Getting started

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

2 Getting started
   2.1 Pre-built binaries ............................................................... 7
   2.2 Obtaining BOUT++ .............................................................. 8
   2.3 Installing dependencies ...................................................... 8
   2.4 Configuring BOUT++ ............................................................ 10
   2.5 CMake .............................................................................. 11
   2.6 Natural Language Support .................................................... 12
   2.7 Configuring analysis routines ............................................... 13
   2.8 Compiling BOUT++ .............................................................. 14
   2.9 Running the test suite ......................................................... 14
   2.10 Installing BOUT++ (experimental) ........................................ 15

3 Advanced installation options .................................................. 17
   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 ........................................................................ 21
   3.5 OpenMP .......................................................................... 22
   3.6 SUNDIALS ........................................................................ 23
   3.7 PETSc .............................................................................. 24
   3.8 LAPACK .......................................................................... 26
   3.9 MPI compilers .................................................................... 26
   3.10 Installing FFTW from source ................................................. 27
   3.11 Compiling and running under AIX ........................................ 27
   3.12 Issues ............................................................................. 29

4 Running BOUT++ .................................................................. 31
   4.1 Quick start ...................................................................... 31
   4.2 Analysing the output using Python ....................................... 32
   4.3 Natural language support ...................................................... 34
   4.4 When things go wrong ......................................................... 34
   4.5 Startup output ................................................................... 35
   4.6 Per-timestep output .............................................................. 39
   4.7 Restarting runs ................................................................... 39
4.8 Stopping simulations ................................................... 40
4.9 Manipulating restart files ........................................... 41

5 BOUT++ physics models .................................................. 43
5.1 Heat conduction ......................................................... 43
5.2 Magnetohydrodynamics (MHD) ......................................... 47
5.3 Reduced MHD .......................................................... 56
5.4 Logging output ........................................................ 58
5.5 Updating Physics Models from v3 to v4 .............................. 58
5.6 More examples ......................................................... 59

6 Makefiles and compiling BOUT++ ...................................... 61
6.1 Executables example .................................................. 61
6.2 Modules example ...................................................... 62
6.3 bout-config script ..................................................... 62

7 Variable initialisation .................................................... 65
7.1 Initialisation of time evolved variables .............................. 66
7.2 Initialising variables with the FieldFactory class ................. 70
7.3 Adding a new function .............................................. 70
7.4 Parser internals ....................................................... 72

8 Boundary conditions ..................................................... 75
8.1 Boundary conditions for non-orthogonal grids ..................... 76
8.2 Relaxing boundaries ................................................ 76
8.3 Shifted boundaries ................................................... 77
8.4 Changing the width of boundaries .................................. 77
8.5 Examples ............................................................. 77
8.6 Boundary regions ..................................................... 78
8.7 Boundary regions ..................................................... 78
8.8 Boundary operations ................................................ 79
8.9 Boundary modifiers .................................................. 81
8.10 Boundary factory .................................................... 81

9 Testing ........................................................................ 85
9.1 Automated tests and code coverage ................................. 85
9.2 Unit tests ............................................................... 85
9.3 Integrated tests ....................................................... 86
9.4 Method of Manufactured Solutions ................................. 86
9.5 Timing ................................................................. 87

10 BOUT++ options ........................................................ 89
10.1 BOUT.inp input file ................................................. 89
10.2 Command line options ............................................. 91
10.3 General options ..................................................... 92
10.4 Communications ................................................... 93
10.5 Differencing methods ............................................... 93
10.6 Model-specific options ............................................. 94
10.7 Input and Output .................................................... 94
10.8 Implementation ..................................................... 95
10.9 Reading options .................................................... 98
10.10 Reading and writing to NetCDF ................................. 99
10.11 FFT ............................................................... 100
10.12 Types for multi-valued options ................................... 100
# Generating input grids

11.1 BOUT++ Topology  
11.2 3D variables  
11.3 From EFIT files  
11.4 From ELITE and GATO files  
11.5 Generating equilibria  
11.6 Zoidberg grid generator

## Post-processing

12.1 Python routines  
12.2 Python analysis routines  
12.3 Reading BOUT++ output into IDL  
12.4 Summary of IDL file routines  
12.5 IDL analysis routines  
12.6 Matlab routines  
12.7 Mathematica routines  
12.8 Octave routines

## The python boutcore module

13.1 Installing  
13.2 Purpose  
13.3 State  
13.4 Functions  
13.5 Examples  
13.6 Functions - undocumented  
13.7 Functions - special and inherited

## Time integration

14.1 Options  
14.2 CVODE  
14.3 IMEX-BDF2  
14.4 Split-RK  
14.5 ODE integration  
14.6 Preconditioning  
14.7 Jacobian function  
14.8 DAE constraint equations  
14.9 IMEX-BDF2  
14.10 Monitoring the simulation output  
14.11 Implementation internals

## Parallel Transforms

15.1 Field-aligned grid  
15.2 Shifted metric  
15.3 FCI method

## Laplacian inversion

16.1 Usage of the laplacian inversion  
16.2 Numerical implementation  
16.3 Implementation internals  
16.4 LaplaceXY  
16.5 LaplaceXZ

## Differential operators

17.1 Differencing methods  
17.2 User registered methods
17.3 Mixed second-derivative operators .............................................. 170
17.4 Non-uniform meshes ................................................................... 171
17.5 General operators ........................................................................ 172
17.6 Clebsch operators ....................................................................... 172
17.7 The bracket operators .................................................................. 173
17.8 Finite volume, conservative finite difference methods ...................... 173
17.9 Derivatives of the Fourier transform .............................................. 177

18 Algebraic operators ...................................................................... 179

19 Staggered grids ............................................................................. 181

20 Eigenvalue solver .......................................................................... 185
   20.1 Configuring with SLEPc ............................................................... 185
   20.2 SLEPc options .......................................................................... 185
   20.3 Examples .................................................................................. 185

21 Nonlocal heat flux models ................................................................. 187
   21.1 Spitzer-Harm heat flux ................................................................. 187
   21.2 SNB model ............................................................................ 188

22 Invertable operators ...................................................................... 193

23 PETSc solvers .............................................................................. 197

24 Field-aligned coordinates ................................................................. 199
   24.1 Introduction ........................................................................... 199
   24.2 Orthogonal toroidal coordinates .................................................. 199
   24.3 Field-aligned coordinates .......................................................... 200
   24.4 Shifted radial derivatives ............................................................ 210
   24.5 Useful identities ........................................................................ 212
   .1 Differential geometry .................................................................. 213
   .2 Derivation of operators in the BOUT++ Clebsch system ................. 214
   .3 Divergence of ExB velocity .......................................................... 219

A BOUT++ preconditioning ................................................................. 221
   A.1 Introduction ............................................................................. 221
   A.2 Physics problems ..................................................................... 222
   A.3 Jacobian-vector multiply ............................................................ 224
   A.4 Preconditioner-vector multiply .................................................... 224
   A.5 Stencils ................................................................................... 225
   A.6 Jacobian calculation .................................................................. 225

B Geometry and Differential Operator ................................................. 227
   B.1 Geometry .............................................................................. 227
   B.2 Geometry and Differential Operators .......................................... 227

C Contributing to BOUT++ ................................................................. 233
   C.1 House rules ............................................................................ 233
   C.2 Development workflow using Git ............................................... 234
   C.3 Coding Style .......................................................................... 235

D Git crash course ............................................................................. 239
   D.1 Accessing github from behind a firewall .................................... 240
   D.2 Creating a private repository ...................................................... 240
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Code layout</td>
<td>243</td>
</tr>
<tr>
<td>E.1</td>
<td>Directories</td>
<td>244</td>
</tr>
<tr>
<td>F</td>
<td>Arrays, scalar and vector field types</td>
<td>249</td>
</tr>
<tr>
<td>F.1</td>
<td>FieldData</td>
<td>249</td>
</tr>
<tr>
<td>F.2</td>
<td>Field</td>
<td>250</td>
</tr>
<tr>
<td>F.3</td>
<td>Vector</td>
<td>251</td>
</tr>
<tr>
<td>F.4</td>
<td>dcomplex</td>
<td>251</td>
</tr>
<tr>
<td>F.5</td>
<td>Memory management</td>
<td>251</td>
</tr>
<tr>
<td>F.6</td>
<td>Global field gather / scatter</td>
<td>252</td>
</tr>
<tr>
<td>F.7</td>
<td>Iterating over fields</td>
<td>253</td>
</tr>
<tr>
<td>F.8</td>
<td>Iterating over ranges</td>
<td>256</td>
</tr>
<tr>
<td>F.9</td>
<td>Field2D/Field3D Arithmetic Operators</td>
<td>257</td>
</tr>
<tr>
<td>G</td>
<td>PETSc interface</td>
<td>259</td>
</tr>
<tr>
<td>G.1</td>
<td>Overall Structure</td>
<td>259</td>
</tr>
<tr>
<td>G.2</td>
<td>OperatorStencil</td>
<td>261</td>
</tr>
<tr>
<td>G.3</td>
<td>GlobalIndexer</td>
<td>261</td>
</tr>
<tr>
<td>G.4</td>
<td>PetscVector</td>
<td>263</td>
</tr>
<tr>
<td>G.5</td>
<td>PetscMatrix</td>
<td>264</td>
</tr>
<tr>
<td>G.6</td>
<td>Use With Other Parts of PETSc</td>
<td>265</td>
</tr>
<tr>
<td>H</td>
<td>Mesh</td>
<td>267</td>
</tr>
<tr>
<td>H.1</td>
<td>Grid data sources</td>
<td>267</td>
</tr>
<tr>
<td>H.2</td>
<td>Loading a mesh</td>
<td>267</td>
</tr>
<tr>
<td>H.3</td>
<td>Index ranges</td>
<td>268</td>
</tr>
<tr>
<td>H.4</td>
<td>Getting data</td>
<td>268</td>
</tr>
<tr>
<td>H.5</td>
<td>Communications</td>
<td>269</td>
</tr>
<tr>
<td>H.6</td>
<td>X communications</td>
<td>270</td>
</tr>
<tr>
<td>H.7</td>
<td>Y-Z surface communications</td>
<td>270</td>
</tr>
<tr>
<td>H.8</td>
<td>Initial profiles</td>
<td>271</td>
</tr>
<tr>
<td>H.9</td>
<td>Differencing</td>
<td>271</td>
</tr>
<tr>
<td>H.10</td>
<td>Metrics</td>
<td>271</td>
</tr>
<tr>
<td>H.11</td>
<td>Miscellaneous</td>
<td>272</td>
</tr>
<tr>
<td>I</td>
<td>File I/O</td>
<td>273</td>
</tr>
<tr>
<td>I.1</td>
<td>FieldPerp I/O</td>
<td>278</td>
</tr>
<tr>
<td>J</td>
<td>Natural Language Support</td>
<td>279</td>
</tr>
<tr>
<td>J.1</td>
<td>Marking strings for translation</td>
<td>279</td>
</tr>
<tr>
<td>J.2</td>
<td>Adding translations</td>
<td>279</td>
</tr>
<tr>
<td>K</td>
<td>Performance profiling</td>
<td>281</td>
</tr>
<tr>
<td>K.1</td>
<td>Scorep/Scalasca profiling</td>
<td>281</td>
</tr>
<tr>
<td>K.2</td>
<td>Extrae/Paraver profiling</td>
<td>283</td>
</tr>
<tr>
<td>L</td>
<td>API reference</td>
<td>287</td>
</tr>
<tr>
<td>L.1</td>
<td>BOUT++ functions (alphabetical)</td>
<td>287</td>
</tr>
<tr>
<td>L.2</td>
<td>File list</td>
<td>289</td>
</tr>
<tr>
<td>L.3</td>
<td>Python routines</td>
<td>711</td>
</tr>
<tr>
<td>L.4</td>
<td>boutdata package</td>
<td>713</td>
</tr>
<tr>
<td>L.5</td>
<td>boututils package</td>
<td>731</td>
</tr>
<tr>
<td>L.6</td>
<td>zoidberg package</td>
<td>748</td>
</tr>
<tr>
<td>M</td>
<td>Indices and tables</td>
<td>771</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 \textit{Getting started}), run the test suite (section \textit{Running the test suite}) and a few examples (section \textit{Running BOUT++}, more detail in section \textit{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 \textit{BOUT++ options}. Checking that the options are doing what you think they should be by looking at the output logs is described in section \textit{Running BOUT++}, and an overview of the IDL analysis routines for data post-processing and visualisation is given in section \textit{Post-processing}. Generating new grid files, particularly for tokamak equilibria, is described in section \textit{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 \textit{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 \textit{Differential operators} describes the differential operators and methods available; section \textit{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.
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:

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

then run:

```sh
$ 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:

```sh
$ 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. You can the latest stable version from 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:

```sh
$ 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:

```sh
$ git pull
```

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

```sh
$ git checkout next
```

### 2.3 Installing dependencies

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

1. A C++ compiler that supports C++14
2. An MPI compiler such as OpenMPI (www.open-mpi.org/), MPICH (https://www.mpich.org/) or LAM (www.lam-mpi.org/)
The FFTW-3 library (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++14 (GCC 5+).

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 `xlc` 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 install 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

```bash
$ pacman -S openmpi fftw netcdf-cxx make gcc
```

### 2.3.4 Fedora

On Fedora the required libraries can be installed by running:

```bash
$ 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:

```bash
$ module load mpi
```

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

```bash
# 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:

```bash
$ ./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:

```bash
$ ./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:

```bash
$ ./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-NetCDF support: no
HDF5 support: yes (parallel: 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:

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

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

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

You can see what build options are available with:

```bash
$ 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:

```bash
$ CC=mpicc CXX=mpic++ cmake . -B build \
    -DBOUT_USE_PETSC=ON -DPETSC_DIR=/path/to/petsc/ \
    -DBOUT_USE_SLEPC=ON -DSLEPC_DIR=/path/to/slepc/ \
    -DBOUT_USE_SUNDIALS=ON -DSUNDIALS_ROOT=/path/to/sundials \
    -DBOUT_USE_NETCDF=ON -DNetCDF_ROOT=/path/to/netcdf \
    -DBOUT_ENABLE_OPENMP=ON
```

(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.

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_MPACK_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. Similarly for fmt, using -DBOUT_USE_SYSTEM_FMT=ON and fmt_ROOT respectively. To turn off both, you can 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.

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)
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 NumPy and SciPy libraries. On Debian or Ubuntu these can be installed with:

```bash
$ sudo apt-get install python-scipy
```

which should then add all the other dependencies like NumPy. To test if everything is installed, run:

```bash
$ python -c "import scipy"
```

If not, see the SciPy website [https://www.scipy.org](https://www.scipy.org) for instructions on installing.

To do this, the path to tools/pylib should be added to the PYTHONPATH environment variable. Instructions for doing this are printed at the end of the configure script, for example:

```bash
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:

```bash
$ python -c "import boutdata"
```

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

### 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:

```bash
$ export IDL_PATH=...
```

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

```bash
$ 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 -DCHECK=2 -DSIGHANDLE -DREVISION=13571f760cecc446d907e1b6e1d7a3b1c6e0212a -DNCDF -DBOUT_HAS_PVODE
CHECKSUM = ff3fb702b13acc092613cffe3869b875
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++/*.h 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++/pylib/* Python analysis routines
- /usr/local/share/bout++/idl3lib/* 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.
CHAPTER 3

Advanced installation options

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

```
```

### 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
```

**With the tcsh shell use**

```
setenv PETSC_DIR ~farley9/projects/petsc/petsc-3.2-p1
setenv PETSC_ARCH arch-c
```

### 3.3.9 Marconi

```
module load intel intelmpi fftw lapack
module load szip 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.

**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`. 

---

### 3.3. Machine-specific installation
-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

```bash
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:**

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

- **netCDF-4.7.4:**

  ```bash
  mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/directory/to/install/netcdf -DCMAKE_BUILD_   
  →TYPE=Release ..
  make
  make install
  ```

- **netCDF-cxx4-4.3.1:**

  ```bash
  mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/directory/to/install/netcdf -DCMAKE_BUILD_   
  →TYPE=Release ..
  make
  make install
  ```

- **FFTW-3.3.8:**

  ```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:**
mkdir build
cd build

cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=/directory/to/install/
    -sundials -DMPI_ENABLE=ON ..
make
make install

• PETSc-3.13.0:

unset PETSC_DIR
./configure COPTFLAGS="-O3" CXXOPTFLAGS="-O3" FOPTFLAGS="-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!):

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

• for a debugging build:

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

### 3.3.10 Ubgl

```bash
./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. In
particular, the NetCDF-4 library can produce files in either NetCDF3 “classic” format, which is backwards-compatible
with NetCDF libraries since 1994 (version 2.3), or in the newer NetCDF4 format, which is based on (and compatible
with) HDF5. 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.

To enable NetCDF support, you will need to install NetCDF version 4.0.1 or later. Note that although the NetCDF-4
library is used for the C++ interface, by default BOUT++ writes the “classic” format. Because of this, you don’t need
to install zlib or HDF5 for BOUT++ NetCDF support to work. If you want to output to HDF5 then you need to first
install the zlib and HDF5 libraries, and then compile NetCDF with HDF5 support. When NetCDF is installed, a script
nc-config should be put into somewhere on the path. If this is found then configure should have all the settings it
needs. If this isn’t found then configure will search for the NetCDF include and library files.

### 3.4. File formats
### 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 January 2017, these are versions 4.4.1.1 and 4.3.0 respectively.

Un.tar the file and `cd` into the resulting directory:

```
$ tar -xzvf netcdf-4.4.1.1.tar.gz
$ cd netcdf-4.4.1.1
```

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
```

Similarly for the C++ API:

```
$ tar -xzvf netcdf-cxx4-4.3.0.tar.gz
$ cd netcdf-cxx4-4.3.0
$ ./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 `libiomp`.

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.

### 3.6 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 4.1.0 as of March 2019. It is advisable to use the highest possible version.

In order for a smooth install it is recommended to install SUNDIALS from an install directory. 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:

```
$ cd ~
$ mkdir -p install/sundials-install
$ cd install/sundials-install
$ # Move the downloaded sundials-4.1.0.tar.gz to sundials-install
$ tar -xzvf sundials-4.1.0.tar.gz
$ mkdir build && cd build
$ cmake -DCMAKE_INSTALL_PREFIX=$HOME/local
  -DLAPACK_ENABLE=ON
  -DOPENMP_ENABLE=ON
  -DMPI_ENABLE=ON
  -DCMAKE_C_COMPILER=$(which mpicc)
  -DCMAKE_CXX_COMPILER=$(which mpicxx)
```

(continues on next page)
$./sundials-4.1.0

$ 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.

To configure BOUT++ with SUNDIALS only (see section PETSc on how to build PETSc with SUNDIALS), go to the root directory of BOUT++ and type:

$ ./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.

### 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 3.4 - 3.9. To install PETSc version 3.4.5, use the following steps:

```bash
$ cd ~
$ tar -xzvf petsc-3.4.5.tar.gz
# Optional
# rm petsc-3.4.5.tar.gz
$ cd petsc-3.4.5
```

To build PETSc without SUNDIALS, configure with:

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

Add `--with-debugging=yes` to `./configure` in order to allow debugging.

**Note:** To build PETSc with SUNDIALS, install SUNDIALS as explained in section SUNDIALS, and append `./configure` with `--with-sundials-dir=$HOME/local`

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

```bash
--download-mumps \ 
--download-scalapack \ 
--download-blacs \ 
--download-fblas-lapack=1 \ 
--download-parmetis \ 
--download-ptscotch \ 
--download-metis
```
To make PETSc, type:

```
$ make PETSC_DIR=$HOME/petsc-3.4.5 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-3.4.5 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-3.4.5
$ export PETSC_ARCH=arch-linux2-cxx-debug
```

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

```
export PETSC_DIR=$HOME/petsc-3.4.5
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
$ 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 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.

3.9.1 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/ 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:
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.

### 3.10 Installing FFTW from source

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

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

Download the latest stable version from [http://www.fftw.org/download.html](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:

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

### 3.11 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.

#### 3.11.1 SUNDIALS under AIX

To compile SUNDIALS, use:

```
export CC=cc
export CXX=x1C
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:

```
x1C_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.

```
l: 0711-317 ERROR: Undefined symbol: .NcError::NcError(NcError::Behavior)
l: 0711-317 ERROR: Undefined symbol: .NcFile::is_valid() const
l: 0711-317 ERROR: Undefined symbol: .NcError::~NcError()
l: 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
```
3.11.2 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:

```
-DBOUT_ENABLE_BACKTRACE=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.

3.12 Issues

3.12.1 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.
### 3.12.2 Compiling cvode.cxx fails

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

```cpp
cvoice.cxx: In member function 'virtual int CvodeSolver::init(rhsfunc, bool, int,,
   BoutR...

cvoice.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/cvoice/cvoice.cxx`, and change line 48 from

```cpp
typedef int CVODEINT;
```

to

```cpp
typedef long CVODEINT;
```

### 3.12.3 Compiling with IBM xIC compiler fails

When using the xIC compiler, an error may occur:

```cpp
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:

```cpp
#ifdef MPARK_TYPE_PACK_ELEMENT
```

to:

```cpp
#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](https://software.intel.com/en-us/forums/intel-c-compiler/topic/501502)
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 $T$:

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

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:

```bash
$ 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:

```bash
$ 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`”:

```bash
$ 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”:

```bash
$ ./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 or IDL (skip to **Using IDL** for IDL).

### 4.2 Analysing the output using Python

The recommended tool for analysing BOUT++ output is xBOUT, a Python library that provides analysis, plotting and animation with human-readable syntax (no magic numbers!) using xarray. See the xBOUT documentation [xbout.readthedocs.io](http://xbout.readthedocs.io).

There is also an older set of NumPy-based Python tools, described below.

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-like `BoutArray` array:

```python
>>> 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]`.

`BoutArray` as a thin wrapper for `numpy.ndarray` which adds BOUT++ attributes.

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.

### 4.2.1 Analysing the output using IDL

First, list the variables in one of the data files:

```idl
IDL> print, file_list("BOUT.dmp.0.nc")
iteration MXSUB MYSUB MXG MYG MZ NXPE NYPE BOUT_VERSION t_array ZMAX ZMIN T
```

All of these except 'T' are in all output files, and they contain information about the layout of the mesh so that the data can be put in the correct place. The most useful variable is 't_array' which is a 1D array of simulation output times. To read this, we can use the `collect` function:

```idl
IDL> time = collect(var="t_array")
IDL> print, time
```

The number of variables in an output file depends on the model being solved, which in this case consists of a single scalar field 'T'. To read this into IDL, again use `collect`:

```idl
IDL> T = collect(var="T")
IDL> help, T
T           FLOAT = Array[5, 64, 1, 20]
```

This is a 4D variable, arranged as `[x, y, z, t]`. The `x` direction has 5 points, consisting of 2 points either side for the boundaries and one point in the middle which is evolving. This case is only solving a 1D problem in `y` with 64 points so to display an animation of this

```idl
IDL> showdata, T[2,*0,*]
```

which selects the only evolving `x` point, all `y`, the only `z` point, and all time points. If given 3D variables, `showdata` will display an animated surface

### 4.2. Analysing the output using Python

33
IDL> showdata, T[*,*,0,*]

and to make this a coloured contour plot

IDL> showdata, T[*,*,0,*], /cont

The equivalent commands in Python are as follows.

### 4.3 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
?: 667c19c136fc3e72fcd7c7b2109d44886fdf818d
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: 667c19c136fc3e72fcd7c7b2109d44886fdf818d
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.

### 4.4 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.

4.5 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.

First comes the introductory blurb:

BOUT++ version 1.0
Revision: c8794400adc256480f72c651dcf186fb6ealda49
MD5 checksum: 8419adb752f9c23b90eb50ea2261963c
B.Dudson (University of York), M.Umansky (LLNL) 2007
Based on BOUT by Xueqiao Xu, 1999

The version number (1.0 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. Because code could be modified from the revision, an MD5 checksum of all the code is also calculated. This information makes it possible to verify precisely which version of the code was used for any given run.

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

Compile-time options:
    Checking enabled, level 2
    Signal handling enabled
    netCDF support enabled
    Parallel netCDF support disabled

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

### Setting X differencing methods
- **First**: Second order central (C2)
- **Second**: Second order central (C2)
- **Upwind**: Third order WENO (W3)
- **Flux**: Split into upwind and central (SPLIT)

### Setting Y differencing methods
- **First**: Fourth order central (C4)
- **Second**: Fourth order central (C4)
- **Upwind**: Third order WENO (W3)
- **Flux**: Split into upwind and central (SPLIT)

### Setting Z differencing methods
- **First**: FFT (FFT)
- **Second**: FFT (FFT)
- **Upwind**: Third order WENO (W3)
- **Flux**: Split into upwind and central (SPLIT)

This is a list of the differential methods for each direction. These are set in the BOUT.inp file ([ddx], [ddy] and [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 \( \frac{d}{dt} = v \cdot \nabla f \), and flux terms of the form \( \frac{d}{dt} = \nabla \cdot (v f) \). By default the flux terms are just split into a central and an upwinding term.

In brackets are the code used to specify the method in BOUT.inp. A list of available methods is given in **Differencing methods**:

### Setting grid format
- **Option /grid_format**: (default)
- **Using NetCDF format for file 'slab.6b5.r1.cdl'**

### Loading mesh
- **Grid size**: 10 by 64
- **Option /mxg**: 2 (data/BOUT.inp)
- **Option /myg**: 2 (data/BOUT.inp)
- **Option /NXPE**: 1 (default)
- **Option /mz**: 65 (data/BOUT.inp)
- **Option /twistshift**: false (data/BOUT.inp)
- **Option /TwistOrder**: 0 (default)
- **Option /ShiftOrder**: 0 (default)
- **Option /shiftxderivs**: false (data/BOUT.inp)
- **Option /IncIntShear**: false (default)
- **Option /BoundaryOnCell**: false (default)
- **Option /StaggerGrids**: false (default)
- **Option /periodicX**: false (default)
- **Option /async_send**: false (default)
- **Option /zmin**: 0 (data/BOUT.inp)
- **Option /zmax**: 0.0028505 (data/BOUT.inp)
Optional quantities (such as \texttt{ny\_inner} in this case) which are not specified are given a default (best-guess) value, and a warning is printed:

\begin{verbatim}
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:
\end{verbatim}

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:

\begin{verbatim}
WARNING: Could not read 'g11' from grid. Setting to 1.000000e+00
WARNING: Could not read 'g22' from grid. Setting to 1.000000e+00
WARNING: Could not read 'g33' from grid. Setting to 1.000000e+00
WARNING: Could not read 'g12' from grid. Setting to 0.000000e+00
WARNING: Could not read 'g13' from grid. Setting to 0.000000e+00
WARNING: Could not read 'g23' from grid. Setting to 0.000000e+00
\end{verbatim}

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:

\begin{verbatim}
WARNING: Could not read 'zShift' from grid. Setting to 0.000000e+00
WARNING: Z shift for radial derivatives not found
\end{verbatim}

To get radial derivatives, the quasi-ballooning coordinate method is used. The upshot of this is that to get radial derivatives, interpolation in Z is needed. This should also always be set to FFT:

\begin{verbatim}
WARNING: Twist-shift angle 'ShiftAngle' not found. Setting from zShift
Option /twistshift\_pf = false (default):
Maximum error in diagonal inversion is 0.000000e+00
Maximum error in off-diagonal inversion is 0.000000e+00
\end{verbatim}

If only the contravariant components (\texttt{g11} etc.) of the metric tensor are specified, the covariant components (\texttt{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:

\begin{verbatim}
WARNING: Could not read 'J' from grid. Setting to 0.000000e+00
WARNING: Jacobian 'J' not found. Calculating from metric tensor:
Maximum difference in Bxy is 1.444077e-02
Calculating differential geometry terms
Communicating connection terms
Boundary regions in this processor: core, sol, target, target,
       done:
Setting file formats
   Using NetCDF format for file 'data/BOUT.dmp.0.nc'
\end{verbatim}

The laplacian inversion code is initialised, and prints out the options used:

\subsection*{4.5. Startup output}
Initialising Laplacian inversion routines

Option comms/async = true (default)
Option laplace/filter = 0.2 (default)
Option laplace/low_mem = false (default)
Option laplace/use_pdd = false (default)
Option laplace/all_terms = false (default)
Option laplace/laplace_nonuniform = false (default)
Using serial algorithm
Option laplace/max_mode = 26 (default)

After this comes the physics module-specific output:

Initialising physics module

Option solver/type = (default)
.
.
.

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

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

Initialising solver

Option /archive = -1 (default)
Option /dump_format = nc (data/BOUT.inp)
Option /restart_format = nc (default)
Using NetCDF format for file 'nc':

Initialising PVODE solver

Boundary region inner X
Boundary region outer X
3d fields = 2, 2d fields = 0 neq=84992, local_N=84992

This last line gives the number of equations being evolved (in this case 84992), and the number of these on this processor (here 84992).

Option solver/mudq = 16 (default)
Option solver/mldq = 16 (default)
Option solver/mukeep = 0 (default)
Option solver/mlkeep = 0 (default)

The absolute and relative tolerances come next:

Option solver/atol = 1e-10 (data/BOUT.inp)
Option solver/rtol = 1e-05 (data/BOUT.inp):

Option solver/use_precon = false (default)
Option solver/precon_dimens = 50 (default)
Option solver/precon_tol = 0.0001 (default)
Option solver/mxstep = 500 (default):

Option fft/fft_measure = false (default)

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

Running simulation

(continues on next page)
4.6 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>
<tbody>
<tr>
<td>1.001e+02</td>
<td>76</td>
<td>2.27e+02</td>
<td>87.1</td>
<td>5.3</td>
<td>1.0</td>
<td>0.0</td>
<td>6.6</td>
</tr>
</tbody>
</table>

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

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.

