() override = default
```

Assignment operators

```cpp
FieldPerp &operator=(\text{const FieldPerp &rhs})
```

```cpp
FieldPerp &operator=(\text{FieldPerp &&rhs}) = default
```

```cpp
FieldPerp &operator=(\text{BoutReal rhs})
```

```cpp
\text{const Region<IndPerp>} &getRegion(\text{REGION region}) \text{ const}
```

Return a Region<IndPerp> reference to use to iterate over this field.

```cpp
\text{const Region<IndPerp>} &getRegion(\text{const std::string &region\_name}) \text{ const}
```

```cpp
\text{inline Region<IndPerp>::RegionIndices::const iterator \_begin() \text{ const}}
```

```cpp
\text{inline Region<IndPerp>::RegionIndices::const iterator \_end() \text{ const}}
```

```cpp
\text{inline BoutReal &operator[](\text{const IndPerp &d})}
```

```cpp
\text{inline const BoutReal &operator[](\text{const IndPerp &d}) \text{ const}}
```

```cpp
\text{inline BoutReal &operator[](\text{const Ind3D &d})}
```

```cpp
\text{inline const BoutReal &operator[](\text{const Ind3D &d}) \text{ const}}
```

```cpp
\text{inline int getIndex() \text{ const}}
```

Return the y index at which this field is defined. This value is local to each processor

```cpp
\text{int getGlobalIndex() \text{ const}}
```

Return the globally defined y index if it’s either an interior (grid) point, or a boundary point. Otherwise, return -1 to indicate a guard cell or an invalid value

```cpp
\text{inline FieldPerp &setIndex}(\text{int y})
```

Set the (local) y index at which this field is defined

This is used in arithmetic operations

```cpp
\text{FieldPerp &setIndexFromGlobal}(\text{int y\_global})
```

Set the (local) y index at which this field is defined from a globally defined y index

Only use the global y index if it’s either an interior (grid) point, or a boundary point. Otherwise, sets yindex to -1 to indicate a guard cell or an invalid value

```cpp
\text{inline FieldPerp &setLocation(CELL\_LOC new\_location)}
```

```cpp
\text{inline FieldPerp &setDirectionY(YDirectionType d)}
```

K.2. File list
FieldPerp &allocate()
Ensure that data array is allocated and unique

inline bool isAllocated() const
True if the underlying data array is allocated.

inline const BoutReal *operator[](int jx) const

inline BoutReal *operator[](int jx)
Returns a C-style array (pointer to first element) in Z at a given X index. Used mainly for FFT routines

inline BoutReal &operator()(int jx, int jz)
Access to the underlying data array at a given x,z index

    If CHECK > 2 then bounds checking is performed, otherwise no checks are performed

inline const BoutReal &operator()(int jx, int jz) const
Const (read-only) access to the underlying data array.

inline BoutReal &operator()(int jx, int jy, int jz)
Access to the underlying data array. (X,Y,Z) indices for consistency with other field types

inline const BoutReal &operator()(int jx, int jy, int jz) const

FieldPerp &operator+=(const FieldPerp &rhs)
Addition, modifying in-place. This loops over the entire domain, including guard/boundary cells

FieldPerp &operator+=(const Field3D &rhs)

FieldPerp &operator+=(const Field2D &rhs)

FieldPerp &operator+=(BoutReal rhs)

FieldPerp &operator-=(const FieldPerp &rhs)
Subtraction, modifying in-place. This loops over the entire domain, including guard/boundary cells

FieldPerp &operator-=(const Field3D &rhs)

FieldPerp &operator-=(const Field2D &rhs)

FieldPerp &operator-=(BoutReal rhs)

FieldPerp &operator*=(const FieldPerp &rhs)
Multiplication, modifying in-place. This loops over the entire domain, including guard/boundary cells

FieldPerp &operator*=(const Field3D &rhs)

FieldPerp &operator*=(const Field2D &rhs)

FieldPerp &operator*=(BoutReal rhs)

FieldPerp &operator/=(const FieldPerp &rhs)
Division, modifying in-place. This loops over the entire domain, including guard/boundary cells
FieldPerp &operator/=(const Field3D &rhs)

FieldPerp &operator/=(const Field2D &rhs)

FieldPerp &operator/=(BoutReal rhs)

inline virtual int getNx() const override
    Return the number of nx points
inline virtual int getNy() const override
    Return the number of ny points
inline virtual int getNz() const override
    Return the number of nz points

Private Members

int yindex = {-1}
    The Y index at which this FieldPerp is defined.
int nx = {-1}
    The size of the data array.
int nz = {-1}

Array<BoutReal> data
    The underlying data array.

K.2.79 File format.hxx

Defines

BOUT_FORMAT_ARGS(i, j)
    Tell GCC that a function has a printf-style like argument. The first argument is the position of format string, and the second is the position of the first variadic argument. Note that it seems to start counting from 1, and also counts a *this pointer, as the first argument, so often 2 would be the first argument.

K.2.80 File formatfactory.cxx

K.2.81 File formatfactory.hxx

class FormatFactory : private Uncopyable
Public Functions

`std::unique_ptr<DataFormat> createDataFormat(const char *filename = nullptr, bool parallel = true, Mesh *mesh_in = nullptr)`

Public Static Functions

static `FormatFactory *getInstance()`

Return a pointer to the only instance.

Private Functions

`int matchString(const char *str, int n, const char **match)`

Private Static Attributes

static `FormatFactory *instance = nullptr`

The only instance of this class (Singleton)

K.2.82 File fv Ops.cxx

namespace FV

K.2.83 File fv_ops.hxx

namespace FV

Functions

const `Field3D Div_a_Laplace_perp(const Field3D &a, const Field3D &x)`

Div (a Laplace_perp(x)) Vorticity

const `Field3D Div_par_K_Grad_par(const Field3D &k, const Field3D &f, bool bndry_flux = true)`

Divergence of a parallel diffusion Div(k * Grad_par(f))

const `Field3D D4DY4(const Field3D &d, const Field3D &f)`

4th-order derivative in Y, using derivatives on cell boundaries.

A one-sided 3rd-order derivative, given a value at a boundary is:

\[
\frac{d^3f}{dx^3} = 16/5 \cdot f_b - 6 \cdot f_0 + 4 \cdot f_1 - 6/5 \cdot f_2
\]

where \( f_b \) is the value on the boundary; \( f_0 \) is the cell to the left of the boundary; \( f_1 \) to the left of \( f_0 \) and \( f_2 \) to the left of \( f_1 \)

\( f_2 \mid f_1 \mid f_0 \mid f_b \)

NB: Uses to/from FieldAligned coordinates
const Field3D D4DY4_Index(const Field3D &f, bool bndry_flux = true)
   4th-order dissipation term
   A one-sided 3rd-order derivative, given a value at a boundary is:
   \[
d\frac{d^3f}{dx^3} \approx \frac{16}{5} f_b - 6 f_0 + 4 f_1 - \frac{6}{5} f_2
   \]
   where \( f_b \) is the value on the boundary; \( f_0 \) is the cell to the left of the boundary; \( f_1 \) to the left of \( f_0 \) and \( f_2 \) to the left of \( f_1 \)
   f_2 | f_1 | f_0 | f_b

void communicateFluxes(Field3D &f)
   Communicate fluxes between processors. Takes values in guard cells, and adds them to cells

template<typename CellEdges = MC>
const Field3D Div_par(const Field3D &f_in, const Field3D &v_in, const Field3D &wave_speed_in, bool fixflux = true)
   Finite volume parallel divergence
   Preserves the sum of \( f*J*dx*dy*dz \) over the domain

NB: Uses to/from FieldAligned coordinates

Parameters
   - \texttt{f\_in} – \texttt{[in]} The field being advected. This will be reconstructed at cell faces using the given CellEdges method
   - \texttt{v\_in} – \texttt{[in]} The advection velocity. This will be interpolated to cell boundaries using linear interpolation
   - \texttt{wave\_speed\_in} – \texttt{[in]} Local maximum speed of all waves in the system at each
   - \texttt{fixflux} – \texttt{[in]} Fix the flux at the boundary to be the value at the midpoint (for boundary conditions)

template<typename CellEdges = MC>
const Field3D Div\_f\_v(const Field3D &n_in, const Vector3D &v, bool bndry_flux)
   \( \text{Div} ( n * v ) \) Magnetic drifts
   This uses the expression
   \[
   \text{Div} ( A ) = 1/J \cdot \frac{d}{di} ( J * A^i )
   \]
   Hence the input vector should be contravariant
   Note: Uses to/from FieldAligned

struct Fromm
   \#include \texttt{<fv\_ops.hxx>} Fromm method
Public Functions

inline void operator() (StencilID &n)

struct MC
#include <fv_ops.hxx> Monotonised Central (MC) second order slope limiter (Van Leer)
Limits the slope based on taking the slope with the minimum absolute value from central, 2*left and 2*right. If any of these slopes have different signs then the slope reverts to zero (i.e. 1st-order upwinding).

Public Functions

inline void operator() (StencilID &n)

Private Functions

inline BoutReal minmod (BoutReal a, BoutReal b, BoutReal c)

struct MinMod
#include <fv_ops.hxx> Second order slope limiter method
Limits slope to minimum absolute value of left and right gradients. If at a maximum or minimum slope set to zero, i.e. reverts to first order upwinding

Public Functions

inline void operator() (StencilID &n)

Private Functions

inline BoutReal _minmod (BoutReal a, BoutReal b)
Internal helper function for minmod slope limiter
If the inputs have different signs then returns zero, otherwise chooses the value with the minimum magnitude.

struct Stencil1D
#include <fv_ops.hxx> Stencil used for Finite Volume calculations which includes cell face values L and R
**Public Members**

```cpp
BoutReal c
BoutReal m
BoutReal p
BoutReal mm
BoutReal pp
BoutReal L
BoutReal R
```

```cpp
struct Upwind
#include <fv_ops.hxx> First order upwind for testing
```

**Public Functions**

```cpp
inline void operator()(Stencil1D &n)
```

### K.2.84 File gen_fieldops.py

```python
module gen_fieldops

Code-generator for arithmetic operators on Field2Ds/Field3Ds

This uses the jinja template in gen_fieldops.jinja to generate code
for the arithmetic operators, and prints to stdout.

The `Field` class provides some helper functions for determining how to
pass a variable by reference or pointer, and how to name arguments in
function signatures. This allows us to push some logic into the
templates themselves.
```

**Functions**

```python
smart_open(filename, mode='r')
```

Open stdin or stdout using a contextmanager

**From:** http://stackoverflow.com/a/29824059/2043465

```python
returnType(f1, f2)
```

Determine a suitable return type, by seeing which field is 'larger'.
Variables

operators = OrderedDict([('*', 'multiplication'), ('/', 'division'), ('+', 'addition'), ('-', 'subtraction')])

header = """// This file is autogenerated - see gen_fieldops.py#include <bout/mesh.hxx>#include <bout/region.hxx>#include <field2d.hxx>#include <field3d.hxx>#include <globals.hxx>#include <interpolation.hxx>"

parser = argparse.ArgumentParser(description="Generate code for the Field arithmetic operators")
default help action False dest

args = parser.parse_args()

index_var = 'index'
jz_var = 'jz'
mixed_base_ind_var = "base_ind"

region_name = "RGN_ALL"

region_loop = 'BOUT_FOR_SERIAL'

field3D = Field('Field3D', ['x', 'y', 'z'], index_var=index_var,jz_var = jz_var,
mixed_base_ind_var = mixed_base_ind_var)

field2D = Field('Field2D', ['x', 'y'], index_var=index_var,jz_var = jz_var,
mixed_base_ind_var = mixed_base_ind_var)

fieldPerp = Field('FieldPerp', ['x', 'z'], index_var=index_var,jz_var = jz_var,
mixed_base_ind_var = mixed_base_ind_var)

boutreal = Field('BoutReal', [], index_var=index_var,jz_var = jz_var,
mixed_base_ind_var = mixed_base_ind_var)

fields = [field3D, field2D, fieldPerp, boutreal]

env = jinja2.Environment(loader=jinja2.FileSystemLoader('.'),trim_blocks=True)
template = env.get_template("gen_fieldops.jinja")
rhs = copy(rhs)
lhs = copy(lhs)

out = returnType(rhs, lhs)

name

template_args = {'operator': operator,'operator_name': operator_name,'#'out': out,'lhs': lhs,'rhs': rhs,'#'region_loop': region_loop,'region_name': region_name,'#'index_var': index_var,'mixed_base_ind': mixed_base_ind_var,'jz_var': jz_var,}
Abstracts over BoutReals and Field2D/3D/Perps
Provides some helper functions for writing function signatures and passing data

Public Functions

__init__(self, field_type, dimensions, name=None, index_var=None, jz_var='jz',
mixed_base_ind_var='base_ind')

passByReference(self)

Returns "Type& name", except if field_type is BoutReal, in which case just returns "Type name"

index(self)

Returns "[[index_var]]", except if field_type is BoutReal, in which case just returns ""

mixed_index(self)

Returns "[[index_var] + {jz_var}]", if field_type is Field3D, self.index if Field2D or just returns "" for BoutReal

base_index(self)

Returns "[[mixed_base_ind_var]]", if field_type is Field3D, Field2D or .
._FieldPerp
or just returns "" for BoutReal

__eq__(self, other)
__ne__(self, other)
__repr__(self)
__str__(self)

Public Members

field_type
dimensions
name
index_var
jz_var
mixed_base_ind_var
region_type

K.2.85 File generated_fieldopscxx

Functions

Field3D operator*(const Field3D &lhs, const Field3D &rhs)

Field3D operator/(const Field3D &lhs, const Field3D &rhs)

Field3D operator+(const Field3D &lhs, const Field3D &rhs)

Field3D operator-(const Field3D &lhs, const Field3D &rhs)

Field3D operator*(const Field3D &lhs, const Field2D &rhs)

Field3D operator/(const Field3D &lhs, const Field2D &rhs)

Field3D operator+(const Field3D &lhs, const Field2D &rhs)

Field3D operator-(const Field3D &lhs, const Field2D &rhs)

FieldPerp operator*(const Field3D &lhs, const FieldPerp &rhs)

FieldPerp operator/(const Field3D &lhs, const FieldPerp &rhs)

FieldPerp operator+(const Field3D &lhs, const FieldPerp &rhs)

FieldPerp operator-(const Field3D &lhs, const FieldPerp &rhs)

Field3D operator*(const Field3D &lhs, const BoutReal rhs)

Field3D operator/(const Field3D &lhs, const BoutReal rhs)

Field3D operator+(const Field3D &lhs, const BoutReal rhs)

Field3D operator-(const Field3D &lhs, const BoutReal rhs)

Field3D operator*(const Field2D &lhs, const Field3D &rhs)

Field3D operator/(const Field2D &lhs, const Field3D &rhs)
Field3D operator+(const Field2D &lhs, const Field3D &rhs)

Field3D operator-(const Field2D &lhs, const Field3D &rhs)

Field2D operator*(const Field2D &lhs, const Field2D &rhs)

Field2D operator/(const Field2D &lhs, const Field2D &rhs)

Field2D operator+(const Field2D &lhs, const Field2D &rhs)

Field2D operator-(const Field2D &lhs, const Field2D &rhs)

FieldPerp operator*(const Field2D &lhs, const FieldPerp &rhs)

FieldPerp operator/(const Field2D &lhs, const FieldPerp &rhs)

FieldPerp operator+(const Field2D &lhs, const FieldPerp &rhs)

FieldPerp operator-(const Field2D &lhs, const FieldPerp &rhs)

Field2D operator*(const Field2D &lhs, const BoutReal rhs)

Field2D operator/(const Field2D &lhs, const BoutReal rhs)

Field2D operator+(const Field2D &lhs, const BoutReal rhs)

Field2D operator-(const Field2D &lhs, const BoutReal rhs)

FieldPerp operator*(const FieldPerp &lhs, const Field3D &rhs)

FieldPerp operator/(const FieldPerp &lhs, const Field3D &rhs)

FieldPerp operator+(const FieldPerp &lhs, const Field3D &rhs)

FieldPerp operator-(const FieldPerp &lhs, const Field3D &rhs)

FieldPerp operator*(const FieldPerp &lhs, const Field2D &rhs)

FieldPerp operator/(const FieldPerp &lhs, const Field2D &rhs)

FieldPerp operator+(const FieldPerp &lhs, const Field2D &rhs)
FieldPerp operator- (const FieldPerp &lhs, const Field2D &rhs)

FieldPerp operator* (const FieldPerp &lhs, const FieldPerp &rhs)

FieldPerp operator/ (const FieldPerp &lhs, const FieldPerp &rhs)

FieldPerp operator+ (const FieldPerp &lhs, const FieldPerp &rhs)

FieldPerp operator- (const FieldPerp &lhs, const FieldPerp &rhs)

FieldPerp operator* (const FieldPerp &lhs, const BoutReal rhs)

FieldPerp operator/ (const FieldPerp &lhs, const BoutReal rhs)

FieldPerp operator+ (const FieldPerp &lhs, const BoutReal rhs)

FieldPerp operator- (const FieldPerp &lhs, const BoutReal rhs)

Field3D operator* (const BoutReal lhs, const Field3D &rhs)

Field3D operator/ (const BoutReal lhs, const Field3D &rhs)

Field3D operator+ (const BoutReal lhs, const Field3D &rhs)

Field3D operator- (const BoutReal lhs, const Field3D &rhs)

Field2D operator* (const BoutReal lhs, const Field2D &rhs)

Field2D operator/ (const BoutReal lhs, const Field2D &rhs)

Field2D operator+ (const BoutReal lhs, const Field2D &rhs)

Field2D operator- (const BoutReal lhs, const Field2D &rhs)

FieldPerp operator* (const BoutReal lhs, const FieldPerp &rhs)

FieldPerp operator/ (const BoutReal lhs, const FieldPerp &rhs)

FieldPerp operator+ (const BoutReal lhs, const FieldPerp &rhs)

FieldPerp operator- (const BoutReal lhs, const FieldPerp &rhs)
**K.2.86 File generic_factory.hxx**

Defines

```cpp
__BOUT_GENERIC_FACTORY_H__
```

template<typename BaseType, typename TypeCreator = std::function<BaseType*>(>)
class Factory
#include <generic_factory.hxx> Generic Factory, adapted from Modern C++ Design/Loki by A. Alexandrescu

Use with RegisterInFactory to provide a generic way of creating new derived types at runtime

Example:

```cpp
class Base {}
class Derived : public Base {}
RegisterInFactory<Base, Derived>::register("derived_type");
auto foo = Factory<Base>::getInstance().create("derived_type");
```

TODO: Use std::unique_ptr<BaseType> instead of BaseType*

MIT Licence

tparam BaseType The base class that this factory creates

tparam TypeCreator The function signature for creating a new BaseType

**Public Functions**

inline virtual bool add(const std::string &name, TypeCreator creator)

Add a new type name to the factory

Parameters

- name – [in] An identifier for this type
- creator – [in] A function for creating this type

Returns true if the type was successfully added

inline virtual bool remove(const std::string &name)

Remove a type name from the factory

Parameters name – [in] The identifier for the type to be removed

Returns true if the type was successfully removed

```cpp
template<typename ...Args>
inline BaseType *create(const std::string &name, Args&&... args)
```

Create a new object of type name

Parameters name – [in] The identifier for the type to be created

Returns a pointer to the new object

```cpp
inline virtual std::vector<std::string> listAvailable()
```

List available types that can be created

Returns a vector of std::string
Public Static Functions

static inline Factory &getInstance()  
Get the singleton instance.

Protected Functions

Factory() = default

Protected Attributes

std::map<std::string, TypeCreator> type_map

template<typename BaseType, typename DerivedType>
class RegisterInFactory

#include <generic_factory.hxx> Helper class for adding new types to Factory

Example:

class Base {};  
class Derived : public Base {};  
RegisterInFactory<Base, Derived> register("derived_type");  
auto foo = Factory<Base>::getInstance().create("derived_type");

Adapted from http://www.drdobbs.com/conversations-abstract-factory-template/184403786

tparam BaseType Which factory to add DerivedType to

tparam DerivedType The new type to add to Factory<BaseType>

Public Functions

inline RegisterInFactory(const std::string &name)

K.2.87 File gettext.hxx

Defines

_(string)  
Support for i18n using GNU gettext.
class GlobalField

#include <globalfield.hxx>  This provides a method for gathering and scattering a field which takes into account
the local and global indices

This is a base class which is inherited by GlobalField2D and GlobalField3D

Subclassed by GlobalField2D, GlobalField3D

Public Functions

GlobalField() = delete

virtual ~GlobalField() = default

virtual bool valid() const = 0
   Is the data valid on any processor?

inline bool dataIsLocal() const
   Data is on this processor.

inline BoutReal & operator()(int jx, int jy, int jz)
   Data access by index. This doesn’t perform any checks, so the user should first test if the data is available
   on this processor by calling dataIsLocal()

inline const BoutReal & operator()(int jx, int jy, int jz) const
   Const data access by index.

inline int xSize() const
   Size of the field in X.

inline int ySize() const
   Size of the field in Y.

inline int zSize() const
   Size of the field in Z.

inline Array<BoutReal> & getData()

Protected Functions

GlobalField(Mesh *m, int proc, int xsize, int ysize, int zsize)

void proc_local_origin(int proc, int *x, int *y, int *z = nullptr) const

void proc_origin(int proc, int *x, int *y, int *z = nullptr) const
   Return the global origin of processor proc.

void proc_size(int proc, int *lx, int *ly, int *lz = nullptr) const
   Return the array size of processor proc.
Protected Attributes

Mesh *mesh
   The mesh we’re gathering/scattering over.

int data_on_proc
   Which processor is this data on?

int nx
int ny
int nz
   Global field sizes.

Array<BoutReal> data
   The global data, if on this processor.

MPI_Comm comm
   Communicator for all mesh.

int npes
int mype
   Number of MPI processes, this processor index.

class GlobalField2D : public GlobalField
    #include <globalfield.hxx> Gather and scatter a Field2D

Example
To create a GlobalField2D, pass a mesh pointer

GlobalField2D g2d(mesh);

By default data is gathered and scattered to/from processor 0. To change this, pass the processor number as a second argument:

GlobalField3D g2d(mesh, 1); // Gather onto processor 1

Gather and scatter methods operate on Field2D objects:

Field2D localdata;
g2d.gather(localdata); // Gather onto one processor

To scatter data back, use the scatter method:

localdata = g2d.scatter();

Note that both gather and scatter are collective operations, which must be performed by all processors.
To test if the data is available on a processor, use:
if (g2d.dataIsLocal()) {
    // g2d data on this processor
}

The data in a `GlobalField2D` can be accessed using (x,y,z) indexing, with the index ranges given by xSize, ySize, zSize methods.

```c
for ( int x=0; x<g2d.xSize(); x++)
    for ( int y=0; y<g2d.ySize(); y++)
        output.write(" Value at (%d,%d) is %e
",
                x, y,
                g2d(x, y));
```

## Public Functions

`GlobalField2D()` = delete
Can’t be constructed without args.

`GlobalField2D(Mesh *mesh, int proc = 0)`
Construct, giving a mesh and an optional processor

Parameters

- **mesh** – [in] The mesh to gather over
- **proc** – [in] The processor index where everything will be gathered/scattered to/from

`~GlobalField2D()` override
Destructor.

`inline virtual bool valid()` const override
Is the data valid and on this processor?

`void gather(const Field2D &f)`
Gather all data onto one processor.

`const Field2D scatter()` const
Scatter data back from one to many processors.

`inline GlobalField2D &operator=(const Field2D &rhs)`
Assignment from a 2D field. Shorthand for a gather, and must be called on all processors The scatter assignment operator needs to be a member of `Field2D`.

`inline BoutReal &operator()(int jx, int jy)`
Data access by global index.

`inline const BoutReal &operator()(int jx, int jy) const`
Private Functions

```c
int msg_len(int proc) const
    The length of message (in BoutReals) to be sent to or from processor proc
```

**Parameters**
- `proc` – [in] MPI processor index

Private Members

```c
BoutReal **buffer
    Buffer for sending and receiving. First index is the processor index, and second is the data

bool data_valid
    Is the data valid and on this processor?
```

class GlobalField3D : public GlobalField
#include <globalfield.hxx> Gather and scatter a Field3D to/from one processor

*Example*

To create a `GlobalField3D`, pass a mesh pointer

```c
GlobalField3D g3d(mesh);
```

By default data is gathered and scattered to/from processor 0. To change this, pass the processor number as a second argument:

```c
GlobalField3D g3d(mesh, 1); // Gather onto processor 1
```

Gather and scatter methods operate on `Field3D` objects:

```c
Field3D localdata;

g3d.gather(localdata); // Gather onto one processor
```

To scatter data back, use the scatter method:

```c
localdata = g3d.scatter();
```

Note that both gather and scatter are collective operations, which must be performed by all processors.

To test if the data is available on a processor, use:

```c
if(g3d.dataIsLocal()) {
    // g3d data on this processor
}
```

The data in a `GlobalField3D` can be accessed using (x,y,z) indexing, with the index ranges given by xSize, ySize, zSize methods.

```c
for ( int x=0; x<g3d.xSize(); x++)
    for ( int y=0; y<g3d.ySize(); y++)
        for ( int z=0; z<g3d.zSize(); z++)
            output.write(" Value at (%d ,%d ,%d) is %e\n" ,
```

(continues on next page)
Public Functions

**GlobalField3D() = delete**
Can’t be constructed without args.

**GlobalField3D(Mesh *mesh, int proc = 0)**
Construct, giving a mesh and an optional processor

- **Parameters**
  - `mesh` – [in] The mesh to gather over
  - `proc` – [in] The processor index where everything will be gathered/scattered to/from

~~GlobalField3D() override~~
Destructor.

**inline virtual bool valid() const override**
Test if the data is valid i.e. has been allocated.

**void gather(const Field3D &f)**
Gather all data onto one processor.

**const Field3D scatter() const**
Scatter data back from one to many processors.

**inline GlobalField3D &operator=(const Field3D &rhs)**
Assignment from a 2D field. Shorthand for a gather, and must be called on all processors The scatter assignment operator needs to be a member of `Field2D`.

Private Functions

**int msg_len(int proc) const**
The length of message (in `BoutReals`) to be sent to or from processor `proc`

- **Parameters**
  - `proc` – [in] MPI processor index

Private Members

`BoutReal **buffer`
Buffer for sending and receiving. First index is the processor index, and second is the data

`bool data_valid`
Is the data valid and on this processor?
K.2.90 File globals.hxx

Defines

GLOBAL

SETTING(name, val)

GRID_LOAD1(var)
    Define for reading a variable from the grid.

GRID_LOAD2(var1, var2)

GRID_LOAD3(var1, var2, var3)

GRID_LOAD4(var1, var2, var3, var4)

GRID_LOAD5(var1, var2, var3, var4, var5)

GRID_LOAD6(var1, var2, var3, var4, var5, var6)

GRID_LOAD(...)  
    Read fields from the global mesh The name of the variable will be used as the name in the input. This should accept up to 10 arguments

namespace bout
    SNB model

namespace globals

    Variables

    Dump file object.

K.2.91 File griddata.hxx

class GridDataSource
    #include <griddata.hxx> Interface class to serve grid data.
    Provides a generic interface for sources of equilibrium data. Could be used to simplify interfacing between BOUT++ and other codes
    Subclassed by GridFile, GridFromOptions
Public Types

enum Direction
{
  enumerator X,
  enumerator Y,
  enumerator Z
};

Public Functions

inline GridDataSource(const bool source_is_file = false)

virtual ~GridDataSource() = default

virtual bool hasVar(const std::string &name) = 0
  Test if source can supply a variable.

virtual bool get(Mesh *m, std::string &sval, const std::string &name, const std::string &def = ") = 0
  Get a string.

virtual bool get(Mesh *m, int &ival, const std::string &name, int def = 0) = 0
  Get an integer.

virtual bool get(Mesh *m, BoutReal &rval, const std::string &name, BoutReal def = 0.0) = 0
  Get a BoutReal number.

virtual bool get(Mesh *m, Field2D &var, const std::string &name, BoutReal def = 0.0) = 0

virtual bool get(Mesh *m, Field3D &var, const std::string &name, BoutReal def = 0.0) = 0

virtual bool get(Mesh *m, FieldPerp &var, const std::string &name, BoutReal def = 0.0) = 0

virtual bool get(Mesh *m, std::vector<int> &var, const std::string &name, int len, int offset = 0, Direction dir = GridDataSource::X) = 0

virtual bool get(Mesh *m, std::vector<BoutReal> &var, const std::string &name, int len, int offset = 0, Direction dir = GridDataSource::X) = 0

virtual bool hasXBoundaryGuards(Mesh *m) = 0
  Are x-boundary guard cells read from the source?

virtual bool hasYBoundaryGuards() = 0
  Are y-boundary guard cells read from the source?
Public Members

const bool is_file
   Is the data source a grid file?

Public Static Attributes

static constexpr Direction X = Direction::X
static constexpr Direction Y = Direction::Y
static constexpr Direction Z = Direction::Z

class GridFile : public GridDataSource
#include <griddata.hxx> Interface to grid data in a file.
   This is a thin wrapper around a DataFormat object. Only needs to implement reading routines.

Public Functions

GridFile() = delete

GridFile(std::unique_ptr<DataFormat> format, std::string gridfilename)
   Creates a GridFile object
   format Pointer to DataFormat. This will be deleted in destructor

~GridFile() override

virtual bool hasVar(const std::string &name) override
   Tests whether a variable exists in the file
   Currently this is done by getting the variable’s size, and testing for zero size.

virtual bool get(Mesh *m, std::string &sval, const std::string &name, const std::string &def = "") override
   Get a string.
   Read a string from file. If the string is not found, then string is set to "" and false is returned.

   Inputs
   m Pointer to mesh, not used name String containing name of variable

   Outputs
   sval Reference to string

   Returns
   Boolean. True on success.

virtual bool get(Mesh *m, int &ival, const std::string &name, int def = 0) override
   Get an integer.
   Read a single integer from file. If the integer is not found, then ival is set to zero and false is returned.

   Inputs
   m Pointer to mesh, not used name String containing name of variable

   Outputs
   ival Reference to integer
Outputs
ival Reference to integer

Returns
Boolean. True on success.

virtual bool get(Mesh *m, BoutReal &rval, const std::string &name, BoutReal def = 0.0) override
Get a BoutReal number.

virtual bool get(Mesh *m, Field2D &var, const std::string &name, BoutReal def = 0.0) override
Reads a 2D, 3D or FieldPerp field variable from a file
   Successfully reads Field2D or FieldPerp if the variable in the file is 0-D or 2-D. Successfully reads Field3D
   if the variable in the file is 0-D, 2-D or 3-D.

virtual bool get(Mesh *m, Field3D &var, const std::string &name, BoutReal def = 0.0) override

inline virtual bool get(Mesh *m, FieldPerp &var, const std::string &name, BoutReal def = 0.0) override

virtual bool get(Mesh *m, std::vector<int> &var, const std::string &name, int len, int offset = 0,
   GridDataSource::Direction dir = GridDataSource::X) override

virtual bool get(Mesh *m, std::vector<BoutReal> &var, const std::string &name, int len, int offset = 0,
   GridDataSource::Direction dir = GridDataSource::X) override

virtual bool hasXBoundaryGuards(Mesh *m) override
   Are x-boundary guard cells read from the source?

inline virtual bool hasYBoundaryGuards() override
   Are y-boundary guard cells read from the source?

Private Functions

bool readgrid_3dvar_fft(Mesh *m, const std::string &name, int yread, int ydest, int ysize, int xread, int
   xdest, int xsize, Field3D &var)
Reads in a portion of the X-Y domain.

bool readgrid_3dvar_real(const std::string &name, int yread, int ydest, int ysize, int xread, int
   xdest, int xsize, Field3D &var)
Reads a 3D variable directly from the file, without any processing

bool readgrid_perpvar_fft(Mesh *m, const std::string &name, int xread, int xdest, int xsize, FieldPerp
   &var)

bool readgrid_perpvar_real(const std::string &name, int xread, int xdest, int xsize, FieldPerp &var)
   Reads a FieldPerp variable directly from the file, without any processing

template<typename T>
bool getField(Mesh *m, T &var, const std::string &name, BoutReal def = 0.0)

void readField(Mesh *m, const std::string &name, int ys, int yd, int ny_to_read, int xs, int x,
   int nx_to_read, const std::vector<int> &size, Field2D &var)
void **readField**(Mesh \*m, const std::string \&name, int ys, int yd, int ny_to_read, int xs, int xd, int nx_to_read, const std::vector<int> \&size, Field3D \&var)

void **readField**(Mesh \*m, const std::string \&name, int ys, int yd, int ny_to_read, int xs, int xd, int nx_to_read, const std::vector<int> \&size, FieldPerp \&var)

**Private Members**

std::unique_ptr<DataFormat> **file**

std::string **filename**

int **grid_yguards** = \{0\}

int **ny_inner** = \{0\}

class GridFromOptions : public GridDataSource

#include <griddata.hxx> Provides a way to create variables from options, which can be set in the input file or on the command line. This is done using FieldFactory to convert string expressions into fields.

**Public Functions**

inline GridFromOptions(Options \*opt = nullptr)

Constructor, passing optional Options object

Parameters

- \texttt{opt} – [in] Options section to use as input. By default the “mesh” section under root will be used.

virtual bool hasVar(const std::string \&name) override

Checks if the options has a given variable

virtual bool get(Mesh \*mesh, std::string \&sval, const std::string \&name, const std::string \&def = \"") override

Reads strings from options. Uses Options::get to handle expressions

Parameters

- \texttt{mesh} – [in] Not used
- \texttt{name} – [in] Name of variable
- \texttt{sval} – [out] Always given a value, defaults to 0

Returns True if option is set, false if ival is default (0)

virtual bool get(Mesh \*mesh, int \&ival, const std::string \&name, int def = 0) override

Reads integers from options. Uses Options::get to handle expressions

Parameters

- \texttt{mesh} – [in] Not used
- \texttt{ival} – [out] The variable which will be set
- \texttt{name} – [in] Name of variable
- \texttt{def} – [in] Default value to use if option not found

Returns True if option is set, false if ival is default (0)
virtual bool get(Mesh *mesh, BoutReal &rval, const std::string &name, BoutReal def = 0.0) override
  Reads BoutReal from options. Uses Options::get to handle expressions

Parameters
  • mesh – [in] Not used
  • name – [in] Name of variable
  • rval – [out] Always given a value, defaults to 0

Returns True if option is set, false if ival is default (0)

virtual bool get(Mesh *mesh, Field2D &var, const std::string &name, BoutReal def = 0.0) override
  Get a Field2D object by finding the option with the given name, and passing the string to FieldFactory

Parameters
  • mesh – [in] The Mesh object over which the field is defined
  • var – [out] The variable which will be set
  • name – [in] The name in the options. Not case sensitive
  • def – [in] Default value to use if option not found

virtual bool get(Mesh *mesh, Field3D &var, const std::string &name, BoutReal def = 0.0) override
  Get a Field3D object by finding the option with the given name, and passing the string to FieldFactory

Parameters
  • mesh – [in] The Mesh object over which the field is defined
  • var – [out] The variable which will be set
  • name – [in] The name in the options. Not case sensitive
  • def – [in] Default value to use if option not found

virtual bool get(Mesh *mesh, FieldPerp &var, const std::string &name, BoutReal def = 0.0) override
  Get a FieldPerp object by finding the option with the given name, and passing the string to FieldFactory

Parameters
  • mesh – [in] The Mesh object over which the field is defined
  • var – [out] The variable which will be set
  • name – [in] The name in the options. Not case sensitive
  • def – [in] Default value to use if option not found

virtual bool get(Mesh *mesh, std::vector<int> &var, const std::string &name, int len, int offset = 0,
  GridDataSource::Direction dir = GridDataSource::X) override
  Get an array of integers. Currently reads a single integer, and sets the whole array to the same value

Parameters
  • mesh – [in] Mesh object
  • var – [out] A vector which will be resized to length len
  • name – [in] The name of the option
  • len – [in] The length of the vector
  • offset – [in] Not currently used
  • dir – [in] The direction (X,Y,Z) of the array
virtual bool get(Mesh *mesh, std::vector<BoutReal> &var, const std::string &name, int len, int offset = 0, GridDataSource::Direction dir = GridDataSource::X) override

Get an array of BoutReals. Uses FieldFactory to generate an expression, then evaluates it at indices depending on the direction (dir) and length (len)

Parameters

- mesh – [in] Mesh object
- var – [out] A vector which will be resized to length len
- name – [in] The name of the option
- len – [in] The length of the vector
- offset – [in] The index where this vector starts i.e. var[0] is at x=offset if dir is X.
- dir – [in] The direction (X,Y,Z) of the array

inline virtual bool hasXBoundaryGuards(Mesh *m) override

Are x-boundary guard cells read from the source?

inline virtual bool hasYBoundaryGuards() override

Are y-boundary guard cells read from the source?

Private Members

Options *options

The options section to use. Could be nullptr.

K.2.92 File gridfromfile.cxx

K.2.93 File gridfromoptions.cxx

K.2.94 File gyro_average.cxx

Functions

Field3D gyroTaylor0(const Field3D &f, const Field3D &rho)

Gyro-average using Taylor series approximation

Γ(f) = f + ρ²∇²⊥(f)

Note: Faster, but less robust than Pade approximations

Parameters

- f – [in] The field to gyro-average

Field3D gyroPade0(const Field3D &f, BoutReal rho, int inner_boundary_flags, int outer_boundary_flags)

Field3D gyroPade0(const Field3D &f, const Field2D &rho, int inner_boundary_flags, int outer_boundary_flags)
Field3D gyroPade0(const Field3D &f, const Field3D &rho, int inner_boundary_flags, int outer_boundary_flags)
Gyro-average using Pade approximation
\[ \Gamma_0 = (1 - \rho^2 \nabla^2_\perp)g = f \]
NOTE: Uses Z average of rho for efficient inversion

Parameters
- f – [in] The field to gyro-average
- flags – [in] Flags to be passed to the Laplacian inversion operator

Field3D gyroPade1(const Field3D &f, BoutReal rho, int inner_boundary_flags, int outer_boundary_flags)
Field3D gyroPade1(const Field3D &f, const Field2D &rho, int inner_boundary_flags, int outer_boundary_flags)
Field3D gyroPade2(const Field3D &f, BoutReal rho, int inner_boundary_flags, int outer_boundary_flags)
Field3D gyroPade2(const Field3D &f, const Field2D &rho, int inner_boundary_flags, int outer_boundary_flags)
Field3D gyroPade2(const Field3D &f, const Field3D &rho, int inner_boundary_flags, int outer_boundary_flags)

Pade approximation \[ \Gamma_1 = (1 - \frac{1}{2} \rho^2 \nabla^2_\perp)g = f \]
Note: Have to use Z average of rho for efficient inversion

Parameters
- f – [in] The field to gyro-average
- flags – [in] Flags to be passed to the Laplacian inversion operator

Field2D gyroPade1(const Field2D &f, const Field2D &rho, int inner_boundary_flags, int outer_boundary_flags)

Field3D gyroPade0(const Field3D &f, BoutReal rho, int flags)
Field3D gyroPade0(const Field3D &f, const Field2D &rho, int flags)
\textit{Field3D} \texttt{gyroPade0} \text{(const} \textit{Field3D} &f, \text{const} \textit{Field3D} &rho, \text{int} flags) \\
\textit{Field3D} \texttt{gyroPade1} \text{(const} \textit{Field3D} &f, \textit{BoutReal} rho, \text{int} flags) \\
\textit{Field3D} \texttt{gyroPade1} \text{(const} \textit{Field3D} &f, \text{const} \textit{Field2D} &rho, \text{int} flags) \\
\textit{Field3D} \texttt{gyroPade1} \text{(const} \textit{Field3D} &f, \text{const} \textit{Field3D} &rho, \text{int} flags) \\
\textit{Field2D} \texttt{gyroPade1} \text{(const} \textit{Field2D} &f, \text{const} \textit{Field2D} &rho, \text{int} flags) \\
\textit{Field3D} \texttt{gyroPade2} \text{(const} \textit{Field3D} &f, \textit{BoutReal} rho, \text{int} flags) \\
\textit{Field3D} \texttt{gyroPade2} \text{(const} \textit{Field3D} &f, \text{const} \textit{Field2D} &rho, \text{int} flags) \\
\textit{Field3D} \texttt{gyroPade2} \text{(const} \textit{Field3D} &f, \text{const} \textit{Field3D} &rho, \text{int} flags) \\

\textbf{K.2.95 File gyro_average.hxx} \\

Gyro-averaging operators \\
2010-09-03 Ben Dudson bd512@york.ac.uk \\
\begin{itemize}
  \item Initial version, simple averaging operator
\end{itemize}
Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu \\
Contact: Ben Dudson, bd512@york.ac.uk \\
This file is part of BOUT++. \\
BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. \\
BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. \\
You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see \url{http://www.gnu.org/licenses/}. \\

\textit{Appendix K. API reference}
Functions

Field3D \texttt{gyroTaylor0}(\texttt{const Field3D} \ & \ \texttt{f}, \ \texttt{const Field3D} \ & \ \texttt{rho})

Gyro-average using Taylor series approximation

\[ \Gamma(f) = f + \rho^2 \nabla_\perp^2(f) \]

Note: Faster, but less robust than Pade approximations

Parameters

- \texttt{f} – \texttt{[in]} The field to gyro-average
- \texttt{rho} – \texttt{[in]} Gyro-radius

Field3D \texttt{gyroPade0}(\texttt{const Field3D} \ & \ \texttt{f}, \ \texttt{const Field3D} \ & \ \texttt{rho}, \ \texttt{int inner_boundary_flags}, \ \texttt{int outer_boundary_flags})

Gyro-average using Pade approximation

\[ \Gamma_0 = (1 - \frac{1}{2} \rho^2 \nabla_\perp^2)g = f \]

NOTE: Uses Z average of \texttt{rho} for efficient inversion

Parameters

- \texttt{f} – \texttt{[in]} The field to gyro-average
- \texttt{rho} – \texttt{[in]} Gyro-radius
- \texttt{flags} – \texttt{[in]} Flags to be passed to the \texttt{Laplacian} inversion operator

Field3D \texttt{gyroPade0}(\texttt{const Field3D} \ & \ \texttt{f}, \ \texttt{BoutReal} \ \texttt{rho}, \ \texttt{int inner_boundary_flags}, \ \texttt{int outer_boundary_flags})

Field3D \texttt{gyroPade0}(\texttt{const Field3D} \ & \ \texttt{f}, \ \texttt{Field2D} \ & \ \texttt{rho}, \ \texttt{int inner_boundary_flags}, \ \texttt{int outer_boundary_flags})

Field3D \texttt{gyroPade0}(\texttt{const Field3D} \ & \ \texttt{f}, \ \texttt{BoutReal} \ \texttt{rho})

Field3D \texttt{gyroPade0}(\texttt{const Field3D} \ & \ \texttt{f}, \ \texttt{Field2D} \ & \ \texttt{rho})

Field3D \texttt{gyroPade0}(\texttt{const Field3D} \ & \ \texttt{f}, \ \texttt{BoutReal} \ \texttt{rho}, \ \texttt{int flags})

Field3D \texttt{gyroPade0}(\texttt{const Field3D} \ & \ \texttt{f}, \ \texttt{Field3D} \ & \ \texttt{rho}, \ \texttt{int flags})

Field3D \texttt{gyroPade0}(\texttt{const Field3D} \ & \ \texttt{f}, \ \texttt{Field2D} \ & \ \texttt{rho}, \ \texttt{int flags})

Field3D \texttt{gyroPade0}(\texttt{const Field3D} \ & \ \texttt{f}, \ \texttt{BoutReal} \ \texttt{rho}, \ \texttt{int flags})

Field3D \texttt{gyroPade1}(\texttt{const Field3D} \ & \ \texttt{f}, \ \texttt{const Field3D} \ & \ \texttt{rho}, \ \texttt{int inner_boundary_flags}, \ \texttt{int outer_boundary_flags})

Pade approximation \( Gamma_1 = (1 - \frac{1}{2} \rho^2 \nabla_\perp^2)g = f \)

Note: Have to use Z average of \texttt{rho} for efficient inversion

Parameters

- \texttt{f} – \texttt{[in]} The field to gyro-average
- \texttt{rho} – \texttt{[in]} Gyro-radius
- \texttt{flags} – \texttt{[in]} Flags to be passed to the \texttt{Laplacian} inversion operator
Field3D gyroPade1(const Field3D &f, const Field2D &rho, int inner_boundary_flags, int outer_boundary_flags)

Field3D gyroPade1(const Field3D &f, BoutReal rho, int inner_boundary_flags, int outer_boundary_flags)

Field2D gyroPade1(const Field2D &f, const Field2D &rho, int inner_boundary_flags, int outer_boundary_flags)

inline Field3D gyroPade1(const Field3D &f, const Field3D &rho)

inline Field3D gyroPade1(const Field3D &f, const Field2D &rho)

inline Field3D gyroPade1(const Field3D &f, BoutReal rho)

inline Field2D gyroPade1(const Field2D &f, const Field2D &rho)

Field3D gyroPade1(const Field3D &f, const Field3D &rho, int flags)

Field3D gyroPade1(const Field3D &f, const Field2D &rho, int flags)

Field3D gyroPade1(const Field3D &f, BoutReal rho, int flags)

Field2D gyroPade1(const Field2D &f, const Field2D &rho, int flags)

Field3D gyroPade2(const Field3D &f, const Field3D &rho, int inner_boundary_flags, int outer_boundary_flags)

Pade approximation

$$
\Gamma_2(f) = \frac{1}{2} \rho^2 \nabla^2 \nabla \cdot (1 - \frac{1}{2} \rho^2 \nabla^2)^{-1} \Gamma_1(f)
$$

Note: Have to use Z average of rho for efficient inversion

Parameters

- \textbf{f} – \textbf{[in]} The field to gyro-average
- \textbf{rho} – \textbf{[in]} Gyro-radius
- \textbf{flags} – \textbf{[in]} Flags to be passed to the Laplacian inversion operator

Field3D gyroPade2(const Field3D &f, const Field2D &rho, int inner_boundary_flags, int outer_boundary_flags)

Field3D gyroPade2(const Field3D &f, BoutReal rho, int inner_boundary_flags, int outer_boundary_flags)

inline Field3D gyroPade2(const Field3D &f, const Field3D &rho)

inline Field3D gyroPade2(const Field3D &f, const Field2D &rho)

inline Field3D gyroPade2(const Field3D &f, BoutReal rho)
Field3D gyroPade2(const Field3D &f, const Field3D &rho, int flags)

Field3D gyroPade2(const Field3D &f, const Field2D &rho, int flags)

Field3D gyroPade2(const Field3D &f, BoutReal rho, int flags)

Variables

constexpr int GYRO_FLAGS = INVERT_BNDRY_ONE + INVERT_RHS
   INVERT_BNDRY_ONE | INVERT_IN_RHS | INVERT_OUT_RHS; uses old-style Laplacian inversion flags

K.2.96 File h5_format.cxx

K.2.97 File h5_format.hxx

HDF5 data format interface.

Records: In netCDF, the time dimension for each dimension must be the same. Hence when a record is appended to a variable, the size of all variables is increased. To work out which record to write to, a map of variable names to record number is kept.

Author John Omotani

Date 2015

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

class H5Format : public DataFormat
Public Functions

H5Format(bool parallel_in = false, Mesh *mesh_in = nullptr)

H5Format(const char *name, bool parallel_in = false, Mesh *mesh_in = nullptr)

inline H5Format(const std::string &name, bool parallel_in = false, Mesh *mesh_in = nullptr)

~H5Format()

virtual bool openr(const char *name) override

virtual bool openw(const char *name, bool append = false) override

virtual bool isValid() override

virtual void close() override

virtual void flush() override

inline const char *filename()

virtual const std::vector<int> getSize(const char *var) override

virtual const std::vector<int> getSize(const std::string &var) override

virtual bool setGlobalOrigin(int x = 0, int y = 0, int z = 0) override

virtual bool setLocalOrigin(int x = 0, int y = 0, int z = 0, int offset_x = 0, int offset_y = 0, int offset_z = 0) override

virtual bool setRecord(int t) override

virtual bool addVarInt(const std::string &name, bool repeat) override

virtual bool addVarIntVec(const std::string &name, bool repeat, size_t size) override

virtual bool addVarString(const std::string &name, bool repeat, size_t size) override

virtual bool addVarBoutReal(const std::string &name, bool repeat) override

virtual bool addVarField2D(const std::string &name, bool repeat) override
virtual bool addVarField3D(const std::string &name, bool repeat) override

virtual bool addVarFieldPerp(const std::string &name, bool repeat) override

virtual bool read(int *var, const char *name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read(int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read(char *var, const char *name, int n = 1) override

virtual bool read(char *var, const std::string &name, int n = 1) override

virtual bool read(BoutReal *var, const char *name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read(BoutReal *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read_perp(BoutReal *var, const std::string &name, int lx = 1, int lz = 0) override

virtual bool write(int *var, const char *name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write(int *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write(char *var, const char *name, int n = 1) override

virtual bool write(char *var, const std::string &name, int n = 1) override

virtual bool write(BoutReal *var, const char *name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write(BoutReal *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write_perp(BoutReal *var, const std::string &name, int lx = 0, int lz = 0) override

virtual bool read_rec(int *var, const char *name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read_rec(int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read_rec(char *var, const char *name, int n = 1) override

virtual bool read_rec(char *var, const std::string &name, int n = 1) override

virtual bool read_rec(BoutReal *var, const char *name, int lx = 1, int ly = 0, int lz = 0) override
virtual bool read_rec(BoutReal *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read_rec_perp(BoutReal *var, const std::string &name, int lx = 1, int lz = 0) override

virtual bool write_rec(int *var, const char *name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write_rec(char *var, const char *name, int n = 1) override

virtual bool write_rec(BoutReal *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write_rec(BoutReal *var, const std::string &name, int lx = 0, int lz = 0) override

virtual bool write_rec_perp(BoutReal *var, const std::string &name, int lx = 0, int lz = 0) override

inline virtual void setLowPrecision() override

virtual void setAttribute(const std::string &varname, const std::string &attrname, const std::string &text) override
Sets a string attribute

Inputs

Parameters

- **varname** – [in] Variable name. The variable must already exist. If varname is the empty string "" then the attribute will be added to the file instead of to a variable.
- **attrname** – [in] Attribute name
- **text** – [in] A string attribute to attach to the variable

virtual void setAttribute(const std::string &varname, const std::string &attrname, int value) override
Sets an integer attribute

Inputs

Parameters

- **varname** – [in] Variable name. The variable must already exist. If varname is the empty string "" then the attribute will be added to the file instead of to a variable.
- **attrname** – [in] Attribute name
- **value** – [in] An int attribute to attach to the variable

virtual void setAttribute(const std::string &varname, const std::string &attrname, BoutReal value) override
Sets a BoutReal attribute

Inputs

Parameters
• **varname** – [in] Variable name. The variable must already exist. If varname is the empty string "" then the attribute will be added to the file instead of to a variable.

• **attrname** – [in] Attribute name

• **value** – [in] A BoutReal attribute to attach to the variable

---

```cpp
virtual bool getAttribute(const std::string &varname, const std::string &attrname, const std::string &text) override
```

Get a string attribute

**Inputs**

- varname
- attrname

**Returns**

- text A string attribute of the variable

---

**Parameters**

- **varname** – [in] Variable name. The variable must already exist. If varname is the empty string "" then get the attribute from the top-level of the file instead of from a variable.

- **attrname** – [in] Attribute name

---

```cpp
virtual bool getAttribute(const std::string &varname, const std::string &attrname, int &value) override
```

Get an integer attribute

**Inputs**

- varname
- attrname

**Returns**

- value An int attribute of the variable

---

**Parameters**

- **varname** – [in] Variable name. The variable must already exist. If varname is the empty string "" then get the attribute from the top-level of the file instead of from a variable.

- **attrname** – [in] Attribute name

---

```cpp
virtual bool getAttribute(const std::string &varname, const std::string &attrname, BoutReal &value) override
```

Get a BoutReal attribute

**Inputs**

- varname
- attrname

**Returns**

- value A BoutReal attribute of the variable

---

**Parameters**

- **varname** – [in] Variable name. The variable must already exist. If varname is the empty string "" then get the attribute from the top-level of the file instead of from a variable.

- **attrname** – [in] Attribute name
Private Functions

bool addVar(const std::string &name, bool repeat, hid_t write_hdf5_type, std::string datatype, int lx = 0, int ly = 0, int lz = 0)

bool read(void *var, hid_t hdf5_type, const char *name, int lx = 1, int ly = 0, int lz = 0)

bool write(void *var, hid_t mem_hdf5_type, const char *name, int lx = 0, int ly = 0, int lz = 0)

bool read_rec(void *var, hid_t hdf5_type, const char *name, int lx = 1, int ly = 0, int lz = 0)

bool write_rec(void *var, hid_t mem_hdf5_type, const char *name, int lx = 0, int ly = 0, int lz = 0)

void setAttribute(const hid_t &dataSet, const std::string &attrname, const std::string &text)

void setAttribute(const hid_t &dataSet, const std::string &attrname, int value)

void setAttribute(const hid_t &dataSet, const std::string &attrname, BoutReal value)

bool getAttribute(const hid_t &dataSet, const std::string &attrname, std::string &text)

bool getAttribute(const hid_t &dataSet, const std::string &attrname, int &value)

bool getAttribute(const hid_t &dataSet, const std::string &attrname, BoutReal &value)

Private Members

char *fname
Current file name.

hid_t dataFile
hid_t dataFile_plist
hid_t dataSet_plist
bool lowPrecision
When writing, down-convert to floats.

bool parallel
int x0
int y0
int z0
int t0
Data origins for file access.
int \texttt{x0\_local}
int \texttt{y0\_local}
int \texttt{z0\_local}

Data origins for memory access.

\texttt{hszie\_t chunk\_length}

\textbf{K.2.98 File hermite\_spline\_cxx}

\textbf{K.2.99 File ida\_cxx}

Defines

\texttt{ZERO}
\texttt{ONE}

Typedefs

using \texttt{IDAINT =bout::utils::function\_traits< IDABBDLocalFn>::arg\_t<0>}

Functions

static int \texttt{idares}(\texttt{BoutReal t, N\_Vector u, N\_Vector du, N\_Vector rr, void *}user\_data)

static int \texttt{ida\_bbd\_res}(\texttt{IDAINT Nlocal, BoutReal t, N\_Vector u, N\_Vector du, N\_Vector rr, void *}user\_data)

Residual function for BBD preconditioner.

static int \texttt{ida\_pre}(\texttt{BoutReal t, N\_Vector yy, N\_Vector yp, N\_Vector rr, N\_Vector rvec, N\_Vector zvec, BoutReal cj, BoutReal delta, void *}user\_data)

static inline int \texttt{ida\_pre\_shim}(\texttt{BoutReal t, N\_Vector yy, N\_Vector yp, N\_Vector rr, N\_Vector rvec, N\_Vector zvec, BoutReal cj, BoutReal delta, void *}user\_data, \texttt{N\_Vector tmp})

\textbf{K.2.100 File ida\_hxx}

class \texttt{IdaSolver} : public \texttt{Solver}
Public Functions

**IdaSolver**(*Options* *opts = nullptr)

~**IdaSolver**()

virtual int **init**(int nout, **BoutReal** tstep) override
   Initialise the solver NOTE: nout and tstep should be passed to run, not init. Needed because of how the PETSc TS code works

virtual int **run**() override
   Run the solver, calling monitors nout times, at intervals of tstep. This function is called by *solve()*, and is specific to each solver type
   This should probably be protected, since it shouldn’t be called by users.

**BoutReal** **run**(**BoutReal** tout)

void **res**(**BoutReal** t, **BoutReal** *udata, **BoutReal** *dudata, **BoutReal** *rdata)

void **pre**(**BoutReal** t, **BoutReal** cj, **BoutReal** delta, **BoutReal** *udata, **BoutReal** *rvec, **BoutReal** *zvec)

Private Members

int **NOUT**

**BoutReal** **TIMESTEP**

N_Vector **uvec** = {nullptr}

N_Vector **duvec** = {nullptr}

N_Vector **id** = {nullptr}

void **idamem** = {nullptr}

**BoutReal** **pre_Wtime** = {0.0}

int **pre_ncalls** = {0}

K.2.101 File identity.cxx

K.2.102 File imex-bdf2.cxx

2nd order IMEX-BDF scheme


Uses PETSc for the SNES interface

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu
Functions

static PetscErrorCode FormFunction(SNES snes, Vec x, Vec f, void *ctx)
    PETSc callback function, which evaluates the nonlinear function to be solved by SNES.
    This function assumes the context void pointer is a pointer to an IMEXBDF2 object.

static PetscErrorCode FormFunctionForDifferencing(void *ctx, Vec x, Vec f)
    PETSc callback function for forming Jacobian
    This function can be a linearised form of FormFunction

static PetscErrorCode FormFunctionForColoring(SNES snes, Vec x, Vec f, void *ctx)
    SNES callback for forming Jacobian with coloring
    This can be a linearised and simplified form of FormFunction

static PetscErrorCode imexbdf2PCapply(PC pc, Vec x, Vec y)

class SaveVarOp

    Public Functions

    inline SaveVarOp(Field2D *var, Field2D *F_var)

    inline SaveVarOp(Field3D *var, Field3D *F_var)

    inline void run(int jx, int jy, BoutReal *u)

    inline void run(int jx, int jy, int jz, BoutReal *u)
Private Members

```
Field2D *var2D
Field3D *var3D
```

class LoadVarOp

Public Functions

```
inline LoadVarOp(Field2D *var, Field2D *F_var)
inline LoadVarOp(Field3D *var, Field3D *F_var)
inline void run(int jx, int jy, BoutReal *u)
inline void run(int jx, int jy, int jz, BoutReal *u)
```

Private Members

```
Field2D *var2D
Field3D *var3D
```

class SaveDerivsOp

Public Functions

```
inline SaveDerivsOp(Field2D *var, Field2D *F_var)
inline SaveDerivsOp(Field3D *var, Field3D *F_var)
inline void run(int jx, int jy, BoutReal *u)
inline void run(int jx, int jy, int jz, BoutReal *u)
```

Private Members

```
Field2D *F_var2D
Field3D *F_var3D
```
K.2.103 File imex-bdf2.hxx

class IMEXBDF2 : public Solver
#include <imex-bdf2.hxx> IMEX-BDF2 time integration solver


The method has been extended to variable order, variable timestep, and includes some adaptive capabilities

Public Functions

IMEXBDF2(Options *opt = nullptr)

~IMEXBDF2()

inline virtual BoutReal getCurrentTimestep() override
Returns the current internal timestep.

virtual int init(int nout, BoutReal tstep) override
Initialise solver. Must be called once and only once

Initialisation routine. Called once before solve.

Parameters

• nout – [in] Number of outputs
• tstep – [in] Time between outputs. NB: Not internal timestep

virtual int run() override
Run the simulation.

PetscErrorCode snes_function(Vec x, Vec f, bool linear)
Nonlinear function. This is called by PETSc SNES object via a static C-style function. For implicit time integration this function calculates:

\[ f = (x - \gamma G(x)) - \text{rhs} \]

Parameters

• x – [in] The state vector
• f – [out] The vector for the result f(x)
• linear – [in] Specifies that the SNES solver is in a linear (KSP) inner loop, so the operator should be linearised if possible

PetscErrorCode precon(Vec x, Vec f)
Preconditioner. Called by PCapply via a C-style static function.

Parameters

• x – [in] The vector to be operated on
• f – [out] The result of the operation
Private Functions

```c
void take_step(BoutReal curtime, BoutReal dt, int order = 2)
```

Take a full step at requested order

Take a full IMEX-BDF step of order “order”. Note that this assumes that enough time points are already available (in u and f).

Inputs: u* - Solution history f* - Non-stiff component history

Outputs: u - Latest Solution f1 - Non-stiff time derivative at current time

**Parameters**

- **curtime** – [in] The current simulation time
- **dt** – [in] The time step to take
- **order** – [in] The order of accuracy

```c
void constructSNES(SNES *snesIn)
```

Setup a SNES object This includes creating, setting functions, options, and internal (KSP) solver, and Jacobian options including coloring.

```c
void shuffleState()
```

Shuffle state along one step.

```c
void calculateCoeffs(int order)
```

Populate the *Fac vectors and dtImp with appropriate coefficients for this order.

```c
PetscErrorCode solve_implicit(BoutReal curtime, BoutReal gamma)
```

```c
template<class Op>
void loopVars(BoutReal *u)
```

Loop over arrays, using template parameter to specify the operation to be performed at each point

```c
void saveVars(BoutReal *u)
```

Save variables from BOUT++ fields into a pre-allocated array u

Copy data from fields into array

```c
void loadVars(BoutReal *u)
```

Load variables from input vector u into BOUT++ fields.

Copy data from array into fields

```c
void saveDerivs(BoutReal *u)
```

Save time derivatives from ddt() fields into a preallocated array u.

Copy time derivatives from fields into array
**Private Members**

```cpp
int maxOrder
   Specify the maximum order of the scheme to use (1/2/3)

BoutReal out_timestep
   The output timestep.

int nsteps
   Number of output steps.

BoutReal timestep
   The internal timestep.

int ninternal
   Number of internal steps per output.

int mxstep
   Maximum number of internal steps between outputs.

bool adaptive
   Use adaptive timestepping?

int nadapt
   How often do we check the error.

int mxstepAdapt
   Maximum no. consecutive times we try to reduce timestep.

BoutReal scaleCushUp
   Don’t increase timestep if scale factor < 1.0+scaleCushUp.

BoutReal scaleCushDown
   Don’t decrease timestep if scale factor > 1.0-scaleCushDown.

BoutReal adaptRtol
   Target relative error for adaptivity.

BoutReal dtMin
   Minimum timestep we want to use.

BoutReal dtMax
   Maximum timestep we want to use.

BoutReal dtMinFatal
   If timestep wants to drop below this we abort. Set -ve to deactivate.
```

```cpp
std::vector<BoutReal> uFac
std::vector<BoutReal> fFac
std::vector<BoutReal> gFac
```
\textit{BoutReal} \texttt{dtImp}

int \texttt{nlocal}

int \texttt{neq}
Number of variables on local processor and in total.

\texttt{Array\<BoutReal\> u}
System state at current time.

\texttt{std::vector\\langle Array\\<BoutReal\\rangle \rangle uV}
The solution history.

\texttt{std::vector\\langle Array\\<BoutReal\\rangle \rangle fV}
The non-stiff solution history.

\texttt{std::vector\<BoutReal\> timesteps}
Timestep history.

\texttt{Array\<BoutReal\> rhs}
\texttt{Array\<BoutReal\> err}

\textit{BoutReal} \texttt{implicit\_gamma}

\textit{BoutReal} \texttt{implicit\_curtime}

int \texttt{predictor}
Predictor method.

\textit{PetscLib} \texttt{lib}
Handles initialising, finalising PETSc.

\texttt{Vec snes\_f}
Used by SNES to store function.

\texttt{Vec snes\_x}
Result of SNES.

\texttt{SNES snes}
SNES context.

\texttt{SNES snesAlt}
Alternative SNES object for adaptive checks.

\texttt{SNES snesUse}
The snes object to use in solve stage. Allows easy switching.

\texttt{Mat Jmf}
Matrix-free Jacobian.

bool \texttt{diagnose}
Output diagnostics every timestep.
bool verbose
  Gives a more verbose output for each timestep.

int linear_fails
  Number of linear (KSP) convergence failures.

int nonlinear_fails
  Number of nonlinear (SNES) convergence failures.

bool have_constraints
  Are there any constraint variables?

Array<BoutReal> is_dae
  If using constraints, 1 -> DAE, 0 -> AE.

MatFDColoring fdcoloring
  Matrix coloring context, used for finite difference Jacobian evaluation.

Private Static Attributes

static const int MAX_SUPPORTED_ORDER = 4

K.2.104 File index_derivs.cxx

Functions

REGISTER_STANDARD_DERIVATIVE (DDX_C2, "C2", 1, DERIV::Standard)
  central, 2nd order

REGISTER_STANDARD_DERIVATIVE (DDX_C4, "C4", 2, DERIV::Standard)
  central, 4th order

REGISTER_STANDARD_DERIVATIVE (DDX_CWENO2, "W2", 1, DERIV::Standard)
  Central WENO method, 2nd order (reverts to 1st order near shocks)

REGISTER_STANDARD_DERIVATIVE (DDX_S2, "S2", 2, DERIV::Standard)

REGISTER_STANDARD_DERIVATIVE (D2DX2_C2, "C2", 1, DERIV::StandardSecond)
  Also CWENO3 but needs an upwind op so define later.
  Second derivative: Central, 2nd order

REGISTER_STANDARD_DERIVATIVE (D2DX2_C4, "C4", 2, DERIV::StandardSecond)
  Second derivative: Central, 4th order.

REGISTER_STANDARD_DERIVATIVE (D4DX4_C2, "C2", 2, DERIV::StandardFourth)

std::tuple<BoutReal, BoutReal> vUpDown(BoutReal v)
  Upwind non-staggered methods
Basic derivative methods. All expect to have an input grid cell at the same location as the output. Hence convert cell centred values -> centred values, or left -> left.

**REGISTER_UPWIND_DERIVATIVE (VDDX_C2, "C2", 1, DERIV::Upwind)**
Upwinding: Central, 2nd order.

**REGISTER_UPWIND_DERIVATIVE (VDDX_C4, "C4", 2, DERIV::Upwind)**
Upwinding: Central, 4th order.

**REGISTER_UPWIND_DERIVATIVE (VDDX_U1, "U1", 1, DERIV::Upwind)**
upwind, 1st order

**REGISTER_UPWIND_DERIVATIVE (VDDX_U2, "U2", 2, DERIV::Upwind)**
upwind, 2nd order

**REGISTER_UPWIND_DERIVATIVE (VDDX_U3, "U3", 2, DERIV::Upwind)**
upwind, 3rd order

**REGISTER_UPWIND_DERIVATIVE (VDDX_WENO3, "W3", 2, DERIV::Upwind)**
3rd-order WENO scheme

**REGISTER_STANDARD_DERIVATIVE (DDX_CWENO3, "W3", 2, DERIV::Standard)**
3rd-order CWENO. Uses the upwinding code and split flux

**REGISTER_FLUX_DERIVATIVE (FDDX_U1, "U1", 1, DERIV::Flux)**
Flux non-staggered methods

**REGISTER_FLUX_DERIVATIVE (FDDX_U2, "U2", 2, DERIV::Flux)**

**REGISTER_FLUX_DERIVATIVE (FDDX_C2, "C2", 2, DERIV::Flux)**

**REGISTER_FLUX_DERIVATIVE (FDDX_C4, "C4", 2, DERIV::Flux)**

**REGISTER_STANDARD_DERIVATIVE_STAGGERED (DDX_C2_stag, "C2", 1, DERIV::Standard)**
Standard methods first order.

Staggered methods

Map Centre -> Low or Low -> Centre

These expect the output grid cell to be at a different location to the input.

The stencil no longer has a value in ‘C’ (centre) instead, points are shifted as follows:

mm -> -3/2 h m -> -1/2 h p -> +1/2 h pp -> +3/2 h

NOTE: Cell widths (dx, dy, dz) are currently defined as centre->centre for the methods above. This is currently not taken account of, so large variations in cell size will cause issues.

**REGISTER_STANDARD_DERIVATIVE_STAGGERED (DDX_C4_stag, "C4", 2, DERIV::Standard)**

**REGISTER_STANDARD_DERIVATIVE_STAGGERED (D2DX2_C2_stag, "C2", 2, DERIV::StandardSecond)**
Standard methods second order.
REGISTER_UPWIND_DERIVATIVE_STAGGERED (VDDX_U1_stag, "U1", 1, DERIV::Upwind)
Upwind methods.
REGISTER_UPWIND_DERIVATIVE_STAGGERED (VDDX_U2_stag, "U2", 2, DERIV::Upwind)
REGISTER_UPWIND_DERIVATIVE_STAGGERED (VDDX_C2_stag, "C2", 1, DERIV::Upwind)
REGISTER_UPWIND_DERIVATIVE_STAGGERED (VDDX_C4_stag, "C4", 2, DERIV::Upwind)
REGISTER_FLUX_DERIVATIVE_STAGGERED (FDDX_U1_stag, "U1", 1, DERIV::Flux)
Flux methods.
REGISTER_FLUX_DERIVATIVE_STAGGERED (FDDX_U2_stag, "U2", 2, DERIV::Flux)

Variables

produceCombinations< Set< WRAP_ENUM(DIRECTION, X), WRAP_ENUM(DIRECTION, Y), WRAP_ENUM(DIRECTION, YOrthogonal), WRAP_ENUM(DIRECTION, Z)>, Set< WRAP_ENUM(STAGGER, None), WRAP_ENUM(STAGGER, C2L), WRAP_ENUM(STAGGER, L2C)>, Set< TypeContainer< Field3D >, TypeContainer< Field2D >>, Set< SplitFluxDerivativeType > > registerSplitDerivative (registerMethod{})

class SplitFluxDerivativeType
Here’s an example of defining and registering a custom method that doesn’t fit into the standard stencil based approach.

Public Functions

template<DIRECTION direction, STAGGER stagger, int nGuards, typename T>
inline void standard(const T &vel, T &var, const std::string region) const

template<DIRECTION direction, STAGGER stagger, int nGuards, typename T>
inline void upwindOrFlux(const T &vel, const T &var, T &result, const std::string region) const

Public Members

<metaData meta = {"SPLIT", 2, DERIV::Flux}>

K.2.105 File index_derivs.hxx

Definition of available derivative methods and registration within store
Copyright 2018 D.Dickinson, P.Hill, B.Dudson
Contact: Ben Dudson, bd512@york.ac.uk
This file is part of BOUT++. 
BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

**Defines**

`DEFINE_STANDARD_DERIV_CORE (name, key, nGuards, type)`

`DEFINE_STANDARD_DERIV (name, key, nGuards, type)`

`DEFINE_UPWIND_DERIV_CORE (name, key, nGuards, type)`

`DEFINE_UPWIND_DERIV (name, key, nGuards, type)`

`DEFINE_FLUX_DERIV_CORE (name, key, nGuards, type)`

`DEFINE_FLUX_DERIV (name, key, nGuards, type)`

`DEFINE_STANDARD_DERIV_STAGGERED (name, key, nGuards, type)`

`DEFINE_UPWIND_DERIV_STAGGERED (name, key, nGuards, type)`

`DEFINE_FLUX_DERIV_STAGGERED (name, key, nGuards, type)`

`WRAP_ENUM (family, value)`

Some helper defines for now that allow us to wrap up enums and the specific methods.

`REGISTER_DERIVATIVE (name)`

`REGISTER_STAGGERED_DERIVATIVE (name)`

`REGISTER_STANDARD_DERIVATIVE (name, key, nGuards, type)`

`REGISTER_UPWIND_DERIVATIVE (name, key, nGuards, type)`

`REGISTER_FLUX_DERIVATIVE (name, key, nGuards, type)`

`REGISTER_STANDARD_STAGGERED_DERIVATIVE (name, key, nGuards, type)`
REGISTER_STANDARD_DERIVATIVE_STAGGERED(name, key, nGuards, type)
REGISTER_UPWIND_STAGGERED_DERIVATIVE(name, key, nGuards, type)
REGISTER_UPWIND_DERIVATIVE_STAGGERED(name, key, nGuards, type)
REGISTER_FLUX_STAGGERED_DERIVATIVE(name, key, nGuards, type)
REGISTER_FLUX_DERIVATIVE_STAGGERED(name, key, nGuards, type)

Functions

inline std::ostream &operator<<(std::ostream &out, const metaData &meta)
    Provide an easy way to report a Region’s statistics.

Variables

const BoutReal WENO_SMALL = 1.0e-8
struct metaData

Public Members

    const char *key
    const int nGuards
    const DERIV derivType

template<typename FF>
class DerivativeType

#include <index_derivs.hxx> Here we define a helper class that provides a means to use a supplied stencil using functor to calculate a derivative over the entire field. Note we currently have a different interface for some of the derivative types to avoid needing different classes to represent the different operations. The use of a functor here makes it possible to wrap up metaData into the type as well.

Public Functions

template<DIRECTION direction, STAGGER stagger, int nGuards, typename T>
inline void standard(const T &var, T &result, const std::string &region) const

template<DIRECTION direction, STAGGER stagger, int nGuards, typename T>
inline void upwindOrFlux(const T &vel, const T &var, T &result, const std::string &region) const

inline BoutReal apply(const stencil &f) const
inline BoutReal apply(BoutReal v, const stencil &f) const

inline BoutReal apply(const stencil &v, const stencil &f) const

**Public Members**

const _FF func = {}  
const metaData meta = func.meta

struct registerMethod  
#include <index_derivs.hxx> Following code is for dealing with registering a method/methods for all template combinations, in conjunction with the template_combinations code.

**Public Functions**

template<typename Direction, typename Stagger, typename FieldTypeContainer, typename Method>
inline void operator()(Direction, Stagger, FieldTypeContainer, Method)

**K.2.106 File index_derivs_interface.hxx**

Definition of main derivative kernels  
Copyright 2018 D.Dickinson  
Contact: Ben Dudson, bd512@york.ac.uk  
This file is part of BOUT++.  
BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.  
BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.  
You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

namespace bout  
SNB model

namespace derivatives

namespace index
Functions

template<typename T, DIRECTION direction, DERIV derivType>
T flowDerivative(const T &vel, const T &f, CELL_LOC outloc, const std::string &method, const std::string &region)

The main kernel used for all upwind and flux derivatives.

template<typename T, DIRECTION direction, DERIV derivType>
T standardDerivative(const T &f, CELL_LOC outloc, const std::string &method, const std::string &region)

The main kernel used for all standard derivatives.

template<typename T>
T DDX(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

template<typename T>
T D2DX2(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

template<typename T>
T D4DX4(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

template<typename T>
T DDY(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

template<typename T>
T D2DY2(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

template<typename T>
T D4DY4(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

template<typename T>
T DDZ(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

template<typename T>
T D2DZ2(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

template<typename T>
T D4DZ4(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
Advection operator in index space in \( \mathbf{[]} \) direction

\[
v \frac{df}{dt}
\]

Parameters

- \( v \) – \textbf{[in]} The velocity in the Y direction
- \( f \) – \textbf{[in]} The field being advected
- \( \text{outloc} \) – \textbf{[in]} The cell location where the result is desired. The default is the same as \( f \)
- \( \text{method} \) – \textbf{[in]} The differencing method to use
- \( \text{region} \) – \textbf{[in]} The region of the grid for which the result is calculated.

\[
\text{template<typename T>} \\
\text{T VDDX(const T &vel, const T &f, } \text{CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")}
\]

\[
\text{template<typename T>} \\
\text{T VDDY(const T &vel, const T &f, } \text{CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")}
\]

\[
\text{template<typename T>} \\
\text{T FDDX(const T &vel, const T &f, } \text{CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")}
\]

\[
\text{template<typename T>} \\
\text{T FDDY(const T &vel, const T &f, } \text{CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")}
\]

\[
\text{template<typename T>} \\
\text{T FDDZ(const T &vel, const T &f, } \text{CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")}
\]

\[
\text{template<typename T>} \\
\text{T FDDZ(const T &vel, const T &f, } \text{CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")}
\]

\text{K.2.107 File initialprofiles.cxx}

\text{Functions}

\text{void initial_profile(const std::string &name, Field3D &var)
}

- Set a field from options

  This is called by \text{Solver} for each evolving field at the beginning of a simulation.

To create the value, it looks for a setting “function” in a section called name. If that is not found, then it looks for “function” in a section called “All”. If that is also not found, then the value defaults to zero.

A second variable, “scale”, can be used to multiply the function, and defaults to 1.0

\text{Example}

Given the input file:
BOUT++ Documentation, Release 4.4.0

[All] function = sin(y)
[pressure]
[density] scale = 0.2
[vorticity] function = cos(y)

Initial_profile would generate:
- pressure -> sin(y)
- density -> 0.2*sin(y)
- vorticity -> cos(y)

Parameters
- name – [in] The name of the field. This will be used as the section name in the options
- var – [out] The field, which will be set to a value depending on the options

```cpp
void initial_profile(const std::string &name, Field2D &var)
Set a Field2D from options
```

Parameters
- name – [in] The name of the field, used as a section name
- var – [out] The field which will be set to a value

```cpp
void initial_profile(const std::string &name, Vector2D &var)
Set a vector to a value. The options used depend on whether the vector is covariant or contravariant.
If covariant, then each component will be initialised by adding "_x", "_y", "_z" to the name.
If contravariant, then each component will be initialised by adding "x", "y" and "z" to the name.
```

```cpp
void initial_profile(const std::string &name, Vector3D &var)
Set a vector to a value. The options used depend on whether the vector is covariant or contravariant.
If covariant, then each component will be initialised by adding "_x", "_y", "_z" to the name.
If contravariant, then each component will be initialised by adding "x", "y" and "z" to the name.
```

**K.2.108 File initialprofiles.hxx**

**Functions**

```cpp
void initial_profile(const std::string &name, Field3D &var)
Set a field from options
```

This is called by Solver for each evolving field at the beginning of a simulation.

To create the value, it looks for a setting “function” in a section called name. If that is not found, then it looks for “function” in a section called “All”. If that is also not found, then the value defaults to zero.

A second variable, “scale”, can be used to multiply the function, and defaults to 1.0

**Example**

Given the input file:

- [All] function = sin(y)
- [pressure]
- [density] scale = 0.2
[vorticity] function = \cos(y)

initial_profile would generate:
- pressure -> \sin(y)
- density -> 0.2*\sin(y)
- vorticity -> \cos(y)

Parameters
- name – [in] The name of the field. This will be used as the section name in the options
- var – [out] The field, which will be set to a value depending on the options

void initial_profile(const std::string &name, Field2D &var)
Set a Field2D from options

Parameters
- name – [in] The name of the field, used as a section name
- var – [out] The field which will be set to a value

void initial_profile(const std::string &name, Vector2D &var)
Set a vector to a value. The options used depend on whether the vector is covariant or contravariant.
If covariant, then each component will be initialised by adding “_x”, “_y”, “_z” to the name.
If contravariant, then each component will be initialised by adding “x”, “y” and “z” to the name.

void initial_profile(const std::string &name, Vector3D &var)
Set a vector to a value. The options used depend on whether the vector is covariant or contravariant.
If covariant, then each component will be initialised by adding “_x”, “_y”, “_z” to the name.
If contravariant, then each component will be initialised by adding “x”, “y” and “z” to the name.

K.2.109 File interpolation.cxx

Functions

void printLocation(const Field3D &var)
Print out the cell location (for debugging)

void printLocation(const Field2D &var)

const char *strLocation(CELL_LOC loc)

const Field3D interpolate(const Field3D &f, const Field3D &delta_x, const Field3D &delta_z)
Interpolate a field onto a perturbed set of points.

const Field3D interpolate(const Field2D &f, const Field3D &delta_x, const Field3D &delta_z)

const Field3D interpolate(const Field2D &f, const Field3D &delta_x)
**K.2.110 File interpolation.hxx**

**Functions**

```cpp
inline BoutReal interp(const stencil &s)
    Perform interpolation between centre - > shifted or vice-versa.
    Interpolate using 4th-order staggered formula
    
    **Parameters**
    
    - **s** – [in] Input stencil. mm -> -3/2, m -> -1/2, p -> +1/2, pp -> +3/2

    template<typename T>
    const T interp_to(const T &var, CELL_LOC loc, const std::string region = "RGN_ALL")
    Interpolate to a give cell location.
    Interpolate between different cell locations
    NOTE: This requires communication if the result is required in guard cells
    NOTE: Since corner guard cells cannot be communicated, it never makes sense to calculate interpolation in guard cells.
    If guard cell values are required, we must communicate (unless interpolating in z). Since mesh->communicate() communicates both x- and y-guard cells by default, there is no difference between RGN_ALL, RGN_NOX and RGN_NOY.
    
    **Parameters**
    
    - **var** – [in] Input variable
    - **loc** – [in] Location of output values
    - **region** – [in] Region where output will be calculated

    template<typename T>
    const T interp_to(const T &var, CELL_LOC loc, REGION region)
```

```cpp
void printLocation(const Field3D &var)
    Print out the cell location (for debugging)

const char *strLocation(CELL_LOC loc)
```

```cpp
const Field3D interpolate(const Field3D &f, const Field3D &delta_x, const Field3D &delta_z)
    Interpolate a field onto a perturbed set of points.

const Field3D interpolate(const Field2D &f, const Field3D &delta_x, const Field3D &delta_z)

const Field3D interpolate(const Field2D &f, const Field3D &delta_x)
```

```cpp
class Interpolation
    Subclassed by Bilinear, HermiteSpline, Lagrange4pt
```
Public Functions

inline Interpolation(int y_offset = 0, Mesh *localmeshIn = nullptr)

inline Interpolation(const BoutMask &mask, int y_offset = 0, Mesh *mesh = nullptr)

virtual ~Interpolation() = default

virtual void calcWeights(const Field3D &delta_x, const Field3D &delta_z) = 0

virtual void calcWeights(const Field3D &delta_x, const Field3D &delta_z, const BoutMask &mask) = 0

virtual Field3D interpolate(const Field3D &f) const = 0

virtual Field3D interpolate(const Field3D &f, const Field3D &delta_x, const Field3D &delta_z) = 0

virtual Field3D interpolate(const Field3D &f, const Field3D &delta_x, const Field3D &delta_z, const BoutMask &mask) = 0

inline void setMask(const BoutMask &mask)

inline void setYOffset(int offset)

Public Members

int y_offset

Protected Attributes

Mesh *localmesh = {nullptr}
BoutMask skip_mask

class HermiteSpline : public Interpolation
Subclassed by MonotonicHermiteSpline

Public Functions

inline HermiteSpline(Mesh *mesh = nullptr)

HermiteSpline(int y_offset = 0, Mesh *mesh = nullptr)

inline HermiteSpline(const BoutMask &mask, int y_offset = 0, Mesh *mesh = nullptr)
virtual void calcWeights(const Field3D &delta_x, const Field3D &delta_z) override

virtual void calcWeights(const Field3D &delta_x, const Field3D &delta_z, const BoutMask &mask) override

virtual Field3D interpolate(const Field3D &f) const override

virtual Field3D interpolate(const Field3D &f, const Field3D &delta_x, const Field3D &delta_z) override

virtual Field3D interpolate(const Field3D &f, const Field3D &delta_x, const Field3D &delta_z, const BoutMask &mask) override

**Public Static Functions**

static inline Interpolation *CreateHermiteSpline(Mesh *mesh)

  Callback function for InterpolationFactory.

**Protected Attributes**

*Tensor<int>* i_corner

  This is protected rather than private so that it can be extended and used by HermiteSplineMonotonic

*Tensor<int>* k_corner

  Field3D h00_x

  Field3D h01_x

  Field3D h10_x

  Field3D h11_x

  Field3D h00_z

  Field3D h01_z

  Field3D h10_z

  Field3D h11_z

class MonotonicHermiteSpline : public HermiteSpline

  #include <interpolation.hxx> Monotonic Hermite spline interpolator

  Similar to HermiteSpline, so uses most of the same code. Forces the interpolated result to be in the range of the neighbouring cell values. This prevents unphysical overshoots, but also degrades accuracy near maxima and minima. Perhaps should only impose near boundaries, since that is where problems most obviously occur.
Public Functions

inline MonotonicHermiteSpline(Mesh *mesh = nullptr)

inline MonotonicHermiteSpline(int y_offset = 0, Mesh *mesh = nullptr)

inline MonotonicHermiteSpline(const BoutMask &mask, int y_offset = 0, Mesh *mesh = nullptr)

virtual Field3D interpolate(const Field3D &f) const override
  Interpolate using precalculated weights. This function is called by the other interpolate functions in the base class HermiteSpline.

Public Static Functions

static inline Interpolation *CreateMonotonicHermiteSpline(Mesh *mesh)
  Callback function for InterpolationFactory.

class Lagrange4pt : public Interpolation

Public Functions

inline Lagrange4pt(Mesh *mesh = nullptr)

Lagrange4pt(int y_offset = 0, Mesh *mesh = nullptr)

inline Lagrange4pt(const BoutMask &mask, int y_offset = 0, Mesh *mesh = nullptr)

virtual void calcWeights(const Field3D &delta_x, const Field3D &delta_z) override

virtual void calcWeights(const Field3D &delta_x, const Field3D &delta_z, const BoutMask &mask) override

virtual Field3D interpolate(const Field3D &f) const override

virtual Field3D interpolate(const Field3D &f, const Field3D &delta_x, const Field3D &delta_z) override

virtual Field3D interpolate(const Field3D &f, const Field3D &delta_x, const Field3D &delta_z, const BoutMask &mask) override

BoutReal lagrange_4pt(BoutReal v2m, BoutReal vm, BoutReal vp, BoutReal v2p, BoutReal offset) const

BoutReal lagrange_4pt(const BoutReal v[], BoutReal offset) const
Public Static Functions

static inline Interpolation *CreateLagrange4pt(Mesh *mesh)
Callback function for InterpolationFactory.

Private Members

Tensor<int> i_corner
Tensor<int> k_corner
Field3D t_x
Field3D t_z
class Bilinear : public Interpolation

Public Functions

inline Bilinear(Mesh *mesh = nullptr)
Bilinear(int y_offset = 0, Mesh *mesh = nullptr)
inline Bilinear(const BoutMask &mask, int y_offset = 0, Mesh *mesh = nullptr)
virtual void calcWeights(const Field3D &delta_x, const Field3D &delta_z) override
virtual void calcWeights(const Field3D &delta_x, const Field3D &delta_z, const BoutMask &mask) override
virtual Field3D interpolate(const Field3D &f) const override
virtual Field3D interpolate(const Field3D &f, const Field3D &delta_x, const Field3D &delta_z) override
virtual Field3D interpolate(const Field3D &f, const Field3D &delta_x, const Field3D &delta_z, const BoutMask &mask) override

Public Static Functions

static inline Interpolation *CreateBilinear(Mesh *mesh)
Callback function for InterpolationFactory.
Private Members

Tensor<int> i_corner
Tensor<int> k_corner
Field3D w0
Field3D w1
Field3D w2
Field3D w3

K.2.111 File interpolation_factory.cxx
K.2.112 File interpolation_factory.hxx

class InterpolationFactory

Public Types

using CreateInterpCallback = Interpolation (*)(Mesh*)
Callback function definition for creating Interpolation objects.

Public Functions

~InterpolationFactory() = default

inline std::string getDefaultInterpType()
A string representing the default interpolation type.

inline Interpolation *create(Mesh *mesh)
Create an interpolation object.

Interpolation *create(Options *options = nullptr, Mesh *mesh = nullptr)

Interpolation *create(const std::string &name, Options *options = nullptr, Mesh *mesh = nullptr)
Create an Interpolation object

Parameters

• name – The name of the interpolation method
• options – An Options object (e.g. an input file)
• mesh – A Mesh object to construct the interpolation on

Returns A new copy of an Interpolation object

void add(CreateInterpCallback interp, const std::string &name)
Add available interpolations to database.
Public Static Functions

static InterpolationFactory *getInstance()
Create or get the singleton instance of the factory.

static void cleanup()
Destroy the singleton instance.

Private Functions

InterpolationFactory()
    Add the available interpolation methods to the internal map
    Private default constructor to prevent instantiation of this class

CreateInterpCallback findInterpolation(const std::string &name)
Find an interpolation method in the list of available methods

    Parameters
        name – Name of the interpolation method

    Returns
        A pointer to the Interpolation object in the map

Private Members

std::map<std::string, CreateInterpCallback> interp_map
Database of available interpolations.

Private Static Attributes

static InterpolationFactory *instance = nullptr
The only instance of this class (singleton)

K.2.113 File invert_laplace.cxx

Perpendicular Laplacian inversion using FFT and Tridiagonal solver.

Equation solved is \(d \ast \nabla_\perp^2 x + (1/c) \nabla \cdot \text{erpc} \cdot \nabla_\perp x + ax = b\), where \(x\) and \(x\) are perpendicular (X-Z) or 3D fields, and \(a\) and \(d\) are 2D fields. If \(d\) is not supplied then it is 1

Flags control the boundary conditions (see header file)

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.
Functions

void laplace_tridag_coefs(int jx, int jy, int jz, dcomplex &a, dcomplex &b, dcomplex &c, const Field2D *ccoef, const Field2D *d, CELL_LOC loc)

Returns the coefficients for a tridiagonal matrix for laplace. Used by Delp2 too.

int invert_laplace(const FieldPerp &b, FieldPerp &x, int flags, const Field2D *a, const Field2D *c, const Field2D *d)

int invert_laplace(const Field3D &b, Field3D &x, int flags, const Field2D *a, const Field2D *c, const Field2D *d)

const Field3D invert_laplace(const Field3D &b, int flags, const Field2D *a, const Field2D *c, const Field2D *d)

More readable API for calling Laplacian inversion. Returns x.

K.2.114 File invert_laplace.hxx

Perpendicular Laplacian inversion using FFT and Tridiagonal solver

Equation solved is: \( d \star \nabla^2 \perp x + \frac{1}{c} \nabla \perp x \cdot \nabla \perp x + ax = b \)

Where a, c and d are functions of x and y only (not z)

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

Defines

PVEC_REAL_MPI_TYPE
Functions

```c
void laplace_tridag_coefs(int jx, int jy, int jz, dcomplex &a, dcomplex &b, dcomplex &c, const Field2D *ccoef = nullptr, const Field2D *d = nullptr, CELL_LOC loc = CELL_DEFAULT)

Returns the coefficients for a tridiagonal matrix for laplace. Used by Delp2 too.
```

```c
int invert_laplace(const FieldPerp &b, FieldPerp &x, int flags, const Field2D *a, const Field2D *c = nullptr, const Field2D *d = nullptr)
```

```c
int invert_laplace(const Field3D &b, Field3D &x, int flags, const Field2D *a, const Field2D *c = nullptr, const Field2D *d = nullptr)
```

```c
const Field3D invert_laplace(const Field3D &b, int flags, const Field2D *a = nullptr, const Field2D *c = nullptr, const Field2D *d = nullptr)

More readable API for calling Laplacian inversion. Returns x.
```

Variables

```c
constexpr int INVERT_DC_GRAD = 1
Zero-gradient for DC (constant in Z) component. Default is zero value.
```

```c
constexpr int INVERT_AC_GRAD = 2
Zero-gradient for AC (non-constant in Z) component. Default is zero value.
```

```c
constexpr int INVERT_AC_LAP = 4
Use zero-laplacian (decaying solution) to AC component.
```

```c
constexpr int INVERT_SYM = 8
Use symmetry to enforce either zero-value or zero-gradient.
```

```c
constexpr int INVERT_SET = 16
Set boundary to value.
```

```c
constexpr int INVERT_RHS = 32
Use input value in RHS boundary.
```

```c
constexpr int INVERT_DC_LAP = 64
Use zero-laplacian solution for DC component.
```

```c
constexpr int INVERT_BNDRY_ONE = 128
Only use one boundary point.
```

```c
constexpr int INVERT_DC_GRADPAR = 256
```

```c
constexpr int INVERT_DC_GRADPARINV = 512
```

```c
constexpr int INVERT_IN_CYLINDER = 1024
For use in cylindrical coordinate system.
```

```c
constexpr int INVERT_ZERO_DC = 1
Zero the DC (constant in Z) component of the solution.
```
constexpr int INVERT_START_NEW = 2
    Iterative method start from solution=0. Has no effect for direct solvers.

constexpr int INVERT_BOTH_BNDRY_ONE = 4
    Sets the width of the boundaries to 1.

constexpr int INVERT_4TH_ORDER = 8
    Use band solver for 4th order in x.

constexpr int INVERT_KX_ZERO = 16
    Zero the kx=0, n = 0 component.

class Laplacian
    #include <invert_laplace.hxx> Base class for Laplacian inversion.


Public Functions

Laplacian(Options *options = nullptr, const CELL_LOC loc = CELL_CENTRE, Mesh *mesh_in = nullptr)
    Laplacian inversion initialisation. Called once at the start to get settings.

virtual ~Laplacian() = default

virtual void setCoefA(const Field2D &val) = 0
    Set coefficients for inversion. Re-builds matrices if necessary.

inline virtual void setCoefA(const Field3D &val)

inline virtual void setCoefA(BoutReal r)

virtual void setCoefC(const Field2D &val) = 0

inline virtual void setCoefC(const Field3D &val)

inline virtual void setCoefC(BoutReal r)

inline virtual void setCoefC1(const Field2D &val)

inline virtual void setCoefC1(const Field3D &val)

inline virtual void setCoefC1(BoutReal r)

inline virtual void setCoefC2(const Field2D &val)

inline virtual void setCoefC2(const Field3D &val)

inline virtual void setCoefC2(BoutReal r)
inline virtual void setCoefC2(BoutReal r)

virtual void setCoefD(const Field2D &val) = 0

inline virtual void setCoefD(const Field3D &val)

inline virtual void setCoefD(BoutReal r)

virtual void setCoefEx(const Field2D &val) = 0

inline virtual void setCoefEx(const Field3D &val)

inline virtual void setCoefEx(BoutReal r)

virtual void setCoefEz(const Field2D &val) = 0

inline virtual void setCoefEz(const Field3D &val)

inline virtual void setCoefEz(BoutReal r)

inline virtual void setGlobalFlags(int f)

inline virtual void setInnerBoundaryFlags(int f)

inline virtual void setOuterBoundaryFlags(int f)

virtual void setFlags(int f)

inline virtual bool uses3DCoefs() const

   Does this solver use Field3D coefficients (true) or only their DC component (false)

virtual FieldPerp solve(const FieldPerp &b) = 0

virtual Field3D solve(const Field3D &b)

virtual Field2D solve(const Field2D &b)

inline virtual FieldPerp solve(const FieldPerp &b, const FieldPerp &x0)

virtual Field3D solve(const Field3D &b, const Field3D &x0)

   Performs the laplacian inversion y-slice by y-slice

   Parameters
   
   - \( b = \text{[in]} \) All the y-slices of b_slice, which is the right hand side of the equation
     \[ A*x\_slice = b\_slice \]
• $x_0$ – [in] All the y-slices of the variable eventually used to set BC

Returns $x$ All the y-slices of $x_{\text{slice}}$ in the equation $A^*x_{\text{slice}} = b_{\text{slice}}$

virtual Field2D solve(const Field2D &b, const Field2D &$x_0$)

void tridagCoefs(int jx, int jy, int jz, dcomplex &a, dcomplex &b, dcomplex &c, const Field2D *ccoef = nullptr, const Field2D *d = nullptr, CELL_LOC loc = CELL_DEFAULT)

Coefficients in tridiagonal inversion.

Public Static Functions

static Laplacian *create(Options *opt = nullptr, const CELL_LOC loc = CELL_CENTER, Mesh *mesh_in = nullptr)

Create a new Laplacian solver

Parameters opt – [in] The options section to use. By default “laplace” will be used

static Laplacian *defaultInstance()

Return pointer to global singleton.

static void cleanup()

Frees all memory.

Protected Functions

inline void tridagCoefs(int jx, int jy, BoutReal kwave, dcomplex &a, dcomplex &b, dcomplex &c, const Field2D *ccoef = nullptr, const Field2D *d = nullptr, CELL_LOC loc = CELL_DEFAULT)

void tridagCoefs(int jx, int jy, BoutReal kwave, dcomplex &a, dcomplex &b, dcomplex &c, const Field2D *ccoef, const Field2D *d, CELL_LOC loc = CELL_DEFAULT)

void tridagMatrix(dcomplex **avec, dcomplex **bvec, dcomplex **cvec, dcomplex **bk, int jy, int flags, int inner_boundary_flags, int outer_boundary_flags, const Field2D *a = nullptr, const Field2D *d = nullptr)

Sets the coefficients for parallel tridiagonal matrix inversion.

Uses the laplace_tridag_coefs routine above to fill a matrix [kz][ix] of coefficients

inline void tridagMatrix(dcomplex **avec, dcomplex **bvec, dcomplex **cvec, dcomplex **bk, int jy, int kz, BoutReal kwave, int flags, int inner_boundary_flags, int outer_boundary_flags, const Field2D *a, const Field2D *c1coef, const Field2D *d, bool includeguards = true)

void tridagMatrix(dcomplex **avec, dcomplex **bvec, dcomplex **cvec, dcomplex **bk, int jy, int kz, BoutReal kwave, int flags, int inner_boundary_flags, int outer_boundary_flags, const Field2D *a, const Field2D *c1coef, const Field2D *c2coef, const Field2D *d, bool includeguards = true)

Sets the matrix components of A in $Ax = b$

This function will

i. Calling tridagCoef, solving

$$D^*\text{Laplace}_\text{perp}(x) + \left(1/C1\right)\text{Grad}_\text{perp}(C2)^*\text{Grad}_\text{perp}(x) + Ax = B$$
for each fourier component

ii. Set the boundary conditions by setting the first and last rows properly

**Parameters**

- **avec** – [in] Lower diagonal of the tridiagonal matrix. DO NOT CONFUSE WITH “A”
- **bvec** – [in] The main diagonal
- **cvec** – [in] The upper diagonal. DO NOT CONFUSE WITH “C” (called ccoef here)
- **bk** – [in] The b in Ax = b
- **jy** – [in] Index of the current y-slice
- **kz** – [in] The mode number index
- **kwave** – [in] The mode number (different from kz only if we are taking a part of the z-domain [and not from 0 to 2*pi])
- **global_flags** – [in] Global flags of the inversion
- **inner_boundary_flags** – [in] Flags used to set the inner boundary
- **outer_boundary_flags** – [in] Flags used to set the outer boundary
- **a** – [in] A in the equation above. DO NOT CONFUSE WITH avec
- **c1coef** – [in] C1 in the equation above. DO NOT CONFUSE WITH cvec
- **c2coef** – [in] C2 in the equation above. DO NOT CONFUSE WITH cvec
- **d** – [in] D in the equation above
- **includeguards** – [in] Whether or not the guard points in x should be used
- **avec** – [out] Lower diagonal of the tridiagonal matrix. DO NOT CONFUSE WITH “A”
- **bvec** – [out] The main diagonal
- **cvec** – [out] The upper diagonal. DO NOT CONFUSE WITH “C” (called ccoef here)

**Protected Attributes**

- **bool async_send**
  If true, use asynchronous send in parallel algorithms.

- **int maxmode**
  The maximum Z mode to solve for.

- **bool low_mem**
  If true, reduce the amount of memory used.

- **bool all_terms**
  Applies to Delp2 operator and laplacian inversion

- **bool nonuniform**
  Non-uniform mesh correction.
bool include_yguards
    solve in y-guard cells, default true.

int extra_yguards_lower
    exclude some number of points at the lower boundary, useful for staggered grids or when boundary conditions make inversion redundant

int extra_yguards_upper
    exclude some number of points at the upper boundary, useful for staggered grids or when boundary conditions make inversion redundant

int global_flags
    Default flags.

int inner_boundary_flags
    Flags to set inner boundary condition.

int outer_boundary_flags
    Flags to set outer boundary condition.

CELL_LOC location
    staggered grid location of this solver

Mesh *localmesh
    Mesh object for this solver.

Coordinates *coords
    Coordinates object, so we only have to call localmesh->getCoordinates(location) once

Private Static Attributes

static Laplacian *instance = nullptr
    Singleton instance.

K.2.115 File invert_parderiv.cxx

K.2.116 File invert_parderiv.hxx

Defines

PARDERIVCYCLIC
class InvertPar
    #include <invert_parderiv.hxx> Base class for parallel inversion solvers.
    Inverts a matrix of the form
    A + B * Grad2_par2 + C*D2DYDZ + D*D2DZ2 + E*DDY
    Example
InvertPar *inv = InvertPar::Create(); inv->setCoefA(1.0); inv->setCoefB(-0.1);

Field3D result = inv->solve(rhs);

Subclassed by InvertParCR

Public Functions

inline InvertPar(Options *opt, CELL_LOC location_in, Mesh *mesh_in = nullptr)
    Constructor. Note that this is a base class, with pure virtual members, so can’t be created directly. To create an InvertPar object call the create() static function.

virtual ~InvertPar() = default

virtual const Field2D solve(const Field2D &f)
    Solve the system of equations Warning: Default implementation very inefficient. This converts the Field2D to a Field3D then calls solve() on the 3D variable

virtual const Field3D solve(const Field3D &f) = 0
    Solve the system of equations
    This method must be implemented

inline virtual const Field3D solve(const Field2D &f, const Field2D &start)
    Solve, given an initial guess for the solution This can help if using an iterative scheme

inline virtual const Field3D solve(const Field3D &f, const Field3D &start)

virtual void setCoefA(const Field2D &f) = 0
    Set the constant coefficient A

inline virtual void setCoefA(const Field3D &f)

inline virtual void setCoefA(BoutReal f)

virtual void setCoefB(const Field2D &f) = 0
    Set the Grad2_par2 coefficient B

inline virtual void setCoefB(const Field3D &f)

inline virtual void setCoefB(BoutReal f)

virtual void setCoefC(const Field2D &f) = 0
    Set the D2DYDZ coefficient C

inline virtual void setCoefC(const Field3D &f)

inline virtual void setCoefC(BoutReal f)

virtual void setCoefD(const Field2D &f) = 0
    Set the D2DZ2 coefficient D

inline virtual void setCoefD(const Field3D &f)
inline virtual void `setCoefD(BoutReal f)`

virtual void `setCoefE(const Field2D &f) = 0`  
Set the DDY coefficient E

inline virtual void `setCoefE(const Field3D &f)`

Public Static Functions

static `InvertPar *Create(Mesh *mesh_in = nullptr)`  
Create an instance of InvertPar

Note: For consistency this should be renamed “create” and take an Options* argument

Protected Attributes

`CELL_LOC location`

`Mesh *localmesh`

Mesh object for this solver.

class ParDerivFactory

Public Functions

`InvertPar *createInvertPar(CELL_LOC location = CELL CENTRE, Mesh *mesh_in = bout::globals::mesh)`

`InvertPar *createInvertPar(const char *type, Options *opt = nullptr, CELL_LOC location = CELL CENTRE, Mesh *mesh_in = bout::globals::mesh)`

`InvertPar *createInvertPar(Options *opts, CELL_LOC location = CELL CENTRE, Mesh *mesh_in = bout::globals::mesh)`

Public Static Functions

static `ParDerivFactory *getInstance()`  
Return a pointer to the only instance.
Private Functions

inline ParDerivFactory() 

Private Static Attributes

static ParDerivFactory *instance = nullptr
The only instance of this class (Singleton)

K.2.117 File invertable_operator.hxx

namespace bout
SNB model

namespace inversion

Functions

template<typename T>
T identity(const T &in)
No-op function to use as a default may wish to remove once testing phase complete.

template<typename T>
PetscErrorCode fieldToPetscVec(const T &in, Vec out)
Pack a PetscVec from a Field<T>

template<typename T>
PetscErrorCode petscVecToField(Vec in, T &out)
Pack a Field<T> from a PetscVec.

template<typename T>
class InvertableOperator
#include <invertable_operator.hxx> Class to define an invertable operator. Provides interface to PETSc routines for solving A.x = b

Public Types

using data_type = T
What type of field does the operator take?

using function_signature = std::function<T(const T&)> The signature of the functor that applies the operator.
Public Functions

```cpp
inline InvertableOperator(const function_signature &func = identity<T>, Options *optIn = nullptr, Mesh *localmeshIn = nullptr)
    Almost empty constructor currently don’t actually use Options for anything.
```

```cpp
inline ~InvertableOperator()
    Destructor just has to cleanup the PETSc owned objects.
```

```cpp
inline void setOperatorFunction(const function_signature &func, bool alsoSetPreconditioner = true)
    Allow the user to override the existing function Note by default we set the preconditioner function to match this as this is the usual mode of operation. If the user doesn’t want to do this they can set alsoSetPreconditioner to false.
```

```cpp
inline void setPreconditionerFunction(const function_signature &func)
    Allow the user to override the existing preconditioner function.
```

```cpp
inline void setOperatorFunction(const function_signature &func, bool alsoSetPreconditioner = true)
    Allow the user to override the existing function Note by default we set the preconditioner function to match this as this is the usual mode of operation. If the user doesn’t want to do this they can set alsoSetPreconditioner to false.
```

```cpp
inline T operator()(const T &input)
    Provide a way to apply the operator to a Field.
```

```cpp
inline T apply(const T &input)
    Provide a synonym for applying the operator to a Field.
```

```cpp
inline PetscErrorCode setup()
    Sets up the PETSc objects required for inverting the operator Currently also takes the functor that applies the operator this class represents. Not actually required by any of the setup so this should probably be moved to a separate place (maybe the constructor).
```

```cpp
inline T invert(const T &rhsField, const T &guess)
```

```cpp
inline T invert(const T &rhsField)
    Triggers the solve of A.x = b for x, where b = rhs and A is the matrix representation of the operator we represent. Should probably provide an overload or similar as a way of setting the initial guess.
```

```cpp
inline bool verify(const T &rhsIn, BoutReal tol = 1.0e-5)
    With checks enabled provides a convinence routine to check that applying the registered function on the calculated inverse gives back the initial values.
```

Public Static Functions

```cpp
static inline void reportTime()
    Reports the time spent in various parts of InvertableOperator. Note that as the Timer “labels” are not unique to an instance the time reported is summed across all different instances.
```

Private Members

Mat matOperator
Mat matPreconditioner
Vec rhs
Vec lhs
KSP ksp
**function_signature** `operatorFunction = identity<T>`

The function that represents the operator that we wish to invert.

**function_signature** `preconditionerFunction = identity<T>`

The function that represents the preconditioner for the operator that we wish to invert.

```cpp
Options *opt = nullptr
Mesh *localmesh = nullptr
bool doneSetup = false
PetscLib lib
```

**Private Static Functions**

```cpp
static inline PetscErrorCode functionWrapper(Mat m, Vec v1, Vec v2)
    Wrapper that gets a pointer to the parent InvertableOperator instance from the Matrix m and uses this to get the actual function to call. Copies data from v1 into a field of type T, calls the function on this and then copies the result into the v2 argument.

static inline PetscErrorCode preconditionerWrapper(Mat m, Vec v1, Vec v2)
    Wrapper that gets a pointer to the parent InvertableOperator instance from the Matrix m and uses this to get the actual function to call. Copies data from v1 into a field of type T, calls the function on this and then copies the result into the v2 argument.
```

**K.2.118 File karniadakis.cxx**

**K.2.119 File karniadakis.hxx**

class **KarniadakisSolver** : public **Solver**

**Public Functions**

```cpp
KarniadakisSolver(Options *options)
```

```cpp
inline ~KarniadakisSolver()
```

```cpp
inline virtual BoutReal getCurrentTimestep() override
    Return the current internal timestep.

virtual int init(int nout, BoutReal tstep) override
    Initialise the solver NOTE: nout and tstep should be passed to run, not init. Needed because of how the PETSc TS code works.

virtual int run() override
    Run the solver, calling monitors nout times, at intervals of tstep. This function is called by solve(), and is specific to each solver type.
    This should probably be protected, since it shouldn't be called by users.

virtual void resetInternalFields() override
    Should wipe out internal field vector and reset from current field object data.
```
Private Functions

void take_step(BoutReal dt)

Private Members

Array<BoutReal> f1
Array<BoutReal> f0
Array<BoutReal> fm1
Array<BoutReal> fm2
Array<BoutReal> S0
Array<BoutReal> Sm1
Array<BoutReal> Sm2
Array<BoutReal> D0
bool first_time
BoutReal out_timestep
int nsteps
BoutReal timestep
int nsubsteps
int nlocal

K.2.120 File lagrange_4pt.cxx

K.2.121 File lapack_routines.cxx

Serial code to invert a complex tridiagonal system

Complex banded matrix solver

Solves a banded matrix given the matrix in compact form a[0...(n-1)][0...(m1+m2)] and the rhs vector b[0...(n-1)]
a is overwritten, and b is replaced by the solution

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.
You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

Functions

void zgtsv_(int *n, int *nrhs, fcplx *dl, fcplx *d, fcplx *du, fcplx *b, int *ldb, int *info)
Complex tridiagonal inversion.

void dgtsv_(int *n, int *nrhs, BoutReal *dl, BoutReal *d, BoutReal *du, BoutReal *b, int *ldb, int *info)
BoutReal (double) tridiagonal inversion.

void zgbsv_(int *n, int *kl, int *ku, int *nrhs, fcplx *ab, int *ldab, int *ipiv, fcplx *b, int *ldb, int *info)
Complex band solver.

int tridag(const dcomplex *a, const dcomplex *b, const dcomplex *c, const dcomplex *r, dcomplex *u, int n)
Use LAPACK routine ZGTSV.

bool tridag(const BoutReal *a, const BoutReal *b, const BoutReal *c, const BoutReal *r, BoutReal *u, int n)

void cyclic_tridag(BoutReal *a, BoutReal *b, BoutReal *c, BoutReal *r, BoutReal *x, int n)

void cband_solve(Matrix<dcomplex> &a, int n, int m1, int m2, Array<dcomplex> &b)
Complex band matrix solver.

Complex band solver using ZGBSV

n Size of the matrix (number of equations) kl Number of subdiagonals ku Number of superdiagonals nrhs Number of RHS vectors to solve ab Array values (2D array) ldab Leading dim. size of ab = 2*KL+KU+1 ipiv output integer array containing pivot permutation b RHS vectors, and solution ldb length of b (= n) info output status

void cyclic_tridag(dcomplex *a, dcomplex *b, dcomplex *c, dcomplex *r, dcomplex *x, int n)
Solve a cyclic tridiagonal matrix.

K.2.122 File lapack_routines.hxx

Functions

int tridag(const dcomplex *a, const dcomplex *b, const dcomplex *c, const dcomplex *r, dcomplex *u, int n)
Use LAPACK routine ZGTSV.

bool tridag(const BoutReal *a, const BoutReal *b, const BoutReal *c, const BoutReal *r, BoutReal *x, int n)

void cyclic_tridag(BoutReal *a, BoutReal *b, BoutReal *c, BoutReal *r, BoutReal *x, int n)

void cyclic_tridag(dcomplex *a, dcomplex *b, dcomplex *c, dcomplex *r, dcomplex *x, int n)
Solve a cyclic tridiagonal matrix.

void cband_solve(Matrix<dcomplex> &a, int n, int m1, int m2, Array<dcomplex> &b)
Complex band matrix solver.

Complex band solver using ZGBSV

n Size of the matrix (number of equations) kl Number of subdiagonals ku Number of superdiagonals nrhs Number of RHS vectors to solve ab Array values (2D array) ldab Leading dim. size of ab = 2*KL+KU+1 ipiv output integer array containing pivot permutation b RHS vectors, and solution ldb length of b (= n) info output status
K.2.123 File laplacefactory.cxx

Defines

LAPLACE_SPT
LAPLACE_PDD
LAPLACE_TRI
LAPLACE_BAND
LAPLACE_PETSC
LAPLACE_MUMPS
LAPLACE_CYCLIC
LAPLACE_SHOOT
LAPLACE_MULTIGRID
LAPLACE_NAULIN

K.2.124 File laplacefactory.hxx

class LaplaceFactory

Public Functions

Laplacian *createLaplacian(Options *options = nullptr, const CELL_LOC loc = CELL CENTRE, Mesh *mesh_in = nullptr)

Public Static Functions

static LaplaceFactory *getInstance()
    Return a pointer to the only instance.

Private Functions

inline LaplaceFactory()
Private Static Attributes

```cpp
static LaplaceFactory *instance = nullptr
The only instance of this class (Singleton)
```

K.2.125 File laplacexy.cxx

Functions

```cpp
static PetscErrorCode laplacePCapply(PC pc, Vec x, Vec y)
```

K.2.126 File laplacexy.hxx

```cpp
class LaplaceXY

Public Functions

LaplaceXY(Mesh *m = nullptr, Options *opt = nullptr, const CELL_LOC loc = CELL_CENTRE)
Constructor

~LaplaceXY()
Destructor

void setCoefs(const Field2D &A, const Field2D &B)
Set coefficients (A, B) in equation: Div( A * Grad_perp(x) ) + B*x = b

const Field2D solve(const Field2D &rhs, const Field2D &x0)
Solve Laplacian in X-Y

Inputs
rhs - The field to be inverted. This must be allocated and contain valid data. x0 - Initial guess at the solution. If this is unallocated then an initial guess of zero will be used.

Returns
The solution as a Field2D. On failure an exception will be raised

int precon(Vec x, Vec y)
Preconditioner function This is called by PETSc via a static function. and should not be called by external users

Preconditioner NOTE: For efficiency, this routine does not use globalIndex() in the inner loop. Instead, the indexing must be ordered in exactly the same way as in the construction of indexXY

void savePerformance(Datafile &outputfile, Solver &solver, std::string name = "")
If this method is called, save some performance monitoring information
```
**Private Functions**

```c
int localSize()  
    Number of grid points on this processor

MPI_Comm communicator()  
    Return the communicator for XY

int globalIndex(int x, int y)  
    Return the global index of a local (x,y) coordinate including guard cells. Boundary cells have a global index of -1
```

To do this, a `Field2D (indexXY)` is used to store the index as a floating point number which is then rounded to an integer. Guard cells are filled by communication so no additional logic is needed in `Mesh`.

**Private Members**

```text
PetscLib lib  
    Requires PETSc library.

Mat MatA  
    Matrix to be inverted.

Vec xs  

Vec bs  
    Solution and RHS vectors.

KSP ksp  
    Krylov Subspace solver.

PC pc  
    Preconditioner.

Mesh *localmesh  
    The mesh this operates on, provides metrics and communication.
```

```c
int my_id = 0
int xstart
int xend
int nloc
int nsys

Matrix<BoutReal> acoef
Matrix<BoutReal> bcoef
Matrix<BoutReal> ccoef
Matrix<BoutReal> xvals
Matrix<BoutReal> bvals
std::unique_ptr<CyclicReduce<BoutReal>> cr
```

Tridiagonal solver.
bool finite_volume = {true}
bool include_y_derivs
bool x_inner_dirichlet
bool x_outer_dirichlet
std::string y_bndry = {"neumann"}

CELL_LOC location

Field2D indexXY
   Global index (integer stored as BoutReal)

bool save_performance = false
BoutReal average_iterations = 0.
BoutReal output_average_iterations = 0.
int n_calls = 0
LaplaceXYMonitor monitor

Private Static Attributes

static int instance_count = 0

Friends

friend class LaplaceXYMonitor

class LaplaceXYMonitor : public Monitor

Public Functions

inline LaplaceXYMonitor(LaplaceXY &owner)

inline int call(Solver*, BoutReal, int, int)

Private Members

LaplaceXY &laplacexy
K.2.127 File laplacez-cyclic.cxx

K.2.128 File laplacez-cyclic.hxx

class LaplaceXZcyclic : public LaplaceXZ

Public Functions

LaplaceXZcyclic(Mesh *m = nullptr, Options *options = nullptr, const CELL_LOC loc = CELL_CENTRE)

inline ~LaplaceXZcyclic()

virtual void setCoefs(const Field2D &A, const Field2D &B) override

virtual Field3D solve(const Field3D &b, const Field3D &x0) override

Private Members

int xstart
int xend
int nmode
int nloc
int nsys
Matrix<dcomplex> acoef
Matrix<dcomplex> bcoef
Matrix<dcomplex> ccoef
Matrix<dcomplex> xcmplx
Matrix<dcomplex> rhscmplx
Array<dcomplex> kid
Array<dcomplex> kid_2
std::unique_ptr<CyclicReduce<dcomplex>> cr
    Tridiagonal solver.

int inner_boundary_flags
    Flags to set inner boundary condition.

int outer_boundary_flags
    Flags to set outer boundary condition.
K.2.129 File laplacexz-petsc.cxx

K.2.130 File laplacexz-petsc.hxx

class LaplaceXZpetsc : public LaplaceXZ

**Public Functions**

LaplaceXZpetsc(*m = nullptr, *options = nullptr, const CELL_LOC loc = CELL_CENTRE)  
Constructor

~LaplaceXZpetsc()  
Destructor

virtual void setCoefs(const Field3D &A, const Field3D &B)

inline virtual void setCoefs(const Field2D &A, const Field2D &B)

virtual Field3D solve(const Field3D &b, const Field3D &x0)  
Solve Laplacian in X-Z

**Private Members**

*PetscLib* lib  
Requires PETSc library.

*std::vector<YSlice>* slice

Vec xs

Vec bs  
Solution and RHS vectors.

int reuse_limit  
How many times can the preconditioner be reused?

int reuse_count  
How many times has it been reused?

bool coefs_set  
Have coefficients been set?

int inner_boundary_flags  
Flags to set inner boundary condition.

int outer_boundary_flags  
Flags to set outer boundary condition.
Private Static Attributes

static const int implemented_boundary_flags = INVERT_AC_GRAD + INVERT_SET + INVERT_RHS

struct YSlice
    Data for a single Y slice

Public Members

int yindex
Mat MatA
    Y index.
    Matrix to be inverted
Mat MatP
    Matrix for preconditioner.
KSP ksp
    Krylov Subspace solver context.

K.2.131 File laplacexz.cxx

K.2.132 File laplacexz.hxx

class LaplaceXZ
    Subclassed by LaplaceXZcyclic, LaplaceXZpetsc

Public Functions

inline LaplaceXZ(Mesh *m = nullptr, Options *options = nullptr, const CELL_LOC loc = CELL_CENTRE)

virtual ~LaplaceXZ() = default

virtual void setCoefs(const Field2D &A, const Field2D &B) = 0

inline virtual void setCoefs(const Field3D &A, const Field3D &B)

virtual Field3D solve(const Field3D &b, const Field3D &x0) = 0
Public Static Functions

static LaplaceXZ *create(Mesh *m = nullptr, Options *opt = nullptr, const CELL_LOC loc = CELL_CENTRE)

Protected Attributes

Mesh *localmesh
The mesh this operates on, provides metrics and communication.

CELL_LOC location

Protected Static Attributes

static const int INVERT_DC_GRAD = 1
static const int INVERT_AC_GRAD = 2
static const int INVERT_SET = 16
static const int INVERT_RHS = 32

K.2.133 File macro_for_each.hxx

Defines

#define BOUT_EXPAND(x)
Intermediate expansion needed for MSVC due to non-compliant preprocessor.

#define _me_1(_call, x)
_me_x set of macros expand a number of arguments without ';' between them

#define _me_2(_call, x, ...)

#define _me_3(_call, x, ...)

#define _me_4(_call, x, ...)

#define _me_5(_call, x, ...)

#define _me_6(_call, x, ...)

#define _me_7(_call, x, ...)

#define _me_8(_call, x, ...)

#define _me_9(_call, x, ...)
_me_10(_call, x, ...)

_fe_1(_call, x)
    _fe_x set of macros expand a number of arguments with ‘;’ between them
_fe_2(_call, x, ...)

_fe_3(_call, x, ...)

_fe_4(_call, x, ...)

_fe_5(_call, x, ...)

_fe_6(_call, x, ...)

_fe_7(_call, x, ...)

_fe_8(_call, x, ...)

_fe_9(_call, x, ...)

_fe_10(_call, x, ...)

_GET_FOR_EACH_EXPANSION(_1, _2, _3, _4, _5, _6, _7, _8, _9, _10, NAME, ...)
    When called with VA_ARGS first, this evaluates to an argument which depends on the length of VA_ARGS.
    This is used to find the appropriate macro to begin the expansion.

MACRO_FOR_EACH(mac, ...)
    Apply a macro (first argument) to each of the following arguments. Currently supports up to 10 arguments.
    Example:
    MACRO_FOR_EACH(test, a, b, c)
    expands to
    test(a) test(b) test(c)
    Notes:
    • No semicolon is inserted after each expansion
    • No braces are put around the expansion. These should usually be added in the top-level macro to avoid surprising results.

MACRO_FOR_EACH_FN(fn, ...)
    Apply a function (first argument) to each of the following arguments. Currently supports up to 10 arguments.
    Example:
    MACRO_FOR_EACH_FN(test, a, b, c)
    expands to
test(a); test(b); test(c);

Notes:

- A ' is inserted after each expansion
- No braces are put around the expansion. These should usually be added in the top-level macro to avoid surprising results.

**K.2.134 File mask.hxx**

class **BoutMask**

```cpp
#include <mask.hxx>
```

3D array of bools to mask Field3Ds

Wrapper around a 3D vector of bools to enable masking of Field3Ds. Masking is not automatic, but can be accomplished by

```cpp
// Create mask the size of mesh with all false values
BoutMask mask(mesh, false);
// Set an index to true to skip this index
mask(3, 4, 5) = true;
// Iterate over field
for (const auto &index : field) {
  // Skip any indices which are set to true in the mask
  if (mask(index.x, index.y, index.z)) continue;
  ...
}
```

**Public Functions**

```cpp
inline BoutMask(int nx, int ny, int nz, bool value = false)

inline BoutMask(Mesh &mesh, bool value = false)

inline BoutMask()

inline BoutMask &operator=(bool value)

inline bool &operator()(int jx, int jy, int jz)

inline const bool &operator()(int jx, int jy, int jz) const
```
Private Members

\textit{Tensor<bool> mask}

K.2.135 File mesh.cxx

K.2.136 File mesh.hxx

Interface for mesh classes. Contains standard variables and useful routines.

\textit{Changelog}

2014-12 Ben Dudson bd512@york.ac.uk
\begin{itemize}
  \item Removing coordinate system into separate \textit{Coordinates} class
  \item Adding index derivative functions from derivs.cxx
\end{itemize}

2010-06 Ben Dudson, Sean Farley
\begin{itemize}
  \item Initial version, adapted from GridData class
  \item Incorporates code from topology.cpp and Communicator
\end{itemize}

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see \url{http://www.gnu.org/licenses/}.

Typedefs

\begin{verbatim}
using comm_handle = void*
  Type used to return pointers to handles.
\end{verbatim}

\texttt{class Mesh}
\begin{itemize}
  \item Subclassed by \texttt{BoutMesh}
\end{itemize}
**Public Functions**

inline **Mesh()**
Constructor for a “bare”, uninitialised **Mesh** Only useful for testing

**Mesh(GridDataSource *s, Options *options)**
Constructor

**Parameters**
- **s** – [in] The source to be used for loading variables
- **options** – [in] The options section for settings

virtual ~**Mesh()**
Destructor.

inline virtual int **load()**
Loads the mesh values
Currently need to create and load mesh in separate calls because creating Fields uses the global “mesh” pointer which isn’t created until **Mesh** is constructed

inline virtual void **outputVars(Datafile &file)**
Add output variables to a data file These are used for post-processing

**Parameters**
- **sval** – [out] The value will be put into this variable
- **name** – [in] The name of the variable to read
- **def** – [in] The default value if not found

**Returns** zero if successful, non-zero on failure

**Parameters**
- **ival** – [out] The value will be put into this variable
- **name** – [in] The name of the variable to read
- **def** – [in] The default value if not found

**Returns** zero if successful, non-zero on failure

**Parameters**
- **rval** – [out] The value will be put into this variable
- **name** – [in] The name of the variable to read
- **def** – [in] The default value if not found

**Returns** zero if successful, non-zero on failure

**Parameters**
- **bval** – [out] The value will be put into this variable
- **name** – [in] The name of the variable to read
- **def** – [in] The default value if not found

**Returns** zero if successful, non-zero on failure
Parameters

• \texttt{bval} – \texttt{[out]} The value will be put into this variable
• \texttt{name} – \texttt{[in]} The name of the variable to read
• \texttt{def} – \texttt{[in]} The default value if not found

Returns zero if successful, non-zero on failure

\begin{verbatim}
int get(\texttt{Field2D} &var, const \texttt{std::string} &name, \texttt{BoutReal} def = 0.0)
\end{verbatim}

Get a Field2D from the input source including communicating guard cells

Parameters

• \texttt{var} – \texttt{[out]} This will be set to the value. Will be allocated if needed
• \texttt{name} – \texttt{[in]} Name of the variable to read
• \texttt{def} – \texttt{[in]} The default value if not found

Returns zero if successful, non-zero on failure

\begin{verbatim}
int get(\texttt{Field3D} &var, const \texttt{std::string} &name, \texttt{BoutReal} def = 0.0, \texttt{bool} communicate = true)
\end{verbatim}

Get a Field3D from the input source

Parameters

• \texttt{var} – \texttt{[out]} This will be set to the value. Will be allocated if needed
• \texttt{name} – \texttt{[in]} Name of the variable to read
• \texttt{def} – \texttt{[in]} The default value if not found
• \texttt{communicate} – \texttt{[in]} Should the field be communicated to fill guard cells?

Returns zero if successful, non-zero on failure

\begin{verbatim}
int get(\texttt{FieldPerp} &var, const \texttt{std::string} &name, \texttt{BoutReal} def = 0.0, \texttt{bool} communicate = true)
\end{verbatim}

Get a FieldPerp from the input source

Parameters

• \texttt{var} – \texttt{[out]} This will be set to the value. Will be allocated if needed
• \texttt{name} – \texttt{[in]} Name of the variable to read
• \texttt{def} – \texttt{[in]} The default value if not found
• \texttt{communicate} – \texttt{[in]} Should the field be communicated to fill guard cells?

Returns zero if successful, non-zero on failure

\begin{verbatim}
int get(\texttt{Vector2D} &var, const \texttt{std::string} &name, \texttt{BoutReal} def = 0.0)
\end{verbatim}

Get a Vector2D from the input source. If \texttt{var} is covariant then this gets three Field2D variables with “_x”, “_y”, “_z” appended to \texttt{name} If \texttt{var} is contravariant, then “x”, “y”, “z” are appended to \texttt{name}

By default all fields revert to zero

Parameters

• \texttt{var} – \texttt{[in]} This will be set to the value read
• \texttt{name} – \texttt{[in]} The name of the vector. Individual fields are read based on this name by appending. See above
• \texttt{def} – \texttt{[in]} The default value if not found (used for all the components)

Returns zero always.
int get(Vector3D &var, const std::string &name, BoutReal def = 0.0)
Get a Vector3D from the input source. If var is covariant then this gets three Field3D variables with “_x”, “_y”, “_z” appended to name. If var is contravariant, then “x”, “y”, “z” are appended to name.

By default all fields revert to zero.

Parameters

- var – [in] This will be set to the value read
- name – [in] The name of the vector. Individual fields are read based on this name by appending. See above
- def – [in] The default value if not found (used for all the components)

Returns zero always.

bool isDataSourceGridFile() const
Test if input source was a grid file.

bool sourceHasVar(const std::string &name)
Wrapper for GridDataSource::hasVar.

bool sourceHasXBoundaryGuards()
Wrapper for GridDataSource::hasXBoundaryGuards.

bool sourceHasYBoundaryGuards()
Wrapper for GridDataSource::hasYBoundaryGuards.

template<typename ...Ts>
inline void communicate(Ts&... ts)
Communicate a list of FieldData objects. Uses a variadic template (C++11) to pack all arguments into a FieldGroup.

template<typename ...Ts>
inline void communicateXZ(Ts&... ts)

void communicate(FieldGroup &g)
Communicate a group of fields.

void communicateXZ(FieldGroup &g)
Communicate guard cells in XZ only i.e. no Y communication.

Parameters g – The group of fields to communicate. Guard cells will be modified.

void communicate(FieldPerp &f)
Communicate an X-Z field.

This is a bit of a hack for now to get FieldPerp communications. The FieldData class needs to be changed to accommodate FieldPerp objects.

template<typename ...Ts>
inline comm_handle send(Ts&... ts)
Send a list of FieldData objects. Packs arguments into a FieldGroup and passes to send(FieldGroup&).

virtual comm_handle send(FieldGroup &g) = 0
Perform communications without waiting for them to finish. Requires a call to wait() afterwards.

Parameters g – Group of fields to communicate.

Returns handle to be used as input to wait().

virtual int wait(comm_handle handle) = 0
Wait for the handle, return error code.
virtual MPI_Request sendToProc(int xproc, int yproc, BoutReal *buffer, int size, int tag) = 0
Low-level communication routine Send a buffer of data from this processor to another This must be matched by a corresponding call to receiveFromProc on the receiving processor

Parameters
• xproc – [in] X index of processor to send to
• yproc – [in] Y index of processor to send to
• buffer – [in] A buffer of data to send
• size – [in] The length of buffer
• tag – [in] A label, must be the same at receive

virtual comm_handle receiveFromProc (int xproc, int yproc, BoutReal *buffer, int size, int tag) = 0
Low-level communication routine Receive a buffer of data from another processor Must be matched by corresponding sendToProc call on the sending processor

Parameters
• xproc – [in] X index of sending processor
• yproc – [in] Y index of sending processor
• buffer – [inout] The buffer to fill with data. Must already be allocated of length size
• size – [in] The length of buffer
• tag – [in] A label, must be the same as send

virtual int getNXPE () = 0
The number of processors in the X direction.

virtual int getNYPE () = 0
The number of processors in the Y direction.

virtual int getXProcIndex () = 0
This processor’s index in X direction.

virtual int getYProcIndex () = 0
This processor’s index in Y direction.

virtual bool firstX () = 0
Is this processor first in X? i.e. is there a boundary to the left in X?

virtual bool lastX () = 0
Is this processor last in X? i.e. is there a boundary to the right in X?

virtual int sendXOut (BoutReal *buffer, int size, int tag) = 0
Send a buffer of data to processor at X index +1

Parameters
• buffer – [in] The data to send. Must be at least length size
• size – [in] The number of BoutReals to send
• tag – [in] A label for the communication. Must be the same at receive

virtual int sendXIn (BoutReal *buffer, int size, int tag) = 0
Send a buffer of data to processor at X index -1

Parameters
• **buffer** – [in] The data to send. Must be at least length `size`
• **size** – [in] The number of BoutReals to send
• **tag** – [in] A label for the communication. Must be the same at receive

```cpp
virtual comm_handle irecvXOut(BoutReal *buffer, int size, int tag) = 0
```
Receive a buffer of data from X index +1

**Parameters**

- **buffer** – [in] A buffer to put the data in. Must already be allocated of length `size`
- **size** – [in] The number of BoutReals to receive and put in `buffer`
- **tag** – [in] A label for the communication. Must be the same as sent

```cpp
virtual comm_handle irecvXIn(BoutReal *buffer, int size, int tag) = 0
```
Receive a buffer of data from X index -1

**Parameters**

- **buffer** – [in] A buffer to put the data in. Must already be allocated of length `size`
- **size** – [in] The number of BoutReals to receive and put in `buffer`
- **tag** – [in] A label for the communication. Must be the same as sent

```cpp
inline MPI_Comm getXcomm()
```
Return communicator containing all processors in X.