4.7 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 either use archived restart files, or create new restart files. Archived restart files have names like

### 4.6. Per-timestep output
“BOUT.restart_0020.#.nc”, and are written every 20 outputs by default. To change this, set “archive” in the BOUT.inp file. To use these files, they must be renamed to “BOUT.restart.#.nc”. A useful tool to do this is “rename”:

```bash
$ rename 's/_0020//' *.nc
```

will strip out “_0020” from any file names ending in “.nc”.

If you don’t have archived restarts, or want to start from a different time-point, there are Python routines for creating new restart files. If your PYTHONPATH environment variable is set up (see Configuring analysis routines) then you can use the `boutdata.restart.create` function in `tools/pylib/boutdata/restart.py`:

```python
>>> from boutdata.restart import create
>>> create(final=10, path='data', output='.
```

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

### 4.8 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.

#### 4.8.1 Stop file

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

```bash
$ 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:

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

just remember to delete the file afterwards.

#### 4.8.2 Send signal USR1

Another option is to send signal user defined signal 1:

```bash
$ 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:

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

This will stop the simulation cleanly, and:

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

will kill the simulation immediately.

## 4.9 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)
```

### 4.9.1 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

$$\frac{\partial T}{\partial t} = \nabla_{\parallel}(\chi \nabla_{\parallel}T)$$

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:
```

(continues on next page)
Let’s go through it line-by-line. First, we include the header that defines the `PhysicsModel` class:

```
#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):

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

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

```
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`:

```
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:

```
int init(bool restarting) override {
    ...
}
```

(continues on next page)
int rhs(BoutReal time) override {
  ...
}

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 PhysicsModel::rhs as a BoutReal.

The 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 init function), the conduction example first reads an option (lines 21 and 24) from the input settings file (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 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 conduction:chi means the variable “chi” in the section “conduction”. When this option is read, a message is printed to the 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 BOUT++ options, as well as the documentation for the Options class.

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

```
// 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 Initialisation of time evolved variables.

Finally an error code is returned, here 0 indicates no error. If init returns non-zero then the simulation will stop.

5.1. Heat conduction
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 $T$:

$$\frac{dT}{dt} = \nabla_{||}(\chi \partial_{||} T)$$

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++
    return 0;
}
```
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:
Similarly, vector objects can be added/subtracted from each other, multiplied/divided by scalar fields and real numbers, for example:

```c
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 $\ast$ and $\wedge$ symbols:

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

f = a \ast b // Dot-product
a = b \wedge 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:

```c
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 $\wedge$ is lower than $+, \ast$ and $/$ so it is recommended to surround $a \wedge b$ 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`:

```c
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 **BOUT++ options**). Input options based on these names are also used to initialise the variables.
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 \( v \) and \( B \):

\[
SOLVE_FOR(v);  
SOLVE_FOR(B);  
\]

To make this even shorter, multiple fields can be passed to \( SOLVE_FOR \) (up to 10 at the time of writing). We can also use macros \( SOLVE_FOR2 \), \( SOLVE_FOR3 \), \ldots, \( SOLVE_FOR6 \) which are used in many models. Our initialisation code becomes:

```cpp
int init(bool restarting) override {
    ...
    bout_solve(rho, "density");
    bout_solve(p, "pressure");
    v.covariant = true; // evolve covariant components
    B.covariant = false; // evolve contravariant components
    SOLVE_FOR(v, B);
    ...
    return 0;
}
```

Vector quantities can be stored in either covariant or contravariant form. The value of the \( \text{Vector3D::covariant} \) property when \( \text{PhysicsModel::bout_solve()} \) (or \( SOLVE_FOR \)) 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 \( \text{rhs} \) function. The value passed to the function (\( \text{BoutReal t} \)) 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 \( \text{var} \), use \( \text{ddt(var)} \). The ideal MHD equations can be written as:

```cpp
int rhs(BoutReal t) 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) + (Curl(B)^B) / rho;
    ddt(B) = Curl(v^B);
}
```

Where the differential operators \( \text{vector = Grad(scalar), scalar = Div(vector), and vector = Curl(vector)} \) are used. For the density and pressure equations, the \( \mathbf{v} \cdot \nabla \rho \) term could be written as \( \mathbf{v} \cdot \text{Grad}(\rho) \), but this would then use central differencing in the Grad operator. Instead, the function \( \text{V_dot_Grad()} \) uses upwinding methods for these advection terms. In addition, the \( \text{Grad()} \) 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 \( \mathbf{v} \cdot \text{Grad}(\mathbf{v}) \).

### 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 \( \text{BOUT++ options} \)), we use the \( \text{Options} \) 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"];  
(continues on next page)
```
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:

\[
\text{Option mhd:} g = 1.66667 \quad \text{(default)}
\]

This function can be used to get integers and booleans. To get strings, there is the function \( \text{char* options.} \text{getString(section, name)} \). 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:

\[
\text{OPTION(options, } g, \ 5.0 / 3.0);
\]

which is equivalent to:

\[
g = \text{options["g"].withDefault( } 5.0/3.0 \ );
\]

See \textit{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 \texttt{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 \texttt{Mesh::communicate()} function directly. For the MHD code, we need to communicate the variables \( \rho, p, v, B \) at the beginning of the \texttt{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 \texttt{FieldGroup} object, in this case called \texttt{comms}, then use the \texttt{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 \texttt{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 \texttt{FieldGroup}), so this can be shortened to:
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 \( \frac{\partial}{\partial t} \rho \) and \( \frac{\partial}{\partial t} p \) 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 reproduceable 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:

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

// Scope ends, message popped
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 runtime 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:

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

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:

```cpp
{  
    TRACE("ddt(rho)"  
    ddt(rho) = -V_dot_Grad(v, rho) - rho*Div(v);  
}  // Scope ends, message popped
```

### 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:

```cpp
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.

### 5.2.7 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:

```
[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 \( NXPE > 2 \).

```cpp
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++)

(continues on next page)
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:

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--)
        for(int z=0;z<mesh->LocalNz;z++)
            ddt(f)(it.ind,y,z) = 0.; // Set time-derivative to zero in boundary
}

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:

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--)
        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:
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:

GRID_LOAD(Ni0);

which is equivalent to:

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:

Field2D Ni0;
...
GRID_LOAD(Ni0);
dump.add(Ni0, "Ni0", 0);

where the '0' 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.:

SAVE_ONCE(Ni0);

is equivalent to:

dump.add(Ni0, "Ni0", 0);

(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:

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.:

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

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

2. Open the file for writing:
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

```cpp
// 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", 1);
```

Whenever you want to write values to the file, for example in rhs or a monitor, just call:

```cpp
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():

```cpp
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:

```cpp
dump.setAttribute("Ni0", "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 $A_{||}$ and vorticity $U$

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

\[
\frac{\partial A_{||}}{\partial t} = -\frac{1}{\beta} \nabla \nabla \nabla \nabla \nabla \phi - \frac{1}{\beta} j_{||}
\]
with $\phi$ and $j_\parallel$ given by

\[
U = \frac{1}{B} \nabla^2 \phi \\
j_\parallel = -\nabla^2 A_\parallel
\]

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

```cpp
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);
        return 0;
    }
};
```

In order to calculate the time derivatives, we need the auxiliary variables $\phi$ and $j_\parallel$. Calculating $j_\parallel$ from $A_\parallel$ 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 $\text{Delp2}$, like all differential operators, needs the values in the guard cells and so $A_{\parallel}$ needs to be communicated before calculating $j_{\parallel}$. Since we will need to take derivatives of $j_{\parallel}$ 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;
}
```

5.3. Reduced MHD
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:

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

For finer control over which messages are printed, several outputs are available, listed in the table below.

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>output_debug</code></td>
<td>For highly verbose output messages, that are normally not needed. Needs to be enabled with a compile switch</td>
</tr>
<tr>
<td><code>output_info</code></td>
<td>For infos like what options are used</td>
</tr>
<tr>
<td><code>output_progress</code></td>
<td>For infos about the current progress</td>
</tr>
<tr>
<td><code>output_warn</code></td>
<td>For warnings</td>
</tr>
<tr>
<td><code>output_error</code></td>
<td>For errors</td>
</tr>
</tbody>
</table>

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 >= 3. To enable it at lower check levels, configure BOUT++ with `--enable-debug-output` (for `./configure`) or `--DISABLE_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.
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:

```makefile
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
In the makefile in each subdirectory, specify

```
TARGET = sub
```

then specify the path to the top-level directory

```
MODULE_DIR = ..
```

and the name of the subdirectory that the makefile is in

```
SUB_NAME = fuu
```

### 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

```
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:

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

so to get the library linking flags, for example

```
$ 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`:  

```
```
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:

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

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.
Variable initialisation

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 `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:
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-\(\beta\) 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:

\[
\text{[p] function = 1 + gauss(x-0.5)\times gauss(y)\times sin(z)}
\]

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>(\pi) 3.1415...</td>
</tr>
</tbody>
</table>

By default, \(x\) is defined as \(i / (nx - 2\times MXG)\), 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

\[
\text{[mesh] symmetricGlobalX = true}
\]

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 \(\text{paralleltransform = shifted}\), 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 \(\text{paralleltransform = shifted}\)), set in BOUT.inp

\[
\text{[input] transform_from_field_aligned = false}
\]

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>clamp(value, low, high)</td>
<td>If value $&lt;$ low, return low; If value $&gt;$ high, return high; otherwise return value</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 $\frac{1}{2} [\tanh(s(x-[c+\frac{w}{2}])]-\tanh(s(x-[c-\frac{w}{2}]])$</td>
</tr>
<tr>
<td>fmod(x)</td>
<td>The modulo operator, returns floating point remainder</td>
</tr>
</tbody>
</table>

In addition there are some special functions which enable control flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>where(expr, gt0, lt0)</td>
<td>If the first expr evaluates to a value greater than zero then the second expression gt0 is evaluated. Otherwise the last expression lt0 evaluated.</td>
</tr>
<tr>
<td>sum(symbol, count, expr)</td>
<td>Evaluate expression expr count times, and sum the result. Each time the symbol is incremented from 0 to count-1. The value of the symbol is accessed by putting it in braces {}. Example: $\sum(i, 3, {i}^2) = 0^2 + 1^2 + 2^2$.</td>
</tr>
<tr>
<td>[var = value,... ][(expr)</td>
<td>Define a new scope with variables whose value can be accessed using braces {}. The value each variable var is set to can be an expression, and is evaluated before the expr expression. Example: $[n=2] {(n)^2} = 2^2$.</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:

$$U_0^{balloon} = \sum_{i=-N}^{N} F(x)G(y+2\pi i)H(z+q2\pi i)$$  \hspace{1cm} (7.1)
Fig. 7.1: Initial profiles in twist-shifted grid. **Left:** Without ballooning transform, showing discontinuity at the matching location **Right:** with ballooning transform

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:

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

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.

### 7.1.2 Context variables and scope

Expressions can use a form of local variables, by using \([ ]()\) to define new scopes:

```plaintext
var = {a = 2,
       b = 3}( {a} + {b}^{a} )
```

Where here the braces \(\{\}\) refer to context variables, to distinguish them from variables in the options which have no braces. One application of these is a (modest) performance improvement: If \(\{a\}\) is a large expression then in the above example it would only be evaluated once, the value stored as \(\{a\}\) and used twice in the expression.

### 7.1.3 Passing data into expressions

A second application of context variables is that they can be set by the calling C++ code, providing a way for data to be passed from BOUT++ into these expressions. The evaluation of expressions is currently not very efficient, but this provides a very flexible way for the input options to modify simulation behaviour.

This can be done by first parsing an expression and then passing values to `generate` in the `Context` object.

```plaintext
Field3D shear = ...; // Value calculated in BOUT++

FieldFactory factory(mesh);
auto gen = factory->parse("model:viscosity");
```
Field3D viscosity;
viscosity.allocate();

BOUT_FOR(i, viscosity.region("RGN_ALL")) {
    viscosity[i] = gen->generate(bout::generator::Context(i, CELL_CENTRE, mesh, 0.0)
        .set("shear", shear[i]));
}

Note that the Context constructor takes the index, the cell location (e.g. staggered), a mesh, and then the time (set to 0.0 here). Additional variables can be set, "shear" in this case. In the input options file (or command line) the viscosity could now be a function of \{shear\}

[model]
viscosity = 1 + \{shear\}

### 7.1.4 Defining functions in input options

Defining context variables in a new scope can be used to define and call functions, as in the above example viscosity is a function of \{shear\}. For example we could define a cosh function using

\[
\text{mycosh} = 0.5 \times (\exp(\{\text{arg}\}) + \exp(-\{\text{arg}\}))
\]

which uses \{arg\} as the input value. We could then call this function:

\[
\text{result} = [\text{arg} = x\times2] (\text{mycosh})
\]

### 7.1.5 Recursive functions

By default recursive expressions are not allowed in the input options, and a ParseException will be thrown if circular dependencies occur. Recursive functions can however be enabled by setting `input:max_recursion_depth != 0`

e.g.:

[input]
max_recursion_depth = 10 # 0 = none, -1 = unlimited

By putting a limit on the depth, expressions should (eventually) terminate or fail with a BoutException, rather than entering an infinite loop. To remove this restriction `max_recursion_depth` can be set to -1 to allow arbitrary recursion (limited by stack, memory sizes).

If recursion is allowed, then the `where` special function and `Context` scopes can be (ab)used to define quite general functions. For example the Fibonacci sequence 1, 1, 2, 3, 5, 8, ... can be generated:

\[
\text{fib} = \text{where}([n] - 2.5, \[n=[n]-1]\text{fib} + [n=[n]-2]\text{fib}, 1)
\]

so if \(n = 1\) or \(2\) then \(\text{fib} = 1\), but if \(n = 3\) or above then recursion is used.

Note: Use of this facility in general is not encouraged, as it can easily lead to very inefficient and hard to understand code. It is here because occasionally it might be necessary, and because making the input language Turing complete was irresistible.

### 7.1. Initialisation of time evolved variables
### 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

```cpp
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.

```cpp
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:

```cpp
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:

```cpp
class FieldGenerator {
public:
  virtual ~FieldGenerator() { }
  virtual FieldGeneratorPtr clone(const list<FieldGeneratorPtr> &args) { return NULL; } 
  virtual BoutReal generate(const bout::generator::Context& ctx) override = 0;
};
```

where `FieldGeneratorPtr` is an alias for `std::shared_ptr`, a shared pointer to a `FieldGenerator`. The `Context` input to `generate` is an object containing values which can be used in expressions, in particular `x`, `y`, `z` and `t` coordinates. Additional values can be stored in the `Context` object, allowing data from BOUT++ to be used in expressions. There are also ways to manipulate `Context` objects for more complex expressions and functions, see below for details.

All classes inheriting from `FieldGenerator` must implement a `FieldGenerator::generate()` function, which returns the value at the given `(x,y,z,t)` 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:

```cpp
class FieldValue : public FieldGenerator {
public:
  FieldValue(BoutReal val) : value(val) {}
  BoutReal generate(const bout::generator::Context& ctx) override { return value; }
private:
  BoutReal value;
};
```

### 7.3 Adding a new function

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(FieldGeneratorPtr g) : gen(g) {} 
    FieldGeneratorPtr clone(const list<FieldGenerator*> args) override;
    BoutReal generate(const bout::generator::Context& ctx) override;
private:
    FieldGeneratorPtr gen;
};
```

The gen member is used to store the input argument. 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.cxx and add the implementation of the clone and generate functions:

```cpp
FieldGeneratorPtr FieldSinh::clone(const list<FieldGeneratorPtr> args) {
    if (args.size() != 1) {
        throw ParseException("Incorrect number of arguments to sinh function. Expecting 1, got %d", args.size());
    }
    return std::make_shared<FieldSinh>(args.front());
}
BoutReal FieldSinh::generate(const bout::generator::Context& ctx) {
    return sinh(gen->generate(ctx));
}
```

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). Note that std::make_shared is used to make a shared pointer.

The generate function for sinh just gets the value of the input by calling gen->generate(ctx) with the input Context object ctx, 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", std::make_shared<FieldSinh>(nullptr));
```

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

The basic expression parser is defined in `include/bout/sys/expressionparser.hxx` and the code in `src/sys/expressionparser.cxx`. The FieldFactory adds the function in table Table 7.1 on top of this basic functionality, and also uses Options to resolve unknown symbols to Options.

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

```cpp
std::map<std::string, FieldGeneratorPtr> gen;
```

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

```cpp
char ExpressionParser::LexInfo::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 symbol (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`
- -3 for a `Context` parameter which appeared surrounded by braces `{}`.
- 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:

```cpp
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

\[
\text{identifier} := \text{name} \\
\quad := \text{name} '(' \text{expression} [ ',' \text{expression} [ ',' ... ] ] ')'
\]

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<FieldGeneratorPtr> 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.
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 `var` on a boundary region, the options are checked in order from most to least specific:

- Section `var, bndry_ + region name`. Depending on the mesh file, regions of the grid are given labels. Currently these are `core, sol, pf` and `target` which are intended for tokamak edge simulations. Hence the variables checked are `bndry_core, bndry_pf` etc.
- Section `var, bndry_ + boundary side`. These names are `xin, xout, yup` and `ydown`.
- Section `var, variable bndry_all`
- The same settings again except in section `All`.

The default setting for everything is therefore `bndry_all` in the `All` section.

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

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

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

which has the solution

\[ \hat{f} = A e^{x k_z \sqrt{g^{xx}/g^{zz}}} + B e^{-x k_z \sqrt{g^{xx}/g^{zz}}} \]

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 later subsections.

### 8.1 Boundary conditions for non-orthogonal grids

If non-orthogonal grids are used (meaning that the x- and y-directions are not orthogonal, so \( g_{12} \neq 0. \), then corner cells may be required. The boundary conditions are applied in corner cells by applying the y-boundary condition using x-boundary values. This requires that x-boundary conditions are applied before y-boundary conditions. The ordering is taken care of by the methods described in this section, but also needs to be respected by any custom boundary conditions in user code (e.g. sheath boundary conditions). Note that the iterators returned by the BoutMesh methods `iterateBndryLowerY`, `iterateBndryLowerInnerY`, `iterateBndryLowerOuterY`, `iterateBndryUpperY`, `iterateBndryUpperInnerY`, and `iterateBndryUpperOuterY` do include the corner cells at the domain boundary corners.

### 8.2 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:

\[ \left. \frac{\partial f}{\partial t} \right|_i = \left. \frac{\partial f}{\partial t} \right|_{i-1} - f_i / \tau \]

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.
8.3 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:

```
bdry_core = shifted( neumann )
```

would ensure that radial derivatives were zero in shifted coordinates on the core boundary.

8.4 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

```
bdry_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.

```
bdry_sol = width( relax( dirichlet ), 3)
```

would relax the last 3 cells towards zero, whereas

```
bdry_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.

8.5 Examples

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

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

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

[Vi]
bdry_ydown = relax(dirichlet(-1.41648)) # -3.095e4/Vi_x
bdry_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 $\text{bndry\_target = neumann}$ needs to be in the $\text{Ni}$ section: If we just had

<table>
<thead>
<tr>
<th>[All]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{bndry_all} = \text{neumann}$ # Default for all variables, boundaries</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[Ni]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{bndry_core} = \text{relax(dirichlet(1.))}$ # 1e13 cm$^{-3}$ on core boundary</td>
</tr>
<tr>
<td>$\text{bndry_all} = \text{relax(dirichlet(0.1))}$ # 1e12 cm$^{-3}$ on other boundaries</td>
</tr>
</tbody>
</table>

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

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

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

The BoundaryRegion class is defined in include/boundary_region.hxx

### 8.7 Boundary regions

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

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

This returns a vector of pointers to 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 boundary_region.hxx:
/// 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?

  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()) {
    ...
  }

8.8 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);
}
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:

```cpp
/// 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

```cpp
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(bndry->first(); !bndry->isDone(); bndry->next())
        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:

```c++
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 `BoundaryFactory`.

### 8.9 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`:

```c++
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`:

```c++
/// 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;
};
```

### 8.10 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

```c++
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:

```c++
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`:

```c++
// Database of available boundary conditions and modifiers
map<string, BoundaryOp*> opmap;
map<string, BoundaryModifier*> modmap;
```

These are then used by `BoundaryFactory::create()`:

```c++
/// 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`:

```c++
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;
        ss >> 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() {};
protected:
    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:

```bash
$ 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](https://codecov.io), the settings for which are stored in `.codecov.yml`.

## 9.2 Unit tests

The unit test suite 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:
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-ic`, `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`, `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:

```c++
#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.

```c++

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:

```c++
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.

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:

```c++
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.
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.

```plaintext
[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 (( {} []), equality (=), 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.

Numerical quantities can be plain numbers or expressions:

```
short_pi = 3.145
foo = 6 * 9
```

Variables can even reference other variables:

```
pressure = temperature * density
temperature = 12
density = 3
```

Note that variables can be used before their definition; all variables are first read, and then processed afterwards on demand. The value \( \pi \) is already defined, as is \( \pi \), 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 \( \pi \), 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 (borrowed from Julia) 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 \( 2*x + 3*y^2 \), which with the usual precedence rules is the same as \( (2*x) + (3*(y^2)) \).

Expressions can span more than one line, which can make long expressions easier to read:

```
pressure = temperature * ( density0 +
                         density1 )
temperature = 12
density0 = 3
density1 = 1
```

The convention is the same as in Python: If brackets are not balanced (closed) then the expression continues on the next line.

All expressions are calculated in floating point and then converted to an integer if needed 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 round function:

```
bad_integer = 256.4
ok_integer = round(256.4)
```

Note that it is still possible to read 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.
10.1.1 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 \ (backslash) can be used, for example plasma\-density * 10 would read the option plasma\-density and multiply it by 10 e.g.

```
plasma-density = 1e19
2ndvalue = 10
value = plasma\-density * \2ndvalue
```

To escape multiple characters, ‘ (backquote) can be used:

```
plasma-density = 1e19
2ndvalue = 10
value = `plasma-density` * `2ndvalue`
```

The character : 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:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOUT = 100</td>
<td>number of time-points output</td>
</tr>
<tr>
<td>TIMESTEP = 1.0</td>
<td>time between outputs</td>
</tr>
</tbody>
</table>

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

```plaintext
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

```plaintext
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):

```plaintext
[mesh]
x = 16  # Number of points in X
y = 16  # Number of points in Y
z = 32  # Number of points in Z
```

It is recommended, but not necessary, that this be \(nz = 2^n\), i.e. \(1, 2, 4, 8, \ldots\). 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 \(2, 3, 5, 9, \ldots\)) 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

```plaintext
ZPERIOD = 10
```
This specifies a $Z$ range from 0 to $2\pi/Z\text{PERIOD}$, 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

\begin{verbatim}
ZMIN = 0.0
ZMAX = 0.1
\end{verbatim}

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:

\begin{verbatim}
NXPE = 1  # Set number of X processors
\end{verbatim}

Alternatively, the number in the Y direction can be specified (if both are given, NXPE takes precedence and NYPE is ignored):

\begin{verbatim}
NYPE = 1  # Set number of Y processors
\end{verbatim}

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:

\begin{verbatim}
grid = "data/cbm18_8_y064_x260.nc"
# Alternatively:
[mesh]
file = "data/cbm18_8_y064_x260.nc"
\end{verbatim}

### 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 differenciating 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

```plaintext
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>
<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
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):

```cpp
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:

```cpp
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:

```cpp
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):

```cpp
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:

```cpp
BoutReal nout = options["nout"].as<BoutReal>();
```

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"].withDefault<BoutReal>(1);
```

### 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:
options["value"].doc("Useful setting info") = 42;

This string is stored in the attributes of the option:

std::string docstring = options["value"].attributes["doc"];

### 10.8.2 Creating Options

Options and subsections can be created by setting values, creating subsections as needed:

```cpp
Options options;
options["value1"] = 42;
options["subsection1"]["value2"] = "some string";
options["subsection1"]["value3"] = 3.1415;
```

or using an initializer list:

```cpp
Options options {{"value1", 42},
    {"subsection1", {{"value2", "some string"},
      {"value3", 3.1415}}}};
```

These are equivalent, but the initializer list method makes the tree structure clearer. Note that the list can contain many of the types which `Options` can hold, including `Field2D` and `Field3D` objects.

### 10.8.3 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);
```

### 10.8.4 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:

```cpp
options->set("nout", 10, "manual");
```

To create a section, just use `getSection`: if it doesn’t exist it will be created:
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:

```cpp
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:

```cpp
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:
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 option for Fourier transforms, fft_measurement_flag (default: estimate). This can be used to control FFTW’s measurement mode: estimate for FFTW_ESTIMATE, measure for FFTW_MEASURE or exhaustive for FFTW_EXHAUSTIVE:

```plaintext
[fft]
fft_measurement_flag = measure
```

In FFTW_MEASURE mode, FFTW runs and measures how long several FFTs take, and tries to find the optimal method; FFTW_EXHAUSTIVE tests even more algorithms.

**Note:** Technically, FFTW_MEASURE and FFTW_EXHAUSTIVE are 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.

### 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 user-defined, 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:
it is possible not only to test for a value, e.g.:

```cpp
myoption x = <something>;
...
if (x == myoption::foo) {
    do a foo thing
}
```

but also to convert the option to a string:

```cpp
std::string s = toString(x);
```

pass it to a stream:

```cpp
output << x;
```

or get an option like `myinput=baz` from an input file or the command line as a `myoption`:

```cpp
myoption y = Options::root()"myinput".as<myoption>();
```

or with a default value:

```cpp
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.
CHAPTER 11

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.

```plaintext
[mesh]
x = 260  # X grid size
y = 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 x and z dimensions have the same number of points, when x has \texttt{mxg} boundary cells on each boundary but z does not (since it is usually periodic):

```plaintext
[mesh]
x = 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:

```plaintext
[mesh]
x = 256.4  # Error!
x = \texttt{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}:
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 \( ny \) **REQUIRED**
- Differentencing 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 BOUT. \texttt{inp} as \texttt{ShiftXderivs = true}. The shifts must be provided in the gridfile in a field \( zshift[nx][ny] \). If not found, \( zshift \) 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 \( ixseps1 \), and \( ixseps2 \) If neither is given, both are set to \( nx \) (i.e. all points in closed “core” region). If only \( ixseps1 \) is found, \( ixseps2 \) is set to \( nx \), and if only \( ixseps2 \) is found, \( ixseps1 \) is set to -1.
- Branch-cut locations \( jyseps1_1, jyseps1_2, jyseps2_1, \) and \( jyseps2_2 \)
- Twist-shift matching condition \( ShiftAngle[nx] \) for field aligned coordinates. This is applied in the “core” region between indices \( jyseps2_2 \), and \( jyseps1_1 + 1 \), if either \( TwistShift = True \) 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 
ShiftAngle[nx]; ShiftAngle must be given in the gridfile or grid-options if 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, \( f(xspes1<x=xseps2,y,z) \) will have boundary condition in the y-direction set by 
the lowermost ydown and yup. If \( f(xspes2<x,y,z) \) 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
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 $jyseps2_2$
To ensure that communications work branch cuts must align with processor boundaries.

Fig. 11.2: Schematic illustration of domain decomposition and communication in BOUT++ with $ixseps_1 = ixseps_2$
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.

```
UDATA_INDEST  UDATA_OUTDEST

UDATA_XSPLIT

IDATA_DEST → ODATA_DEST

DDATA_XSPLIT

DDATA_INDEST  DDATA_OUTDEST
```

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 <= x < UDATA_XSPLIT going to the processor index UDATA_INDEST, and UDATA_INDEST <= x < 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 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 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 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 tools/tokamak_grids/gridgen, and has its own manual under the doc subdirectory.

11.4 From ELITE and GATO files

Currently conversions exist for ELITE .eqin and GATO 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 tokamak_grids/shifted_circle contains IDL code to generate shifted circle (large aspect ratio) Grad-Shafranov equilibria.

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:
EXTERNAL CODES SUPPLYING EQUILIBRIA

UEDGE

UEDGE OUTPUT
uedgegrd.pdb
uedge.pdb

UEDGE OUTPUT
uedge2bout.pro

PDB G-S FILE
Grad-shafranov

gato2pdb

ELITE EQIN
elite2pdb

DSKGATO

TOQ, CORSICA

shifted_circle

 buret_output.pro

BOUT++ GRID

Fig. 11.4: IDL routines and file formats used in taking output from different codes and converting into input to BOUT++.
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,...,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).

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-Eqdk input file (this is in examples/zoidberg/tokamak.py):

```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
ylength = 10.0
# Define the y locations
ycoords = np.linspace(0.0, ylength, 10, endpoint=False)
# Create the 3D grid by putting together 2D poloidal grids
grid = zoidberg.grid.Grid(poloidal_grid, ycoords, ylength, yperiodic=True)
```
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)
```

(continues on next page)
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:

![Grid produced by grid_elliptic](image)

Fig. 11.5: A grid produced by grid_elliptic from shaped inner and outer lines
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:

```python
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:

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:

```python
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:
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.
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 \( f(\psi) = R \times B_t \) 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 \( (2 \times \text{np.pi}) \) 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:

\[
\begin{align*}
R &= R_0 - b + (a + b \cos(\theta) \cos(\theta + \delta \sin(\theta))) \\
Z &= (1 + \epsilon) a \sin(\theta)
\end{align*}
\]

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 recommended tool for analysing BOUT++ output is xBOUT, a Python library that provides analysis, plotting and animation with human-readable syntax (no magic numbers!) using xarray. See the xBOUT documentation xbout.readthedocs.io.