```cpp
virtual MPI_Comm getXcomm(int jy) const = 0
```
Return X communicator.

```cpp
virtual MPI_Comm getYcomm(int jx) const = 0
```
Return Y communicator.

```cpp
virtual bool periodicY(int jx) const
```
Is local X index `jx` periodic in Y?

**Parameters**

- **jx** – [in] The local (on this processor) index in X

```cpp
virtual bool periodicY(int jx, BoutReal &ts) const = 0
```
Is local X index `jx` periodic in Y?

**Parameters**

- **jx** – [in] The local (on this processor) index in X
- **ts** – [out] The Twist-Shift angle if periodic

```cpp
virtual std::pair<bool, BoutReal> hasBranchCutLower(int jx) const = 0
```
Is there a branch cut at this processor’s lower y-boundary?

**Parameters**

- **jx** – [in] The local (on this processor) index in X

```cpp
Returns pair<bool, BoutReal> - bool is true if there is a branch cut, BoutReal gives the total zShift for a 2pi poloidal circuit if there is a branch cut
```

```cpp
virtual std::pair<bool, BoutReal> hasBranchCutUpper(int jx) const = 0
```
Is there a branch cut at this processor’s upper y-boundary?

**Parameters**

- **jx** – [in] The local (on this processor) index in X

```cpp
Returns pair<bool, BoutReal> - bool is true if there is a branch cut, BoutReal gives the total zShift for a 2pi poloidal circuit if there is a branch cut
```
virtual int ySize(int jx) const  
The number of points in Y at fixed X index jx.

virtual bool firstY() const = 0  
Is this processor first in Y? i.e. is there a boundary at lower Y?

virtual bool lastY() const = 0  
Is this processor last in Y? i.e. is there a boundary at upper Y?

virtual bool firstY(int xpos) const = 0  
Is this processor first in Y? i.e. is there a boundary at lower Y?

virtual bool lastY(int xpos) const = 0  
Is this processor last in Y? i.e. is there a boundary at upper Y?

virtual int UpXSplitIndex() = 0  
If the upper Y guard cells are split in two, return the X index where the split occurs.

virtual int DownXSplitIndex() = 0  
If the lower Y guard cells are split in two, return the X index where the split occurs.

virtual bool sendYOutIndest(BoutReal *buffer, int size, int tag) = 0  
Send data.

virtual bool sendYOutOutdest(BoutReal *buffer, int size, int tag) = 0

virtual bool sendYInIndest(BoutReal *buffer, int size, int tag) = 0

virtual bool sendYInOutdest(BoutReal *buffer, int size, int tag) = 0

virtual comm_handle irecvYOutIndest(BoutReal *buffer, int size, int tag) = 0  
Non-blocking receive. Must be followed by a call to \texttt{wait()}

Parameters

- \texttt{buffer} – [out] A buffer of length \texttt{size} which must already be allocated
- \texttt{size} – [in] The number of BoutReals expected
- \texttt{tag} – [in] The tag number of the expected message

virtual comm_handle irecvYOutOutdest(BoutReal *buffer, int size, int tag) = 0
Non-blocking receive. Must be followed by a call to \texttt{wait()}

Parameters

- \texttt{buffer} – [out] A buffer of length \texttt{size} which must already be allocated
- \texttt{size} – [in] The number of BoutReals expected
- \texttt{tag} – [in] The tag number of the expected message

virtual comm_handle irecvYInIndest(BoutReal *buffer, int size, int tag) = 0
Non-blocking receive. Must be followed by a call to \texttt{wait()}

Parameters

- \texttt{buffer} – [out] A buffer of length \texttt{size} which must already be allocated
- \texttt{size} – [in] The number of BoutReals expected
- \texttt{tag} – [in] The tag number of the expected message
virtual comm_handle irecYIOutdest(BoutReal *buffer, int size, int tag) = 0
    Non-blocking receive. Must be followed by a call to wait()

Parameters

- **buffer** – [out] A buffer of length size which must already be allocated
- **size** – [in] The number of BoutReals expected
- **tag** – [in] The tag number of the expected message

virtual const Rangelttorator iterateBndryLowerY() const = 0
    Iterate over the lower Y boundary.

virtual const Rangelttorator iterateBndryUpperY() const = 0
    Iterate over the upper Y boundary.

virtual const Rangelttorator iterateBndryLowerOuterY() const = 0

virtual const Rangelttorator iterateBndryLowerInnerY() const = 0

virtual const Rangelttorator iterateBndryUpperOuterY() const = 0

virtual const Rangelttorator iterateBndryUpperInnerY() const = 0

bool hasBndryLowerY()
    Is there a boundary on the lower guard cells in Y?

bool hasBndryUpperY()
    Is there a boundary on the upper guard cells in Y?

virtual std::vector<BoundaryRegion*> getBoundaries() = 0
    Return a vector containing all the boundary regions on this processor.

inline virtual void addBoundary(BoundaryRegion *bndry)
    Add a boundary region to this processor.

virtual std::vector<BoundaryRegionPar*> getBoundariesPar() = 0
    Get all the parallel (Y) boundaries on this processor.

inline virtual void addBoundaryPar(BoundaryRegionPar *bndry)
    Add a parallel(Y) boundary to this processor.

inline virtual const Field3D smoothSeparatrix(const Field3D &f)
    Branch-cut special handling (experimental)

virtual BoutReal GlobalX(int jx) const = 0
    Continuous X index between 0 and 1.

virtual BoutReal GlobalY(int jy) const = 0
    Continuous Y index (0 -> 1)

virtual BoutReal GlobalX(BoutReal jx) const = 0
    Continuous X index between 0 and 1.

virtual BoutReal GlobalY(BoutReal jy) const = 0
    Continuous Y index (0 -> 1)

inline int XGLOBAL(int xloc) const
    Returns the global X index given a local index If the local index includes the boundary cells, then so does the global.
inline int YGLOBAL(int yloc) const
    Returns the global Y index given a local index. The local index must include the boundary, the global index does not.

inline int XLOCAL(int xglo) const
    Returns the local X index given a global index. If the global index includes the boundary cells, then so does the local.

inline int YLOCAL(int yglo) const
    Returns the local Y index given a global index. If the global index includes the boundary cells, then so does the local.

virtual int getGlobalXIndex(int xlocal) const = 0
    Returns a global X index given a local index. Global index includes boundary cells, local index includes boundary or guard cells.

virtual int getGlobalXIndexNoBoundaries(int xlocal) const = 0
    Returns a global X index given a local index. Global index excludes boundary cells, local index includes boundary or guard cells.

virtual int getLocalXIndex(int xglobal) const = 0
    Returns a local X index given a global index. Global index includes boundary cells, local index includes boundary or guard cells.

virtual int getLocalXIndexNoBoundaries(int xglobal) const = 0
    Returns a local X index given a global index. Global index excludes boundary cells, local index includes boundary or guard cells.

virtual int getGlobalYIndex(int ylocal) const = 0
    Returns a global Y index given a local index. Global index includes boundary cells, local index includes boundary or guard cells.

virtual int getGlobalYIndexNoBoundaries(int ylocal) const = 0
    Returns a global Y index given a local index. Global index excludes boundary cells, local index includes boundary or guard cells.

virtual int getLocalYIndex(int yglobal) const = 0
    Returns a local Y index given a global index. Global index includes boundary cells, local index includes boundary or guard cells.

virtual int getLocalYIndexNoBoundaries(int yglobal) const = 0
    Returns a local Y index given a global index. Global index excludes boundary cells, local index includes boundary or guard cells.

virtual int getGlobalZIndex(int zlocal) const = 0
    Returns a global Z index given a local index. Global index includes boundary cells, local index includes boundary or guard cells.

virtual int getGlobalZIndexNoBoundaries(int zlocal) const = 0
    Returns a global Z index given a local index. Global index excludes boundary cells, local index includes boundary or guard cells.

virtual int getLocalZIndex(int zglobal) const = 0
    Returns a local Z index given a global index. Global index includes boundary cells, local index includes boundary or guard cells.

virtual int getLocalZIndexNoBoundaries(int zglobal) const = 0
    Returns a local Z index given a global index. Global index excludes boundary cells, local index includes boundary or guard cells.
inline Coordinates *getCoordinates(const CELL_LOC location = CELL_CENTRE)
    Coordinate system.
inline std::shared_ptr<Coordinates> getCoordinatesSmart(const CELL_LOC location = CELL_CENTRE)

inline CELL_LOC getAllowedStaggerLoc(DIRECTION direction) const
    Returns the non-CELL_CENTRE location allowed as a staggered location
inline int getNpoints(DIRECTION direction) const
    Returns the number of grid points in the particular direction
inline int getNguard(DIRECTION direction) const
    Returns the number of guard points in the particular direction

void recalculateStaggeredCoordinates()
    Re-calculate staggered Coordinates, useful if CELL_CENTRE Coordinates are changed.
STAGGER getStagger(const CELL_LOC inloc, const CELL_LOC outloc, const CELL_LOC allowedloc) const
    Determines the resultant output stagger location in derivatives given the input and output location. Also checks that the combination of locations is allowed
STAGGER getStagger(const CELL_LOC vloc, const CELL_LOC inloc, const CELL_LOC outloc, const CELL_LOC allowedloc) const
    Determines the resultant output stagger location in derivatives given the input and output location. Also checks that the combination of locations is allowed. This overload also checks the location of a second input field (velocity) is consistent.

template<typename T>
inline T indexDDX(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

template<typename T>
inline T indexD2DX2(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

template<typename T>
inline T indexD4DX4(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

template<typename T>
inline T indexDDY(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

template<typename T>
inline T indexD2DY2(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

template<typename T>
inline T indexD4DY4(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const
inline `T indexDDZ(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

template<typename T>
inline `T indexD2DZ2(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

template<typename T>
inline `T indexD4DZ4(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

template<typename T>
inline `T indexVDDX(const T &vel, const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

Advection operator in index space in [] direction

\[
\frac{d}{dt} f
\]

Parameters

• `v` – [in] The velocity in the Y direction
• `f` – [in] The field being advected
• `outloc` – [in] The cell location where the result is desired. The default is the same as `f`
• `method` – [in] The differencing method to use
• `region` – [in] The region of the grid for which the result is calculated.

template<typename T>
inline `T indexFDDX(const T &vel, const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

template<typename T>
inline `T indexVDDY(const T &vel, const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

template<typename T>
inline `T indexFDDY(const T &vel, const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

template<typename T>
inline `T indexFDDZ(const T &vel, const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

template<typename T>
inline `T indexFDDZ(const T &vel, const T &f, CELLLOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const
inline const Field3D toFieldAligned(const Field3D &f, const REGION region = RGN_ALL)

inline const Field3D fromFieldAligned(const Field3D &f, const REGION region = RGN_ALL)

inline const Field2D toFieldAligned(const Field2D &f, const REGION region = RGN_ALL)

inline const Field2D fromFieldAligned(const Field2D &f, const REGION region = RGN_ALL)

inline bool canToFromFieldAligned()

inline void setParallelTransform(std::unique_ptr<ParallelTransform> pt)

inline void setParallelTransform()

inline ParallelTransform &getParallelTransform()

template<class T>
const Region<typename T::ind_type> &getRegion(const std::string &region_name) const
    Get the named region from the region_map for the data iterator
    Throws if region_name not found
inline const Region &getRegion(const std::string &region_name) const

const Region<Ind3D> &getRegion3D(const std::string &region_name) const

const Region<Ind2D> &getRegion2D(const std::string &region_name) const

const Region<IndPerp> &getRegionPerp(const std::string &region_name) const

bool hasRegion3D(const std::string &region_name) const
    Indicate if named region has already been defined.
bool hasRegion2D(const std::string &region_name) const

bool hasRegionPerp(const std::string &region_name) const

inline void addRegion(const std::string &region_name, const Region<> &region)
    Add a new region to the region_map for the data iterator
    Outputs an error message if region_name already exists
inline void addRegion(const std::string &region_name, const Region<Ind2D> &region)

inline void addRegion(const std::string &region_name, const Region<IndPerp> &region)

void addRegion3D(const std::string &region_name, const Region<Ind3D> &region)
void addRegion2D(const std::string &region_name, const Region<Ind2D> &region)

void addRegionPerp(const std::string &region_name, const Region<IndPerp> &region)

inline Ind3D ind2Dto3D(const Ind2D &ind2D, int jz = 0)
    Converts an Ind2D to an Ind3D using calculation.

inline Ind2D ind3Dto2D(const Ind3D &ind3D)
    Converts an Ind3D to an Ind2D using calculation.

inline IndPerp ind3DtoPerp(const Ind3D &ind3D)
    Converts an Ind3D to an IndPerp using calculation.

inline Ind3D indPerpto3D(const IndPerp &indPerp, int jy = 0)
    Converts an IndPerp to an Ind3D using calculation.

inline Ind2D map3Dto2D(const Ind3D &ind3D)
    Converts an Ind3D to an Ind2D representing a 2D index using a lookup to be used with care.

void createDefaultRegions()
    Create the default regions for the data iterator
    Creates RGN_{ALL,NOBNDRY,NOX,NOY}

template<>
inline const Region<Ind3D> &getRegion(const std::string &region_name) const

template<>
inline const Region<Ind2D> &getRegion(const std::string &region_name) const

template<>
inline const Region<IndPerp> &getRegion(const std::string &region_name) const

**Public Members**

bool periodicX = {false}
    Domain is periodic in X?

int NXPE

int PE_XIND
    Number of processors in X, and X processor index.

int GlobalNx

int GlobalNy

int GlobalNz
    Size of the global arrays. Note: can have holes.

int OffsetX

int OffsetY
int OffsetZ
    Offset of this mesh within the global array so startx on this processor is OffsetX in global

int LocalNx
    Size of the mesh on this processor including guard/boundary cells.

int LocalNy
int LocalNz
int xstart
    Local ranges of data (inclusive), excluding guard cells.

int xend
int ystart
int yend
int zstart
int zend
bool StaggerGrids = {false}
    Enable staggered grids (Centre, Lower). Otherwise all vars are cell centred (default).

bool IncIntShear = {false}
    Include integrated shear (if shifting X)

int numberOfXPoints = {0}

BoutReal fft_derivs_filter = {0.0}
    Fraction of modes to filter. This is set in derivs_init from option “ddz:fft_filter”.

int maxregionblocksize

Public Static Functions

static Mesh *create(GridDataSource *source, Options *opt = nullptr)
    Create a Mesh object

Parameters
    • source – [in] The data source to use for loading variables
    • opt – [in] The option section. By default this is “mesh”

static Mesh *create(Options *opt = nullptr)
    Create a Mesh object

    The source is determined by 1) If “file” is set in the options, read that 2) If “grid” is set in global options, read that 3) Use options as data source

    Parameters opt – [in] Input options. Default is “mesh” section
Protected Functions

const std::vector<int> readInts(const std::string &name, int n)
    Read a 1D array of integers.

int msg_len(const std::vector<FieldData*> &var_list, int xge, int xlt, int yge, int ylt)
    Calculates the size of a message for a given x and y range.

void derivs_init(Options *options)
    Initialise derivatives.
    Initialise the derivative methods. Must be called before any derivatives are used.

Protected Attributes

GridDataSource *source = {nullptr}
    Source for grid data.

std::map<CELL_LOC, std::shared_ptr<Coordinates>> coords_map
    Coordinate systems at different CELL_LOCs.

Options *options = {nullptr}
    Mesh options section.

bool calcParallelSlices_on_communicate = {true}
    Set whether to call calcParallelSlices on all communicated fields (true) or not (false)

Private Functions

std::shared_ptr<Coordinates> createDefaultCoordinates(const CELL_LOC location, bool force_interpolate_from_centre = false)
    Allocates default Coordinates objects By default attempts to read staggered Coordinates from grid data source, interpolating from CELL_CENTRE if not present. Set force_interpolate_from_centre argument to true to always interpolate (useful if CELL_CENTRE Coordinates have been changed, so reading from file would not be correct).

Private Members

std::map<std::string, Region<Ind3D>> regionMap3D
std::map<std::string, Region<Ind2D>> regionMap2D
std::map<std::string, Region<IndPerp>> regionMapPerp
Array<int> indexLookup3Dto2D
K.2.137 File meshfactory.cxx

Defines

MESH_BOUT
    Name of BoutMesh for Options.

K.2.138 File meshfactory.hxx

class MeshFactory : private Uncopyable

    Public Functions

        Mesh *createMesh(GridDataSource *source, Options *options = nullptr)

    Public Static Functions

        static MeshFactory *getInstance()
            Return a pointer to the only instance.

    Private Functions

        inline MeshFactory()

    Private Static Attributes

        static MeshFactory *instance = nullptr
            The only instance of this class (Singleton)

K.2.139 File monitor.hxx

Functions

    inline bool isMultiple(BoutReal a, BoutReal b)
        Return true if either a is a multiple of b or vice-versa
        Assumes both arguments are greater than zero

    class Monitor
        #include <monitor.hxx> Monitor baseclass for the Solver
        Can be called ether with a specified frequency, or with the frequency of the BOUT++ output monitor.
        Subclassed by BoutMonitor, PhysicsModel::PhysicsModelMonitor
Public Functions

inline Monitor(BoutReal timestep_ = -1)
   A timestep_ of -1 defaults to the the frequency of the BOUT++ output monitor

virtual ~Monitor() = default

virtual int call(Solver *solver, BoutReal time, int iter, int nout) = 0
   Callback function for the solver, called after timestep_ has passed

Parameters
   • solver – [in] The solver calling this monitor
   • time – [in] The current simulation time
   • iter – [in] The current simulation iteration
   • nout – [in] The total number of iterations for this simulation

Returns non-zero if simulation should be stopped

inline virtual void cleanup()
   Callback function for when a clean shutdown is initiated.

Protected Functions

inline BoutReal getTimestep() const
   Get the currently set timestep for this monitor.

inline void setTimestep(BoutReal new_timestep)
   Set the timestep for this Monitor

   Can only be called before the Monitor is added to a Solver

Private Members

bool is_added = {false}
   Set to true when Monitor is added to a Solver.

BoutReal timestep = { -1 }
   The desired physical timestep.

int period = { 1 }
   How often this monitor should be called, in internal Solver steps.
**Friends**

friend class Solver

struct RunMetrics

**Public Functions**

void outputVars(Datafile &file)
    Adds variables to the output file, for post-processing

void calculateDerivedMetrics()
    Calculates derived metrics

void writeProgress(BoutReal simtime, bool output_split)
    Write job progress to screen

**Public Members**

BoutReal t_elapsed = 0
    cumulative wall clock time in seconds

BoutReal wtime = 0
    time step’s wall clock time in seconds

int ncalls = 0
    number of RHS calls

int ncalls_e = 0
    number of RHS calls for fast timescale

int ncalls_i = 0
    number of RHS calls for slow timescale

BoutReal wtime_rhs = 0
    wall time spent calculating RHS

BoutReal wtime_invert = 0
    wall time spent inverting Laplacian

BoutReal wtime_comms = 0
    wall time spent communicating (part of RHS)

BoutReal wtime_io = 0
    wall time spent on I/O

BoutReal wtime_per_rhs = 0
    wall time per RHS evaluation

BoutReal wtime_per_rhs_e = 0
    wall time per fast timescale RHS evaluation
$BoutReal$ \texttt{wtime\_per\_rhs\_i} = 0

wall time per slow timescale RHS evaluation

K.2.140 File \textit{monotonic\_hermite\_spline.cxx}

K.2.141 File \textit{msg\_stack.cxx}

K.2.142 File \textit{msg\_stack.hxx}

Defines

\texttt{MSG\_MAX\_SIZE}

The maximum length (in chars) of messages, not including terminating ‘0’.

\_\_\_\texttt{thefunc\_\_}\_

The \texttt{PRETTY\_FUNCTION} variable is defined by GCC (and some other families) but is not a part of the standard. The \texttt{func} variable is a part of the c++11 standard so we’d like to fall back to this if possible. However as these are variables/constants and not macros we can’t just check if \texttt{PRETTY\_FUNCTION} is defined or not. Instead we need to say if we support this or not by defining \texttt{HAS\_PRETTY\_FUNCTION} (to be implemented in configure)

\texttt{GLOBAL}

This is a way to define a global object, so that it is declared extern in all files except one where \texttt{GLOBALORIGIN} is defined.

\texttt{CONCATENATE\_DIRECT(s1, s2)}

To concatenate strings for a variable name.

\texttt{CONCATENATE(s1, s2)}

Need to use two levels due to macro strangeness.

\texttt{TRACE(...)}

The \texttt{TRACE} macro provides a convenient way to put messages onto the \texttt{msg\_stack} It pushes a message onto the stack, and pops it when the scope ends

\textit{Example}

\{
  TRACE(“Starting calculation”)
\} // Scope ends, message popped

\texttt{AUTO\_TRACE()}

The \texttt{AUTO\_TRACE} macro provides a convenient way to put messages onto the \texttt{msg\_stack} It pushes a message onto the stack, and pops it when the scope ends The message is automatically derived from the function signature as identified by the compiler. This will be \texttt{PRETTY\_FUNCTION} if available else it will be the mangled form.

This is implemented as a use of the \texttt{TRACE} macro with specific arguments.

\textit{Example}

\{
  AUTO\_TRACE();
\} // Scope ends, message popped
Variables

(MsgStack msg_stack)

Global object. Will eventually replace with better system.

class MsgStack
   #include <msg_stack.hxx> Message stack

   Implements a stack of messages which can be pushed onto the top and popped off the top. This is used for
debugging: messages are put into this stack at the start of a section of code, and removed at the end. If an error
occurs in between push and pop, then the message can be printed.

   This code is only enabled if CHECK > 1. If CHECK is disabled then this message stack code reverts to empty
functions which should be removed by the optimiser

Public Functions

(MsgStack() = default

inline ~MsgStack()

int push(const char *, ...)
   Add a message to the stack. Returns a message id.

   Provides a message stack to print more useful error messages.

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser
General Public License as published by the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even
the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not,
see http://www.gnu.org/licenses/.

int setPoint()
   get a message point

void pop()
   Remove the last message.

void pop(int id)
   Remove all messages back to msg id.

void clear()
   Clear all message.

void dump()
   Write out all messages (using output)
std::string getDump()
   Write out all messages to a string.

Private Members

char buffer[256]
   Buffer for vsnprintf.

std::vector<std::string> stack
   Message stack.

std::vector<std::string>::size_type position = {0}
   Position in stack.

class MsgStackItem
   #include <msg_stack.hxx> MsgStackItem
   Simple class to manage pushing and popping messages from the message stack. Pushes a message in the constructor, and pops the message on destruction.

Public Functions

inline MsgStackItem(const char *msg)

inline MsgStackItem(const char *msg, const char *file, int line)

inline MsgStackItem(const char *file, int line, const char *msg, ...)

inline ~MsgStackItem()

Private Members

int point
char buffer[256]

K.2.143 File multigrid_alg.cxx

K.2.144 File multigrid_laplace.cxx

Variables

BoutReal soltime = 0.0
BoutReal settime = 0.0
K.2.145 File multigrid_laplace.hxx

Defines

**MAXGM**

class **MultigridAlg**

Subclassed by **Multigrid1DP, Multigrid2DPf1D, MultigridSerial**

**Public Functions**

**MultigridAlg**(int, int, int, int, MPI_Comm, int)

virtual ~**MultigridAlg**()

void **setMultigridC**(int)

void **getSolution**( BoutReal*, BoutReal*, int)

**Public Members**

int **mglevel**
int **mgplag**
int **cftype**
int **mgsm**
int **pcheck**
int **xNP**
int **zNP**
int **rProcI**
BoutReal **rtol**
BoutReal **atol**
BoutReal **dtol**
BoutReal **omega**
Array<int> **gnx**
Array<int> **gnz**
Array<int> **lnx**
Array<int> **lnz**
BoutReal **matmg**
Protected Functions

void communicate(BoutReal*, int)

void setMatrixC(int)

void cycleMG(int, BoutReal*, BoutReal*)

void smoothings(int, BoutReal*, BoutReal*)

void projection(int, BoutReal*, BoutReal*)

void prolongation(int, BoutReal*, BoutReal*)

void pGMRES(BoutReal*, BoutReal*, int, int)

void solveMG(BoutReal*, BoutReal*, int)

void multiAVec(int, BoutReal*, BoutReal*)

void residualVec(int, BoutReal*, BoutReal*, BoutReal*)

BoutReal vectorProd(int, BoutReal*, BoutReal*)

virtual void lowestSolver(BoutReal*, BoutReal*, int)

Protected Attributes

int numP
int xProcI
int zProcI
int xProcP
int xProcM
int zProcP
int zProcM
MPI_Comm commMG

class MultigridSerial : public MultigridAlg
**Public Functions**

**MultigridSerial**(int level, int gx, int gz, MPI_Comm comm, int check)

inline **~MultigridSerial**()

void **convertMatrixF**(BoutReal*)

class **Multigrid2DPf1D** : public **MultigridAlg**

**Public Functions**

**Multigrid2DPf1D**(int, int, int, int, int, int, int, MPI_Comm, int)

inline **~Multigrid2DPf1D**()

void **setMultigridC**(int)

void **setPcheck**(int)

void **setValueS**()

**Public Members**

int **kflag**

**Private Functions**

void **convertMatrixFS**(int)

virtual void **lowestSolver**(BoutReal*, BoutReal*, int)

**Private Members**

std::unique_ptr<MultigridSerial> **sMG**

class **Multigrid1DP** : public **MultigridAlg**
**Public Functions**

`Multigrid1DP(int, int, int, int, int, int, MPI_Comm, int)`

`inline ~Multigrid1DP()`

void `setMultigridC(int)`

void `setPcheck(int)`

void `setValueS()`

**Public Members**

int `kflag`

**Private Functions**

void `convertMatrixF2D(int)`

void `convertMatrixFS(int)`

virtual void `lowestSolver(BoutReal*, BoutReal*, int)`

**Private Members**

`MPI_Comm comm2D`

`std::unique_ptr<MultigridSerial> sMG`

`std::unique_ptr<Multigrid2DPf1D> rMG`

class `LaplaceMultigrid` : public Laplacian

**Public Functions**

`LaplaceMultigrid(Options *opt = nullptr, const CELL_LOC loc = CELL_CENTRE, Mesh *mesh_in = nullptr)`

`inline ~LaplaceMultigrid()`

`inline virtual void setCoefA(const Field2D &val) override`

`Set coefficients for inversion. Re-builds matrices if necessary.`

`inline virtual void setCoefC(const Field2D &val) override`
inline virtual void `setCoefC1` (const `Field2D` &val) override

inline virtual void `setCoefC2` (const `Field2D` &val) override

inline virtual void `setCoefD` (const `Field2D` &val) override

inline virtual void `setCoefEx` (const `Field2D` &val) override

inline virtual void `setCoefEz` (const `Field2D` &val) override

inline virtual void `setCoefA` (const `Field3D` &val) override

inline virtual void `setCoefC` (const `Field3D` &val) override

inline virtual void `setCoefC1` (const `Field3D` &val) override

inline virtual void `setCoefC2` (const `Field3D` &val) override

inline virtual void `setCoeffD` (const `Field3D` &val) override

inline virtual bool `uses3DCoefs` () const override

    Does this solver use `Field3D` coefficients (true) or only their DC component (false)

inline virtual `FieldPerp` `solve` (const `FieldPerp` &b) override

virtual `FieldPerp` `solve` (const `FieldPerp` &b_in, const `FieldPerp` &x0) override

**Private Functions**

void `generateMatrixF` (int)

**Private Members**

`Field3D A`

`Field3D C1`

`Field3D C2`

`Field3D D`

int `Nx_local`

int `Nx_global`

int `Nz_local`

int `Nz_global`
int yindex
Array<BoutReal> x
Array<BoutReal> b
std::unique_ptr< Multigrid1DP > kMG
int mglevel
int mgplag
int cftype
int mgsm
int pcheck
int mgcount
int mgmpi
Options *opts
BoutReal rtol
BoutReal atol
BoutReal dtol
BoutReal omega
MPI_Comm commX
int comms_tagbase

K.2.146 File multigrid_solver.cxx

K.2.147 File multiostream.hxx

Typedefs

using cmultiostream = multiostream<char>
using wmultiostream = multiostream<wchar_t>
template< typename char_type, typename traits = std::traits< char_type > >
class multioutbuf : public std::basic_streambuf< char_type, traits >
#include <multiostream.hxx> Template class to split streams.
from http://accu.org/index.php/journals/260
Public Functions

inline void add(std::basic_ostream<char_type, traits> &str)

inline void remove(std::basic_ostream<char_type, traits> &str)

Protected Functions

inline std::streamsize xsputn(const char_type *sequence, std::streamsize num) override

inline int overflow(int c) override

Private Types

using stream_container = std::vector<std::basic_ostream<char_type, traits>>*

Private Members

stream_container streams_

template<typename char_type, typename traits>
class multioutbuf_init
    Subclassed by multiostream< char_type, traits >, Output

Public Functions

inline multioutbuf <char_type, traits> *buf()

Private Members

multioutbuf <char_type, traits> buf_

template<typename char_type, typename traits = std::char_traits<char_type>>
class multiostream : private multioutbuf_init<char_type, traits>, public std::basic_ostream<char_type, traits>
Public Functions

inline multiostream()

inline void add(std::basic_ostream<char_type, traits> &str)

inline void remove(std::basic_ostream<char_type, traits> &str)

Private Types

using multioutbuf_init = ::multioutbuf_init<char_type, traits>

K.2.148 File mumps_laplace.cxx

K.2.149 File mumps_laplace.hxx

Defines

MUMPS_JOB_INIT
MUMPS_JOB_END
MUMPS_JOB_ANALYSIS
MUMPS_JOB_FACTORIZATION
MUMPS_JOB_SOLUTION
MUMPS_JOB_ANALYSIS_AND_FACTORIZATION
MUMPS_JOB_BOTH
MUMPS_JOB_ALL

class LaplaceMumps : public Laplacian

Public Functions

LaplaceMumps(Options *opt = nullptr, const CELL_LOC loc = CELL CENTRE, Mesh *mesh_in = nullptr)

inline ~LaplaceMumps()

inline virtual void setCoefA(const Field2D &val) override
Set coefficients for inversion. Re-builds matrices if necessary.
inline virtual void setCoefC(const Field2D &val) override

inline virtual void setCoefCl(const Field2D &val) override
inline virtual void setCoefC2(const Field2D &val) override
inline virtual void setCoefD(const Field2D &val) override
inline virtual void setCoefEx(const Field2D &val) override
inline virtual void setCoefEz(const Field2D &val) override
inline virtual void setCoefA(const Field3D &val) override
inline virtual void setCoefC(const Field3D &val) override
inline virtual void setCoefC1(const Field3D &val) override
inline virtual void setCoefC2(const Field3D &val) override
inline virtual void setCoefD(const Field3D &val) override
inline virtual void setCoefEx(const Field3D &val) override
inline virtual void setCoefEz(const Field3D &val) override
inline virtual bool uses3DCoefs() const override
    Does this solver use Field3D coefficients (true) or only their DC component (false)
inline virtual void setFlags(int f)

virtual FieldPerp solve(const FieldPerp &b) override
virtual FieldPerp solve(const FieldPerp &b, const FieldPerp &x0) override

Private Functions

void solve(BoutReal *rhs, int y)

void Coefs(int x, int y, int z, BoutReal &A1, BoutReal &A2, BoutReal &A3, BoutReal &A4, BoutReal &A5)
Private Members

Field3D A
Field3D C1
Field3D C2
Field3D D
Field3D Ex
Field3D Ez
bool issetD
bool issetC
bool issetE
Array<BoutReal> rhs
Array<BoutReal> localrhs
int localrhssize
Array<int> localrhs_size_array
Array<int> rhs_positions
FieldPerp sol
int Istart
int Iend
int meshx
int meshz
int size
int localN
int nxguards
MPI_Comm comm
Options *opts
bool fourth_order
int implemented_flags
int implemented_boundary_flags
DMUMPS_STRUC_C mumps_struc
K.2.150 File naulin_laplace.cxx

Iterative solver to handle non-constant-in-z coefficients.

Scheme suggested by Volker Naulin: solve
\[ \text{Delp}(\phi[i+1]) + \frac{1}{DC(C1*D)} \text{Grad}_\perp(\text{DC}(C2)) \text{Grad}_\perp(\phi[i+1]) + \frac{DC(A/D)*\phi[i+1]}{DC(C1*D)*\text{Grad}_\perp(\text{DC}(C2)) \text{Grad}_\perp(\phi[i]) + DC(A/D)*\phi[i]} \]

using standard FFT-based solver, iterating to include other terms by evaluating them on rhs using phi from previous iteration. DC part (i.e. Field2D part) of C1*D, C2 and A/D is kept in the FFT inversion to improve convergence by including as much as possible in the direct solve and so that all Neumann boundary conditions can be used at least when DC(A/D)\text{!}=0.

CHANGELOG

Copyright 2018 B.D.Dudson, M. Loiten, J. Omotani

Contact: Ben Dudson, benjamin.dudson@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

Explanation of the procedure:

A way to invert the equation \( \Omega^D = \nabla \cdot (n \nabla_\perp \phi) \) invented by Naulin, V. In an orthogonal system, we have that:

\begin{align*}
\Omega^D &= \nabla \cdot (n \nabla_\perp \phi) \\
&= n \nabla_\perp^2 \phi + \nabla n \cdot \nabla_\perp \phi \\
&= n \Omega + \nabla n \cdot \nabla_\perp \phi \\
&= n \Omega + \nabla_\perp n \cdot \nabla_\perp \phi \quad (11.1)
\end{align*}
Rearranging gives

\[ \Omega = \frac{\Omega^D}{n} - \nabla_\perp \ln(n) \cdot \nabla_\perp \phi \]
\[ \nabla_\perp^2 \phi = \]
\[ \frac{\Omega^D}{n} - \nabla_\perp \ln(n) \cdot \nabla_\perp \phi \quad (11.1) \]

In fact we allow for the slightly more general form

\[ \nabla_\perp^2 \phi + < \frac{A}{D} > \phi = \]
\[ \text{rhs}/D - \frac{1}{D C_1} \nabla_\perp C^2 \cdot \nabla_\perp \phi - (\frac{A}{D} - < \frac{A}{D} >) * \phi \quad (11.1) \]
The iteration can be under-relaxed to help it converge. Amount of under-relaxation is set by the parameter 'underrelax_factor'. 0<underrelax_factor<=1, with underrelax_factor=1 corresponding to no under-relaxation. The amount of under-relaxation is temporarily increased if the iteration starts diverging, the starting value of underrelax_factor can be set with the initial_underrelax_factor option.

The iteration now works as follows:

1. Get the vorticity from

\[
vort = (\text{vortD}/n) - \text{grad_perp(ln}_n)\cdot\text{grad_perp(\phiCur)}\\
\text{[\text{Delp2(\phiNext)} + 1/\text{DC(C2*D)}\cdot\text{grad_perp(\text{DC(C2)})}\cdot\text{grad_perp(\phiNext)} + \text{DC(A/D)}\cdot\phiNext = \text{b(\phiCur)} = (\text{rhs/D)} - (1/\text{C1/D}\cdot\text{grad_perp(C2)}\cdot\text{grad_perp(\phiCur)} - 1/\text{DC(C2*D)}\cdot\text{grad_{perp(DC(C2)\cdot\text{grad_perp(\phiNext)) - (A/D - DC(A/D)}\cdot\phiCur}]\right)
\]

where \phiCur is \phi of the current iteration [and DC(f) is the constant-in-z component of f]

2. Invert \phiCur to find the vorticity using

\[
\phiNext = \text{invert_laplace_perp(vort)}\\
\text{[set Acoef of laplace_perp solver to DC(A/D)}\\
\text{and C1coef of laplace_perp solver to DC(C1*D)}\\
\text{and C2coef of laplace_perp solver to DC(C2)}\\
\text{then \phiNext = invert_laplace_perp(underrelax_factor}\cdot\text{b(\phiCur)} - (1-\text{underrelax_factor})\cdot\text{b(\phiPrev)})]
\]

where b(\phiPrev) is the previous rhs value, which (up to rounding errors) is the same as the lhs of the direct solver applied to \phiCur.

where \phiNext is the newly obtained \phi

3. Calculate the error at \phi=\phiNext

\[
\text{error3D} = \text{Delp2(\phiNext)} + 1/\text{C1}\cdot\text{grad_perp(C2)}\cdot\text{grad_perp(\phiNext)} + \text{A/D}\cdot\phiNext - \text{\text{rhs/D}}\\
\text{as \text{b(\phiCur)} = Delp2(\phiNext) + 1/\text{DC(C2*D)}\cdot\text{grad_perp(\text{DC(C2)})}\cdot\text{grad_perp(\phiNext)} + \text{\text{DC(A/D)}\cdot\phiNext up to rounding errors}
\]

4. Calculate the infinity norms of the error

\[
\text{EAbsLInf} = \max(\text{error3D})\\
\text{ERelLInf} = \text{EAbsLInf}/\sqrt{\max((\text{rhs/D})^2)}
\]

5. Check whether

\[
\text{EAbsLInf > atol}
\]

• If yes
  – Check whether
ERelLInf > rtol

- If yes
  * Check whether
    EAbsLInf > EAbsLInf(previous step)
    
    · If yes
      underrelax_factor *= 0.9
      Restart iteration
    · If no
      
      - Set
        phiCur = phiNext
        increase curCount and start from step 1
      · If number of iteration is above maxit, throw exception
    - If no
      * Stop: Function returns phiNext
      • if no
        
        - Stop: Function returns phiNext

K.2.151 File naulin_laplace.hxx

class LaplaceNaulin : public Laplacian
#include <naulin_laplace.hxx> Solves the 2D Laplacian equation.

Public Functions

LaplaceNaulin(Options *opt = NULL, const CELL_LOC loc = CELL_CENTRE, Mesh *mesh_in = nullptr)
~LaplaceNaulin()

inline virtual void setCoefA(const Field2D &val) override
  Set coefficients for inversion. Re-builds matrices if necessary.
inline virtual void setCoefA(const Field3D &val) override

inline virtual void setCoefC(const Field2D &val) override
inline virtual void setCoefC(const Field3D &val) override

inline virtual void setCoefC1(const Field3D &val) override
inline virtual void setCoefC1(const Field2D &val) override

inline virtual void setCoefC2(const Field3D &val) override

inline virtual void setCoefC2(const Field2D &val) override

inline virtual void setCoefD(const Field3D &val) override

inline virtual void setCoefD(const Field2D &val) override

inline virtual void setCoefEx(const Field2D &val) override

inline virtual void setCoefEz(const Field2D &val) override

inline virtual bool uses3DCoeffs() const override

    Does this solver use Field3D coefficients (true) or only their DC component (false)

inline virtual FieldPerp solve(const FieldPerp &b) override

inline virtual FieldPerp solve(const FieldPerp &b, const FieldPerp &x0) override

virtual Field3D solve(const Field3D &b, const Field3D &x0) override

    Performs the laplacian inversion y-slice by y-slice

    Parameters
    • b – [in] All the y-slices of b_slice, which is the right hand side of the equation
      A*x_slice = b_slice
    • x0 – [in] All the y-slices of the variable eventually used to set BC

    Returns x All the y-slices of x_slice in the equation A*x_slice = b_slice

inline virtual Field3D solve(const Field3D &b) override

inline virtual void setGlobalFlags(int f) override

inline virtual void setInnerBoundaryFlags(int f) override

inline virtual void setOuterBoundaryFlags(int f) override

inline BoutReal getMeanIterations() const

inline void resetMeanIterations()
Private Functions

LaplaceNaulin(const LaplaceNaulin&)

LaplaceNaulin &operator=(const LaplaceNaulin&)

void copy_x_boundaries(Field3D &x, const Field3D &x0, Mesh *mesh)
  Copy the boundary guard cells from the input ‘initial guess’ x0 into x. These may be used to set non-zero-value boundary conditions

Private Members

Field3D Acoef
Field3D C1coef
Field3D C2coef
Field3D Dcoef

Laplacian *delp2solver
  Laplacian solver used to solve the equation with constant-in-z coefficients.

BoutReal rtol
  Solver tolerances.

BoutReal atol

int maxits
  Maximum number of iterations.

BoutReal initial_underrelax_factor = {1.}
  Initial choice for under-relaxation factor, should be greater than 0 and less than or equal to 1. Value of 1 means no underrelaxation

BoutReal naulinsolver_mean_its
  Mean number of iterations taken by the solver.

BoutReal naulinsolver_mean_underrelax_counts = {0.}
  Mean number of times the underrelaxation factor is reduced.

int ncalls
  Counter for the number of times the solver has been called.
K.2.152 File nc_format.cxx

K.2.153 File nc_format.hxx

netCDF data format interface

Records: In netCDF, the time dimension for each dimension must be the same. Hence when a record is appended to a variable, the size of all variables is increased. To work out which record to write to, a map of variable names to record number is kept.

Author  B.Dudson
Date    April 2009
Copyright 2010 B.D. Dudson, S. Farley, M.V. Umansky, X.Q. Xu
Contact: Ben Dudson, bd512@york.ac.uk
This file is part of BOUT++.
BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.
You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

class NcFormat : public DataFormat

Public Functions

NcFormat(Mesh *mesh_in = nullptr)

NcFormat(const char *name, Mesh *mesh_in = nullptr)

inline NcFormat(const std::string &name, Mesh *mesh_in = nullptr)

~NcFormat()

virtual bool openr(const char *name) override

virtual bool openw(const char *name, bool append = false) override

virtual bool is_valid() override

virtual void close() override
virtual void flush() override

inline const char *filename()

virtual const std::vector<int> getSize(const char *var) override

virtual const std::vector<int> getSize(const std::string &var) override

virtual bool setGlobalOrigin(int x = 0, int y = 0, int z = 0) override

inline virtual bool setLocalOrigin(int x = 0, int y = 0, int z = 0, int offset_x = 0, int offset_y = 0, int offset_z = 0) override

virtual bool setRecord(int t) override

virtual bool addVarInt(const std::string &name, bool repeat) override

virtual bool addVarIntVec(const std::string &name, bool repeat, size_t size) override

virtual bool addVarString(const std::string &name, bool repeat, size_t size) override

virtual bool addVar BoutReal(const std::string &name, bool repeat) override

virtual bool addVarField2D(const std::string &name, bool repeat) override

virtual bool addVarField3D(const std::string &name, bool repeat) override

virtual bool addVarFieldPerp(const std::string &name, bool repeat) override

virtual bool read(int *var, const char *name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read(int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read(char *var, const char *name, int n = 1) override

virtual bool read(char *var, const std::string &name, int n = 1) override

virtual bool read(BoutReal *var, const char *name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read(BoutReal *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read_perp(BoutReal *var, const std::string &name, int lx = 1, int lz = 0) override
virtual bool write(int *var, const char *name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write(int *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write(char *var, const char *name, int n = 1) override

virtual bool write(char *var, const std::string &name, int n = 1) override

virtual bool write(BoutReal *var, const char *name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write(BoutReal *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write_perp(BoutReal *var, const std::string &name, int lx = 0, int lz = 0) override

virtual bool read_rec(int *var, const char *name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read_rec(int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read_rec(char *var, const char *name, int n = 1) override

virtual bool read_rec(char *var, const std::string &name, int n = 1) override

virtual bool read_rec(BoutReal *var, const char *name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read_rec(BoutReal *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read_rec_perp(BoutReal *var, const std::string &name, int lx = 1, int lz = 0) override

virtual bool write_rec(int *var, const char *name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write_rec(int *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write_rec(char *var, const char *name, int n = 1) override

virtual bool write_rec(char *var, const std::string &name, int n = 1) override

virtual bool write_rec(BoutReal *var, const char *name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write_rec(BoutReal *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write_rec_perp(BoutReal *var, const std::string &name, int lx = 0, int lz = 0) override
inline virtual void setLowPrecision() override

virtual void setAttribute(const std::string &varname, const std::string &attrname, const std::string &text) override
Sets a string attribute

Inputs
Parameters
  • varname – [in] Variable name. The variable must already exist. If varname is the empty string "" then the attribute will be added to the file instead of to a variable.
  • attrname – [in] Attribute name
  • text – [in] A string attribute to attach to the variable

virtual void setAttribute(const std::string &varname, const std::string &attrname, int value) override
Sets an integer attribute

Inputs
Parameters
  • varname – [in] Variable name. The variable must already exist. If varname is the empty string "" then the attribute will be added to the file instead of to a variable.
  • attrname – [in] Attribute name
  • value – [in] An int attribute to attach to the variable

virtual void setAttribute(const std::string &varname, const std::string &attrname, BoutReal value) override
Sets a BoutReal attribute

Inputs
Parameters
  • varname – [in] Variable name. The variable must already exist. If varname is the empty string "" then the attribute will be added to the file instead of to a variable.
  • attrname – [in] Attribute name
  • value – [in] A BoutReal attribute to attach to the variable

virtual bool getAttribute(const std::string &varname, const std::string &attrname, std::string &text) override
Gets a string attribute

Inputs

Returns

  text A string attribute of the variable

Parameters
  • varname – [in] Variable name. The variable must already exist. If varname is the empty string "" then get the attribute from the top-level of the file instead of from a variable.
  • attrname – [in] Attribute name
virtual bool getAttribute(const std::string &varname, const std::string &attrname, int &value) override

Gets an integer attribute

**Inputs**

value An int attribute of the variable

**Parameters**

- **varname** – [in] Variable name. The variable must already exist. If varname is the empty string "" then get the attribute from the top-level of the file instead of from a variable.
- **attrname** – [in] Attribute name

virtual bool getAttribute(const std::string &varname, const std::string &attrname, BoutReal &value)

Gets a BoutReal attribute

**Inputs**

value A BoutReal attribute of the variable

**Parameters**

- **varname** – [in] Variable name. The variable must already exist. If varname is the empty string "" then get the attribute from the top-level of the file instead of from a variable.
- **attrname** – [in] Attribute name

**Private Functions**

void checkName(const char *name)

Check if a name contains invalid characters.

**Private Members**

char *fname
Current file name.

NcFile *dataFile
Pointer to netCDF file.

NcDim *xDim
Dimensions.

NcDim *yDim
NcDim *zDim
NcDim *tDim

const NcDim **dimList
List of dimensions (x,y,z)
const NeDim **recDimList
   List of dimensions (t,x,y,z)

bool lowPrecision
   When writing, down-convert to floats.

int x0
int y0
int z0
int t0
   Data origins.

std::map<std::string, int> rec_nr

int default_rec

K.2.154 File ncxx4.cxx

namespace netCDF

K.2.155 File ncxx4.hxx

netCDF-4 data format interface

Records: In netCDF, the time dimension for each dimension must be the same. Hence when a record is appended to a variable, the size of all variables is increased. To work out which record to write to, a map of variable names to record number is kept.

Author  B.Dudson

Date    September 2012

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

class Ncxx4 : public DataFormat
Public Functions

Ncxx4(Mesh *mesh_in = nullptr)

Ncxx4(const char *name, Mesh *mesh_in = nullptr)

inline Ncxx4(const std::string &name, Mesh *mesh_in = nullptr)

~Ncxx4()

virtual bool openr(const char *name) override

virtual bool openw(const char *name, bool append = false) override

virtual bool isValid() override

virtual void close() override

virtual void flush() override

inline const char *filename()

virtual const std::vector<int> getSize(const char *var) override

virtual const std::vector<int> getSize(const std::string &var) override

virtual bool setGlobalOrigin(int x = 0, int y = 0, int z = 0) override

inline virtual bool setLocalOrigin(int x = 0, int y = 0, int z = 0, int offset_x = 0, int offset_y = 0, int offset_z = 0) override

virtual bool setRecord(int t) override

virtual bool addVarInt(const std::string &name, bool repeat) override

virtual bool addVarIntVec(const std::string &name, bool repeat, size_t size) override

virtual bool addVarString(const std::string &name, bool repeat, size_t size) override

virtual bool addVarBoutReal(const std::string &name, bool repeat) override

virtual bool addVarField2D(const std::string &name, bool repeat) override
virtual bool addVarField3D(const std::string &name, bool repeat) override

virtual bool addVarFieldPerp(const std::string &name, bool repeat) override

virtual bool read(int *var, const char *name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read(int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read(char *var, const char *name, int n = 1) override

virtual bool read(char *var, const std::string &name, int n = 1) override

virtual bool read(BoutReal *var, const char *name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read(BoutReal *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read_perp(BoutReal *var, const std::string &name, int lx = 1, int lz = 0) override

virtual bool write(int *var, const char *name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write(int *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write(char *var, const char *name, int n = 1) override

virtual bool write(char *var, const std::string &name, int n = 1) override

virtual bool write(BoutReal *var, const char *name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write(BoutReal *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write_perp(BoutReal *var, const std::string &name, int lx = 0, int lz = 0) override

virtual bool read_rec(int *var, const char *name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read_rec(int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read_rec(char *var, const char *name, int n = 1) override

virtual bool read_rec(char *var, const std::string &name, int n = 1) override

virtual bool read_rec(BoutReal *var, const char *name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read_rec(BoutReal *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) override
virtual bool read_rec(BoutReal *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read_rec_perp(BoutReal *var, const std::string &name, int lx = 1, int lz = 0) override

virtual bool write_rec(int *var, const char *name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write_rec(const char *var, const std::string &name, int n = 1) override

virtual bool write_rec(char *var, const std::string &name, int n = 1) override

virtual bool write_rec(BoutReal *var, const char *name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write_rec(BoutReal *var, const std::string &name, int lx = 0, int lz = 0) override

virtual bool write_rec_perp(BoutReal *var, const std::string &name, int lx = 0, int lz = 0) override

inline virtual void setLowPrecision() override

virtual void setAttribute(const std::string &varname, const std::string &attrname, const std::string &text) override

Sets a string attribute

**Inputs**

**Parameters**

- **varname** – [in] Variable name. The variable must already exist. If varname is the empty string "" then the attribute will be added to the file instead of to a variable.
- **attrname** – [in] Attribute name
- **text** – [in] A string attribute to attach to the variable

virtual void setAttribute(const std::string &varname, const std::string &attrname, int value) override

Sets an integer attribute

**Inputs**

**Parameters**

- **varname** – [in] Variable name. The variable must already exist. If varname is the empty string "" then the attribute will be added to the file instead of to a variable.
- **attrname** – [in] Attribute name
- **value** – [in] An int attribute to attach to the variable

virtual void setAttribute(const std::string &varname, const std::string &attrname, BoutReal value) override

Sets a BoutReal attribute

**Inputs**

**Parameters**
• **varname** – [in] Variable name. The variable must already exist. If varname is the empty string """, then the attribute will be added to the file instead of to a variable.

• **attrname** – [in] Attribute name

• **value** – [in] A BoutReal attribute to attach to the variable

virtual bool **getAttribute** (const std::string &varname, const std::string &attrname, std::string &text) override

Gets a string attribute

**Inputs**

**Returns**

text A string attribute of the variable

**Parameters**

• **varname** – [in] Variable name. The variable must already exist. If varname is the empty string """, then get the attribute from the top-level of the file instead of from a variable.

• **attrname** – [in] Attribute name

virtual bool **getAttribute** (const std::string &varname, const std::string &attrname, int &value) override

Gets an integer attribute

**Inputs**

**Returns**

value An int attribute of the variable

**Parameters**

• **varname** – [in] Variable name. The variable must already exist. If varname is the empty string """, then get the attribute from the top-level of the file instead of from a variable.

• **attrname** – [in] Attribute name

virtual bool **getAttribute** (const std::string &varname, const std::string &attrname, BoutReal &value) override

Gets a BoutReal attribute

**Inputs**

**Returns**

value A BoutReal attribute of the variable

**Parameters**

• **varname** – [in] Variable name. The variable must already exist. If varname is the empty string """, then get the attribute from the top-level of the file instead of from a variable.

• **attrname** – [in] Attribute name
Private Functions

\texttt{std::vector<netCDF::NcDim> getDimVec(int nd)}

\texttt{std::vector<netCDF::NcDim> getRecDimVec(int nd)}

Private Members

\texttt{char * fname}
Current file name.

\texttt{netCDF::NcFile * dataFile}
Pointer to netCDF file.

\texttt{netCDF::NcDim xDim}
Dimensions.

\texttt{netCDF::NcDim yDim}

\texttt{netCDF::NcDim zDim}

\texttt{netCDF::NcDim tDim}

\texttt{const netCDF::NcDim ** dimList}
List of dimensions (x,y,z)

\texttt{const netCDF::NcDim ** recDimList}
List of dimensions (t,x,y,z)

\texttt{bool appending}

\texttt{bool lowPrecision}
When writing, down-convert to floats.

\texttt{int x0}
\texttt{int y0}
\texttt{int z0}
\texttt{int t0}
Data origins.

\texttt{std::map<std::string, int> rec_nr}

\texttt{int default_rec}
K.2.156 File openmpwrap.hxx

Defines

INDIRECT0(a)

Openmp utility wrappers.

Copyright 2017

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General
Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option)
any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even
the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see
http://www.gnu.org/licenses/.

INDIRECT1(a)

INDIRECT2(a)

BOUT_OMP(...)

K.2.157 File optionparser.hxx

class OptionParser

#include <optionparser.hxx> Base class for input file types.

Subclassed by OptionINI

Public Functions

OptionParser() = default

virtual ~OptionParser() = default

virtual void read(Options *options, const std::string &filename) = 0

Read filename into options.

virtual void write(Options *options, const std::string &filename) = 0

Write options to filename.
**K.2.158 File options.cxx**

Functions

template<>  
`std::string as< std::string > (const std::string &similar_to)` const  
Specialised as routines.

**K.2.159 File options.hxx**

Defines

```cpp
__OPTIONS_H__

OPTION(options, var, def)  
Define for reading options which passes the variable name.

OPTION2(options, var1, var2, def)

OPTION3(options, var1, var2, var3, def)

OPTION4(options, var1, var2, var3, var4, def)

OPTION5(options, var1, var2, var3, var4, var5, def)

OPTION6(options, var1, var2, var3, var4, var5, var6, def)

VAROPTION(options, var, def)

BOUT_OVERRIDE_DEFAULT_OPTION(name, value)  
Define for over-riding library defaults for options, should be called in global namespace so that the new default is set before main() is called.
```

Functions

template<>  
`std::string as< std::string > (const std::string &similar_to)` const  
Specialised as routines.

class `Options`  
#include <options.hxx> Class to represent hierarchy of options.

**Getting and setting values**

Each `Options` object represents a collection of key-value pairs which can be used as a map.

```c++
Options options;

// Set values
```

(continues on next page)
options["key"] = 1.0;

// Get values. Throws BoutException if not found
int val = options["key"]; // Sets val to 1

// Return as specified type. Throws BoutException if not found
BoutReal var = options["key"].as<BoutReal>();

// A default value can be used if key is not found
BoutReal value = options["pi"].withDefault(3.14);

// Assign value with source label. Throws if already has a value from same source
options["newkey"].assign(1.0, "some source");

// Force assign a new value
options["newkey"].force(2.0, "some source");

A legacy interface is also supported:

options.set("key", 1.0, "code"); // Sets a key from source "code"

int val;
options.get("key", val, 0); // Sets val to 1, default to 0 if not found

If a variable has not been set then the default value is used

int other;
options.get("otherkey", other, 2.0); // Sets other to 2 because "otherkey" not found

Conversion is performed silently:

options.set("value", "2.34", "here"); // Set a string

BoutReal value;
options.get("value", value, 0.0); // Sets value to 2.34

If a conversion cannot be done, then an exception is thrown

Sections
Each Options object can also contain any number of sections, which are themselves Options objects.

Options &section = options["section"];

which can be nested:

options["section"]["subsection"]["value"] = 3;

This always succeeds; if the section does not exist then it is created.

The legacy interface uses pointers:

Options *section = options.getSection("section");

e.g. options->getSection("section")->getSection("subsection")->set("value", 3);
Options also know about their parents:

Options &parent = section.parent();

or

Options *parent = section->getParent();

Root options object

For convenience, to avoid having to pass Options objects around everywhere, there is a global singleton Options object which can be accessed with a static function

Options &root = Options::root();

or

Options *root = Options::getRoot();

This is used to represent all the options passed to BOUT++ either in a file or on the command line.

Public Types

using ValueType = bout::utils::variant<bool, int, BoutReal, std::string, Field2D, Field3D, FieldPerp, Array<BoutReal>, Matrix<BoutReal>, Tensor<BoutReal>>

The type used to store values.

using ValuesMap = std::map<std::string, OptionValue>

Read-only access to internal options and sections to allow iteration over the tree

Public Functions

Options() = default

Constructor. This is called to create the root object.

inline Options(Options *parent_instance, std::string full_name)

Constructor used to create non-root objects

Parameters

• parent – [in] Parent object

• sectionName – [in] Name of the section, including path from the root

Options(const Options &other)

Copy constructor.

~Options() = default

inline bool hasAttribute(const std::string &key) const

Return true if this value has attribute key.

Options &operator[](const std::string &name)

Get a sub-section or value

Example:

Options parent; auto child = parent[“child”];
inline `Options &operator[]`(const char *name)

```cpp
const Options &operator[](const std::string &name) const
```
Get a sub-section or value If this object is not a section, or if name is not a child, then a `BoutException` will be thrown

```cpp
inline const Options &operator[](const char *name)
```

```cpp
template<typename T>
inline T operator=(T inputvalue)
```
Assignment from any type T Note: Using this makes this object a value.
Tries to stream the input to a std::stringstream and then saves the resulting string.

```cpp
Options &operator=(const Options &other)
```
Copy assignment
This replaces the value, attributes and all children
Note that if only the value is desired, then that can be copied using the value member directly e.g. option2.value = option1.value;

```cpp
template<typename T>
inline void assign(T val, const std::string source = "")
```
Assign a value to the option. This will throw an exception if already has a value
Example:

```cpp
Options option; option[“test”].assign(42, “some source”);
```
Note: Specialised versions for types stored in ValueType

```cpp
template<typename T>
inline void force(T val, const std::string source = "")
```
Force to a value Overwrites any existing setting

```cpp
bool isSet() const
```
Test if a key is set by the user. Values set via default values are ignored.