There is also older analysis and post-processing code, the majority 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]`:

```python
>>> Ni.shape
[10,1,2,3]
```

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.

```python
>>> Ni.attributes("bout_type")
'Field3D_t'
>>> Ni.attributes("location")
'CELL_CENTRE'
```

Attributes can also be read using the `attributes` routine:

```python
from boutdata.collect import attributes
attrs = attributes("Ni")
```

The result is a dictionary (map) of attribute name to attribute value.

If the data has less than 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 neither evolved in time, nor does it have 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"]
options
    |- timestep = 50
    |- myg = 0
    |- nout = 50
    |- mxg = 2
    |- all
        |- bndry_all = neumann
        |- scale = 0.0
        |- phisolver
            |- fourth_order = true
...
```

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
f = DataFile("file.nc")  # Open the file
var = f.read("variable")  # Read a variable from the file
f.close()  # Close the file
```

or similarly for an HDF5 file

```python
from boututils.datafile import DataFile
f = DataFile("file.hdf5")  # Open the file
var = f.read("variable")  # Read a variable from the file
f.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
(continues on next page)
```

12.1. Python routines
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.

```python
>>> f = DataFile("test_io.grd.nc")
>>> print(f.list())
['f3d', 'f2d', 'nx', 'ny', 'rvar', 'ivar']
```

To list the names of the dimensions

```python
>>> print(f.dimensions("f3d"))
('x', 'y', 'z')
```

or to get the sizes of the dimensions

```python
>>> print(f.size("f3d"))
[12, 12, 5]
```

or the dictionary of attributes

```python
>>> print(f.attributes("f3d"))
{}
```

To read in all variables in a file into a dictionary there is the `file_import` function

```python
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`

```python
from boutdata import collect
n = collect("n") # Read data as NumPy array [t,x,y,z]
from boututils.plotdata import plotdata
plotdata(n[-1,:,:,:])
```

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`

```python
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 \texttt{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 \texttt{data/}), you can list the variables available using

```idl
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 \texttt{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:

- **\texttt{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.
- **\texttt{MXG}, \texttt{MYG}**. These are the sizes of the X and Y guard cells
- **\texttt{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 \( \text{MXSUB} + 2 \times \text{MXG} \).
- **\texttt{MYSUB}**, the number of Y grid points per processor (like \texttt{MXSUB})
- **\texttt{MZ}**, the number of Z points
- **\texttt{NXPE}, \texttt{NYPE}**, the number of processors in the X and Y directions. \( \text{NXPE} \times \text{MXSUB} + 2 \times \text{MXG} = \text{NX}, \text{NYPE} \times \text{MYSUB} = \text{NY} \)
- **\texttt{ZMIN}, \texttt{ZMAX}**, the range of Z in fractions of \( 2\pi \).
- **\texttt{iteration}**, the last timestep in the file
- **\texttt{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 \texttt{collect}, and are seldom needed directly. To read a single variable from a file, there is the \texttt{file_read} function:

```idl
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 \texttt{file_import} function:

```idl
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
IDL> ni = collect(var="ni")
Variable 'ni' not found
-> 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
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
IDL> ni = collect(var="Ni", xind=[10,20], yind=[2,2], zind=[0,31],
tind=[300,399])
Reading from .//BOUT.dmp.0.nc: [10-20][4-4] -> [10-20][2-2]
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:

```idl
handle = file_open("filename", /write, /create)
```

Array of variable names:

```idl
list = file_list(handle)
list = file_list("filename")
```

Number of dimensions:

```idl
nd = file_ndims(handle, "variable")
nd = file_ndims("filename", "variable")
```

Read a variable from file. Inds = [xmin, xmax, ymin, ymax, ...]

```idl
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

```idl
status = file_write(handle, "variable", data)
```

Close a file after use

```idl
file_close, handle
```

To read in all the data in a file into a structure:
data = file_import("filename")

and to write a structure to file:

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
IDL> showdata, p[*,32,*,*]
```

To turn this into a contour plot,

```idl
IDL> showdata, p[*,32,*,*], /cont
```

To show a slice through this at a particular toroidal location (0 here):

```idl
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
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,nts);
```

Here, `path` and `var_name` are same variables as described before. `nt` is the number of time steps user wish to load data. `nts` 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 `nts` i.e. 100 here. Details of other Matlab routines provided with BOUT++ package can be looked in to README.txt of tools/matlablib directory. The Matlab users can develop their own routines using *ncread, ncinfo, ncwrite, 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.

```octave
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
```
13.
The python boutcore module

13.1 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.llinux-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 fft 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\}
```
13.2 Purpose

The boutcore module exposes (part) of the BOUT++ C++ library to python. It allows to calculate e.g. BOUT++ derivatives in python.

13.3 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.

13.4 Functions

13.5 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
upar=boutcore.Field3D.fromCollect("Upar")
upar=boutcore.interp_to(upar,"CELL CENTRE")
```

(continues on next page)
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"

# convert to numpy array
upar = upar.getAll()

times = collect("t_array", path=path)

boutcore.init("-d data -f BOUT.settings -o BOUT.post")
with Dataset(path + '/vort.nc', 'w', 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()
```

13.5. Examples
13.6 Functions - undocumented

13.7 Functions - special and inherited
14.1 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 14.1 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>
</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 14.2.

<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</td>
</tr>
<tr>
<td>rtol</td>
<td>Relative tolerance</td>
<td>rk4, pvode, cvode, ida, imexbdf2</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, karniadakis, euler, imexbdf2</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>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</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.

14.2 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.
14.3 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 `imexbdf2`, e.g. on the command-line add `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 `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 `atol` and `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 `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.

14.3. IMEX-BDF2 135
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 timestep</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 timestep</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>nadapt</td>
<td>4</td>
<td>How often is error checked and timestep adjusted?</td>
</tr>
<tr>
<td>adaptRtol</td>
<td>1e-3</td>
<td>Target relative tolerance for adaptive timestep</td>
</tr>
<tr>
<td>scale-Cush-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-CushUp</td>
<td>1.5</td>
<td>Minimum timestep scale factor based on adaptRtol above which the timestep will be modified. Currently the timestep increase is limited to 25%</td>
</tr>
</tbody>
</table>

### 14.4 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.

2. 3rd order SSP-RK3 scheme for the advection (hyperbolic) part

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:

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timestep</td>
<td>output timestep</td>
<td>If adaptive sets the starting timestep. If not adaptive, timestep fixed at this value</td>
</tr>
<tr>
<td>nstages</td>
<td>10</td>
<td>Number of stages in RKL step. Must be &gt; 1</td>
</tr>
<tr>
<td>diagnose</td>
<td>false</td>
<td>Print diagnostic information</td>
</tr>
</tbody>
</table>
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>

### 14.5 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`.

```cpp
class MyFunction : public PhysicsModel {
  public:
    int init(bool restarting) {
      // Initialise ODE
      // Add variables to solver as usual
      solver->add(result, "result");
      ...
    }

    int rhs(BoutReal time) {
      // Specify derivatives of fields as usual
      ddt(result) = ...
    }

  private:
    Field3D result;
};
```

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:
MyFunction* model = new MyFunction();
ode->setModel(model);

Finally tell the solver to perform the integration:

doode->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

doode->solve();

will cause ode to look in the input for nout and timestep options:

[ode]
  nout = 5
  timestep = 0.1

Finally, delete the model and solver when finished:

deleter model;
deleter 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.

14.6 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 $\mathcal{J}$ 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:

\[
\frac{\partial u}{\partial t} = \partial_{||} v \quad \frac{\partial v}{\partial t} = \partial_{||} u
\]

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

\[
\mathcal{J} = \begin{pmatrix}
\frac{\partial}{\partial u} \frac{\partial u}{\partial t} & \frac{\partial}{\partial v} \frac{\partial u}{\partial t} \\
\frac{\partial}{\partial u} \frac{\partial v}{\partial t} & \frac{\partial}{\partial v} \frac{\partial v}{\partial t}
\end{pmatrix} = \begin{pmatrix}
0 & \partial_{||} u \\
\partial_{||} v & 0
\end{pmatrix}
\]

In this case $\frac{\partial}{\partial u}$ doesn’t depend on $u$ nor $\frac{\partial}{\partial v}$ on $v$, so the diagonal is empty. Since the equations are linear, the Jacobian doesn’t depend on $u$ or $v$ and so

\[
\frac{\partial}{\partial t} \begin{pmatrix}
u \\
v
\end{pmatrix} = \mathcal{J} \begin{pmatrix}
u \\
v
\end{pmatrix}
\]

\(^1\) Taken from a talk by L.Chacon available here https://bout2011.llnl.gov/pdf/talks/Chacon_bout2011.pdf
In general for non-linear functions $\mathcal{J}$ 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

$$\mathcal{I} - \gamma \mathcal{J}$$

where $\mathcal{I}$ 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

$$\mathcal{I} - \gamma \mathcal{J} = \begin{pmatrix} 1 & -\gamma \partial_{||} \\ -\gamma \partial_{||} & 1 \end{pmatrix}$$

This matrix can be block inverted using Schur factorisation:\footnote{See paper https://arxiv.org/abs/1209.2054 for an application to 2-fluid equations}

$$\begin{pmatrix} E & U \\ L & D \end{pmatrix}^{-1} = \begin{pmatrix} I & -E^{-1}U \\ 0 & I \end{pmatrix} \begin{pmatrix} E^{-1} & 0 \\ 0 & P_{\text{Schur}}^{-1} \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & -L E^{-1} I \end{pmatrix}$$

where $P_{\text{Schur}} = D - L E^{-1} U$ Using this, the wave problem becomes:

$$\begin{pmatrix} 1 & -\gamma \partial_{||} \\ -\gamma \partial_{||} & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & \gamma \partial_{||} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & (1 - \gamma^2 \partial_{||}^2)^{-1} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \gamma \partial_{||} & 1 \end{pmatrix}$$

(14.1)

The preconditioner is implemented by defining a function of the form

```c
int precon(BoutReal t, BoutReal gamma, BoutReal delta) {
  ...
}
```

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:

```c
int precon(BoutReal t, BoutReal gamma, BoutReal delta) {
}
```

To implement the preconditioner in equation (14.1), first apply the rightmost matrix to the given vector:

$$\begin{pmatrix} \text{ddt}(u) \\ \text{ddt}(v) \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \gamma \partial_{||} & 1 \end{pmatrix} \begin{pmatrix} \text{ddt}(u) \\ \text{ddt}(v) \end{pmatrix}$$

```c
int precon(BoutReal t, BoutReal gamma, BoutReal delta) {
  mesh->communicate(ddt(u));
  //ddt(u) = ddt(u);
  ddt(v) = gamma*Grad_par(ddt(u)) + 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

$$\begin{pmatrix} \text{ddt}(u) \\ \text{ddt}(v) \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 \\ 0 & (1 - \gamma^2 \partial_{||}^2)^{-1} \end{pmatrix} \begin{pmatrix} \text{ddt}(u) \\ \text{ddt}(v) \end{pmatrix}$$

doesn’t alter $u$, but solves a parabolic equation in the parallel direction. There is a solver class to do this called $\text{InvertPar}$ which solves the equation $(A + B \partial_{||}^2)x = b$ where $A$ and $B$ are $\text{Field2D}$ or constants:\footnote{This $\text{InvertPar}$ class can handle cases with closed field-lines and twist-shift boundary conditions for tokamak simulations}

In $\text{PhysicsModel::init()}$ we create one of these solvers:

14.6. Preconditioning 139
InvertPar *inv; // Parallel inversion class

int init(bool restarting) {  
    ...  
    inv = InvertPar::Create();  
    inv->setCoefA(1.0);  
    ...  
}

In the preconditioner we then use this solver to update $v$:

```cpp
inv->setCoefB(-SQ(gamma));
ddt(v) = inv->solve(ddt(v));
```

which solves $ddt(v) \rightarrow (1 - \gamma^2 \partial||_u^2)^{-1} ddt(v)$. The final matrix just updates $u$ using this new solution for $v$

$$
\begin{pmatrix}
    ddt(u) \\
    ddt(v)
\end{pmatrix}
\rightarrow
\begin{pmatrix}
    1 & \gamma \partial\|
    \\
    0 & 1
\end{pmatrix}
\begin{pmatrix}
    ddt(u) \\
    ddt(v)
\end{pmatrix}
$$

mesh->communicate(ddt(v));
ddt(u) = ddt(u) + gamma*Grad_par(ddt(v));

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

```cpp
[solver]  
  type = cvode # Need CVODE or PETSc  
  use_precon = true # Use preconditioner  
  rightprec = false # Use Right preconditioner (default left)
```

### 14.7 Jacobian function

### 14.8 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-constrain) we have:

\[ \text{ddt}(\phi) = \text{Delp2}(\phi) - \text{Vort}; \]

so the time integration solver includes the algebraic constraint \( \text{Delp2}(\phi) = \text{Vort} \) i.e. \( \nabla^2_{\perp} \phi = \omega \).

### 14.9 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.
14.10 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:

```cpp
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 {
    public:
    MyOutputMonitor(BoutReal timestep=-1):Monitor(timestep){};
    int call(Solver *solver, BoutReal simtime, int iter, int NOUT) override {
        ...
    }
};
```

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);
        }
};
```

(continues on next page)
MyOutputMonitor my_monitor;

int init(bool restarting) {
    solver->addMonitor(my_monitor);
}

See the monitor example (examples/monitor) for full code.

**Timestep monitoring:** This uses functions instead of objects. First define a monitor function:

```c
int my_timestep_monitor(Solver *solver, BoutReal simtime, BoutReal lastdt) {
    ...
}
```

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.

## 14.11 Implementation internals

The solver is the interface between BOUT++ and the time-integration code 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:

14.11. Implementation internals
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 $\mathbf{B}$ can be expressed as

$$\mathbf{B} = \nabla z \times \nabla x$$

then the Clebsch operators can be used. See section *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 *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 *ParallelTransform* classes. Each *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 *Iterating over fields*):

```cpp
Field3D result;
result.allocate(); // Need to allocate before indexing

for (const auto &i : result.region(RGN_NOBNDRY)) {
```

(continues on next page)
result[i] = f.yup()[i.yp()] - f.ydown()[i.ym()];

Note the use of yp() and ym() to increase and decrease the Y index.

### 15.1 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

```plaintext
[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).

### 15.2 Shifted metric

The shifted metric method is selected using:

```plaintext
[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.

### 15.3 FCI method

To use the FCI method for parallel transforms, set

```plaintext
[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.

15.3. FCI method
A common problem in plasma models is to solve an equation of the form

\[ d\nabla^2 \perp x + \frac{1}{c_1} (\nabla \perp c_2) \cdot \nabla \perp x + ax = b \]  \hspace{1cm} (16.1)

For example,

\[ \nabla^2 \perp x + ax = b \]

appears in reduced MHD for the vorticity inversion and \( j_\parallel \).

Alternative formulations and ways to invert equation (16.1) can be found in sections 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 16.1.

<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>petsc3dmg</td>
<td>Serial/parallel. Solves full 3D operator (with y-derivatives) with algebraic multigrid.</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</td>
<td></td>
</tr>
<tr>
<td>serial_tri</td>
<td>Serial only. Thomas algorithm for tridiagonal system.</td>
<td>Lapack (section LAPACK)</td>
</tr>
<tr>
<td>serial_band</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>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>
16.1 Usage of the laplacian inversion

In BOUT++, equation (16.1) 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 (16.1) plane by 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 (16.1) 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);
```

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 16.2 to Table 16.5.

In particular boundary conditions on the \( x \) boundaries can be set using the and outer_boundary_flags variables, as detailed in table Table 16.4. 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.
lap->setOuterBoundaryFlags(Outer_Flags_Value);
lap->setFlags(Flags_Value);

Table 16.2: 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 16.1</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 \textit{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 \textit{Laplace global flags}</td>
<td>0</td>
</tr>
<tr>
<td>inner_boundary</td>
<td>Sets boundary conditions on inner boundary. See table \textit{Laplace boundary flags}</td>
<td>0</td>
</tr>
<tr>
<td>outer_boundary</td>
<td>Sets boundary conditions on outer boundary. See table \textit{Laplace boundary flags}</td>
<td>0</td>
</tr>
<tr>
<td>flags</td>
<td>DEPRECATED. Sets global solver options and boundary conditions. See \textit{Laplace flags} or \textit{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 16.3: Laplacian inversion \textit{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>

16.1. Usage of the laplacian inversion
Table 16.4: 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 ( \frac{\partial^2}{\partial x^2} + k^2 ) 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 ( \text{Laplacian}::\text{solve}(\text{const Field3D} &amp;b, \text{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 ( \text{Laplacian}::\text{solve}(\text{const Field3D} &amp;b, \text{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 ( \frac{\partial^2}{\partial x^2} ) 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 16.5: 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 MXG)</td>
</tr>
<tr>
<td>128</td>
<td>Use 4(^{\text{th}})-order band solver (default is 2(^{\text{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 \texttt{solve} method

\[
x = \text{lap}->\text{solve}(b);
\]

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 = invert_laplace(b, 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.

16.2 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 \( f \)

\[
d \nabla^2 f + \frac{1}{c_1} (\nabla \cdot c_2) \cdot \nabla f + af = b
\]  
(16.2)

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 (16.2) becomes (see Field-aligned coordinates for derivation)

\[
d (g^{xz} \partial_x^2 + G^x \partial_x + g^{zz} \partial_z^2 + G^z \partial_z + 2g^{xz} \partial_x \partial_z) f
\]

\[
+ \frac{1}{c_1} (e^x \partial_x + e^z \partial_z) c_2 \cdot (e^x \partial_x + e^z \partial_z) f
\]

\[
+ af = b
\]  
(16.3)

16.2.1 Using tridiagonal solvers

Since there are no parallel \( y \)-derivatives if \( g_{xy} = g_{yz} = 0 \) (or if they are neglected), equation (16.2) 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 (16.3), we can use a tridiagonal solver to solve the equation for each Fourier mode.

Using the discrete Fourier transform

\[
F(x, y)_k = \frac{1}{N} \sum_{Z=0}^{N-1} f(x, y)_Z \exp(-\frac{2\pi i k Z}{N})
\]

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

\[
\frac{1}{N} \sum_{Z=0}^{N-1} a(x, y)_Z f(x, y)_Z \exp(-\frac{2\pi i k Z}{N})
\]

Thus, in order to use a tridiagonal solver, \( a, c_1, c_2 \) and \( d \) cannot be functions of \( z \). Because of this, the \( e^x \partial_x c_2 \) term in equation (16.3) is zero. Thus the tridiagonal solvers solve equations of the form

\[
d(x, y) (g^{xx}(x, y) \partial_x^2 + G^x(x, y) \partial_x + g^{xz}(x, y) \partial_z^2 + G^z(x, y) \partial_z + 2g^{xz}(x, y) \partial_x \partial_z) f(x, y, z)
\]

\[
+ \frac{1}{c_1(x, y)} (e^x \partial_x c_2(x, y)) \cdot (e^x \partial_x + e^z \partial_z) f(x, y, z)
\]

\[
+ af(x, y, z) = b(x, y, z)
\]
after using the discrete Fourier transform (see section Derivatives of the Fourier transform), we get
\[
d(g^{xz} \partial_z^2 F_z + G^x \partial_z F_z + g^{zz} [ik]^2 F_z + G^z [ik] F_z + 2g^{xz} \partial_x [ik] F_z) \\
+ \frac{1}{c_1} (e^x \partial_x c_2) \cdot (e^x \partial_x F_z + e^z ik F_z) \\
+ a F_z = B_z
\]
which gives
\[
d(g^{xz} \partial_z^2 F_z + G^x \partial_z F_z - k^2 g^{zz} + ikG^z + ik2g^{xz} \partial_x) F_z \\
+ \frac{1}{c_1} (\partial_x c_2)(g^{xz} \partial_z F_z + g^{zz} ik F_z) \\
+ a F_z = B_z \tag{16.4}
\]
As nothing in equation (16.4) 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 (16.4) \(k\) 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
\[
\partial_x f \approx \frac{-f_{n-1} + f_{n+1}}{2 dx} \\
\partial_x^2 f \approx \frac{f_{n-1} - f_n + f_{n+1}}{dx^2}
\]
This gives
\[
d \left( g^{xx} \frac{F_{z,n-1} - 2F_{z,n} + F_{z,n+1}}{dx^2} + G^x \frac{F_{z,n-1} + F_{z,n+1}}{2dx} - k^2 g^{zz} F_{z,n} \\
+ ikG^z F_{z,n} + ik2g^{xz} \frac{F_{z,n-1} + F_{z,n+1}}{2dx} \right) \\
+ \frac{1}{c_1} \left( \frac{-c_{2,n-1} + c_{2,n+1}}{2dx} \right) \left( g^{xz} \frac{F_{z,n-1} + F_{z,n+1}}{2dx} + g^{zz} ik F_{z,n} \right) \\
+ a F_{z,n} = B_{z,n}
\]
collecting point by point
\[
\left( \frac{dg^{xx}}{dx^2} - \frac{dG^x}{c_{1,n}} \frac{g^{xx} - c_{2,n-1} + c_{2,n+1}}{4dx^2} - i \frac{dk2g^{xz}}{2dx} \right) F_{z,n-1} \\
+ \left( \frac{-dg^{xx}}{dx^2} - \frac{dk2g^{zz} + a + ikG^z + i \frac{g^{xz} - c_{2,n-1} + c_{2,n+1} k}{c_{1,n}}}{4dx^2} \right) F_{z,n} \\
+ \left( \frac{dg^{xx}}{dx^2} + \frac{dG^x}{c_{1,n}} \frac{g^{xx} - c_{2,n-1} + c_{2,n+1}}{4dx^2} + i \frac{dk2g^{xz}}{2dx} \right) F_{z,n+1} \tag{16.5}
\]
We now introduce
\[
C_1 = \frac{dg^{xx}}{dx^2} \\
C_2 = dg^{zz} \\
C_3 = \frac{2dg^{xz}}{2dx} \\
C_4 = \frac{dG^x + g^{xx} \frac{-c_{2,n-1} + c_{2,n+1}}{2c_{1,n} dx}}{2dx} \\
C_5 = \frac{dG^z + g^{xz} \frac{-c_{2,n-1} + c_{2,n+1}}{2dx}}{c_{1,n}}
\]
which inserted in equation (16.5) 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 (16.2), but from version 4.3 allow \(c_1 \neq c_2\).

### 16.2.2 Using PETSc solvers

When using PETSc, all terms of equation (16.3) 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 (16.3) 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 \(=(0,0) = 0\) to top right \(=(\text{meshx} - 1, \text{meshz} - 1) = \text{meshx} \cdot \text{meshz} - 1\) of the two dimensional field. This is done in such a way so that a row \(i\) in \(A\) increments by 1 for an increase of 1 in the \(z\)-direction, and by \(\text{meshz}\) for an increase of 1 in the \(x\)-direction, where the variables \(\text{meshx}\) and \(\text{meshz}\) represents the highest value of the field in the given direction.

Similarly to equation (16.5), the discretised version of equation (16.3) can be written. Doing the same for the full two dimensional case yields:

#### Second order approximation
\[
c_{i,j} f_{i,j} + c_{i-1,j-1} f_{i-1,j-1} + c_{i-1,j} f_{i-1,j} + c_{i-1,j+1} f_{i-1,j+1} + c_{i,j} f_{i,j-1} + c_{i,j+1} f_{i,j+1} + c_{i+1,j} f_{i+1,j} = b_{i,j}
\]

#### Fourth order approximation
\[
c_{i,j} f_{i,j} + c_{i-2,j-2} f_{i-2,j-2} + c_{i-2,j-1} f_{i-2,j-1} + c_{i-2,j} f_{i-2,j} + c_{i-2,j+1} f_{i-2,j+1} + c_{i-1,j-2} f_{i-1,j-2} + c_{i-1,j-1} f_{i-1,j-1} + c_{i-1,j} f_{i-1,j} + c_{i-1,j+1} f_{i-1,j+1} + c_{i,j-2} f_{i,j-2} + c_{i,j-1} f_{i,j-1} + c_{i,j} f_{i,j} + c_{i,j+1} f_{i,j+1} + c_{i,j+2} f_{i,j+2} + c_{i+1,j-2} f_{i+1,j-2} + c_{i+1,j-1} f_{i+1,j-1} + c_{i+1,j} f_{i+1,j} + c_{i+1,j+1} f_{i+1,j+1} + c_{i+2,j-2} f_{i+2,j-2} + c_{i+2,j-1} f_{i+2,j-1} + c_{i+2,j} f_{i+2,j} + c_{i+2,j+1} f_{i+2,j+1} + c_{i+2,j+2} f_{i+2,j+2} = b_{i,j}
\]
To determine the coefficient for each node point, it is convenient to introduce some quantities

\[ A_0 = a(x, y_{\text{current}}, z) \]
\[ A_1 = dg^{xz} \]
\[ A_2 = dg^{zz} \]
\[ A_3 = 2dg^{xz} \]  \hspace{1cm} (16.6)
\[ A_4 = dG^{x} + g^{xz}ddx_c + g^{zz}ddz_c \]
\[ A_5 = dG^{z} + g^{xz}ddx_c + g^{zz}ddz_c \]  \hspace{1cm} (16.7)

In addition, we have:

Second order approximation (5-point stencil)

\[
\begin{align*}
ddx_c &= \frac{c_{2x+1} - c_{2x-1}}{2c_1dx} \\
ddz_c &= \frac{c_{2z+1} - c_{2z-1}}{2c_1dz}
\end{align*}
\]

Fourth order approximation (9-point stencil)

\[
\begin{align*}
ddx_c &= \frac{-c_{2x+2} + 8c_{2x+1} - 8c_{2x-1} + c_{2x-1}}{12c_1dx} \\
ddz_c &= \frac{-c_{2z+2} + 8c_{2z+1} - 8c_{2z-1} + c_{2z-1}}{12c_1dz}
\end{align*}
\]

This gives

\[ A_4 = dG^{x} + g^{xz}ddx_c + g^{zz}ddz_c \]
\[ A_5 = dG^{z} + g^{xz}ddx_c + g^{zz}ddz_c \]

The coefficients \( c_{i+j+n} \) are finally being set according to the appropriate order of discretisation. The coefficients can be found in the file `petsc_laplace.cxx`.

### 16.2.3 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. 16.1).

![Fig. 16.1: 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 \( f_{2z} \) this mesh will result in Fig. 16.2.

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. 16.3.

As we are using a row-major implementation, the global indices of the matrix will be as in Fig. 16.4.
Fig. 16.2: 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.

<table>
<thead>
<tr>
<th>$f_{03}$</th>
<th>$f_{13}$</th>
<th>$zp_{23}$</th>
<th>$f_{23}$</th>
<th>$f_{33}$</th>
<th>$f_{43}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_{02}$</td>
<td>$f_{12}$</td>
<td>$f_{22}$</td>
<td>$f_{32}$</td>
<td>$zp_{32}$</td>
<td>$f_{42}$</td>
</tr>
<tr>
<td>$f_{01}$</td>
<td>$f_{11}$</td>
<td>$f_{21}$</td>
<td>$f_{31}$</td>
<td>$zp_{31}$</td>
<td>$f_{41}$</td>
</tr>
</tbody>
</table>

$z$ $x$

Fig. 16.3: Matrix problem for our $3 \times 3$ mesh: The colors follow that of figure Fig. 16.1 and Fig. 16.2. The first index of the elements refers to the $x$-position in figure Fig. 16.1, and the last index of the elements refers to the $z$-position in figure Fig. 16.1. $ig$ 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.

$$
\begin{bmatrix}
1 & 0 & 0 & i_{g11} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & i_{g21} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & i_{g31} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & i_{g41} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & i_{g51} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & i_{g61} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & i_{g71} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & i_{g81} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & i_{g91} & 0 & 0 & 0 & 0 \\
\end{bmatrix} \cdot \begin{bmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5 \\
x_6 \\
x_7 \\
x_8 \\
x_9 \\
\end{bmatrix} = \begin{bmatrix}
x_0 \\
x_0 \\
x_0 \\
x_0 \\
x_0 \\
x_0 \\
x_0 \\
x_0 \\
x_0 \\
\end{bmatrix}
$$

Fig. 16.4: Global indices of the matrix in figure Fig. 16.3
16.3 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 \( 4^{th} \)-order differencing).

To support multiple implementations, a base class \textit{Laplacian} is defined in \textit{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 \textit{src/invert/laplace/invert\_laplace.cxx}. The actual creation of new Laplacian implementations is done in the \textit{LaplaceFactory} class, defined in \textit{src/invert/laplace/laplacefactory.cxx}. This file includes all the headers for the implementations, and chooses which one to create based on the \textit{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 \textit{src/invert/laplace/impls} and is discussed below.

16.3.1 Serial tridiagonal solver

This is the simplest implementation, and is in \textit{src/invert/laplace/impls/serial\_tri/}
### 16.3.2 Serial band solver

This is band-solver which performs a 4th-order inversion. Currently this is only available when \(NXPE=1\); when more than one processor is used in \(x\), the Laplacian algorithm currently reverts to 3rd-order.

### 16.3.3 SPT parallel tridiagonal

This is a reference code which performs the same operations as the serial code. To invert a single XZ slice (\(FieldPerp\) object), data must pass from the innermost processor (\(mesh->PE_XIND = 0\)) to the outermost \(mesh->PE_XIND = mesh->NXPE-1\) 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 (\(MYSUB > mesh->NXPE\)). If \(MYSUB < mesh->NXPE\) then not all processors can be busy at once, and so efficiency will fall sharply. Fig. 16.5 shows the usage of 4 processors inverting a set of 3 poloidal slices (i.e. \(MYSUB=3\))

<table>
<thead>
<tr>
<th>Processor</th>
<th># 1</th>
<th># 2</th>
<th># 3</th>
<th># 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Forward 2</td>
<td>Forward 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Forward 3</td>
<td>Forward 2</td>
<td>Forward 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Forward 3</td>
<td>Forward 3</td>
<td>Forward 2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Forward 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Back 1</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Back 1</td>
<td>Back 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Back 1</td>
<td>Back 2</td>
<td>Back 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Back 2</td>
<td>Back 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Back 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 16.5: 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
16.3.4 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.

16.3.5 Cyclic algorithm

This is now the default solver in both serial and parallel. It is an FFT-based solver using a cyclic reduction algorithm.

16.3.6 Multigrid solver

A solver using a geometric multigrid algorithm was introduced by projects in 2015 and 2016 of CCFE and the EURO-fusion HLST.

16.3.7 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 < 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 LaplaceNaulin solver, counting in the order they are created in the physics model:

- naulin_solver<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.
- naulin_solver<i>_mean_its is the mean number of iterations taken to converge. Try to minimise when adjusting initial_underrelax_factor.

16.4 LaplaceXY

Perpendicular Laplacian solver in X-Y.

\[ \nabla_{\perp} f = \nabla f - \mathbf{b} \cdot \nabla \]

\[ = \left( \frac{\partial f}{\partial x} - \frac{g_{xy}}{g_{yy}} \frac{\partial f}{\partial y} \right) \nabla x + \left( \frac{\partial f}{\partial z} - \frac{g_{yz}}{g_{yy}} \frac{\partial f}{\partial y} \right) \nabla z \]

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

\[ \nabla_{\perp} f = \frac{\partial f}{\partial x} \nabla x - \frac{g_{yz}}{g_{yy}} \frac{\partial f}{\partial y} \nabla z \]
The divergence operator in conservative form is
\[ \nabla \cdot A = \frac{1}{J} \frac{\partial}{\partial u^i} (J g^{ij} A_j) \]
and so the perpendicular Laplacian in X-Y is
\[ \nabla^2 \perp f = \frac{1}{J} \frac{\partial}{\partial x} \left( J g^{xx} \frac{\partial f}{\partial x} \right) - \frac{1}{J} \frac{\partial}{\partial y} \left( J g^{yz} g_{yy} \frac{\partial f}{\partial y} \right) \]
In field-aligned coordinates, the metrics in the \( y \) derivative term become:
\[ g_{yz} g_{yz} g_{yy} = B^2 \frac{1}{h^2_{\varphi}} \]
In the LaplaceXY operator this is implemented in terms of fluxes at cell faces.
\[ 1 \frac{1}{J} \frac{\partial}{\partial x} \left( J g^{xx} \frac{\partial f}{\partial x} \right) \rightarrow \frac{1}{J_i \Delta x_i} \left[ J_{i+1/2} g_{i+1/2}^{xx} \left( f_{i+1} - f_i \right) \Delta x_{i+1/2} - J_{i-1/2} g_{i-1/2}^{xx} \left( f_i - f_{i-1} \right) \Delta x_{i-1/2} \right] \]
Notes:
- The ShiftedMetric or FCITransform ParallelTransform must be used (i.e. `mesh:paralleltransform:type = shifted` or `mesh:paralleltransform:type = fci`) for this to work, since it assumes that \( g^{xz} = 0 \)
- 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 \( A \cdot \text{Laplace}_\perp(f) + \text{Grad}_\perp(A) \cdot \text{Grad}_\perp(f) + B \cdot f \) is used. This also supports non-orthogonal grids with \( g^{xy} \neq 0 \). The difference is that when \( g^{xy} \neq 0 \), \( \text{Laplace}_\perp \) calls \( \text{D2DXDY}(f) \) which applies a boundary condition to \( \text{dfdy} = \text{DDY}(\text{dfdx}) \) before calculating \( \text{DDX}(\text{dfdy}) \) 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.)

### 16.5 LaplaceXZ

This is a Laplacian 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 ((16.8)), but with negligible \( y \)-parallel derivatives if
\( g_{xy}, \ g_{yz} \ \text{and} \ g_{xz} \) is non-vanishing. The Laplacian is written in conservative form like the \textit{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 \texttt{include/bout/invert/laplacexz.hxx}. The solver is constructed by using the \texttt{LaplaceXZ::create()} function:

```cpp
LaplaceXZ *lap = LaplaceXZ::create(mesh);
```

Note that a pointer to a \texttt{Mesh} object must be given, which for now is the global variable \texttt{mesh}. By default the options section \texttt{laplacexz} is used, so to set the type of solver created, set in the options:

```yaml
[laplacexz]
type = petsc  # Set LaplaceXZ type
```

or on the command-line \texttt{laplacexz:type=petsc}.

The coefficients must be set using \texttt{setCoefs}. All coefficients must be set at the same time:

```cpp
lap->setCoefs(1.0, 0.0);
```

Constants, \texttt{Field2D} or \texttt{Field3D} values can be passed. If the implementation doesn’t support \texttt{Field3D} values then the average over \( z \) will be used as a \texttt{Field2D} value.

To perform the inversion, call the \texttt{solve} function:

```cpp
Field3D vort = ...;
Field3D phi = lap->solve(vort, 0.0);
```

The second input to \texttt{solve} is an initial guess for the solution, which can be used by iterative schemes e.g. using PETSc.

### 16.5.1 Implementations

The currently available implementations are:

- \texttt{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 \texttt{CyclicReduction} solver). Code in \texttt{src/invert/laplacexz/impls/cyclic/}.

- \texttt{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.

### 16.5.2 Test case

The code in \texttt{examples/test-laplacexz} is a simple test case for \texttt{LaplaceXZ}. First it creates a \texttt{LaplaceXZ} object:

```cpp
LaplaceXZ *inv = LaplaceXZ::create(mesh);
```

For this test the \texttt{petsc} implementation is the default:
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 `-mat_view ::ascii_info` information on the assembled matrix is printed:

```bash
$ 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:

```cpp
Field3D rhs = FieldFactory::get()->create3D("rhs", Options::getRoot(), mesh);
```

which is currently set to a simple function in the options:

```cpp
rhs = sin(x - z)
```

and then the system is solved:

```cpp
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
```

(continues on next page)
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:

```c++
inv->setCoefs(Field3D(2.0),Field3D(0.1));
```

and solved again:

```c++
SOLVE time 0.00
Solve flops 8.245800e+04 Mflops 84.15
0 KSP Residual norm 5.169560044060e+02
SOLVE time 0.00
Solve flops 8.245800e+04 Mflops 90.42
1 KSP Residual norm 2.813291076609e+02
SOLVE time 0.00
Solve flops 8.245800e+04 Mflops 98.51
2 KSP Residual norm 1.688683980433e+02
SOLVE time 0.00
Solve flops 8.245800e+04 Mflops 87.27
3 KSP Residual norm 7.436784980024e+01
SOLVE time 0.00
Solve flops 8.245800e+04 Mflops 88.77
4 KSP Residual norm 1.835640800835e+01
SOLVE time 0.00
Solve flops 8.245800e+04 Mflops 89.55
5 KSP Residual norm 2.431147365563e+00
SOLVE time 0.00
Solve flops 8.245800e+04 Mflops 88.00
6 KSP Residual norm 5.386963293959e-01
SOLVE time 0.00
Solve flops 8.245800e+04 Mflops 93.50
7 KSP Residual norm 2.093714782067e-01
SOLVE time 0.00
Solve flops 8.245800e+04 Mflops 91.91
8 KSP Residual norm 1.306701698197e-02
SOLVE time 0.00
Solve flops 8.245800e+04 Mflops 89.44
9 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.

16.5.3 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:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.000e+02</td>
<td>125</td>
<td>8.28e-01</td>
<td>71.8</td>
<td>8.2</td>
<td>0.4</td>
<td>0.6</td>
</tr>
<tr>
<td>2.000e+02</td>
<td>44</td>
<td>3.00e-01</td>
<td>69.4</td>
<td>8.1</td>
<td>0.4</td>
<td>2.1</td>
</tr>
</tbody>
</table>

whilst the PETSc solver with LU preconditioner outputs:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.000e+02</td>
<td>146</td>
<td>1.15e+00</td>
<td>61.9</td>
<td>20.5</td>
<td>0.5</td>
<td>0.9</td>
</tr>
<tr>
<td>2.000e+02</td>
<td>42</td>
<td>3.30e-01</td>
<td>58.2</td>
<td>20.2</td>
<td>0.4</td>
<td>3.7</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:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.000e+02</td>
<td>130</td>
<td>2.66e+00</td>
<td>24.1</td>
<td>68.3</td>
<td>0.2</td>
<td>0.8</td>
</tr>
<tr>
<td>2.000e+02</td>
<td>78</td>
<td>1.16e+00</td>
<td>33.8</td>
<td>54.9</td>
<td>0.3</td>
<td>1.1</td>
</tr>
</tbody>
</table>

and with SOR preconditioner:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.000e+02</td>
<td>124</td>
<td>1.54e+00</td>
<td>38.6</td>
<td>50.2</td>
<td>0.3</td>
<td>0.4</td>
</tr>
<tr>
<td>2.000e+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>
</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>
</tr>
</thead>
<tbody>
<tr>
<td>1.000e+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>
</tr>
<tr>
<td>2.000e+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>
</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>
</tr>
</thead>
<tbody>
<tr>
<td>1.000e+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>
</tr>
<tr>
<td>2.000e+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>
</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>
</tr>
</thead>
<tbody>
<tr>
<td>1.000e+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>
</tr>
<tr>
<td>2.000e+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>
</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>
</tr>
</thead>
<tbody>
<tr>
<td>1.000e+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>
</tr>
<tr>
<td>2.000e+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>
</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>
</tr>
</thead>
<tbody>
<tr>
<td>1.000e+02</td>
<td>142</td>
<td>1.86e+00</td>
<td>83.3</td>
<td>9.5</td>
<td>0.2</td>
</tr>
<tr>
<td>2.000e+02</td>
<td>41</td>
<td>5.04e-01</td>
<td>83.1</td>
<td>8.0</td>
<td>0.3</td>
</tr>
</tbody>
</table>

so around 9% of the run-time is in setting the coefficients, and the remaining ~ 60% in the solve itself.
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 $\nabla_{||}$).

17.1 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\textsuperscript{nd} order $f_{-1} - 2f_0 + f_1$
  - C4: 4\textsuperscript{th} order $(-f_{-2} + 16f_{-1} - 30f_0 + 16f_1 - f_2)/12$
  - S2: 2\textsuperscript{nd} order smoothing derivative
  - W2: 2\textsuperscript{nd} order CWENO
  - W3: 3\textsuperscript{rd} order CWENO

- Upwinding methods for advection operators $v_x \frac{df}{dx}$
  - U1: 1\textsuperscript{st} order upwinding
  - U2: 2\textsuperscript{nd} order upwinding
  - U3: 3\textsuperscript{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
  - C2: $2^{nd}$ order central
  - C4: $4^{th}$ order central

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 $\frac{d}{dx}(v_x f) = v_x \frac{df}{dx} + f \frac{dv}{dx}$

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:

```
#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 17.1: Coordinate derivatives

<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>D2DX2(f)</td>
<td>$\frac{\partial^2 f}{\partial x^2}$</td>
</tr>
<tr>
<td>D2DY2(f)</td>
<td>$\frac{\partial^2 f}{\partial y^2}$</td>
</tr>
<tr>
<td>D2DZ2(f)</td>
<td>$\frac{\partial^2 f}{\partial z^2}$</td>
</tr>
<tr>
<td>D4DX4(f)</td>
<td>$\frac{\partial^4 f}{\partial x^4}$</td>
</tr>
<tr>
<td>D4DY4(f)</td>
<td>$\frac{\partial^4 f}{\partial y^4}$</td>
</tr>
<tr>
<td>D4DZ4(f)</td>
<td>$\frac{\partial^4 f}{\partial z^4}$</td>
</tr>
<tr>
<td>D2DXDZ(f)</td>
<td>$\frac{\partial^2 f}{\partial x \partial z}$</td>
</tr>
<tr>
<td>D2DYDZ(f)</td>
<td>$\frac{\partial^2 f}{\partial y \partial z}$</td>
</tr>
<tr>
<td>VDDX(f, g)</td>
<td>$f \frac{\partial g}{\partial x}$</td>
</tr>
<tr>
<td>VDDY(f, g)</td>
<td>$f \frac{\partial g}{\partial y}$</td>
</tr>
<tr>
<td>VDDZ(f, g)</td>
<td>$f \frac{\partial g}{\partial z}$</td>
</tr>
<tr>
<td>FDDX(f, g)</td>
<td>$\frac{\partial}{\partial x}(f \ast g)$</td>
</tr>
<tr>
<td>FDDY(f, g)</td>
<td>$\frac{\partial}{\partial y}(f \ast g)$</td>
</tr>
<tr>
<td>FDDZ(f, g)</td>
<td>$\frac{\partial}{\partial z}(f \ast g)$</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.

### 17.2 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.

```cpp
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:

```cpp
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:

```cpp
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.

### 17.3 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

\[
D2DXY(f) = DDX(DDY(f)) \\
D2DYDZ(f) = DDZ(DDY(f))
\]

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 D2DXY(Field3D f) {
    auto result = emptyFrom(f);
    // Calculate D2DXY here
}
```

(continues on next page)
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 yup and 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.

### 17.4 Non-uniform meshes

**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

\[
\frac{\partial^2 f}{\partial x^2} \approx \frac{1}{\Delta x^2} \frac{\partial^2 f}{\partial i^2} + \frac{1}{\Delta x} \frac{\partial f}{\partial i} \cdot \frac{1}{\Delta x} \frac{\partial}{\partial i} \left( \frac{1}{\Delta x} \right)
\]

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.

\(^1\) 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 (?).
17.5 General operators

These are differential operators which are for a general coordinate system.

\[ \mathbf{v} = \nabla f \quad \text{Vector} = \quad \text{Grad}(\text{Field}) \]
\[ f = \nabla \cdot \mathbf{a} \quad \text{Field} = \quad \text{Div}(\text{Vector}) \]
\[ \mathbf{v} = \nabla \times \mathbf{a} \quad \text{Vector} = \quad \text{Curl}(\text{Vector}) \]
\[ f = \mathbf{v} \cdot \nabla g \quad \text{Field} = \quad \text{V}_\text{dot}_\text{Grad}(\text{Vector, Field}) \]
\[ \mathbf{v} = \mathbf{a} \cdot \nabla c \quad \text{Vector} = \quad \text{V}_\text{dot}_\text{Grad}(\text{Vector, Vector}) \]
\[ f = \nabla^2 f \quad \text{Field} = \quad \text{Laplace}(\text{Field}) \]

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).

17.6 Clebsch operators

Another set of operators assume that the equilibrium magnetic field is written in Clebsch form as

\[ \mathbf{B}_0 = \nabla z \times \nabla x \quad \mathbf{B}_0 = \sqrt{g_{yy}} \]

where

\[ \mathbf{B}_0 = |\mathbf{B}_0| \mathbf{b}_0 = B_0 \mathbf{b}_0 \]

is the background equilibrium magnetic field.

<table>
<thead>
<tr>
<th>Function</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grad_par</td>
<td>[ \partial^p</td>
</tr>
<tr>
<td>Div_par</td>
<td>[ \nabla^2 \phi = B_0 \partial^p</td>
</tr>
<tr>
<td>Grad2_par2</td>
<td>[ \partial^p_2 \phi = \partial^p \partial^p \phi = \frac{1}{\sqrt{g_{xx}}} \frac{\partial}{\partial y} \left( \frac{\partial}{\partial x} \frac{\partial}{\partial y} \phi \right) + \frac{1}{\sqrt{g_{xx}}} \frac{\partial^2 \phi}{\partial u^i \partial u^j} ]</td>
</tr>
<tr>
<td>Laplace_par</td>
<td>[ \nabla^2 \phi = \nabla_\parallel \phi = \mathbf{b}<em>0 \cdot \nabla \phi = \frac{1}{\sqrt{g</em>{yy}}} \frac{\partial}{\partial y} \left( \frac{\partial}{\partial x} \frac{\partial}{\partial y} \phi \right) ]</td>
</tr>
<tr>
<td>Laplace_perp</td>
<td>[ \nabla^2_{\perp} = \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 brackets</td>
</tr>
<tr>
<td></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

\[ \mathbf{b}_0 \cdot \nabla \phi \times \nabla A = \frac{1}{J \sqrt{g_{yy}}} \left[ (g_{yy} \frac{\partial \phi}{\partial z} - g_{yz} \frac{\partial \phi}{\partial y}) \frac{\partial A}{\partial x} + (g_{yz} \frac{\partial \phi}{\partial x} - g_{xy} \frac{\partial \phi}{\partial y}) \frac{\partial A}{\partial y} + (g_{xy} \frac{\partial \phi}{\partial y} - g_{yx} \frac{\partial \phi}{\partial x}) \frac{\partial A}{\partial z} \right] \]
\[ \nabla_\perp \equiv \nabla - b (b \cdot \nabla) \]
\[ b \cdot \nabla = \frac{1}{JB} \frac{\partial}{\partial y} \]
\[ b = \frac{1}{JB} e_y = \frac{1}{JB} [g_{xy} \nabla x + g_{yy} \nabla y + g_{yz} \nabla z] \]

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:

\[ \nabla_\perp = \nabla x \left( \frac{\partial}{\partial x} - \frac{g_{xy}}{(JB)^2} \frac{\partial}{\partial y} \right) + \nabla z \left( \frac{\partial}{\partial z} - \frac{g_{yz}}{(JB)^2} \frac{\partial}{\partial y} \right) \]

### 17.7 The bracket operators

The bracket operator \( \text{brackets}(\phi, f, \text{method}) \) aims to differentiate equations on the form

\[ -\nabla \phi \times b \cdot \nabla f \]

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.

### 17.8 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

\[ \sum_i (fJdx dy dz)_i = \text{const} \]

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:

```
#include "bout/fv_ops.hxx"
```

**Note** The methods are defined in a namespace \( \text{FV} \).

Some methods (those with templates) are defined in the header, but others are defined in \( \text{src/mesh/fv_ops.cxx} \).

#### 17.8.1 Parallel divergence \( \text{Div\_par} \)

This function calculates the divergence of a flow in \( y \) (parallel to the magnetic field) by a given velocity.

```
template<\text{typename CellEdges} = \text{MC}>
\text{const Field3D Div\_par(\text{const Field3D } &f\_in, \text{const Field3D } &v\_in,}
\text{\text{const Field3D } &a, \text{bool fixflux=\text{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.

\[
\text{ddt}(n) = -\text{FV::Div}_{\text{par}}(n, v, \text{cs});
\]

By default the MC slope limiter is used to calculate cell edges, but this can be changed at compile time e.g:

\[
\text{ddt}(n) = -\text{FV::Div}_{\text{par}}<\text{FV::Fromm}> (n, v, \text{cs});
\]

A list of available limiters is given in section \textit{Slope limiters} below.

**Example and convergence test**

The example code \texttt{examples/finite-volume/fluid/} solves the Euler equations for a 1D adiabatic fluid, using \texttt{FV::Div}_{\text{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 \( nv_{||} \) is the momentum in the direction parallel to the magnetic field. The operator \( \nabla_{||} \) represents the divergence of a parallel flow (\texttt{Div}_{\text{par}}), and \( \partial_{||} = b \cdot \nabla \) is the gradient in the parallel direction.

There is a convergence test using the Method of Manufactured Solutions (MMS) for this example. See section \textit{Method of Manufactured Solutions} for details of the testing method. Running the \texttt{runtest} script should produce the graph.

### 17.8.2 Parallel diffusion

The parallel diffusion operator calculates \( \nabla_{||} [k\partial_{||} (f)] \)

\[
\text{const Field3D Div}_{\text{par}}_{\text{K} \_ \text{Grad}_{\text{par}}}(\text{const Field3D} \&k, \text{const Field3D} \&f, \text{bool} \text{bndry}\_\text{flux}=\text{true});
\]

This is done by calculating the flux \( k\partial_{||} (f) \) on cell boundaries using central differencing.

### 17.8.3 Advection in 3D

This operator calculates \( \nabla \cdot (nv) \) where \( v \) is a 3D vector. It is written in flux form by discretising the expression

\[
\nabla \cdot (A) = \frac{1}{J} \partial_i (JA^i)
\]

Like the \texttt{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.

\[
\text{template<typename CellEdges = MC>}
\text{\text{const Field3D Div}_{f \_v}(\text{\text{const Field3D} \&n, \text{\text{const Vector3D} \&v, \text{\text{bool} bndry}\_\text{flux})}}
\]
Fig. 17.1: Convergence test, 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.
17.8.4 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.

17.8.5 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:

```plaintext
StaggerGrids = true
```

allows quantities to be defined on cell boundaries. Functions such as $\text{DDX}$ 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 differencing using 4-point stencil:

$$y = \frac{(9y_{-1/2} + 9y_{1/2} - y_{-3/2} - y_{3/2})}{16}$$

$$\frac{\partial y}{\partial x} = \frac{(27y_{1/2} - 27y_{-1/2} - y_{3/2} + y_{-3/2})}{24\Delta x}$$

$$\frac{\partial^2 y}{\partial x^2} = \frac{(y_{3/2} + y_{-3/2} - y_{1/2} - y_{-1/2})}{2\Delta x^2}$$
17.9 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

\[ \int_{-\infty}^{\infty} (\partial_z f[x, y, z]) \exp(-2\pi i z \xi) \, dz \]
\[ = \int_{-\infty}^{\infty} \partial_z (f[x, y, z] \exp[-2\pi i z \xi]) \, dz - \int_{-\infty}^{\infty} f(x, y, z) \partial_z \exp(-2\pi i z \xi) \, dz \]
\[ = (f[x, y, z] \exp[-2\pi i z \xi]) \bigg|_{-\infty}^{\infty} - (-2\pi i \xi) \int_{-\infty}^{\infty} f(x, y, z) \exp(-2\pi i z \xi) \, dz \]
\[ = 2\pi i \xi F(x, y, \xi) \]

where we have used that \( f(x, y, \pm \infty) = 0 \) in order to have a well defined Fourier transform. This means that

\[ \partial^n_z F(x, y, \xi) = (2\pi i \xi)^n F(x, y, \xi) \]

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

---

\(^2\) For more detail see Bracewell, R. N. - The Fourier Transform and Its Applications 3rd Edition chapter 10
\(N + 1\) 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

\[
F(x, y)_k = \frac{1}{N} \sum_{Z=0}^{N-1} f(x, y)_Z \exp\left(-\frac{2\pi i k Z}{N}\right)
\]  

(17.2)

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 \(z_Z = Zdz\). As our domain goes from \([0, 2\pi]\), we have that (since we have one less line segment than point) \(dz(N - 1) = L_z = 2\pi - dz\), which gives \(dz = \frac{2\pi}{N}\). Inserting this in equation ((17.2)) yields

\[
F(x, y)_k = \frac{1}{N} \sum_{Z=0}^{N-1} f(x, y)_Z \exp(-i k Z dz) = \frac{1}{N} \sum_{Z=0}^{N-1} f(x, y)_Z \exp(-i k z_Z)
\]

The discrete version of equation ((17.1)) thus gives

\[
\partial_z^n F(x, y)_k = (i k)^n F(x, y)_k
\]
BOUT++ provides a wide variety of algebraic operators acting on fields.
The algebraic operators are listed in Table 18.1. For a completely up-to-date list, see the Non-member functions part of field2d.hxx, field3d.hxx, fieldperp.hxx.
<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>$l^h s^{r h s}$</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 $z_{\text{max}}$</td>
</tr>
<tr>
<td>lowpass(f, nmax, nmin, region)</td>
<td>Remove Fourier modes (in the z-direction) with mode number higher than $z_{\text{max}}$ or lower than $z_{\text{min}}$</td>
</tr>
<tr>
<td>shiftZ(f, angle, region)</td>
<td>Rotate $f$ by angle in the z-direction. $\text{angle}/2\pi$ 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 be\(^1\) (see *Iterating over fields*)
- \texttt{RGN\_ALL}, which is the whole mesh;
- \texttt{RGN\_NOBNDRY}, which skips all boundaries;
- \texttt{RGN\_NOX}, which skips the x boundaries
- \texttt{RGN\_NOY}, which skips the y boundaries

The default value for the region argument is \texttt{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.

\(^1\) More regions may be added in future, for example to act on only subsets of the physical domain.
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. 19.1.

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:
Fig. 19.1: The four possible cell locations for defining quantities
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

```cpp
#include <interpolation.hxx>
```

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 Grad_par(v) would be CELL_YLOW. If this is not what is wanted, give the location of the result as an additional argument: Grad_par(v, CELL_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 Vpar_Grad_par(v, f) calculates \(v \nabla || f\) and returns a result at the same location as \(f\). If \(v\) and \(f\) 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 (\(f\) 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 \(A||\) 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.

20.1 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.

20.2 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.

20.3 Examples

20.3.1 Wave in a box

examples/eigen-box
21.1 Spitzer-Harm heat flux

The Spitzer-Harm heat flux \( q_{SH} \) is calculated using

\[
q_{SH} = -\frac{n_e e T_e}{m_e} \frac{3\sqrt{\pi}}{4} \tau_{ei,T} \kappa_0 \frac{Z + 0.24}{Z + 4.2} \partial || T_e
\]

where \( n_e \) is the electron density in \( m^{-3} \), \( T_e \) is the electron temperature in eV, \( \kappaappa_0 = 13.58 \), \( Z \) is the average ion charge. The resulting expression is in units of \( eV/m^2/s \).