```cpp
template<typename T>
inline operator T() const
```
Cast operator, which allows this class to be assigned to type T
Example:

```cpp
Options option; option[“test”] = 2.0; int value = option[“test”];
```

```cpp
template<typename T>
inline T as(const T &similar_to = {})
```
Get the value as a specified type If there is no value then an exception is thrown Note there are specialised versions of this template for some types.
Example:

```cpp
Options option; option[“test”] = 2.0; int value = option[“test”].as<int>();
```
An optional argument is an object which the result should be similar to. The main use for this is in `Field2D` and `Field3D` specialisations, where the `Mesh` and cell location are taken from this input.
inline \texttt{T \_withDefault}(T \texttt{def})

Get the value of this option. If not found, set to the default value

inline \texttt{std::string \_withDefault(const char \_*def)\n}

Overloaded version for const char* Note: Different from template since return type is different to input

inline \texttt{Options \&\_withDefault(const Options \&def)\n}

Overloaded version to copy from another option.

template<typename T>
inline \texttt{T \_withDefault}(T \texttt{def})\texttt{\,}\texttt{const}\n
Get the value of this option. If not found, return the default value but do not set

template<typename T>
inline \texttt{T \_overrideDefault}(T \texttt{def})\texttt{\,}\texttt{const}\n
Allow the user to override defaults set later, also used by the BOUT\_OVERRIDE\_DEFAULT\_OPTION.

inline \texttt{std::string \_overrideDefault(const char \_*def)\n}

Overloaded version for const char* Note: Different from template since return type is different to input

inline \texttt{Options \&\_parent()\n}

Get the parent Options object.

template<typename T>
inline bool \_operator\texttt{==}(const T \&other)\texttt{\,}\texttt{const}\n
Equality operator Converts to the same type and compares This conversion may fail, throwing std::bad\_cast

bool \_operator\texttt{==}(const char \*other)\texttt{\,}\texttt{const}\n
Overloaded equality operator for literal strings.

template<typename T>
inline bool \_operator\texttt{<}(const T \&other)\texttt{\,}\texttt{const}\n
Comparison operator.

bool \_operator\texttt{<}(const char \*other)\texttt{\,}\texttt{const}\n
Overloaded comparison operator for literal strings.

template<typename T>
inline void \_set(const std::string &key, T val, const std::string &source = \texttt{""}, bool force = \texttt{false})\texttt{\,}\texttt{\,}\texttt{inline}\n
Get options, passing in a reference to a variable which will be set to the requested value.

template<typename T, typename U>
inline void \_get(const std::string &key, T \&val, U \texttt{def}, bool log = \texttt{false})\texttt{\,}\texttt{const}\n
Get options, passing in a reference to a variable which will be set to the requested value.

template<typename T, typename U>
inline void \_get(const std::string &key, T \&val, U \texttt{def}, bool log = \texttt{false})\texttt{\,}\texttt{const}\n

inline \texttt{Options \*\_getSection(const std::string \&name)\n}

Creates new section if doesn't exist.

inline const \texttt{Options \*\_getSection(const std::string \&name)\texttt{\,}\texttt{const}}
inline Options *getParent() const

inline std::string str() const
    Print string representation of this object and all sections in a tree structure
void printUnused() const
    Print the options which haven’t been used.

ValuesMap values() const

std::map<std::string, const Options*> subsections() const

inline const std::map<std::string, Options> &getChildren() const

inline bool isValue() const

bool isSection(const std::string &name = "") const

inline bool valueUsed() const
    If the option value has been used anywhere.
inline Options &doc(const std::string &docstring)
    Set a documentation string as an attribute “doc” Returns a reference to this, to allow chaining

template<typename T, typename Source>
inline void assign(T val, const std::string source)

template
inline void assign(bool, const std::string source)

template
inline void assign(int, const std::string source)

template
inline void assign(BoutReal, const std::string source)

template
inline void assign(std::string, const std::string source)

template
inline void assign(const char *, const std::string source)

void assign(Field2D, const std::string source)

void assign(Field3D, const std::string source)

void assign(FieldPerp, const std::string source)
void assign(Array<BoutReal> val, const std::string source)

template<> void assign(Matrix<BoutReal> val, const std::string source)

template<> void assign(Tensor<BoutReal> val, const std::string source)

template<> int as(const int &similar_to) const

template<> BoutReal as(const BoutReal &similar_to) const

template<> bool as(const bool &similar_to) const

template<> Field2D as(const Field2D &similar_to) const

template<> Field3D as(const Field3D &similar_to) const

template<> FieldPerp as(const FieldPerp &similar_to) const

Public Members

ValueType value
The value stored.

std::map<std::string, AttributeType> attributes
A collection of attributes belonging to the value Special attributes:

- time_dimension [string] If this is set then changes to the value do not need to be forced. The string will be used when writing the output as the name of the time dimension (unlimited first dimension in NetCDF files).
- source [string] Describes where the value came from e.g. a file name, or “default”.
- type [string] The type the Option is converted to when used.
- doc [string] Documentation, describing what the variable does
Public Static Functions

static **Options &root()**
Get a reference to the only root instance.

static void **cleanup()**
Free all memory.

static inline **Options *getRoot()**
Get a pointer to the only root instance (singleton)

static void **cleanCache()**
Cache of parsed options

Private Functions

template<typename T>
inline void **__set(T val, std::string source, bool force)**

template<typename T>
inline bool **similar(T a, T b) const**
Tests if two values are similar.

template<>
inline bool **similar(BoutReal a, BoutReal b) const**
Specialised similar comparison methods.

Private Members

**Options *parent_instance = {nullptr}**

**std::string full_name**

bool **is_section = false**
An Option object can be a section and/or a value, or neither (empty)

Is this **Options** object a section?

**std::map<std::string, Options> children**
If a section then has children.

bool **is_value = false**
Is this **Options** object a value?

mutable bool **value_used = false**
Record whether this value is used.
Private Static Attributes

static const std::string DEFAULT_SOURCE = {"default")
The source label given to default values.

static Options *root_instance = {nullptr}
Only instance of the root section.

class AttributeType : public mpark::variant<bool, int, BoutReal, std::string>
#include <options.hxx> The type used to store attributes Extends the variant class so that cast operator can be implemented and assignment operator overloaded

Note: Due to default initialisation rules, if an attribute is used without being set, it will be false, 0, 0.0 and throw std::bad_cast if cast to std::string

Public Types

using Base = bout::utils::variant<bool, int, BoutReal, std::string>

Public Functions

AttributeType() = default
Constructor.

AttributeType(const AttributeType &other) = default
Copy constructor.

inline AttributeType(AttributeType &&other)
Move constructor.

~AttributeType() = default
Destructor.

inline AttributeType &operator=(const char *str)
Assignment from const char*.

template<typename T>
inline operator T() const
Cast operator, which allows this class to be assigned to type T This will throw std::bad_cast if it can't be done

template<typename T>
inline T as() const
Get the value as a specified type This will throw std::bad_cast if it can't be done

struct OptionValue
#include <options.hxx> Class used to store values, together with information about their origin and usage
Public Functions

inline OptionValue(std::string value, std::string source, bool used)
  This constructor needed for map::emplace Can be removed in C++17 with map::insert and brace initialisation

Public Members

std::string value
std::string source
mutable bool used = false

K.2.160 File options_ini.cxx

K.2.161 File options_ini.hxx

class OptionINI : public OptionParser
  #include <options_ini.hxx> Class for reading INI style configuration files.

Public Functions

virtual void read(Options *options, const std::string &filename) override
  Read options from file.

virtual void write(Options *options, const std::string &filename) override
  Write options to file.

Private Functions

void parse(const std::string&, std::string&, std::string&)

std::string getNextLine(std::ifstream &fin)

void writeSection(const Options *options, std::ofstream &fout)

K.2.162 File options_netcdf.cxx

namespace bout
  SNB model

  namespace experimental
K.2.163 File options_netcdf.hxx

Defines

__OPTIONS_NETCDF_H__

namespace bout
    SNB model

namespace experimental

class OptionsNetCDF

Public Types

denum FileMode
    Values:

    enumerator replace
        Overwrite file when writing.

    enumerator append
        Append to file when writing.

Public Functions

inline OptionsNetCDF(std::string filename, FileMode mode = FileMode::replace)

Options read()
    Read options from file.

void write(const Options &options)
    Write options to file.

Private Members

std::string filename

FileMode file_mode

std::map<int, size_t> time_index
    NetCDF doesn't seem to keep track of the current index for each variable. This map keeps track of the current index being written for each time dimension
class OptionsReader
#include <optionsreader.hxx> Class to handle reading options from file

Example
Options opt; OptionsReader::getInstance()->read(&opt, “somefile.inp”);

opt now contains a tree of sections and options from the input file “somefile.inp”

Public Functions

void read(Options *options, const char *file, ...)
Read the given file, parse options into the options tree.

Parameters
• options – [inout] The options section to insert values and subsections into
• file – [in] The name of the file. printf style arguments can be used to create the
  file name.

void write(Options *options, const char *file, ...)
Write options to file

Parameters
• options – [in] The options tree to be written
• file – [in] The name of the file to (over)write

void parseCommandLine(Options *options, int argc, char **argv)
Parse options from the command line

Example
int main(int argc, char** argv) { Options opt; OptionsReader::getInstance()->read(&opt, argc, argv); …
return 0; }

Parameters
• options – [inout] The options section to insert values and subsections into
• argc – [in] The number of command-line arguments
• argv – [in] The command line arguments

Public Static Functions

static OptionsReader *getInstance()
Return a pointer to the instance singleton.

static inline void cleanup()
Delete the instance.
Private Static Attributes

static OptionsReader *instance = nullptr
    The instance of this singleton.

K.2.166 File output.cxx

Defines

bout_vsnprintf_(buf, len, fmt, va)

Variables

DummyOutput output_debug
    To allow statements like “output.write(…)” or “output << …” Output for debugging

ConditionalOutput output_warn(Output::getInstance())
    warnings

ConditionalOutput output_info(Output::getInstance())
    information

ConditionalOutput output_progress(Output::getInstance())
    progress

ConditionalOutput output_error(Output::getInstance())
    errors

ConditionalOutput output_verbose (Output::getInstance(), false)
    less interesting messages

ConditionalOutput output(Output::getInstance())
    Generic output, given the same level as output_progress.

K.2.167 File output.hxx

Defines

__OUTPUT_H__
Typedefs

using stream_manipulator = std::ostream (*)(std::ostream&)

Functions

template<typename T>
DummyOutput &operator<<(DummyOutput &out, T const &t)
    Catch stream outputs to DummyOutput objects. This is so that statements like output_debug << “debug message”; compile but have no effect if DEBUG_ENABLED is false

template<typename T>
DummyOutput &operator<<(DummyOutput &out, const T *t)

inline DummyOutput &operator<<(DummyOutput &out, stream_manipulator pf)

inline ConditionalOutput &operator<<(ConditionalOutput &out, stream_manipulator pf)

template<typename T>
ConditionalOutput &operator<<(ConditionalOutput &out, T const &t)

template<typename T>
ConditionalOutput &operator<<(ConditionalOutput &out, const T *t)

Variables

DummyOutput output_debug
    To allow statements like “output.write(…)” or “output << …” Output for debugging

ConditionalOutput output_warn
    warnings

ConditionalOutput output_progress
    progress

ConditionalOutput output_info
    information

ConditionalOutput output_error
    errors

ConditionalOutput output_verbose
    less interesting messages

ConditionalOutput output
    Generic output, given the same level as output_progress.
class **Output** : private *multioutbuf_init*<char, *std*:char_traits<char>>, public *std*:basic_ostream<char, *std*:char_traits<char>>

```cpp
#include <output.hxx>
```

Class for text output to stdout and/or log file.

This class can be used to output either in C printf format:

```cpp
output.write("A string %s and number %d\n", str, i);
```

or as a C++ stream buffer:

```cpp
output << "A string " << str << " and number " << i << endl;
```

If a file has been opened (i.e. the processor's log file) then the string will be written to the file. In addition, output to stdout can be enabled and disabled.

Subclassed by **ConditionalOutput**, **DummyOutput**

**Public Functions**

inline **Output** ()

inline **Output** (const char *fname)

Specify a log file to open.

inline **~Output** () override

virtual void **enable** ()

Enables writing to stdout (default)

virtual void **disable** ()

Disables stdout.

int **open** (const char *fname, ...)

Open an output log file.

void **close** ()

Close the log file.

virtual void **write** (const char *string, ...)

Write a string using C printf format.

virtual void **print** (const char *string, ...)

Same as write, but only to screen.

virtual void **vwrite** (const char *string, va_list args)

Write a string using C vprintf format.

virtual void **vprint** (const char *string, va_list args)

Same as vwrite, but only to screen.

inline void **add** (*std*:basic_ostream<char, _Tr> &str)

Add an output stream. All output will be sent to all streams.

inline void **remove** (*std*:basic_ostream<char, _Tr> &str)

Remove an output stream.
Public Static Functions

static Output *getInstance()
    Return pointer to instance.

Protected Functions

inline virtual Output *getBase()

inline virtual bool isEnabled()

Private Types

using _Tr = std::char_traits<char>
using multioutbuf_init = ::multioutbuf_init<char, _Tr>

Private Members

std::ofstream file
    Log file stream.

int buffer_len
    the current length

char *buffer
    Buffer used for C style output.

bool enabled
    Whether output to stdout is enabled.

Private Static Attributes

static const int BUFFER_LEN = 1024
    default length

Friends

friend class ConditionalOutput

class DummyOutput : public Output
    #include <output.hxx> Class which behaves like Output, but has no effect. This is to allow debug outputs to be disabled at compile time
**Public Functions**

inline virtual void **write**(const char *str, ...) override  
Write a string using C printf format.

inline virtual void **print**(const char *str, ...) override  
Same as write, but only to screen.

inline virtual void **enable**() override  
Enables writing to stdout (default)

inline virtual void **disable**() override  
Disables stdout.

inline void **enable**(bool enable)

inline virtual bool **isEnabled**() override  

class **ConditionalOutput** : public **Output**  
#include <output.hxx> Layer on top of **Output** which passes through calls to write, print etc if it is enabled, but discards messages otherwise. This is used to provide different levels of output (info, prog, warn, error) which can be enabled and disabled at run time.

**Public Functions**

inline **ConditionalOutput**( **Output** *base, bool enabled = true)

Parameters

- **base** – [in] The **Output** object which will be written to if enabled
- **enabled** – [in] Should this be enabled by default?

inline **ConditionalOutput**( **ConditionalOutput** *base)

Constructor taking **ConditionalOutput**. This allows several layers of conditions

Parameters **base** – [in] A **ConditionalOutput** which will be written to if enabled

virtual void **write**(const char *str, ...) override  
If enabled, writes a string using C printf formatting by calling base->vwrite This string is then sent to log file and stdout (on processor 0)

inline virtual void **vwrite**(const char *str, va_list va) override  
Write a string using C vprintf format.

virtual void **print**(const char *str, ...) override  
If enabled, print a string to stdout using C printf formatting note: unlike write, this is not also sent to log files

inline virtual void **vprint**(const char *str, va_list va) override  
Same as vwrite, but only to screen.

inline virtual **Output** * **getBase**() override  
Get the lowest-level **Output** object which is the base of this **ConditionalOutput**.

inline void **enable**(bool enable_)  
Set whether this **ConditionalOutput** is enabled If set to false (disabled), then all print and write calls do nothing
inline virtual void enable() override
   Turn on outputs through calls to print and write.

inline virtual void disable() override
   Turn off outputs through calls to print and write. This includes log files and stdout.

inline virtual bool isEnabled() override
   Check if output is enabled.

**Private Members**

*Output *base
   The lower-level Output to send output to.

bool enabled
   Does this instance output anything?

**K.2.168 File parallel_boundary_op.cxx**

**K.2.169 File parallel_boundary_op.hxx**

class BoundaryOpPar : public BoundaryOpBase
   Subclassed by BoundaryOpPar_dirichlet, BoundaryOpPar_dirichlet_interp, BoundaryOpPar_dirichlet_O3, BoundaryOpPar_neumann

**Public Functions**

BoundaryOpPar() = default

inline BoundaryOpPar(BoundaryRegionPar *region, std::shared_ptr<FieldGenerator> value)

inline BoundaryOpPar(BoundaryRegionPar *region, Field3D *value)

inline BoundaryOpPar(BoundaryRegionPar *region, BoutReal value)

~BoundaryOpPar() override = default

inline virtual BoundaryOpPar *clone(BoundaryRegionPar *region, const std::list<std::string> &args)

inline virtual BoundaryOpPar *clone(BoundaryRegionPar *region, Field3D *f)

inline virtual BoundaryOpPar *clone(BoundaryRegionPar *region, const std::list<std::string> &args, const std::map<std::string, std::string> &keywords)

inline virtual void apply(Field2D &f) override
   Apply a boundary condition on field f.
inline virtual void apply(Field2D &f, BoutReal t) override

Public Members

BoundaryRegionPar *bndry = {nullptr}

Protected Types

enum ValueType
    Where to take boundary values from - the generator, field or BoutReal.
    Values:

    enumerator GEN
    enumerator FIELD
    enumerator REAL

Protected Functions

BoutReal getValue(int x, int y, int z, BoutReal t)

BoutReal getValue(const BoundaryRegionPar &bndry, BoutReal t)

Protected Attributes

std::shared_ptr<FieldGenerator> gen_values
    Possible ways to get boundary values.

Field3D *field_values

BoutReal real_value = {0.}

const ValueType value_type = {ValueType::REAL}

class BoundaryOpPar_dirichlet : public BoundaryOpPar

Public Functions

inline BoundaryOpPar_dirichlet()

inline BoundaryOpPar_dirichlet(BoundaryRegionPar *region)

inline BoundaryOpPar_dirichlet(BoundaryRegionPar *region, std::shared_ptr<FieldGenerator> value)
inline `BoundaryOpPar_dirichlet(BoundaryRegionPar *region, Field3D *value)`

inline `BoundaryOpPar_dirichlet(BoundaryRegionPar *region, BoutReal value)`

virtual `BoundaryOpPar *clone(BoundaryRegionPar *region, const std::list<std::string> &args)` override

virtual `BoundaryOpPar *clone(BoundaryRegionPar *region, Field3D *f)` override

inline virtual void `apply(Field3D &f)` override

virtual void `apply(Field3D &f, BoutReal t)` override

class `BoundaryOpPar_dirichlet_03` : public `BoundaryOpPar`

**Public Functions**

inline `BoundaryOpPar_dirichlet_03()`

inline `BoundaryOpPar_dirichlet_03(BoundaryRegionPar *region)`

inline `BoundaryOpPar_dirichlet_03(BoundaryRegionPar *region, std::shared_ptr<FieldGenerator> value)`

inline `BoundaryOpPar_dirichlet_03(BoundaryRegionPar *region, Field3D *value)`

inline `BoundaryOpPar_dirichlet_03(BoundaryRegionPar *region, BoutReal value)`

virtual `BoundaryOpPar *clone(BoundaryRegionPar *region, const std::list<std::string> &args)` override

virtual `BoundaryOpPar *clone(BoundaryRegionPar *region, Field3D *f)` override

inline virtual void `apply(Field3D &f)` override

virtual void `apply(Field3D &f, BoutReal t)` override

class `BoundaryOpPar_dirichlet_interp` : public `BoundaryOpPar`
**Public Functions**

inline `BoundaryOpPar_dirichlet_interp()`

inline `BoundaryOpPar_dirichlet_interp(BoundaryRegionPar *region)`

inline `BoundaryOpPar_dirichlet_interp(BoundaryRegionPar *region, std::shared_ptr<FieldGenerator> value)`

inline `BoundaryOpPar_dirichlet_interp(BoundaryRegionPar *region, Field3D *value)`

inline `BoundaryOpPar_dirichlet_interp(BoundaryRegionPar *region, BoutReal value)`

virtual `BoundaryOpPar *clone(BoundaryRegionPar *region, const std::list<std::string> &args)` override

virtual `BoundaryOpPar *clone(BoundaryRegionPar *region, Field3D *f)` override

inline virtual void `apply(Field3D &f)` override

virtual void `apply(Field3D &f, BoutReal t)` override

class `BoundaryOpPar_neumann` : public `BoundaryOpPar`

**Public Functions**

inline `BoundaryOpPar_neumann()`

inline `BoundaryOpPar_neumann(BoundaryRegionPar *region)`

inline `BoundaryOpPar_neumann(BoundaryRegionPar *region, std::shared_ptr<FieldGenerator> value)`

inline `BoundaryOpPar_neumann(BoundaryRegionPar *region, Field3D *value)`

inline `BoundaryOpPar_neumann(BoundaryRegionPar *region, BoutReal value)`

virtual `BoundaryOpPar *clone(BoundaryRegionPar *region, const std::list<std::string> &args)` override

virtual `BoundaryOpPar *clone(BoundaryRegionPar *region, Field3D *f)` override

inline virtual void `apply(Field3D &f)` override

virtual void `apply(Field3D &f, BoutReal t)` override
class **BoundaryRegionPar** : public **BoundaryRegionBase**

`#include <parallel_boundary_region.hxx>` Boundary region for parallel direction. This contains a vector of points that are inside the boundary.

**Public Functions**

```
inline BoundaryRegionPar(const std::string &name, int dir, Mesh *passmesh)
```

```
inline BoundaryRegionPar(const std::string &name, BndryLoc loc, int dir, Mesh *passmesh)
```

```
void add_point(int jx, int jy, int jz, BoutReal x, BoutReal y, BoutReal z, BoutReal length, BoutReal angle)
    Add a point to the boundary.
```

```
virtual void first() override
    Move the region iterator to the start.
```

```
virtual void next() override
    Get the next element in the loop over every element from inside out (in X or Y first)
```

```
virtual bool isDone() override
    Returns true if outside domain. Can use this with nested nextX, nextY.
```

**Public Members**

```
int x
    Index of the point in the boundary.
```

```
int y
```

```
int z
```

```
BoutReal s_x
```

```
BoutReal s_y
```

```
BoutReal s_z
```

```
BoutReal length
```

```
BoutReal angle
```

const int dir
Private Types

using IndicesVec = std::vector<Indices>
using IndicesIter = IndicesVec::iterator

Private Members

IndicesVec bndry_points
   Vector of points in the boundary.

IndicesIter bndry_position
   Current position in the boundary points.

struct IndexPoint

Public Members

int jx
int jy
int jz

struct Indices

Public Members

IndexPoint index
RealPoint intersection
BoutReal length
BoutReal angle

struct RealPoint

Public Members

BoutReal s_x
BoutReal s_y
BoutReal s_z
K.2.172 File paralleltransform.hxx

class ParallelTransform
#include <paralleltransform.hxx> Calculates the values of a field along the magnetic field (y direction)
This is used inside Mesh to represent a variety of possible numerical schemes, including Shifted Metric and FCI
Subclassed by FCITransform, ParallelTransformIdentity, ShiftedMetric

Public Functions

inline ParallelTransform(Mesh &mesh_in)

virtual ~ParallelTransform() = default

virtual void calcParallelSlices(Field3D &f) = 0
Given a 3D field, calculate and set the Y up down fields.
inline void calcYupYdown(Field3D &f)

inline virtual void integrateParallelSlices(Field3D &f)
Calculate Yup and Ydown fields by integrating over mapped points This should be used for parallel divergence operators
inline void integrateYupYdown(Field3D &f)

virtual const Field3D toFieldAligned(const Field3D &f, const std::string &region = "RGN_ALL") = 0
Convert a field into field-aligned coordinates so that the y index is along the magnetic field
inline const Field3D toFieldAligned(const Field3D &f, REGION region)

virtual const FieldPerp toFieldAligned(const FieldPerp &f, const std::string &region = "RGN_ALL") = 0
inline const FieldPerp toFieldAligned(const FieldPerp &f, REGION region)

virtual const Field3D fromFieldAligned(const Field3D &f, const std::string &region = "RGN_ALL") = 0
Convert back from field-aligned coordinates into standard form
inline const Field3D fromFieldAligned(const Field3D &f, REGION region)

virtual const FieldPerp fromFieldAligned(const FieldPerp &f, const std::string &region = "RGN_ALL") = 0
inline const FieldPerp fromFieldAligned(const FieldPerp &f, REGION region)

virtual bool canToFromFieldAligned() = 0
inline virtual void outputVars(Datafile &file)
Output variables used by a ParallelTransform instance to the dump files.
virtual bool requiresTwistShift(bool twist_shift_enabled, YDirectionType ytype) = 0
If twist_shift_enabled is true, does a Field3D with Y direction ytype require a twist-shift at branch
cuts on closed field lines?

Protected Functions

virtual void checkInputGrid() = 0
This method should be called in the constructor to check that if the grid has a ‘parallel_transform’ variable,
it has the correct value

Protected Attributes

Mesh &mesh
The mesh this paralleltransform is part of.

class ParallelTransformIdentity : public ParallelTransform
#include <paralleltransform.hxx> This class implements the simplest form of ParallelTransform where the do-
main is a logically rectangular domain, and yup() and ydown() refer to the same field.

Public Functions

inline ParallelTransformIdentity(Mesh &mesh_in)

virtual void calcParallelSlices(Field3D &f) override
Merges the yup and ydown() fields of f, so that f.yup() = f.ydown() = f
inline virtual const Field3D toFieldAligned(const Field3D &f, const std::string &region = "RGN_ALL") override
The field is already aligned in Y, so this does nothing
inline virtual const FieldPerp toFieldAligned(const FieldPerp &f, const std::string &region = "RGN_ALL") override
inline virtual const Field3D fromFieldAligned(const Field3D &f, const std::string &region = "RGN_ALL") override
The field is already aligned in Y, so this does nothing
inline virtual const FieldPerp fromFieldAligned(const FieldPerp &f, const std::string &region = "RGN_ALL") override

inline virtual bool canToFromFieldAligned() override
inline virtual bool requiresTwistShift(bool twist_shift_enabled, YDirectionType ytype) override
If twist_shift_enabled is true, does a Field3D with Y direction ytype require a twist-shift at branch
cuts on closed field lines?
Protected Functions

virtual void checkInputGrid() override
    This method should be called in the constructor to check that if the grid has a ‘parallel_transform’ variable, it has the correct value

class ShiftedMetric : public ParallelTransform
#include <paralleltransform.hxx>
Shifted metric method Each Y location is shifted in Z with respect to its neighbours so that the grid is orthogonal in X-Z, but requires interpolation to calculate the values of points along field-lines.

In this implementation the interpolation is done using FFTs in Z

Public Functions

ShiftedMetric() = delete

ShiftedMetric(Mesh &mesh, CELL_LOC location, Field2D zShift, BoutReal zlength_in)

virtual void calcParallelSlices(Field3D &f) override
    Calculates the yup() and ydown() fields of f by taking FFTs in Z and applying a phase shift.

virtual const Field3D toFieldAligned(const Field3D &f, const std::string &region = "RGN_ALL") override
    Uses FFTs and a phase shift to align the grid points with the y coordinate (along magnetic field usually).
    Note that the returned field will no longer be orthogonal in X-Z, and the metric tensor will need to be changed if X derivatives are used.
    Shift the field so that X-Z is not orthogonal, and Y is then field aligned.

evirtual const FieldPerp toFieldAligned(const FieldPerp &f, const std::string &region = "RGN_ALL")

virtual const Field3D fromFieldAligned(const Field3D &f, const std::string &region = "RGN_ALL")
    override
    Converts a field back to X-Z orthogonal coordinates from field aligned coordinates.
    Shift back, so that X-Z is orthogonal, but Y is not field aligned.

evirtual const FieldPerp fromFieldAligned(const FieldPerp &f, const std::string &region = "RGN_ALL")
    override

    inline virtual bool canToFromFieldAligned() override

evirtual void outputVars(Datafile &file) override
    Save zShift to the output.

    inline virtual bool requiresTwistShift(bool twist_shift_enabled, YDirectionType ytype) override
    If twist_shift_enabled is true, does a Field3D with Y direction ytype require a twist-shift at branch cuts on closed field lines?
Protected Functions

virtual void checkInputGrid() override

This method should be called in the constructor to check that if the grid has a ‘parallel_transform’ variable, it has the correct value

Private Functions

inline const Field2D shiftZ(const Field2D &f, const Field2D &zangle, const std::string region = "RGN_NOX") const

Shift a 2D field in Z. Since 2D fields are constant in Z, this has no effect

inline const Field2D shiftZ(const Field2D &f, const Field2D &zangle, REGION region) const

const Field3D shiftZ(const Field3D &f, const Field2D &zangle, const std::string &region = "RGN_NOX") const

Shift a 3D field f in Z by the given zangle

Parameters

• f – [in] The field to shift
• zangle – [in] Toroidal angle (z)

inline const Field3D shiftZ(const Field3D &f, const Field2D &zangle, REGION region) const

const Field3D shiftZ(const Field3D &f, const Tensor<dcomplex> &phs, const YDirectionType y_direction_out, const std::string &region = "RGN_NOX") const

Shift a 3D field or FieldPerp f by the given phase phs in Z

Calculates FFT in Z, multiplies by the complex phase and inverse FFTS.

Parameters

• f – [in] The field to shift
• phs – [in] The phase to shift by
• y_direction_out – [in] The value to set yDirectionType of the result to

const FieldPerp shiftZ(const FieldPerp &f, const Tensor<dcomplex> &phs, const YDirectionType y_direction_out, const std::string &region = "RGN_NOX") const

void shiftZ(const BoutReal *in, int len, BoutReal zangle, BoutReal *out) const

Shift a given 1D array, assumed to be in Z, by the given zangle

Parameters

• in – [in] A 1D array of length len
• len – [in] Length of the in and out arrays
• zangle – [in] The angle (z coordinate) to shift by
• out – [out] A 1D array of length len, already allocated

void shiftZ(const BoutReal *in, const dcomplex *phs, BoutReal *out) const

Shift a given 1D array, assumed to be in Z, by the given zangle

Parameters

• in – [in] A 1D array of length len
• len – [in] Length of the in and out arrays
• zangle – [in] The angle (z coordinate) to shift by
• out – [out] A 1D array of length len, already allocated
• **in** – [in] A 1D array of length mesh.LocalNz
• **phs** – [in] Phase shift, assumed to have length (mesh.LocalNz/2 + 1) i.e. the number of modes
• **out** – [out] A 1D array of length mesh.LocalNz, already allocated

```c++
void cachePhases()
{
    Calculate and store the phases for to/from field aligned and for the parallel slices using zShift
}
```

```c++
std::vector<Field3D> shiftZ(const Field3D &f, const std::vector<ParallelSlicePhase> &phases) const
{
    Shift a 3D field `f` in Z to all the parallel slices in `phases`
}
```

**Parameters**

- **f** – [in] The field to shift
- **phases** – [in] The phase and offset information for each parallel slice

**Returns** The shifted parallel slices

### Private Members

**CELL_LOC location** = **{CELL_CENTRE}**

**Field2D zShift**

This is the shift in toroidal angle (z) which takes a point from X-Z orthogonal to field-aligned along Y.

**BoutReal zlength** = {0.}

Length of the z-domain in radians.

**int nmodes**

**Tensor<dcomplex> toAlignedPhs**

Cache of phase shifts for transforming from X-Z orthogonal coordinates to field-aligned coordinates

**Tensor<dcomplex> fromAlignedPhs**

Cache of phase shifts for transforming from field-aligned coordinates to X-Z orthogonal

```c++
std::vector<ParallelSlicePhase> parallel_slice_phases
{
    Cache of phase shifts for the parallel slices. Slices are stored in the following order: {+1, ..., +n, -1, ..., -n} slice[i] stores offset i+1 slice[2*i + 1] stores offset -i+1 where i goes from 0 to (n-1), with n the number of y guard cells
}
```

**struct ParallelSlicePhase**

Helper POD for parallel slice phase shifts
Public Members

\[ \text{Tensor}<\text{dcomplex}>, \text{phase}\_\text{shift} \]

\[ \text{int } \text{y}_{-}\text{offset} \]

K.2.173 File parderiv\_factory.cxx

Variables

static const char *\text{default\_section} = "parderiv"

Default \text{Options} section to look for configuration.

K.2.174 File pdd.cxx

K.2.175 File pdd.hxx

class \text{LaplacePDD} : public \text{Laplacian}

Public Functions

inline \text{LaplacePDD}(\text{Options} *\text{opt} = \text{nullptr}, \text{const } \text{CELL}\_\text{LOC} \ \text{loc} = \text{CELL}\_\text{CENTRE}, \text{Mesh} *\text{mesh\_in} = \text{nullptr})

inline \text{~LaplacePDD}()

inline virtual void \text{setCoefA}(\text{const Field2D} &\text{val}) \text{override}

Set coefficients for inversion. Re-builds matrices if necessary.

inline virtual void \text{setCoefC}(\text{const Field2D} &\text{val}) \text{override}

inline virtual void \text{setCoefD}(\text{const Field2D} &\text{val}) \text{override}

inline virtual void \text{setCoefEx}(\text{const Field2D} &\text{val}) \text{override}

inline virtual void \text{setCoefEz}(\text{const Field2D} &\text{val}) \text{override}

virtual \text{FieldPerp} \text{solve}(\text{const FieldPerp} &\text{b}) \text{override}

virtual \text{Field3D} \text{solve}(\text{const Field3D} &\text{b}) \text{override}
Private Functions

```c
void start(const FieldPerp &b, PDD_data &data)
Laplacian inversion using Parallel Diagonal Dominant (PDD) method
July 2008: Adapted from serial version to run in parallel (split in X) for tridiagonal system i.e. no 4th order inversion yet.
```

**Note:** This code stores intermediate results and takes significantly more memory than the serial version. This can be balanced against communication time i.e. faster communications can allow less memory use.

**Parameters**
- `b` – [in] RHS values (Ax = b)
- `data` – [in] Internal data used for multiple calls in parallel mode

```c
void next(PDD_data &data)
Middle part of the PDD algorithm.
```

```c
void finish(PDD_data &data, FieldPerp &x)
Last part of the PDD algorithm.
```

Private Members

- `Field2D Acoef`
- `Field2D Ccoef`
- `Field2D Dcoef`
- `const int PDD_COMM_XV`
- `const int PDD_COMM_Y`
- `struct PDD_data`
  Data structure for PDD algorithm.

Public Members

- `Matrix<dcomplex> bk`
  b vector in Fourier space
- `Matrix<dcomplex> avec`
- `Matrix<dcomplex> bvec`
- `Matrix<dcomplex> cvec`
  Diagonal bands of matrix.
- `int jy`
  Y index.
- `Matrix<dcomplex> xk`
Matrix<\text{dcomplex}> \mathbf{v}
Matrix<\text{dcomplex}> \mathbf{w}
Array<\text{BoutReal}> \mathbf{snd}
Array<\text{BoutReal}> \mathbf{rcv}
\text{comm\_handle} \text{recv\_handle}
Array<\text{dcomplex}> \mathbf{y2i}

K.2.176 File petsc.cxx

Functions

PetscErrorCode \text{solver\_f}(\text{TS} ts, \text{BoutReal} t, \text{Vec} \text{globalin}, \text{Vec} \text{globalout}, \text{void} \*f\_data)

PetscErrorCode \text{solver\_rhsjacobian}(\text{TS} ts, \text{BoutReal} t, \text{Vec} \text{globalin}, \text{Mat} \*J, \text{Mat} \*Jpre, \text{MatStructure} \*str, \text{void} \*f\_data)

PetscErrorCode \text{solver\_ijacobianfd}(\text{TS}, \text{PetscReal}, \text{Vec}, \text{Vec}, \text{PetscReal}, \text{Mat}*, \text{Mat}*, \text{MatStructure}*, \text{void}*)

PetscErrorCode \text{solver\_if}(\text{TS}, \text{BoutReal}, \text{Vec}, \text{Vec}, \text{Vec}, \text{void}*)

PetscErrorCode \text{Physics\_PCApply}(\text{PC}, \text{Vec} \text{x}, \text{Vec} \text{y})
  \text{KSP preconditioner PCShell routines for physics preconditioners.}

PetscErrorCode \text{Physics\_Jacobian\_Apply}(\text{Mat} \text{J}, \text{Vec} \text{x}, \text{Vec} \text{y})

PetscErrorCode \text{Physics\_SNES\_Apply}(\text{SNES}, \text{Vec})

PetscErrorCode \text{solver\_rhsjacobian}(\text{MAYBE\_UNUSED}(\text{TS} ts), \text{MAYBE\_UNUSED}(\text{BoutReal} t),
  \text{MAYBE\_UNUSED}(\text{Vec} \text{globalin}), \text{Mat} \*J, \text{Mat} \*Jpre,
  \text{MAYBE\_UNUSED}(\text{MatStructure} \*str), \text{MAYBE\_UNUSED}(\text{void}
  \*f\_data))

PetscErrorCode \text{solver\_ijacobian}(\text{TS} ts, \text{BoutReal} t, \text{Vec} \text{globalin}, \text{MAYBE\_UNUSED}(\text{Vec} \text{globalindot}),
  \text{MAYBE\_UNUSED}(\text{PetscReal} \text{a}), \text{Mat} \*J, \text{Mat} \*Jpre, \text{MatStructure} \*str, \text{void}
  \*f\_data))

PetscErrorCode \text{solver\_ijacobianfd}(\text{TS} ts, \text{BoutReal} t, \text{Vec} \text{globalin}, \text{Vec} \text{globalindot}, \text{PetscReal} \text{a}, \text{Mat} \*J, \text{Mat}
  \*Jpre, \text{MatStructure} \*str, \text{void} \*f\_data)

PetscErrorCode \text{Petsc\_Monitor}(\text{TS} ts, \text{PetscInt} step, \text{PetscReal} t, \text{Vec} \text{X}, \text{void} \*ctx)
  \text{Monitor} function called on every internal timestep.

PetscErrorCode \text{Petsc\_SNES\_Monitor}(\text{SNES} \text{snes}, \text{PetscInt} \text{its}, \text{PetscReal} \text{norm}, \text{void} \*ctx)
  \text{Monitor} function for SNES.
K.2.177 File petsc.hxx

Defines

OPT_SIZE

Typedefs

using BoutReal = PetscScalar
using rhsfunc = int (*)(BoutReal)

Functions

PetscErrorCode PetscMonitor(TS, PetscInt, PetscReal, Vec, void *ctx)
         Monitor function called on every internal timestep.

PetscErrorCode PetscSNESMonitor(SNES, PetscInt, PetscReal, void *ctx)
         Monitor function for SNES.

PetscErrorCode solver_ijacobian(TS, PetscReal, Vec, Vec, PetscReal, Mat*, Mat*, MatStructure*, void*)
         Compute IJacobian = dF/dU + a dF/dUdot - a dummy matrix used for pc=none.

Variables

BoutReal simtime
struct snes_info
"include <petsc.hxx> Data for SNES.

Public Members

PetscInt it
PetscInt linear_its
PetscReal time
PetscReal norm
class PetscSolver : public Solver
Public Functions

PetcsSolver(Options *opts = nullptr)

~PetcsSolver()

virtual int init(int NOUT, BoutReal TIMESTEP) override
   Initialise the solver NOTE: nout and tstep should be passed to run, not init. Needed because of how the
   PETSc TS code works

virtual int run() override
   Run the solver, calling monitors nout times, at intervals of tstep. This function is called by solve(), and is
   specific to each solver type
   This should probably be protected, since it shouldn't be called by users.

PetscErrorCode rhs(TS ts, PetscReal t, Vec globalin, Vec globalout)
   Wrapper for the RHS function.

PetscErrorCode pre(PC pc, Vec x, Vec y)
   Wrapper for the preconditioner.

PetscErrorCode jac(Vec x, Vec y)
   Wrapper for the Jacobian function.

Public Members

PetcsLogEvent solver_event
PetcsLogEvent loop_event
PetcsLogEvent init_event

Private Members

BoutReal shift
   Shift (alpha) parameter from TS.

Vec state

BoutReal ts_time
   Internal PETSc timestepper time.

PetscLib lib
   Handles initialising, finalising PETSc.

Vec u
   PETSc solution vector.

TS ts
   PETSc timestepper object.

Mat J
Mat $J_{mf}$
   RHS Jacobian.

MatFDColoring $\text{matfdcoloring}$

int $\text{nout}$
   The number of outputs.

$\text{BoutReal tstep}$
   Time between outputs.

bool $\text{diagnose}$
   If true, print some information about current stage.

$\text{BoutReal next_output}$
   When the monitor should be called next.

PetscBool $\text{interpolate}$
   Whether to interpolate or not.

char $\text{output_name}$[PETSC_MAX_PATH_LEN]

PetscBool $\text{output_flag}$

PetscInt $\text{prev_linear_its}$

$\text{BoutReal bout_snes_time}$

$\text{std::vector<snes_info> snes_list}$

bool $\text{adaptive}$
   Use adaptive timestepping.

**Friends**

friend friend PetscErrorCode PetscMonitor (TS, PetscInt, PetscReal, Vec, void *ctx)
   Monitor function called on every internal timestep.

friend friend PetscErrorCode PetscSNESMonitor (SNES, PetscInt, PetscReal, void *ctx)
   Monitor function for SNES.

friend friend PetscErrorCode solver_ijacobian (TS, PetscReal, Vec, Vec, PetscReal, Mat *, Mat *, MatStructure *, void *)
   Compute $\text{IJacobian} = dF/dU + a \ dF/dUdot$ - a dummy matrix used for $\text{pc=none}$. 
K.2.178 File petsc_laplace.cxx

Defines

KSP_RICHARDSON
KSP_CHEBYSHEV
KSP.CG
KSP_GMRES
KSP_TCQMR
KSP_BCGS
KSP_CGS
KSP_TFQMR
KSP_CR
KSP_LSQR
KSP_BICG
KSP_PREONLY

Functions

static PetscErrorCode laplacePCapply(PC pc, Vec x, Vec y)

K.2.179 File petsc_laplace.hxx

class LaplacePetsc : public Laplacian

Public Functions

LaplacePetsc(Options *opt = nullptr, const CELL_LOC loc = CELL_CENTRE, Mesh *mesh_in = nullptr)

inline ~LaplacePetsc()

inline virtual void setCoefA(const Field2D &val) override
    Set coefficients for inversion. Re-builds matrices if necessary.
inline virtual void setCoefC(const Field2D &val) override
inline virtual void setCoefC1(const Field2D &val) override
inline virtual void setCoefC2(const Field2D &val) override
inline virtual void setCoefD(const Field2D &val) override
inline virtual void setCoefEx(const Field2D &val) override
inline virtual void setCoefEz(const Field2D &val) override
inline virtual void setCoefA(const Field3D &val) override
inline virtual void setCoefC(const Field3D &val) override
inline virtual void setCoefC1(const Field3D &val) override
inline virtual void setCoefC2(const Field3D &val) override
inline virtual void setCoefD(const Field3D &val) override
inline virtual void setCoefEx(const Field3D &val) override
inline virtual void setCoefEz(const Field3D &val) override

virtual FieldPerp solve(const FieldPerp &b) override

virtual FieldPerp solve(const FieldPerp &b, const FieldPerp &x0) override
Solves Ax=b for x given a b and an initial guess for x (x0)
This function will:
  i. Set the matrix element of the matrix A, used to solve Ax=b (this includes setting the values for the
     boundary condition)
  ii. Solve the matrix Ax = b

Parameters
  • b – [in] The RHS of the equation Ax=b. This is an y-slice of the original field. The
    field will be flattened to an 1D array in order to write the equation on the form Ax=b
  • x0 – [in] The initial guess for the solver. May also contain the boundary condition
    if flag 32 - INVERT_SET is set

Returns  sol The solution x of the problem Ax=b.

int precon(Vec x, Vec y)
Preconditioner function.
Private Functions

void Element(int i, int x, int z, int xshift, int zshift, PetscScalar ele, Mat &MatA)

Sets the elements of the matrix A, which is used to solve the problem Ax=b.

Parameters

- **i** – [in] The row of the PETSc matrix
- **x** – [in] Local x index of the mesh
- **z** – [in] Local z index of the mesh
- **xshift** – [in] The shift in rows from the index x
- **zshift** – [in] The shift in columns from the index z
- **ele** – [in] Value of the element
- **MatA** – [in] The matrix A used in the inversion
- **MatA** – [out] The matrix A used in the inversion

void Coeffs(int x, int y, int z, BoutReal &A1, BoutReal &A2, BoutReal &A3, BoutReal &A4, BoutReal &A5)

Set the matrix components of A in Ax=b, solving D*Laplace_perp(x) + (1/C1)*Grad_perp(C2)*Grad_perp(x) + Ax = B

Note: “A” in the equation above is not added here. For calculations of the coefficients, please refer to the user manual.

Parameters

- **x** – [in] The current x index
- **y** – [in] The current y index
- **z** – [in] The current y index
- **coef1** – [in] Placeholder for convenient variable used to set matrix (see manual for details)
- **coef2** – [in] Convenient variable used to set matrix (see manual for details)
- **coef3** – [in] Placeholder for convenient variable used to set matrix (see manual for details)
- **coef4** – [in] Placeholder for convenient variable used to set matrix (see manual for details)
- **coef5** – [in] Placeholder for convenient variable used to set matrix (see manual for details)
- **coef1** – [out] Convenient variable used to set matrix (see manual for details)
- **coef2** – [out] Convenient variable used to set matrix (see manual for details)
- **coef3** – [out] Convenient variable used to set matrix (see manual for details)
- **coef4** – [out] Convenient variable used to set matrix (see manual for details)
- **coef5** – [out] Convenient variable used to set matrix (see manual for details)
void vecToField(Vec x, FieldPerp &f)

void fieldToVec(const FieldPerp &f, Vec x)

Private Members

Field3D A
Field3D C1
Field3D C2
Field3D D
Field3D Ex
Field3D Ez
bool issetD
bool issetC
bool issetE
int lastflag
FieldPerp sol
int Istart
int Iend
int meshx
int meshz
int size
int localN
MPI_Comm comm
Mat MatA
Vec xs
Vec bs
KSP ksp
Options *opts
std::string ksptype
   KSP solver type.

std::string pctype
   Preconditioner type.

BoutReal richardson_damping_factor
BoutReal chebyshev_max
BoutReal chebyshev_min
int gmres_max_steps

BoutReal rtol

BoutReal atol

BoutReal dtol

int maxits

bool direct

bool fourth_order

PetscLib lib

bool use_precon

bool rightprec

Laplacian *pcsolve

int implemented_flags

int implemented_boundary_flags

K.2.180 File petsclib.cxx

K.2.181 File petsclib.hxx

Defines

PETSC_VERSION_GE(MAJOR, MINOR, SUBMINOR)

class PetscLib

#include <petsclib.hxx> Handles initialisation and finalisation of PETSc library. The first instance which is created initialises PETSc. Keeps a count of the number of how many instances exist. When the last instance is destroyed it finalises PETSc.

Public Functions

explicit PetscLib(Options *opt = nullptr)

Ensure that PETSc has been initialised

~PetscLib() Calls PetscFinalize when all PetscLib instances are destroyed

void setOptionsFromInputFile(KSP &ksp)

Set options for a KSP linear solver that uses the options specific to this PetscLib, by setting an options prefix for the KSP, and adding that prefix to all the options set in the [petsc] section, or [petsc] subsection of the options, if non-null ‘opt’ was passed to the constructor.

void setOptionsFromInputFile(SNES &snes)

Set options for a SNES linear solver that uses the options specific to this PetscLib, by setting an options prefix for the SNES, and adding that prefix to all the options set in the [petsc] section, or [petsc] subsection of the options, if non-null ‘opt’ was passed to the constructor.
Public Static Functions

static inline void setArgs(int &c, char **&v)
This is called once to set the command-line options. Should be done early in the program, before any instances of PetscLib are created. The arguments will be passed to PetscInitialize()

static void cleanup()
Force cleanup. This will call PetscFinalize, printing a warning if any instances of PetscLib still exist

Private Functions

void setPetscOptions(Options &options, const std::string &pass_options_prefix)

Private Members

std::string options_prefix

Private Static Attributes

static int count = 0
How many instances?

static char help[] = "BOUT++: Uses finite difference methods to solve plasma fluid problems in curvilinear coordinates"
Help string.

static int *pargc = nullptr
static char ***pargv = nullptr
static PetscLogEvent USER_EVENT = 0

K.2.182 File physicsmodel.cxx
Defines

BOUT_NO_USING_NAMESPACE_BOUTGLOBALS

K.2.183 File physicsmodel.hxx
Base class for Physics Models.

Changelog:
2013-08 Ben Dudson benjamin.dudson@york.ac.uk
• Initial version
Defines

**BOUTMAIN**(ModelClass)

Macro to define a simple main() which creates the given model and runs it. This should be sufficient for most use cases, but a user can define their own main() function if needed.

*Example*

```cpp
class MyModel : public PhysicsModel { .. };
BOUTMAIN(MyModel);
```

**SOLVE_FOR1**(var)

Macro to replace solver->add, passing variable name.

**SOLVE_FOR2**(var1, var2)

**SOLVE_FOR3**(var1, var2, var3)

**SOLVE_FOR4**(var1, var2, var3, var4)

**SOLVE_FOR5**(var1, var2, var3, var4, var5)

**SOLVE_FOR6**(var1, var2, var3, var4, var5, var6)

**SOLVE_FOR**(...)

Add fields to the solver. This should accept up to ten arguments

`PhysicsModel`

*include <physicmodel.hxx> Base class for physics models

Subclassed by `LegacyModel`
Public Types

using preconfunc = int (PhysicsModel::*)(BoutReal t, BoutReal gamma, BoutReal delta)
using jacobianfunc = int (PhysicsModel::*)(BoutReal t)

Public Functions

PhysicsModel()

virtual ~PhysicsModel() = default

inline void initialise(Solver *s)
    Initialise the model, calling the init() and postInit() methods
    Note: this is usually only called by the Solver

int runRHS(BoutReal time)
    Run the RHS function, to calculate the time derivatives
    Input

    The system state should be in the evolving variables

    The time derivatives will be put in the ddt() variables
    Returns a flag: 0 indicates success, non-zero an error flag
    Parameters
time – [in] The simulation time

bool splitOperator()
    True if this model uses split operators

int runConvective(BoutReal time)
    Run the convective (usually explicit) part of the model

int runDiffusive(BoutReal time, bool linear)
    Run the diffusive (usually implicit) part of the model

bool hasPrecon()
    True if a preconditioner has been defined

int runPrecon(BoutReal t, BoutReal gamma, BoutReal delta)
    Run the preconditioner. The system state should be in the evolving variables, and the vector to be solved
    in the ddt() variables. The result will be put in the ddt() variables.
    Note: this is usually only called by the Solver

bool hasJacobian()
    True if a Jacobian function has been defined

int runJacobian(BoutReal t)
    Run the Jacobian-vector multiplication function
    Note: this is usually only called by the Solver
inline int \texttt{runTimestepMonitor}(\texttt{BoutReal \texttt{simtime}, BoutReal \texttt{dt}})

\textbf{Protected Functions}

virtual int \texttt{init}(bool \texttt{restarting}) = 0
This function is called once by the solver at the start of a simulation.
A valid \texttt{PhysicsModel} must implement this function
Variables should be read from the inputs, and the variables to be evolved should be specified.

virtual int \texttt{postInit}(bool \texttt{restarting})
Post-initialise. This reads the restart file

\begin{verbatim}
\textbf{Parameters}
\texttt{restarting} \textbf{– [in]} If true, will load state from restart file
\end{verbatim}

inline virtual int \texttt{rhs}(\texttt{BoutReal \texttt{t}})
This function is called by the time integration solver at least once per time step.
Variables being evolved will be set by the solver before the call, and this function must calculate and set the time-derivatives.
By default this function just returns an error, which will stop the simulation.

inline virtual int \texttt{convective}(\texttt{BoutReal \texttt{t}})

inline virtual int \texttt{diffusive}(\texttt{BoutReal \texttt{t}})

inline virtual int \texttt{diffusive}(\texttt{BoutReal \texttt{t}}, \texttt{bool linear})

inline virtual int \texttt{outputMonitor}(\texttt{BoutReal \texttt{simtime}, int \texttt{iter}, int \texttt{NOUT}})
Implemented by user code to monitor solution at output times

inline virtual int \texttt{timestepMonitor}(\texttt{BoutReal \texttt{simtime}, BoutReal \texttt{dt}})
Timestep monitor. If enabled by setting solver:monitor_timestep=true then this function is called every internal timestep.

inline void \texttt{setSplitOperator}(bool \texttt{split = true})
Specify that this model is split into a convective and diffusive part.

inline void \texttt{setPrecon}(\texttt{preconfunc pset})
Specify a preconditioner function.

inline void \texttt{setJacobian}(\texttt{jacobianfunc jset})
Specify a Jacobian-vector multiply function.

void \texttt{bout\_solve}(\texttt{Field2D \&var, const char *name, const std::string &description = ""})
Specify a variable for the solver to evolve

Note that the variable must not be destroyed (e.g. go out of scope) after this call, since a pointer to \texttt{var} is stored in the solver.

To evolve the state, the solver will set \texttt{var}, and the user-supplied \texttt{rhs()} function should calculate ddt(var).

\begin{verbatim}
\textbf{Parameters}
\begin{itemize}
\item \texttt{var} \textbf{– [in]} The variable to evolve
\end{itemize}
\end{verbatim}
• name – [in] The name to use for variable initialisation and output

```cpp
void bout_solve(Field3D &var, const char *name, const std::string &description = "")
```

```cpp
void bout_solve(Vector2D &var, const char *name, const std::string &description = "")
```

```cpp
void bout_solve(Vector3D &var, const char *name, const std::string &description = "")
```

bool bout_constrain(Field3D &var, Field3D &F_var, const char *name)

Specify a constrained variable var, which will be adjusted to make F_var equal to zero. If the solver does not support constraints then this will throw an exception

**Parameters**

- var – [in] The variable the solver should modify
- F_var – [in] The control variable, which the user will set
- name – [in] The name to use for initialisation and output

### Protected Attributes

* Solver *solver = {nullptr}*

This is set by a call to initialise, and can be used by models to specify evolving variables.

* Datafile restart*

Stores the state for restarting.

* PhysicsModelMonitor modelMonitor*

write restarts and pass outputMonitor method inside a Monitor subclass

### Private Members

bool splitop = {false}

Split operator model?

```cpp
preconfunc userprecon = {nullptr}
```

Pointer to user-supplied preconditioner function.

```cpp
jacobianfunc userjacobian = {nullptr}
```

Pointer to user-supplied Jacobian-vector multiply function.

bool initialised = {false}

True if model already initialised.

class PhysicsModelMonitor : public Monitor

#include <physicsmodel.hxx> Monitor class for PhysicsModel
**Public Functions**

*PhysicsModelMonitor()* = delete

```cpp
inline PhysicsModelMonitor(PhysicsModel *model)
```

```cpp
inline virtual int call(Solver *solver, BoutReal simtime, int iter, int nout) override
```

Callback function for the solver, called after timestep_ has passed

**Parameters**

- `solver` – [in] The solver calling this monitor
- `time` – [in] The current simulation time
- `iter` – [in] The current simulation iteration
- `nout` – [in] The total number of iterations for this simulation

**Returns** non-zero if simulation should be stopped

**Private Members**

```
PhysicsModel *model
```

**K.2.184 File pnetcdf.cxx**

**Defines**

```
CHKERR(ret)
```

**Functions**

```cpp
int pnc_get_var_all(int nfile, int var, double *data)
```

```cpp
int pnc_get_var_all(int nfile, int var, float *data)
```

```cpp
int pnc_get_vara_all(int nfile, int var, MPI_Offset *start, MPI_Offset *count, double *data)
```

```cpp
int pnc_get_vara_all(int nfile, int var, MPI_Offset *start, MPI_Offset *count, float *data)
```

```cpp
int pnc_put_var_all(int nfile, int var, double *data)
```

```cpp
int pnc_put_var_all(int nfile, int var, float *data)
```

```cpp
int pnc_put_vara_all(int nfile, int var, MPI_Offset *start, MPI_Offset *count, double *data)
```
int pnc_put_vara_all(int ncfile, int var, MPI_Offset *start, MPI_Offset *count, float *data)

**K.2.185 File pnetcdf.hxx**

Parallel NetCDF data format interface.

Records: In netCDF, the time dimension for each dimension must be the same. Hence when a record is appended to a variable, the size of all variables is increased. To work out which record to write to, a map of variable names to record number is kept.

*Author* B.Dudson

*Date* May 2012

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see [http://www.gnu.org/licenses/](http://www.gnu.org/licenses/).

class PncFormat : public DataFormat

**Public Functions**

PncFormat(*Mesh *mesh_in = nullptr*)

PncFormat(const char *name, *Mesh *mesh_in = nullptr*)

inline PncFormat(const std::string &name, *Mesh *mesh_in = nullptr*)

~PncFormat() virtual bool openr(const char *name) override

inline virtual bool openr(const std::string &name, int mype)

virtual bool openw(const char *name, bool append = false) override

inline virtual bool openw(const std::string &name, int mype, bool append = false)
inline virtual bool is_valid() override

virtual void close() override

virtual void flush() override

inline const char *filename()

virtual const vector<int> getSize(const char *var) override

inline virtual const vector<int> getSize(const std::string &var) override

virtual bool setGlobalOrigin(int x = 0, int y = 0, int z = 0) override

virtual bool setRecord(int t) override

virtual bool read(int *var, const char *name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read(int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read(BoutReal *var, const char *name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read(BoutReal *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool write(int *var, const char *name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write(int *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write(BoutReal *var, const char *name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write(BoutReal *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool read_rec(int *var, const char *name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read_rec(int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read_rec(BoutReal *var, const char *name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool read_rec(BoutReal *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0) override

virtual bool write_rec(int *var, const char *name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write_rec(int *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write_rec(BoutReal *var, const char *name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write_rec(BoutReal *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) override
virtual bool write_rec(int *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write_rec(BoutReal *var, const char *name, int lx = 0, int ly = 0, int lz = 0) override

virtual bool write_rec(BoutReal *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0) override

inline virtual void setLowPrecision() override

**Private Members**

char *fname  
Current file name.

int ncf ile  
ID of netCDF file.

bool valid  
int xDim  
Dimensions.

int yDim  
int zDim  
int tDim  
int *dimList  
List of dimensions (x,y,z)

int recDimList[4]  
List of dimensions (t,x,y,z)

bool appending  
bool lowPrecision  
When writing, down-convert to floats.

int x0  
int y0  
int z0  
int t0  
Data origins (global offsets)

std::map<std::string, int> rec_nr

int default_rec
K.2.186 File power.cxx

K.2.187 File power.hxx

class **PowerSolver** : public **Solver**

**Public Functions**

```cpp
inline **PowerSolver**()
```

```cpp
inline **PowerSolver**(Options*)
```

```cpp
inline ~**PowerSolver**()
```

virtual int **init**(int nout, BoutReal tstep) override

Initialise the solver

NOTE: nout and tstep should be passed to run, not init.
Needed because of how the PETSc TS code works

```cpp
virtual int **run**() override
```

Run the solver, calling monitors nout times, at intervals of tstep.
This function is called by **solve()**, and is specific to each solver type

This should probably be protected, since it shouldn’t be called by users.

```cpp
inline virtual void **outputVars**(Datafile &outputfile, bool save_repeat = true) override
```

Add evolving variables to output (dump) file or restart file

**Parameters**

- **outputfile** – [inout] The file to add variable to
- **save_repeat** – [in] If true, add variables with time dimension

**Private Functions**

```cpp
BoutReal **norm**(Array<BoutReal> &state)
```

```cpp
void **divide**(Array<BoutReal> &in, BoutReal value)
```

**Private Members**

```cpp
BoutReal curtime
```

```cpp
BoutReal eigenvalue
```

```cpp
int nlocal
```

```cpp
int nglobal
```

```cpp
Array<BoutReal> f0
```

```cpp
int nsteps
```
K.2.188 File pvode.cxx

Functions

void solver_f(integer N, BoutReal t, N_Vector u, N_Vector udot, void *f_data)

void solver_gloc(integer N, BoutReal t, BoutReal *u, BoutReal *udot, void *f_data)

void solver_cfn(integer N, BoutReal t, N_Vector u, void *f_data)

Variables

const BoutReal ZERO = 0.0
long int iopt[OPT_SIZE]
BoutReal ropt[OPT_SIZE]
namespace pvode

K.2.189 File pvode.hxx

class PvedSolver : public Solver

 Public Functions

PvedSolver(Options *opts)

-PvedSolver()

inline virtual BoutReal getCurrentTimestep() override
 Return the current internal timestep.

virtual int init(int nout, BoutReal tstep) override
 Initialise the solver NOTE: nout and tstep should be passed to run, not init. Needed because of how the
PETSc TS code works

virtual int run() override
 Run the solver, calling monitors nout times, at intervals of tstep. This function is called by solve(), and is
specific to each solver type

 This should probably be protected, since it shouldn’t be called by users.

BoutReal run(BoutReal tout)

void rhs(int N, BoutReal t, BoutReal *udata, BoutReal *dudata)

void gloc(int N, BoutReal t, BoutReal *udata, BoutReal *dudata)
Private Members

int NOUT
BoutReal TIMESTEP
BoutReal hcur
pvode::N_Vector u
pvode::machEnvType machEnv
void *cvode_mem
BoutReal abstol
BoutReal reltol
pvode::PVBBDData pdata
bool pvode_initialised = false

K.2.190 File range.cxx

K.2.191 File range.hxx

class RangeIterator

Public Functions

RangeIterator() = default
Can be given a single range.

RangeIterator(int start, int end, RangeIterator *join = nullptr)

RangeIterator(int start, int end, const RangeIterator &join)

RangeIterator(const RangeIterator &r)

~RangeIterator()

void first()

void next()

bool isDone() const

inline int operator*()

inline RangeIterator &operator++()
inline RangeIterator operator++(int)

inline bool operator==(const RangeIterator &x) const

inline bool operator!=(const RangeIterator &x) const

bool intersects(const RangeIterator &other, bool all = true) const

bool intersects(int ind, bool all = true) const

RangeIterator &operator=(const RangeIterator &r)

RangeIterator &operator+=(const RangeIterator &r)

RangeIterator &operator-=(const RangeIterator &r)

inline int min() const

inline int max() const

inline RangeIterator *nextRange() const

Public Members

int ind

Public Static Functions

static inline RangeIterator end()

Private Members

int is = {1}
int ie = {0}

RangeIterator *n = {nullptr}

RangeIterator *cur = {nullptr}

int curend

bool delete_next = false
K.2.192 File region.hxx

Defines

MAXREGIONBLOCKSIZE

Flexible iterator for Field2D/Field3D

The Region class keeps hold of a set of indices which are used to index a Field. Here, a Field refers to either a Field2D or a Field3D. These indices can either be used directly, or blocks of contiguous indices may be used instead, which allows OpenMP parallelisation.

The BOUT_FOR helper macro is provided as a replacement for for-loops.

Separate index classes are available for Field2Ds (Ind2D) and Field3Ds (Ind3D). Note that while an Ind3D can be used to index a Field2D, an Ind2D cannot be used to index a Field3D. This is because an Ind2D essentially doesn’t keep track of the z-dimension. The MAXREGIONBLOCKSIZE value can be tuned to try to optimise performance on specific hardware. It determines what the largest contiguous block size can be. As we hope the compiler will vectorise the access to these contiguous blocks, the optimal MAXREGIONBLOCKSIZE is likely related to the vector size etc.

BOUT_FOR_SERIAL(index, region)

Helper macros for iterating over a Region making use of the contiguous blocks of indices

These macros all have the same basic form: an outer loop over blocks of contiguous indices, and an inner loop over the indices themselves. This allows the outer loop to be parallelised with OpenMP while the inner loop can be vectorised with the CPU’s native SIMD instructions.

Alternative forms are also provided for loops that must be done serially, as well as for more control over the OpenMP directives used.

The different macros:

- BOUT_FOR: OpenMP-aware version that allows speedup with both OpenMP and native SIMD vectorisation. This should be the preferred form for most loops
- BOUT_FOR_SERIAL: for use with inherently serial loops. If BOUT++ was not compiled with OpenMP, BOUT_FOR falls back to using this form
- BOUT_FOR_INNER: for use on loops inside OpenMP parallel regions, e.g. in order to declare thread private variables outside of the loop
- BOUT_FOR_OMP: the most generic form, that takes arbitrary OpenMP directives as an extra argument

Example

The following for-loop:

```cpp
for (auto index = begin(region); index < end(region); ++index) {
    A[index] = B[index] + C[index];
}
```

can be converted to a block region loop like so:

```cpp
BOUT_FOR(index, region) {
    A[index] = B[index] + C[index];
}
```

Parameters

- `index` – [in] The name of the index variable to use in the loop
• region – [in] An already existing Region

`BOUT_FOR_OMP(index, region, omp_pragmas)`

`BOUT_FOR(index, region)`

`BOUT_FOR_INNER(index, region)`

**Typedefs**

using `Ind3D = SpecificInd<IND_TYPE::IND_3D>`
Define aliases for global indices in 3D and 2D.

using `Ind2D = SpecificInd<IND_TYPE::IND_2D>`

using `IndPerp = SpecificInd<IND_TYPE::IND_PERP>`

**Enums**

enum `IND_TYPE`
Values:

enumerator `IND_3D`
enumerator `IND_2D`
enumerator `IND_PERP`

**Functions**

template<`IND_TYPE N`>
inline bool `operator==`(const `SpecificInd<N>` &lhs, const `SpecificInd<N>` &rhs)
Relational operators.

template<`IND_TYPE N`>
inline bool `operator!=`(const `SpecificInd<N>` &lhs, const `SpecificInd<N>` &rhs)


template<`IND_TYPE N`>
inline bool `operator<`(const `SpecificInd<N>` &lhs, const `SpecificInd<N>` &rhs)


template<`IND_TYPE N`>
inline bool `operator>`(const `SpecificInd<N>` &lhs, const `SpecificInd<N>` &rhs)


template<`IND_TYPE N`>
inline bool `operator>==`(const `SpecificInd<N>` &lhs, const `SpecificInd<N>` &rhs)
inline bool operator<=(const SpecificInd<N> &lhs, const SpecificInd<N> &rhs)

#define template<IND_TYPE N>
inline SpecificInd<N> operator+(SpecificInd<N> lhs, const SpecificInd<N> &rhs)
Arithmetic operators with integers.

define template<IND_TYPE N>
inline SpecificInd<N> operator+(SpecificInd<N> lhs, int n)
define template<IND_TYPE N>
inline SpecificInd<N> operator+(int n, SpecificInd<N> rhs)

define template<IND_TYPE N>
inline SpecificInd<N> operator-(SpecificInd<N> lhs, int n)
define template<IND_TYPE N>
inline SpecificInd<N> operator-(SpecificInd<N> lhs, const SpecificInd<N> &rhs)

define inline const std::string toString(const Ind3D &i)
Get string representation of Ind3D.
    Get string representation of IndPerp.
    Get string representation of Ind2D.

define inline std::ostream &operator<<(std::ostream &out, const RegionStats &stats)
Provide an easy way to report a Region’s statistics.

define template<typename T>
Region<T> sort(Region<T> &region)
    Return a new region with sorted indices.

define template<typename T>
Region<T> unique(Region<T> &region)
    Return a new region with unique indices.

define template<typename T>
Region<T> mask(const Region<T> &region, const Region<T> &mask)
    Return a masked version of a region.

define template<typename T>
Region<T> operator+(const Region<T> &lhs, const Region<T> &rhs)
    Return a new region with combined indices from two Regions This doesn’t attempt to avoid duplicate elements or enforce any sorting etc. but could be done if desired.
    • Addition is currently simple and just extends. Probably mostly ok but we could seek to remove duplicate points. Note we do want to allow duplicate points (one reason we use vector and not set) but what if we add a region that has some duplicates? We could retain them but common usage would probably not want the duplicates.

define template<typename T>
Region<T> offset(const Region<T> &region, int offset)
    Returns a new region based on input but with indices offset by a constant

define template<typename T>
unsigned int size(const Region<T> &region)
    Return the number of indices in a Region.
template<IND_TYPE N>
class SpecificInd
    #include <region.hxx> Indices base class for Fields Regions are dereferenced into these

    Provides methods for offsetting by fixed amounts in x, y, z, as well as a generic method for offsetting by any
    amount in multiple directions.

    Assumes that the offset is less than the grid size in that direction. This assumption is checked for at CHECK=3.
    This assumption implies that a FieldPerp cannot be offset in y, and a Field2D cannot be offset in z. A stronger,
    more expensive check that the resulting offset index doesn’t go out of bounds can be enabled at CHECK=4.

    Also provides helper methods for converting Ind2D/Ind3D/IndPerp to x, y, z indices

    Examples

    Field3D field, result;
    auto index = std::begin(region);

    result = field[index->yp()] - field[index->ym()];

Public Functions

SpecificInd() = default

inline SpecificInd(int i, int ny, int nz)

inline explicit SpecificInd(int i)

inline SpecificInd &operator++()
    Pre-increment operator.

inline SpecificInd operator++(int)
    Post-increment operator.

inline SpecificInd &operator--()
    Pre-decrement operator.

inline SpecificInd operator--(int)
    Post-decrement operator.

inline SpecificInd &operator+=(SpecificInd n)
    In-place addition.

inline SpecificInd &operator+=(int n)

inline SpecificInd &operator-=(SpecificInd n)
    In-place subtraction.

inline SpecificInd &operator-=(int n)

inline SpecificInd operator%(int n)
    Modulus operator.
inline int \texttt{x}() const  
Convenience functions for converting to (x, y, z)

inline int \texttt{y}() const

inline int \texttt{z}() const

template<int \texttt{dd}, \texttt{DIRECTION} \texttt{dir}>  
inline const \texttt{SpecificInd} plus() const  
Templated routine to return index.?p(offset), where ? is one of \{x,y,z\} and is determined by the \texttt{dir} template argument. The offset corresponds to the \texttt{dd} template argument.

template<int \texttt{dd}, \texttt{DIRECTION} \texttt{dir}>  
inline const \texttt{SpecificInd} minus() const  
Templated routine to return index.?m(offset), where ? is one of \{x,y,z\} and is determined by the \texttt{dir} template argument. The offset corresponds to the \texttt{dd} template argument.

inline const \texttt{SpecificInd} \texttt{xp} (int \texttt{dx} = 1) const

inline const \texttt{SpecificInd} \texttt{xm} (int \texttt{dx} = 1) const  
The index one point -1 in x.

inline const \texttt{SpecificInd} \texttt{yp} (int \texttt{dy} = 1) const  
The index one point +1 in y.

inline const \texttt{SpecificInd} \texttt{ym} (int \texttt{dy} = 1) const  
The index one point -1 in y.

inline const \texttt{SpecificInd} \texttt{zp} (int \texttt{dz} = 1) const  
The index one point -1 in z. Wraps around zend to zstart An alternative, non-branching calculation is : \texttt{ind} + \texttt{dz} - \texttt{nz} * ( (\texttt{ind} + \texttt{dz}) / \texttt{nz} - \texttt{ind} / \texttt{nz} ) but this appears no faster (and perhaps slower).

inline const \texttt{SpecificInd} \texttt{zm} (int \texttt{dz} = 1) const  
The index one point -1 in z. Wraps around zstart to zend An alternative, non-branching calculation is : \texttt{ind} - \texttt{dz} + \texttt{nz} * ( (\texttt{nz} + \texttt{ind}) / \texttt{nz} - (\texttt{nz} + \texttt{ind} - \texttt{dz}) / \texttt{nz} ) but this appears no faster (and perhaps slower).

inline const \texttt{SpecificInd} \texttt{xpp}() const

inline const \texttt{SpecificInd} \texttt{xmm}() const

inline const \texttt{SpecificInd} \texttt{ypp}() const

inline const \texttt{SpecificInd} \texttt{ymm}() const

inline const \texttt{SpecificInd} \texttt{zpp}() const

inline const \texttt{SpecificInd} \texttt{zmm}() const

inline const \texttt{SpecificInd} \texttt{offset} (int \texttt{dx}, int \texttt{dy}, int \texttt{dz}) const  
Generic offset of \texttt{index} in multiple directions simultaneously.
Public Members

int \texttt{ind} = -1

Private Members

int \texttt{ny} = -1
int \texttt{nz} = -1

struct \texttt{RegionStats} 
# include <region.hxx> Structure to hold various derived “statistics” from a particular region.

Public Members

int \texttt{numBlocks} = 0
How many blocks.

int \texttt{minBlockSize} = 0
Size of smallest block.

int \texttt{numMinBlocks} = 0
Number of blocks with min size.

int \texttt{maxBlockSize} = 0
Size of largest block.

int \texttt{numMaxBlocks} = 0
Number of blocks with max size.

int \texttt{numSmallBlocks} = 0
Number of “small” blocks, for definition see \texttt{Region::getStats}.

\texttt{BoutReal maxImbalance} = 0
Ratio of largest block to smallest.

template<\texttt{typename T} = \texttt{Ind3D}>

class \texttt{Region} 
# include <region.hxx> Specifies a set of indices which can be iterated over and \texttt{begin()} and \texttt{end()} methods for range-based for loops.

\texttt{Region} is templated on either \texttt{Ind2D} or \texttt{Ind3D} for \texttt{Field2Ds} or \texttt{Field3Ds}, respectively. Trying to create a \texttt{Region} using any other type is a compile time error.

The set of indices is also broken down into sets of contiguous blocks of at most MAXREGIONBLOCKSIZE indices. This allows loops to be parallelised with OpenMP. Iterating using a “block region” may be more efficient, although it requires a bit more set up. The helper macro \texttt{BOUT\_FOR} is provided to simplify things.

Example

The indices that form a region can be defined manually:
Region<Ind3D>::RegionIndices indices {0, 2, 4, 8, 3};
Region<Ind3D> region(indices);

then iterated over using begin() and end()

Field3D f(0.0);
for (auto i = region.begin(); i < region.end(); i++) {
  f[i] = 1.0;
}

For the region constructed above the following would display 0, 2, 4, 8, 3

for (auto i = region.begin(); i < region.end(); i++) {
  output << i.ind << ",";
}

or the more convenient region for loop:

for (const auto &i : r) {
  f[i] = a[i] + b[i];
}

For performance the BOUT_FOR macro should allow OpenMP parallelisation and hardware vectorisation.

BOUT_FOR(i, region) {
  f[i] = a[i] + b[i];
}

If you wish to vectorise but can't use OpenMP then there is a serial version of the macro:

BoutReal max=0.;
BOUT_FOR_SERIAL(i, region) {
  max = f[i] > max ? f[i] : max;
}

Public Types

using data_type = T
using RegionIndices = std::vector<T>
  Indices to iterate over.

using ContiguousBlock = std::pair<T, T>
  Start and end of contiguous region. This describes a range [block.first,block.second)

using ContiguousBlocks = std::vector<ContiguousBlock>
  Collection of contiguous regions.

using value_type = T
using reference = value_type&
using const_reference = const value_type&
using \texttt{size\_type} = \texttt{typename RegionIndices::size\_type}
using \texttt{iterator} = \texttt{typename RegionIndices::iterator}
using \texttt{const\_iterator} = \texttt{typename RegionIndices::const\_iterator}

\textbf{Public Functions}

\texttt{Region()} = default

\texttt{inline Region(int xstart, int xend, int ystart, int yend, int zstart, int zend, int ny, int nz, int maxregionblocksize = 64)}

\texttt{inline Region(RegionIndices \&indices, int maxregionblocksize = 64)}

\texttt{inline Region(ContiguousBlocks \&blocks)}

\texttt{\textasciitilde Region() = default}
Destroyer.

\texttt{inline RegionIndices::iterator begin()}
Exposé the iterator over indices for use in range-based for-loops or with STL algorithms, etc.

Note that if the indices are altered using these iterators, the blocks may become out of sync and will need to manually updated

\texttt{inline RegionIndices::const\_iterator begin() const}

\texttt{inline RegionIndices::const\_iterator cbegin() const}

\texttt{inline RegionIndices::iterator end()}

\texttt{inline RegionIndices::const\_iterator end() const}

\texttt{inline RegionIndices::const\_iterator cend() const}

\texttt{inline const ContiguousBlocks \&getBlocks() const}

\texttt{inline const RegionIndices \&getIndices() const}

\texttt{inline void setIndices(RegionIndices \&indicesIn, int maxregionblocksize = 64)}
\texttt{Set the indices and ensure blocks updated.}

\texttt{inline void setBlocks(ContiguousBlocks \&blocksIn)}
\texttt{Set the blocks and ensure indices updated.}

\texttt{inline Region<T> asSorted()}
Return a new \texttt{Region} that has the same indices as this one but ensures the indices are sorted.

\texttt{inline Region<T> \&sort()}
Sort this \texttt{Region} in place.
inline Region<T> asUnique()
    Return a new Region that has the same indices as this one but ensures the indices are sorted and unique (i.e. not duplicate indices). Note this sorts the input.

inline Region<T> &unique()
    Make this Region unique in-place.

inline Region<T> &mask(const Region<T> &maskRegion)
    Return a new region equivalent to *this but with indices contained in mask Region removed

inline Region<T> &operator+=(const Region<T> &rhs)
    Accumulate operator.

inline Region<T> &offset(int offset)
    Offset all indices by fixed value.

inline Region<T> &periodicShift(int shift, int period)
    Shift all indices by fixed value but wrap around on a given period. This is intended to act in a similar way as numpy’s roll. It should be helpful to calculate offset arrays for periodic directions (e.g. z). For example for shift = 1, period = mesh->LocalNz we would find the zplus indices. For shift = mesh->LocalNy*mesh->LocalNz, period = mesh->LocalNx*mesh->LocalNy*mesh->LocalNz we find xplus indices.

inline unsigned int size() const
    Number of indices (possibly repeated)

inline RegionStats getStats() const
    Returns a RegionStats struct describing the region.

**Private Functions**

inline RegionIndices createRegionIndices(int xstart, int xend, int ystart, int yend, int zstart, int zend, int ny, int nz)
    Helper function to create a RegionIndices, given the start and end points in x, y, z, and the total y, z lengths

inline ContiguousBlocks getContiguousBlocks(int maxregionblocksize) const
    Returns a vector of all contiguous blocks contained in the passed region. Limits the maximum size of any contiguous block to maxBlockSize. A contiguous block is described by the inclusive start and the exclusive end of the contiguous block.

inline RegionIndices getRegionIndices()
    Constructs the vector of indices from the stored blocks information.

**Private Members**

RegionIndices indices
ContiguousBlocks blocks

int ny = -1
int nz = -1
K.2.193 File rk3-ssp.cxx

K.2.194 File rk3-ssp.hxx

class RK3SSP : public Solver

Public Functions

RK3SSP (Options *opt = nullptr)

inline ~RK3SSP()

virtual void setMaxTimestep (BoutReal dt) override
    Set a maximum internal timestep (only for explicit schemes)

inline virtual BoutReal getCurrentTimestep () override
    Return the current internal timestep.

virtual int init (int nout, BoutReal tstep) override
    Initialise the solver NOTE: nout and tstep should be passed to run, not init. Needed because of how the
    PETSc TS code works

virtual int run () override
    Run the solver, calling monitors nout times, at intervals of tstep. This function is called by solve(), and is
    specific to each solver type

    This should probably be protected, since it shouldn’t be called by users.

Private Functions

void take_step (BoutReal curtime, BoutReal dt, Array<BoutReal> &start, Array<BoutReal> &result)

Private Members

BoutReal max_timestep
int mxstep
Array<BoutReal> f
BoutReal out_timestep
int nsteps
BoutReal timestep
int nlocal
int neq
Array<BoutReal> u1
Array<BoutReal> u2
Array<BoutReal> u3
class RK4Solver : public Solver

public:
  RK4Solver(Options *options)
  virtual void resetInternalFields() override
  {
    Should wipe out internal field vector and reset from current field object data.
  }
  virtual void setMaxTimestep(BoutReal dt) override
  {
    Set a maximum internal timestep (only for explicit schemes)
  }
  inline virtual BoutReal getCurrentTimestep() override
  {
    Return the current internal timestep.
  }
  virtual int init(int nout, BoutReal tstep) override
  {
    Initialise the solver NOTE: nout and tstep should be passed to run, not init. Needed because of how the
    PETSc TS code works
  }
  virtual int run() override
  {
    Run the solver, calling monitors nout times, at intervals of tstep. This function is called by solve(), and is
    specific to each solver type
    This should probably be protected, since it shouldn’t be called by users.
  }

private:
  void take_step(BoutReal curtime, BoutReal dt, Array<BoutReal> &start, Array<BoutReal> &result)

private:
  BoutReal atol
  BoutReal rtol
  BoutReal max_timestep
  int mxstep
  Array<BoutReal> f0
  Array<BoutReal> f1
  Array<BoutReal> f2
  BoutReal out_timestep
  int nsteps
BoutReal timestep
bool adaptive
int nlocal
int neq
Array<BoutReal> k1
Array<BoutReal> k2
Array<BoutReal> k3
Array<BoutReal> k4
Array<BoutReal> k5

K.2.197 File rk4simple.cxx
K.2.198 File rk4simple.hxx

class RK4SIMPLEScheme : public RKScheme

   Public Functions

   RK4SIMPLEScheme(Options *options)

   virtual BoutReal setOutputStates(const Array<BoutReal> &start, BoutReal dt, Array<BoutReal> &resultFollow)

K.2.199 File rkf34.cxx
K.2.200 File rkf34.hxx

class RKF34Scheme : public RKScheme

   Public Functions

   RKF34Scheme(Options *options)
K.2.201 File rkf45.cxx
K.2.202 File rkf45.hxx

class RKF45Scheme : public RKScheme

Public Functions

RKF45Scheme(Options *options)

K.2.203 File rkgeneric.cxx
K.2.204 File rkgeneric.hxx

class RKGenericSolver : public Solver

Public Functions

RKGenericSolver(Options *options)

~RKGenericSolver()

virtual void resetInternalFields() override
    Should wipe out internal field vector and reset from current field object data.

virtual void setMaxTimestep(BoutReal dt) override
    Set a maximum internal timestep (only for explicit schemes)

inline virtual BoutReal getCurrentTimestep() override
    Return the current internal timestep.

virtual int init(int nout, BoutReal tstep) override
    Initialise the solver NOTE: nout and tstep should be passed to run, not init. Needed because of how the
    PETSc TS code works

virtual int run() override
    Run the solver, calling monitors nout times, at intervals of tstep. This function is called by solve(), and is
    specific to each solver type

    This should probably be protected, since it shouldn’t be called by users.
Private Functions

\texttt{BoutReal take\_step(BoutReal timeIn, BoutReal dt, const Array\langle BoutReal \rangle &start, Array\langle BoutReal \rangle &resultFollow)}

Private Members

\texttt{Array\langle BoutReal \rangle f0}
\texttt{Array\langle BoutReal \rangle f2}
\texttt{Array\langle BoutReal \rangle tmpState}
\texttt{BoutReal atol}
\texttt{BoutReal rtol}
\texttt{BoutReal max\_timestep}
\texttt{int mxstep}
\texttt{bool adaptive}
\texttt{BoutReal out\_timestep}
\texttt{int nsteps}
\texttt{BoutReal timestep}
\texttt{int nlocal}
\texttt{int neq}
\texttt{RKScheme *scheme}

K.2.205 File rkscheme.cxx

K.2.206 File rkscheme.hxx

Defines

RKSchemeType
RKSCHEME_RKF45
RKSCHEME_CASHKARP
RKSCHEME_RK4
RKSCHEME_RKF34

class RKScheme
Subclassed by CASHKARPScheme, RK4SIMPLEScheme, RKF34Scheme, RKF45Scheme
Public Functions

RKScheme(Options *opts = nullptr)

virtual ~RKScheme() = default

void init(int nlocalIn, int neqIn, bool adaptiveIn, BoutReal atolIn, BoutReal rtolIn, Options *options = nullptr)

BoutReal setCurTime(BoutReal timeIn, BoutReal dt, int curStage)

virtual void setCurState(const Array<BoutReal> &start, Array<BoutReal> &out, int curStage, BoutReal dt)

virtual BoutReal setOutputStates(const Array<BoutReal> &start, BoutReal dt, Array<BoutReal> &resultFollow)

virtual BoutReal updateTimestep(BoutReal dt, BoutReal err)

inline virtual std::string getType()

inline int getStageCount()

inline int getNumOrders()

Public Members

Matrix<BoutReal> steps

Protected Functions

virtual BoutReal getErr(Array<BoutReal> &solA, Array<BoutReal> &solB)

virtual void constructOutput(const Array<BoutReal> &start, BoutReal dt, int index, Array<BoutReal> &sol)

virtual void constructOutputs(const Array<BoutReal> &start, BoutReal dt, int indexFollow, int indexAlt, Array<BoutReal> &solFollow, Array<BoutReal> &solAlt)
**Protected Attributes**

bool followHighOrder
std::string label
int numStages
int numOrders
int order
Matrix&lt;BoutReal&gt; stageCoeffs
Matrix&lt;BoutReal&gt; resultCoeffs
Array&lt;BoutReal&gt; timeCoeffs
Array&lt;BoutReal&gt; resultAlt
int nlocal
int neq
BoutReal atol
BoutReal rtol
bool adaptive
BoutReal dtfac

**Private Functions**

void verifyCoeffs()

void printButcherTableau()

void zeroSteps()
Public Functions

    inline RKSchemeType getDefaultRKSchemeType()

    RKScheme *createRKScheme(Options *opts = nullptr)

    RKScheme *createRKScheme(RKSchemeType&, Options *opts = nullptr)

Public Static Functions

    static RKSchemeFactory *getInstance()
    Return a pointer to the only instance.

Private Functions

    inline RKSchemeFactory()

Private Static Attributes

    static RKSchemeFactory *instance = nullptr
    The only instance of this class (Singleton)

K.2.209 File rvec.hxx

Defines

    __RVEC_H__

Typedefs

    using rvec = std::vector<BoutReal>

K.2.210 File scorepwrapper.hxx

Defines

    SCOREP_LVL
    SCOREP_BASE_CALL(...)  
    Instrument a region/function with scorep
    The scorep call is identical for all levels, so just define it here. If we don’t have scorep support then just define a null function
This is always defined.

K.2.211 File serial_band.cxx

K.2.212 File serial_band.hxx

class LaplaceSerialBand : public Laplacian

Public Functions

LaplaceSerialBand(Options *opt = nullptr, const CELL_LOC = CELL_CENTRE, Mesh *mesh_in = nullptr)

inline ~LaplaceSerialBand()

inline virtual void setCoefA(const Field2D &val) override
  Set coefficients for inversion. Re-builds matrices if necessary.

inline virtual void setCoefC(const Field2D &val) override

inline virtual void setCoefD(const Field2D &val) override

inline virtual void setCoefEx(const Field2D &val) override

inline virtual void setCoefEz(const Field2D &val) override

virtual FieldPerp solve(const FieldPerp &b) override

virtual FieldPerp solve(const FieldPerp &b, const FieldPerp &x0) override
Private Members

*Field2D Acoef*
*Field2D Ccoef*
*Field2D Dcoef*
*Matrix<
dcomplex> bk*
*Matrix<
dcomplex> xk*
*Matrix<
dcomplex> A*
*Array<dcomplex> bkid*
*Array<dcomplex> xkid*

K.2.213 File serial_tri.cxx

K.2.214 File serial_tri.hxx

class LaplaceSerialTri : public Laplacian

Public Functions

LaplaceSerialTri(*Options *opt = nullptr, const CELL_LOC loc = CELL_CENTRE, Mesh *mesh_in = nullptr*)

inline ~LaplaceSerialTri()

inline virtual void setCoefA(const Field2D &val) override
  *Set* coefficients for inversion. Re-builds matrices if necessary.

inline virtual void setCoefC(const Field2D &val) override

inline virtual void setCoefD(const Field2D &val) override

inline virtual void setCoefEx(const Field2D &val) override

inline virtual void setCoefEz(const Field2D &val) override

virtual FieldPerp solve(const FieldPerp &b) override

virtual FieldPerp solve(const FieldPerp &b, const FieldPerp &x0) override
  *Solve* Ax=b for x given b
  This function will
  i. Take the fourier transform of the y-slice given in the input
  ii. For each fourier mode a) *Set* up the tridiagonal matrix b) Call the solver which inverts the matrix
    Ax_mode = b_mode
iii. Collect all the modes in a 2D array
iv. Back transform the y-slice

Input:

Parameters

• \( b \) – [in] A 2D variable that will be fourier decomposed, each fourier mode of this variable is going to be the right hand side of the equation \( Ax = b \)

• \( x0 \) – [in] Variable used to set BC (if the right flags are set, see the user manual)

Returns The inverted variable.

Private Members

\( Field2D \, A \)
\( Field2D \, C \)
\( Field2D \, D \)

K.2.215 File shiftedmetric.cxx

K.2.216 File shoot_laplace.cxx

Laplacian solver using shooting method.

CHANGELOG

Feb 2014: Ben Dudson benjamin.dudson@york.ac.uk
• Initial version

Copyright 2014 B.D.Dudson
Contact: Ben Dudson, benjamin.dudson@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.
K.2.217 File shoot_laplace.hxx

class LaplaceShoot : public Laplacian

Public Functions

LaplaceShoot(Options *opt = nullptr, const CELL_LOC = CELL_CENTRE, Mesh *mesh_in = nullptr)

inline ~LaplaceShoot()

inline virtual void setCoefA(const Field2D &val) override
    Set coefficients for inversion. Re-builds matrices if necessary.
inline virtual void setCoefC(const Field2D &val) override
inline virtual void setCoefD(const Field2D &val) override
inline virtual void setCoefEx(const Field2D &val) override
inline virtual void setCoefEz(const Field2D &val) override

virtual FieldPerp solve(const FieldPerp &b) override
inline virtual FieldPerp solve(const FieldPerp &b, const FieldPerp &x0) override

Private Members

Field2D Acoef
Field2D Ccoef
Field2D Dcoef
int nmode
Array<dcomplex> km
Array<dcomplex> kc
Array<dcomplex> kp
Array<dcomplex> rhsk
Array<BoutReal> buffer
K.2.218 File slepc.cxx

Functions

std::string formatEig(BoutReal reEig, BoutReal imEig)

PetscErrorCode advanceStepWrapper(Mat matOperator, Vec inData, Vec outData)

PetscErrorCode compareEigsWrapper(PetscScalar ar, PetscScalar ai, PetscScalar br, PetscScalar bi, PetscInt *res, void *ctx)

PetscErrorCode monitorWrapper(EPS eps, PetscInt its, PetscInt nconv, PetscScalar *eigr, PetscScalar *eigi, PetscReal *errest, PetscInt nest, void *mctx)

PetscErrorCode stApplyWrapper(ST st, Vec vecIn, Vec vecOut)

PetscErrorCode stBackTransformWrapper(ST st, PetscInt nEig, PetscScalar *eigr, PetscScalar *eigi)

K.2.219 File slepc.hxx

Defines

OPT_SIZE

SOLVERSLEPCSELF

class SlepcSolver : public Solver

Public Functions

SlepcSolver(Options *options)

~SlepcSolver()

int advanceStep(Mat &matOperator, Vec &inData, Vec &outData)

int compareEigs(PetscScalar ar, PetscScalar ai, PetscScalar br, PetscScalar bi)

void monitor(PetscInt its, PetscInt nconv, PetscScalar eig[], PetscScalar eigi[], PetscReal errest[], PetscInt nest)

virtual int init(int NOUT, BoutReal TIMESTEP) override

Initialise the solver NOTE: nout and tstep should be passed to run, not init. Needed because of how the PETSc TS code works
virtual int run() override
Run the solver, calling monitors nout times, at intervals of tstep. This function is called by solve(), and is specific to each solver type
This should probably be protected, since it shouldn't be called by users.

inline virtual void setModel(PhysicsModel *model) override
OVERRIDE Here we override all other virtual functions in order to pass through control to the actual solver (advanceSolver) This is only required if allow use of additional solver

inline virtual void setRHS(rhsfunc f) override
Set the RHS function.

inline virtual void add(Field2D &v, const std::string &name, const std::string &description = "") override
Add a variable to be solved. This must be done in the initialisation stage, before the simulation starts.

inline virtual void add(Field3D &v, const std::string &name, const std::string &description = "") override

inline virtual void add(Vector2D &v, const std::string &name, const std::string &description = "") override

inline virtual void add(Vector3D &v, const std::string &name, const std::string &description = "") override

inline virtual void setJacobian(Jacobian j) override
Specify a Jacobian (optional)

inline virtual void setSplitOperator(rhsfunc fC, rhsfunc fD) override
Split operator solves.

inline virtual bool constraints() override
Returns true if constraints available.

inline virtual void constraint(Field2D &v, Field2D &C_v, std::string name) override
Add constraint functions (optional). These link a variable v to a control parameter C_v such that v is adjusted to keep C_v = 0.

inline virtual void constraint(Field3D &v, Field3D &C_v, std::string name) override

inline virtual void constraint(Vector2D &v, Vector2D &C_v, std::string name) override

inline virtual void constraint(Vector3D &v, Vector3D &C_v, std::string name) override

inline virtual int n2Dvars() const override
Number of 2D variables. Vectors count as 3.

inline virtual int n3Dvars() const override
Number of 3D variables. Vectors count as 3.

inline virtual voidsetMaxTimestep(BoutReal dt) override
Set a maximum internal timestep (only for explicit schemes)

inline virtual BoutReal getCurrentTimestep() override
Return the current internal timestep.

void slepcToBout(PetscScalar &reEigIn, PetscScalar &imEigIn, BoutReal &reEigOut, BoutReal &imEigOut, bool force = false)
Public Members

int compareState
Mat shellMat

Private Functions

void vecToFields(Vec &inVec)

void fieldsToVec(Vec &outVec)

void createShellMat()

void createEPS()

void analyseResults()

void boutToSlepc(BoutReal &reEigIn, BoutReal &imEigIn, PetscScalar &reEigOut, PetscScalar &imEigOut, bool force = false)

Private Members

MPI_Comm comm
EPS eps
ST st
PetscBool stIsShell
Solver *advanceSolver
SlepcLib slib
bool ddtMode
bool selfSolve
bool eigenValOnly
Array<BoutReal> f0
Array<BoutReal> f1
int nout
BoutReal tstep
int nEig
int maxIt
int mpd
PetscReal tol
PetscReal target
BoutReal targRe
BoutReal targIm
bool userWhich
bool useInitial
bool debugMonitor
PetscInt localSize

K.2.220  File sleplib.cxx

K.2.221  File sleplib.hxx

class SlepcLib

Public Functions

SlepcLib()
~SlepcLib()

Public Static Functions

static inline void setArgs(int &, char **&v)

static void cleanup()

Private Static Attributes

static int count = 0
static char help[] = "BOUT++: Uses finite difference methods to solve plasma fluid problems in curvilinear coordinates"
static int *pargc = nullptr
static char ***pargv = nullptr
static PetscLogEvent USER_EVENT = 0
**K.2.222 File smoothing.cxx**

**Functions**

```cpp
class Field3D
{
public:
    Field3D smooth_x(const Field3D &f)
    {
        // Smooth in X using simple 1-2-1 filter.
    }

    Field3D smooth_y(const Field3D &f)
    {
        // Smooth in Y using 1-2-1 filter.
    }

    Field2D averageX(const Field2D &f)
    {
        // Average over X.
        // Issues
        // Assumes every processor has the same domain shape
        // Will only work if X communicator is constant in Y so no processor/branch cuts in X
    }

    Field3D averageX(const Field3D &f)
    {
        // Issues
        // Creates static arrays
        // Not thread safe
        // Assumes every processor has the same domain shape
        // Will only work if X communicator is constant in Y so no processor/branch cuts in X
    }

    Field2D averageY(const Field2D &f)
    {
        // Average over Y
        // Issues
        // Important: Only works if there are no branch cuts
        // Assumes every processor has the same domain shape
    }

    Field3D averageY(const Field3D &f)
    {
        // Issues
        // Important: Only works if there are no branch cuts
        // Creates static arrays
        // Not thread safe
        // Assumes every processor has the same domain shape
    }

    BoutReal Average_XY(const Field2D &var)
    {
        // Volume integral of Field2D variable Developed by T. Rhee and S. S. Kim
        // Issues
        // Assumes every processor has the same domain shape
        // Will only work if X communicator is constant in Y so no processor/branch cuts in X
    }

    BoutReal Vol_Integral(const Field2D &var)
    {
        // Volume integral of Field2D variable which uses Average_XY
    }

    Field3D smoothXY(const Field3D &f)
    {
        // Smooth using a stencil in X and Y.
    }
};
```
void \texttt{nl\_filter}(\texttt{rvec} &f, \texttt{BoutReal} w)

\texttt{const Field3D nl\_filter\_x}(\texttt{const Field3D} &f, \texttt{BoutReal} w)
Nonlinear filtering to remove grid-scale noise in X.
\begin{quote}
From a paper:
W.Shyy et. al. JCP 102 (1) September 1992 page 49
“On the Suppression of Numerical Oscillations Using a Non-Linear Filter”
\end{quote}

\texttt{const Field3D nl\_filter\_y}(\texttt{const Field3D} &f, \texttt{BoutReal} w)
Nonlinear filtering to remove grid-scale noise in Y.
\begin{quote}
From a paper:
W.Shyy et. al. JCP 102 (1) September 1992 page 49
“On the Suppression of Numerical Oscillations Using a Non-Linear Filter”
\end{quote}

\texttt{const Field3D nl\_filter\_z}(\texttt{const Field3D} &f, \texttt{BoutReal} w)
Nonlinear filtering to remove grid-scale noise in Z.
\begin{quote}
From a paper:
W.Shyy et. al. JCP 102 (1) September 1992 page 49
“On the Suppression of Numerical Oscillations Using a Non-Linear Filter”
\end{quote}

\texttt{const Field3D nl\_filter}(\texttt{const Field3D} &f, \texttt{BoutReal} w)
Nonlinear filtering to remove grid-scale noise in X,Y and Z.
\begin{quote}
From a paper:
W.Shyy et. al. JCP 102 (1) September 1992 page 49
“On the Suppression of Numerical Oscillations Using a Non-Linear Filter”
\end{quote}

\textbf{K.2.223 File smoothing.hxx}

\textbf{Functions}

\texttt{const Field3D smooth\_x}(\texttt{const Field3D} &f)
Smooth in X using simple 1-2-1 filter.

\texttt{const Field3D smooth\_y}(\texttt{const Field3D} &f)
Smooth in Y using 1-2-1 filter.

\texttt{const Field3D smooth\_XY}(\texttt{const Field3D} &f)
Smooth using a stencil in X and Y.

\texttt{const Field2D average\_Y}(\texttt{const Field2D} &f)
Average over Y

\textbf{Issues}

Important: Only works if there are no branch cuts
Assumes every processor has the same domain shape

\texttt{const Field3D average\_Y}(\texttt{const Field3D} &f)
Average in Y

\textbf{Issues}
Important: Only works if there are no branch cuts

Creates static arrays

Not thread safe

Assumes every processor has the same domain shape

```cpp
const Field2D averageX(const Field2D &f)
    Average over X.
```

**Issues**

Assumes every processor has the same domain shape

Will only work if X communicator is constant in Y so no processor/branch cuts in X

```cpp
const Field3D averageX(const Field3D &f)
```

**Issues**

Creates static arrays

Not thread safe

Assumes every processor has the same domain shape

Will only work if X communicator is constant in Y so no processor/branch cuts in X

```cpp
BoutReal Average_XY(const Field2D &var)
    Volume integral of Field2D variable Developed by T. Rhee and S. S. Kim
```

**Issues**

Assumes every processor has the same domain shape

Will only work if X communicator is constant in Y so no processor/branch cuts in X

```cpp
BoutReal Vol_Integral(const Field2D &var)
    Volume integral of Field2D variable which uses Average_XY
```

```cpp
const Field3D nl_filter_x(const Field3D &f, BoutReal w = 1.0)
    Nonlinear filtering to remove grid-scale noise in X.
```

From a paper:

W.Shyy et. al. JCP 102 (1) September 1992 page 49

“On the Suppression of Numerical Oscillations Using a Non-Linear Filter”

```cpp
const Field3D nl_filter_y(const Field3D &f, BoutReal w = 1.0)
    Nonlinear filtering to remove grid-scale noise in Y.
```

From a paper:

W.Shyy et. al. JCP 102 (1) September 1992 page 49

“On the Suppression of Numerical Oscillations Using a Non-Linear Filter”

```cpp
const Field3D nl_filter_z(const Field3D &f, BoutReal w = 1.0)
    Nonlinear filtering to remove grid-scale noise in Z.
```

From a paper:

W.Shyy et. al. JCP 102 (1) September 1992 page 49

“On the Suppression of Numerical Oscillations Using a Non-Linear Filter”
const Field3D nl_filter(const Field3D &f, BoutReal w = 1.0)
Nonlinear filtering to remove grid-scale noise in X, Y and Z.
From a paper:
W.Shyy et. al. JCP 102 (1) September 1992 page 49
“On the Suppression of Numerical Oscillations Using a Non-Linear Filter”

K.2.224 File snb.cxx
	namespace bout
	SNB model

K.2.225 File snb.hxx
	namespace bout
	SNB model

class HeatFluxSNB
#include <snb.hxx>
Calculate heat flux using the Shurtz-Nicolai-Busquet (SNB) model
Useful references:
Braginskii equations by R.Fitzpatrick: http://farside.ph.utexas.edu/teaching/plasma/Plasmahtml/node35.html
Shurtz, Nicolai and Busquet 2000: https://doi.org/10.1063/1.1289512

Public Functions

inline HeatFluxSNB()
Construct using the options in the “snb” section.
inline explicit HeatFluxSNB(Options &options)
Construct using options in given section.
~HeatFluxSNB() = default

HeatFluxSNB(HeatFluxSNB&&) = default

HeatFluxSNB &operator=(HeatFluxSNB&&) = default

HeatFluxSNB(const HeatFluxSNB&) = delete

HeatFluxSNB &operator=(const HeatFluxSNB&) = delete

Field3D divHeatFlux(const Field3D &Te, const Field3D &Ne, Field3D *Div_Q_SH_out = nullptr)
Calculate divergence of heat flux Te: Electron temperature in eV Ne: Electron density in m^-3
Div_Q_SH_out : An optional output field to store the Spitzer-Harm heat flux
Returns the divergence of heat flux in units of eV per cubic meter per second -> multiply by \( e = 1.602 \times 10^{-19} \) to get Watts per cubic meter.

**Private Functions**

inline `BoutReal int_beta4_exp(BoutReal beta)`
Indefinite integral of \( \beta^4 \exp(-\beta) \) with constant set to zero

inline `BoutReal groupWeight(BoutReal beta_min, BoutReal beta_max)`
\( \frac{1}{24} \) * Integral of \( \beta^4 \exp(-\beta) \) from \( \beta_{\text{min}} \) to \( \beta_{\text{max}} \)

**Private Members**

`std::unique_ptr<InvertPar> invertpar = nullptr`  
Parallel inversion of tridiagonal matrices.

`BoutReal Z = {1}`  
Average ion charge (1 = Hydrogen)

`BoutReal r = {2}`  
Electron-electron mean free path scaling factor.

`BoutReal beta_max = {10.0}`  
Maximum energy group to consider (multiple of \( eT \))

`int ngroups = {40}`  
Number of energy groups.

**K.2.226 File snes.cxx**

**Functions**

static `PetscErrorCode FormFunction(SNES snes, Vec x, Vec f, void *ctx)`

static `PetscErrorCode FormFunctionForDifferencing(void *ctx, Vec x, Vec f)`  
PETSce callback function for forming Jacobian

This function can be a linearised form of FormFunction

static `PetscErrorCode FormFunctionForColoring(SNES snes, Vec x, Vec f, void *ctx)`  
SNES callback for forming Jacobian with coloring

This can be a linearised and simplified form of FormFunction

static `PetscErrorCode snesPCapply(PC pc, Vec x, Vec y)`
K.2.227 File snes.hxx

Functions

BOUT_ENUM_CLASS (BoutSnesEquationForm, pseudo_transient, rearranged_backward_euler, backward_euler, direct_newton)

class SNESolver : public Solver
#include <snes.hxx> Uses PETSc’s SNES interface to find a steady state solution to a nonlinear ODE by integrating in time with Backward Euler

Public Functions

explicit SNESolver (Options *opt = nullptr)

inline ~SNESolver ()

virtual int init (int nout, BoutReal tstep) override
Initialise solver. Must be called once and only once

Parameters

• nout – [in] Number of outputs
• tstep – [in] Time between outputs. NB: Not internal timestep

virtual int run () override
Run the simulation.

PetscErrorCode snes_function (Vec x, Vec f)
Nonlinear function.

Nonlinear function. This is called by PETSc SNES object via a static C-style function. For implicit time integration this function calculates:

\[
f = (x - \gamma^*G(x)) - rhs
\]

Parameters

• x – [in] The state vector
• f – [out] The vector for the result f(x)

PetscErrorCode precon (Vec x, Vec f)
Preconditioner. Called by PCapply via a C-style static function.

Parameters

• x – [in] The vector to be operated on
• f – [out] The result of the operation
**Private Members**

* BoutReal `timestep`  
  Internal timestep.

* BoutReal `dt`  
  Current timestep used in snes_function.

* BoutReal `dt_min_reset`  
  If dt falls below this, reset solve.

* int `lower_its`  
  Limits on iterations for timestep adjustment.

* int `upper_its`  
  Limits on iterations for timestep adjustment.

* BoutReal `out_timestep`  
  Output timestep.

* int `nsteps`  
  Number of steps to take.

* bool `diagnose`  
  Output additional diagnostics.

* int `nlocal`  
  Number of variables on local processor.

* int `neq`  
  Number of variables in total.

* BoutSnesEquationForm `equation_form`  
  Form of the equation to solve.

* PetscLib `lib`  
  Handles initialising, finalising PETSc.

* Vec `snes_f`  
  Used by SNES to store function.

* Vec `snes_x`  
  Result of SNES.

* Vec `x0`  
  Solution at start of current timestep.

* Vec `delta_x`  
  Change in solution.

* bool `predictor`  
  Use linear predictor?
Vec x1
    Previous solution.

BoutReal time1 = {-1.0}
    Time of previous solution.

SNES snes
    SNES context.

Mat Jmf
    Matrix-free Jacobian.

MatFDColoring fdcoloring
    Matrix coloring context, used for finite difference Jacobian evaluation

K.2.228 File solver.cxx

Functions

template<class T>
int local_N_sum(int value, const Solver::VarStr<T>& f)
    Helper function for getLocalN: return the number of points to evolve in f, plus the accumulator value
    If f.evolve_bndry, includes the boundary (NB: not guard!) points
    FIXME: This could be a lambda local to getLocalN with an auto argument in C++14

Variables

bool user_requested_exit

K.2.229 File solver.hxx

Defines

BOUT_NO_USING_NAMESPACE_BOUTGLOBALS

Typedefs

using rhsfunc = int (*)(BoutReal)
    RHS function pointer.

using PhysicsPrecon = int (*)(BoutReal t, BoutReal gamma, BoutReal delta)
    User-supplied preconditioner function.

using Jacobian = int (*)(BoutReal t)
    User-supplied Jacobian function.
using `TimestepMonitorFunc` = int (*)(Solver *solver, BoutReal simtime, BoutReal lastdt)
Solution monitor, called each timestep.

using `SolverType` = std::string

** Enums **

enum `SOLVER_VAR_OP`
Values:
- enumerator `LOAD_VARS`
- enumerator `LOAD_DERIVS`
- enumerator `SET_ID`
- enumerator `SAVE_VARS`
- enumerator `SAVE_DERIVS`

enum `MonitorPosition`
A type to set where in the list monitors are added.
Values:
- enumerator `BACK`
- enumerator `FRONT`

** Variables **

constexpr auto `SOLVERCVODE` = "cvode"
constexpr auto `SOLVERPVODE` = "pvode"
constexpr auto `SOLVERIDA` = "ida"
constexpr auto `SOLVERPETSC` = "petsc"
constexpr auto `SOLVERSLEPC` = "slepc"
constexpr auto `SOLVERKARNIADAKIS` = "karniadakis"
constexpr auto `SOLVERRK4` = "rk4"
constexpr auto `SOLVEREULER` = "euler"
constexpr auto `SOLVERRK3SSP` = "rk3ssp"
constexpr auto `SOLVERPOWER` = "power"
constexpr auto `SOLVERARKODE` = "arkode"
constexpr auto `SOLVERIMEXBDF2` = "imexbdf2"
constexpr auto `SOLVERSNESS` = "snes"
constexpr auto `SOLVERRKGENERIC` = "rkgeneric"
class **Solver**

```
#include <solver.hxx>
```

Interface to integrators, mainly for time integration

**Creation**

*Solver* is a base class and can’t be created directly:

```
Solver *solver = Solver(); // Error
```

Instead, use the **create()** static function:

```
Solver *solver = Solver::create(); // ok
```

By default this will use the options in the “solver” section of the options, equivalent to:

```
Options *opts = Options::getRoot()->getSection("solver");
Solver *solver = Solver::create(opts);
```

To use a different set of options, for example if there are multiple solvers, use a different option section:

```
Options *opts = Options::getRoot()->getSection("anothersolver");
Solver *anothersolver = Solver::create(opts);
```

**Problem specification**

The equations to be solved are specified in a *PhysicsModel* object

```
class MyProblem : public PhysicsModel {
    protected:
        // This function called once at beginning
        int init(bool restarting) {
            SOLVE_FOR(f); // Specify variables to solve
            // Set f to initial value if needed
            // otherwise read from input options
            return 0;
        }

        // This function called to evaluate time derivatives
        int rhs(BoutReal t) {
            ddt(f) = 1.0; // Calculate time derivatives
            return 0;
        }

    private:
        Field3D f; // A variable to evolve
}
```

The **init()** and **rhs()** functions must be defined, but there are other functions which can be defined. See *PhysicsModel* documentation for details.

Create an object, then add to the solver:

```
MyProblem *prob = MyProblem();
solver->setModel(prob);
```

**Running simulation**

To run a calculation
This will use NOUT and TIMESTEP in the solver options (specified during creation). If these are not present then the global options will be used.

To specify NOUT and TIMESTEP, pass the values to solve:

```cpp
solver->solve(NOUT, TIMESTEP);
```

virtual void add(Field3D &v, const std::string &name, const std::string &description = "")

virtual void add(Vector2D &v, const std::string &name, const std::string &description = "")

virtual void add(Vector3D &v, const std::string &name, const std::string &description = "")

inline virtual bool constraints()
    Returns true if constraints available.

virtual void constraint(Field2D &v, Field2D &C_v, std::string name)
    Add constraint functions (optional). These link a variable v to a control parameter C_v such that v is adjusted to keep C_v = 0.

virtual void constraint(Field3D &v, Field3D &C_v, std::string name)

virtual void constraint(Vector2D &v, Vector2D &C_v, std::string name)

virtual void constraint(Vector3D &v, Vector3D &C_v, std::string name)

inline virtual void setMaxTimestep(BoutReal dt)
    Set a maximum internal timestep (only for explicit schemes)

inline virtual BoutReal getCurrentTimestep()
    Return the current internal timestep.

int solve(int nout = -1, BoutReal dt = 0.0)
    Start the solver. By default solve() uses options to determine the number of steps and the output timestep. If nout and dt are specified here then the options are not used

    Parameters

    • nout – [in] Number of output timesteps
    • dt – [in] The time between outputs

virtual int init(int nout, BoutReal tstep)
    Initialise the solver NOTE: nout and tstep should be passed to run, not init. Needed because of how the PETSc TS code works

virtual int run() = 0
    Run the solver, calling monitors nout times, at intervals of tstep. This function is called by solve(), and is specific to each solver type
    This should probably be protected, since it shouldn’t be called by users.

inline virtual void resetInternalFields()
    Should wipe out internal field vector and reset from current field object data.

inline virtual int n2Dvars() const
    Number of 2D variables. Vectors count as 3.

inline virtual int n3Dvars() const
    Number of 3D variables. Vectors count as 3.

int resetRHSCounter()
    Get and reset the number of calls to the RHS function.
int resetRHSCounter_e()
   Same but for explicit timestep counter - for IMEX.

int resetRHSCounter_i()
   Same but fur implicit timestep counter - for IMEX.

inline bool splitOperator()
   Test if this solver supports split operators (e.g. implicit/explicit)

virtual void outputVars(Datafile &outputfile, bool save_repeat = true)
   Add evolving variables to output (dump) file or restart file

   Parameters

   • outputfile – [inout] The file to add variable to
   • save_repeat – [in] If true, add variables with time dimension

std::string getRunID() const
   A unique identifier for this run. Throws if the identifier hasn’t been set yet.

std::string getRunRestartFrom() const
   The run from which this was restarted. Throws if the identifier hasn’t been set yet.

Public Members

bool canReset = {false}

Public Static Functions

static Solver *create(Options *opts = nullptr)
   Create a Solver object. This uses the “type” option in the given Option section to determine which solver type to create.

static Solver *create(const SolverType &type, Options *opts = nullptr)
   Create a Solver object, specifying the type.

static inline void setArgs(int &c, char **&v)
   Pass the command-line arguments. This static function is called by BoutInitialise, and puts references into protected variables. These may then be used by Solvers to control behavior.

Public Static Attributes

static constexpr MonitorPosition BACK = MonitorPosition::BACK

static constexpr MonitorPosition FRONT = MonitorPosition::FRONT
Protected Functions

int getLocalN()
   Calculate the number of evolving variables on this processor.

template<class T>
inline bool contains(const std::vector<VarStr<T>> &vars, const std::string &name)
   Does vars contain a field with name?

inline void add_int_diagnostic(int &i, const std::string &name, const std::string &description = "")

inline void add_BoutReal_diagnostic(BoutReal &r, const std::string &name, const std::string &description = "")

int run_rhs(BoutReal t)
   Run the user’s RHS function.

int run_convective(BoutReal t)
   Calculate only the convective parts.
   NOTE: This calls add_mms_sources.

int run_diffusive(BoutReal t, bool linear = true)
   Calculate only the diffusive parts.

int call_monitors(BoutReal simtime, int iter, int NOUT)
   Calls all monitor functions

   There are two important things to note about how iter is passed along to each monitor:
   • The solvers all start their iteration numbering from zero, so the initial state is calculated at iter = -1
   • Secondly, iter is passed along to each monitor relative to that monitor’s period

   In practice, this means that each monitor is called like:

   ```
   monitor->call(solver, simulation_time,
   ((iter + 1) / monitor->period) - 1,
   NOUT / monitor->period);
   ```

   e.g. for a monitor with period 10, passing iter = 9 will result in it being called with a value of (9 + 1)/10 - 1 == 0

int call_timestep_monitors(BoutReal simtime, BoutReal lastdt)

bool hasPreconditioner()
   Do we have a user preconditioner?

inline bool have_user_precon()

int runPreconditioner(BoutReal time, BoutReal gamma, BoutReal delta)
   Run the user preconditioner.

inline int run_precon(BoutReal time, BoutReal gamma, BoutReal delta)

bool hasJacobian()
   Do we have a user Jacobian?
int runJacobian(BoutReal time)
    Run the user Jacobian.

void load_vars(BoutReal *udata)

void load_derivs(BoutReal *udata)

void save_vars(BoutReal *udata)

void save_derivs(BoutReal *dudata)

void set_id(BoutReal *udata)

Field3D globalIndex(int localStart)
    Returns a Field3D containing the global indices.

inline auto getMonitors() const -> const std::list<Monitor*>&
    Get the list of monitors.

Protected Attributes

Options *options = {nullptr}
    Settings to use during initialisation (set by constructor)

int NPES = {1}
    Number of processors.

int MYPE = {0}
    This processor’s index.

std::vector<VarStr<Field2D>> f2d
    Vectors of variables to evolve.

std::vector<VarStr<Field3D>> f3d

std::vector<VarStr<Vector2D>> v2d

std::vector<VarStr<Vector3D>> v3d

std::vector<VarStr<int>> diagnostic_int
    Vectors of diagnostic variables to save.

std::vector<VarStr<BoutReal>> diagnostic_BoutReal

bool has_constraints = {false}
    Can this solver handle constraints? Set to true if so.

bool initialised = {false}
    Has init been called yet?

BoutReal simtime = {0.0}
    Current simulation time.
int iteration = {0}
    Current iteration (output time-step) number.

bool monitor_timestep = {false}
    Should timesteps be monitored?

BoutReal max_dt = {-1.0}
    Maximum internal timestep.

Protected Static Attributes

static int *pargc = nullptr
    Number of command-line arguments.

static char ***pargv = nullptr
    Command-line arguments.

Private Functions

std::string createRunID() const
    Generate a random UUID (version 4) and broadcast it to all processors.

void add_mms_sources(BoutReal t)

void calculate_mms_error(BoutReal t)

void pre_rhs(BoutReal t)
    Should be run before user RHS is called.

void post_rhs(BoutReal t)
    Should be run after user RHS is called.

void loop_vars_op(Ind2D i2d, BoutReal *udata, int &p, SOLVER_VAR_OP op, bool bndry)
    Loading data from BOUT++ to/from solver.

    Perform an operation at a given Ind2D (jx,jy) location, moving data between BOUT++ and CVODE.

void loop_vars(BoutReal *udata, SOLVER_VAR_OP op)
    Loop over variables and domain. Used for all data operations for consistency.

bool varAdded(const std::string &name)
    Check if a variable has already been added.

BoutReal adjustMonitorPeriods(Monitor *monitor)
    (Possibly) adjust the periods of monitor, and the monitors timesteps, returning the new Solver timestep

void finaliseMonitorPeriods(int &NOUT, BoutReal &output_timestep)
    Fix all the monitor periods based on output_timestep, as well as adjusting NOUT and output_timestep to be consistent
**Private Members**

```cpp
std::string run_id = default_run_id
    Randomly generated run ID Initialise with 36 characters so the allocated array is the right size

std::string run_restart_from = "yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy"
    The run from which this was restarted.

int rhs_ncalls = {0}
    Number of calls to the RHS function.

int rhs_ncalls_e = {0}
    Number of calls to the explicit (convective) RHS function.

int rhs_ncalls_i = {0}
    Number of calls to the implicit (diffusive) RHS function.

int default_monitor_period = {1}
    Default sampling rate at which to call monitors - same as output to screen.

BoutReal internal_timestep = {-1}
    timestep - shouldn't be changed after init is called.

PhysicsModel *model = {nullptr}
    Physics model being evolved.

rhssfnc phys_run = {nullptr}
    The user's RHS function.

PhysicsPrecon prefunc = {nullptr}
    The user's preconditioner function.

Jacobian user_jacobian = {nullptr}
    The user's Jacobian function.

bool split_operator = {false}
    Is the physics model using separate convective (explicit) and diffusive (implicit) RHS functions?

rhssfnc phys_conv = {nullptr}
    Convective part (if split operator)

rhssfnc phys_diff = {nullptr}
    Diffusive part (if split operator)

bool is_nonsplit_model_diffusive = {true}
    Should non-split physics models be treated as diffusive?

bool mms = {false}
    Enable sources and solutions for Method of Manufactured Solutions.
```
bool mms_initialise = {false}
     Initialise variables to the manufactured solution.

std::list<Monitor*> monitors
     List of monitor functions.

std::list<TimestepMonitorFunc> timestep_monitors
     List of timestep monitor functions.

Private Static Attributes

static constexpr auto default_run_id = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
     Default value for run_id. Use ‘z’ because it is not a valid hex character, so this is an invalid UUID

Friends

template<class T> inline friend friend bool operator==(const VarStr<T> &var, const std::string &name)
     Does var represent field name?

template<class T> friend friend int local_N_sum (int value, const VarStr<T> &f)
     Helper function for getLocalN: return the number of points to evolve in f, plus the accumulator value
     If f.evolve_bndry, includes the boundary (NB: not guard!) points
     FIXME: This could be a lambda local to getLocalN with an auto argument in C++14

template<class T>
struct VarStr
     #include <solver.hxx> A structure to hold an evolving variable.

Public Members

bool constraint = {false}

T *var = {nullptr}
     Does F_var represent a constraint?

T *F_var = {nullptr}
     The evolving variable.

std::unique_ptr<T> MMS_err = {nullptr}
     The time derivative or constraint on var.

CELL_LOC location = {CELL_DEFAULT}
     Error for MMS.

bool covariant = {false}
     For fields and vector components.
bool evolve_bndry = {false}
For vectors.

std::string name
Are the boundary regions being evolved?

std::string description = {""}
Name of the variable.

K.2.230 File solverfactory.cxx
K.2.231 File solverfactory.hxx

Typedefs

using RegisterSolver = RegisterInFactory<Solver, DerivedType>
Simpler name for Factory registration helper class

Usage:

```
#include <bout/solverfactory.hxx>
namespace {
  RegisterSolver<MySolver> registersolvermine("mysolver");
}
```

class SolverFactory : public Factory<Solver, std::function<Solver*(Options*)>>
#include <solverfactory.hxx> Factory specialisation for Solvers

TODO: Inherits from Factory<> in order to keep the same interface during transition period. After next major version, replace whole class with:

```
using SolverFactory = Factory<Solver, SolverType, std::function<Solver*(Options*)>>;
```

Public Functions

Solver *createSolver(Options *options = nullptr)

inline Solver *createSolver(const SolverType &name)

inline Solver *createSolver(const SolverType &name, Options *options)
Public Static Functions

static SolverType getDefaultSolverType()
Return the name of the default Solver type.

static SolverFactory *getInstance()

Private Functions

SolverFactory() = default

Private Static Attributes

static SolverFactory *instance = nullptr

template<typename DerivedType>

class RegisterInFactory<Solver, DerivedType>
#include <solverfactory.hxx> Specialisation of Factory registration helper class.

Public Functions

inline RegisterInFactory(const std::string &name)

K.2.232 File sourcex.cxx

Functions

BoutReal TanH(BoutReal a)

const Field2D source_tanhx(const Field2D &f, BoutReal swidth, BoutReal slength)

const Field2D source_expx2(const Field2D &f, BoutReal swidth, BoutReal slength)

const Field3D sink_tanhx(const Field2D &f0, const Field3D &f, BoutReal swidth, BoutReal slength, bool BoutRealspace)

const Field3D mask_x(const Field3D &f, bool BoutRealspace)

const Field3D sink_tanhxl(const Field2D &f0, const Field3D &f, BoutReal swidth, BoutReal slength, bool BoutRealspace)

const Field3D sink_tanhxr(const Field2D &f0, const Field3D &f, BoutReal swidth, BoutReal slength, bool BoutRealspace)
const Field3D buff_x(const Field3D &f, bool BoutRealspace)

### K.2.233 File sourcex.hxx

#### Functions

const Field3D mask_x(const Field3D &f, bool BoutRealspace = true)

const Field2D source_tanhx(const Field2D &f, BoutReal swidth, BoutReal slength)

const Field2D source_expx2(const Field2D &f, BoutReal swidth, BoutReal slength)

const Field3D sink_tanhx(const Field2D &f0, const Field3D &f, BoutReal swidth, BoutReal slength, bool BoutRealspace = true)

const Field3D sink_tanhxl(const Field2D &f0, const Field3D &f, BoutReal swidth, BoutReal slength, bool BoutRealspace = true)

const Field3D sink_tanhxr(const Field2D &f0, const Field3D &f, BoutReal swidth, BoutReal slength, bool BoutRealspace = true)

const Field3D buff_x(const Field3D &f, bool BoutRealspace = true)

### K.2.234 File split-rk.cxx

### K.2.235 File split-rk.hxx

#### Defines

SPLITRK_HXX

class SplitRK : public Solver

##### Public Functions

inline explicit SplitRK(Options *opt = nullptr)

~SplitRK() = default

virtual int init(int nout, BoutReal tstep) override

Initialise the solver NOTE: nout and tstep should be passed to run, not init. Needed because of how the PETSc TS code works
virtual int run() override  
    Run the solver, calling monitors nout times, at intervals of tstep. This function is called by solve(), and is specific to each solver type

    This should probably be protected, since it shouldn't be called by users.

**Private Functions**

void take_step(BoutReal curtime, BoutReal dt, Array<BoutReal> &start, Array<BoutReal> &result)
    Take a combined step Uses 2nd order Strang splitting
    Note: start and result can be the same

void take_diffusion_step(BoutReal curtime, BoutReal dt, Array<BoutReal> &start, Array<BoutReal> &result)
    Take a step of the diffusion terms Uses the Runge-Kutta-Legendre 2nd order method
    Note: start and result can be the same

void take_advection_step(BoutReal curtime, BoutReal dt, Array<BoutReal> &start, Array<BoutReal> &result)
    Take a step of the advection terms Uses the Strong Stability Preserving Runge-Kutta 3rd order method
    Note: start and result can be the same

**Private Members**

int nstages = {2}
    Number of stages in the RKL.

BoutReal out_timestep = {0.0}
    The output timestep.

int nsteps = {0}
    Number of output steps.

BoutReal timestep = {0.0}
    The internal timestep.

bool adaptive = {true}
    Adapt timestep using tolerances?

BoutReal atol = {1e-10}
    Absolute tolerance.

BoutReal rtol = {1e-5}
    Relative tolerance.

BoutReal max_timestep = {1.0}
    Maximum timestep.

BoutReal max_timestep_change = {2.0}
    Maximum factor by which the timestep should be changed.
int \texttt{mxstep} = \{1000\}
\hspace{1em} Maximum number of internal steps between outputs.

int \texttt{adapt\_period} = \{1\}
\hspace{1em} Number of steps between checks.

bool \texttt{diagnose} = \{false\}
\hspace{1em} Turn on diagnostic output.

int \texttt{nlocal} = \{0\}
int \texttt{neq} = \{0\}
\hspace{1em} Number of variables on local processor and in total.

\texttt{Array<BoutReal> state}
\hspace{1em} System state.

\texttt{Array<BoutReal> u1}
\hspace{1em} Temporary time-stepping arrays These are used by both diffusion and advection time-step routines

\texttt{Array<BoutReal> u2}
\texttt{Array<BoutReal> u3}
\texttt{Array<BoutReal> dydt}
\texttt{Array<BoutReal> state1}
\hspace{1em} Arrays used for adaptive timestepping.

\texttt{Array<BoutReal> state2}

\textbf{K.2.236 File spt.cxx}

Simple Parallel Tridiagonal solver.

\textit{Changelog}

2014-06 Ben Dudson benjamin.dudson@york.ac.uk
\hspace{1em} • Removed static variables in functions, changing to class members.

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu
Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but \textbf{WITHOUT ANY WARRANTY}; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.
class LaplaceSPT : public Laplacian
#include <spt.hxx> Simple parallelisation of the Thomas tridiagonal solver algorithm (serial code)

This is a reference code which performs the same operations as the serial code. To invert a single XZ slice (**FieldPerp** object), data must pass from the innermost processor (**localmesh->PE_XIND = 0**) to the outermost (**localmesh->PE_XIND = localmesh->NXPE-1**) and back again.

Some parallelism is achieved by running several inversions simultaneously, so while processor #1 is inverting Y=0, processor #0 is starting on Y=1. This works ok as long as the number of slices to be inverted is greater than the number of X processors (**MYSUB > localmesh->NXPE**). If **MYSUB < localmesh->NXPE** then not all processors can be busy at once, and so efficiency will fall sharply.

```cpp
param b [in] RHS values (Ax = b)
param flags [in] Inversion settings (see boundary.h for values)
param a [in] This is a 2D matrix which allows solution of A = Delp2 + a
param data [out] Structure containing data needed for second half of inversion
param ccoef [in] Optional coefficient for first-order derivative
param d [in] Optional factor to multiply the Delp2 operator
```

**Public Functions**

```cpp
LaplaceSPT(Options *opt = nullptr, const CELL_LOC = CELL_CENTRE, Mesh *mesh_in = nullptr)

~LaplaceSPT()
```

inline virtual void setCoefA(const **Field2D** &val) override
Set coefficients for inversion. Re-builds matrices if necessary.

inline virtual void setCoefC(const **Field2D** &val) override

inline virtual void setCoefD(const **Field2D** &val) override

inline virtual void setCoefEx(const **Field2D** &val) override

inline virtual void setCoefEz(const **Field2D** &val) override

virtual **FieldPerp** solve(const **FieldPerp** &b) override

virtual **FieldPerp** solve(const **FieldPerp** &b, const **FieldPerp** &x0) override

virtual **Field3D** solve(const **Field3D** &b) override
Extracts perpendicular slices from 3D fields and inverts separately.

In parallel (**localmesh->NXPE > 1**) this tries to overlap computation and communication. This is done at the expense of more memory usage. Setting low_mem in the config file uses less memory, and less communication overlap.
virtual Field3D solve(const Field3D &b, const Field3D &x0) override

Performs the laplacian inversion y-slice by y-slice

Parameters

- b – [in] All the y-slices of b_slice, which is the right hand side of the equation
  \( A^*x_{\text{slice}} = b_{\text{slice}} \)
- x0 – [in] All the y-slices of the variable eventually used to set BC

Returns x All the y-slices of x_slice in the equation \( A^*x_{\text{slice}} = b_{\text{slice}} \)

Private Types

enum [anonymous]

Values:

- enumerator SPT_DATA

Private Functions

void tridagForward(dcomplex *a, dcomplex *b, dcomplex *c, dcomplex *r, dcomplex *u, int n, dcomplex *gam, dcomplex &bet, dcomplex &um, bool start = false)

This is the first half of the Thomas algorithm for parallel calculations.

Two complex quantities have to be propagated between processors: bet and u[-1]. This routine takes bet and um from the last processor (if start == false), and returns the values to be passed to the next processor in the same variables.

Parameters

- a – [in] Vector of matrix coefficients (Left of diagonal)
- b – [in] Vector of matrix coefficients (Diagonal)
- c – [in] Vector of matrix coefficients (Right of diagonal)
- r – [in] RHS vector
- u – [in] Result vector (Au = r)
- n – [in] Size of the matrix
- gam – [out] Intermediate values used for backsolve stage
- bet – [inout]
- um – [inout]
- start – [in]

void tridagBack(dcomplex *u, int n, dcomplex *gam, dcomplex &gp, dcomplex &up)

Second (backsolve) part of the Thomas algorithm.

Parameters

- u – [inout] Result to be solved (Au = r)
- n – [in] Size of the problem
- gam – [in] Intermediate values produced by the forward part
• \textbf{gp} – [inout] \texttt{gam} from the processor \texttt{localmesh->PE\_XIND + 1}, and returned to \texttt{localmesh->PE\_XIND - 1}

• \textbf{up} – [inout] \texttt{u} from processor \texttt{localmesh->PE\_XIND + 1}, and returned to \texttt{localmesh->PE\_XIND - 1}

def \textbf{start}(const \texttt{FieldPerp \&b, SPT\_data \&data})

Simple parallelisation of the Thomas tridiagonal solver algorithm (serial code)

This is a reference code which performs the same operations as the serial code. To invert a single XZ slice (\texttt{FieldPerp} object), data must pass from the innermost processor (localmesh->PE\_XIND = 0) to the outermost (localmesh->PE\_XIND = localmesh->NXPE-1) and back again.

Some parallelism is achieved by running several inversions simultaneously, so while processor #1 is inverting \texttt{Y=0}, processor #0 is starting on \texttt{Y=1}. This works ok as long as the number of slices to be inverted is greater than the number of X processors (MYSUB > localmesh->NXPE). If MYSUB < localmesh->NXPE then not all processors can be busy at once, and so efficiency will fall sharply.

Parameters

• \texttt{b} – [in] RHS values (Ax = b)

• \texttt{data} – [out] Structure containing data needed for second half of inversion

def \textbf{next}(SPT\_data \&data)

Shifts the parallelised Thomas algorithm along one processor.

Returns non-zero when the calculation is complete.

Parameters \texttt{data} – [inout] Structure which keeps track of the calculation

def \textbf{finish}(SPT\_data \&data, FieldPerp \&x)

Finishes the parallelised Thomas algorithm

Parameters

• \texttt{data} – [inout] Structure keeping track of calculation

• \texttt{x} – [out] The result

Private Members

\texttt{Field2D Acoef}

\texttt{Field2D Ccoef}

\texttt{Field2D Dcoef}

\texttt{int ys}

\texttt{int ye}

\texttt{SPT\_data slicedata}

\texttt{SPT\_data *alldata}

\texttt{Array<dccomplex> dcl1d}

1D in Z for taking FFTs

\texttt{struct SPT\_data}

Data structure for SPT algorithm.
Public Functions

```cpp
inline SPT_data()  
void allocate(int mm, int nx)  
inline ~SPT_data()  
```

Public Members

```cpp
int jy  
    Y index.  