The thermal collision time \( \tau_{ei,T} = \lambda_{ei,T}/v_T \) is calculated using the thermal mean free path and thermal velocity:

\[
\lambda_{ee,T} = \frac{v_T^4}{Y n_e \ln \Lambda}
\]
\[
\lambda_{ei,T} = \frac{v_T^4}{Y Z^2 n_i \ln \Lambda}
\]
\[
v_T = \sqrt{\frac{2e T_e}{m_e}}
\]

where it is assumed that \( n_i = n_e \), and the following are used:

\[
Y = 4\pi \left( \frac{e^2}{4\pi e_0 m_e} \right)^2
\]
\[
\ln \Lambda = 6.6 - 0.5 \log \left( \frac{n_e}{10^{20}} \right) + 1.5 \log (T_e)
\]

Note: If comparing to online notes, \( \kappaappa_0 \frac{Z + 0.24}{Z + 4.2} \approx 3.2 \), a different definition of collision time \( \tau_{ei} \) is used here, but the other factors are included so that the heat flux \( q_{SH} \) is the same here as in those notes.
21.2 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 \( \beta = E_g/eT_e \) where \( E_g \) is the energy of the group.

\[
\left[ \frac{1}{\lambda'_{g,ee}} - \nabla_{||} \left( \frac{\lambda'_{g,ei}}{3} \partial_{||} \right) \right] H_g = -\nabla_{||} U_g
\]

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:

\[
\lambda'_{g,ee} = \beta^2 \lambda_{ee, T}/r
\]
\[
\lambda'_{g,ei} = \beta^2 \lambda_{ei, T} \frac{Z + 0.24}{Z + 4.2} - 1
\]

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:

\[
\nabla_{||} \left( \frac{\lambda'_{g,ei}}{3} \partial_{||} \right) H_g = \nabla_{||} U_g + \frac{H_g}{\lambda'_{g,ee}}
\]

and so calculate the divergence of the heat flux as:

\[
\nabla_{||} q_{SNB} = \nabla_{||} q_{SH} - \sum_g \left( \nabla_{||} U_g + \frac{H_g}{\lambda'_{g,ee}} \right)
\]

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:

\[
\nabla_{||} \left( k \partial_{||} T \right) = \frac{1}{J} \frac{\partial}{\partial y} \left( \frac{kJ}{g_{22}} \frac{\partial T}{\partial y} \right) = k \frac{1}{g_{22}} \frac{\partial^2 T}{\partial y^2} + \frac{1}{J} \frac{\partial}{\partial y} \left( \frac{kJ}{g_{22}} \right) \frac{\partial T}{\partial y}
\]

21.2.1 Using the SNB model

To use the SNB model, first include the header:

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

then create an instance:

```cpp
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 21.1: 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</td>
</tr>
<tr>
<td>ngroups</td>
<td>Number of energy groups</td>
<td>40</td>
</tr>
<tr>
<td>r</td>
<td>Scaling down the electron-electron mean free path</td>
<td>2</td>
</tr>
</tbody>
</table>

The divergence of the heat flux can then be calculated:

```
Field3D Div_q = snb.divHeatFlux(Te, Ne);
```

where $Te$ is the temperature in eV, and $Ne$ 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.

### 21.2.2 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. 21.1.

### 21.2.3 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. 21.2.
Fig. 21.1: 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. 21.2: Temperature profile and heat flux calculated using Spitzer-Harm and the SNB model, for a temperature step profile, at a density of 5e26 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 `std::function<T(const T&)>`. This can be a `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:

```c++
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.
    }
}
```

(continues on next page)
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::InveratbleOperator<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
// 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_laplacexyksp_type 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.
24.1 Introduction

This manual covers the field-aligned coordinate system used in many BOUT++ tokamak models, and useful derivations and expressions.

24.2 Orthogonal toroidal coordinates

Starting with an orthogonal toroidal coordinate 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

\[
B = B_\theta \nabla \theta + B_\zeta \nabla \zeta \\
= B_\theta e_\theta + B_\zeta e_\zeta \\
= B_{pol} h_\theta e_\theta + B_{tor} R e_\zeta \\
= B_{pol} \hat{e}_\theta + B_{tor} \hat{e}_\zeta
\]

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

\[ |\nabla \psi| = R |B_{\text{pol}}| \quad |\nabla \theta| = \frac{1}{h_{\theta}} \quad |\nabla \zeta| = \frac{1}{R} \]

Because the coordinate system is orthogonal, \( g^{ij} = 1/g_{ij} \) and so the cross-products can be calculated as

\[ \nabla \psi \times \nabla \theta = \hat{e}_\psi \times \hat{e}_\theta = g^{\psi \psi} e_\psi \times g^{\theta \theta} e_\theta = \frac{1}{h_{\psi} h_{\theta}} \hat{e}_\zeta = R |B_{\text{pol}}| \hat{e}_\zeta \]

Similarly,

\[ \nabla \psi \times \nabla \zeta = -|B_{\text{pol}}| \hat{e}_\theta \quad \nabla \theta \times \nabla \zeta = \frac{1}{h_{\theta} R^2 |B_{\text{pol}}|} \nabla \psi \]

### 24.3 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:

\[ x = \sigma_{B \theta} (\psi - \psi_0) \quad y = \theta \quad z = \sigma_{B \theta} \left( \zeta - \int_{\psi_0}^{\theta} \nu (\psi, \theta) d\theta \right) \quad (24.1) \]

Where \( \nu \) is the local field-line pitch given by

\[ \nu (\psi, \theta) = \frac{B \cdot \nabla \zeta}{B \cdot \nabla \theta} = \frac{B_{\text{tor}} h_{\theta}}{B_{\text{pol}} R} = \frac{(F/R) h_{\theta}}{B_{\text{pol}} R} = F J/R^2 \]

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

\[ \nabla x = \sigma_{B \theta} \nabla \psi \quad \nabla y = \nabla \theta \quad \nabla z = \sigma_{B \theta} \left( \nabla \zeta - \left[ \int_{\psi_0}^{\theta} \frac{\partial \nu (\psi, \theta)}{\partial \psi} d\theta \right] \nabla \psi - \nu (\psi, \theta) \nabla \theta \right) \]

The term in square brackets is the integrated local shear:

\[ I = \int_{\psi_0}^{\theta} \frac{\partial \nu (x, y)}{\partial \psi} dy \]

#### 24.3.1 Magnetic field

Magnetic field is given in Clebsch form by:

\[ B = \nabla z \times \nabla x = \frac{1}{J} e_y \]

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

$$B_x = \sigma_{B_\theta} B_{\text{tor}} IR \quad B_y = \frac{B^2 h_\theta}{B_{\text{pol}}} \quad B_z = \sigma_{B_\theta} B_{\text{tor}} R$$

The unit vector in the direction of equilibrium $B$ is therefore

$$b = \frac{1}{JB} e_y = \frac{1}{JB} [g_{xy} \nabla x + g_{yy} \nabla y + g_{yz} \nabla z]$$

### 24.3.2 Jacobian and metric tensors

The Jacobian of this coordinate system is

$$J^{-1} \equiv (\nabla x \times \nabla y) \cdot \nabla z = B_{\text{pol}}/h_\theta$$

which can be either positive or negative, depending on the sign of $B_{\text{pol}}$. The contravariant metric tensor is given by:

$$g^{ij} \equiv e^i \cdot e^j \equiv \nabla u^i \cdot \nabla u^j = \begin{pmatrix}
\frac{(R B_{\text{pol}})^2}{h_\theta^2} & \frac{1}{h_\theta^2} & -I \frac{(R B_{\text{pol}})^2}{h_\theta^2} \\
-I \frac{(R B_{\text{pol}})^2}{h_\theta^2} & -\frac{\sigma_{B_\theta} \nu}{h_\theta^2} & I^2 \frac{(R B_{\text{pol}})^2}{R^2 (R B_{\text{pol}})^2}
\end{pmatrix}$$

and the covariant metric tensor:

$$g_{ij} \equiv e_i \cdot e_j = \begin{pmatrix}
\frac{I^2 R^2 + 1/(R B_{\text{pol}})^2}{\sigma_{B_\theta} B_{\text{tor}} h_\theta IR/B_{\text{pol}}} & \frac{\sigma_{B_\theta} B_{\text{tor}} h_\theta IR/B_{\text{pol}}}{IR^2} \\
\frac{\sigma_{B_\theta} B_{\text{tor}} h_\theta IR/B_{\text{pol}}}{IR^2} & \frac{B^2 h_\theta^2}{B_{\text{pol}}^2} & \frac{I R^2}{B_{\text{pol}}^2}
\end{pmatrix}$$

### 24.3.3 Differential operators

The derivative of a scalar field $f$ along the unperturbed magnetic field $b_0$ is given by

$$\partial_0^0 f \equiv b_0 \cdot \nabla f = \frac{1}{\sqrt{g_{yy}}} \frac{\partial f}{\partial y} = \frac{B_{\text{pol}}}{B h_\theta} \frac{\partial f}{\partial y}$$

whilst the parallel divergence is given by

$$\nabla_0^0 f = B_0 \partial_0^0 \left( \frac{f}{B_0} \right)$$

Using equation (11), the Laplacian operator is given by

$$\nabla^2 = \frac{\partial^2}{\partial x^2} |\nabla x|^2 + \frac{\partial^2}{\partial y^2} |\nabla y|^2 + \frac{\partial^2}{\partial z^2} |\nabla z|^2$$

$$-2 \frac{\partial^2}{\partial x \partial z} I (R B_{\text{pol}})^2 - 2 \frac{\partial^2}{\partial y \partial z} h_\theta^2$$

$$+ \frac{\partial}{\partial x} \nabla^2 x + \frac{\partial}{\partial y} \nabla^2 y + \frac{\partial}{\partial z} \nabla^2 z$$

Using equation (10) for $\nabla^2 x = G^x$ etc, the values are

$$\nabla^2 x = \frac{B_{\text{pol}}}{h_\theta} \frac{\partial}{\partial x} (h_\theta R^2 B_{\text{pol}}) \quad \nabla^2 y = \frac{B_{\text{pol}}}{h_\theta} \frac{\partial}{\partial y} \left( \frac{1}{B_{\text{pol}} h_\theta} \right)$$

### 24.3. Field-aligned coordinates
\[ \nabla^2 z = -\frac{B_{\text{pol}}}{h_\theta} \left[ \frac{\partial}{\partial x} (IR^2 B_{\text{pol}} h_\theta) + \frac{\partial}{\partial y} \left( \frac{\nu}{B_{\text{pol}} h_\theta} \right) \right] \]

Neglecting some parallel derivative terms, the perpendicular Laplacian can be written:

\[ \nabla^2_\perp = (RB_{\text{pol}})^2 \left[ \frac{\partial^2}{\partial x^2} - 2I \frac{\partial^2}{\partial z \partial x} + \left( I^2 + \frac{B^2}{(RB_{\text{pol}})^4} \right) \frac{\partial^2}{\partial z^2} \right] + \nabla_x^2 \frac{\partial}{\partial x} + \nabla_z^2 \frac{\partial}{\partial z} \]

The second derivative along the equilibrium field

\[ \frac{\partial^2 \phi}{\partial \perp^2} = \frac{\partial^2 \phi}{\partial x^2} \left( \frac{1}{\sqrt{g_{yy} g_{zz}}} \frac{\partial}{\partial y} \frac{1}{\sqrt{g_{yy} g_{zz}}} \frac{\partial}{\partial z} + \frac{1}{g_{yy} g_{zz}} \frac{\partial^2 \phi}{\partial x^2} \right) \]

A common expression (the Poisson bracket in reduced MHD) is (from equation (14)):

\[ b_0 \cdot \nabla \times A = \frac{1}{J \sqrt{g_{yy} g_{zz}}} \left[ \left( g_{yy} \frac{\partial \phi}{\partial z} - g_{yz} \frac{\partial \phi}{\partial y} \right) \frac{\partial A}{\partial x} + \left( g_{yz} \frac{\partial \phi}{\partial y} - g_{xy} \frac{\partial \phi}{\partial x} \right) \frac{\partial A}{\partial y} + \left( g_{xy} \frac{\partial \phi}{\partial x} - g_{yy} \frac{\partial \phi}{\partial y} \right) \frac{\partial A}{\partial z} \right] \]

The perpendicular nabla operator:

\[ \nabla_\perp \equiv \nabla - b (b \cdot \nabla) \]

\[ = \nabla_x \left( \frac{\partial}{\partial x} - \frac{g_{xy}}{(JB)^2} \frac{\partial}{\partial y} \right) + \nabla_z \left( \frac{\partial}{\partial z} - \frac{g_{yz}}{(JB)^2} \frac{\partial}{\partial y} \right) \]

### 24.3.4 J x B in field-aligned coordinates

Components of the magnetic field in field-aligned coordinates:

\[ B^y = \frac{B_{\text{pol}}}{h_\theta} \quad B^x = B^z = 0 \]

and

\[ B_x = \sigma_{B\theta} B_{\text{tor}} IR \quad B_y = \frac{B^2 h_\theta}{B_{\text{pol}}} \quad B_z = \sigma_{B\theta} B_{\text{tor}} R \]

Calculate current \( J = \frac{1}{\mu} \nabla \times B \)

\[ (\nabla \times B)^x = \frac{1}{J} \left( \frac{\partial B_z}{\partial y} - \frac{\partial B_y}{\partial z} \right) = 0 \]

since \( B_{\text{tor}} R \) is a flux-surface quantity, and \( B \) is axisymmetric.

\[ (\nabla \times B)^y = -\sigma_{B\theta} \frac{B_{\text{pol}}}{h_\theta} \frac{\partial}{\partial x} (B_{\text{tor}} R) \]

\[ (\nabla \times B)^z = \frac{B_{\text{pol}}}{h_\theta} \left[ \frac{\partial}{\partial x} \left( \frac{B^2 h_\theta}{B_{\text{pol}}} \right) - \sigma_{B\theta} \frac{\partial}{\partial y} (B_{\text{tor}} IR) \right] \]

The second term can be simplified, again using \( B_{\text{tor}} R \) constant on flux-surfaces:

\[ \frac{\partial}{\partial y} (B_{\text{tor}} IR) = \sigma_{B\theta} B_{\text{tor}} R \frac{\partial \nu}{\partial x} \quad \nu = \frac{h_\theta B_{\text{tor}} R}{RB_{\text{pol}}} \]

From these, calculate covariant components:

\[ (\nabla \times B)_x = -B_{\text{tor}} R \frac{\partial}{\partial x} (B_{\text{tor}} R) + \frac{IR^2 B_{\text{pol}}}{h_\theta} \left[ \frac{\partial}{\partial x} \left( \frac{B^2 h_\theta}{B_{\text{pol}}} \right) - B_{\text{tor}} R \frac{\partial \nu}{\partial x} \right] \]

\[ (\nabla \times B)_y = -\sigma_{B\theta} \frac{B_{\text{pol}}}{h_\theta} \frac{\partial}{\partial x} (B_{\text{tor}} R) + \sigma_{B\theta} B_{\text{tor}} R \left[ \frac{\partial}{\partial x} \left( \frac{B^2 h_\theta}{B_{\text{pol}}} \right) - B_{\text{tor}} R \frac{\partial \nu}{\partial x} \right] \]

\[ (\nabla \times B)_z = -B_{\text{tor}} R \frac{\partial}{\partial x} (B_{\text{tor}} R) + \frac{R^2 B_{\text{pol}}}{h_\theta} \left[ \frac{\partial}{\partial x} \left( \frac{B^2 h_\theta}{B_{\text{pol}}} \right) - B_{\text{tor}} R \frac{\partial \nu}{\partial x} \right] \]
Calculate $J \times B$ using

$$e^i = \frac{1}{J} (e_j \times e_k) \quad e_i = J (e^j \times e^k) \quad i, j, k \in \{1, 2, 3\}$$

gives

$$\mu_0 (J \times B)^x = \frac{1}{J} \left[ (\nabla \times B)_y B_z - (\nabla \times B)_z B_y \right]$$

$$= - \frac{B_{pol}^2 R^2}{h_\theta} \left( \frac{\partial}{\partial x} \left( \frac{B^2 h_\theta}{B_{pol}} \right) - B_{tor} R \frac{\partial \nu}{\partial x} \right)$$

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 $J \times B = \nabla P$,

$$\frac{\partial}{\partial x} \left( B^2 h_\theta \frac{B_{pol}}{B_{pol}} \right) - B_{tor} R \frac{\partial}{\partial x} \left( \frac{B_{tor} h_\theta}{RB_{pol}} \right) + \mu_0 h_\theta \frac{\partial P}{\partial x} = 0 \quad \text{(24.2)}$$

Use this to calculate $h_\theta$ profiles (needs 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

$$B_{tor} h_\theta \frac{\partial B_{tor}}{\partial x} + \frac{B_{tor}^2 h_\theta}{R} \frac{\partial R}{\partial x} + \mu_0 h_\theta \frac{\partial P}{\partial x} = -B_{pol} \frac{\partial}{\partial x} \left( B_{pol} h_\theta \right)$$

A quick way to calculate $f$ is to rearrange this to:

$$\frac{\partial B_{tor}}{\partial x} = B_{tor} \left[ -\frac{1}{R} \frac{\partial R}{\partial x} \right] + \frac{1}{B_{tor}} \left[ -\mu_0 \frac{\partial P}{\partial x} - \frac{\partial B_{pol}}{\partial h_\theta} \frac{\partial}{\partial x} \left( B_{pol} h_\theta \right) \right]$$

and then integrate this using LSODE.

### 24.3.5 Parallel current

$$J_{||} = b \cdot J \quad b^y = \frac{B_{pol}}{B h_\theta}$$

and from equation (24.2):

$$J_y = \frac{\sigma B_\theta}{\mu_0} \left\{ -\frac{B^2 h_\theta}{B_{pol}} \frac{\partial}{\partial x} \left( B_{tor} R + B_{tor} R \left( \frac{\partial}{\partial x} \left( \frac{B^2 h_\theta}{B_{pol}} \right) - \sigma B_\theta B_{tor} R \frac{\partial \nu}{\partial x} \right) \right) \right\}$$

since $J_{||} = b^y J_y$,

$$\mu_0 J_{||} = \sigma B_\theta \frac{B_{pol} B_{tor} R}{B h_\theta} \left[ \frac{\partial}{\partial x} \left( \frac{B^2 h_\theta}{B_{pol}} \right) - B_{tor} R \frac{\partial \nu}{\partial x} \right] - \sigma B_\theta B \frac{\partial}{\partial x} \left( B_{tor} R \right)$$
24.3.6 Curvature

For reduced MHD, need to calculate curvature term \( b \times \kappa \), where \( \kappa = (b \cdot \nabla) b = -b \times (\nabla \times b) \). Re-arranging, this becomes:

\[
b \times \kappa = \nabla \times b - b (\nabla \times b)
\]

Components of \( \nabla \times b \) are:

\[
(\nabla \times b)^x = \sigma_{B \theta} B_{\text{pol}} \frac{\partial}{\partial y} \left( \frac{B_{\text{tor}} R}{B} \right)
\]

\[
(\nabla \times b)^y = -\sigma_{B \theta} B_{\text{pol}} \frac{\partial}{\partial x} \left( \frac{B_{\text{tor}} R}{B} \right)
\]

\[
(\nabla \times b)^z = \frac{B_{\text{pol}}}{B_h \theta} \frac{\partial}{\partial x} \left( \frac{B_{\text{tor}}}{B_{\text{pol}}} \right) - \sigma_{B \theta} \frac{B_{\text{pol}} B_{\text{tor}}}{B h \theta} \frac{\partial}{\partial x} \left( \frac{B_{\text{tor}} R}{B} \right) - \sigma_{B \theta} \frac{B_{\text{pol}}}{B h \theta} \frac{\partial}{\partial y} \left( \frac{B_{\text{tor}} R}{B} \right)
\]

giving:

\[
\kappa = -\frac{B_{\text{pol}}}{B h \theta} \left[ \frac{\partial}{\partial x} \left( \frac{B_{\text{tor}}}{B_{\text{pol}}} \right) - \sigma_{B \theta} \frac{\partial}{\partial y} \left( \frac{B_{\text{tor}} R}{B} \right) \right] \nabla x
\]

\[
+ \sigma_{B \theta} \frac{B_{\text{pol}}}{B h \theta} \frac{\partial}{\partial y} \left( \frac{B_{\text{tor}} R}{B} \right) \nabla z
\]

\[
b \cdot (\nabla \times b) = -\sigma_{B \theta} B \frac{\partial}{\partial x} \left( \frac{B_{\text{tor}} R}{B} \right) + \sigma_{B \theta} \frac{B_{\text{pol}}}{B h \theta} \frac{\partial}{\partial x} \left( \frac{B_{\text{tor}}}{B_{\text{pol}}} \right) - \frac{B_{\text{pol}} B_{\text{tor}}^2 R^2}{B h \theta} \frac{\partial \nu}{\partial x}
\]

therefore,

\[
(b \times \kappa)^x = \sigma_{B \theta} B_{\text{pol}} \frac{\partial}{\partial y} \left( \frac{B_{\text{tor}} R}{B} \right) = -\sigma_{B \theta} B_{\text{pol}} B_{\text{tor}} R \frac{\partial B}{B h \theta} \frac{\partial B}{\partial y}
\]

\[
(b \times \kappa)^y = \frac{B_{\text{pol}} B_{\text{tor}}^2 R^2}{B^3 h \theta^2} \frac{\partial \nu}{\partial x} - \sigma_{B \theta} \frac{B_{\text{pol}} B_{\text{tor}}}{B^2 h \theta^2} \frac{\partial}{\partial x} \left( \frac{B_{\text{tor}}}{B_{\text{pol}}} \right)
\]

\[
(b \times \kappa)^z = \frac{B_{\text{pol}}}{B h \theta} \frac{\partial}{\partial x} \left( \frac{B_{\text{tor}}}{B_{\text{pol}}} \right) - \sigma_{B \theta} \frac{B_{\text{pol}} B_{\text{tor}}}{B h \theta} \frac{\partial}{\partial x} \left( \frac{B_{\text{tor}} R}{B} \right) - I (b \times \kappa)^x
\]

Using equation (24.2):

\[
B \frac{\partial}{\partial x} \left( \frac{B_{\text{tor}}}{B_{\text{pol}}} \right) + B_{\text{tor}} \frac{\partial B}{B_{\text{pol}}} - \sigma_{B \theta} B_{\text{tor}} R \frac{\partial}{\partial x} \left( \frac{B_{\text{tor}} h \theta}{B_{\text{pol}}} \right) + \frac{\mu_0 h \theta}{B_{\text{pol}}} \frac{\partial P}{\partial x} = 0
\]

we can re-write the above components as:

\[
(b \times \kappa)^y = \sigma_{B \theta} \frac{B_{\text{pol}} B_{\text{tor}}}{B^2 h \theta} \left[ \frac{\mu_0}{B} \frac{\partial P}{\partial x} + \frac{\partial B}{\partial x} \right]
\]

\[
(b \times \kappa)^z = -\frac{\mu_0}{B} \frac{\partial P}{\partial x} - \frac{\partial B}{\partial x} - I (b \times \kappa)^x
\]

24.3.7 Curvature from \( b \times \kappa \)

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):

\[
\nabla \times \begin{pmatrix} b \\ B \end{pmatrix} = \frac{B_{\text{pol}}}{h_\theta} \left[ \left( \frac{\partial}{\partial x} \left( \frac{h_\theta}{B_{\text{pol}}} \right) - \frac{\partial}{\partial y} \left( \frac{\sigma_B B_{\text{pol}} I R}{B^2} \right) \right) e_z \right. \\
+ \left. \frac{\partial}{\partial y} \left( \frac{\sigma_B B_{\text{pol}} R}{B^2} \right) e_x \right]
\]

This can be simplified using

\[
\frac{\partial}{\partial y} \left( \frac{\sigma_B B_{\text{pol}} I R}{B^2} \right) = I \sigma_B B_{\text{tor}} R \frac{\partial}{\partial y} \left( \frac{1}{B^2} \right) + \frac{B_{\text{pol}} R \partial \nu}{B^2} \frac{\partial}{\partial x}
\]

to give

\[
\begin{align*}
(b \times \kappa)^x &= - \sigma_B B_{\text{pol}} B_{\text{tor}} R \frac{\partial B}{h_\theta B^2} \\
(b \times \kappa)^y &= - \sigma_B B_{\text{pol}} \frac{B_{\text{tor}} R}{2 h_\theta} \\
(b \times \kappa)^z &= \frac{B_{\text{pol}}}{2 h_\theta} \frac{\partial}{\partial x} \left( \frac{h_\theta}{B_{\text{pol}}} \right) - \frac{B_{\text{pol}} B_{\text{tor}} R \partial \nu}{2 h_\theta B} \frac{\partial}{\partial x} - I (b \times \kappa \cdot \nabla)^z
\end{align*}
\]

The first and second terms in \((b \times \kappa \cdot \nabla)^z\) almost cancel, so by expanding out \(\nu\) a better expression is

\[
(b \times \kappa)^z = \frac{B_{\text{pol}}}{2 h_\theta B} \frac{\partial}{\partial x} \left( \frac{h_\theta}{B_{\text{pol}}} \right) - \frac{B_{\text{pol}} B_{\text{tor}} R \partial \nu}{2 h_\theta B} \frac{\partial}{\partial x} - I (b \times \kappa \cdot \nabla)^z
\]

### 24.3.8 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

\[
\hat{T} = \frac{dr}{ds} = \frac{dR}{ds} \hat{R} + \frac{dZ}{ds} \hat{Z} + R \frac{d\phi}{ds} \hat{\phi}
\]

where \(s\) is the distance along the field-line. From this, the curvature vector is given by

\[
\kappa = \frac{d\hat{T}}{ds} = \frac{d^2 R}{ds^2} \hat{R} + \frac{dR}{ds} \frac{d\phi}{ds} \hat{\phi} + \frac{d^2 Z}{ds^2} \hat{Z}
\]
\[
+ \frac{dR}{ds} \frac{d\phi}{ds} \hat{\phi} + R \frac{d^2 \phi}{ds^2} \hat{\phi} - R \left( \frac{d\phi}{ds} \right)^2 \hat{R}
\]

i.e.

\[
\kappa = \left[ \frac{d^2 R}{ds^2} - R \left( \frac{d\phi}{ds} \right)^2 \right] \hat{R} + \frac{d^2 Z}{ds^2} \hat{Z} + \left[ \frac{dR}{ds} + R \frac{d^2 \phi}{ds^2} \right] \hat{\phi}
\]  \(24.3\)
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

\[
\hat{R} \times \hat{Z} = -\hat{\phi} \quad \hat{\phi} \times \hat{Z} = \hat{R} \quad \hat{R} \times \hat{\phi} = \hat{Z}
\]

This vector must then be dotted with \( \nabla \psi, \nabla \theta, \) and \( \nabla \phi \). This is done by writing these vectors in cylindrical coordinates:

\[
\nabla \psi = \frac{\partial \psi}{\partial R} \hat{R} + \frac{\partial \psi}{\partial Z} \hat{Z}
\]

\[
\nabla \theta = \frac{1}{B_{pol} h_\theta} \nabla \phi \times \nabla \psi = \frac{1}{RB_{pol} h_\theta} \left( \frac{\partial \psi}{\partial Z} \hat{R} - \frac{\partial \psi}{\partial R} \hat{Z} \right)
\]

An alternative is to use

\[
b \times \nabla \phi = \frac{\sigma_{B\theta}}{BR^2} \nabla \psi
\]

and that the tangent vector \( T = b \). This gives

\[
\nabla \psi = \sigma_{B\theta} BR \left[ \frac{dR}{ds} Z - \frac{dZ}{ds} R \right]
\]

and so because \( d\phi/ds = B_{tot}/(RB) \)

\[
k \cdot \nabla \psi = \sigma_{B\theta} BR \left[ \left( \frac{B_{tot}^2}{RB^2} - \frac{d^2 R}{ds^2} \right) \frac{dZ}{ds} + \frac{d^2 Z}{ds^2} \frac{dR}{ds} \right]
\]

Taking the cross-product of the tangent vector with the curvature in equation (24.3) above gives

\[
b \times \kappa = \left[ \frac{B_{tot} d^2 Z}{B \frac{ds}{ds^2}} - \frac{dZ}{ds} \left( \frac{2 dR}{ds} \frac{d\phi}{ds} + R \frac{d^2 \phi}{ds^2} \right) \right] R
\]

\[
+ \left[ \frac{dR}{ds} \left( \frac{2 dR}{ds} \frac{d\phi}{ds} + R \frac{d^2 \phi}{ds^2} \right) - \frac{B_{tot}}{B} \left( \frac{d^2 R}{ds^2} - R \left( \frac{d\phi}{ds} \right)^2 \right) \right] Z
\]

\[
+ \left[ \frac{dZ}{ds} \left( \frac{d^2 R}{ds^2} - R \left( \frac{d\phi}{ds} \right)^2 \right) - \frac{dR}{ds} \frac{d^2 Z}{ds^2} \right] \hat{\phi}
\]

The components in field-aligned coordinates can then be calculated:

\[
(b \times \kappa)^T = \sigma_{B\theta} (b \times \kappa) \cdot \nabla \psi
\]

= \( \frac{RB_{pol}^2}{B} \left( \frac{dR}{ds} \frac{d\phi}{ds} + R \frac{d^2 \phi}{ds^2} \right) - RB_{tot} \left( \frac{dR}{ds} \frac{d^2 R}{ds^2} + \frac{dZ}{ds} \frac{d^2 Z}{ds^2} \right) + \frac{B_{tot}^3}{B^2} \frac{dR}{ds} \)

### 24.3.9 Curvature in toroidal coordinates

In toroidal coordinates \((\psi, \theta, \phi)\), the \( b \) vector is

\[
b = \frac{B_{pol}}{B} \hat{e}_\theta + \frac{B_{tor}}{B} \hat{e}_\phi = \frac{B_{pol} h_\theta}{B} \nabla \theta + \frac{RB_{tot}}{B} \nabla \phi
\]

The curl of this vector is

\[

abla \times b = \frac{1}{\sqrt{g}} \left( \frac{\partial b_\phi}{\partial \phi} - \frac{\partial b_\theta}{\partial \phi} \right)
\]

\[
 \nabla \times b = \frac{1}{\sqrt{g}} \left( \frac{\partial b_\psi}{\partial \psi} - \frac{\partial b_\phi}{\partial \psi} \right)
\]

\[
 \nabla \times b = \frac{1}{\sqrt{g}} \left( \frac{\partial b_\psi}{\partial \psi} - \frac{\partial b_\theta}{\partial \theta} \right)
\]
where \( 1/\sqrt{g} = B_{\text{pol}}/h_{\theta} \). Therefore, in terms of unit vectors:

\[
\nabla \times \mathbf{b} = \frac{1}{R h_{\theta}} \frac{\partial}{\partial \theta} \left( \frac{R B_{\text{tor}}}{B} \right) \hat{e}_\phi - B_{\text{pol}} \frac{\partial}{\partial \psi} \left( \frac{RB_{\text{tor}}}{B} \right) \hat{e}_\theta + \frac{B_{\text{pol}} R}{h_{\theta}} \frac{\partial}{\partial \psi} \left( h_{\theta} B_{\text{pol}} \right) \hat{e}_\phi
\]

### 24.3.10 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).

\[
B_{\text{pol}} = \frac{\left| \nabla \psi \right|}{R} = \frac{1}{R} \sqrt{ \left( \frac{\partial \psi}{\partial R} \right)^2 + \left( \frac{\partial \psi}{\partial R} \right)^2 }
\]

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

\[
\frac{\partial B_{\text{pol}}}{\partial \psi} = \nabla B_{\text{pol}} \cdot \nabla \psi / (RB_{\text{pol}})^2
\]

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 \( B = \sqrt{B_{\text{tor}}^2 + B_{\text{pol}}^2} \),

\[
\frac{\partial B}{\partial \psi} = \frac{1}{B} \left( B_{\text{tor}} \frac{\partial B_{\text{tor}}}{\partial \psi} + B_{\text{pol}} \frac{\partial B_{\text{pol}}}{\partial \psi} \right)
\]
24.3.11 Parallel derivative of the B field

To get the parallel nabla ients 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 \( R B_{\text{tor}} \) 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 (24.4), \( \nabla \psi \cdot \nabla \psi \) can also be written as

\[ \nabla \psi \cdot \nabla \psi = B^2 R^3 \left[ \left( \frac{\partial R}{\partial s} \right)^2 + \left( \frac{\partial Z}{\partial s} \right)^2 \right] \]

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

\[ \frac{\partial}{\partial s} (B_{\text{pol}}^2) = B^2 \frac{\partial}{\partial s} \left[ \left( \frac{\partial R}{\partial s} \right)^2 + \left( \frac{\partial Z}{\partial s} \right)^2 \right] + B_{\text{pol}}^2 \frac{\partial}{\partial s} (B^2) \]

Which gives

\[ \frac{\partial}{\partial s} (B^2) = - \frac{B_{\text{pol}}^2}{R^2} \frac{\partial}{\partial s} (R^2) + B^4 \frac{\partial}{\partial s} \left[ \left( \frac{\partial R}{\partial s} \right)^2 + \left( \frac{\partial Z}{\partial s} \right)^2 \right] \]

\[ \frac{\partial}{\partial s} (B_{\text{pol}}^2) = \left( 1 + \frac{B_{\text{pol}}^2}{B_{\text{tor}}^2} \right) B^2 \frac{\partial}{\partial s} \left[ \left( \frac{\partial R}{\partial s} \right)^2 + \left( \frac{\partial Z}{\partial s} \right)^2 \right] - \frac{B_{\text{pol}}^2}{R^2} \frac{\partial}{\partial s} (B^2) \]

24.3.12 Magnetic shear from \( J \times B \)

Re-arranging the radial force balance equation (24.2) gives

\[ \frac{B_{\text{pol}}^2 R}{B_{\text{tor}}} \frac{\partial v}{\partial \psi} + \nu \left( \frac{2 RB}{B_{\text{pol}}^2} \frac{\partial B}{\partial \psi} + B^2 \frac{\partial R}{\partial \psi} + B_{\text{pol}}^2 \frac{\partial B_{\text{tor}}}{\partial \psi} - B^2 R \frac{\partial B_{\text{tor}}}{\partial \psi} \right) + \mu_0 h_0 \frac{\partial P}{\partial \psi} = 0 \]
24.3.13 Magnetic shear

The field-line pitch is given by

\[ \nu = \frac{h_\theta B_{tor}}{B_{pol} R} \]

and so

\[ \frac{\partial \nu}{\partial \psi} = \frac{\nu}{h_\theta} \frac{\partial h_\theta}{\partial \psi} + \frac{\nu}{B_{tor}} \frac{\partial B_{tor}}{\partial \psi} - \frac{\nu}{B_{pol}} \frac{\partial B_{pol}}{\partial \psi} - \frac{\nu}{R} \frac{\partial R}{\partial \psi} \]

The last three terms are given in the previous section, but \( \frac{\partial h_\theta}{\partial \psi} \) needs to be evaluated

24.3.14 psi derivative of h

From the expression for curvature (equation (24.3)), and using \( \nabla x \cdot \nabla \psi = \sigma_{B\theta} (R B_{pol})^2 \) and \( \nabla z \cdot \nabla \psi = -\sigma_{B\theta} I (R B_{pol})^2 \)

\[ \kappa \cdot \nabla \psi = -\sigma_{B\theta} \frac{B_{pol}}{B_{h_\theta}} (R B_{pol})^2 \left[ \frac{\partial}{\partial x} \left( \frac{B h_\theta}{B_{pol}} \right) - \sigma_{B\theta} \frac{\partial}{\partial y} \left( \frac{B_{tor} R}{B} \right) \right] \]

\[ - I (R B_{pol})^2 \frac{B_{pol}}{B_{h_\theta}} \frac{\partial}{\partial y} \left( \frac{B_{tor} R}{B} \right) \]

The second and third terms partly cancel, and using \( \frac{\partial I}{\partial y} = \sigma_{B\theta} \frac{\partial \nu}{\partial x} \)

\[ \kappa \cdot \nabla \psi \left( \frac{R B_{pol}}{B h_\theta} \right) = -\sigma_{B\theta} \frac{B_{pol}}{B_{h_\theta}} \frac{\partial}{\partial x} \left( \frac{B h_\theta}{B_{pol}} \right) + \sigma_{B\theta} \frac{B_{pol}}{B_{h_\theta}} \frac{\partial B_{tor} R}{\partial \psi} \]

\[ = -\sigma_{B\theta} \frac{B_{pol}}{B_{h_\theta}} \left[ \frac{h_\theta}{B_{pol}} \frac{\partial}{\partial x} \left( \frac{B}{B_{pol}} \right) - \frac{h_\theta}{B_{pol}} \frac{\partial}{\partial x} \left( \frac{B_{tor} R}{B_{pol} R} \right) + \frac{B^2}{B_{pol}} \frac{\partial h_\theta}{\partial y} B_{pol} \frac{\partial}{\partial x} \left( \frac{B_{tor} R}{B_{pol} R} \right) \right] \]

\[ - \sigma_{B\theta} \frac{B_{pol}}{B_{pol}^2} \frac{\partial h_\theta}{\partial x} - \sigma_{B\theta} \frac{B_{pol}}{B_{pol}^2} \frac{\partial}{\partial x} \left( \frac{B_{tor} R}{B_{pol} R} \right) \]

Writing

\[ \frac{B \frac{\partial}{\partial x} \left( \frac{B}{B_{pol}} \right)}{B_{tor} R \frac{\partial}{\partial x} \left( \frac{B_{tor} R}{B_{pol} R} \right)} = \frac{\partial}{\partial x} \left( \frac{B^2}{B_{pol}} \right) - \frac{B}{B_{pol} R} \frac{\partial B}{\partial x} \]

\[ B_{tor} R \frac{\partial}{\partial x} \left( \frac{B_{tor} R}{B_{pol} R} \right) = \frac{\partial}{\partial x} \left( \frac{B_{tor}^2}{B_{pol} R} \right) - \frac{B_{tor}}{B_{pol} R} \frac{\partial}{\partial x} \left( B_{tor} R \right) \]

and using \( B \frac{\partial B}{\partial x} = B_{tor} \frac{\partial h_\theta}{\partial x} + B_{pol} \frac{\partial B_{pol}}{\partial x} \), this simplifies to give

\[ \frac{\kappa \cdot \nabla \psi}{(R B_{pol})^2} = -\sigma_{B\theta} \frac{B_{pol}^2}{B^2 h_\theta} \frac{\partial h_\theta}{\partial x} - \sigma_{B\theta} \frac{B_{tor}^2}{B^2 R} \frac{\partial R}{\partial x} \] (24.6)

This can be transformed into an expression for \( \frac{\partial h_\theta}{\partial x} \) involving only derivatives along field-lines. Writing \( \nabla R = \frac{\partial R}{\partial x} \nabla \psi + \frac{\partial R}{\partial \theta} \nabla \theta \),

\[ \nabla R \cdot \nabla \psi = \frac{\partial R}{\partial \psi} (R B_{pol})^2 \]
Using (24.4),

\[ \nabla \psi \cdot \nabla R = -\sigma_{B\theta} B R \frac{dZ}{ds} \]

and so

\[ \frac{\partial R}{\partial x} = -\frac{BR}{(RB_{pol})^2} \frac{dZ}{ds} \]

Substituting this and equation (24.5) for \( \kappa \cdot \nabla \psi \) into equation (24.6) the \( \frac{\partial R}{\partial x} \) term cancels with part of the \( \kappa \cdot \nabla \psi \) term, simplifying to

\[ \frac{\partial \theta}{\partial x} = -h_\theta \frac{B^3 R}{B_{pol} (RB_{pol})^2} \left[ \frac{d^2 Z}{ds^2} \frac{dR}{ds} - \frac{d^2 R}{ds^2} \frac{dZ}{ds} \right] \]

### 24.4 Shifted radial derivatives

The coordinate system given by equation (24.1) 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

\[ \frac{\partial}{\partial z \partial x} = \frac{\partial}{\partial z} \frac{\partial}{\partial \psi} + \frac{\partial I}{\partial z} \frac{\partial}{\partial \psi} + I \frac{\partial^2}{\partial z^2} \]

and second-order \( x \) derivative

\[ \frac{\partial^2}{\partial x^2} = \frac{\partial^2}{\partial \psi^2} + \frac{\partial}{\partial \psi} \left( I \frac{\partial}{\partial z} \right) + I \frac{\partial^2}{\partial \psi \partial z} + \frac{\partial I}{\partial \psi} \frac{\partial}{\partial z} \]

#### 24.4.1 Perpendicular Laplacian

\[ \nabla^2_\perp = (RB_{pol})^2 \left[ \frac{\partial^2}{\partial x^2} - 2I \frac{\partial^2}{\partial z \partial x} + \left( I^2 + \frac{B^2}{(RB_{pol})^4} \right) \frac{\partial^2}{\partial z^2} \right] \]

transforms to

\[ \nabla^2_\perp = (RB_{pol})^2 \left[ \frac{\partial^2}{\partial \psi^2} + \frac{\partial I}{\partial \psi} \frac{\partial}{\partial \psi} + \frac{B^2}{(RB_{pol})^4} \frac{\partial^2}{\partial z^2} \right] \]

(24.7)

The extra term involving \( I \) disappears, but only if both the \( x \) and \( z \) first derivatives are taken into account:

\[ \nabla^2_\perp = (RB_{pol})^2 \left[ \frac{\partial^2}{\partial x^2} - 2I \frac{\partial^2}{\partial z \partial x} + \left( I^2 + \frac{B^2}{(RB_{pol})^4} \right) \frac{\partial^2}{\partial z^2} \right] + \nabla^2_x \frac{\partial}{\partial x} + \nabla^2_z \frac{\partial}{\partial z} \]
with
\[\nabla^2 x = \frac{1}{J} \frac{\partial}{\partial x} \left[ J(R_{\text{pol}})^2 \right] = \frac{1}{J} \left[ -\frac{\partial}{\partial x} \left( J(R_{\text{pol}})^2 \right) - \frac{\partial}{\partial y} \left( \frac{B_{\text{tor}}}{B_{\text{pol}}^2 R} \right) \right],\]
\[\nabla^2 z = \frac{1}{J} \left[ -\frac{1}{\partial x} \left( J(R_{\text{pol}})^2 \right) - \frac{\partial}{\partial z} \left( \frac{B_{\text{tor}}}{B_{\text{pol}}^2 R} \right) \right],\]

where \( J = h_{\theta}/B_{\text{pol}} \) is the Jacobian. Transforming into \( \psi \) derivatives, the middle term of equation (24.8) cancels the \( I \) term in equation (24.7), but introduces another \( I \) term (first term in equation (24.8)). This term cancels with the \( \nabla^2 x \) term when \( \frac{\partial}{\partial x} \) is expanded, so the full expression for \( \nabla^2 \parallel \) using \( \psi \) derivatives is:

\[
\nabla^2 \parallel = (R_{\text{pol}})^2 \left[ \frac{\partial^2}{\partial \psi^2} + \frac{B^2}{(R_{\text{pol}})^4} \frac{\partial^2}{\partial z^2} \right] + \frac{1}{J} \frac{\partial}{\partial \psi} \left[ J(R_{\text{pol}})^2 \right] \frac{\partial}{\partial \psi} - \frac{1}{J} \frac{\partial}{\partial y} \left( \frac{B_{\text{tor}}}{B_{\text{pol}}^2 R} \right) \frac{\partial}{\partial z}
\]

In orthogonal (\( \psi, \theta, \zeta \)) flux coordinates

For comparison, the perpendicular Laplacian can be derived in orthogonal “flux” coordinates

\[|\nabla \psi| = R_{\text{pol}} \quad |\nabla \theta| = 1/h_{\theta} \quad |\nabla \zeta| = 1/R\]

The Laplacian operator is given by

\[
\nabla^2 A = (R_{\text{pol}})^2 \frac{\partial^2 A}{\partial \psi^2} + \frac{1}{h_{\theta}^2} \frac{\partial^2 A}{\partial \theta^2} + \frac{1}{R^2} \frac{\partial^2 A}{\partial \zeta^2}
\]

parallel derivative by

\[\partial \parallel \equiv b \cdot \nabla = \frac{B_{\text{pol}}}{B h_{\theta}} \frac{\partial}{\partial \theta} + \frac{B_{\text{tor}}}{R B} \frac{\partial}{\partial \zeta}\]

and so

\[
\partial^2 A = \partial \parallel \left( \partial \parallel A \right) = \left( \frac{B_{\text{pol}}}{B h_{\theta}} \right)^2 \frac{\partial^2 A}{\partial \theta^2} + \left( \frac{B_{\text{tor}}}{R B} \right)^2 \frac{\partial^2 A}{\partial \zeta^2}
\]

\[
+ 2 \frac{B_{\text{pol}} B_{\text{tor}}}{B^2 h_{\theta} R} \frac{\partial}{\partial \theta} \left( \frac{B_{\text{pol}}}{B h_{\theta}} \right) \frac{\partial}{\partial \theta} \left( \frac{B_{\text{tor}}}{R B} \right) \frac{\partial}{\partial \zeta}
\]

Hence in orthogonal flux coordinates, the perpendicular Laplacian is:

\[
\nabla^2 \equiv \nabla^2 - \partial^2 \parallel = (R_{\text{pol}})^2 \left[ \frac{\partial^2}{\partial \psi^2} + \frac{1}{R^4 B^2} \frac{\partial^2}{\partial \zeta^2} \right] + \frac{B_{\text{tor}}^2}{h_{\theta} R^2 B^2} \frac{\partial^2}{\partial \theta^2} + \cdots
\]

(24.8)

where the neglected terms are first-order derivatives. The coefficient for the second-order \( z \) derivative differs from equation (24.8), and equation (24.8) still contains a derivative in \( \theta \). This shows that the transformation made to get equation (24.8) doesn’t result in the same answer as orthogonal flux coordinates: equation (24.8) is in field-aligned coordinates.

Note that in the limit of \( B_{\text{pol}} = B \), both equations (24.8) and (24.8) are the same, as they should be.
24.4.2 Operator $B \times \nabla \Phi \cdot \nabla A$

\[
B \times \nabla \phi \cdot \nabla A = \left( \frac{\partial A}{\partial y} - \frac{\partial A}{\partial x} \right) \left( -B_{\text{tor}} \frac{RB_{\text{pol}}}{h_\theta} \right) \\
+ \left( \frac{\partial A}{\partial z} - \frac{\partial A}{\partial y} \right) (-B^2) \\
- \left( \frac{\partial A}{\partial y} - \frac{\partial A}{\partial z} \right) \left( IB_{\text{tor}} \frac{RB_{\text{pol}}}{h_\theta} \right)
\]

\[
B \times \nabla \phi \cdot \nabla A = \left( \frac{\partial A}{\partial \psi} + IB \frac{\partial A}{\partial \psi} \right) \left( -B_{\text{tor}} \frac{RB_{\text{pol}}}{h_\theta} \right) \\
+ \left( \frac{\partial A}{\partial z} - \frac{\partial A}{\partial \psi} \right) (-B^2) \\
- \left( \frac{\partial A}{\partial \psi} - \frac{\partial A}{\partial z} \right) \left( IB_{\text{tor}} \frac{RB_{\text{pol}}}{h_\theta} \right)
\]

24.5 Useful identities

24.5.1 $b \times \kappa \cdot \nabla \psi \simeq -RB_{\zeta} \partial_{||} \ln B$

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:

\[
b = B_\theta \frac{h_\theta}{B} \nabla \theta + B_\zeta \frac{R}{B} \nabla \zeta
\]

and using the definition of curl (equation (11)) we can write

\[
b \times \kappa \simeq \frac{B}{2} \nabla \times \frac{b}{B} = \frac{B}{2} B_\theta \left[ \frac{\partial}{\partial \theta} \left( \frac{B_\zeta R}{B^2} \right) - \frac{\partial}{\partial \zeta} \left( B_\theta h_\theta \frac{R}{B^2} \right) \right] e_\psi + [ ] e_\theta + [ ] e_\zeta
\]

so that when dotted with $\nabla \psi$, only the first bracket survives. The parallel gradient is

\[
\partial_{||} = b \cdot \nabla = \frac{B_\theta}{Bh_\theta} \frac{\partial}{\partial \theta} + \frac{B_\theta}{BR} \frac{\partial}{\partial \zeta}
\]

Neglecting derivatives for axisymmetric equilibrium

\[
\frac{B}{2} \nabla \times \frac{b}{B} \cdot \nabla \psi = \frac{B}{2} B \partial_{||} \left( \frac{B_\zeta R}{B^2} \right)
\]

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

\[
\begin{align*}
A \cdot a &= B \cdot b = C \cdot c = 1 \\
A \cdot b &= A \cdot c = B \cdot a = B \cdot c = C \cdot a = C \cdot b = 0
\end{align*}
\]

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:

\[
\nabla \Phi \text{ of a function } \Phi \text{ is defined so that } d\Phi = \nabla \Phi \cdot d\mathbf{R}
\]

From the chain rule, \(d\mathbf{R} = \frac{\partial \mathbf{R}}{\partial u^i} du^i = e_i du^i\) 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 \(\{e_i\}\) are reciprocal, any vector \(D\) can be written as \(D = D_i e^i = D^i e_i\) where \(D_i = D \cdot e_i\) are the covariant components and \(D^i = D \cdot e^i\) 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 \(g_{ii} = h_i^2\)

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

\[
\nabla^2 \phi = \frac{1}{J} \frac{\partial}{\partial u^i} \left( J g^{ij} \frac{\partial \phi}{\partial u^j} \right)
\]

which can be expanded as

\[
\nabla^2 \phi = g^{ij} \frac{\partial^2 \phi}{\partial u^i \partial u^j} + \frac{1}{J} \frac{\partial}{\partial u^i} \left( J g^{ij} \right) \frac{\partial \phi}{\partial u^j}
\]
where $G^j_i$ 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 $\nabla^2 u^k = G^k_i$. 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:

$$\nabla \times A = \frac{1}{\sqrt{g}} \sum_k \left( \frac{\partial A_j}{\partial u_i} - \frac{\partial A_i}{\partial u_j} \right) e_k \quad i, j, k \text{ cny } 1, 2, 3$$

Cross-product relation between contravariant and covariant vectors:

$$e^i = \frac{1}{J} (e_j \times e_k) \quad e_i = J \left( e^j \times e^k \right) \quad i, j, k \text{ cny } 1, 2, 3$$

### 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

$$B = \sqrt{B \cdot B} = \sqrt{J^{-1} e_y \cdot J^{-1} e_y} = \sqrt{J^{-2} g_{yy}} = J^{-1} \sqrt{g_{yy}}$$

Further on

$$B = BB$$

$$b = \frac{B}{B} = \frac{J^{-1} e_y}{J^{-1} \sqrt{g_{yy}}} = \frac{e_y}{\sqrt{g_{yy}}}$$

### 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

$$\nabla \parallel = (b \cdot \nabla) b = \frac{e_y e_y}{g_{yy}} \cdot \nabla = \frac{e_y e_y}{g_{yy}} e^i \partial_i = \frac{e_y}{g_{yy}} \partial_y$$
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 [haeseler])

\[
\nabla \cdot \mathbf{A} = \frac{1}{J} \partial_i \left( J A^i \right)
\]

(13)

and that

\[
A^i = \mathbf{A} \cdot e^i
\]

In our case \( \mathbf{A} \rightarrow \nabla \), so that

\[
\nabla^i = (\nabla) \cdot e^i = e^i \cdot (\nabla) = e^i \cdot (e^j \partial_j) = g^{ij} \partial_j
\]

Thus

\[
\nabla^2 = \frac{1}{J} \partial_i \left( J g^{ij} \partial_j \right) \\
= \frac{1}{J} g^{ij} J \partial_i \partial_j + \frac{1}{J} \partial_i \left( J g^{ij} \right) \partial_j \\
= g^{ij} \partial_i \partial_j + G^j \partial_j
\]

where we have defined\(^1\)

\[
G^j = \frac{1}{J} \partial_i \left( J g^{ij} \right) \\
= \frac{1}{J} \left( \partial_x \left[ J g^{xj} \right] + \partial_y \left[ J g^{yj} \right] + \partial_z \left[ J g^{zj} \right] \right)
\]

\(^1\) Notice that \( G^j \) 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. \(^1\) We find that \( \Gamma^i_{jk} = e^i \cdot \partial_k e^j = \nabla \cdot e^i \), whereas using equation (13) leads to \( G^i = e^i \cdot \partial_k e^j = \nabla \cdot e^i \), since \( g^{ij} = g^{ji} \) due to symmetry.
By writing the terms out, we get
\[ \nabla^2 = g^{ij} \partial_i \partial_j + G^i \partial_j \]
\[ = (g^{zx} \partial_x \partial_z + g^{zy} \partial_y \partial_z + g^{zz} \partial_z \partial_z) + (G^x \partial_z) \]
\[ + (g^{xy} \partial_x \partial_y + g^{yy} \partial_y^2 + g^{yx} \partial_y \partial_y) + (G^y \partial_y) \]
\[ + (g^{xz} \partial_x \partial_z + g^{yz} \partial_y \partial_z + g^{zz} \partial_z) + (G^z \partial_z) \]

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 commute for smooth functions \( \partial_i \partial_j = \partial_j \partial_i \). This gives
\[ \nabla^2 = (g^{xx} \partial_x^2) + (G^x \partial_x) \]
\[ + (g^{yy} \partial_y^2) + (G^y \partial_y) \]
\[ + (g^{zz} \partial_z^2) + (G^z \partial_z) \]
\[ + 2 (g^{xy} \partial_x \partial_y + g^{yz} \partial_y \partial_z + g^{zx} \partial_z) \]

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
\[ \nabla_i = (b \cdot \nabla) b = bb \cdot \nabla = \frac{e_y e_y}{g_{yy}} \cdot \nabla = \frac{e_y e_x}{g_{yy}} \cdot e^i \partial_i = \frac{e_y}{g_{yy}} \partial_y \]

we have that
\[ \nabla^i = \left( \frac{e_y}{g_{yy}} \partial_y \right) \cdot e^i = e^i \cdot \left( \frac{e_y}{g_{yy}} \partial_y \right) \]

so that by equation (13),
\[ \nabla^2 = \nabla \cdot (bb \cdot \nabla) \]
\[ = \nabla \cdot \left( \frac{e_y}{g_{yy}} \partial_y \right) \]
\[ = \frac{1}{J} \partial_i \left( Je^i \cdot \left[ \frac{e_y}{g_{yy}} \partial_y \right] \right) \]
\[ = \frac{1}{J} \partial_y \left( \frac{J e^i}{g_{yy}} \partial_y \right) \]
.2.4 The perpendicular Laplacian

For the perpendicular Laplacian, we have that

\[ \nabla^2_\perp = \nabla^2 - \nabla^2_\parallel \]

\[ = g_{ij} \partial_i \partial_j + G^j \partial_j - \frac{1}{J} \partial_y \left( \frac{J}{g_{yy}} \partial_y \right) \]

\[ = (g^{xx} \partial_x^2) + \left( \frac{1}{J} \left[ \partial_x \{ Jg^{xx} \} + \partial_y \{ Jg^{yx} \} + \partial_z \{ Jg^{zx} \} \right] \partial_x \right) \]

\[ + (g^{yy} \partial_y^2) + \left( \frac{1}{J} \left[ \partial_x \{ Jg^{yx} \} + \partial_y \{ Jg^{yy} \} + \partial_z \{ Jg^{zy} \} \right] \partial_y \right) \]

\[ + (g^{zz} \partial_z^2) + \left( \frac{1}{J} \left[ \partial_x \{ Jg^{zx} \} + \partial_y \{ Jg^{zy} \} + \partial_z \{ Jg^{zz} \} \right] \partial_z \right) \]

\[ + 2 \left( g^{xz} \partial_x \partial_y + g^{yz} \partial_y \partial_z + g^{zy} \partial_y \partial_z \right) \]

\[ - \frac{1}{J} \partial_y \left( \frac{J}{g_{yy}} \partial_y \right) \]

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

\[ \nabla^2_\perp \approx (g^{xx} \partial_x^2) + \left( \frac{1}{J} \left[ \partial_x \{ Jg^{xx} \} + \partial_y \{ Jg^{yx} \} + \partial_z \{ Jg^{zx} \} \right] \partial_x \right) \]

\[ + (g^{yy} \partial_y^2) + \left( \frac{1}{J} \left[ \partial_x \{ Jg^{yx} \} + \partial_y \{ Jg^{yy} \} + \partial_z \{ Jg^{zy} \} \right] \partial_y \right) \]

\[ + 2 (g^{xz} \partial_x \partial_y + g^{yz} \partial_y \partial_z + g^{zy} \partial_y \partial_z) \]

\[ - \frac{1}{J} \partial_y \left( \frac{J}{g_{yy}} \partial_y \right) \]

.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

\[
v_E = -\frac{\nabla \phi \times b}{B}
= -\frac{\nabla \phi \times e_y}{\sqrt{g_{yy}} J^{-1} \sqrt{g_{yy}}}
= \frac{J}{g_{yy}} e_y \times \nabla \phi
= \frac{J}{g_{yy}} e_y \times (e^x \partial_x + e^y \partial_y + e^z \partial_z) \phi
= \frac{J}{g_{yy}} (g_{yx} e^x + g_{yy} e^y + g_{yz} e^z) \times (e^x \partial_x + e^y \partial_y + e^z \partial_z) \phi
= \frac{J}{g_{yy}} (g_{yx} e^x \times e^z \partial_z + g_{yy} e^y \times e^x \partial_x + g_{yz} e^z \times e^y \partial_y
+ g_{yx} e^x \times e^y \partial_y + g_{yy} e^y \times e^x \partial_x + g_{yz} e^z \times e^y \partial_y
+ g_{yx} e^x \times e^z \partial_z + g_{yy} e^y \times e^z \partial_z + g_{yz} e^z \times e^x \partial_x) \phi
= \frac{1}{g_{yy}} (-g_{yy} e_z \partial_x + g_{yz} e_y \partial_x + g_{yx} e_z \partial_y - g_{yz} e_x \partial_y - g_{yx} e_y \partial_x + g_{yy} e_z \partial_z) \phi
\]

The electrostatic ExB advection

The electrostatic \( E \times B \) advection operator thus becomes

\[
v_E \cdot \nabla = -\frac{\nabla \phi \times b}{B} \cdot \nabla
= \frac{1}{g_{yy}} (-g_{yy} e_z \partial_x + g_{yz} e_y \partial_x + g_{yx} e_z \partial_y - g_{yz} e_x \partial_y - g_{yx} e_y \partial_x + g_{yy} e_z \partial_z) \phi \cdot (e^x \partial_x + e^y \partial_y + e^z \partial_z)
= \frac{1}{g_{yy}} (-g_{yy} \partial_x \phi \partial_z + g_{yz} \partial_x \phi \partial_y + g_{yx} \partial_y \phi \partial_z - g_{yz} \partial_y \phi \partial_x - g_{yx} \partial_z \phi \partial_y + g_{yy} \partial_z \phi \partial_x)
= \frac{1}{g_{yy}} ([g_{yy} \partial_z \phi - g_{yz} \partial_y \phi] \partial_z + [g_{yz} \partial_x \phi - g_{yx} \partial_x \phi] \partial_y + [g_{yx} \partial_y \phi - g_{yy} \partial_z \phi] \partial_z)
= \frac{1}{g_{yy}} (g_{yx} \{\phi, \cdot\}_y, z + g_{yy} \{\phi, \cdot\}_z, x + g_{yz} \{\phi, \cdot\}_x, y)
\]

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

\[ Bv_E \cdot \nabla = -\nabla \phi \times b \cdot \nabla \]

\[ = b \times \nabla \phi \cdot \nabla \]

\[ = \frac{\sqrt{g_{yy}}}{g_{yy}} (g_{yx} \{ \phi, \cdot \}_{y,z} + g_{yy} \{ \phi, \cdot \}_{z,x} + g_{yz} \{ \phi, \cdot \}_{x,y}) \]

\[ = \frac{1}{J \sqrt{g_{yy}}} (g_{yx} \{ \phi, \cdot \}_{y,z} + g_{yy} \{ \phi, \cdot \}_{z,x} + g_{yz} \{ \phi, \cdot \}_{x,y}) \]  

(14)

**The brackets operator in BOUT++**

Notice that the (phi,f)@ operators in BOUT++ returns \(-\nabla \phi \times b \cdot \nabla f\) rather than \(-\nabla \phi \times b \cdot \nabla f\).

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

\[ v_E \cdot \nabla = -\frac{\nabla \phi \times b}{B} \cdot \nabla \]

\[ \simeq \frac{1}{g_{yy}} (g_{yy} \{ \phi, \cdot \}_{z,x}) \]

\[ = \partial_z \phi \partial_x - \partial_x \phi \partial_z \]

**3 Divergence of ExB velocity**

\[ v_{ExB} = \frac{b \times \nabla \phi}{B} \]

Using

\[ \nabla \cdot (F \times G) = (\nabla \times F) \cdot G - F \cdot (\nabla \times G) \]

the divergence of the \(E \times B\) velocity can be written as

\[ \nabla \cdot \left( \frac{1}{B} b \times \nabla \phi \right) = \left[ \nabla \times \left( \frac{1}{B} b \right) \right] \cdot \nabla \phi - \frac{1}{B} b \cdot \nabla \times \nabla \phi \]

(15)

The second term on the right is identically zero (curl of a nablaient). The first term on the right can be expanded as

\[ \left[ \nabla \times \left( \frac{1}{B} b \right) \right] \cdot \nabla \phi = \left[ \nabla \left( \frac{1}{B} b \right) \times b + \frac{1}{B} \nabla \times b \right] \cdot \nabla \phi \]

Using

\[ b \times \kappa = \nabla \times b - b \left[ b \cdot \left( \nabla \times b \right) \right] \]

this becomes:

\[ \nabla \cdot \left( \frac{1}{B} b \times \nabla \phi \right) = -b \times \nabla \left( \frac{1}{B} b \right) \cdot \nabla \phi \]

\[ + \frac{1}{B} b \times \kappa \cdot \nabla \phi \]

\[ + \left[ b \cdot \left( \nabla \times b \right) \right] b \cdot \nabla \phi \]
Alternatively, equation (15) can be expanded as
\[
\nabla \cdot \left( \frac{1}{B} b \times \nabla \phi \right) = - B b \times \nabla \left( \frac{1}{B^2} \right) \cdot \nabla \phi + \frac{1}{B^2} \nabla \times B \cdot \nabla \phi
\]

\[
= - B b \times \nabla \left( \frac{1}{B^2} \right) \cdot \nabla \phi + \frac{1}{B^2} J \cdot \nabla \phi
\]

\[
\nabla \cdot \left( \frac{n}{B} b \times \nabla \phi \right) = \frac{1}{J} \frac{\partial}{\partial \psi} \left( J_n \frac{\partial \phi}{\partial z} \right) - \frac{1}{J} \frac{\partial}{\partial z} \left( J_n \frac{\partial \phi}{\partial \psi} \right)
\]

\[
+ \frac{1}{J} \frac{\partial}{\partial \psi} \left( J_n g_{\phi \psi} g_{\psi z} \frac{\partial \phi}{\partial y} \right) - \frac{1}{J} \frac{\partial}{\partial y} \left( J_n g_{\phi \psi} g_{\psi y} \frac{\partial \phi}{\partial \psi} \right)
\]
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

$$ \mathbb{J} = \frac{\partial \mathbf{F}}{\partial \mathbf{f}} $$

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 $\mathbb{J}$
2. Preconditioner multiply: Given a vector, multiply by an approximate inverse of $\mathbb{M} = \mathbb{I} - \gamma \mathbb{J}$
3. Calculate the stencils i.e. non-zero elements in $\mathbb{J}$
4. Calculate the non-zero elements of $\mathbb{J}$
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.

\[
\frac{\partial N_i}{\partial t} + v_E \cdot \nabla N_i = 0
\]

\[
\frac{\partial \omega}{\partial t} + v_E \cdot \nabla \omega = 2\omega_e \mathbf{b} \times \kappa \cdot \nabla P + N_i Z_e e \frac{4\pi V^2}{c^2} \nabla ||j||
\]

\[
\nabla^2 \omega / N_i = \phi
\]

\[
0.51 \nu_{ce} ||j|| = \frac{e}{m_e} \frac{\partial \phi}{\partial t} + \frac{T_e}{N_i m_e} \frac{\partial ||N_i||}{\partial t}
\]

A.2.2 Reduced 3-field MHD

This is a 3-field system of pressure \( P \), magnetic flux \( \psi \) and vorticity \( U \):

\[
f = \begin{pmatrix}
P \\
\psi \\
U
\end{pmatrix}
\]

\[
\frac{\partial \psi}{\partial t} = -\frac{1}{B_0} \nabla ||\phi||
\]

\[
= -\frac{1}{B_0} [\mathbf{b}_0 - (\mathbf{b}_0 \times \nabla \psi)] \cdot \nabla \phi
\]

\[
= -\frac{1}{B_0} \mathbf{b}_0 \cdot \nabla \phi - \frac{1}{B_0} (\mathbf{b}_0 \times \nabla \phi) \cdot \nabla \psi
\]

\[
\Rightarrow \frac{d\psi}{dt} = -\frac{1}{B_0} \mathbf{b}_0 \cdot \nabla \phi
\]

The coupled set of equations to be solved are therefore

\[
\frac{1}{B_0} \nabla^2 \psi = U
\]

\[
\left( \frac{\partial}{\partial t} + v_E \cdot \nabla \right) \psi = -\frac{1}{B_0} \mathbf{b}_0 \cdot \nabla \phi
\]

\[
\left( \frac{\partial}{\partial t} + v_E \cdot \nabla \right) P = 0
\]

\[
\left( \frac{\partial}{\partial t} + v_E \cdot \nabla \right) U = \frac{1}{\rho} B_0^2 [\mathbf{b}_0 - (\mathbf{b}_0 \times \nabla \psi)] \cdot \left( \frac{J_{||0}}{B_0} - \frac{1}{\mu_0} \nabla^2 \psi \right) + \frac{1}{\rho} \mathbf{b}_0 \times \kappa_0 \cdot \nabla P
\]

\[
v_E = \frac{1}{B_0} \mathbf{b}_0 \times \nabla \phi
\]
The Jacobian of this system is therefore:

\[
\begin{bmatrix}
-\mathbf{v}_E \cdot \nabla & 0 & [\mathbf{b}_0 \times \nabla (P_0 + P) \cdot \nabla] \nabla^{-2} \\\n0 & -\mathbf{v}_E \cdot \nabla & (\mathbf{b}_0 \cdot \nabla) \nabla^{-2} \\
2\mathbf{b}_0 \times \kappa_0 \cdot \nabla & -\frac{B_0^2}{\mu_0 \rho} (\mathbf{b}_0 - \mathbf{b}_0 \times \nabla \psi) \cdot \nabla \nabla^2 \\
& + \frac{B_0^2}{\rho} \left[ \mathbf{b}_0 \times \nabla \left( \frac{J_{||}}{B_0^2} \right) \right] \cdot \nabla \\
& + \frac{B_0^2}{\mu_0 \rho} \nabla (\nabla^2 \psi) \cdot (\mathbf{b}_0 \times \nabla)
\end{bmatrix}
\]

(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 \( \nabla^{-2} \) which couples together all points in an X-Z plane. The way to make \( \mathbf{J} \) 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:

\[
\frac{\partial N_i}{\partial t} + V_{||} \frac{\partial}{\partial t} N_i = -N_i \nabla_{||} V_{||} + \nabla_{\psi} (D_{\perp} \partial_{\psi} N_i)
\]

\[
\frac{\partial}{\partial t} \left( N_i V_{||} \right) + V_{||} \frac{\partial}{\partial t} (N_i V_{||}) = -\partial_{||} P + \nabla_{\psi} (N_i \mu_{\perp} \partial_{\psi} V_{||})
\]

\[
\frac{3}{2} \frac{\partial}{\partial t} (N_i T_e) = \nabla_{||} (\kappa_e \partial_{||} T_e) + \nabla_{\psi} (N_i \chi_{\perp} \partial_{\psi} T_e)
\]

\[
\frac{3}{2} \frac{\partial}{\partial t} (N_i T_i) = \nabla_{||} (\kappa_i \partial_{||} T_i) + \nabla_{\psi} (N_i \chi_{\perp} \partial_{\psi} T_i)
\]

This set of equations is good in that there is no inversion needed, and so the Jacobian is sparse everywhere. The state vector is

\[
f = \begin{pmatrix} N_i \\ V_{||} \\ T_e \\ T_i \end{pmatrix}
\]

The Jacobian is:

\[
\mathbf{J} = \begin{pmatrix}
-V_{||} \frac{\partial}{\partial t} - \nabla_{||} V_{||} + \nabla_{\psi} D_{\perp} \partial_{\psi} & -\partial_{||} N_i - N_i \nabla_{||} \\
0 & 0 & 0 \\
-\frac{1}{N_i} \frac{\partial V_{||}}{\partial t} - \frac{1}{N_i} V_{||} \nabla N_i, N_i & 0 \\
\end{pmatrix}
\]

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.2.5 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

$$
M = \begin{bmatrix}
D & 0 & U_P \\
0 & D & U_\psi \\
L_P & L_\psi & D
\end{bmatrix}
$$

where

$$
D = I + \gamma v_E \cdot \nabla
$$

For small flow velocities, the inverse of $D$ can be approximated using the Binomial theorem:

$$
D^{-1} \approx I - \gamma v_E \cdot \nabla (1.2)
$$

Following [chacon-2008], [chacon-2002], $M$ can be re-written as

$$
M = \begin{bmatrix}
E & U \\
L & D
\end{bmatrix} = \begin{bmatrix}
D & 0 \\
0 & D
\end{bmatrix} = \begin{bmatrix}
U_P \\
U_\psi
\end{bmatrix} = \begin{bmatrix}
L_P & L_\psi
\end{bmatrix}
$$

The Schur factorization of $M$ yields ([chacon-2008])

$$
M^{-1} = \begin{bmatrix}
E & U \\
L & D
\end{bmatrix}^{-1} = \begin{bmatrix}
I & -E^{-1}U \\
0 & I
\end{bmatrix} = \begin{bmatrix}
E^{-1} & 0 \\
0 & P_Schur^{-1}
\end{bmatrix} = \begin{bmatrix}
I & 0 \\
0 & -LE^{-1}
\end{bmatrix}
$$

Where $P_{Schur} = D - LE^{-1}U$ 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 $P_{Schur}$, which is much smaller than $M$.

A possible approximation to $P_{Schur}$ 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

$$
P_{Schur} \approx I + \gamma^2 \frac{B_0^2}{\mu_0 \rho} \left( b_0 \cdot \nabla \right) \nabla^2_\perp \left( b_0 \cdot \nabla \right) \nabla^{-2}_\perp
$$

$$
\approx I + \gamma^2 V_A^2 \left( b_0 \cdot \nabla \right)^2
$$

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.
B.1 Geometry

In an axisymmetric toroidal system, the magnetic field can be expressed as

\[ B = I(\psi) \nabla \zeta + \nabla \zeta \times \nabla \psi, \]

where \( \psi \) is the poloidal flux, \( \theta \) is the poloidal angle-like coordinate, and \( \zeta \) is the toroidal angle. Here, \( I(\psi) = RB_t \).

The two important geometrical parameters are: the curvature, \( \kappa \), and the local pitch, \( \nu(\psi, \theta) \),

\[ \nu(\psi, \theta) = I(\psi)\mathcal{J}/R^2. \]

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 \( \mathcal{J} = (\nabla \psi \times \nabla \theta \cdot \nabla \zeta)^{-1} \) is the coordinate Jacobian, \( R \) is the major radius, and \( Z \) is the vertical position.

B.2 Geometry and Differential Operators

In an axisymmetric toroidal system, the magnetic field can be expressed as \( B = I(\psi) \nabla \zeta + \nabla \zeta \times \nabla \psi, \) where \( \psi \) is the poloidal flux, \( \theta \) is the poloidal angle-like coordinate, and \( \zeta \) is the toroidal angle. Here, \( I(\psi) = RB_t \). The two important geometrical parameters are: the curvature, \( \kappa \), and the local pitch, \( \nu(\psi, \theta) \), and \( \nu(\psi, \theta) = I(\psi)\mathcal{J}/R^2 \). 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 \( \mathcal{J} = (\nabla \psi \times \nabla \theta \cdot \nabla \zeta)^{-1} \) 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: $\nabla_E \cdot \nabla_\perp$, $\nabla_\parallel$ and $\nabla_\perp^2$.

$$\nabla_\parallel = b_0 \cdot \nabla = \frac{1}{\mathcal{J}B} \frac{\partial}{\partial \psi} + \frac{I}{BR^2} \frac{\partial}{\partial \varphi} = \frac{B_p}{hB} \frac{\partial}{\partial \varphi} + \frac{B_t}{RB} \frac{\partial}{\partial \zeta},$$

$$\mathcal{J} \nabla^2 = \mathcal{J} \nabla_1 \nabla_2 = \frac{\partial}{\partial \psi} \left( \mathcal{J} J_{11} \frac{\partial}{\partial \psi} + \mathcal{J} J_{12} \frac{\partial}{\partial \varphi} \right) + \frac{\partial}{\partial \varphi} \left( \mathcal{J} J_{22} \frac{\partial}{\partial \varphi} + \mathcal{J} J_{12} \frac{\partial}{\partial \psi} \right) + \frac{1}{R^2} \frac{\partial^2}{\partial \zeta^2},$$

$$\nabla_\perp^2 = b_0 \cdot \nabla (b_0 \cdot \nabla) = \frac{1}{\mathcal{J}B} \frac{\partial}{\partial \psi} \left( \frac{1}{\mathcal{J}B} \frac{\partial}{\partial \varphi} \right) = \frac{1}{\mathcal{J}B} \frac{\partial}{\partial \varphi} \left( \frac{B_t}{RB} \frac{\partial}{\partial \zeta} \right) + \frac{B_t}{\mathcal{J}R^2B^2} \frac{\partial^2}{\partial \zeta^2} \left( \mathcal{J} \nabla_1 \nabla_2 \right) + \frac{B_t}{R} \frac{\partial^2}{\partial \zeta^2},$$

$$\nabla_\perp^2 \Phi = -\nabla \cdot |b \times (b \times \nabla \Phi)| = \nabla^2 \Phi - (\nabla \cdot b)(b \cdot \nabla \Phi) - \nabla_\parallel^2 \Phi$$

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:

\[ R = R_0 + r \cos \theta, \]
\[ Z = r \sin \theta, \]
\[ B_t = \frac{B_{t0} R_0}{R}, \]
\[ B_p = \frac{1}{R} \frac{\partial \psi}{\partial r}, \]
\[ R_\psi = \frac{\cos \theta}{R B_p}, \]
\[ R_\theta = -r \sin \theta, \]
\[ Z_\psi = \frac{\sin \theta}{R B_p}, \]
\[ Z_\theta = r \cos \theta, \]
\[ J = \frac{r}{B_p}, \]
\[ h = r, \]
\[ J_{11} = |\nabla \psi|^2 = r^2 B_p^2, \]
\[ J_{12} = J_{21} = \nabla \psi \cdot \nabla \theta = 0, \]
\[ J_{13} = J_{31} = 0, \]
\[ J_{22} = |\nabla \theta|^2 = \frac{1}{r^2}, \]
\[ J_{23} = J_{32} = 0, \]
\[ J_{33} = |\nabla \zeta|^2 = \frac{1}{R^2}, \]
\[ \nabla^2 \approx \frac{1}{r} \frac{\partial}{\partial r} \left( r \frac{\partial}{\partial r} \right) + \frac{1}{r^2} \frac{\partial^2}{\partial \theta^2} + \frac{1}{R^2} \frac{\partial^2}{\partial \zeta^2} \]

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

\[ x = \psi - \psi_s, \]
\[ y = \theta, \]
\[ z = \zeta - \int_{\theta_0}^{\theta} \nu(x, y) dy. \]

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:

\[ \mathcal{J} = \nabla \psi \times \nabla \theta \cdot \nabla \zeta = \frac{h}{B_p}, \]

\[ h = \sqrt{Z_\theta^2 + R_\theta^2}, \]

\[ \mathcal{J}_{11} = |\nabla x|^2 = \frac{R^2}{\mathcal{J}^2} (Z_\psi^2 + R_\psi^2), \]

\[ \mathcal{J}_{12} = \mathcal{J}_{21} = \nabla x \cdot \nabla y = -\frac{R^2}{\mathcal{J}^2} (Z_\psi Z_\theta + R_\psi R_\theta), \]

\[ \mathcal{J}_{22} = |\nabla y|^2 = \frac{R^2}{\mathcal{J}^2} (Z_\psi^2 + R_\psi^2), \]

\[ \mathcal{J}_{13} = \mathcal{J}_{31} = \nabla x \cdot \nabla z = -\nu \nabla x \cdot \nabla y - |\nabla x|^2 \left( \int_{y_0}^y \frac{\partial \mu(x,y)}{\partial \psi} dy \right) = -|\nabla x|^2 I_s, \]

\[ \mathcal{J}_{23} = \mathcal{J}_{32} = \nabla y \cdot \nabla z = -\nu |\nabla y|^2 - \nu \nabla x \cdot \nabla y \left( \int_{y_0}^y \frac{\partial \mu(x,y)}{\partial \psi} dy \right), \]

\[ \mathcal{J}_{33} = |\nabla z|^2 = \left| \nabla \zeta - \nu \nabla \theta - \nabla \psi \left( \int_{y_0}^y \frac{\partial \mu(x,y)}{\partial \psi} dy \right) \right|^2, \]

\[ I_s = \frac{|\nabla z|^2}{|\nabla \psi|^2} \nu(x,y) + \left( \int_{y_0}^y \frac{\partial \mu(x,y)}{\partial \psi} dy \right). \]

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:

\[ \frac{d}{d\psi} = \frac{\partial}{\partial x} - \left( \int_{y_0}^y \frac{\partial \mu(x,y)}{\partial \psi} dy \right) \frac{\partial}{\partial z}, \]

\[ \frac{d}{d\theta} = \frac{\partial}{\partial y} - \nu(x,y) \frac{\partial}{\partial z}, \]

\[ \frac{d}{d\zeta} = \frac{\partial}{\partial z}. \]

In the field-aligned ballooning coordinates, the parallel differential operator is simple, involving only one coordinate \( y \)

\[ \partial_0 = b_0 \cdot \nabla_\parallel = \left( \frac{B_p}{hB} \right) \frac{\partial}{\partial y}. \]

which requires a few grid points. The total axisymmetric drift operator becomes

The perturbed \( \mathbf{E} \times \mathbf{B} \) drift operator becomes

\[ \delta \mathbf{v}_E \cdot \nabla_\perp = \frac{c}{BB_\parallel} \left\{ \frac{I}{J} \frac{\partial (\delta \phi)}{\partial \theta} + B_p^2 \frac{\partial (\delta \phi)}{\partial z} \right\} \frac{\partial}{\partial \psi} \]

\[ + \frac{c}{BB_\parallel} \left\{ \frac{I}{J} \frac{\partial (\delta \phi)}{\partial \psi} + \frac{\mathcal{J}_{12}}{R^2} \frac{\partial (\delta \phi)}{\partial z} \right\} \frac{\partial}{\partial \theta}, \]

\[ - \frac{c}{BB_\parallel} \left\{ B_p^2 \frac{\partial (\delta \phi)}{\partial \psi} + \frac{\mathcal{J}_{12}}{R^2} \frac{\partial (\delta \phi)}{\partial \theta} \right\} \frac{\partial}{\partial z}, \]

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

\[ \delta \mathbf{v}_E \cdot \nabla_\perp = \frac{cB}{BB_\parallel} \left( \frac{\partial (\delta \phi)}{\partial z} \frac{\partial}{\partial x} - \frac{\partial (\delta \phi)}{\partial x} \frac{\partial}{\partial z} \right). \]
where \( \partial/\partial \theta \approx -\nu \partial/\partial z \) 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

\[
\mathcal{J} \nabla^2_t \Phi = \frac{\partial}{\partial x} \left( \mathcal{J} \mathcal{J}_{11} \frac{\partial \Phi}{\partial x} + \mathcal{J} \mathcal{J}_{12} \frac{\partial \Phi}{\partial y} + \mathcal{J} \mathcal{J}_{13} \frac{\partial \Phi}{\partial z} \right) \\
+ \frac{\partial}{\partial y} \left( \mathcal{J} \mathcal{J}_{21} \frac{\partial \Phi}{\partial x} + \mathcal{J} \mathcal{J}_{22} \frac{\partial \Phi}{\partial y} + \mathcal{J} \mathcal{J}_{23} \frac{\partial \Phi}{\partial z} \right) \\
+ \frac{\partial}{\partial z} \left( \mathcal{J} \mathcal{J}_{31} \frac{\partial \Phi}{\partial x} + \mathcal{J} \mathcal{J}_{32} \frac{\partial \Phi}{\partial y} + \mathcal{J} \mathcal{J}_{33} \frac{\partial \Phi}{\partial z} \right) \\
- \frac{B_p}{hB} \frac{\partial}{\partial y} \left[ \left( \frac{B_p}{hB} \right) \frac{\partial \Phi}{\partial y} \right] \\
- \left( \frac{B_p}{hB} \right)^2 \frac{\partial \ln B}{\partial y} \frac{\partial \Phi}{\partial y}.
\]

The general perpendicular Laplacian operator for axisymmetric potential \( \Phi_0(x, y) \) is

\[
\mathcal{J} \nabla^2_t \Phi_0 = \frac{\partial}{\partial x} \left( \mathcal{J} \mathcal{J}_{11} \frac{\partial \Phi_0}{\partial x} + \mathcal{J} \mathcal{J}_{12} \frac{\partial \Phi_0}{\partial y} \right) \\
+ \frac{\partial}{\partial y} \left( \mathcal{J} \mathcal{J}_{21} \frac{\partial \Phi_0}{\partial x} + \mathcal{J} \mathcal{J}_{22} \frac{\partial \Phi_0}{\partial y} \right) \\
- \frac{B_p}{hB} \frac{\partial}{\partial y} \left[ \left( \frac{B_p}{hB} \right) \frac{\partial \Phi_0}{\partial y} \right] \\
- \left( \frac{B_p}{hB} \right)^2 \frac{\partial \ln B}{\partial y} \frac{\partial \Phi_0}{\partial y}.
\]

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

\[
\mathcal{J} \nabla^2_t \delta \phi = \frac{\partial}{\partial x} \left( \mathcal{J} \mathcal{J}_{11} \frac{\partial \delta \phi}{\partial x} + \mathcal{J} \mathcal{J}_{13} \frac{\partial \delta \phi}{\partial z} \right) \\
+ \frac{\partial}{\partial z} \left( \mathcal{J} \mathcal{J}_{31} \frac{\partial \delta \phi}{\partial x} + \mathcal{J} \mathcal{J}_{33} \frac{\partial \delta \phi}{\partial z} \right). 
\]

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):