Matrix<dcomplex> bk  
    b vector in Fourier space  

Matrix<dcomplex> xk  

Matrix<dcomplex> gam  

Matrix<dcomplex> avec  

Matrix<dcomplex> bvec  

Matrix<dcomplex> cvec  
    Diagonal bands of matrix.  

int proc  
int dir  
comm_handle recv_handle  
int comm_tag  
Array<BoutReal> buffer  
```

K.2.238 File stencils.hxx

Sets stencils for differencing

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.
You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

**Functions**

```cpp
template<
    DIRECTION direction, STAGGER stagger = STAGGER::None, int nGuard = 1, typename FieldType>
inline void populateStencil(stencil &s, const FieldType &f, const typename FieldType::ind_type i)
```

```cpp
template<
    DIRECTION direction, STAGGER stagger = STAGGER::None, int nGuard = 1, typename FieldType>
inline stencil populateStencil(const FieldType &f, const typename FieldType::ind_type i)
```

```cpp
struct stencil
    #include <stencils.hxx> Defines a set of values in 1D in the neighbourhood of an index Used for calculating derivatives
```

**Public Members**

- `BoutReal mm = BoutNaN`
  
  stencil 2 each side of the centre in effect means M\(\geq\)G > 2 is not supported

- `BoutReal m = BoutNaN`
  
- `BoutReal c = BoutNaN`
  
- `BoutReal p = BoutNaN`
  
- `BoutReal pp = BoutNaN`

**K.2.239 File surfaceiter.cxx**

**K.2.240 File surfaceiter.hxx**

Defines a class for iterating over flux surfaces (surfaces of constant x)

```cpp
class SurfaceIter
    #include <surfaceiter.hxx> Iterates over Y-Z surfaces, optionally distributing work between processors
```

**Example**

```cpp
SurfaceIter si(mesh);
for( si.first(); !si.isDone(); si.next() ) { // Perform operation at x = si.xpos if(si.closed()) { // A closed flux surface (no boundaries in Y) } else { // Open, so boundaries in Y if(si.firstY()) { // Boundary at lower Y on this processor } if(si.lastY()) { // Boundary at upper Y on this processor } } }
```
Public Functions

```cpp
inline SurfaceIter(Mesh *mesh, bool include_guards = false)
    Constructor, needs a mesh to iterate over

    Parameters mesh – [in] The mesh to iterate over

int ySize()
    Return the length of the current surface in Y.

bool closed()
    Test if the current surface is closed.

bool closed(BoutReal &ts)
    Test if the current surface (x = xpos) is closed

    Parameters ts – [out] The twist-shift angle by which points are shifted in Z between the end and beginning of Y

MPI_Comm communicator()
    Communicator for this surface.

int yGlobal(int yloc)
    Return global y index of given local index yloc.

bool firstY()
    Is this processor at the lower end?

bool lastY()
    Is this processor at the upper end?

void first()
    Begin iteration.

void next()
    Move to next flux surface.

bool isDone()
    Are we done iterating?
```

Public Members

```cpp
int xpos
    X position where iteration is currently at.
```

Private Members

```cpp
Mesh *m
    The mesh being iterated over.

const int firstpos
const int lastpos
```
K.2.241 File template_combinations.hxx

Routines and helper types for calling a templated function with all combinations of various sets of types/values.

Copyright 2018 D. Dickinson, P. Hill

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

Functions

template<typename item, typename theFunction>
void addItemToDeferredFunction(theFunction func, item)

Now we define routines for dealing with Sets of types We use recursion to unpack each Set such that we can form all combinations of the contained types. As we empty Sets and get individual items we change the number of arguments and hence provide several overloads with different numbers of Sets. Finally we end up with a single item this is the point at which we have a unique combination of the template types and can final invoke the DeferredFunction

Note that we define the routines from the bottom up so that we don’t have to pre-declare routines.

Note we make use of type inference with templates to be able to refer to the first item in a Set/pack allowing us to extract items from Sets. This is the lowest level routine we now have a unique combination of template parameters provided to the DeferredFunction (completed by passing in item) and can therefore invoke this functor.

template<typename item, typename lastSet, typename theFunction>
void addItemToDeferredFunction(theFunction func, item, lastSet)

One Set left to process so no template pack required.

template<typename item, typename nextSet, typename ...otherSets, typename theFunction>
void addItemToDeferredFunction(theFunction func, item, nextSet, otherSets...)

More than one Set left to process.

template<typename ...Sets, typename theFunction>
void processSet(theFunction func, Set<>... Sets...)

Terminal routine the current Set is empty so nothing left to do.

template<typename firstItem, typename ...otherItems, typename ...otherSets, typename theFunction>
void processSet(theFunction func, Set<firstItem, otherItems...>, otherSets... others)

Here we use type inference to allow us to refer to the first Item in the first Set and the other Items in this Set. We use this to pass the first Item off to the routines that will add this to the DeferredFunction. Following this we use recursion to call this routine again to process the rest of this Set.

template<typename ...Ts>
struct Set
#include <template_combinations.hxx> Here we define an empty templated struct that can represent a collection of arbitrary types. This is useful for passing template packs (typename...) around whilst being able to distinguish between different template packs.

template<typename T>

struct TypeContainer
#include <template_combinations.hxx> Here we provide a container type that can be used to pass around a type without needing to create instances of the specific type (instead we create instances of the container type).

Public Types

using type = T
template<typename currentFunction, typename currentType>

struct DeferredFunction
#include <template_combinations.hxx> Define a struct (functor) that we use to build up the final collection of template values. Each time we create one of these objects we provide one more type to the templatePack. We may call the operator() method by providing all the required template parameters for whatever the storedFunc is at that point once we have built a complete templatePack we don’t need to specify any of these template parameters as they can be deduced/inferred.

Public Functions

inline explicit DeferredFunction(currentFunction f)

template<typename ...templatePack>
inline void operator()(templatePack... args)

Public Members

currentFunction storedFunc
template<typename FirstSet, typename ...otherSets>

struct produceCombinations
#include <template_combinations.hxx> This is the top level routine that takes the different Sets of types and triggers the construction of instances of theFunction with all the combinations of the template types defined by the Sets.

A use of this might look like: produceCombinations< Set<typeA, typeB, typeC>, Set<int, double, std::string> >(someFunctionWhichTakesTwoTemplateTypeArguments);

Note we wrap this in a struct such that by declaring a global variable of this type we trigger the creation of the combinations.
Public Functions

template<typename theFunction>
inline explicit produceCombinations(theFunction func)

K.2.242 File timer.cxx

K.2.243 File timer.hxx

Defines

AUTO_TIME()

class Timer

#include <timer.hxx> Timing class for performance benchmarking and diagnosis

To record the time spent in a particular function, create a Timer object when you wish to start timing

```cpp
void someFunction() {
    Timer timer("test"); // Starts timer
}
```

// Timer stops when goes out of scope

Each time this function is called, the total time spent in someFunction will be accumulated. To get the total time spent use getTime()

```cpp
Timer::getTime("test"); // Returns time in seconds as double
```

To reset the timer, use resetTime

```cpp
Timer::resetTime("test"); // Timer reset to zero, returning time as double
```

Public Types

using clock_type = typename std::conditional<std::chrono::high_resolution_clock::is_steady,
std::chrono::high_resolution_clock, std::chrono::steady_clock>::type

using seconds = std::chrono::duration<double, std::chrono::seconds::period>

Public Functions

Timer()

Create a timer. This constructor is equivalent to Timer(""")

explicit Timer(const std::string &label)

Create a timer, continuing from last time if the same label has already been used

~Timer()

Stop the timer
inline double getTime()
Get the time in seconds for time particular Timer object

```cpp
Timer timer("test");
// Some calculation
output << timer.getTime();
// timer still counting
```

inline double getTotalTime()
Get the total time in seconds since the very first initialisation.

inline double resetTime()
Get the time in seconds, reset timer to zero

**Public Static Functions**

static inline double getTime(const std::string &label)
The total time in seconds

static inline double getTotalTime(const std::string &label)
Total time elapsed since the very first initialisation.

static inline double resetTime(const std::string &label)
The total time in seconds, resets the timer to zero

static void cleanup()
Clears all timers, freeing memory

static inline std::map<std::string, timer_info> getAllInfo()
Return the map of all the individual timers.

static void printTimeReport()
Print a table listing all known timers to output

Table is sorted by descending largest total time and has columns for total time, percentage of largest total time, total number of hits, and mean time per hit

**Private Members**

`timer_info &timing`
The current timing information.

**Private Static Functions**

static `timer_info &getInfo(const std::string &label)`
Get a timing info object by name or return a new instance.

static `double getTime(const timer_info &info)`
Get the elapsed time in seconds for timing info.

static `double getTotalTime(const timer_info &info)`
Get the total elapsed time in seconds since the first initialisation.

static `double resetTime(timer_info &info)`
Get the elapsed time, reset timing info to zero.
Private Static Attributes

static std::map<std::string, timer_info> info
Store of existing timing info objects.

struct timer_info
Structure to contain timing information.

Public Members

seconds time
Time of last duration/since last reset.

seconds total_time
Total time since initial creation.

bool running
Is the timer currently running?

clock_type::time_point started
Start time.

unsigned int counter
Number of Timer objects associated with this timer_info

unsigned int hits
Number of times this Timer was hit.

K.2.244 File traits.hxx

namespace bout
SNB model

namespace utils

Typedefs

using is_Field = std::is_base_of<Field, T>
If T is derived from Field, provides the member constant value equal to true. Otherwise value is false.

The following is C++14, but simplifies the use of is_field:

```
template <class T>
cconstexpr bool is_field_v = is_field<T>::value;
```

Examples
template <class T>
void print_field(const T& field) {
    static_assert(bout::utils::is_field<T>::value,
        "print_field only works with Field2Ds, Field3Ds or FieldPerps")
    // implementation
}

using is_Field2D = std::is_base_of<Field2D, T>
If T is derived from Field2D, provides the member constant value equal to true. Otherwise value is false.

using is_Field3D = std::is_base_of<Field3D, T>
If T is derived from Field3D, provides the member constant value equal to true. Otherwise value is false.

using is_FieldPerp = std::is_base_of<FieldPerp, T>
If T is derived from FieldPerp, provides the member constant value equal to true. Otherwise value is false.

using EnableIfField = typename std::enable_if<details::and_all(is_Field<Ts>::value...), typename std::common_type<Ts...>::type>::type
Enable a function if all the Ts are subclasses of Field, and returns the common type: i.e. Field3D if at least one argument is Field3D, otherwise Field2D if they are all Field2D

This is most useful in two particular cases:

A. when there are multiple overloads for a function but some only make sense for fields (as opposed to BoutReal, say) or vice-versa, and some overloads should not be used for fields

B. when a function takes multiple fields and the return type is also a field and must be “big enough"

In other cases, such as a function without overloads that only works for fields, consider using static_assert with is_Field to give a nice compile-time error

Examples
Consider the following template function:

```cpp
template <class T, class U, class V,
    class ResultType = typename bout::utils::EnableIfField<T, U, V>> auto where(const T& test, const U& gt0, const V& le0) -> ResultType {
    // function body
}
```

This function only “appears” if T, U and V are all subclasses of Field. ResultType is the common type of T, U and V. If T and U are both Field2D, ResultType is Field2D if V is Field2D, and Field3D if V is Field3D.

using EnableIfField2D = typename std::enable_if<details::and_all(is_Field2D<Ts>::value...), typename std::common_type<Ts...>::type>::type
Enable a function if all the Ts are subclasses of Field2D, and returns the common type

using EnableIfField3D = typename std::enable_if<details::and_all(is_Field3D<Ts>::value...), typename std::common_type<Ts...>::type>::type
Enable a function if all the Ts are subclasses of Field3D, and returns the common type
using `EnableIfFieldPerp = typename std::enable_if<
<details::and_all(is_FieldPerp<Ts>::value...),
typename std::common_type<Ts...>::type>::type

Enable a function if all the Ts are subclasses of `FieldPerp`, and returns the common type

namespace `details` 

**Functions**

```cpp
constexpr bool and_all()
    Helper class for fold expressions pre-C++17


template<class T>
constexpr bool and_all(T cond)

template<class T, class ...Ts>
constexpr bool and_all(T cond, Ts... conds)
```

**K.2.245 File type_name.cxx**

namespace `bout`
    SNB model

    namespace `utils`

**K.2.246 File type_name.hxx**

**Defines**

```cpp
TYPE_NAME_HXX
```

namespace `bout`
    SNB model

    namespace `utils`
**Functions**

template<typename T>
`std::string typeName()`

template<> `std::string typeName<bool>()`

template<> `std::string typeName<int>()`

template<> `std::string typeName<std::string>()`

template<> `std::string typeName<BoutReal>()`

template<> `std::string typeName<Field2D>()`

template<> `std::string typeName<Field3D>()`

template<> `std::string typeName<FieldPerp>()`

**K.2.247 File uncopyable.hxx**

class Uncopyable
`#include <uncopyable.hxx>` Inherit from this class (private) to prevent copying.

Subclassed by *FormatFactory, MeshFactory*

**Public Functions**

`Uncopyable(const Uncopyable&) = delete`

`Uncopyable &operator=(const Uncopyable&) = delete`
Protected Functions

Uncopyable() = default

~Uncopyable() = default

K.2.248 File unused.hxx

Defines

UNUSED(x)
Mark a function parameter as unused in the function body
For GCC, expands to

```cpp
UNUSED_x __attribute__((unused))
```

Macro taken from http://stackoverflow.com/q/7090998/2043465
This will add the “unused” attribute to parameters in function signatures, telling the compiler that we know the
parameter isn’t used. This should cut down on false positives when using -Wunused-parameters.
Additionally, this macro will also rename the parameter so that if it is accidentally used, the compiler will throw
an error.
A better way to do this might be to detect how to silence the warning in configure and use that in the macro
instead.

Example

```cpp
void someFunction(int UNUSED(x)) {};
```

MAYBE_UNUSED(x)
Mark a function parameter as possibly unused in the function body
Unlike UNUSED, this has to go around the type as well:

MAYBE_UNUSED(int foo);

K.2.249 File utils.cxx

Functions

char *copy_string(const char *s)
Allocate memory and copy string s

const std::string lowercase(const std::string &str)
Convert a string to lower case

const std::string uppercase(const std::string &str)
Convert a string to upper case

const std::string lowercasequote(const std::string &str)
Convert to lower case, except inside quotes (" or ")
**BoutReal** `stringToReal` (const `std::string` &s)

Convert a string to a BoutReal. Throws ` BoutException` if can’t be done.

`int stringToInt` (const `std::string` &s)

Convert a string to an int.

Throws `BoutException` if can’t be done.

`std::list<std::string> &strsplit` (const `std::string` &s, char delim, `std::list<std::string> &elems`)

Split a string on a given delimiter.

**Parameters**

- `s` – **[in]** The string to split (not modified by call)
- `delim` – **[in]** The delimiter to split on (single char)
- `elems` – **[inout]** A list to which the pieces will be appended using `push_back`

`std::list<std::string> strsplit` (const `std::string` &s, char delim)

Split a string on a given delimiter.

**Parameters**

- `s` – **[in]** The string to split (not modified by call)
- `delim` – **[in]** The delimiter to split on (single char)

`std::string trim` (const `std::string` &s, const `std::string` &c)

Strips leading and trailing spaces from a string.

**Parameters**

- `s` – **[in]** The string to trim (not modified)
- `c` – **[in]** Collection of characters to remove

`std::string trimRight` (const `std::string` &s, const `std::string` &c)

Strips leading spaces from a string.

**Parameters**

- `s` – **[in]** The string to trim (not modified)
- `c` – **[in]** Collection of characters to remove

`std::string trimLeft` (const `std::string` &s, const `std::string` &c)

Strips leading spaces from a string.

**Parameters**

- `s` – **[in]** The string to trim (not modified)
- `c` – **[in]** Collection of characters to remove

`std::string trimComments` (const `std::string` &s, const `std::string` &c)

Strips the comments from a string.

**Parameters**

- `s` – **[in]** The string to trim (not modified)
- `c` – **[in]** Collection of characters to remove

`std::string toString` (const `time_t` &time)

Convert a time stamp to a string. This uses `std::localtime` and `std::put_time`
K.2.250 File utils.hxx

A mix of short utilities for memory management, strings, and some simple but common calculations
Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu
Contact: Ben Dudson, bd512@york.ac.uk
This file is part of BOUT++.
BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.
You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

Defines

bout_vsnprintf(buf, len, fmt)
the bout_vsnprintf macro: The first argument is an char * buffer of length len. It needs to have been allocated with new[], as it may be reallocated. len: the length of said buffer. May be changed, mussn’t be const. fmt: the const char * describing the format. note that fmt should be the first argument of the function of type const char * and has to be directly followed by the variable arguments.

BOUT_CONCAT_(A, B)
Utility to evaluate and concatenate macro symbols Note that ## operator doesn’t evaluate symols A or B

BOUT_CONCAT(A, B)

Functions

template<typename T>
int invert3x3(Matrix<T> &a, BoutReal small = 1.0e-15)
Explicit inversion of a 3x3 matrix a
The input small determines how small the determinant must be for us to throw due to the matrix being singular (ill conditioned); If small is less than zero then instead of throwing we return 1. This is ugly but can be used to support some use cases.

inline BoutReal randomu()
Get Random number between 0 and 1

template<typename T>
T SQ(const T &t)
Calculate the square of a variable t i.e. t * t

inline int ROUND(BoutReal x)
Round x to the nearest integer
**BOUT**++ Documentation, Release 4.4.0

\[ T \ \text{BOUTMAX}(T \ a) \]
Calculate the maximum of a list of values using a > b operator

```
template<typename T, typename ...Args>
T BOUTMAX(T a, T b, Args... args)
```

\[ T \ \text{BOUTMIN}(T \ a) \]
Calculate the minimum of a list of values using the a < b operator

```
template<typename T, typename ...Args>
T BOUTMIN(T a, T b, Args... args)
```

inline bool is_pow2(int x)
Check if a number is a power of 2

```
template<typename T>
T SIGN(T a)
Return the sign of a number a by testing if a > 0
```

```
inline BoutReal MINMOD(BoutReal a, BoutReal b)
The minimum absolute value of a and b
if a and b have opposite signs, return zero
if |a| < |b| then return a, otherwise return b
```

inline void checkData(BoutReal f)
Throw an exception if f is not finite.

```
char *copy_string(const char *s)
Allocate memory and copy string s
```

```
template<class T>
std::string toString(const T &val)
Convert a value to a string by writing to a stringstream
```

```
inline std::string toString(const std::string &val)
Simple case where input is already a string This is so that toString can be used in templates where the type may be std::string.
```

```
template<>
inline std::string toString(const Array<BoutReal> &val)
```

```
template<>
inline std::string toString(const Matrix<BoutReal> &val)
```

```
template<>
inline std::string toString(const Tensor<BoutReal> &val)
```

```
inline std::string toString(const bool &val)
Convert a bool to “true” or “false”.
```

```
std::string toString(const time_t &time)
Convert a time stamp to a string This uses std::localtime and std::put_time
```

```
const std::string lowercase(const std::string &str)
Convert a string to lower case
```
const std::string uppercase(const std::string &str)
Convert a string to upper case

const std::string lowercasequote(const std::string &str)
Convert to lower case, except inside quotes (" or ‘)

BoutReal stringToReal(const std::string &s)
Convert a string to a BoutReal Throws BoutException if can’t be done

int stringToInt(const std::string &s)
Convert a string to an int
Throws BoutException if can’t be done

std::list<std::string> &strsplit(const std::string &s, char delim, std::list<std::string> &elems)
Split a string on a given delimiter
Parameters
• s – [in] The string to split (not modified by call)
• delim – [in] The delimiter to split on (single char)
• elems – [inout] A list to which the pieces will be appended using push_back

std::list<std::string> strsplit(const std::string &s, char delim)
Split a string on a given delimiter
Parameters
• s – [in] The string to split (not modified by call)
• delim – [in] The delimiter to split on (single char)

std::string trim(const std::string &s, const std::string &c = " ")
Strips leading and trailing spaces from a string
Parameters
• s – [in] The string to trim (not modified)
• c – [in] Collection of characters to remove

std::string trimLeft(const std::string &s, const std::string &c = " ")
Strips leading spaces from a string
Parameters
• s – [in] The string to trim (not modified)
• c – [in] Collection of characters to remove

std::string trimRight(const std::string &s, const std::string &c = " ")
Strips leading spaces from a string
Parameters
• s – [in] The string to trim (not modified)
• c – [in] Collection of characters to remove

std::string trimComments(const std::string &s, const std::string &c = "#;")
Strips the comments from a string
Parameters
• s – [in] The string to trim (not modified)
• c – [in] Collection of characters to remove
`T *pointer(T *val)`

Convert pointer or reference to pointer. This allows consistent handling of both in macros, templates.

template<typename T>
`T *pointer(T &val)`

template<typename T>

class `Matrix`

#include <utils.hxx> Helper class for 2D arrays

Allows bounds checking through `operator()` with CHECK > 1

If either n1 or n2 are 0, the `Matrix` is empty and should not be indexed

**Public Types**

using `data_type = T`

using `size_type = int`

**Public Functions**

`Matrix() = default`

`inline Matrix(size_type n1, size_type n2)`

`inline Matrix(const Matrix &other)`

`inline void reallocate(size_type new_size_1, size_type new_size_2)`

Reallocate the `Matrix` to shape `new_size_1` by `new_size_2`

Note that this invalidates the existing data!

`inline Matrix &operator=(const Matrix &other)`

`inline T &operator[](size_type i1, size_type i2)`

`inline const T &operator[](size_type i1, size_type i2) const`

`inline Matrix &operator=(const T &val)`

`inline T *begin()`

`inline const T *begin() const`

`inline T *end()`

`inline const T *end() const`
inline std::tuple<
   size_type, size_type>
   shape() const

inline bool empty() const

inline void ensureUnique()
   Ensures that this Matrix does not share data with another This should be called before performing any
write operations on the data.

inline Array<T> &getData()
   Access the underlying storage.

inline const Array<T> &getData() const

**Private Members**

size_type n1 = {0}
size_type n2 = {0}
Array<T> data
   Underlying 1D storage array.

template<typename T>
class Tensor
   #include <utils.hxx> Helper class for 3D arrays
   Allows bounds checking through operator() with CHECK > 1
   If any of n1, n2 or n3 are 0, the Tensor is empty and should not be indexed

**Public Types**

using data_type = T
using size_type = int

**Public Functions**

Tensor() = default

inline Tensor(
   size_type n1, size_type n2, size_type n3)

inline Tensor(const Tensor &other)

inline void reallocate(
   size_type new_size_1, size_type new_size_2, size_type new_size_3)
   Reallocate the Tensor with shape new_size_1 by new_size_2 by new_size_3
   Note that this invalidates the existing data!

inline Tensor &operator=(
   const Tensor &other)
inline constexpr T &operator()(size_type i1, size_type i2, size_type i3)

inline const T &operator()(size_type i1, size_type i2, size_type i3) const

inline Tensor &operator=(const T &val)

inline T *begin()

inline const T *begin() const

inline T *end()

inline const T *end() const

inline std::tuple<size_type, size_type, size_type> shape() const

inline bool empty() const

inline void ensureUnique()

Ensures that this Tensor does not share data with another. This should be called before performing any write operations on the data.

inline Array<T> &getData()

Access the underlying storage.

inline const Array<T> &getData() const

Private Members

size_type n1 = {0}
size_type n2 = {0}
size_type n3 = {0}

Array<T> data
Underlying 1D storage array.

namespace bout
SNB model

namespace utils
Functions

```cpp
template<class T, class ... Args>
_Unique_if<T>::_Single_object make_unique(Args&&... args)
```

```cpp
template<class T>
_Unique_if<T>::_Unknown_bound make_unique(size_t n)
```

```cpp
template<class T, class ... Args>
_Unique_if<T>::_Known_bound make_unique(Args&&...) = delete
```

```cpp
template<class T>
struct _Unique_if
```

Public Types

```
using _Single_object = std::unique_ptr<T>
```

```cpp
template<class T>
struct _Unique_if<T[]>
```

Public Types

```
using _Unknown_bound = std::unique_ptr<T[]>
```

```cpp
template<class T, size_t N>
struct _Unique_if<T[N]>
```

Public Types

```
using _Known_bound = void
```

```cpp
template<typename T>
struct function_traits
```

```cpp
template<typename R, typename ... Args>
struct function_traits<R (*)(Args...)>
```

```
#include <utils.hxx> Traits class to get the types of function arguments for function pointers
```

Use like: using some_function = int(*)(int, double, std::string); // Get the type of the first argument: using first_argument_type = bout::utils::function_traits<some_function>::arg<1>::type;

```
// The following prints "true": std::cout << std::boolalpha << std::is_same<double, first_argument_type>::value;
```

Adapted from https://stackoverflow.com/a/9065203/2043465
Public Types

using result_type = R
using arg_t = typename arg<i>::type

Public Static Attributes

static constexpr size_t nargs = sizeof...(Args)
  Total number of arguments.

template<typename i>
struct arg

Public Types

using type = typename std::tuple_element<i, std::tuple<Args...>::type

K.2.251 File uuid.h

Defines

BOUT_UUID_H

namespace uuids

Typedefs

using uuid_random_generator = basic_uuid_random_generator<std::mt19937>

Enums

enum uuid_variant

  Values:

  enumerator ncs
  enumerator rfc
  enumerator microsoft
  enumerator reserved

enum uuid_version

  Values:

  enumerator none
  enumerator time_based
  enumerator dce_security
  enumerator name_based_md5
  enumerator random_number_based
  enumerator name_based_sha1

Functions

inline bool operator==(uuid const &lhs, uuid const &rhs) noexcept

inline bool operator!=(uuid const &lhs, uuid const &rhs) noexcept

inline bool operator<(uuid const &lhs, uuid const &rhs) noexcept

template<class Elem, class Traits>
std::basic_ostream<Elem, Traits> &operator<<(std::basic_ostream<Elem, Traits> &s, uuid const &id)

template<class CharT = char, class Traits = std::char_traits<CharT>, class Allocator = 
std::allocator<CharT>>
inline std::basic_string<CharT, Traits, Allocator> to_string(uuid const &id)

inline void swap(uuids::uuid &lhs, uuids::uuid &rhs) noexcept
### Variables

static `uuid uuid_namespace_dns`{{0x6b, 0xa7, 0xb8, 0x10, 0xad, 0x11, 0xd1, 0x80, 0xb4, 0x00, 0xc0, 0x4f, 0xd4, 0x30, 0xc8}}

static `uuid uuid_namespace_url`{{0x6b, 0xa7, 0xb8, 0x11, 0xad, 0x11, 0xd1, 0x80, 0xb4, 0x00, 0xc0, 0x4f, 0xd4, 0x30, 0xc8}}

static `uuid uuid_namespace_oid`{{0x6b, 0xa7, 0xb8, 0x12, 0xad, 0x11, 0xd1, 0x80, 0xb4, 0x00, 0xc0, 0x4f, 0xd4, 0x30, 0xc8}}

static `uuid uuid_namespace_x500`{{0x6b, 0xa7, 0xb8, 0x14, 0xad, 0x11, 0xd1, 0x80, 0xb4, 0x00, 0xc0, 0x4f, 0xd4, 0x30, 0xc8}}

template<typename `UniformRandomNumberGenerator`>

class `basic_uuid_random_generator`

#### Public Types

using `engine_type` = UniformRandomNumberGenerator

#### Public Functions

inline explicit `basic_uuid_random_generator`(`engine_type` &gen)

inline explicit `basic_uuid_random_generator`(`engine_type` *gen)

inline `uuid` operator() ()

#### Private Members

`std::uniform_int_distribution<uint32_t>` distribution

`std::shared_ptr<UniformRandomNumberGenerator>` generator

class `uuid`

#### Public Types

using `value_type` = uint8_t
**Public Functions**

inline constexpr uuid() noexcept

inline uuid(value_type (&arr)[16]) noexcept

inline uuid(std::array<value_type, 16> const &arr) noexcept

template<typename ForwardIterator>
inline explicit uuid(ForwardIterator first, ForwardIterator last)

inline constexpr uuid_variant variant() const noexcept

inline constexpr uuid_version version() const noexcept

inline constexpr bool is_nil() const noexcept

inline void swap(uuid &other) noexcept

inline char const *as_bytes() const

**Public Static Functions**

template<class CharT = char>
static inline bool is_valid_uuid(CharT const *str) noexcept

template<class CharT = char, class Traits = std::char_traits<CharT>, class Allocator = std::allocator<CharT>>
static inline bool is_valid_uuid(std::basic_string<CharT, Traits, Allocator> const &str) noexcept

template<class CharT = char>
static inline uuid from_string(CharT const *str) noexcept

template<class CharT = char, class Traits = std::char_traits<CharT>, class Allocator = std::allocator<CharT>>
static inline uuid from_string(std::basic_string<CharT, Traits, Allocator> const &str) noexcept
Private Members

\[std::array<value_type, 16> \textbf{data} = \{0\}\]

Friends

friend friend bool operator== (uuid const &lhs, uuid const &rhs) noexcept
friend friend bool operator< (uuid const &lhs, uuid const &rhs) noexcept

\textbf{class uuid\_name\_generator}

Public Functions

inline explicit \textbf{uuid\_name\_generator} (uuid const &namespace\_uuid) noexcept

\textbf{template\<class CharT = char> inline uuid operator\(\) (CharT const *name)}

\textbf{template\<class CharT = char, class Traits = std::char_traits<CharT>, class Allocator = std::allocator<CharT>\> inline uuid operator\(\) (std::basic_string<CharT, Traits, Allocator> const &name)}

Private Functions

inline void \textbf{reset\(\)()}

\textbf{template<typename char\_type, typename = std::enable_if_t<std::is_integral<char\_type>::value> inline void process\_characters(char\_type const *const characters, size\_t const count)}

inline void \textbf{process\_characters(char const *const characters, size\_t const count)}

inline \textbf{uuid make\_uuid}\(\)
**Private Members**

```cpp
uuid nsuuid
detail::sha hasher
class uuid_time_generator
```

**Public Functions**

```cpp
inline uuid_time_generator()

inline uuid operator()()
```

**Private Types**

using `mac_address` = `std::array<unsigned char, 6>`

**Private Functions**

```cpp
inline bool get_mac_address()

inline long long get_time_intervals()
```

**Private Members**

```cpp
mac_address device_address
bool has_mac_address = false
```

**Functions**

```cpp
template<typename TChar>
inline constexpr unsigned char hex2char(TChar const ch)

template<typename TChar>
inline constexpr bool is_hex(TChar const ch)

template<typename TChar>
inline constexpr unsigned char hexpair2char(TChar const a, TChar const b)

static std::mt19937 clock_gen (std::random_device{}())
```
Variables

static std::uniform_int_distribution<short> clock_dis = {-32768, 32767}
static std::atomic_short clock_sequence = {clock_dis(clock_gen)}

class sha1

Public Types

using digest32_t = uint32_t[5]
using digest8_t = uint8_t[20]

Public Functions

inline sha1()

inline void reset()

inline void process_byte(uint8_t octet)

inline void process_block(void const *const start, void const *const end)

inline void process_bytes(void const *const data, size_t const len)

inline uint32_t const *get_digest(digest32_t digest)

inline uint8_t const *get_digest_bytes(digest8_t digest)

Public Static Functions

static inline uint32_t left_rotate(uint32_t value, size_t const count)

Public Static Attributes

static constexpr unsigned int block_bytes = 64
Private Functions

inline void process_block()

Private Members

digest32_t m_digest
uint8_t m_block[64]
size_t m_blockByteIndex
size_t m_byteCount

namespace std
  STL namespace.

  template<> uuid >

Public Types

using argument_type = uuids::uuid
using result_type = std::size_t

Public Functions

inline result_type operator()(argument_type const &uuid) const

K.2.252 File variant.hxx

Defines

__VARIANT_HXX__
  Variant utilities
    All in namespace bout::utils
    variant visit holds_alternative get
    variantEqualTo variantStaticCastOrThrow variantToString
    Internal implementation in bout::utils::details

namespace bout
  SNB model

namespace utils


Functions

template<typename Variant, typename T>
bool variantEqualTo(const Variant &v, const T &t)

Return true only if the given variant v has the same type and value as t

Note: Handles the case that t is not of a type which can hold.

template<typename Variant, typename T>
T variantStaticCastOrThrow(const Variant &v)

Cast a variant to a given type using static_cast If this can’t be done then a std::bad_cast exception is thrown

Note: T can be a type which variant cannot hold in which case std::bad_cast will be thrown at runtime

template<typename Variant>
std::string variantToString(const Variant &v)

namespace details

template<typename T, typename U>
struct CompareTypes

#include <variant.hxx> Compare two values. Different types -> false

Public Functions

inline bool operator()(const T &v, const U &t)

template<typename T>
struct CompareTypes<T, T>

#include <variant.hxx> Compare two values Same type -> use == operator to compare

Public Functions

inline bool operator()(const T &v, const T &t)

template<typename T>
struct IsEqual

#include <variant.hxx> A visitor for std::variant which compares the value stored in the variant with a given value using CompareTypes
Public Functions

inline IsEqual(const T &t)

template<typename U>
inline bool operator()(const U &u)

Public Members

const T &t

template<typename Target>
struct StaticCastOrThrow

#include <variant.hxx> Functor to perform static casting with std::visit If the Target cannot be constructed from the Source then an exception (std::bad_cast) will be thrown at run time.

Note: This needs to be at runtime because the particular type which a variant is holding is only known at runtime.

Public Functions

template<typename Source>
inline Target operator()(Source &&source) const

template<typename Source>
inline Target operator()(Source &&source, std::false_type)

template<typename Source>
inline Target operator()(Source &&source, std::true_type)

struct ToString

Public Functions

template<typename T>
inline std::string operator()(T &&val)
K.2.253 File vecops.cxx

Functions

const Vector2D Grad(const Field2D &f, CELL_LOC outloc, const std::string &method)
Gradient of scalar field \( f \), returning a covariant vector
All locations supported
Parameters
• \( f \) – [in] The field to differentiate
• \( \text{outloc} \) – [in] The location where the result is desired. By default, this is the same location as the input \( f \)
• \( \text{method} \) – [in] The method to use. The default is set in the options.

cost Vector3D Grad(const Field3D &f, CELL_LOC outloc, const std::string &method)

cost Vector3D Grad_perp(const Field3D &f, CELL_LOC outloc, const std::string &method)
Perpendicular gradient of scalar field \( f \)
outloc must be either CELL_DEFAULT or f.getLocation() > argument can be removed
result.x = df/dx - g_12/(JB)^2 df/dy result.y = 0 result.z = df/dz - g_23/(JB)^2 df/dy
Parameters
• \( f \) – [in] The field to differentiate
• \( \text{outloc} \) – [in] The cell location where the result is desired
• \( \text{method} \) – [in] The method to use. The default is set in the options.

cost Vector2D Grad_perp(const Field2D &f, CELL_LOC outloc, const std::string &method)

cost Field2D Div(const Vector2D &v, CELL_LOC outloc, const std::string &method)
Divergence of a vector \( v \), returning a scalar
All locations except CELL_VSHIFT supported. Note that if \( v \) is at CELL_VSHIFT, then outloc must be CELL_CENTRE
Parameters
• \( v \) – [in] The vector to differentiate
• \( \text{outloc} \) – [in] The cell location where the result is desired
• \( \text{method} \) – [in] The method to use. The default is set in the options.

cost Field3D Div(const Vector3D &v, CELL_LOC outloc, const std::string &method)

cost Field2D Div(const Vector2D &v, const Field2D &f, CELL_LOC outloc, const std::string &method)

cost Field3D Div(const Vector3D &v, const Field3D &f, CELL_LOC outloc, const std::string &method)

cost Vector2D Curl(const Vector2D &v)
Curl of a vector
Does not currently support any output locations. \( v \) must not be at CELL_VSHIFT
We can’t support VSHIFT here as, e.g. DDY can’t produce an output at CELL_XLOW unless the input field is at CELL_XLOW, but then that field will also be needed at CELL_YLOW, for example for another component.

**Parameters**

v – [in] The vector to differentiate

```cpp
const Vector3D Curl(const Vector3D &v)
```

```cpp
const Field2D V_dot_Grad(const Vector2D &v, const Field2D &f)
```

- Advection of a scalar field \( f \) by a velocity vector \( v \)
- The vector and the field must be at the same location, which cannot be CELL_VSHIFT

```cpp
const Field3D V_dot_Grad(const Vector2D &v, const Field3D &f)
```

```cpp
const Field3D V_dot_Grad(const Vector3D &v, const Field3D &f)
```

```cpp
template<typename T, typename F, typename R = decltype(T{} + F{})>
R V_dot_Grad(const T &v, const F &a)
```

```cpp
const Vector2D V_dot_Grad(const Vector2D &v, const Vector2D &a)
```

- Advection of a vector field \( a \) by a velocity vector \( v \)
- Both vectors must be at the same location, which cannot be CELL_VSHIFT

```cpp
const Vector3D V_dot_Grad(const Vector2D &v, const Vector3D &a)
```

```cpp
const Vector3D V_dot_Grad(const Vector3D &v, const Vector2D &a)
```

```cpp
const Vector3D V_dot_Grad(const Vector3D &v, const Vector3D &a)
```

---

**K.2.254 File vecops.hxx**

Operators on vector objects B.Dudson, October 2007

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see [http://www.gnu.org/licenses/](http://www.gnu.org/licenses/).
Functions

const Vector2D Grad(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")
Gradient of scalar field \( f \), returning a covariant vector
All locations supported

Parameters

- \( f \) – [in] The field to differentiate
- \( \text{outloc} \) – [in] The location where the result is desired By default this is the same location as the input \( f \)
- \( \text{method} \) – [in] The method to use. The default is set in the options.

const Vector3D Grad(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")

const Vector3D Grad_perp(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")
Perpendicular gradient of scalar field \( f \)
outloc must be either CELL_DEFAULT or f.getLocation() > argument can be removed
result.x = df/dx - g_12/(JB)^2 df/dy result.y = 0 result.z = df/dz - g_23/(JB)^2 df/dy

Parameters

- \( f \) – [in] The field to differentiate
- \( \text{outloc} \) – [in] The cell location where the result is desired
- \( \text{method} \) – [in] The method to use. The default is set in the options.

inline const Vector3D Grad_perp(const Field3D &f, CELL_LOC outloc, DIFF_METHOD method)

const Vector2D Grad_perp(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")

const Field2D Div(const Vector2D &v, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")
Divergence of a vector \( v \), returning a scalar
All locations except CELL_VSHIFT supported. Note that if \( v \) is at CELL_VSHIFT, then outloc must be CELL_CENTRE

Parameters

- \( v \) – [in] The vector to differentiate
- \( \text{outloc} \) – [in] The cell location where the result is desired
- \( \text{method} \) – [in] The method to use. The default is set in the options.

const Field3D Div(const Vector3D &v, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")

const Field2D Div(const Vector2D &v, const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")
const Field3D Div(const Vector3D &v, const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")

inline const Field3D Div(const Vector3D &v, const Field3D &f, const std::string &method, CELL_LOC outloc = CELL_DEFAULT)

inline const Field3D Div(const Vector3D &v, const Field3D &f, DIFF_METHOD method = DIFF_DEFAULT)

inline const Field3D Div(const Vector3D &v, const Field3D &f, DIFF_METHOD method, CELL_LOC outloc = CELL_DEFAULT)

const Vector2D Curl(const Vector2D &v)
  Curl of a vector
  Does not currently support any output locations. v must not be at CELL_VSHIFT
  We can’t support VSHIFT here as, e.g. DDY can’t produce an output at CELL_XLOW unless the input field is at CELL_XLOW, but then that field will also be needed at CELL_YLOW, for example for another component.
  Parameters v – [in] The vector to differentiate

const Vector3D Curl(const Vector3D &v)

const Field2D V_dot_Grad(const Vector2D &v, const Field2D &f)
  Advection of a scalar field f by a velocity vector v
  The vector and the field must be at the same location, which cannot be CELL_VSHIFT

const Field3D V_dot_Grad(const Vector2D &v, const Field3D &f)

const Field3D V_dot_Grad(const Vector3D &v, const Field2D &f)

const Field3D V_dot_Grad(const Vector3D &v, const Field3D &f)

const Vector2D V_dot_Grad(const Vector2D &v, const Vector2D &a)
  Advection of a vector field a by a velocity vector v
  Both vectors must be at the same location, which cannot be CELL_VSHIFT

const Vector3D V_dot_Grad(const Vector2D &v, const Vector3D &a)

const Vector3D V_dot_Grad(const Vector3D &v, const Vector2D &a)

const Vector3D V_dot_Grad(const Vector3D &v, const Vector3D &a)
K.2.255 File vector2d.cxx

Functions

const Vector2D operator* (const BoutReal lhs, const Vector2D &rhs)
const Vector2D operator* (const Field2D &lhs, const Vector2D &rhs)
const Vector3D operator* (const Field3D &lhs, const Vector2D &rhs)

const Field2D abs(const Vector2D &v, const std::string &region)
   Absolute value (Modulus) of given vector v
   |v| = sqrt( v dot v )

K.2.256 File vector2d.hxx

Class for 2D vectors. Built on the Field2D class, all operators relating to vectors are here (none in Field classes). As with Field2D, Vector2D are constant in z (toroidal angle) Components are either co- or contra-variant, depending on a flag. By default they are covariant.

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Author  B. Dudson, October 2007

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

Defines

__VECTOR2D_H__
Functions

const Vector2D operator*(BoutReal lhs, const Vector2D &rhs)

const Vector2D operator*(const Field2D &lhs, const Vector2D &rhs)

const Vector3D operator*(const Field3D &lhs, const Vector2D &rhs)

cross(const Vector2D &lhs, const Vector2D &rhs)

Cross product.

cross(const Vector2D &lhs, const Vector3D &rhs)

Cross product.

Field2D abs(const Vector2D &v, const std::string &region = "RGN_ALL")

Absolute value (Modulus) of given vector v

|v| = sqrt(v dot v)

inline const Field2D abs(const Vector2D &v, REGION region)

inlinetofieldaligned(Vector2D v, const std::string &region = "RGN_ALL")

Transform to and from field-aligned coordinates.

inline fromfieldaligned(Vector2D v, const std::string &region = "RGN_ALL")

cross(const Vector2D &lhs, const Vector3D &rhs)

Create new Vector2D with same attributes as the argument, but uninitialised components.

cross(const Vector2D &lhs, const Vector3D &rhs)

Create new Vector2D with same attributes as the argument, and zero-initialised components.

ddt(Vector2D &f)

Time derivative of 2D vector field.

class Vector2D : public FieldData

#include <vector2d.hxx> A vector with three components (x,y,z) which only vary in 2D (x and y). Implemented as a collection of three Field2D objects.

Public Functions

Vector2D(Mesh *fieldmesh = nullptr)

Vector2D(const Vector2D &f)

Vector2D(Mesh *localmesh, bool covariant, CELL_LOC location)

Many-argument constructor for fully specifying the initialisation of a Vector3D.

~Vector2D() override

void toCovariant()

In-place conversion to covariant form.
void toContravariant()
   In-place conversion to contravariant form.

Vector2D *timeDeriv()
   Return a pointer to the time-derivative field.

Vector2D &operator=(const Vector2D &rhs)
   Assignment.

Vector2D &operator=(BoutReal val)
   Assign a BoutReal value. This sets all components to the same value val.

   Vector2D v = 0.0;
   is equivalent to

   Vector2D v; v.x = 0.0; v.y = 0.0; v.z = 0.0;
   The only real use for this is setting vector to zero.

Vector2D &operator+=(const Vector2D &rhs)

const Vector2D operator-() const
   Unary minus, changes sign of all components.

Vector2D &operator-=(const Vector2D &rhs)
   Subtract another vector.

Vector2D &operator*=(BoutReal rhs)
   Multiply all components by rhs.

Vector2D &operator*=(const Field2D &rhs)
   Multiply all components by rhs.

Vector2D &operator/=(BoutReal rhs)
   Divide all components by rhs.

Vector2D &operator/=(const Field2D &rhs)
   Divide all components by rhs.

const Vector2D operator+(const Vector2D &rhs) const
   Addition.

const Vector3D operator+(const Vector3D &rhs) const
   Addition.

const Vector2D operator-(const Vector2D &rhs) const
   Subtract vector rhs.

const Vector3D operator-(const Vector3D &rhs) const
   Subtract vector rhs.

const Vector2D operator*(BoutReal rhs) const
   Multiply all components by rhs.

const Vector2D operator*(const Field2D &rhs) const
   Multiply all components by rhs.

const Vector3D operator*(const Field3D &rhs) const
   Multiply all components by rhs.

const Vector2D operator/(BoutReal rhs) const
   Divides all components by rhs.
const Vector2D operator/(const Field2D &rhs) const
Divides all components by rhs.

const Vector3D operator/(const Field3D &rhs) const
Divides all components by rhs.

const Field2D operator*(const Vector2D &rhs) const
Dot product.

const Field3D operator*(const Vector3D &rhs) const
Dot product.

void setLocation(CELL_LOC loc)
Set variable cell location

CELL_LOC getLocation() const

virtual void accept(FieldVisitor &v) override
Visitor pattern support.

inline virtual bool isReal() const override
Returns true if field consists of BoutReal values.

inline virtual bool is3D() const override
True if variable is 3D.

inline virtual int byteSize() const override
Number of bytes for a single point.

inline virtual int BoutRealSize() const override
Number of BoutReals (not implemented if not BoutReal)

virtual void applyBoundary(bool init = false) override
Apply boundary condition to all fields.

inline void applyBoundary(const std::string &condition)

inline void applyBoundary(const char *condition)

virtual void applyTDerivBoundary() override

**Public Members**

Field2D x
Field2D y
Field2D z
components

bool covariant = {true}
true if the components are covariant (default)
Private Members

Vector2D *deriv = {nullptr}
Time-derivative, can be NULL.

CELL_LOC location = {CELL_CENTRE}
Location of the variable in the cell.

K.2.257 File vector3d.cxx

Defines

CROSS(v0, v1, v2)

Functions

const Vector3D cross(const Vector3D &lhs, const Vector3D &rhs)
Cross-product of two vectors.

const Vector3D cross(const Vector3D &lhs, const Vector2D &rhs)
Cross-product of two vectors.

const Vector3D cross(const Vector2D &lhs, const Vector3D &rhs)
Cross product.

const Vector2D cross(const Vector3D &lhs, const Vector2D &rhs)
Cross product.

const Vector3D operator*(const BoutReal lhs, const Vector3D &rhs)

const Vector3D operator*(const Field2D &lhs, const Vector3D &rhs)

const Vector3D operator*(const Field3D &lhs, const Vector3D &rhs)

const Field3D abs(const Vector3D &v, const std::string &region)
Absolute magnitude (modulus) of a vector |v|

sqrt( v.x^2 + v.y^2 + v.z^2 )

Vector3D toFieldAligned(const Vector3D &v, const std::string &region)
Transform to and from field-aligned coordinates.

Vector3D fromFieldAligned(const Vector3D &v, const std::string &region)
K.2.258 File vector3d.hxx

Class for 3D vectors. Built on the Field3D class.

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Author  B. Dudson, October 2007
Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.
BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see http://www.gnu.org/licenses/.

Defines

__VECTOR3D_H__

Functions

const Vector3D operator*(BoutReal lhs, const Vector3D &rhs)

const Vector3D operator*(const Field2D &lhs, const Vector3D &rhs)

const Vector3D operator*(const Field3D &lhs, const Vector3D &rhs)

const Vector3D cross(const Vector3D &lhs, const Vector3D &rhs)
   Cross-product of two vectors.

const Vector3D cross(const Vector3D &lhs, const Vector2D &rhs)
   Cross-product of two vectors.

const Field3D abs(const Vector3D &v, const std::string &region = "RGN_ALL")
   Absolute magnitude (modulus) of a vector |v|
   sqrt(v.x^2 + v.y^2 + v.z^2)

inline const Field3D abs(const Vector3D &v, REGION region)

Vector3D toFieldAligned(const Vector3D &v, const std::string &region = "RGN_ALL")
   Transform to and from field-aligned coordinates.

Vector3D fromFieldAligned(const Vector3D &v, const std::string &region = "RGN_ALL")
inline Vector3D emptyFrom(const Vector3D &v)
        Create new Vector3D with same attributes as the argument, but uninitialised components.

inline Vector3D zeroFrom(const Vector3D &v)
        Create new Vector3D with same attributes as the argument, and zero-initialised components.

inline Vector3D &ddt(Vector3D &f)
        Time derivative of 3D vector field.

class Vector3D : public FieldData
        #include <vector3d.hxx> Represents a 3D vector, with x,y,z components stored as separate Field3D objects

Example

Vector3D f;
a.x; // Error! a.x not allocated

Public Functions

Vector3D(Mesh *fieldmesh = nullptr)
        Constructor. Just sets covariant = true and deriv = NULL
        Does not initialise any of the fields

Vector3D(const Vector3D &f)
        Copy constructor. After this the components (x,y,z) will refer to the same data as f.(x,y,z)

Vector3D(Mesh *localmesh, bool covariant, CELL_LOC location)
        Many-argument constructor for fully specifying the initialisation of a Vector3D.

~Vector3D() override
        Destructor. If the time derivative has been used, then some book-keeping is needed to ensure that fields
        are only destroyed once.

void toCovariant()
        In-place conversion to covariant form.
        If already covariant (covariant = true) then does nothing If contravariant, multiplies by metric tensor g_{ij}

void toContravariant()
        In-place conversion to contravariant form
        If already contravariant (covariant = false) then does nothing If covariant, multiplies by metric tensor g^{ij}

Vector3D *timeDeriv()
        Return a pointer to the time-derivative field
        The first time this is called, a new Vector3D object is created. Subsequent calls return a pointer to this
        same object
        For convenience, a standalone function “ddt” exists, so that
        ddt(v) is equivalent to *(v.timeDeriv())
        This does some book-keeping to ensure that the time derivative of the components is the same as the
        components of the time derivative
        ddt(v).x == ddt(v.x)

Vector3D &operator=(const Vector3D &rhs)
Vector3D &operator=(const Vector2D &rhs)

Vector3D &operator=(BoutReal val)

Vector3D &operator+=(const Vector3D &rhs)

Vector3D &operator+=(const Vector2D &rhs)

const Vector3D operator-() const

Vector3D &operator-=(const Vector3D &rhs)

Vector3D &operator-=(const Vector2D &rhs)

Vector3D &operator*=(BoutReal rhs)

Vector3D &operator*=(const Field2D &rhs)

Vector3D &operator*=(const Field3D &rhs)

Vector3D &operator/=(BoutReal rhs)

Vector3D &operator/=(const Field2D &rhs)

Vector3D &operator/=(const Field3D &rhs)

const Vector3D operator+(const Vector3D &rhs) const

const Vector3D operator+(const Vector2D &rhs) const

const Vector3D operator-(const Vector3D &rhs) const

const Vector3D operator-(const Vector2D &rhs) const

const Vector3D operator*(BoutReal rhs) const

const Vector3D operator*(const Field2D &rhs) const

const Vector3D operator*(const Field3D &rhs) const

const Vector3D operator/(BoutReal rhs) const
const Vector3D operator/(const Field2D &rhs) const
const Vector3D operator/(const Field3D &rhs) const
const Field3D operator*(const Vector3D &rhs) const
const Field3D operator*(const Vector2D &rhs) const

void setLocation(CELL_LOC loc)
Set variable cell location
CELL_LOC getLocation() const

virtual void accept(FieldVisitor &v) override
Visitor pattern support.

inline virtual bool isReal() const override
Returns true if field consists of BoutReal values.

inline virtual bool is3D() const override
True if variable is 3D.

inline virtual int byteSize() const override
Number of bytes for a single point.

inline virtual int BoutRealSize() const override
Number of BoutReals (not implemented if not BoutReal)

virtual void applyBoundary(bool init = false) override

inline void applyBoundary(const std::string &condition)

inline void applyBoundary(const char *condition)

virtual void applyTDerivBoundary() override

Public Members

Field3D x
The components of the vector. These can be either co- or contra-variant, depending on the boolean flag “covariant”

Field3D y

Field3D z
bool covariant = {true}
Flag to specify whether the components (x,y,z) are co- or contra-variant.
true if the components are covariant (default) false if the components are contravariant
Conversion between forms should be done by calling the toContravariant and toCovariant methods.
Only modify this variable directly if you know what you are doing!

**Private Members**

```cpp
Vector3D *deriv = nullptr
    Time-derivative, can be NULL.

CELL_LOC location = {CELLCENTRE}
    Location of the variable in the cell.
```

**K.2.259 File where.cxx**

**K.2.260 File where.hxx**

A set of functions which choose between two values

Copyright 2010 B.D.Dudson, S.Farley, M.V.Umansky, X.Q.Xu

Contact: Ben Dudson, bd512@york.ac.uk

This file is part of BOUT++.

BOUT++ is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

BOUT++ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with BOUT++. If not, see [http://www.gnu.org/licenses/](http://www.gnu.org/licenses/).

**Functions**

```cpp
template<class T, class U, class V, class ResultType = typename bout::utils::EnableIfField<T, U, V> auto where(const T &test, const U &gt0, const V &le0) -> ResultType
    For each point, choose between two inputs based on a third input

Parameters

- **test** – [in] The value which determines which input to use
- **gt0** – [in] Uses this value if test > 0.0
- **le0** – [in] Uses this value if test <= 0.0
```

```cpp
template<class T, class U, class ResultType = typename bout::utils::EnableIfField<T, U> auto where(const T &test, const U &gt0, BoutReal le0) -> ResultType
```

```cpp
template<class T, class V, class ResultType = typename bout::utils::EnableIfField<T, V> auto where(const T &test, BoutReal gt0, const V &le0) -> ResultType
```

```cpp
template<class T, class ResultType = T>
```

K.2. File list 755
auto where(const T &test, BoutReal gt0, BoutReal le0) -> ResultType

K.3 Python routines

K.3.1 boututils

• class Datafile provides a convenient way to read and write NetCDF or HDF5 files. There are many different NetCDF libraries available for Python, so this class tries to provide a consistent interface to many of them, as well as to h5py.
• deriv()
• determineNumberOfCPUs()
• file_import() reads the contents of a NetCDF file into a dictionary
• integrate()
• launch()
• linear_regression()
• showdata() visualises and animates 2D data (time + 1 spatial dimension) or 3D data (time + 2 spatial dimensions). The animation object can be returned, or the animation can be saved to a file or displayed on screen.
• boutwarnings contains functions to raise warning messages. alwayswarn() by default prints the warning every time it is called. defaultwarn() by default prints the warning only the first time an instance of it is called. This module is a wrapper for the Python warnings module, so printing the warnings can be controlled using warnings.simplefilter() or warnings.filterwarnings().

K.3.2 boutdata

• collect() provides an interface to read BOUT++ data outputs, returning NumPy arrays of data. It deals with the processor layout, working out which file contains each part of the domain.

from boutdata.collect import collect
t = collect("t_array")  # Collect the time values

• pol_slice() takes a 3 or 4-D dataset for a toroidal equilibrium, and calculates a slice through it at fixed toroidal angle.
• gen_surface() is a generator for iterating over flux surfaces

K.3.3 bout_runners

bout_runners contains classes which gives an alternative way of running BOUT++ simulations either normally using the class basic_runner, or on a cluster through a generated Portable Batch System (PBS) script using the child class PBS_runner. Examples can be found in examples/bout_runners_example/.

bout_runners is especially useful if one needs to make several runs with only small changes in the options (which is normally written in BOUT.inp or in the command-line), as is the case when performing a parameter scan, or when performing a MMS test.
Instead of making several runs with several different input files with only small changes in the option, one can with `bout_runners` specify the changes as member data of an instance of the appropriate `bout_runners` class. One way to do this is to write a `driver` in the same directory as the executable. The `driver` is just a python script which imports `bout_runners`, creates an instance, specifies the running option as member data of that instance and finally calls the member function `self.execute_runs()`.

In addition, the `bout_runners` provides a way to run any python post-processing script after finished simulations (as long as it accept at least one parameter containing the folder name(s) of the run(s)). If the simulations have been performed using the `PBS_runner`, the post-processing function will be submitted to the cluster (although it is possible to submit it to a different queue, using a different amount of nodes etc.).

When the function `self.execute_runs()` is executed, a folder structure like the one presented in Fig. 11.1 is created. BOUT.inp is copied to the folder of execution, where the BOUT.*.dmp files are stored. Secondly a list of combination of the options specified in the driver is made. Eventually unset options are obtained from BOUT.inp or given a default value if the option is nowhere to be found.

![Fig. 11.1: Longest possible folder tree made by the self.execute_runs() function.](image)
K.4 boutdata package

K.4.1 Module contents
K.4.2 Submodules
K.4.3 boutdata.cbdtoeqdsk module
K.4.4 boutdata.collect module
K.4.5 boutdata.data module
K.4.6 boutdata.gen_surface module
K.4.7 boutdata.griddata module
K.4.8 boutdata.input module
K.4.9 boutdata.mayavi2 module
K.4.10 boutdata.mms module
K.4.11 boutdata.pol_slice module
K.4.12 boutdata.processor_rearrange module
K.4.13 boutdata.restart module
K.4.14 boutdata.settings module
K.4.15 boutdata.shiftz module
K.4.16 Module contents

K.5 boututils package

K.5.1 Module contents
K.5.2 Submodules
K.5.3 boututils.analyse_equil_2 module
K.5.4 boututils.anim module
K.5.5 boututils.ask module
K.5.6 boututils.boutarray module
K.5.7 boututils.boutgrid module
K.5.8 boututils.boutwarnings module
K.5.9 boututils.calculus module
**class** zoidberg.boundary.

**NoBoundary**
No boundary, so no points outside

**outside**(x, y, z)
Returns True if the point is outside the boundary

**Parameters**
- x, y, z (*array_like*) – Coordinates of the point(s) to check

**Returns**
- True if point is outside boundary

**Return type**
- bool

**class** zoidberg.boundary.

**PolygonBoundaryXZ**(xarr, zarr)

**outside**(x, y, z)
Returns true if the given point is outside the domain

**Parameters**
- x, y, z (*array_like*) – Coordinates of the point(s) to check

**Returns**
- True if point is outside boundary

**Return type**
- bool

**class** zoidberg.boundary.

**RectangularBoundaryXZ**(xmin, xmax, zmin, zmax)

**outside**(x, y, z)
Returns true if the given point is outside the domain

**Parameters**
- x, y, z (*array_like*) – Coordinates of the point(s) to check

**Returns**
- True if point is outside boundary

**Return type**
- bool

## K.6.4 zoidberg.field module

**class** zoidberg.field.

**CurvedSlab**(By=1.0, Bz=0.1, xcentre=0.0, Bzprime=1.0, Rmaj=1.0)
Represents a magnetic field in a curved slab geometry

Magnetic field in $z = Bz + (x - xcentre) \times Bzprime$

x - Distance in radial direction [m] y - Azimuthal (toroidal) angle z - Height [m]

**Parameters**

- **By** (*float*) – Magnetic field in y direction
- **Bz** (*float*) – Magnetic field in z at xcentre (*float*)
- **xcentre** (*float*) – Reference x coordinate
- **Bzprime** (*float*) – Rate of change of Bz with x
- **Rmaj** (*float*) – Major radius of the slab

**Bxfunc**(x, z, phi)
Magnetic field in x direction at given coordinates

**Parameters**
- x, z, phi (*array_like*) – X, Z, and toroidal coordinates

**Returns**
- X-component of the magnetic field

**Return type**
- ndarray
Byfunc\((x, z, \phi)\)
Magnetic field in \(y\) direction at given coordinates

**Parameters**
\(x, z, \phi\) \((\text{array} \_\text{like})\) – \(X, Z,\) and toroidal coordinates

**Returns**
\(Y\)-component of the magnetic field

**Return type**
\(\text{ndarray}\)

Bzfunc\((x, z, \phi)\)
Magnetic field in \(z\) direction at given coordinates

**Parameters**
\(x, z, \phi\) \((\text{array} \_\text{like})\) – \(X, Z,\) and toroidal coordinates

**Returns**
\(Z\)-component of the magnetic field

**Return type**
\(\text{ndarray}\)

Rfunc\((x, z, \phi)\)

Major radius \([\text{meters}]\)

Returns None if in Cartesian coordinates

**Parameters**
\(x, z, \phi\) \((\text{array} \_\text{like})\) – \(X, Z,\) and toroidal coordinates

**Returns**
The major radius

**Return type**
\(\text{ndarray}\)

class zoidberg.field.GEQDSK\((gfile)\)
Read a EFIT G-Eqdsk file for a toroidal equilibrium

This generates a grid in cylindrical geometry

**Parameters**
gfile \((\text{str})\) – Name of the file to open

Bxfunc\((x, z, \phi)\)
Magnetic field in \(x\) direction at given coordinates

**Parameters**
\(x, z, \phi\) \((\text{array} \_\text{like})\) – \(X, Z,\) and toroidal coordinates

**Returns**
\(X\)-component of the magnetic field

**Return type**
\(\text{ndarray}\)

Byfunc\((x, z, \phi)\)
Magnetic field in \(y\) direction at given coordinates

**Parameters**
\(x, z, \phi\) \((\text{array} \_\text{like})\) – \(X, Z,\) and toroidal coordinates

**Returns**
\(Y\)-component of the magnetic field

**Return type**
\(\text{ndarray}\)

Bzfunc\((x, z, \phi)\)
Magnetic field in \(z\) direction at given coordinates

**Parameters**
\(x, z, \phi\) \((\text{array} \_\text{like})\) – \(X, Z,\) and toroidal coordinates

**Returns**
\(Z\)-component of the magnetic field

**Return type**
\(\text{ndarray}\)

Rfunc\((x, z, \phi)\)
Major radius \([\text{meters}]\)

Returns None if in Cartesian coordinates

**Parameters**
\(x, z, \phi\) \((\text{array} \_\text{like})\) – \(X, Z,\) and toroidal coordinates

**Returns**
The major radius

**Return type**
\(\text{ndarray}\)

K.6. zoidberg package
Returns The major radius
Return type ndarray

pressure(x, z, phi)
Pressure [Pascals]
Parameters x, z, phi (array_like) – X, Z, and toroidal coordinates
Returns The plasma pressure
Return type ndarray

class zoidberg.field.MagneticField
Represents a magnetic field in either Cartesian or cylindrical geometry

This is the base class, you probably don’t want to instantiate one of these directly. Instead, create an instance of one of the subclasses.

Functions which can be overridden
• Bxfunc = Function for magnetic field in x
• Bzfunc = Function for magnetic field in z
• Byfunc = Function for magnetic field in y (default = 1.)
• Rfunc = Function for major radius. If None, y is in meters

boundary
An object with an “outside” function. See zoidberg.boundary

attributes
Contains attributes to be written to the output
   Type A dictionary of string -> function(x,z,phi)

See also:
Slab A straight field in normal Cartesian coordinates
CurvedSlab A field in curvilinear coordinates
StraightStellarator A rotating ellipse stellarator without curvature
VMEC A numerical field from a VMEC equilibrium file
GEQDSK A numerical field from an EFIT g-file

Bmag(x, z, phi)
Magnitude of the magnetic field

\[ B_{mag} = \sqrt{B_x^2 + B_y^2 + B_z^2} \]

Parameters x, z, phi (array_like) – X, Z, and toroidal coordinates
Returns The magnitude of the magnetic field
Return type ndarray

Bxfunc(x, z, phi)
Magnetic field in x direction at given coordinates
Parameters x, z, phi (array_like) – X, Z, and toroidal coordinates
Returns X-component of the magnetic field
Return type ndarray

Byfunc(x, z, phi)
Magnetic field in y direction at given coordinates
Parameters x, z, phi (array_like) – X, Z, and toroidal coordinates
Returns  Y-component of the magnetic field
Return type  ndarray

`Bzfunc(x, z, phi)`
Magnetic field in z direction at given coordinates

Parameters  `x, z, phi (array_like)` – X, Z, and toroidal coordinates
Returns  Z-component of the magnetic field
Return type  ndarray

`Rfunc(x, z, phi)`
Major radius [meters]
Returns None if in Cartesian coordinates

Parameters  `x, z, phi (array_like)` – X, Z, and toroidal coordinates
Returns  The major radius
Return type  ndarray

attributes = {}
boundary = <zoidberg.boundary.NoBoundary object>

`field_direction(pos, ycoord, flatten=False)`
Calculate the direction of the magnetic field Returns the change in x with phi and change in z with phi

Parameters
- `pos (ndarray)` – 2-D NumPy array, with the second dimension being [x,z], with x and z in meters
- `ycoord (float)` – Toroidal angle in radians if cylindrical coordinates, metres if Cartesian
- `flatten (bool, optional)` – If True, return a flattened form of the vector components. This is useful for passing to `FieldTracer`

Returns  
(dx/dy, dz/dy) –
- = (R*Bx/Bphi, R*Bz/Bphi) if cylindrical
- = (Bx/By, Bz/By) if Cartesian

Return type  list of floats or ndarray

`pressure(x, z, phi)`
Pressure [Pascals]

Parameters  `x, z, phi (array_like)` – X, Z, and toroidal coordinates
Returns  The plasma pressure
Return type  ndarray

class `zoidberg.field.Slab(By=1.0, Bz=0.1, xcentre=0.0, Bzprime=1.0)`
Represents a magnetic field in an infinite flat slab

Magnetic field in z = Bz + (x - xcentre) * Bzprime

Coordinates (x,y,z) assumed to be Cartesian, all in metres

Parameters
• **By** *(float, optional)* – Magnetic field in y direction
• **Bz** *(float, optional)* – Magnetic field in z at xcentre
• **xcentre** *(float, optional)* – Reference x coordinate
• **Bzprime** *(float, optional)* – Rate of change of Bz with x

**Bxfunc**(*x, z, phi*)
Magnetic field in x direction at given coordinates

**Parameters**  
* x, z, phi *(array_like)* – X, Z, and toroidal coordinates

**Returns**  
X-component of the magnetic field

**Return type**  
ndarray

**Byfunc**(*x, z, phi*)
Magnetic field in y direction at given coordinates

**Parameters**  
* x, z, phi *(array_like)* – X, Z, and toroidal coordinates

**Returns**  
Y-component of the magnetic field

**Return type**  
ndarray

**Bzfunc**(*x, z, phi*)
Magnetic field in z direction at given coordinates

**Parameters**  
* x, z, phi *(array_like)* – X, Z, and toroidal coordinates

**Returns**  
Z-component of the magnetic field

**Return type**  
ndarray

class zoidberg.field.SmoothedMagneticField(field, grid, xboundary=None, zboundary=None)
 Represents a magnetic field which is smoothed so it never leaves the boundaries of a given grid.

**Parameters**

• **field** *(zoidberg.field.MagneticField)* – A MagneticField object
• **grid** *(zoidberg.grid.Grid)* – A Grid object
• **xboundary** *(int, optional)* – Number of grid points in x over which the magnetic field is smoothed
• **zboundary** *(int, optional)* – Number of grid points in x over which the magnetic field is smoothed

**Bxfunc**(*x, z, phi*)
Magnetic field in x direction at given coordinates

**Parameters**  
* x, z, phi *(array_like)* – X, Z, and toroidal coordinates

**Returns**  
X-component of the magnetic field

**Return type**  
ndarray

**Byfunc**(*x, z, phi*)
Not modified by smoothing

**Rfunc**(*x, z, phi*)
Major radius [meters]

**Returns**  
None if in Cartesian coordinates

**Parameters**  
* x, z, phi *(array_like)* – X, Z, and toroidal coordinates

**Returns**  
The major radius
**Return type**  ndarray

**smooth_field_line**\((x_a, z_a)\)
Linearly damp the field to be parallel to the edges of the box
Should take some parameters to adjust rate of smoothing, etc.

**class**  \texttt{zoidberg.field.StraightStellarator}(*\texttt{args}, **\texttt{kwargs})*
Invalid StraightStellarator, since no Sympy module.
Rather than printing an error on startup, which may be missed or ignored, this raises an exception if StraightStellarator is ever used.

**class**  \texttt{zoidberg.field.VMEC}(\texttt{vmec\_file}, \texttt{ntheta}=None, \texttt{nzeta}=None, \texttt{nr}=32, \texttt{nz}=32)
A numerical magnetic field from a VMEC equilibrium file

**Parameters**
- **\texttt{vmec\_file}** (str) – Name of the VMEC file to read
- **\texttt{ntheta}** (int, optional) – Number of theta points to use (default: use ‘mpol’ from VMEC file)
- **\texttt{nzeta}** (int, optional) – Number of zeta points to use (default: use ‘ntor’ from VMEC file)
- **\texttt{nr}** (int) – Number of R points to use
- **\texttt{nz}** (int) – Number of Z points to use

**Bxfunc**\((x, z, \phi)\)
Magnetic field in x direction at given coordinates

**Parameters**  \(x, z, \phi\) (array\_like) – \(X, Z, \) and toroidal coordinates

**Returns**  X-component of the magnetic field

**Return type**  ndarray

**Byfunc**\((x, z, \phi)\)
Magnetic field in y direction at given coordinates

**Parameters**  \(x, z, \phi\) (array\_like) – \(X, Z, \) and toroidal coordinates

**Returns**  Y-component of the magnetic field

**Return type**  ndarray

**Bzfunc**\((x, z, \phi)\)
Magnetic field in z direction at given coordinates

**Parameters**  \(x, z, \phi\) (array\_like) – \(X, Z, \) and toroidal coordinates

**Returns**  Z-component of the magnetic field

**Return type**  ndarray

**Rfunc**\((x, z, \phi)\)
Major radius

**cfunct**\((\text{field})\)
VMEC DCT

**read_vmec\_file**(\texttt{vmec\_file}, \texttt{ntheta}=None, \texttt{nzeta}=None)
Read a VMEC equilibrium file

**sfunct**\((\text{field})\)
VMEC DST
K.6.5 zoidberg.fieldtracer module

```python
class zoidberg.fieldtracer.FieldTracer(field):
    A class for following magnetic field lines
    Parameters
    field (MagneticField) – A Zoidberg MagneticField instance

    follow_field_lines(x_values, z_values, y_values, rtol=None)
    Uses field_direction to follow the magnetic field from every grid (x,z) point at toroidal angle y through a
to change in toroidal angle dy

    Parameters
    • x_values (array_like) – Starting x coordinates
    • z_values (array_like) – Starting z coordinates
    • y_values (array_like) – y coordinates to follow the field line to. y_values[0] is the
      starting position
    • rtol (float, optional) – The relative tolerance to use for the integrator. If None, use
      the default value

    Returns
    result – Field line ending coordinates
    The first dimension is y, the last is (x,z). The middle dimensions are the same shape
    as [x|z]: [0,...] is the initial position [...0] are the x-values [...,1] are the z-values If
    x_values is a scalar and z_values a 1D array, then result has the shape [len(y), len(z), 2],
    and vice-versa. If x_values and z_values are 1D arrays, then result has the shape [len(y),
    len(x), 2]. If x_values and z_values are 2D arrays, then result has the shape [len(y),
    x.shape[0], x.shape[1], 2].