```
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”.

```
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:

```
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:

```
git remote add yourfork https://github.com/YourUsername/BOUT-dev.git
```

then they will be able to checkout your branch:

```
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:

```
... make changes ...
git add <files>
git commit
git push  # Pushes to origin/myfeature
```

You can switch between branches using `checkout`:

```
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 ///.

Doxygen done right:

```cpp
/// Foo the bar
///
/// Apply the standard foo method to @p bar
///
/// Typical usage:
///
/// foo(bar, "simple", result);
///
/// @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();
}
```
if (expr) {
    doSomething();
} else {
    doOtherThing();
}

This especially helps readability, making conditional statements stand out over function calls.

- Braces on same line as statement:

  Wrong:

```c
void doFoo(bool expr) {
    if (expr) {
        doSomething();
    } else {
        doOtherThing();
    }
}
```

  Right:

```c
void 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.
Git crash course

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:

```
[user]
    name = A. Developer
    email = a.developer@example.com

[alias]
    st = status
    ci = commit
    br = branch
    co = checkout
    df = diff
    lg = log -p
    who = shortlog -s --
```

(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:

```
$ 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:

```
$ 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:

$$
\text{ssh username@franklin.nersc.gov}
$$

2. Create a “bare” Git repository by cloning a repository with the `-bare` option:

$$
\text{cd ~}
\text{git clone --bare git@github.com:boutproject/BOUT-dev.git bout_private}
$$

where you can replace `git@github.com:boutproject/BOUT-dev.git` with any other repository you can access. `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:

$$
\text{git clone bout_private bout}
$$

which creates a repository `bout` from your private repository. Running `git pull` and `git push` from within this new repository will exchange patches with your `bout_private` repository.

3. You can now clone, pull and push changes to your private repository over SSH e.g.:

$$
\text{git clone username@franklin.nersc.gov:bout_private}
$$

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:

$$
\text{cd ~}
\text{git clone bout_private bout_tmp}
$$

This creates a repository `bout_tmp` from your private repository. Now cd to the new directory and pull the latest changes from github:

$$
\text{cd bout_tmp}
\text{git pull git://github.com/boutproject/BOUT-dev.git}
$$

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, git will tell you, and running:

$$
\text{git status}
$$

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:

$$
\text{git commit -a}
$$

6. Your `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:

$$
\text{git push}
$$

You can now delete the `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.

Fig. 5.1: Overview of BOUT++ control flow during initialisation (red), and running (blue)
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.

![Diagram of class relationships](image)

Fig. 5.2: Relationship between important classes and functions used in calculating the RHS function

### 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`. 

---

244 Appendix E. Code layout
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`).

- **vector2d.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.

- **vector3d.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.hxx** 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
    · **pnetcdf.hxx** 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.
  – 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.
  – impls
    * cyclic
      · **cyclic.hxx**
      · **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

  – **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 Differential operators.
  – 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
  – 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 outlined 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 setMark() 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 $\sim 100$).

To handle time-derivatives, and enable expressions to be written in the following form:

$$\frac{d}{dt}(N_i) = -b_0 x \text{Grad} \cdot \text{Grad} (\phi, N_i);$$

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.

```cpp
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 setLocation(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:

$$\text{ddt}(v.x) = \text{ddt}(v).x$$

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
            x,y,z,
            g3d(x,y,z) ");
```

or by getting a pointer to the underlying data, which is stored as a 1D array:

```cpp
BoutReal *data = g3d.getData();
x = g3d.xSize();
y = g3d.ySize();
z = 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:

```cpp
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:

```cpp
Field3D f(0.0);
for
    for
        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:
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:

\[
\text{BOUT_OMP(parallel for schedule(guided))}
\]

\[
\text{for (auto block = region.getBlocks().cbegin();}
\]

\[
\text{block < region.getBlocks().cend(); ++block}
\]

\[
\text{for (auto index = block->first; index < block->second; ++index)}
\]

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 \text{mesh:maxregionblocksize} option on the command line or in the BOUT.inp input file:

\[
\text{[mesh]}
\]

\[
\text{maxregionblocksize = 64}
\]

The default value is set in include/bout/region.hxx:

\[
\text{#define MAXREGIONBLOCKSIZE 64}
\]

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:

\[
\text{auto region = mesh->getRegion2D("RGN_NOBNDRY") + mesh->getRegion2D("RGN_BNDRY");}
\]

This new region could contain duplicated indices, so if unique points are required then the unique function can be used:

\[
\text{auto region = unique(mesh->getRegion2D("RGN_NOBNDRY") + mesh->getRegion2D("RGN_BNDRY"));}
\]

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):

F.7. Iterating over fields
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.hxx provides this.

`RangeIterator` can represent a single continuous range, constructed by passing the minimum and maximum values.

```cpp
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:

```cpp
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:

```cpp
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:

```cpp
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 Portable, Extensible Toolkit for Scientific Computation (PETSc) provides a large collection of numerical routines and solvers. It is used in BOUT++ for Time integration and Laplacian inversion. However, it provides quite a low-level C interface which is often difficult to use and bears little resemblance to the data model of BOUT++. This is particularly the case when making use of the Mat and Vec data-types for linear solvers (such as the Laplacian inversions). Doing so requires the developer to:

1. Flatten a Field into a 1-D PETSc Vec object.
   - Must decide which guard cells to include in Vec
   - Must convert between BOUT++ indices (local) and indices used in PETSc Vec (global)
2. Use a PETSc Mat object to represent a finite-difference operator.
   - Again, must convert between local and global indices
   - Must determine sparsity pattern of matrix
   - If taking field-aligned derivatives, must perform interpolation
3. Call a Krylov solver with a preconditioner.
4. Convert the resulting Vec object back into a Field.

These tasks are error-prone and have historically been re-implemented each time a PETSc solver is to be used. An interface to PETSc has now been added which greatly simplifies its use. Furthermore, it has been thoroughly unit-tested which should ensure improved reliability.

**G.1 Overall Structure**

Based on the finite difference operator being inverted, a user constructs an OperatorStencil object which is used to describe the interdependencies between cells in the grid. This is used when constructing a GlobalIndexer to work out which cells should be included in a Vec object. If needed, it will also be used to work out the sparsity pattern of a matrix. PetscVector and PetscMatrix objects are constructed from a GlobalIndexer and wrap the PETSc Vec and Mat objects, respectively. They provide routines for accessing individual elements of the...
Fig. 7.1: A UML diagram showing the design of the PETSc interface and the relationships between different components.
vector/matrix using PetscVector::Element and PetscMatrix::Element objects. All of these classes are templates. OperatorStencil works for SpecificInd types (Ind3D, Ind2D, IndPerp), while the remaining classes work for the various Field types.

**G.2 OperatorStencil**

This data type describes the structure of the finite difference operator which is to be inverted. It uses a number of small helper-types to do this (see Fig. 7.2), the most important of which is the IndexOffset. This is a simple structure which represents an offset from a SpecificInd<> type. It can be added to or subtracted from SpecificInd<> objects, returning an index which is suitably offset.

Vectors of IndexOffset objects are coupled with tests which take a SpecificInd<> as an argument and return a boolean result indicating whether the offsets describe the finite difference stencil at that location. The OperatorStencil class contains a vector of these pairs. When the stencil is requested for a given index, the vector gets traversed in order, with offsets returned from the first passing test. Pairs of tests and offsets are placed in the OperatorStencil object using the add() method. It will generally not be necessary for you to use any of the other methods on this class unless you are doing further development work on the PETSc interface.

Consider a 2-D Laplacian operator with the discretisation
\[
\nabla^2 f(i,j) = \frac{f(i+1,j) - 2f(i,j) + f(i-1,j)}{\Delta x^2} + \frac{f(i,j+1) - 2f(i,j) + f(i,j-1)}{\Delta y^2}
\]
and Neumann boundary conditions. Then an appropriate stencil could be created using the following code.

```cpp
OperatorStencil<Ind2D> stencil;
OffsetInd2D zero;

// Add Laplace stencil for interior grid points
stencil.add([&xs = localmesh->xstart, xe = localmesh->xend, ys = localmesh->ystart, ye = localmesh->yend](Ind2D ind) -> bool {
  return ind.x() >= xs && ind.x() <= xe && ind.y() >= ys && ind.y() <= ye;
}, {zero, zero.xp(), zero.xm(), zero.yp(), zero.ym()});

// Add first-order differences for Neumann boundaries
// Inner X
stencil.add([&xs = localmesh->xstart](Ind2D ind) -> bool {
  return ind.x() < xs; }, {zero, zero.xp()});

// Outer X
stencil.add([&xe = localmesh->xend](Ind2D ind) -> bool {
  return ind.x() > xe; }, {zero, zero.xm()});

// Lower Y
stencil.add([&ys = localmesh->ystart](Ind2D ind) -> bool {
  return ind.y() < ys; }, {zero, zero.yp()});

// Upper Y
stencil.add([&ye = localmesh->yend](Ind2D ind) -> bool {
  return ind.y() > ye; }, {zero, zero.ym()});
```

**G.3 GlobalIndexer**

Using an OperatorStencil, the GlobalIndexer constructor can determine which cells should be included in the PETSc Vec object representing a Field. All interior cells are always included. Boundary cells which are required by the stencil to compute the operation on internal cells are also included. A globally-unique index is assigned to each cell which is meant to be included and the communication routines on the Mesh type are used to determine the
Fig. 7.2: UML diagram describing the structure of the OperatorStencil class.
indices in guard cells. There must be a unique `GlobalIndexer` object for each `Mesh` and `OperatorStencil` pair. You will need to pass a `std::shared_ptr<GlobalIndexer>` when constructing `PetscVector` and `PetscMatrix` objects. As the process of creating a `GlobalIndexer` is quite expensive and each one contains a field of indices, you will not want to create any copies (hence the use of `std::shared_ptr`).