    Return type
    numpy.ndarray
```

```python
class zoidberg.fieldtracer.FieldTracerReversible(field, rtol=1e-08, eps=1e-05, nsteps=20)
    Traces magnetic field lines in a reversible way by using trapezoidal integration:

    \[
    pos_{n+1} = pos_n + 0.5 \times (f(pos_n) + f(pos_{n+1})) \times dy
    \]

    This requires a Newton iteration to solve the nonlinear set of equations for the unknown pos_{n+1}.

    Parameters
    • field (MagneticField) – A Zoidberg MagneticField instance
    • rtol (float, optional) – Tolerance applied to changes in dx**2 + dz**2
    • eps (float, optional) – Change in x,z used to calculate finite differences of magnetic field
direction
    • nsteps (int, optional) – Number of sub-steps between outputs

    follow_field_lines(x_values, z_values, y_values, rtol=None, eps=None, nsteps=None)
    Uses field_direction to follow the magnetic field from every grid (x,z) point at toroidal angle y through a
change in toroidal angle dy

    Parameters
    • x_values (array_like) – Starting x coordinates
    • z_values (array_like) – Starting z coordinates
    • y_values (array_like) – y coordinates to follow the field line to. y_values[0] is the
      starting position
```
• **rtol** (*float, optional*) – Tolerance applied to changes in $dx^2 + dz^2$. If None, use the default value

• **eps** (*float, optional*) – Change in x,z used to calculate finite differences of magnetic field direction

• **nsteps** (*int, optional*) – Number of sub-steps between outputs

**Returns**

**result** – Field line ending coordinates

The first dimension is y, the last is (x,z). The middle dimensions are the same shape as [x|z]: [0,...] is the initial position [...,0] are the x-values [...,1] are the z-values If x_values is a scalar and z_values a 1D array, then result has the shape [len(y), len(z), 2], and vice-versa. If x_values and z_values are 1D arrays, then result has the shape [len(y), len(x), 2]. If x_values and z_values are 2D arrays, then result has the shape [len(y), x.shape[0], x.shape[1], 2].

**Return type** `numpy.ndarray`

`zoidberg.fieldtracer.trace_poincare(magnetic_field, xpos, zpos, yperiod, nplot=3, y_slices=None, revs=20, nover=20)`

Trace a Poincare graph of the field lines

Does no plotting, see `zoidberg.plot.plot_poincare()`

**Parameters**

• **magnetic_field** (*MagneticField*) – Magnetic field object

• **xpos, zpos** (*array_like*) – Starting X, Z locations

• **yperiod** (*float*) – Length of period in y domain

• **nplot** (*int, optional*) – Number of equally spaced y-slices to trace to

• **y_slices** (*list of ints*) – List of y-slices to plot; overrides nplot

• **revs** (*int, optional*) – Number of revolutions (times around y)

• **nover** (*int, optional*) – Over-sample. Produced additional points in y then discards. This seems to be needed for accurate results in some cases

**Returns**

coords is a Numpy array of data:

```
[revs, nplot, ..., R/Z]
```

where the first index is the revolution, second is the y slice, and last is 0 for R, 1 for Z. The middle indices are the shape of the input xpos,zpos

**Return type** `coords, y_slices`
K.6.6 zoidberg.grid module

class zoidberg.grid.Grid(poloidal_grids, ycoords, Ly, yperiodic=False, name='fci_grid')

Represents a 3D grid, consisting of a collection of poloidal grids

shape
Tuples of grid sizes (nx, ny, nz)

Parameters

• poloidal_grids (list of PoloidalGrid) – The collection of poloidal grids to group together

• ycoords (array_like) – The y-coordinate corresponding to each element of poloidal_grids

Examples

>>> poloidal_grids = [RectangularPoloidalGrid(5, 5, 1, 1)]
>>> ycoords = [0.0]
>>> grid = Grid(poloidal_grids, ycoords)

To iterate over the poloidal grids, and get the grids to either side:

>>> for i in range(grid.numberOfPoloidalGrids()):
...    pol, y = grid.getPoloidalGrid(i)
...    pol_next, y_next = grid.getPoloidalGrid(i+1)
...    pol_last, y_last = grid.getPoloidalGrid(i-1)

The getPoloidalGrid method ensures that y_last <= y <= y_next

getPoloidalGrid(yindex)

Returns the poloidal grid and y value at the given y index

This handles negative values and values out of range, if the domain is periodic

Parameters yindex (int) – The desired index in y

Returns

• PoloidalGrid – The poloidal grid at yindex

• float – The value of the y coordinate at yindex

gmetric()

Return the metric tensor, dx and dz

Returns Dictionary containing:
- dx, dy, dz: Grid spacing
- gxx, gxz, gyy, gzz: Covariant components
- g_xx, g_xz, g_yy, g_zz: Contravariant components

Return type dict

numberOfPoloidalGrids()

Returns the number of poloidal grids i.e. number of points in Y

Returns Number of poloidal grids

Return type int
Create a rectangular grid in (x,y,z)

Here y is along the magnetic field (typically toroidal angle), and (x,z) are in the poloidal plane

**Parameters**

- `nx, ny, nz (int)` – Number of points in x, y, z
- `Lx, Ly, Lz (float, optional)` – Size of the domain in x, y, z
- `xcentre, zcentre (float, optional)` – The middle of the domain
- `yperiodic (bool, optional)` – Determines if the y direction is periodic

**Returns** A Grid representing a rectangular domain

**Return type** Grid

### K.6.7 zoidberg.plot module

**class** zoidberg.plot.AnimateVectorField(X, Y, U, V)

Very basic/experimental class for animating vector fields

Transpose U, V to have dimension to animate at index 0, e.g. to animate along y, pass:

```python
>>> AnimateVectorField(X, Y, U.transpose((1,0,2)), V.transpose((1,0,2)))
```

**Parameters**

- `X, Y (array_like)` – The X, Y coordinates
- `U, V (ndarray)` – Vector components in X, Y respectively

**Examples**

```python
>>> anim = AnimateVectorField(X, Y, U, V)
>>> anim.animate()
```

**animate()**

**zoidberg.plot.plot_3d_field_line**(magnetic_field, xpos, zpos, yperiod, cycles=20, y_res=50)

Make a 3D plot of field lines

**Parameters**

- `magnetic_field (zoidberg.field.MagneticField)` – Magnetic field object
- `xpos, zpos (array_like)` – Starting X, Z locations
- `yperiod (float)` – Length of period in y domain
- `cycles (int, optional)` – Number of times to go round in y
- `y_res (int, optional)` – Number of points in y in each cycle

**Returns** The matplotlib figure and axis used

**Return type** fig, ax

**zoidberg.plot.plot_backward_map**(grid, maps, yslice=0)

Plots the backward map from yslice to yslice-1

**Parameters**
zoidberg.plot.plot_forward_map(grid, maps, yslice=0)
Plots the forward map from yslice to yslice + 1

Parameters
• grid (zoidberg.grid.Grid) – Grid generated by Zoidberg
• maps (dict) – Dictionary containing the forward FCI maps
• y_slice (int, optional) – Originating y-index to plot map from

zoidberg.plot.plot_poincare(magnetic_field, xpos, zpos, yperiod, nplot=3, y_slices=None, revs=40, nover=20, interactive=False)
Plot a Poincare graph of the field lines.

Parameters
• magnetic_field (zoidberg.field.MagneticField) – Magnetic field object
• xpos, zpos (array_like) – Starting X, Z locations
• yperiod (float) – Length of period in y domain
• nplot (int, optional) – Number of equally spaced y-slices to plot
• y_slices (list of int, optional) – List of y-slices to plot; overrides nplot
• revs (int, optional) – Number of revolutions (times around phi)
• interactive (bool, optional) – If True, plots can be interacted with via the mouse: - Left-click on the plot to trace a field-line from that point - Right-click to add an additional trace - Middle-click to clear added traces

Returns The matplotlib figure and axis used

Return type fig, ax

zoidberg.plot.plot_streamlines(grid, magnetic_field, y_slice=0, width=None, **kwargs)
Plot streamlines of the magnetic field in the poloidal plane

Parameters
• grid (zoidberg.grid.Grid) – Grid generated by Zoidberg
• magnetic_field (zoidberg.field.MagneticField) – Zoidberg magnetic field object
• y_slice (int, optional) – y-index to plot streamlines at
• width (float, optional) – If not None, line widths are proportional to the magnitude of the magnetic_field times width

Returns The matplotlib figure and axis used

Return type fig, ax
K.6.8 zoidberg.poloidal_grid module

Routines for generating structured meshes on poloidal domains

Classes

RectangularPoloidalGrid  Simple rectangles in R-Z

StructuredPoloidalGrid  Curvilinear structured grids in R-Z

Functions

grid_annulus  Create a StructuredPoloidalGrid from inner and outer RZLine objects using a simple algorithm

grid_elliptic  Create a StructuredPoloidalGrid from inner and outer RZLine objects using elliptic meshing method

class zoidberg.poloidal_grid.PoloidalGrid
  Represents a poloidal grid
  Note: Here the 2D plane (R,Z) is labelled by (x,z) indices
  nx, nz  
    Number of points in x and z
    Type int
  R  
    2D Numpy array of R coordinates
    Type ndarray
  Z  
    2D Numpy array of Z coordinates
    Type ndarray
  plot(axis=None, show=True)
    Plot grid using matplotlib
    Parameters
      • axis (matplotlib axis, optional) – A matplotlib axis to plot on. By default a new figure is created
      • show (bool, optional) – Calls plt.show() at the end
    Returns
      The matplotlib axis that was used
    Return type axis

class zoidberg.poloidal_grid.RectangularPoloidalGrid(nx, nz, Lx, Lz, Rcentre=0.0, Zcentre=0.0, MXG=2)
  Represents a poloidal grid consisting of a rectangular domain
  Note: Here the 2D plane (R,Z) is labelled by (x,z) indices
  nx, nz  
    Number of points in x and z
    Type int
R
2D Numpy array of R coordinates
Type ndarray

Z
2D Numpy array of Z coordinates
Type ndarray

Parameters
• nx (int) – Number of points in major radius (including boundaries)
• nz (int) – Number of points in height (including boundaries)
• Lx (float) – Radial domain size [m]
• Lz (float) – Vertical domain size [m]
• Rcentre (float, optional) – Coordinate at the middle of the domain
• Zcentre (float, optional) – Coordinate at the middle of the domain
• MXG (int, optional) – Number of guard cells in X. The boundary is put half-way between the guard cell and the domain

findIndex(R, Z)
Finds the (x,z) index corresponding to the given (R,Z) coordinate

Parameters R, Z (array_like) – Locations to find indices for

Returns x, z – Index as a float, same shape as R,Z

Return type (ndarray, ndarray)

getCoordinate(xind, zind, dx=0, dz=0)
Get coordinates (R,Z) at given (xind,zind) index

Parameters
• xind, zind (array_like) – Indices in X and Z. These should be the same shape
• dx (int, optional) – Order of x derivative
• dz (int, optional) – Order of z derivative

Returns R, Z – Locations of point or derivatives of R,Z with respect to indices if dx,dz != 0

Return type (ndarray, ndarray)

metric()
Return the metric tensor, dx and dz

For this rectangular grid the metric is the identity

Returns Dictionary containing: - dx, dz: Grid spacing - gxx, gxz, gzz: Covariant components - g_xx, g_xz, g_zz: Contravariant components

Return type dict

class zoidberg.poloidal_grid.StructuredPoloidalGrid(R, Z)
Represents a structured poloidal grid in R-Z

nx, nz
Number of points in x and z
Type int
R
2D Numpy array of R coordinates

Type ndarray

Z
2D Numpy array of Z coordinates

Type ndarray

Parameters R, Z (ndarray) – 2D Numpy arrays of R,Z points

Note: R,Z are not copied, so these arrays should not be modified afterwards

findIndex(R, Z, tol=1e-10, show=False)
Finds the (x, z) index corresponding to the given (R, Z) coordinate

Parameters
• R, Z (array_like) – Locations. Can be scalar or array, must be the same shape
• tol (float, optional) – Maximum tolerance on the square distance

Returns x, z – Index as a float, same shape as R, Z

Return type (ndarray, ndarray)

getCoordinate(xind, zind, dx=0, dz=0)
Get coordinates (R, Z) at given (xind, zind) index

Parameters
• xind, zind (array_like) – Indices in X and Z. These should be the same shape
• dx (int, optional) – Order of x derivative
• dz (int, optional) – Order of z derivative

Returns R, Z – Locations of point or derivatives of R,Z with respect to indices if dx,dz != 0

Return type (ndarray, ndarray)

metric()
Return the metric tensor, dx and dz

Returns Dictionary containing: - dx, dz: Grid spacing - gxx, gxz, gzz: Covariant components - g_xx, g_xz, g_zz: Contravariant components

Return type dict

zoidberg.poloidal_grid.grid_annulus(inner, outer, nx, nz, show=True, return_coords=False)
Grid an annular region, given inner and outer boundaries both of which are RZline objects

This is a very simple algorithm which just draws straight lines between inner and outer boundaries.

Parameters
• inner, outer (RZline) – Inner and outer boundaries of the domain
• nx (int) – The required radial resolution, including boundaries
• nz (int) – The required poloidal resolution
• show (bool, optional) – If True, plot the resulting grid
• return_coords (bool, optional) – If True, return the R, Z coordinates of the grid points, instead of a StructuredPoloidalGrid
Returns  A grid of the region

Return type  StructuredPoloidalGrid

zoidberg.poloidal_grid.grid_elliptic(inner, outer, nx, nz, show=False, tol=1e-10, align=True,
restrict_size=20, restrict_factor=2, return_coords=False)

Create a structured grid between inner and outer boundaries using elliptic method

Coordinates x = x(R, Z) and z = z(R, Z) obey an elliptic equation:

\[
\frac{d^2 x}{dR^2} + \frac{d^2 x}{dZ^2} = 0
\]
\[
\frac{d^2 z}{dR^2} + \frac{d^2 z}{dZ^2} = 0
\]

where here x is in the domain (0, 1) and z in (0, 2pi)

The above equations are inverted, giving:

\[
a \cdot R_x x - 2 \cdot b \cdot R_x z + c \cdot R_z z = 0
\]
\[
a \cdot Z_x x - 2 \cdot b \cdot Z_x z + c \cdot Z_z z = 0
\]

where

\[
a = R_x^2 + Z_x^2
\]
\[
b = R_x \cdot R_z + Z_x \cdot Z_z
\]
\[
c = R_z^2 + Z_z^2
\]

This is a nonlinear system of equations which is solved iteratively.

Parameters

- **inner, outer** (RZline) – Inner and outer boundaries of the domain
- **nx** (int) – The required radial resolution, including boundaries
- **nz** (int) – The required poloidal resolution
- **show** (bool, optional) – Display plots of intermediate results
- **tol** (float, optional) – Controls when iteration stops
- **align** (bool, optional) – Attempt to align the inner and outer boundaries
- **restrict_size** (int, optional) – The size (nx or nz) above which the grid is coarsened
- **restrict_factor** (int, optional) – The factor by which the grid is divided if coarsened
- **return_coords** (bool, optional) – If True, return the R, Z coordinates of the grid points, instead of a StructuredPoloidalGrid

Returns

- If return_coords is true, returns R, Z as arrays.
- If return_coords is false, returns a StructuredPoloidalGrid object
K.6.9 zoidberg.progress module

K.6.10 zoidberg.rzline module

Routines and classes for representing periodic lines in R-Z poloidal planes

class zoidberg.rzline.RZline(r, z, anticlockwise=True)

Represents (R,Z) coordinates of a periodic line

R
  Major radius [m]
  Type array_like

Z
  Height [m]
  Type array_like

theta
  Angle variable [radians]
  R, Z and theta all have the same length
  Type array_like

Parameters
  • r, z (array_like) – 1D arrays of the major radius (r) and height (z) which are of the same length. A periodic domain is assumed, so the last point connects to the first.
  • anticlockwise (bool, optional) – Ensure that the line goes anticlockwise in the R-Z plane (positive theta)
  • Note that the last point in (r,z) arrays should not be the same
  • as the first point. The (r,z) points are in [0,2pi)
  • The input r,z points will be reordered, so that the theta angle goes anticlockwise in the R-Z plane

Rvalue(theta=None, deriv=0)
  Calculate the value of R at given theta locations

Parameters
  • theta (array_like, optional) – Theta locations to find R at. If None (default), use the values of theta stored in the instance
  • deriv (int, optional) – The order of derivative to compute (default is just the R value)

Returns
  Value of R at each input theta point

Return type
  ndarray

Zvalue(theta=None, deriv=0)
  Calculate the value of Z at given theta locations
Parameters

- **theta** (array_like, optional) – Theta locations to find Z at. If None (default), use the values of theta stored in the instance
- **deriv** (int, optional) – The order of derivative to compute (default is just the Z value)

Returns Value of Z at each input theta point

Return type ndarray

closestPoint(R, Z, niter=3, subdivide=20)

Find the closest point on the curve to the given (R,Z) point

Parameters

- **R, Z** (float) – The input R, Z point
- **niter** (int, optional) – How many iterations to use

Returns The value of theta (angle)

Return type float

distance(sample=20)

Integrates the distance along the line.

Parameters **sample** (int, optional) – Number of samples to take per point

Returns

- An array one longer than theta. The first element is zero,
- and the last element is the total distance around the loop

equallySpaced(n=None)

Returns a new RZline which has a theta uniform in distance along the line

Parameters **n** (int, optional) – Number of points. Default is the same as the current line

Returns A new RZline based on this instance, but with uniform theta-spacing

Return type RZline

plot(axis=None, show=True)

Plot the RZline, either on the given axis or a new figure

Parameters

- **axis** (matplotlib axis, optional) – A matplotlib axis to plot on. By default a new figure is created
- **show** (bool, optional) – Calls plt.show() at the end

Returns The matplotlib axis that was used

Return type axis

position(theta=None)

Calculate the value of both R, Z at given theta locations

Parameters **theta** (array_like, optional) – Theta locations to find R, Z at. If None (default), use the values of theta stored in the instance

Returns **R, Z** – Value of R, Z at each input theta point

Return type (ndarray, ndarray)
**positionPolygon**(theta=None)
Calculates (R,Z) position at given theta angle by joining points by straight lines rather than a spline. This avoids the overshoots which can occur with splines.

**Parameters**
theta *(array_like, optional)* – Theta locations to find R, Z at. If None (default), use the values of theta stored in the instance

**Returns**
R, Z – Value of R, Z at each input theta point

**Return type** *(ndarray, ndarray)*

**zoidberg.rzline.circle**(R0=1.0, r=0.5, n=20)
creates a pair of RZline objects, for inner and outer boundaries

**Parameters**
- R0 *(float, optional)* – Centre point of the circle
- r *(float, optional)* – Radius of the circle
- n *(int, optional)* – Number of points to use in the boundary

**Returns**
A circular RZline

**Return type** RZline

**zoidberg.rzline.line_from_points**(rarray, zarray, show=False)
Find a periodic line which goes through the given (r,z) points

This function starts at a point, and finds the nearest neighbour which is not already in the line

**Parameters**
rarray, zarray *(array_like)* – R, Z coordinates. These arrays should be the same length

**Returns**
An RZline object representing a periodic line

**Return type** RZline

**zoidberg.rzline.line_from_points_poly**(rarray, zarray, show=False)
Find a periodic line which goes through the given (r,z) points

This function starts with a triangle, then adds points one by one, inserting into the polygon along the nearest edge

**Parameters**
rarray, zarray *(array_like)* – R, Z coordinates. These arrays should be the same length

**Returns**
An RZline object representing a periodic line

**Return type** RZline

**zoidberg.rzline.shaped_line**(R0=3.0, a=1.0, elong=0.0, triang=0.0, indent=0.0, n=20)
Parametrisation of plasma shape from J. Manickam, Nucl. Fusion 24 595 (1984)

**Parameters**
- R0 *(float, optional)* – Major radius
- a *(float, optional)* – Minor radius
- elong *(float, optional)* – Elongation, 0 for a circle
- triang *(float, optional)* – Triangularity, 0 for a circle
- indent *(float, optional)* – Indentation, 0 for a circle

**Returns**
An RZline matching the given parameterisation

**Return type** RZline
K.6.11 zoidberg.test_field module
K.6.12 zoidberg.test_fieldtracer module
K.6.13 zoidberg.test_grid module
K.6.14 zoidberg.test_poloidal_grid module
K.6.15 zoidberg.test_rzline module
K.6.16 zoidberg.test_zoidberg module
K.6.17 zoidberg.zoidberg module
K.6.18 Module contents
INDICES AND TABLES

- genindex
- search
[Dudson2009] https://doi.org/10.1016/j.cpc.2009.03.008


b
boutcore, 136

Z
zoidberg.boundary, 759
zoidberg.field, 760
zoidberg.fieldtracer, 766
zoidberg.grid, 768
zoidberg.plot, 769
zoidberg.poloidal_grid, 771
zoidberg.rzline, 775
Symbols

_ (C macro), 488
(GET_FOR_EACH_EXPANSION (C macro), 558
_BOUT_GENERIC_FACTORY_H__ (C macro), 487
_FIELD2D_H__ (C macro), 429
_FIELD3D_H__ (C macro), 436
_FIELD_DATA_H__ (C macro), 455
_OPTIONS_H__ (C macro), 609
_OPTIONS_NETCDF_H__ (C macro), 619
_OUTPUT_H__ (C macro), 621
_RVEC_H__ (C macro), 677
_VARIANT_HXX__ (C macro), 739
_VECTOR2D_H__ (C macro), 746
_VECTOR3D_H__ (C macro), 751
_thefunc__ (C macro), 578
_ec_expand_1 (C macro), 321
_ec_expand_10 (C macro), 322
_ec_expand_2 (C macro), 321
_ec_expand_3 (C macro), 321
_ec_expand_4 (C macro), 321
_ec_expand_5 (C macro), 322
_ec_expand_6 (C macro), 322
_ec_expand_7 (C macro), 322
_ec_expand_8 (C macro), 322
_ec_expand_9 (C macro), 322
_fe_1 (C macro), 558
_fe_10 (C macro), 558
_fe_2 (C macro), 558
_fe_3 (C macro), 558
_fe_4 (C macro), 558
_fe_5 (C macro), 558
_fe_6 (C macro), 558
_fe_7 (C macro), 558
_fe_8 (C macro), 558
_fe_9 (C macro), 558
_me_1 (C macro), 557
_me_10 (C macro), 557
_me_2 (C macro), 557
_me_3 (C macro), 557
_me_4 (C macro), 557
_me_5 (C macro), 557
_me_6 (C macro), 557

A

abs (C++ function), 449, 746, 747, 750, 751
action (gen_fieldops attribute), 482
AdamsBashforthSolver (C++ class), 285
AdamsBashforthSolver::AdamsBashforthSolver (C++ function), 285
AdamsBashforthSolver::AdamsBashforthSolver (C++ function), 285
AdamsBashforthSolver::adaptive (C++ member), 286
AdamsBashforthSolver::adaptive_order (C++ member), 286
AdamsBashforthSolver::atol (C++ member), 286
AdamsBashforthSolver::current_order (C++ member), 286
AdamsBashforthSolver::dtFac (C++ member), 286
AdamsBashforthSolver::followHighOrder (C++ member), 286
AdamsBashforthSolver::getCurrentTimestep (C++ function), 285
AdamsBashforthSolver::history (C++ member), 286
AdamsBashforthSolver::init (C++ function), 285
AdamsBashforthSolver::max_timestep (C++ member), 286
AdamsBashforthSolver::maximum_order (C++ member), 286
AdamsBashforthSolver::mxstep (C++ member), 286
AdamsBashforthSolver::neq (C++ member), 286
AdamsBashforthSolver::nextState (C++ member), 286
AdamsBashforthSolver::nlocal (C++ member), 286
AdamsBashforthSolver::nsteps (C++ member), 286
AdamsBashforthSolver::out_timestep (C++ member), 286
AdamsBashforthSolver::resetInternalFields (C++ function), 285
AdamsBashforthSolver::rtol (C++ member), 286
ArkodeSolver::run (C++ function), 285
ArkodeSolver::setMaxTimestep (C++ function), 285
ArkodeSolver::setState (C++ member), 286
ArkodeSolver::takeStep (C++ function), 286
ArkodeSolver::times (C++ member), 286
AdamsBashforthSolver::times (C++ member), 286
add (C++ function), 411
Add (C++ struct), 416
addItemToDeferredFunction (C++ function), 715
animate () (zoidberg.plot/animate() method), 769
AnimateVectorField (class in zoidberg.plot), 769
areFieldsCompatible (C++ function), 322, 326
areFieldsCompatible (C++ function), 447
args (gen_fields attribute), 482
arkode_bbd_rhs (C++ function), 287
arkode_jac (C++ function), 287
arkode_pre (C++ function), 287
arkode_preShim (C++ function), 287
arkode rhs (C++ function), 287
arkode rhs_explicit (C++ function), 287
arkode rhs_implicit (C++ function), 287
ARKODEINT (C++ type), 287
ArkodeSolver (C++ class), 288
ArkodeSolver::~ArkodeSolver (C++ function), 288
ArkodeSolver::arkode_mem (C++ member), 289
ArkodeSolver::diagnose (C++ function), 289
ArkodeSolver::getCurrentTimeStep (C++ function), 288
ArkodeSolver::hcur (C++ member), 289
ArkodeSolver::init (C++ function), 288
ArkodeSolver::jac (C++ function), 289
ArkodeSolver::loop_abstol_values_op (C++ function), 289
ArkodeSolver::nfe_evals (C++ member), 289
ArkodeSolver::nfi_evals (C++ member), 289
ArkodeSolver::nlters (C++ member), 289
ArkodeSolver::nniters (C++ member), 289
ArkodeSolver::NOUT (C++ member), 289
ArkodeSolver::npevals (C++ member), 289
ArkodeSolver::nsteps (C++ member), 289
ArkodeSolver::pre_Wtime (C++ function), 289
AdamsBashforthSolver::setMaxTimestep (C++ function), 288
AdamsBashforthSolver::rhs (C++ function), 289
AdamsBashforthSolver::rhs_e (C++ function), 289
AdamsBashforthSolver::rhs_i (C++ function), 289
AdamsBashforthSolver::run (C++ function), 288
ArkodeSolver::set_abstol_values (C++ function), 289
ArkodeSolver::Timestep (C++ member), 289
ArkodeSolver::uvec (C++ member), 289
ARKStepCreate (C++ function), 287
ARKStepEvolve (C++ member), 287
ARKStepFree (C++ member), 287
ARKStepGetCurrentTime (C++ member), 287
ARKStepGetDyk (C++ member), 287
ARKStepGetLastStep (C++ member), 287
ARKStepGetNumLinIters (C++ member), 287
ARKStepGetNumNonlinSolvIters (C++ member), 287
ARKStepGetNumPrecEvals (C++ member), 287
ARKStepGetNumRhsSolvEvals (C++ member), 287
ARKStepGetNumSteps (C++ member), 287
ARKStepReInit (C++ member), 287
ARKStepSetAdaptivityMethod (C++ member), 288
ARKStepSetCFLFraction (C++ member), 288
ARKStepSetEpsLin (C++ member), 288
ARKStepSetExplicit (C++ member), 288
ARKStepSetFixedPoint (C++ member), 288
ARKStepSetFixedStep (C++ member), 288
ARKStepSetImEx (C++ member), 288
ARKStepSetImplicit (C++ member), 288
ARKStepSetInitStep (C++ member), 288
ARKStepSetJacTimes (C++ function), 287
ARKStepSetLinear (C++ member), 288
ARKStepSetMaxNumSteps (C++ member), 288
ARKStepSetMaxStep (C++ member), 288
ARKStepSetMinStep (C++ member), 288
ARKStepSetOptimalParams (C++ member), 288
ARKStepSetOrder (C++ member), 288
ARKStepSetPreconditioner (C++ member), 288
ARKStepSetUserData (C++ member), 288
ARKStepSTolerances (C++ member), 287
ARKStepSVtolerances (C++ member), 288
Array (C++ class), 290
Array::~Array (C++ function), 291
Array::arenaType (C++ type), 292
Array::Array (C++ function), 291
Array::backing_type (C++ type), 291
Array::begin (C++ function), 292
Array::cleanup (C++ function), 292
Array::clear (C++ function), 291
Array::data_type (C++ type), 291
Array::dataBlock (C++ type), 292
Array::dataPtrType (C++ type), 292
Array::empty (C++ function), 291
Array::end (C++ function), 292
Array::ensureUnique (C++ function), 291
Array::get (C++ function), 292
Array::operator= (C++ function), 291
Array::operator[] (C++ function), 292
Array::ptr (C++ member), 293
BoundaryDirichlet_2ndOrder::apply (C++ function), 303
BoundaryDirichlet_2ndOrder::BoundaryDirichlet_2ndOrder (C++ function), 303
BoundaryDirichlet_2ndOrder::clone (C++ function), 303
BoundaryDirichlet_2ndOrder::val (C++ member), 304
BoundaryDirichlet_4thOrder (C++ class), 306
BoundaryDirichlet_4thOrder::apply (C++ function), 306
BoundaryDirichlet_4thOrder::apply_ddt (C++ function), 306
BoundaryDirichlet_4thOrder::BoundaryDirichlet_4thOrder (C++ function), 306
BoundaryDirichlet_4thOrder::clone (C++ function), 306
BoundaryDirichlet_4thOrder::val (C++ member), 306
BoundaryDirichlet_03 (C++ class), 304
BoundaryDirichlet_03::apply (C++ function), 305
BoundaryDirichlet_03::apply_ddt (C++ function), 305
BoundaryDirichlet_03::BoundaryDirichlet_03 (C++ function), 305
BoundaryDirichlet_03::clone (C++ function), 305
BoundaryDirichlet_03::gen (C++ member), 305
BoundaryDirichlet_04 (C++ class), 305
BoundaryDirichlet_04::apply (C++ function), 305
BoundaryDirichlet_04::apply_ddt (C++ function), 305, 306
BoundaryDirichlet_04::BoundaryDirichlet_04 (C++ function), 305
BoundaryDirichlet_04::clone (C++ function), 305
BoundaryDirichlet_04::gen (C++ member), 306
BoundaryDivCurl (C++ class), 312
BoundaryDivCurl::apply (C++ function), 312
BoundaryDivCurl::BoundaryDivCurl (C++ function), 312
BoundaryDivCurl::clone (C++ function), 312
BoundaryFactory (C++ class), 294
BoundaryFactory::~BoundaryFactory (C++ function), 295
BoundaryFactory::add (C++ function), 295
BoundaryFactory::addMod (C++ function), 295
BoundaryFactory::BoundaryFactory (C++ function), 295
BoundaryFactory::cleanup (C++ function), 295
BoundaryFactory::create (C++ function), 295
BoundaryFactory::createFromOptions (C++ function), 295
BoundaryFactory::findBoundaryMod (C++ function), 295
BoundaryFactory::findBoundaryOp (C++ function), 295
BoundaryFactory::findBoundaryOpPar (C++ function), 295
BoundaryFactory::getInstance (C++ function), 295
BoundaryFactory::instance (C++ member), 296
BoundaryFactory::modmap (C++ member), 296
BoundaryFactory::opmap (C++ member), 296
BoundaryFactory::par_opmap (C++ member), 296
BoundaryFree (C++ class), 313
BoundaryFree::apply (C++ function), 313
BoundaryFree::apply_ddt (C++ function), 313
BoundaryFree::BoundaryFree (C++ function), 313
BoundaryFree::clone (C++ function), 313
BoundaryFree::val (C++ member), 313
BoundaryFree_02 (C++ class), 313
BoundaryFree_02::apply (C++ function), 313
BoundaryFree_02::apply_ddt (C++ function), 313
BoundaryFree_02::BoundaryFree_02 (C++ function), 313
BoundaryFree_03 (C++ class), 314
BoundaryFree_03::apply (C++ function), 314
BoundaryFree_03::apply_ddt (C++ function), 314
BoundaryFree_03::BoundaryFree_03 (C++ function), 314
BoundaryFree_03::clone (C++ function), 314
BoundaryFromFieldAligned (C++ class), 316
BoundaryFromFieldAligned::apply (C++ function), 316
BoundaryFromFieldAligned::apply_ddt (C++ function), 316
BoundaryFromFieldAligned::BoundaryFromFieldAligned (C++ function), 316
BoundaryModifier (C++ class), 297
BoundaryModifier::BoundaryModifier (C++ function), 297
BoundaryModifier::cloneMod (C++ function), 297
BoundaryModifier::cloneMod (C++ function), 297
BoundaryNeumann (C++ class), 308
BoundaryNeumann2 (C++ class), 307
BoundaryNeumann2::apply (C++ function), 307
BoundaryNeumann2::BoundaryNeumann2 (C++ function), 307
BoundaryNeumann2::clone (C++ function), 307
BoundaryNeumann2::BoundaryNeumann2 (C++ function), 307
BoundaryNeumann2::clone (C++ function), 308
BoundaryNeumann2::gen (C++ member), 309
BoundaryNeumann_2ndOrder (C++ class), 307
BoundaryNeumann_2ndOrder::apply (C++ function), 308
BoundaryNeumann_2ndOrder::apply_ddt (C++ function), 308
BoundaryNeumann_2ndOrder::clone (C++ function), 308
BoundaryNeumann_2ndOrder::val (C++ member), 308
BoundaryNeumann_4thOrder (C++ class), 309
BoundaryNeumann_4thOrder::apply (C++ function), 309
BoundaryNeumann_4thOrder::apply_ddt (C++ function), 309
BoundaryNeumann_4thOrder::clone (C++ function), 309
BoundaryNeumann_4thOrder::val (C++ member), 309
BoundaryNeumann_NonOrthogonal (C++ class), 306
BoundaryNeumann_NonOrthogonal::apply (C++ function), 307
BoundaryNeumann_NonOrthogonal::BoundaryNeumann_NonOrthogonal (C++ function), 307
BoundaryNeumann_NonOrthogonal::clone (C++ function), 307
BoundaryNeumann_NonOrthogonal::val (C++ member), 307
BoundaryNeumann_04 (C++ class), 309
BoundaryNeumann_04::apply (C++ function), 309, 310
BoundaryNeumann_04::apply_ddt (C++ function), 310
BoundaryNeumann_04::BoundaryNeumann_04 (C++ function), 309
BoundaryNeumann_04::clone (C++ function), 309
BoundaryNeumann_04::gen (C++ member), 310
BoundaryNeumannPar (C++ class), 310
BoundaryNeumannPar::apply (C++ function), 310
BoundaryNeumannPar::BoundaryNeumannPar (C++ function), 310
BoundaryNeumannPar::clone (C++ function), 310
BoundaryOp (C++ class), 296
BoundaryOp:::-BoundaryOp (C++ function), 297
BoundaryOp::apply_ddt (C++ function), 297
BoundaryOp::apply_to_ddt (C++ member), 297
BoundaryOp::bdry (C++ member), 297
BoundaryOp::BoundaryOp (C++ function), 297
BoundaryOp::clone (C++ function), 297
BoundaryOp::clone (C++ class), 296
BoundaryOpBase (C++ class), 296
BoundaryOpBase:::-BoundaryOpBase (C++ function), 296
BoundaryOpBase::apply (C++ function), 296
BoundaryOpBase::BoundaryOpBase (C++ function), 296
BoundaryOpPar (C++ class), 626
BoundaryOpPar::apply (C++ function), 626
BoundaryOpPar::bdry (C++ member), 627
BoundaryOpPar::BoundaryOpPar (C++ function), 626
BoundaryOpPar::clone (C++ function), 626
BoundaryOpPar::field_values (C++ member), 627
BoundaryOpPar::gen_values (C++ member), 627
BoundaryOpPar::getValue (C++ function), 627
BoundaryOpPar::real_value (C++ member), 627
BoundaryOpPar::value_type (C++ member), 627
BoundaryOpPar::Value_Type (C++ enum), 627
BoundaryOpPar::Value_Type::FIELD (C++ enumerator), 627
BoundaryOpPar::Value_Type::GEN (C++ enumerator), 627
BoundaryOpPar::Value_Type::REAL (C++ enumerator), 627
BoundaryOpPar_dirichlet (C++ class), 627
BoundaryOpPar_dirichlet::apply (C++ function), 627
BoundaryOpPar_dirichlet::BoundaryOpPar_dirichlet (C++ function), 627, 628
BoundaryOpPar_dirichlet::clone (C++ function), 628
BoundaryOpPar_dirichlet_interp (C++ class), 628
BoundaryOpPar_dirichlet_interp::apply (C++ function), 629
BoundaryOpPar_dirichlet_interp::BoundaryOpPar_dirichlet_interp (C++ function), 629
BoundaryOpPar_dirichlet_interp::clone (C++ function), 629
BoundaryOpPar_dirichlet_03 (C++ class), 628
BoundaryOpPar_dirichlet_03::apply (C++ function), 628
BoundaryOpPar_dirichlet_03::BoundaryOpPar_dirichlet_03 (C++ function), 628
BoundaryOpPar_dirichlet_03::clone (C++ function), 628
BoundaryOpPar_neumann (C++ class), 629
BoundaryOpPar_neumann::apply (C++ function), 629
BoundaryOpPar_neumann::BoundaryOpPar_neumann (C++ function), 629
BoundaryOpPar_neumann::clone (C++ function), 629
BoundaryRegion (C++ class), 299
BoundaryRegion:::-BoundaryRegion (C++ function), 299
BoundaryRegion::BoundaryRegion (C++ function), 299
BoundaryRegion::bx (C++ member), 300

Index 789
BoundaryRegion::by (C++ member), 300
BoundaryRegion::next1d (C++ function), 299
BoundaryRegion::nextX (C++ function), 300
BoundaryRegion::nextY (C++ function), 300
BoundaryRegion::width (C++ member), 300
BoundaryRegion::x (C++ member), 300
BoundaryRegion::y (C++ member), 300
BoundaryRegionBase (C++ class), 298
BoundaryRegionBase::BoundaryRegionBase (C++ function), 299
BoundaryRegionBase::BoundaryRegionBase::BoundaryRegionBase (C++ function), 299
BoundaryRegionBase::first (C++ function), 299
BoundaryRegionBase::isDone (C++ function), 299
BoundaryRegionBase::isParallel (C++ member), 299
BoundaryRegionBase::label (C++ member), 299
BoundaryRegionBase::localmesh (C++ member), 299
BoundaryRegionBase::location (C++ member), 299
BoundaryRegionBase::next (C++ function), 299
BoundaryRegionPar (C++ class), 630
BoundaryRegionPar::add_point (C++ function), 630
BoundaryRegionPar::angle (C++ member), 630
BoundaryRegionPar::bdry_points (C++ member), 631
BoundaryRegionPar::bdry_position (C++ member), 631
BoundaryRegionPar::BoundaryRegionPar (C++ function), 630
BoundaryRegionPar::dir (C++ member), 630
BoundaryRegionPar::first (C++ function), 630
BoundaryRegionPar::IndexPoint::C++ struct), 631
BoundaryRegionPar::IndexPoint::jx (C++ member), 631
BoundaryRegionPar::IndexPoint::jy (C++ member), 631
BoundaryRegionPar::IndexPoint::jz (C++ member), 631
BoundaryRegionPar::Indices (C++ struct), 631
BoundaryRegionPar::Indices::angle (C++ member), 631
BoundaryRegionPar::Indices::index (C++ member), 631
BoundaryRegionPar::Indices::intersection (C++ member), 631
BoundaryRegionPar::Indices::length (C++ member), 631
BoundaryRegionPar::IndicesIter (C++ type), 631
BoundaryRegionPar::IndicesVec (C++ type), 631
BoundaryRegionPar::isDone (C++ function), 630
BoundaryRegionPar::length (C++ member), 630
BoundaryRegionPar::next (C++ function), 630
BoundaryRegionPar::RealPoint (C++ struct), 631
BoundaryRegionPar::RealPoint::s_x (C++ member), 631
BoundaryRegionPar::RealPoint::s_y (C++ member), 631
BoundaryRegionPar::RealPoint::s_z (C++ member), 631
BoundaryRegionPar::s_x (C++ member), 630
BoundaryRegionPar::s_y (C++ member), 630
BoundaryRegionPar::s_z (C++ member), 630
BoundaryRegionPar::x (C++ member), 630
BoundaryRegionPar::y (C++ member), 630
BoundaryRegionPar::z (C++ member), 630
BoundaryRegionXIn (C++ class), 300
BoundaryRegionXIn::BoundaryRegionXIn (C++ function), 300
BoundaryRegionXIn::first (C++ function), 300
BoundaryRegionXIn::isDone (C++ function), 300
BoundaryRegionXIn::next (C++ function), 300
BoundaryRegionXIn::next1d (C++ function), 300
BoundaryRegionXIn::nextX (C++ function), 300
BoundaryRegionXIn::nextY (C++ function), 300
BoundaryRegionXIn::ye (C++ member), 301
BoundaryRegionXIn::ys (C++ member), 301
BoundaryRegionXOut (C++ class), 301
BoundaryRegionXOut::BoundaryRegionXOut (C++ function), 301
BoundaryRegionXOut::first (C++ function), 301
BoundaryRegionXOut::isDone (C++ function), 301
BoundaryRegionXOut::next (C++ function), 301
BoundaryRegionXOut::next1d (C++ function), 301
BoundaryRegionXOut::nextX (C++ function), 301
BoundaryRegionXOut::nextY (C++ function), 301
BoundaryRegionXOut::ye (C++ member), 301
BoundaryRegionXOut::ys (C++ member), 301
BoundaryRegionYDown (C++ class), 301
BoundaryRegionYDown::BoundaryRegionYDown (C++ function), 301
BoundaryRegionYDown::first (C++ function), 301
BoundaryRegionYDown::isDone (C++ function), 301
BoundaryRegionYDown::next (C++ function), 301
BoundaryRegionYDown::next1d (C++ function), 301
BoundaryRegionYDown::nextX (C++ function), 301
BoundaryRegionYDown::nextY (C++ function), 301
BoundaryRegionYDown::xe (C++ member), 302
BoundaryRegionYDown::xs (C++ member), 302
BoundaryRegionYUp (C++ class), 302
BoundaryRegionYUp::BoundaryRegionYUp (C++ function), 302
BoundaryRegionYUp::first (C++ function), 302
BoundaryRegionYUp::isDone (C++ function), 302
BoundaryRegionYUp::next (C++ function), 302
BoundaryRegionYUp::next1d (C++ function), 302
BoundaryRegionYUp::nextX (C++ function), 302
BoundaryRegionYUp::nextY (C++ function), 302
bout::inversion::InvertableOperator::operatorFunction (C++ function), 741
(bout::inversion::InvertableOperator::opt (C++ member), 547
bout::inversion::InvertableOperator::preconditionerFunction (C++ function), 547
bout::inversion::InvertableOperator::reportTime (C++ function), 546
bout::inversion::InvertableOperator::rhs (C++ member), 546
bout::inversion::InvertableOperator::setOperatorFunction (C++ function), 546
bout::inversion::InvertableOperator::setPreconditionerFunction (C++ function), 546
bout::inversion::InvertableOperator::setup (C++ function), 546
bout::inversion::InvertableOperator::verify (C++ function), 546
bout::inversion::petscVecToField (C++ function), 745
bout::normaliseLocation (C++ function), 446
bout::utils (C++ type), 719, 721, 730, 739
bout::utils::details::ToString (C++ struct), 446
bout::utils::details::ToString (C++ struct), 546
bout::utils::details::ToString (C++ struct), 731
bout::utils::unique_if <C++ struct>, 731
bout::utils::unique_if <T[N] > (C++ struct), 731
bout::utils::unique_if <T[N] >:::known_bound (C++ type), 731
bout::utils::unique_if <T[] > (C++ struct), 731
bout::utils::unique_if <T[] >:::unknown_bound (C++ type), 731
bout::utils::details (C++ type), 721, 740
bout::utils::details::and_all (C++ function), 721
bout::utils::details::compareTypes (C++ struct), 740
bout::utils::details::compareTypes::operator () (C++ function), 740
bout::utils::details::compareTypes<T, T >::operator () (C++ function), 740
bout::utils::details::isEqual (C++ struct), 740
bout::utils::details::isEqual::isEqual (C++ function), 741
bout::utils::details::isEqual::operator () (C++ function), 741
bout::utils::details::isEqual:::t (C++ member), 741
bout::utils::details::staticCastOrThrow (C++ struct), 741
bout::utils::details::staticCastOrThrow::operator () (C++ function), 740

Index
Coordinates::Vpar_Grad_par (C++ function), 345, 346
Coordinates::zlength (C++ function), 344
copy (C++ function), 290, 451
copy_string (C++ function), 723, 726
cos (C++ function), 450
cosh (C++ function), 451
CROSS (C macro), 750
cross (C++ function), 747, 750, 751
Curl (C++ function), 742, 743, 745
CurvedSlab (class in zoidberg.field), 760
cvode_bbd_rhs (C++ function), 349
cvode_jac (C++ function), 349
cvode_pre (C++ function), 349
cvode_pre_shim (C++ function), 349
cvode_rhs (C++ function), 349
CVODEINT (C++ type), 349
CvodeSolver (C++ class), 349
CvodeSolver::CvodeSolver (C++ function), 349
CvodeSolver::create_constraints (C++ function), 350
CvodeSolver::cvode_initialised (C++ member), 351
CvodeSolver::cvode_mem (C++ member), 350
CvodeSolver::CvodeSolver (C++ function), 349
CvodeSolver::diagnose (C++ function), 350
CvodeSolver::getCurrentTimestep (C++ function), 349
CvodeSolver::hcur (C++ member), 350
CvodeSolver::jac (C++ function), 349
CvodeSolver::last_order (C++ member), 351
CvodeSolver::last_step (C++ member), 351
CvodeSolver::loop_vector_option_values_op (C++ function), 350
CvodeSolver::nfevals (C++ member), 350
CvodeSolver::nliters (C++ member), 351
CvodeSolver::nniters (C++ member), 350
CvodeSolver::nonlin_fails (C++ member), 351
CvodeSolver::NOUT (C++ member), 350
CvodeSolver::npevals (C++ member), 351
CvodeSolver::nsteps (C++ member), 350
CvodeSolver::num_fails (C++ member), 351
CvodeSolver::pre (C++ function), 350
CvodeSolver::pre_ncalls (C++ member), 350
CvodeSolver::pre_Wtime (C++ member), 350
CvodeSolver::resetInternalFields (C++ function), 350
CvodeSolver::rhs (C++ function), 350
CvodeSolver::run (C++ function), 350
CvodeSolver::set_vector_option_values (C++ function), 350
CvodeSolver::stab_lims (C++ member), 351
CvodeSolver::TIMESTEP (C++ member), 350
CvodeSolver::TIMESTEP (C++ member), 351
CvodeSolver::uvec (C++ member), 350
CVSpilsSetJacTimes (C++ function), 349
cyclic_tridag (C++ function), 549
CyclicReduce (C++ class), 354
CyclicReduce::~CyclicReduce (C++ function), 354
CyclicReduce::allocMemory (C++ function), 355
CyclicReduce::back_solve (C++ function), 355
CyclicReduce::coeffs (C++ member), 356
CyclicReduce::comm (C++ member), 355
CyclicReduce::CyclicReduce (C++ function), 354
CyclicReduce::if2x2 (C++ member), 356
CyclicReduce::ifcs (C++ member), 356
CyclicReduce::ifp (C++ member), 356
CyclicReduce::ifx (C++ member), 356
CyclicReduce::myif (C++ member), 356
CyclicReduce::myns (C++ member), 355
CyclicReduce::myproc (C++ member), 355
CyclicReduce::N (C++ member), 355
CyclicReduce::nprocs (C++ member), 355
CyclicReduce::Nsyst (C++ member), 355
CyclicReduce::periodic (C++ member), 356
CyclicReduce::recvBuffer (C++ member), 356
CyclicReduce::reduce (C++ function), 353
CyclicReduce::setCoeffs (C++ function), 354
CyclicReduce::setPeriodic (C++ function), 354
CyclicReduce::setup (C++ function), 354
CyclicReduce::solve (C++ function), 354, 355
CyclicReduce::sys0 (C++ member), 355
CyclicReduce::x1 (C++ member), 356
CyclicReduce::xn (C++ member), 356

D
D2DX2 (C++ function), 372, 373, 385
D2DXDY (C++ function), 376, 395–397
D2DXXZ (C++ function), 376, 397
D2DY2 (C++ function), 373, 386
D2DYZ (C++ function), 376, 377, 397, 398
D2DZZ (C++ function), 373, 374, 386, 387
D4DX4 (C++ function), 374, 387, 388
D4DT4 (C++ function), 374, 375, 388, 389
D4DZ4 (C++ function), 375, 389
data_format (C++ function), 362
Datafile (C++ class), 357
Datafile::Datafile (C++ function), 357
Datafile::add (C++ function), 358
Datafile::addOnce (C++ function), 358
Datafile::addRepeat (C++ function), 358
Datafile::appending (C++ member), 360
Datafile::bool_arr (C++ member), 360
Datafile::BoutReal_arr (C++ member), 360
Datafile::can_write_strings (C++ function), 359
Datafile::close (C++ function), 358
Datafile::Datafile (C++ function), 357, 359
Datafile::enabled (C++ member), 360

798
Datafile::f2d_arr (C++ member), 360
Datafile::f3d_arr (C++ member), 360
Datafile::file (C++ member), 360
Datafile::filename (C++ member), 360
Datafile::FILENAMELEN (C++ member), 361
Datafile::filenameLen (C++ member), 360
Datafile::first_time (C++ member), 360
Datafile::floats (C++ member), 360
Datafile::flushFrequency (C++ member), 360
Datafile::flushFrequencyCounter (C++ member), 360
Datafile::fperp_arr (C++ member), 360
Datafile::guards (C++ member), 360
Datafile::init_missing (C++ member), 360
Datafile::int_arr (C++ member), 360
Datafile::int_vec_arr (C++ member), 360
Datafile::Lx (C++ member), 360
Datafile::Ly (C++ member), 360
Datafile::Lz (C++ member), 360
Datafile::mesh (C++ member), 360
Datafile::open (C++ function), 358
Datafile::openclose (C++ member), 360
Datafile::openr (C++ function), 358
Datafile::opendir (C++ function), 358
Datafile::operator= (C++ member), 360
Datafile::read (C++ function), 359
Datafile::read_f2d (C++ function), 359
Datafile::read_f3d (C++ function), 359
Datafile::read_fperp (C++ function), 359
Datafile::read_int (C++ function), 359
Datafile::read_int_vec (C++ function), 359
Datafile::read_real (C++ function), 359
Datafile::read_string (C++ function), 359
DataFormat (C++ class), 362
DataFormat::addVarBoutReal (C++ function), 363
DataFormat::addVarField2D (C++ function), 363
DataFormat::addVarField3D (C++ function), 363
DataFormat::addVarFieldPerp (C++ function), 363
DataFormat::addVarInt (C++ function), 362
DataFormat::addVarIntVec (C++ function), 363
DataFormat::addVarString (C++ function), 363
DataFormat::close (C++ function), 362
DataFormat::DataFormat (C++ function), 362
DataFormat::flush (C++ function), 362
DataFormat::getAttribute (C++ function), 365
DataFormat::getSize (C++ function), 362
DataFormat::is_valid (C++ function), 362
DataFormat::mesh (C++ member), 366
DataFormat::opener (C++ function), 362
DataFormat::openw (C++ function), 362
DataFormat::read (C++ function), 363
DataFormat::read_perp (C++ function), 363
DataFormat::read_rec (C++ function), 363, 364
DataFormat::read_rec_perp (C++ function), 364
DataFormat::readFieldAttributes (C++ function), 366
DataFormat::setAttribute (C++ function), 364, 365
DataFormat::setGlobalOrigin (C++ function), 362
DataFormat::setLocalOrigin (C++ function), 362
DataFormat::setLowPrecision (C++ function), 364
DataFormat::setRecord (C++ function), 362
DataFormat::write (C++ function), 363
DataFormat::write_perp (C++ function), 363
DataFormat::write_rec (C++ function), 364
DataFormat::write_rec_perp (C++ function), 364
DataFormat::writeFieldAttributes (C++ function), 366
DataFormat::~DataFormat (C++ function), 435, 438, 439
dcomplex (C++ type), 366
ddt (C++ function), 430, 439, 747, 752
DDX (C++ function), 370, 382
DDY (C++ function), 371, 382, 383
DDZ (C++ function), 371, 372, 383–385
default (gen_fields attribute), 482
DEFAULT_DIR (C++ member), 318
default_func (C++ function), 303
default_section (C++ member), 637
DeferredFunction (C++ struct), 716
DeferredFunction::DeferredFunction (C++ function), 716
DeferredFunction::operator () (C++ function), 716
DeferredFunction::storedFunc (C++ member), 716
DEFINE_BINARY_OP (C macro), 411
DEFINE_FLUX_DERIV (C macro), 522
DEFINE_FLUX_DERIV_Core (C macro), 522
DEFINE_FLUX_DERIV_Staggered (C macro), 522
DEFINE_OVERLOAD_FUNC (C macro), 411
DEFINE_STANDARD_DERIV (C macro), 522
DEFINE_STANDARD_DERIV_Core (C macro), 522
DEFINE_UPWIND_DERIV (C macro), 522
DEFINE_UPWIND_DERIV_Core (C macro), 522
DEFINE_UPWIND_DERIV_Staggered (C macro), 522
Delp2 (C++ function), 401, 407
DEPRECATED (C macro), 367
DERIV (C++ enum), 325
DERIV::Flux (C++ enumerator), 325
DERIV::Standard (C++ enumerator), 325
DERIV::StandardFourth (C++ enumerator), 325
DERIV::StandardSecond (C++ enumerator), 325
DERIV::Upwind (C++ enumerator), 325
DERIV_FUNC_REGION_ENUM_TO_STRING (C macro), 344, 381
DERIV_STRING (C++ function), 326
DerivativeStore (C++ struct), 367
DerivativeStore::clear (C++ function), 369
DerivativeStore::defaultMethods (C++ member), 370
DerivativeStore::DerivativeStore (C++ function), 368, 369
DerivativeStore::flowFunc (C++ type), 368
DerivativeStore::flux (C++ member), 370
DerivativeStore::fluxFunc (C++ type), 368
DerivativeStore::forceDefaultMethod (C++ function), 369
DerivativeStore::getAvailableMethods (C++ function), 368
DerivativeStore::getFlowDerivative (C++ function), 369
DerivativeStore::getFluxDerivative (C++ function), 369
DerivativeStore::getInstance (C++ function), 369
DerivativeStore::getKey (C++ function), 369, 370
DerivativeStore::getMethodName (C++ function), 369
DerivativeStore::getStandard2ndDerivative (C++ function), 369
DerivativeStore::getStandard4thDerivative (C++ function), 369
DerivativeStore::getUpwindDerivative (C++ function), 369
DerivativeStore::initialise (C++ function), 369
DerivativeStore::isEmpty (C++ function), 368
DerivativeStore::listAvailableMethods (C++ function), 368
DerivativeStore::nameLookup (C++ function), 369
DerivativeStore::registerDerivative (C++ function), 368
DerivativeStore::registeredMethods (C++ member), 370
DerivativeStore::reset (C++ function), 369
DerivativeStore::setDefault (C++ function), 369
DerivativeStore::standard (C++ member), 370
DerivativeStore::standardFourth (C++ member), 370
DerivativeStore::standardFunc (C++ type), 368
DerivativeStore::standardSecond (C++ member), 370
DerivativeStore::storageType (C++ type), 368
DerivativeType (C++ class), 323
DerivativeType::apply (C++ function), 523, 524
DerivativeType::func (C++ member), 524
DerivativeType::meta (C++ member), 524
DerivativeType::standard (C++ function), 523
dest (gen_fieldops attribute), 482
dgtsv_ (C++ function), 549
DIFF_C2 (C++ member), 327
DIFF_C4 (C++ member), 327
DIFF_DEFAULT (C++ member), 327
DIFF_FFT (C++ member), 327
DIFF_METHOD (C++ enum), 324
DIFF_METHOD::c2 (C++ enumerator), 324
DIFF_METHOD::c4 (C++ enumerator), 324
DIFF_METHOD::deflt (C++ enumerator), 324
DIFF_METHOD::fft (C++ enumerator), 324
DIFF_METHOD::s2 (C++ enumerator), 324
DIFF_METHOD::split (C++ enumerator), 324
DIFF_METHOD::u1 (C++ enumerator), 324
DIFF_METHOD::u2 (C++ enumerator), 324
DIFF_METHOD::u3 (C++ enumerator), 324
DIFF_METHOD::w2 (C++ enumerator), 324
DIFF_METHOD::w3 (C++ enumerator), 324
DIFF_METHOD_STRING (C++ function), 326
DIFF_S2 (C++ member), 327
DIFF_SPLIT (C++ member), 327
DIFF_U1 (C++ member), 327
DIFF_U2 (C++ member), 327
DIFF_U3 (C++ member), 327
dimensions (gen_fieldops attribute), 483
DIRECTION (C++ enum), 324
DIRECTION::X (C++ enumerator), 324
DIRECTION::Y (C++ enumerator), 324
DIRECTION::Z (C++ enumerator), 324
DIRECTION::YOrthogonal (C++ enumerator), 324
DIRECTION::YAligned (C++ enumerator), 324
DIRECTION_STRING (C++ function), 326
DirectionTypes (C++ struct), 327
DirectionTypes::y (C++ member), 328
DirectionTypes::z (C++ member), 328
distance() (zoidberg.rzline.RZline method), 776
Div (C++ function), 742, 744, 745
Div_par (C++ function), 399, 400, 404, 405
Div_par_Ctol (C++ function), 406
Div_par_flux (C++ function), 400, 405
Div_par_K_Grad_par (C++ function), 400, 401, 406, 407
Div_par_LtoC (C++ function), 406
Divide (C++ struct), 417
Divide::apply (C++ function), 417
double_arg_op (C++ type), 459
DST (C++ function), 425
DST_rev (C++ function), 425
DummyOutput (C++ class), 624
DummyOutput::disable (C++ function), 625
DummyOutput::enable (C++ function), 625
DummyOutput::isEnabled (C++ function), 625
DummyOutput::print (C++ function), 625
eval3D (C++ function), 411
ExpressionParser (C++ class), 418
ExpressionParser::addExpressionParser (C++ function), 418
ExpressionParser::addBinaryOp (C++ function), 419
ExpressionParser::addGenerator (C++ function), 418
ExpressionParser::bin_op (C++ member), 420
ExpressionParser::ExpressionParser (C++ function), 418
ExpressionParser::gen (C++ member), 420
ExpressionParser::LexInfo (C++ struct), 420
ExpressionParser::LexInfo::curident (C++ member), 420
ExpressionParser::LexInfo::curtok (C++ member), 420
ExpressionParser::LexInfo::curval (C++ member), 420
ExpressionParser::LexInfo::LexInfo (C++ function), 420
ExpressionParser::LexInfo::LexInfo (C++ function), 420
ExpressionParser::LexInfo::setRecord (C++ function), 420
ExpressionParser::LexInfo::ss (C++ member), 420
ExpressionParser::LexInfo::ss (C++ member), 420
ExpressionParser::LexInfo::parseBinOpRHS (C++ function), 419
ExpressionParser::LexInfo::parseExpression (C++ function), 419
ExpressionParser::LexInfo::parseIdentifierExpr (C++ function), 419
ExpressionParser::LexInfo::parseParenExpr (C++ function), 419
ExpressionParser::parsePrimary (C++ function), 419
ExpressionParser::parseString (C++ function), 419
ExpressionParser::reserved_chars (C++ member), 419
ExpressionParser::resolve (C++ function), 419
exprTraits (C++ struct), 413
exprTraits::expr_type (C++ type), 413
exprTraits<double> (C++ struct), 413
exprTraits<double>::expr_type (C++ type), 413
eexprTraits<float> (C++ struct), 413
eexprTraits<float>::expr_type (C++ type), 413
eexprTraits<int> (C++ struct), 413
eexprTraits<int>::expr_type (C++ type), 414

F
Factory (C++ class), 487
Factory::add (C++ function), 487
Factory::create (C++ function), 487
Factory::Factory (C++ function), 488
Factory::getInstance (C++ function), 487
Factory::listAvailable (C++ function), 487
Factory::remove (C++ function), 487
Factory::type_map (C++ member), 488
False (gen_fieldops attribute), 482
FCIMap (C++ class), 422
FCIMap::boundary_mask (C++ member), 423
FCIMap::corner_boundary_mask (C++ member), 423
FCIMap::FCIMap (C++ function), 422
FCIMap::FCIMap::initialize (C++ function), 422
FCIMap::interp (C++ member), 423
FCIMap::interp_corner (C++ member), 423
FCIMap::interpolate (C++ function), 422
FCIMap::map_mesh (C++ member), 423
FCIMap::offset (C++ member), 423
FCITransform (C++ class), 423
FCITransform::calcParallelSlices (C++ function), 423
FCITransform::canToFromFieldAligned (C++ function), 423
FCITransform::checkInputGrid (C++ function), 424
FCITransform::FCITransform (C++ function), 423
FCITransform::field_line_maps (C++ member), 424
FCITransform::fromFieldAligned (C++ function), 423
FCITransform::integrateParallelSlices (C++ function), 423
FCITransform::requiresTwistShift (C++ function), 423
FCITransform::toFieldAligned (C++ function), 423
fcmplx (C++ struct), 367
fcmplx::i (C++ member), 367
fcmplx::r (C++ member), 367
FDDX (C++ function), 379, 393, 394
FDDY (C++ function), 380, 394
FDDZ (C++ function), 380, 381, 395
FFT_MEASUREMENT_FLAG (C++ enum), 424
FFT_MEASUREMENT_FLAG::estimate (C++ enumerator), 424
FFT_MEASUREMENT_FLAG::exhaustive (C++ enumerator), 424
FFT_MEASUREMENT_FLAG::measure (C++ enumerator), 424
FFT_MEASUREMENT_FLAG::fromString (C++ function), 425
Field (C++ class), 452
Field2D (C++ class), 430
Field2D (gen_fieldops attribute), 482
Field2D::Field2D (C++ function), 431
Field2D::accept (C++ function), 433
Field2D::allocate (C++ function), 431
Field2D::applyBoundary (C++ function), 433
Field2D::applyTDerivBoundary (C++ function), 433
Field2D::begin (C++ function), 432
Field2D::BoutRealSize (C++ function), 433
Field2D::byteSize (C++ function), 433
Field2D::data (C++ member), 434
Field2D::deriv (C++ member), 434
Field2D::doneComms (C++ function), 433
Field2D::end (C++ function), 432
Field2D::Field2D (C++ function), 431
Field2D::getNx (C++ function), 431
Field2D::getNy (C++ function), 431
Field2D::getNz (C++ function), 431
Field2D::getField2D (C++ function), 431
Field2D::ind_type (C++ function), 431
Field2D::is3D (C++ function), 433
Field2D::isAllocated (C++ function), 431
Field2D::isReal (C++ function), 433
Field2D::nx (C++ member), 434
Field2D::ny (C++ member), 434
Field2D::Field2D::operator* (C++ function), 432, 433
Field2D::Field2D::operator*=(C++ function), 433
Field2D::Field2D::operator+=(C++ function), 433
Field2D::Field2D::operator-=(C++ function), 432
Field2D::Field2D::operator-=(C++ function), 433
Field2D::Field2D::operator[] (C++ function), 432
Field2D::setBoundaryTo (C++ function), 434
Field2D::setDirectionY (C++ function), 431
Field2D::setLocation (C++ function), 431
Field2D::timeDeriv (C++ function), 431
Field2D::value_type (C++ type), 431
Field2D::ydown (C++ function), 432
Field2D::ynext (C++ function), 432
Field2D::yup (C++ function), 432
Field2DExpr (C++ class), 412
Field2DExpr::data (C++ member), 413
Field2DExpr::Field2DExpr (C++ function), 413
Field2DExpr::operator() (C++ function), 413
Field2DExpr::type (C++ type), 413
Field3D (C++ class), 439
field3D (gen_fieldops attribute), 482
Field3D::Field3D (C++ function), 442
Field3D::accept (C++ function), 445
Field3D::allocate (C++ function), 443
Field3D::applyBoundary (C++ function), 445
Field3D::applyParallelBoundary (C++ function), 445
Field3D::applyTDerivBoundary (C++ function), 445
Field3D::background (C++ member), 446
Field3D::begin (C++ function), 444
Field3D::BoutRealSize (C++ function), 445
Field3D::byteSize (C++ function), 445
Field3D::calcParallelSlices (C++ function), 445
Field3D::clearParallelSlices (C++ function), 443
Field3D::data (C++ member), 446
Field3D::deriv (C++ member), 446
Field3D::doneComms (C++ function), 445
Field3D::end (C++ function), 444
Field3D::Field3D (C++ function), 442
Field3D::getNx (C++ function), 443
Field3D::getNy (C++ function), 443
Field3D::getNz (C++ function), 443
Field3D::getRegion (C++ function), 444
Field3D::getRegion2D (C++ function), 444
Field3D::hasParallelSlices (C++ function), 443
Field3D::hasYupYdown (C++ function), 443
Field3D::ind_type (C++ type), 442
Field3D::is3D (C++ function), 445
Field3D::isAllocated (C++ function), 443
Field3D::isReal (C++ function), 445
Field3D::mergeYupYdown (C++ function), 443
Field3D::nx (C++ member), 446
Field3D::ny (C++ member), 446
Field3D::nz (C++ member), 446
Field3D::operator() (C++ function), 444, 445
Field3D::operator=(C++ function), 442
Field3D::operator=(C++ function), 441
Field3D::operator=(C++ function), 442
Field3D::operator=(C++ function), 443
Field3D::operator=(C++ function), 441
Field3D::operator() (C++ function), 444
Field3D::requiresTwistShift (C++ function), 444
Field3D::setBoundaryTo (C++ function), 445
Field3D::setDirectionY (C++ function), 443
Field3D::setLocation (C++ function), 443
Field3D::splitParallelSlices (C++ function), 443
Field3D::splitYupYdown (C++ function), 443
Field3D::timeDeriv (C++ function), 443
Field3D::value_type (C++ type), 443
Field3D::ydown (C++ function), 443
Field3D::ydown_fields (C++ member), 446
Field3D::ynext (C++ function), 443
Field3D::yup (C++ function), 443
Field3D::yup_fields (C++ member), 446
Field3DExpr (C++ class), 412
Field3DExpr::data (C++ member), 412
Field3DExpr::Field3DExpr (C++ function), 412
Field3DExpr::operator() (C++ function), 412
Field3DExpr::type (C++ type), 412
Field::Field (C++ function), 452
Field::bdry_xin (C++ member), 453
Field::bdry_xout (C++ member), 453
Field::bdry_ydown (C++ member), 453
Field::bdry_yup (C++ member), 453
Field::bdryValid (C++ function), 452
Field::copyFieldMembers (C++ function), 453
Field::directions (C++ member), 453
Field::Field (C++ function), 452
Field::fieldCoordinates (C++ member), 453
Field::fieldmesh (C++ member), 453
Field::getCoordinates (C++ function), 453
Field::getDirections (C++ function), 452
Field::getDirectionY (C++ function), 452
Field::getDirectionZ (C++ function), 452
Field::getLocation (C++ function), 452
Field::getMesh (C++ function), 453
Field::getNx (C++ function), 453
Field::getNy (C++ function), 453
Field::getNz (C++ function), 453
Field::location (C++ member), 453
Field::name (C++ member), 453
Field::operator= (C++ function), 452
Field::setDirectionY (C++ function), 452
Field::setDirectionZ (C++ function), 452
Field::setLocation (C++ function), 452
field_direction() (zoidberg.field.MagneticField method), 763
FIELD_FUNC (C macro), 447
field_type (gen_fieldops attribute), 483
FieldAbs (C++ class), 464
FieldAbs::clone (C++ function), 464
FieldAbs::FieldAbs (C++ function), 464
FieldAbs::gen (C++ member), 465
FieldAbs::generate (C++ function), 464
FieldATan (C++ class), 462
FieldATan::A (C++ member), 462
FieldATan::B (C++ member), 462
FieldATan::clone (C++ function), 462
FieldATan::FieldATan (C++ function), 462
FieldATan::generate (C++ function), 462
FieldBallooning (C++ class), 468
FieldBallooning::arg (C++ member), 468
FieldBallooning::ball_n (C++ member), 468
FieldBallooning::clone (C++ function), 468
FieldBallooning::FieldBallooning (C++ function), 468
FieldBallooning::generate (C++ function), 468
FieldBallooning::clone (C++ member), 420
FieldBinary::clone (C++ function), 421
FieldBinary::FieldBinary (C++ function), 421
FieldBinary::generate (C++ function), 421
FieldBinary::lhs (C++ member), 421
FieldBinary::op (C++ member), 421
FieldBinary::str (C++ function), 421
FieldCos (C++ class), 460
FieldCos::clone (C++ function), 461
FieldCos::FieldCos (C++ function), 461
FieldCos::gen (C++ member), 461
FieldCos::generate (C++ function), 461
FieldCos::str (C++ function), 461
FieldCos::clone (C++ function), 463
FieldCos::FieldCos (C++ function), 463
FieldCos::gen (C++ member), 463
FieldCos::generate (C++ function), 463
FieldData (C++ class), 455
FieldData::FieldData (C++ function), 455
FieldData::accept (C++ function), 455
FieldData::addBndryFunction (C++ function), 455
FieldData::addBndryGenerator (C++ function), 455
FieldData::applyBoundary (C++ function), 455
FieldData::applyYDerivBoundary (C++ function), 455
FieldData::bdry_generator (C++ member), 456
FieldData::bdry_op (C++ member), 456
FieldData::bdry_op_par (C++ member), 456
FieldData::boundaryIsCopy (C++ member), 456
FieldData::boundaryIsSet (C++ member), 456
FieldData::BoutRealSize (C++ function), 455
FieldData::byteSize (C++ function), 455
FieldData::copyBoundary (C++ function), 455
FieldData::doneComms (C++ function), 455
FieldData::FieldData (C++ function), 455
FieldData::getBndryGenerator (C++ function), 455
FieldData::is3D (C++ function), 455
FieldData::isReal (C++ function), 455
FieldData::setBoundary (C++ function), 455
FieldErf (C++ class), 466
FieldErf::clone (C++ function), 466
FieldErf::FieldErf (C++ function), 466
FieldErf::gen (C++ member), 466
FieldErf::generate (C++ function), 466
FieldFactory (C++ class), 456
FieldFactory::FieldFactory (C++ function), 456
FieldFactory::cache (C++ member), 458
FieldFactory::cleanCache (C++ function), 457
FieldFactory::create2D (C++ function), 456, 457
FieldFactory::create3D (C++ function), 456, 457
FieldFactory::createPerp (C++ function), 457
FieldFactory::FieldFactory (C++ function), 456
FieldFactory::fieldmesh (C++ member), 457
FieldFactory::findOption (C++ function), 457
FieldFactory::get (C++ function), 457
FieldFactory::lookup (C++ member), 458
FieldFactory::options (C++ member), 457
FieldFactory::parse (C++ function), 457
FieldFactory::resolve (C++ function), 457
FieldFactory::transform_from_field_aligned (C++ member), 457
FieldFunction (C++ class), 458
FieldFunction::FieldFunction (C++ function), 458
FieldFunction::func (C++ member), 458
FieldFunction::generate (C++ function), 458
FieldGaussian (C++ class), 464
FieldGaussian::clone (C++ function), 464
FieldGaussian::FieldGaussian (C++ function), 464
FieldGaussian::generate (C++ function), 464
FieldGaussian::s (C++ member), 464
FieldGaussian::x (C++ member), 464
FieldGenerator (C++ class), 418
FieldGenerator::FieldGenerator (C++ function), 418
FieldGenerator::clone (C++ function), 418
FieldGenerator::generate (C++ function), 418
FieldGenerator::str (C++ function), 418
FieldGeneratorPtr (C++ type), 418
FieldGenOneArg (C++ class), 461
FieldGenOneArg::clone (C++ function), 461
FieldGenOneArg::FieldGenOneArg (C++ function), 461
FieldGenOneArg::gen (C++ member), 461
FieldGenOneArg::generate (C++ function), 461
FieldGenOneArg::str (C++ function), 461
FieldGenTwoArg (C++ class), 461
FieldGenTwoArg::clone (C++ function), 462
FieldGenTwoArg::FieldGenTwoArg (C++ function), 462
FieldGenTwoArg::generate (C++ function), 462
FieldGenTwoArg::str (C++ function), 462
FieldGroup (C++ class), 470
FieldGroup::add (C++ function), 471
FieldGroup::begin (C++ function), 471, 472
FieldGroup::clear (C++ function), 471
FieldGroup::const_iterator (C++ type), 470
FieldGroup::empty (C++ function), 471
FieldGroup::end (C++ function), 472
FieldGroup::f3vec (C++ member), 472
FieldGroup::field3d (C++ function), 472
FieldGroup::FieldGroup (C++ function), 470
FieldGroup::fvec (C++ member), 472
FieldGroup::get (C++ function), 472
FieldGroup::iterator (C++ type), 470
FieldGroup::makeUnique (C++ function), 472
FieldGroup::operator+= (C++ function), 471
FieldGroup::operator= (C++ function), 470
FieldGroup::size (C++ function), 471
FieldGroup::size_field3d (C++ function), 471
FieldHeaviside::clone (C++ class), 465
FieldHeaviside::clone (C++ function), 465
FieldHeaviside::FieldHeaviside (C++ function), 465
FieldHeaviside::gen (C++ member), 466
FieldHeaviside::generate (C++ function), 465
FieldHeaviside::str (C++ function), 465
FieldMax::clone (C++ class), 467
FieldMax::clone (C++ function), 467
FieldMax::FieldMax (C++ function), 467
FieldMax::generate (C++ function), 467
FieldMax::input (C++ class), 466
FieldMin::clone (C++ member), 467
FieldMin::clone (C++ function), 467
FieldMin::FieldMin (C++ function), 466
FieldMin::generate (C++ function), 466
FieldMin::input (C++ member), 467
FieldMixmode (C++ class), 468
FieldMixmode::arg (C++ member), 469
FieldMixmode::clone (C++ function), 468
FieldMixmode::FieldMixmode (C++ function), 468
FieldMixmode::generate (C++ function), 468
FieldMixmode::genRand (C++ function), 469
FieldMixmode::phase (C++ member), 469
FieldNull (C++ class), 458
FieldNull::clone (C++ function), 458
FieldNull::FieldNull (C++ function), 458
FieldNull::generate (C++ function), 458
FieldNull::get (C++ function), 459
FieldPerp (C++ class), 474
FieldPerp (gen_fieldops attribute), 482
FieldPerp::FieldPerp (C++ function), 475
FieldPerp::allocate (C++ function), 475
FieldPerp::begin (C++ function), 475
FieldPerp::data (C++ member), 477
FieldPerp::end (C++ function), 475
FieldPerp::FieldPerp (C++ function), 474
FieldPerp::getGlobalIndex (C++ function), 475
FieldPerp::getIndex (C++ function), 475
FieldPerp::getNx (C++ function), 477
FieldPerp::getNy (C++ function), 477
FieldPerp::getNz (C++ function), 477
FieldPerp::getRegion (C++ function), 475
FieldPerp::ind_type (C++ type), 474
FieldPerp::isAllocated (C++ function), 476
FieldPerp::nx (C++ member), 477
FieldPerp::nz (C++ member), 477
FieldPerp::operator() (C++ function), 476
FieldPerp::operator*=(C++ function), 476
FieldPerp::operator/=(C++ function), 476
FieldPerp::operator=(C++ function), 476
FieldPerp::operator[] (C++ function), 475
FieldPerp::operator= (C++ function), 476
FieldPerp::setDirectionY (C++ function), 475
FieldPerp::setIndex (C++ function), 475
FieldPerp::setIndexFromGlobal (C++ function), 475
FieldPerp::setLocation (C++ function), 475
FieldPerp::yindex (C++ member), 477
FieldRound (C++ class), 467
FieldRound::clone (C++ function), 467
FieldRound::FieldRound (C++ function), 467
FieldRound::gen (C++ member), 468
FieldRound::generate (C++ function), 467
fields (gen_fieldops attribute), 482
FieldSin (C++ class), 460
FieldSin::clone (C++ function), 460
FieldSin::FieldSin (C++ function), 460
FieldSin::gen (C++ member), 460
FieldSin::generate (C++ function), 460
FieldSin::str (C++ function), 460
FieldSin (C++ class), 462
FieldSin::clone (C++ function), 463
FieldSin::FieldSin (C++ function), 463
FieldSin::gen (C++ member), 463
FieldSin::generate (C++ function), 463
FieldSqrt (C++ class), 465
FieldSqrt::clone (C++ function), 465
FieldSqrt::FieldSqrt (C++ function), 465
FieldSqrt::gen (C++ member), 465
FieldSqrt::generate (C++ function), 465
FieldSqrt::getRegion (C++ function), 465
FieldTanh (C++ class), 463
FieldTanh::clone (C++ function), 463
FieldTanh::FieldTanh (C++ function), 463
FieldTanh::gen (C++ member), 464
FieldTanh::generate (C++ function), 463
FieldTanhHat (C++ class), 469
FieldTanhHat::center (C++ member), 469
FieldTanhHat::clone (C++ function), 469
FieldTanhHat::FieldTanhHat (C++ function), 469
FieldTanhHat::generate (C++ function), 469
FieldTanhHat::steepness (C++ member), 469
FieldTanhHat::width (C++ member), 469
FieldTanhHat::X (C++ member), 469
FieldTracer (class in zoidberg.fieldtracer), 766
| FieldTracerReversible (class in zoidberg.fieldtracer), 766 |
| FieldValue (C++ class), 421 |
| FieldValue::clone (C++ function), 421 |
| FieldValue::FieldValue (C++ function), 421 |
| FieldValue::generate (C++ function), 421 |
| FieldValue::str (C++ function), 421 |
| FieldValue::str (C++ member), 422 |
| FieldValuePtr (C++ class), 459 |
| FieldValuePtr::clone (C++ function), 460 |
| FieldValuePtr::clone (C++ function), 460 |
| FieldValuePtr::FieldValuePtr (C++ function), 460 |
| FieldValuePtr::ptr (C++ member), 460 |
| FieldVisitor (C++ class), 459 |
| FieldVisitor::accept (C++ function), 459 |
| filledFrom (C++ function), 447 |
| filter (C++ function), 435, 437 |
| findIndex() (zoidberg.poloidal_grid.RectangularPoloidalGrid method), 772 |
| findIndex() (zoidberg.poloidal_grid.StructuredPoloidalGrid method), 773 |
| finite (C++ function), 451 |
| floor (C++ function), 451, 452 |
| follow_field_lines() (zoidberg.fieldtracer.FieldTracer method), 766 |
| follow_field_lines() (zoidberg.fieldtracer.FieldTracerReversible method), 766 |
| formatEig (C++ function), 682 |
| FormatFactory (C++ class), 477 |
| FormatFactory::createDataFormat (C++ function), 478 |
| FormatFactory::createDataFormat (C++ function), 478 |
| FormatFactory::getInstance (C++ function), 478 |
| FormatFactory::instance (C++ member), 478 |
| FormatFactory::matchString (C++ function), 478 |
| FormFunction (C++ function), 513, 690 |
| FormFunctionForColoring (C++ function), 513, 690 |
| FormFunctionForDifferencing (C++ function), 513, 690 |
| fromFieldAligned (C++ function), 430, 448, 472, 747, 750, 751 |
| FuncPtr (C++ type), 323 |
| FV (C++ type), 478 |
| FV::communicateFluxes (C++ function), 479 |
| FV::DDY4 (C++ function), 478 |
| FV::DDY4Index (C++ function), 478 |
| FV::Div_a_Laplace_perp (C++ function), 478 |
| FV::Div_f_v (C++ function), 479 |
| FV::Div_par (C++ function), 479 |
| FV::Div_par_K_Grad_par (C++ function), 478 |
| FV::Fromm (C++ struct), 479 |
| FV::Fromm::operator() (C++ function), 480 |
| FV::MC (C++ struct), 480 |
| FV::MC::minmod (C++ function), 480 |
| FV::MC::operator() (C++ function), 480 |
| FV::MinMod (C++ struct), 480 |
| FV::MinMod::_minmod (C++ function), 480 |
| FV::MinMod::operator() (C++ function), 480 |
| FV::Stencil1D (C++ struct), 480 |
| FV::Stencil1D::c (C++ member), 481 |
| FV::Stencil1D::L (C++ member), 481 |
| FV::Stencil1D::m (C++ member), 481 |
| FV::Stencil1D::mm (C++ member), 481 |
| FV::Stencil1D::p (C++ member), 481 |
| FV::Stencil1D::pp (C++ member), 481 |
| FV::Stencil1D::R (C++ member), 481 |
| FV::Upwind (C++ struct), 481 |
| FV::Upwind::operator() (C++ function), 481 |

G

gen_fieldops (built-in class), 481

gen_fieldops.__eq__ ()

built-in function, 483
gen_fieldops.__init__() 

built-in function, 483
gen_fieldops.__ne__() 

built-in function, 483
gen_fieldops.__repr__() 

built-in function, 483
gen_fieldops.__str__() 

built-in function, 483
gen_fieldops.base_index() 

built-in function, 483
gen_fieldops.index() 

built-in function, 483
gen_fieldops.mixed_index() 

built-in function, 483
gen_fieldops.passByReference() 

built-in function, 483
gen_fieldops.returnType() 

built-in function, 481
gen_fieldops.smart_open() 

built-in function, 481
generator (C++ function), 456

GEQDSK (class in zoidberg.field), 761

globalField (C++ class), 489

globalField2D (C++ class), 490

globalField2D::GlobalField2D (C++ function), 491
GlobalField2D::buffer (C++ member), 492
GlobalField2D::data_valid (C++ member), 492
GlobalField2D::gather (C++ function), 491
GlobalField2D::GlobalField2D (C++ function), 491
GlobalField2D::msg_len (C++ function), 492
GlobalField2D::operator() (C++ function), 491
GlobalField2D::operator=(C++ function), 491
GlobalField2D::scatter (C++ function), 491
GlobalField2D::valid (C++ function), 491
GlobalField3D (C++ class), 492
GlobalField3D::GlobalField3D (C++ function), 493
GlobalField3D::gather (C++ function), 493
GlobalField3D::data_valid (C++ function), 493
GlobalField3D::msg_len (C++ function), 493
GlobalField3D::operator= (C++ function), 493
GlobalField3D::scatter (C++ function), 493
GlobalField3D::valid (C++ function), 493
GlobalField::GlobalField (C++ function), 489
GlobalField::comm (C++ member), 490
GlobalField::data (C++ member), 490
GlobalField::data_on_proc (C++ member), 490
GlobalField::dataIsLocal (C++ function), 489
GlobalField::get_data (C++ function), 489
GlobalField::getGlobalField (C++ function), 489
GlobalField::mesh (C++ member), 490
GlobalField::mype (C++ member), 490
GlobalField::npes (C++ member), 490
GlobalField::nx (C++ member), 490
GlobalField::my (C++ member), 490
GlobalField::nxz (C++ member), 490
GlobalField::proc_local_origin (C++ function), 489
GlobalField::proc_origin (C++ function), 489
GlobalField::proc_size (C++ function), 489
GlobalField::valid (C++ function), 489
GlobalField::xSize (C++ function), 489
GlobalField::ySize (C++ function), 489
GlobalField::zSize (C++ function), 489
GLOBALORIGIN (C macro), 317
Grad (C++ function), 742, 744
Grad2_par2 (C++ function), 400, 405, 406
GRAD_FUNC_REGION_ENUM_TO_STRING (C macro), 344
Grad_par (C++ function), 399, 403
Grad_par_CtoL (C++ function), 406
Grad_par_LtoC (C++ function), 406
Grad_parP (C++ function), 399, 403
Grad_perp (C++ function), 742, 744
Grid (class in zoidberg.grid), 768
grid_annulus() (in module zoidberg.poloidal_grid), 773
grid_elliptic() (in module zoidberg.poloidal_grid), 774
GRID_LOAD (C macro), 494
GRID_LOAD1 (C macro), 494
GRID_LOAD2 (C macro), 494
GRID_LOAD3 (C macro), 494
GRID_LOAD4 (C macro), 494
GRID_LOAD5 (C macro), 494
GRID_LOAD6 (C macro), 494
GridDataSource (C++ class), 494
GridDataSource::~GridDataSource (C++ function), 495
GridDataSource::Direction (C++ enum), 495
GridDataSource::Direction::X (C++ enumerator), 495
GridDataSource::Direction::Y (C++ enumerator), 495
GridDataSource::Direction::Z (C++ enumerator), 495
GridDataSource::get (C++ function), 495
GridDataSource::GridDataSource (C++ function), 495
GridDataSource::hasVar (C++ function), 495
GridDataSource::hasXBoundaryGuards (C++ function), 495
GridDataSource::hasYBoundaryGuards (C++ function), 495
GridDataSource::is_file (C++ member), 496
GridDataSource::X (C++ member), 496
GridDataSource::Y (C++ member), 496
GridDataSource::Z (C++ member), 496
GridFile (C++ class), 496
GridFile::~GridFile (C++ function), 496
GridFile::file (C++ member), 498
GridFile::filename (C++ member), 498
GridFile::get (C++ function), 496, 497
GridFile::get_field (C++ function), 497
GridFile::grid_yguards (C++ member), 498
GridFile::GridFile (C++ function), 496
GridFile::hasVar (C++ function), 496
GridFile::hasXBoundaryGuards (C++ function), 497
GridFile::hasYBoundaryGuards (C++ function), 497
GridFile::ny_inner (C++ member), 498
GridFile::readField (C++ function), 497, 498
GridFile::readgrid_3dvar_fft (C++ function), 497
GridFile::readgrid_3dvar_real (C++ function), 497
GridFile::readgrid_perpvar_fft (C++ function), 497
GridFile::readgrid_perpvar_real (C++ function), 497
GridFromOptions (C++ class), 498
GridFromOptions::get (C++ function), 498, 499
LaplaceMumps::Ex (C++ member), 590
LaplaceMumps:: Ez (C++ member), 590
LaplaceMumps::fourth_order (C++ member), 590
LaplaceMumps::Iend (C++ member), 590
LaplaceMumps::implemented_boundary_flags (C++ member), 590
LaplaceMumps::implemented_flags (C++ member), 590
LaplaceMumps::issetC (C++ member), 590
LaplaceMumps::issetD (C++ member), 590
LaplaceMumps::issetE (C++ member), 590
LaplaceMumps::Istart (C++ member), 590
LaplaceMumps::LaplaceMumps (C++ function), 594, 596
LaplaceMumps::maxits (C++ member), 596
LaplaceMumps::naulinsolver_mean_its (C++ member), 596
LaplaceMumps::naulinsolver_mean_underrelax_counts (C++ member), 596
LaplaceMumps::ncalls (C++ member), 596
LaplaceMumps::operator= (C++ function), 596
LaplaceMumps::resetMeanIterations (C++ function), 595
LaplaceMumps::rtol (C++ member), 596
LaplaceMumps::setCoeffA (C++ function), 594
LaplaceMumps::setCoeffC (C++ function), 594
LaplaceMumps::setCoeffC1 (C++ function), 594
LaplaceMumps::setCoeffC2 (C++ function), 595
LaplaceMumps::setCoeffD (C++ function), 595
LaplaceMumps::setCoeffEx (C++ function), 595
LaplaceMumps::setCoeffEz (C++ function), 595
LaplaceMumps::setGlobalFlags (C++ function), 595
LaplaceMumps::setInnerBoundaryFlags (C++ function), 595
LaplaceMumps::setOuterBoundaryFlags (C++ function), 595
LaplaceNaulin::solve (C++ function), 595
LaplaceNaulin::uses3DCoefs (C++ function), 595
laplacePCapply (C++ function), 551, 643
LaplacePDD::LaplacePDD (C++ class), 637
LaplacePDD::LaplacePDD (C++ function), 637
LaplacePDD::Acoef (C++ member), 638
LaplacePDD::Ccoef (C++ member), 638
LaplacePDD::Dcoef (C++ member), 638
LaplacePDD::finish (C++ function), 638
LaplacePDD::LaplacePDD (C++ function), 637
LaplacePDD::next (C++ function), 638
LaplacePDD::PDD_COMM_XV (C++ member), 638
LaplacePDD::PDD_COMM_Y (C++ member), 638
LaplacePDD::PDD_DATA (C++ struct), 638
LaplacePDD::PDD_DATA::ave (C++ member), 638
LaplacePDD::PDD_DATA::bk (C++ member), 638
LaplacePDD::PDD_DATA::bvec (C++ member), 638
LaplacePDD::PDD_DATA::cvec (C++ member), 638
LaplacePDD::PDD_DATA::jy (C++ member), 638
LaplacePDD::PDD_DATA::rv (C++ member), 639
LaplacePDD::PDD_DATA::recv_handle (C++ member), 639
LaplacePDD::PDD_DATA::snd (C++ member), 639
LaplacePDD::PDD_DATA::v (C++ member), 638
LaplacePDD::PDD_DATA::w (C++ member), 639
LaplacePDD::PDD_DATA::xx (C++ member), 638
LaplacePDD::rtol (C++ member), 596
LaplaceNaulin::operator= (C++ function), 596
LaplaceNaulin::solve (C++ function), 595
LaplaceNaulin::setCoeffA (C++ function), 594
LaplaceNaulin::setCoeffC (C++ function), 594
LaplaceNaulin::setCoeffC1 (C++ function), 594
LaplaceNaulin::setCoeffC2 (C++ function), 595
LaplaceNaulin::setCoeffD (C++ function), 595
LaplaceNaulin::setCoeffEx (C++ function), 595
LaplaceNaulin::setCoeffEz (C++ function), 595
LaplaceNaulin::setGlobalFlags (C++ function), 595
LaplaceNaulin::setInnerBoundaryFlags (C++ function), 595
LaplaceNaulin::setOuterBoundaryFlags (C++ function), 595
LaplaceNaulin::solve (C++ function), 595
LaplaceNaulin::uses3DCoefs (C++ function), 595
laplacePCapply (C++ function), 551, 643
LaplacePDD::LaplacePDD (C++ class), 637
LaplacePDD::LaplacePDD (C++ function), 637
LaplacePDD::Acoef (C++ member), 638
LaplacePDD::Ccoef (C++ member), 638
LaplacePDD::Dcoef (C++ member), 638
LaplacePDD::finish (C++ function), 638
LaplacePDD::LaplacePDD (C++ function), 637
LaplacePDD::next (C++ function), 638
LaplacePDD::PDD_COMM_XV (C++ member), 638
LaplacePDD::PDD_COMM_Y (C++ member), 638
LaplacePDD::PDD_DATA (C++ struct), 638
LaplacePDD::PDD_DATA::ave (C++ member), 638
LaplacePDD::PDD_DATA::bk (C++ member), 638
LaplacePDD::PDD_DATA::bvec (C++ member), 638
LaplacePDD::PDD_DATA::cvec (C++ member), 638
LaplacePDD::PDD_DATA::jy (C++ member), 638
LaplacePDD::PDD_DATA::rv (C++ member), 639
LaplacePDD::PDD_DATA::recv_handle (C++ member), 639
LaplacePDD::PDD_DATA::snd (C++ member), 639
LaplacePDD::PDD_DATA::v (C++ member), 638
LaplacePDD::PDD_DATA::w (C++ member), 639
LaplacePDD::PDD_DATA::xx (C++ member), 638
LaplaceShoot::LaplaceShoot (C++ function), 681
LaplaceShoot::nmode (C++ member), 681
LaplaceShoot::rshk (C++ member), 681
LaplaceShoot::setCoeffA (C++ function), 681
LaplaceShoot::setCoeffC (C++ function), 681
LaplaceShoot::setCoeffD (C++ function), 681
LaplaceShoot::setCoeffEx (C++ function), 681
LaplaceShoot::setCoeffEz (C++ function), 681
LaplaceShoot::solve (C++ function), 681
LaplaceSPT::SPT_data (C++ class), 709
LaplaceSPT::~LaplaceSPT (C++ function), 709
LaplaceSPT::Acoef (C++ member), 711
LaplaceSPT::alldata (C++ member), 711
LaplaceSPT::aone (C++ member), 711
LaplaceSPT::dc1d (C++ member), 711
LaplaceSPT::dcoeff (C++ member), 711
LaplaceSPT::finish (C++ function), 711
LaplaceSPT::LaplaceSPT (C++ function), 709
LaplaceSPT::next (C++ function), 711
LaplaceSPT::setCoeffA (C++ function), 709
LaplaceSPT::setCoeffC (C++ function), 709
LaplaceSPT::setCoeffD (C++ function), 709
LaplaceSPT::setCoeffEx (C++ function), 709
LaplaceSPT::slicedata (C++ member), 711
LaplaceSPT::solve (C++ function), 709, 710
LaplaceSPT::SPT_data (C++ struct), 711
LaplaceSPT::SPT_data::~SPT_data (C++ function), 712
LaplaceSPT::SPT_data::allocate (C++ function), 712
LaplaceSPT::SPT_data::avec (C++ member), 712
LaplaceSPT::SPT_data::bk (C++ member), 712
LaplaceSPT::SPT_data::buffer (C++ member), 712
LaplaceSPT::SPT_data::bvec (C++ member), 712
LaplaceSPT::SPT_data::comm_tag (C++ member), 712
LaplaceSPT::SPT_data::cvec (C++ member), 712
LaplaceSPT::SPT_data::dir (C++ member), 712
LaplaceSPT::SPT_data::gam (C++ member), 712
LaplaceSPT::SPT_data::jy (C++ member), 712
LaplaceSPT::SPT_data::proc (C++ member), 712
LaplaceSPT::SPT_data::recv_handle (C++ member), 712
LaplaceSPT::SPT_data::SPT_data (C++ function), 712
LaplaceSPT::SPT_data::xk (C++ member), 712
LaplaceSPT::start (C++ function), 711
LaplaceSPT::tridagBack (C++ function), 710
LaplaceSPT::tridagForward (C++ function), 710
LaplaceSPT::ye (C++ member), 711
LaplaceSPT::ys (C++ member), 711
LaplaceSPT::[anonymous] (C++ enum), 710
LaplaceSPT::[anonymous]::SPT_DATA (C++ enumerator), 710
LaplaceXY (C++ class), 551
LaplaceXY::~LaplaceXY (C++ function), 551
LaplaceXY::a coef (C++ member), 552
LaplaceXY::average_iterations (C++ member), 553
LaplaceXY::bcoef (C++ member), 552
LaplaceXY::bs (C++ member), 552
LaplaceXY::bvals (C++ member), 552
LaplaceXY::c-c coef (C++ member), 552
LaplaceXY::communicator (C++ function), 552
LaplaceXY::cr (C++ member), 552
LaplaceXY::finite_volume (C++ member), 553
LaplaceXY::globalIndex (C++ function), 552
LaplaceXY::include_y_derivs (C++ member), 553
LaplaceXY::indexXY (C++ member), 553
LaplaceXY::instance_count (C++ member), 553
LaplaceXY::ksp (C++ member), 552
LaplaceXY::LaplaceXY (C++ function), 551
LaplaceXY::LaplaceXYMonitor (C++ class), 553
LaplaceXY::LaplaceXYMonitor::call (C++ function), 553
LaplaceXY::LaplaceXYMonitor::laplacey (C++ member), 553
LaplaceXY::LaplaceXYMonitor::LaplaceXYMonitor (C++ function), 553
LaplaceXY::lib (C++ member), 552
LaplaceXY::localmesh (C++ member), 552
LaplaceXY::localSize (C++ function), 552
LaplaceXY::location (C++ member), 553
LaplaceXY::MatA (C++ member), 552
LaplaceXY::monitor (C++ member), 553
LaplaceXY::my_id (C++ member), 552
LaplaceXY::n_calls (C++ member), 553
LaplaceXY::nloc (C++ member), 552
LaplaceXY::nsys (C++ member), 552
LaplaceXY::output_average_iterations (C++ member), 553
LaplaceXY::pc (C++ member), 552
LaplaceXY::precon (C++ function), 551
LaplaceXY::save_performance (C++ member), 553
LaplaceXY::savePerformance (C++ function), 551
LaplaceXY::setCoefs (C++ function), 551
LaplaceXY::solve (C++ function), 551
LaplaceXY::x_inner_dirichlet (C++ member), 553
LaplaceXY::x_outer_dirichlet (C++ member), 553
LaplaceXY::xstart (C++ member), 552
LaplaceXY::xs (C++ member), 552
LaplaceXY::xvals (C++ member), 552
LaplaceXY::y_bdry (C++ member), 553
LaplaceXZ (C++ class), 556
LaplaceXZ::~LaplaceXZ (C++ function), 556
line_from_points_poly() (in module zoidberg.rline), 777
Literal (C++ class), 411
Literal::Literal (C++ function), 412
Literal::Literal (C++ function), 412
Literal::operator() (C++ function), 412
Literal::type (C++ type), 412
Literal::val (C++ member), 412
LoadVarOp (C++ class), 514
LoadVarOp::LoadVarOp (C++ function), 514
LoadVarOp::run (C++ function), 514
LoadVarOp::var2D (C++ function), 514
LoadVarOp::var3D (C++ function), 514
local_N_sum (C++ function), 693
log (C++ function), 449
lowercase (C++ function), 723, 726
lowercasequote (C++ function), 723, 727
lowPass (C++ function), 435, 438
M
MACRO_FOR_EACH (C macro), 558
MACRO_FOR_EACH_FN (C macro), 558
MagneticField (class in zoidberg.field), 762
main (C++ function), 332
mask (C++ function), 663
mask_x (C++ function), 705, 706
Matrix (C++ class), 728
Matrix::begin (C++ function), 728
Matrix::data (C++ member), 729
Matrix::data_type (C++ type), 728
Matrix::empty (C++ function), 729
Matrix::end (C++ function), 728
Matrix::ensureUnique (C++ function), 729
Matrix::getData (C++ function), 729
Matrix::Matrix (C++ function), 728
Matrix::n1 (C++ member), 729
Matrix::n2 (C++ member), 729
Matrix::operator() (C++ function), 728
Matrix::operator= (C++ function), 728
Matrix::reallocate (C++ function), 728
Matrix::shape (C++ function), 728
Matrix::size_type (C++ type), 728
max (C++ function), 448
MAXGM (C macro), 581
MAXREGIONBLOCKSIZE (C macro), 661
MAYBE_UNUSED (C macro), 723
mean (C++ function), 448
Mesh (C++ class), 560
Mesh::Mesh (C++ function), 561
Mesh::addBoundary (C++ function), 567
Mesh::addBoundaryPar (C++ function), 567
Mesh::addRegion (C++ function), 571
Mesh::addRegion2D (C++ function), 571
Mesh::addRegion3D (C++ function), 571
Mesh::addRegionPerp (C++ function), 572
Mesh::calcParallelSlices_on_communicate (C++ member), 574
Mesh::canToFromFieldAligned (C++ function), 571
Mesh::communicate (C++ function), 563
Mesh::communicateXZ (C++ function), 563
Mesh::coords_map (C++ member), 574
Mesh::create (C++ function), 573
Mesh::createDefaultCoordinates (C++ function), 574
Mesh::createDefaultRegions (C++ function), 572
Mesh::derivs_init (C++ function), 574
Mesh::DownXSPLITindex (C++ function), 566
Mesh::fft_derivs_filter (C++ member), 573
Mesh::firstX (C++ function), 564
Mesh::firstY (C++ function), 566
Mesh::fromFieldAligned (C++ function), 571
Mesh::get (C++ function), 561, 562
Mesh::getAllowedStaggerLoc (C++ function), 569
Mesh::getBoundaries (C++ function), 567
Mesh::getBoundariesPar (C++ function), 567
Mesh::getCoordinates (C++ function), 568
Mesh::getCoordinatesSmart (C++ function), 569
Mesh::getGlobalXIndex (C++ function), 568
Mesh::getGlobalXIndexNoBoundaries (C++ function), 568
Mesh::getGlobalYIndex (C++ function), 568
Mesh::getGlobalYIndexNoBoundaries (C++ function), 568
Mesh::getGlobalZIndex (C++ function), 568
Mesh::getGlobalZIndexNoBoundaries (C++ function), 568
Mesh::getLocalXIndex (C++ function), 568
Mesh::getLocalXIndexNoBoundaries (C++ function), 568
Mesh::getLocalYIndex (C++ function), 568
Mesh::getLocalYIndexNoBoundaries (C++ function), 568
Mesh::getLocalZIndex (C++ function), 568
Mesh::getLocalZIndexNoBoundaries (C++ function), 568
Mesh::getNguard (C++ function), 569
Mesh::getNpoints (C++ function), 569
Mesh::getNXPE (C++ function), 564
Mesh::getNYPE (C++ function), 564
Mesh::getParallelTransform (C++ function), 571
Mesh::getRegion (C++ function), 571, 572
Mesh::getRegion2D (C++ function), 571
Mesh::getRegion3D (C++ function), 571
Mesh::getRegionPerp (C++ function), 571
Mesh::getStagger (C++ function), 569
Mesh::getXcomm (C++ function), 565
Mesh::getXProcIndex (C++ function), 564
Mesh::getYcomm (C++ function), 565
Mesh::addBoundaryPar (C++ function), 567
Mesh::createDefaultCoordinates (C++ function), 574
Mesh::createDefaultRegions (C++ function), 572
Mesh::derivs_init (C++ function), 574
Mesh::DownXSPLITindex (C++ function), 566
Mesh::fft_derivs_filter (C++ member), 573
Mesh::firstX (C++ function), 564
Mesh::firstY (C++ function), 566
Mesh::fromFieldAligned (C++ function), 571
Mesh::get (C++ function), 561, 562
Mesh::getAllowedStaggerLoc (C++ function), 569
Mesh::getBoundaries (C++ function), 567
Mesh::getBoundariesPar (C++ function), 567
Mesh::getCoordinates (C++ function), 568
Mesh::getCoordinatesSmart (C++ function), 569
Mesh::getGlobalXIndex (C++ function), 568
Mesh::getGlobalXIndexNoBoundaries (C++ function), 568
Mesh::getGlobalYIndex (C++ function), 568
Mesh::getGlobalYIndexNoBoundaries (C++ function), 568
Mesh::getGlobalZIndex (C++ function), 568
Mesh::getGlobalZIndexNoBoundaries (C++ function), 568
Mesh::getLocalXIndex (C++ function), 568
Mesh::getLocalXIndexNoBoundaries (C++ function), 568
Mesh::getLocalYIndex (C++ function), 568
Mesh::getLocalYIndexNoBoundaries (C++ function), 568
Mesh::getLocalZIndex (C++ function), 568
Mesh::getLocalZIndexNoBoundaries (C++ function), 568
Mesh::getNguard (C++ function), 569
Mesh::getNpoints (C++ function), 569
Mesh::getNXPE (C++ function), 564
Mesh::getNYPE (C++ function), 564
Mesh::getParallelTransform (C++ function), 571
Mesh::getRegion (C++ function), 571, 572
Mesh::getRegion2D (C++ function), 571
Mesh::getRegion3D (C++ function), 571
Mesh::getRegionPerp (C++ function), 571
Mesh::getStagger (C++ function), 569
Mesh::getXcomm (C++ function), 565
Mesh::getXProcIndex (C++ function), 564
Mesh::getYcomm (C++ function), 565
Mesh::getYProcIndex (C++ function), 564
Mesh::GlobalNx (C++ member), 572
Mesh::GlobalNy (C++ member), 572
Mesh::GlobalNz (C++ member), 572
Mesh::GlobalIX (C++ function), 567
Mesh::GlobalY (C++ function), 567
Mesh::hasBndryLowerY (C++ function), 567
Mesh::hasBndryUpperY (C++ function), 567
Mesh::hasBranchCutLower (C++ function), 565
Mesh::hasBranchCutUpper (C++ function), 565
Mesh::hasRegion2D (C++ function), 571
Mesh::hasRegion3D (C++ function), 571
Mesh::hasRegionPerp (C++ function), 571
Mesh::IncIntShear (C++ member), 573
Mesh::ind2Dto3D (C++ function), 572
Mesh::ind3Dto2D (C++ function), 572
Mesh::ind3DtoPerp (C++ function), 572
Mesh::indexD4DX (C++ function), 569
Mesh::indexD4DY (C++ function), 569
Mesh::indexD4DZ (C++ function), 570
Mesh::indexDDX (C++ function), 569
Mesh::indexDDY (C++ function), 569
Mesh::indexDDZ (C++ function), 569
Mesh::indexD2DZ2 (C++ function), 569
Mesh::indexD2DY2 (C++ function), 569
Mesh::indexD2DX2 (C++ function), 570
Mesh::indexD4DX4 (C++ function), 569
Mesh::indexD4DY4 (C++ function), 569
Mesh::indexD4DZ4 (C++ function), 570
Mesh::indexVDDX (C++ function), 569
Mesh::indexVDDY (C++ function), 569
Mesh::indexVDDZ (C++ function), 570
Mesh::indexPerpto3D (C++ function), 572
Mesh::irecvXIn (C++ function), 565
Mesh::irecvXOut (C++ function), 565
Mesh::irecvYInDest (C++ function), 566
Mesh::irecvYOutDest (C++ function), 566
Mesh::isDataSourceFile (C++ function), 563
Mesh::iterateBndryLowerInnerY (C++ function), 567
Mesh::iterateBndryLowerOuterY (C++ function), 567
Mesh::iterateBndryLowerY (C++ function), 567
Mesh::iterateBndryUpperInnerY (C++ function), 567
Mesh::iterateBndryUpperOuterY (C++ function), 567
Mesh::iterateBndryUpperY (C++ function), 567
Mesh::lastX (C++ function), 564
Mesh::lastY (C++ function), 566
Mesh::load (C++ function), 561
Mesh::LocalNx (C++ member), 573
Mesh::LocalNy (C++ member), 573
Mesh::LocalNz (C++ member), 573
Mesh::map3Dto2D (C++ function), 572
Mesh::maxRegionBlocksize (C++ member), 573
Mesh::Mesh (C++ function), 561
Mesh::msg_len (C++ function), 574
Mesh::numberOfXPoints (C++ member), 573
Mesh::NXPE (C++ member), 572
Mesh::OffsetX (C++ member), 572
Mesh::OffsetY (C++ member), 572
Mesh::OffsetZ (C++ member), 572
Mesh::options (C++ member), 574
Mesh::outputVars (C++ function), 561
Mesh::PE_XIND (C++ member), 572
Mesh::periodicX (C++ member), 572
Mesh::periodicY (C++ function), 565
Mesh::readInts (C++ function), 574
Mesh::recalculateStaggeredCoordinates (C++ function), 569
Mesh::receiveFromProc (C++ function), 564
Mesh::regionMap2D (C++ member), 574
Mesh::regionMap3D (C++ member), 574
Mesh::regionMapPerp (C++ member), 574
Mesh::send (C++ function), 563
Mesh::sendToProc (C++ function), 563
Mesh::sendXIn (C++ function), 564
Mesh::sendXOut (C++ function), 564
Mesh::sendYInDest (C++ function), 566
Mesh::sendYOutDest (C++ function), 566
Mesh::sendYOutIndest (C++ function), 566
Mesh::sendYOutIndest (C++ function), 566
Mesh::sourceHasXBoundaryGuards (C++ member), 563
Mesh::sourceHasYBoundaryGuards (C++ function), 563
Mesh::sourceHasYBoundaryGuards (C++ function), 563
Mesh::sourceHasVar (C++ function), 563
Mesh::StaggerGrids (C++ member), 573
Mesh::toFieldAligned (C++ function), 570, 571
Mesh::UpXSplitIndex (C++ function), 566
Mesh::wait (C++ function), 563
Mesh::xend (C++ member), 573
Mesh::XGLOBAL (C++ function), 567
Mesh::XLOCAL (C++ function), 568
Mesh::xstart (C++ member), 573
Mesh::yend (C++ member), 573
Mesh::YGLOBAL (C++ function), 567
Mesh::YLOCAL (C++ function), 568
Mesh::ySize (C++ function), 566
Mesh::ystart (C++ member), 573
Mesh::zend (C++ member), 573
Ncxx4::read (C++ function), 599
Ncxx4::write_rec (C++ function), 599
Ncxx4::write_rec_perp (C++ function), 599
Ncxx4::x0 (C++ member), 602
Ncxx4::xDim (C++ member), 601
Ncxx4::y0 (C++ member), 602
Ncxx4::yDim (C++ member), 601
Ncxx4::z0 (C++ member), 602
Ncxx4::zDim (C++ member), 601
Ncxx4::isValid (C++ function), 601
Ncxx4::getDimVec (C++ function), 601
Ncxx4::getAttribute (C++ function), 601
Ncxx4::fname (C++ function), 602
Ncxx4::flush (C++ function), 602
Ncxx4::~Ncxx4 (C++ class), 602
Ncxx4::dimList (C++ function), 602
Ncxx4::addVarBoutReal (C++ function), 603
Ncxx4::addVarField3D (C++ function), 603
Ncxx4::addVarField2D (C++ function), 603
Ncxx4::addVarFieldPerp (C++ function), 604
Ncxx4::addVarIntVec (C++ function), 603
Ncxx4::addVarString (C++ function), 603
Ncxx4::appending (C++ member), 607
Ncxx4::close (C++ function), 607
Ncxx4::dataFile (C++ member), 607
Ncxx4::default_rec (C++ member), 607
Ncxx4::dimList (C++ member), 607
Ncxx4::filename (C++ function), 603
Ncxx4::flush (C++ function), 603
Ncxx4::fname (C++ member), 607
Ncxx4::getAttribute (C++ function), 606
Ncxx4::getDimVec (C++ function), 607
Ncxx4::getRecDimVec (C++ function), 607
Ncxx4::getSize (C++ function), 603
Ncxx4::isValid (C++ function), 603
Ncxx4::lowPrecision (C++ member), 607
Ncxx4::addVarInt (C++ function), 603
Ncxx4::addVarFieldPerp (C++ function), 604
Ncxx4::addVarReal (C++ function), 603
Ncxx4::offset (C++ function), 663
Ncxx4::openw (C++ function), 599
NetCDF (C++ type), 602
nl_filter (C++ function), 686–688
nl_filter_x (C++ function), 687, 688
nl_filter_y (C++ function), 687, 688
nl_filter_z (C++ function), 687, 688
NoBoundary (class in zoidberg boundary), 759
NumberOfPoloidalGrids () (zoidberg.grid.Grid method), 768

OptionParser::~OptionParser (C++ function), 608
OptionParser::write (C++ function), 609
OptionParser::getOptionParser (C++ function), 609
OptionParser::read (C++ function), 609
OptionParser::getOptionParser (C++ function), 609
OptionParser::read (C++ function), 609
OptionParser::write (C++ function), 610
Options::set (C++ function), 616
Options::~Options (C++ function), 611

820
Options::as (C++ function), 612, 615
Options::as<std::string> (C++ function), 609
Options::assign (C++ function), 612, 614, 615
Options::attributes (C++ member), 615
Options::AttributeType (C++ class), 617
Options::AttributeType::~AttributeType (C++ function), 617
Options::AttributeType::as (C++ function), 617
Options::AttributeType::AttributeType (C++ function), 617
Options::AttributeType::Base (C++ type), 617
Options::AttributeType::operator T (C++ function), 617
Options::AttributeType::operator= (C++ function), 617
Options::child (C++ member), 616
Options::cleanCache (C++ function), 616
Options::cleanup (C++ function), 616
Options::DEFAULT_SOURCE (C++ member), 617
Options::doc (C++ function), 614
Options::force (C++ function), 612
Options::forceSet (C++ function), 613
Options::full_name (C++ member), 616
Options::get (C++ function), 613
Options::getChildren (C++ function), 614
Options::getChildren (C++ function), 614
Options::getParent (C++ function), 613
Options::getRoot (C++ function), 616
Options::getSection (C++ function), 613
Options::hasAttribute (C++ function), 611
Options::is_section (C++ member), 616
Options::is_value (C++ member), 616
Options::isSection (C++ function), 614
Options::isSet (C++ function), 612, 613
Options::isValue (C++ function), 614
Options::operator T (C++ function), 612
Options::operator= (C++ function), 612
Options::operator= (C++ function), 613
Options::operator= (C++ function), 613
Options::operator[] (C++ function), 611, 612
Options::Options (C++ function), 611
Options::OptionValue (C++ struct), 617
Options::OptionValue::OptionValue (C++ function), 618
Options::OptionValue::source (C++ member), 618
Options::OptionValue::used (C++ member), 618
Options::OptionValue::value (C++ member), 618
Options::overrideDefault (C++ function), 613
Options::parent (C++ function), 613
Options::parent_instance (C++ member), 616
Options::printUnused (C++ function), 614
Options::root (C++ function), 616
Options::root_instance (C++ member), 617
Options::set (C++ function), 613
Options::similar (C++ function), 616
Options::str (C++ function), 614
Options::subsections (C++ function), 614
Options::value (C++ member), 615
Options::value_used (C++ member), 616
Options::values (C++ function), 614
Options::ValuesMap (C++ type), 611
Options::ValueType (C++ type), 611
Options::valueUsed (C++ function), 614
Options::withDefault (C++ function), 612, 613
OptionsReader (C++ class), 620
OptionsReader::cleanup (C++ function), 620
OptionsReader::getInstance (C++ function), 620
OptionsReader::instance (C++ member), 621
OptionsReader::parseCommandLine (C++ function), 620
OptionsReader::read (C++ function), 620
OptionsReader::write (C++ function), 620
out (gen_fieldops attribute), 482
OUT_SENT_DOWN (C++ member), 333
OUT_SENT_IN (C++ member), 333
OUT_SENT_UP (C++ member), 333
Output (C++ class), 622
output (C++ member), 621, 622
Output::_Tr (C++ type), 624
Output::~Output (C++ function), 623
Output::add (C++ function), 623
Output::buffer (C++ member), 624
Output::BUFFER_LEN (C++ member), 624
Output::buffer_len (C++ member), 624
Output::close (C++ function), 623
Output::disable (C++ function), 623
Output::enable (C++ function), 623
Output::enabled (C++ member), 624
Output::file (C++ member), 624
Output::getBase (C++ function), 624
Output::getInstance (C++ function), 624
Output::isEnabled (C++ function), 624
Output::multioutbuf_init (C++ type), 624
Output::open (C++ function), 623
Output::Output (C++ function), 623
Output::print (C++ function), 623
Output::remove (C++ function), 623
Output::vprint (C++ function), 623
Output::vwrite (C++ function), 623
Output::write (C++ function), 623
output::debug (C++ member), 621, 622
output::error (C++ member), 621, 622
output::progress (C++ member), 621, 622
output::info (C++ member), 621, 622
output::warn (C++ member), 621, 622
outside() (zoidberg.boundary.NoBoundary method), 760
outside() (zoidberg.boundary.PolygonBoundaryXZ method), 760
outside() (zoidberg.boundary.RectangularBoundaryXZ method), 760

Par
ParallelTransform (C++ class), 632
ParallelTransform::-ParallelTransform (C++ function), 632
ParallelTransform::calcParallelSlices (C++ function), 632
ParallelTransform::calcYupYdown (C++ function), 632
ParallelTransform::canToFromFieldAligned (C++ function), 632
ParallelTransform::checkInputGrid (C++ function), 633
ParallelTransform::fromFieldAligned (C++ function), 632
ParallelTransform::integrateParallelSlices (C++ function), 632
ParallelTransform::integrateYupYdown (C++ function), 632
ParallelTransform::mesh (C++ member), 633
ParallelTransform::outputVars (C++ function), 632
ParallelTransform::ParallelTransform (C++ function), 632
ParallelTransform::requiresTwistShift (C++ function), 632
ParallelTransform::toFieldAligned (C++ function), 632
ParallelTransformIdentity (C++ class), 633
ParallelTransformIdentity::calcParallelSlices (C++ function), 633
ParallelTransformIdentity::canToFromFieldAligned (C++ function), 633
ParallelTransformIdentity::checkInputGrid (C++ function), 634
ParallelTransformIdentity::fromFieldAligned (C++ function), 633
ParallelTransformIdentity:: ParallelTransformIdentity (C++ function), 633
ParallelTransformIdentity::requiresTwistShift (C++ function), 633
ParallelTransformIdentity::toFieldAligned (C++ function), 633
PARDERIVCYCLIC (C macro), 542
ParDerivFactory (C++ class), 544
ParDerivFactory::createInvertPar (C++ function), 544
ParDerivFactory::getInstance (C++ function), 544
ParDerivFactory::instance (C++ member), 545
ParDerivFactory::ParDerivFactory (C++ function), 545
ParseException (C++ class), 422
ParseException::~ParseException (C++ function), 422
ParseException::message (C++ member), 422
ParseException::what (C++ function), 422
parser (gen_fieldops attribute), 482
PETSC_VERSION_GE (C macro), 647
PetscLib (C++ class), 647
PetscLib::~PetscLib (C++ function), 647
PetscLib::cleanup (C++ function), 648
PetscLib::count (C++ member), 648
PetscLib::help (C++ member), 648
PetscLib::options_prefix (C++ member), 648
PetscLib::pargc (C++ member), 648
PetscLib::pargv (C++ member), 648
PetscLib::PetscLib (C++ function), 647
PetscLib::setArgs (C++ function), 648
PetscLib::setOptionsFromFile (C++ function), 647
PetscLib::setPetscOptions (C++ function), 648
PetscLib::USER_EVENT (C++ member), 648
PetscMonitor (C++ function), 639, 640
PetscSNESMonitor (C++ function), 639, 640
PetscSolver (C++ class), 640
PetscSolver::-PetscSolver (C++ function), 641
PetscSolver::adaptive (C++ member), 642
PetscSolver::bout_snes_time (C++ member), 642
PetscSolver::diagnose (C++ member), 642
PetscSolver::init (C++ function), 641
PetscSolver::init_event (C++ member), 641
PetscSolver::interpolate (C++ member), 642
PetscSolver::J (C++ member), 641
PetscSolver::jac (C++ function), 641
PetscSolver::Jmf (C++ member), 641
PetscSolver::lib (C++ member), 641
PetscSolver::loop_event (C++ member), 641
PetscSolver::matfdcoloring (C++ member), 642
PetscSolver::next_output (C++ member), 642
PetscSolver::nout (C++ member), 642
PetscSolver::output_flag (C++ member), 642
PetscSolver::output_name (C++ member), 642
PetscSolver::PetscSolver (C++ function), 641
PetscSolver::pre (C++ function), 641
PetscSolver::prev_linear_its (C++ member), 642
PetscSolver::rhs (C++ function), 641
PetscSolver::run (C++ function), 641
PetscSolver::shift (C++ member), 641
PetscSolver::snes_list (C++ member), 642
PetscSolver::solver_event (C++ member), 641
PetscSolver::state (C++ member), 641
PetscSolver::ts (C++ member), 641
PetscSolver::ts_time (C++ member), 641
RK4Solver::f2 (C++ member), 671
RK4Solver::getCurrentTimestep (C++ function), 671
RK4Solver::init (C++ function), 671
RK4Solver::k1 (C++ member), 672
RK4Solver::k2 (C++ member), 672
RK4Solver::k3 (C++ member), 672
RK4Solver::k4 (C++ member), 672
RK4Solver::k5 (C++ member), 672
RK4Solver::max_timestep (C++ member), 671
RK4Solver::mxstep (C++ member), 671
RK4Solver::neq (C++ member), 672
RK4Solver::nlocal (C++ member), 672
RK4Solver::nsteps (C++ member), 671
RK4Solver::out_timestep (C++ member), 671
RK4Solver::resetInternalFields (C++ function), 671
RK4Solver::timestep (C++ member), 672
RK4Solver::RK4Solver (C++ class), 672
RK4Solver::run (C++ function), 671
RK4Solver::setMaxTimestep (C++ function), 671
RK4Solver::take_step (C++ function), 671
RKGenericSolver::take_step (C++ function), 674
RKGenericSolver::timestep (C++ member), 674
RKGenericSolver::tmpState (C++ member), 674
RKScheme (C++ class), 674
RKScheme::~RKScheme (C++ function), 675
RKScheme::adaptive (C++ member), 676
RKScheme::atol (C++ member), 676
RKScheme::constructOutput (C++ function), 675
RKScheme::constructOutputs (C++ function), 675
RKScheme::dtfac (C++ member), 676
RKScheme::followHighOrder (C++ member), 676
RKScheme::getErr (C++ function), 675
RKScheme::getNumOrders (C++ function), 675
RKScheme::getStageCount (C++ function), 675
RKScheme::getType (C++ function), 675
RKScheme::init (C++ function), 675
RKScheme::label (C++ member), 676
RKScheme::max_timestep (C++ function), 675
RKGenericSolver::neq (C++ member), 676
RKGenericSolver::nlocal (C++ member), 676
RKGenericSolver::numOrders (C++ member), 676
RKGenericSolver::numStages (C++ member), 676
RKGenericSolver::order (C++ member), 676
RKScheme::printButcherTableau (C++ function), 676
RKScheme::resultAlt (C++ member), 676
RKScheme::resultCoeffs (C++ member), 676
RKScheme::RKScheme (C++ function), 675
RKScheme::rtol (C++ member), 676
RKScheme::setCurState (C++ function), 675
RKScheme::setCurTime (C++ function), 675
RKScheme::setOutputStates (C++ function), 675
RKScheme::stageCoeffs (C++ member), 676
RKScheme::steps (C++ member), 675
RKScheme::timeCoeffs (C++ member), 676
RKScheme::updateTimestep (C++ function), 675
RKScheme::verifyCoeffs (C++ function), 676
RKScheme::zeroSteps (C++ function), 676
RKSCHEME_CASHKARP (C macro), 674
RKSCHEME_RK4 (C macro), 674
RKSCHEME_RKF34 (C macro), 674
RKSCHEME_RKF45 (C macro), 674
RKSchemeFactory (C++ class), 676
RKSchemeFactory::createRKScheme (C++ function), 677
RKSchemeFactory::createRKScheme (C++ function), 677
RKSchemeFactory::getDefaultRKSchemeType (C++ function), 677
RKSchemeFactory::getCurSchemeType (C++ member), 677
RKSchemeFactory::getCurSchemeType (C++ function), 677
RKSchemeFactory::getCurSchemeType (C++ function), 677
RKSchemeFactory::instance (C++ member), 677
RKSchemeFactory::instance (C++ member), 677
RKSchemeFactory::RKSchemeFactory (C++ function), 677
RKSchemeFactory::RKSchemeFactory (C++ function), 677
RKSchemeFactory::RKSchemeFactory (C++ function), 677
RunMetrics (C++ struct), 577
uuids::uuid::variant (C++ function), 735
uuids::uuid::version (C++ function), 735
uuids::uuid_name_generator (C++ class), 736
uuids::uuid_name_generator:: hasher (C++ member), 737
uuids::uuid_name_generator::make_uuid (C++ function), 736
uuids::uuid_name_generator::nsuuid (C++ member), 737
uuids::uuid_name_generator::operator () (C++ function), 736
uuids::uuid_name_generator::process_characters (C++ function), 736
uuids::uuid_name_generator::reset (C++ function), 736
uuids::uuid_name_generator::uuid_name_generator (C++ function), 736
uuids::uuid_namespace_dns (C++ member), 734
uuids::uuid_namespace_oid (C++ member), 734
uuids::uuid_namespace_url (C++ member), 734
uuids::uuid_namespace_x500 (C++ type), 733
uuids::uuid_name_generator (C++ class), 737
uuids::uuid_time_generator (C++ class), 737
uuids::uuid_time_generator::device_address (C++ member), 737
uuids::uuid_time_generator::get_mac_address (C++ function), 737
uuids::uuid_time_generator::get_time_intervals (C++ function), 737
uuids::uuid_time_generator::has_mac_address (C++ member), 737
uuids::uuid_time_generator::mac_address (C++ type), 737
uuids::uuid_time_generator::operator () (C++ function), 737
uuids::uuid_time_generator::uuid_time_generator (C++ function), 737
uuids::uuid_variant (C++ enum), 733
uuids::uuid_variant::ncs (C++ enumerator), 733
uuids::uuid_variant::reserved (C++ enumerator), 733
uuids::uuid_variant::rfc (C++ enumerator), 733
uuids::uuid_variant::version (C++ enum), 733
uuids::uuid_variant::dce_security (C++ enumerator), 733
uuids::uuid_variant::name_based_md5 (C++ enumerator), 733
uuids::uuid_version::time_based (C++ enumerator), 733

V

V_dot_Grad (C++ function), 743, 745
VARIANT (C macro), 609
VDDX (C++ function), 377, 390
VDDY (C++ function), 377, 378, 390, 391
VDDZ (C++ function), 378, 391–393
VDERIV_FUNC_REGION_ENUM_TO_STRING (C macro), 381

Vector2D (C++ class), 747
Vector2D::-Vector2D (C++ function), 747
Vector2D::accept (C++ function), 749
Vector2D::applyBoundary (C++ function), 749
Vector2D::applyTDerivBoundary (C++ function), 749
Vector2D::BoutRealSize (C++ function), 749
Vector2D::byteSize (C++ function), 749
Vector2D::covariant (C++ member), 749
Vector2D::deriv (C++ member), 750
Vector2D::getLocation (C++ function), 749
Vector2D::is3D (C++ function), 749
Vector2D::isReal (C++ function), 749
Vector2D::location (C++ member), 750
Vector2D::operator* (C++ function), 748, 749
Vector2D::operator= (C++ function), 748
Vector2D::operator+= (C++ function), 748
Vector2D::operator* (C++ function), 748
Vector2D::operator/= (C++ function), 748
Vector2D::operator== (C++ function), 748
Vector2D::operator-= (C++ function), 748
Vector2D::operator- (C++ function), 748
Vector2D::setLocation (C++ function), 749
Vector2D::timeDeriv (C++ function), 748
Vector2D::toContravariant (C++ function), 747
Vector2D::Vector2D (C++ function), 747
Vector2D::Vector2D (C++ class), 752
Vector2D::Vector3D (C++ class), 752
Vector2D::Vector3D::Vector2D (C++ function), 754
Vector2D::Vector3D::accept (C++ function), 754
Vector2D::Vector3D::applyBoundary (C++ function), 754
Vector2D::Vector3D::applyTDerivBoundary (C++ function), 754
Vector2D::Vector3D::BoutRealSize (C++ function), 754
Vector2D::Vector3D::byteSize (C++ function), 754
Vector2D::Vector3D::covariant (C++ member), 754
Vector2D::Vector3D::deriv (C++ member), 755
Vector2D::Vector3D::getLocation (C++ function), 754
Vector3D::Vector3D::is3D (C++ function), 754

Index 833
Vector3D::isReal (C++ function), 754
Vector3D::location (C++ member), 755
Vector3D::operator* (C++ function), 753, 754
Vector3D::operator*= (C++ function), 753
Vector3D::operator+ (C++ function), 753
Vector3D::operator+= (C++ function), 753
Vector3D::operator/ (C++ function), 753, 754
Vector3D::operator/= (C++ function), 753
Vector3D::operator- (C++ function), 752, 753
Vector3D::operator-= (C++ function), 753
Vector3D::setLocation (C++ function), 754
Vector3D::timeDeriv (C++ function), 752
Vector3D::toContravariant (C++ function), 752
Vector3D::toCovariant (C++ function), 752
Vector3D::Vector3D (C++ function), 752
Vector3D::x (C++ member), 754
Vector3D::y (C++ member), 754
Vector3D::z (C++ member), 754
verifyNumPoints (C++ function), 303
VMEC (class in zoidberg.field), 765
Vol_Integral (C++ function), 686, 688
Vpar_Grad_par (C++ function), 399, 404
Vpar_Grad_par_LCtoC (C++ function), 406
vUpDown (C++ function), 519

W
WENO_SMALL (C++ member), 523
where (C++ function), 755
wmultiostream (C++ type), 586
WRAP_ENUM (C macro), 522

Y
YDirectionType (C++ enum), 324
YDirectionType::Aligned (C++ enumerator), 325
YDirectionType::Standard (C++ enumerator), 325
YDirectionTypeFromString (C++ function), 323, 326

Z
Z (zoidberg.poloidal_grid.PoloidalGrid attribute), 771
Z (zoidberg.poloidal_grid.RectangularPoloidalGrid attribute), 772
Z (zoidberg.poloidal_grid.StructuredPoloidalGrid attribute), 773
Z (zoidberg.rzline.RZline attribute), 775
ZDirectionType (C++ enum), 325
ZDirectionType::Average (C++ enumerator), 325
ZDirectionType::Standard (C++ enumerator), 325
ZDirectionTypeFromString (C++ function), 323, 326
ZERO (C macro), 286, 349, 511
ZERO (C++ member), 658
zeroFrom (C++ function), 447, 747, 752
zgbsv_ (C++ function), 549
zgtsv_ (C++ function), 549
zoidberg.boundary module, 759
zoidberg.field module, 760
zoidberg.fieldtracer module, 766
zoidberg.grid module, 768
zoidberg.plot module, 769
zoidberg.poloidal_grid module, 771
zoidberg.rzline module, 775
Zvalue () (zoidberg.rzline.RZline method), 775

834 Index