In comparison to initialising an `OperatorStencil` object, creating a `GlobalIndexer` is quite simple. The constructor takes 3 arguments, two of which are optional:

- A pointer to the `Mesh` object for the indexer
- An `OperatorStencil`; if absent then the indexer will not include any guard cells in the PETSc objects and will not compute matrix sparsity patterns
- A boolean specifying whether communication of indices in guard cells will be performed in the constructor; defaults to `true`, otherwise will need to call the `initialise()` method prior to use (you would generally only do that if creating a fake indexer for testing purposes)

Continuing on from the previous example, the code below shows how to create a `GlobalIndexer`.

```cpp
IndexerPtr<Field2D> indexer = 
    std::make_shared<GlobalIndexer<Field2D>>(localmesh, stencil);
```

The `GlobalIndexer` class provides `Region<>` objects which can be used for iterating over the cells which are included in PETSc `Vec` objects (see `Iterating over fields`). This is useful for setting vector and matrix elements. The relevant methods are:

- `getRegionAll()` returns a region containing all cells included in the PETSc objects
- `getRegionNobndry()` contains only the non-guard cells include in the PETSc objects (identical to `RGN_NOBNDRY`)
- `getRegionBndry()` contains only guard cells which are also boundary cells
- `getRegionLowerY()` contains only guard cells in the lower Y-boundary
- `getRegionUpperY()` contains only guard cells in the upper Y-boundary
- `getRegionInnerX()` contains only guard cells in the inner X-boundary
- `getRegionOuterX()` contains only guard cells in the outer X-boundary

Note that not all guard-cells will be boundary cells; most will just be used for communication between processors.

## G.4 PetscVector

This class wraps PETSc `Vec` objects, split across multiple processors. The constructors/destructors ensure memory will be allocated/freed as necessary. To create a new vector, pass a `Field` and `IndexerPtr` to the constructor. This will create a `Vec` object which is split between processors. The `IndexerPtr` will be used to convert between the local `BOUT++` coordinate system and the global PETSc indices used to access elements of the `Vec` object. The values in the `Field` will be copied into the `Vec`. The user can set individual elements using local `BOUT++` indices and the parentheses operator (`[]`). Once this is done, call the `assemble()` method. Elements can be set using either assignment (`=`) or in-place addition (`+=`). However, as in PETSc itself, these operations can not be mixed, unless there is call to `assemble()` in between. A `PetscVector` can be converted back to a `Field` object using the `toField()` method.

Below is an example of creating a vector which could be used as input for a linear solver.
Field2D rhs_vals; // Assume this is initialised with some data
PetscVector<Field2D> rhs_vec(rhs_vals, indexer);

// Set boundary values to 0
BOUT_FOR(i, indexer.getRegionBndry()) {
    rhs_vec(i) = 0.0;
}
rhs_vec.assemble();

If you plan to do any development of the PETSc interface (or simply wish to understand how it works), see the UML sequence diagram in Fig. 7.3 for a description of how vector elements are set.

Fig. 7.3: A UML sequence diagram showing what happens when setting an element of a PetscVector. The GlobalIndexer is used to convert from the BOUT++ index to the one used by PETSc. A placeholder PetscVector::Element object is returned containing the index and a pointer to the Vec object. The assignment operator on this class makes a call to the PETSc routine VecSetValue.

G.5 PetscMatrix

This class wraps a PETSc Mat object, including managing memory in its constructors and destructor. This is a sparse matrix using the AIJ storage method. It is split across multiple processors. The PetscMatrix object is constructed from a IndexerPtr object; unlike for a PetscVector it would not make sense to copy data from a Field into a PetscMatrix object in the constructor. If the GlobalIndexer has this data available, the sparsity pattern of the Mat object will be passed to PETSc. This allows memory to be pre-allocated for it by PETSc, which dramatically improved performance.

As with PetscVector objects, individual elements of a PetscMatrix can be accessed using BOUT++ indices and the parentheses operator, except that now two indices are required (corresponding to the row and column of the matrix). These elements can be set using either assignment or in-place addition. Once again, these two modes can not be mixed unless the matrix is assembled in between, this time using the partialAssemble() method. Before using the matrix a call must be made to the assemble() method. This can be used between modes of setting matrix elements as well, but is slower than partialAssemble().
It is possible to use one of these matrix objects to represent finite-difference operations in the field-aligned direction. Much like when working with Fields (see Parallel Transforms), this can be achieved using the `yup()` and `ydown()` methods. These return a shallow-copy of the matrix object, with a flag indicating it is offset up or downwards in the y-direction. When using the parentheses operator to get a particular matrix element, the mesh’s `ParallelTransform` object will be queried to find the positions and weights needed to interpolate values onto field lines. This information is stored in the `PetscMatrix::Element` object which is returned. When that object is assigned to, it will set multiple matrix elements in the specified row, corresponding to each cell used to interpolate the along-field value. Note that the same cell might be used for interpolating more than one along-field value and it is thus possible you would end up overwriting a matrix element that you need. As such, you should always use in-place addition when using `yup()` and `ydown()`.

Putting all of this together, a matrix can be created corresponding to the Laplace operator defined in `OperatorStencil`.

```cpp
Bout++ Documentation, Release 5.0.0-alpha

PetscMatrix<Field2D> matrix(indexer);
Field2D &dx = localmesh->getCoordinates()->dx,
       &dy = localmesh->getCoordinates()->dy;

// Set up x-derivatives
BOUT_FOR(i, indexer->getRegionNobndry()) {
    matrix(i, i.xp()) = 1./SQ(dx[i]);
    matrix(i, i) = -2./SQ(dx[i]);
    matrix(i, i.xm()) = 1./SQ(dx[i]);
}
BOUT_FOR(i, indexer->getRegionInnerX()) {
    matrix(i, i.xp()) = 1./dx[i];
    matrix(i, i) = -1./dx[i];
}
BOUT_FOR(i, indexer->getRegionOuterX()) {
    matrix(i, i) = 1./dx[i];
    matrix(i, i.xm()) = -1./dx[i];
}
matrix.partialAssemble();

// Set up y-derivatives
BOUT_FOR(i, indexer->getRegionNobndry()) {
    matrix.yup()(i, i.yp()) += 1./SQ(dy[i]);
    matrix(i, i) += -2./SQ(dy[i]);
    matrix.ydown()(i, i.ym()) += 1./SQ(dy[i]);
}
BOUT_FOR(i, indexer->getRegionLowerY()) {
    matrix.yup()(i, i.yp()) += 1./dy[i];
    matrix(i, i) += -1./dy[i];
}
BOUT_FOR(i, indexer->getRegionUpperY()) {
    matrix(i, i) += 1./dy[i];
    matrix.ydown()(i, i.ym()) += -1./dy[i];
}
matrix.assemble();
```

### G.6 Use With Other Parts of PETSc

At present, only the `Mat` and `Vec` objects in PETSc have been wrapped. This is because they are by far the most difficult components to use and benefit the most from providing this interface. While in future a C++ interface may be provided to other components of PETSc, for the time being it is not too difficult to use the raw C API. This can be done by getting a pointer to the raw `Mat` and `Vec` objects using the `PetscMatrix::get()` and `PetscVector::get()`
methods. For example, to set up and use a linear solver for the problem in previous sections could be done as below:

```cpp
MatSetBlockSize(*matrix.get(), 1);
KSP solver;
KSPSetOperators(solver, *matrix.get(), *matrix.get());
KSPSetType(solver, "richardson")
KSPRichardsonSetScale(solver, 1.0)
KSPSetTolerances(solver, 1e-8, 1e-8, 1e6, 100000);
KSPSetInitialGuessNonzero(solver, PETSC_TRUE);

// Set up an algebraic multigrid preconditioner
PC precond;
KSPGetPC(solver, &precond);
PCSetType(precond, PCGAMGAGG);
PCGAMGSetSymGraph(precond, PETSC_TRUE);

PetscVector<Field2D> guess = rhs_vec;
guess.assemble();

KSPSolve(solver, *rhs_vec.get(), *guess.get());
KSPConvergedReason reason;
KSPGetConvergedReason(solver, &reason);
if (reason <= 0) {
    throw BoutException("PETSc solver failed");
}

Field2D solution = guess.toField();
```
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).

### H.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);
```

### H.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
...

// Save mesh configuration into output file
mesh->outputVars(dump);
```

which is called during BOUT++ initialisation.

### H.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 $ixseps1 \leq x < LocalNx$ are outside. See *BOUT++ Topology* for more details.

### H.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;
```

### H.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.).

### H.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:
To read 2D and 3D fields, the branch-cuts need to be taken into account.

### H.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:

```cpp
template<typename... Ts>
int communicate(Ts&... ts); // Communicate one or more fields
int communicate(FieldGroup); // Communicate a group of fields

int 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:

```cpp
// 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:

```cpp
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()`:

```cpp
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
```

There are also methods that allow communications only in the X or only in Y directions:

```cpp
template<typename... Ts>
int communicateXZ(Ts&... ts); // Communicate one or more fields
int communicateXZ(FieldGroup); // Communicate a group of fields
comm_handle sendX(FieldGroup); // Send data

template<typename... Ts>
int communicateYZ(Ts&... ts); // Communicate one or more fields
int communicateYZ(FieldGroup); // Communicate a group of fields
comm_handle sendY(FieldGroup); // Send data
```
When the option `mesh:include_corner_cells` is set to `true` (which is the default), the guard cells are communicating first in the y-direction and then in the x-direction, so that the corner cells are communicated consistently.

Setting `mesh:include_corner_cells = false` turns this off, so that corner cells are communicated only in y, and x- and y-direction communications are sent concurrently. This was the default behaviour in BOUT++ v4.3 and earlier, and might possibly be faster in some cases, when corner cells are not needed.

**H.5.1 Implementation: BoutMesh**

In *BoutMesh*, 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;
```

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 < \text{LocalNn}`

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.

**H.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.

**H.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:

```cpp
for(surf->first(); !surf->isDone(); surf->next()) {
    ...
}
```

To test if the surface is closed, there's the test `SurfaceIter::closed()`:

```cpp
bool surf->closed(BoutReal &ts)
```

which returns true if the surface is closed, along with the twist-shift angle.

## H.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()`:

```cpp
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.

## H.9 Differencing

The mesh spacing is given by the public members `Mesh::dx`, `Mesh::dy` and `Mesh::dx`:

```cpp
// 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)
```

## H.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`:

```cpp
// 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:
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;
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.

**H.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:

```cpp
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:

```cpp
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\(^1\). 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>
}
```

\(^1\) Support for PDB files was removed in BOUT++ 4.0.0
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);
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, 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);

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;
};

(continues on next page)
// 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 = 0, int ly = 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(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 ly = 0, int lz = 0) = 0;

// Read / Write record-based variables
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_rec(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 ly = 0, int lz = 0) = 0;
virtual bool write_rec_perp(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;

// Optional functions

virtual void setLowPrecision() { } // By default doesn't do anything

// Attributes

/// Sets a string attribute
///
/// @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
///
/// @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
///
/// @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
///
/// @param[in] varname Variable name. The variable must already exist. If
virtual bool getAttribute(const std::string &varname, const std::string &attrname, std::string &text) = 0;

virtual bool getAttribute(const std::string &varname, const std::string &attrname, int &value) = 0;

virtual bool getAttribute(const std::string &varname, const std::string &attrname, BoutReal &value) = 0;

void writeFieldAttributes(const std::string& name, const Field& f);
void writeFieldAttributes(const std::string& name, const FieldPerp& f);

void readFieldAttributes(const std::string& name, Field& f);
void readFieldAttributes(const std::string& name, FieldPerp& f);
I.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 `boutdata.restart.redistribute` will fail because they find inconsistent `yindex_global` values.
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.

### J.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.

### J.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:
<table>
<thead>
<tr>
<th>Building language:</th>
<th>fr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building language:</td>
<td>zh_CN</td>
</tr>
<tr>
<td>Building language:</td>
<td>zh_TW</td>
</tr>
</tbody>
</table>

The new language should now be available (no need to recompile BOUT++).
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.

K.1 Scorep/Scalasca profiling

K.1.1 Instrumentation

Scorep automatically reports the time spent 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;

};
```

Regions of a function can also be timed by enclosing the region in braces and using the `BOUT_SCOREP_REGION` macro. For example,
void Field2D::applyBoundary(BoutReal time) {
SCOREP0();

checkData(*this);
{
BOUT_SCOREP_REGION("display name");
for (const auto& bndry : bndry_op) {
    bndry->apply(*this, time);
}
}
};

Here, the SCOREP0 macro ensures the whole applyBoundary function is timed. In addition, the for loop is also timed and appears in the Scalasca profile as a region inside applyBoundary with the name “display name”. Any number of Scorep user regions can be used in a function; user regions can also be nested.

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.

K.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).

K.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>_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.
K.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`

K.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.

K.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.

K.2.2 Run

To run, add a trace script into the normal run command, so that for example

```
$ aprun -n 16 blob2d -d delta_l
```

becomes

```
$ aprun -n 16 ./trace.sh blob2d -d delta_l
```

where `trace.sh` is the script file

```
#!/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.mpir`. To generate the `.prv` trace file that can be read by Paraver, do
K.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.

K.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 `-ffopenmp` to `BOUT_FLAGS`, when profiling OpenMP-parallelized code.
L.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)`
  $\nabla^2_\perp$ operator
- `Field = Div(Vector)`
  Divergence of a vector
- `Field = Div_par(Field f)`
  Parallel divergence $B_0b \cdot \nabla(f/B_0)$
- `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)

• Field = invert_parderiv(Field2D|BoutReal A, Field2D|BoutReal B, Field3D r)
  Inverts an equation $A \times x + B \times \text{Grad}_2(x) = r$

• 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.

L.2 File list

L.2.1 File adams_bashforth.cxx

L.2.2 File adams_bashforth.hxx

class AdamsBashforthSolver: public Solver

Public Functions

AdamsBashforthSolver (Options *options = nullptr)
~AdamsBashforthSolver ()
void resetInternalFields ()
  Should wipe out internal field vector and reset from current field object data.
void setMaxTimestep (BoutReal dt)
  Set a maximum internal timestep (only for explicit schemes)
BoutReal getCurrentTimestep ()
  Return the current internal timestep.

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

int run ()
  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

```cpp
BoutReal take_step(BoutReal timeIn, BoutReal dt, int order, Array<BoutReal> &current, Array<BoutReal> &result)
```

Private Members

- Array<BoutReal> state
- Array<BoutReal> nextState
- std::deque<Array<BoutReal>> history
- std::deque<BoutReal> times
- BoutReal atol
- BoutReal rtol
- BoutReal max_timestep
- int mxstep
- bool adaptive
- bool adaptive_order
- bool followHighOrder
- BoutReal dtFac
- int maximum_order
- BoutReal timestep
- BoutReal out_timestep
- int current_order
- int nsteps
- int nlocal
- int neq

L.2.3 File arkode.cxx

Defines

```cpp
#define ZERO
#define ONE
```

Typedefs

```cpp
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 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.

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
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

L.2.4 File arkode.hxx

class ArkodeSolver : public Solver

Public Functions

ArkodeSolver (Options *opts = nullptr)

~ArkodeSolver ()

void setJacobian (Jacobian j)
    Specify a Jacobian (optional)

BoutReal getCurrentTimestep ()
    Return the current internal timestep.

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

int run ()
    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 *
    vvec)

void jac (BoutReal t, BoutReal *ydata, BoutReal *vdata, BoutReal *Jvdata)
**Private Functions**

```c
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**

```c
int NOUT
BoutReal TIMESTEP
BoutReal hcur
Jacobian jacfunc = {nullptr}
bool diagnose = {false}
N_Vector uvec = {nullptr}
void *arkode_mem = {nullptr}
BoutReal pre_Wtime = {0.0}
int pre_ncalls = {0}
```

**L.2.5 File array.hxx**

**Functions**

```c
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**

```c
ArrayData (int size)
~ArrayData ()
iterator<T> begin () const
iterator<T> end () const
int size () const
void operator= (ArrayData<T> &in)
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:
   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

template<>
using data_type = T

template<>
using backing_type = Backing

template<>
using size_type = int

Public Functions

Array()
    Create an empty array
    
    Array a(); a.empty(); // True

Array(size_type len)
    Create an array of given length

~Array()
    Destructor. Releases the underlying dataBlock

Array(const Array &other)
    Copy constructor
Array \&operator= (Array other)
Assignment operator After this both Arrays share the same dataBlock
Uses copy-and-swap idiom

Array (Array \&\&other)
Move constructor

void reallocate (size_type new_size)
Reallocate the array with size = new_size
Note that this invalidates the existing data!

void clear()
Release data. After this the Array is empty and any data access will be invalid

bool empty() const
Returns true if the Array is empty

size_type size() const
Return size of the array. Zero if the array is empty.

bool unique() const
Returns true if the data is unique to this Array.

void ensureUnique()
Ensures that this Array does not share data with another. This should be called before performing any write
operations on the data.

iterator<T> begin()

iterator<T> end()

const_iterator<T> begin() const

const_iterator<T> end() const

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.

const T \&operator[] (size_type ind) const

Public Static Functions

static bool useStore (bool keep_using = true)
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 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

template<>
using dataBlock = Backing

template<>
using dataPtrType = std::shared_ptr<dataBlock>

template<>
using storeType = std::map<size_type, std::vector<dataPtrType>>

template<>
using arenaType = std::vector<storeType>

Private Functions

dataPtrType get (size_type len)
    Returns a pointer to a dataBlock object of size \texttt{len} with no references. This is either from the store, or newly allocated
    Expects \texttt{len} \texttt{>= 0}

void release (dataPtrType &d)
    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) \texttt{store()[<size>]} already exists, and b) it has space for at least one data block. Of course, \texttt{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 \texttt{Array}. May be null

Private Static Functions

static 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 initialsed on first use, and doesn’t need to be separately declared for each type \texttt{T}

\textit{Inputs}

\textit{Parameters}

\begin{itemize}
    \item cleanup: If set to true, deletes all dataBlock and clears the store
\end{itemize}

Friends

void swap (Array<T> &first, Array<T> &second)
    Exchange contents with another \texttt{Array} of the same type. Sizes of the arrays may differ.
L.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
condition - The expression to test
e.g. ASSERT2( condition ) will only test condition if CHECK >= 2

ASSERT0 (condition)
ASSERT1 (condition)
ASSERT2 (condition)
ASSERT3 (condition)

L.2.7 File bilinear_xz.cxx

L.2.8 File boundary_factory.cxx

L.2.9 File boundary_factory.hxx

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.

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();
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()");
```

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)`

#### add (BoundaryOp * bop, const std::string & name)

Add available boundary conditions and modifiers Supply an object, and the name to be used

#### add (BoundaryOp * bop, const char * name)

Add a boundary condition.

Note: This method should be removed, as the string method is sufficient

#### addMod (BoundaryModifier * bmod, const std::string & name)

Add a boundary condition modifier

#### addMod (BoundaryModifier * bmod, const char * name)

Note: This method should be removed, as the string method is sufficient

#### add (BoundaryOpPar * bop, const std::string & name)

#### add (BoundaryOpPar * bop, const char * name)

### Public Static Functions

#### BoundaryFactory * getInstance ()

Return a pointer to the only instance.

#### 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)

Private Members

std::map<std::string, BoundaryOp *> opmap
std::map<std::string, BoundaryModifier *> modmap
std::map<std::string, BoundaryOpPar *> par_opmap

Private Static Attributes

BoundaryFactory *instance = nullptr  
The only instance of this class (Singleton)

L.2.10 File boundary_op.hxx

class BoundaryOpBase  
Subclassed by BoundaryOp, BoundaryOpPar

Public Functions

BoundaryOpBase()  
virtual ~BoundaryOpBase()  

virtual void apply(Field2D &f) = 0  
Apply a boundary condition on field f.

virtual void apply(Field2D &f, BoutReal t)

virtual void apply(Field3D &f) = 0

virtual void apply(Field3D &f, BoutReal t)

virtual void apply(Vector2D &f)

virtual void apply(Vector3D &f)

class BoundaryOp: public BoundaryOpBase  
#include <boundary_op.hxx> An operation on a boundary.

**Public Functions**

BoundaryOp()  
BoundaryOp(BoundaryRegion *region)  
~BoundaryOp()  

virtual BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args)

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

virtual void apply_ddt(Field2D &f)

Apply a boundary condition on ddt(f)

virtual void apply_ddt(Field3D &f)

virtual void apply_ddt(Vector2D &f)

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**

BoundaryModifier()  
BoundaryModifier(BoundaryOp *operation)

virtual BoundaryOp *cloneMod(BoundaryOp *op, const std::list<std::string> &args) = 0

**Protected Attributes**

BoundaryOp *op = {nullptr}
L.2.11 File boundary_region.cxx

L.2.12 File boundary_region.hxx

Enums

enum BndryLoc
    Location of boundary.
    Values:
    xin
    xout
    ydown
    yup
    all
    par_fwd
    par_bkwd

Variables

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()

BoundaryRegionBase(std::string name, Mesh *passmesh = nullptr)

BoundaryRegionBase(std::string name, BndryLoc loc, Mesh *passmesh = nullptr)

virtual ~BoundaryRegionBase()

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 ()

BoundaryRegion (std::string name, BndryLoc loc, Mesh *passmesh = nullptr)

BoundaryRegion (std::string name, int xd, int yd, Mesh *passmesh = nullptr)

~BoundaryRegion ()

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
    Indices of the point in the boundary.

int y
    Direction of the boundary [x+bx][y+by] is going outwards.

int bx

int by

int width
    Width of the boundary.

class BoundaryRegionXIn: public BoundaryRegion
Public Functions

BoundaryRegionXIn (std::string name, int ymin, int ymax, Mesh *passmesh = nullptr)

void first ()
    Move the region iterator to the start.

void next ()
    Get the next element in the loop over every element from inside out (in X or Y first)

void next1d ()
    Loop over the innermost elements.

void nextX ()
    Just loop over X.

void nextY ()
    Just loop over Y.

bool isDone ()
    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)

void first ()
    Move the region iterator to the start.

void next ()
    Get the next element in the loop over every element from inside out (in X or Y first)

void next1d ()
    Loop over the innermost elements.

void nextX ()
    Just loop over X.

void nextY ()
    Just loop over Y.

bool isDone ()
    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)

void first ()
    Move the region iterator to the start.

void next ()
    Get the next element in the loop over every element from inside out (in X or Y first)

void next1d ()
    Loop over the innermost elements.

void nextX ()
    Just loop over X.

void nextY ()
    Just loop over Y.

bool isDone ()
    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)

void first ()
    Move the region iterator to the start.

void next ()
    Get the next element in the loop over every element from inside out (in X or Y first)

void next1d ()
    Loop over the innermost elements.

void nextX ()
    Just loop over X.

void nextY ()
    Just loop over Y.
bool isDone()
    Returns true if outside domain. Can use this with nested nextX, nextY.

**Private Members**

int xs
int xe

```cpp
namespace bout
    SNB model

namespace globals
```

**Variables**

Mesh *mesh
    Global mesh.
    The mesh object.

### L.2.13 File boundary_standard.cxx

**Functions**

```cpp
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.
```

### L.2.14 File boundary_standard.hxx

**Functions**

```cpp
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**

```cpp
BoundaryDirichlet_2ndOrder()

BoundaryDirichlet_2ndOrder(BoutReal setval)

BoundaryDirichlet_2ndOrder(BoundaryRegion *region, BoutReal setval = 0.)

BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args)
```
void **apply** *(Field2D &f)*

Apply a boundary condition on field f.

void **apply** *(Field3D &f)*

void **apply** *(Field2D &f)*

Apply a boundary condition on ddt(f)

void **apply** *(Field3D &f)*

### Private Members

BoutReal **val**

class **BoundaryDirichlet** : public BoundaryOp

#include <boundary_standard.hxx> Dirichlet (set to zero) boundary condition.

### Public Functions

BoundaryDirichlet ()

BoundaryDirichlet *(BoundaryRegion *region, std::shared_ptr<FieldGenerator> g)*

BoundaryOp *clone *(BoundaryRegion *region, const std::list<std::string> &args)*

void **apply** *(Field2D &f)*

Apply a boundary condition on field f.

void **apply** *(Field2D &f, BoutReal t)*

void **apply** *(Field3D &f)*

void **apply** *(Field3D &f, BoutReal t)*

void **apply** *(Field2D &f)*

Apply a boundary condition on ddt(f)

void **apply** *(Field3D &f)*

### Private Members

std::shared_ptr<FieldGenerator> **gen**

class **BoundaryDirichlet_O3** : public BoundaryOp

#include <boundary_standard.hxx> 3nd-order boundary condition

### Public Functions

BoundaryDirichlet_O3 ()

BoundaryDirichlet_O3 *(BoundaryRegion *region, std::shared_ptr<FieldGenerator> g)*

BoundaryOp *clone *(BoundaryRegion *region, const std::list<std::string> &args)*

void **apply** *(Field2D &f)*

Apply a boundary condition on field f.
void apply (Field2D & f, BoutReal t)

void apply (Field3D & f)

void apply (Field3D & f, BoutReal t)

void apply_ddt (Field2D & f)
    Apply a boundary condition on ddt(f)

void apply_ddt (Field3D & f)

Private Members

std::shared_ptr<FieldGenerator> gen

class BoundaryDirichlet_04 : public BoundaryOp
    #include <boundary_standard.hxx> 4th-order boundary condition

Public Functions

BoundaryDirichlet_04 ()

BoundaryDirichlet_04 (BoundaryRegion * region, std::shared_ptr<FieldGenerator> g)

BoundaryOp * clone (BoundaryRegion * region, const std::list<std::string> & args)

void apply (Field2D & f)
    Apply a boundary condition on field f.

void apply (Field2D & f, BoutReal t)

void apply (Field3D & f)

void apply (Field3D & f, BoutReal t)

void apply_ddt (Field2D & f)
    Apply a boundary condition on ddt(f)

void apply_ddt (Field3D & f)

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

BoundaryDirichlet_4thOrder ()

BoundaryDirichlet_4thOrder (BoutReal setval)

BoundaryDirichlet_4thOrder (BoundaryRegion * region, BoutReal setval = 0.)
**BoundaryOp** `*clone (BoundaryRegion *region, const std::list<std::string> &args)`

```cpp
void apply (Field2D &f)
    Apply a boundary condition on field f.

void apply (Field3D &f)
```

```cpp
void apply_ddt (Field2D &f)
    Apply a boundary condition on ddt(f)

void apply_ddt (Field3D &f)
```

**Private Members**

```cpp
BoutReal val
```

```cpp
class BoundaryNeumann_NonOrthogonal : public BoundaryOp
    #include <boundary_standard.hxx> Neumann (zero-gradient) boundary condition for non-orthogonal meshes.
```

**Public Functions**

```cpp
BoundaryNeumann_NonOrthogonal ()

BoundaryNeumann_NonOrthogonal (BoutReal setval)

BoundaryNeumann_NonOrthogonal (BoundaryRegion *region, BoutReal setval = 0.)

BoundaryOp *clone (BoundaryRegion *region, const std::list<std::string> &args)
```

```cpp
void apply (Field2D &f)
    Apply a boundary condition on field f.

void apply (Field3D &f)
```

**Private Members**

```cpp
BoutReal val
```

```cpp
class BoundaryNeumann2 : public BoundaryOp
    #include <boundary_standard.hxx> Neumann (zero-gradient) boundary condition, using 2nd order on boundary.
```

**Public Functions**

```cpp
BoundaryNeumann2 ()

BoundaryNeumann2 (BoundaryRegion *region)

BoundaryOp *clone (BoundaryRegion *region, const std::list<std::string> &args)
```

```cpp
void apply (Field2D &f)
    Apply a boundary condition on field f.

void apply (Field3D &f)
```
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

BoundaryNeumann_2ndOrder()
BoundaryNeumann_2ndOrder(BoutReal setval)
BoundaryNeumann_2ndOrder(BoundaryRegion *region, BoutReal setval = 0.)

BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args)

void apply(Field2D &f)
Apply a boundary condition on field f.

void apply(Field3D &f)

void apply_ddt(Field2D &f)
Apply a boundary condition on ddt(f)

void apply_ddt(Field3D &f)

Private Members

BoutReal val

class BoundaryNeumann : public BoundaryOp

Public Functions

BoundaryNeumann()

BoundaryNeumann(BoundaryRegion *region, std::shared_ptr<FieldGenerator> g)

BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args)

void apply(Field2D &f)
Apply a boundary condition on field f.

void apply(Field2D &f, BoutReal t)

void apply(Field3D &f)

void apply(Field3D &f, BoutReal t)

void apply_ddt(Field2D &f)
Apply a boundary condition on ddt(f)

void apply_ddt(Field3D &f)
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

BoundaryNeumann_4thOrder()
BoundaryNeumann_4thOrder(BoutReal setval)
BoundaryNeumann_4thOrder(BoundaryRegion *region, BoutReal setval = 0.)
BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args)
void apply(Field2D &f)
    Apply a boundary condition on field f.
void apply(Field3D &f)
void apply_ddt(Field2D &f)
    Apply a boundary condition on ddt(f)
void apply_ddt(Field3D &f)

Private Members

BoutReal val

class BoundaryNeumann_O4 : public BoundaryOp
#include <boundary_standard.hxx> Neumann boundary condition set half way between guard cell and grid cell at 4th order accuracy.

Public Functions

BoundaryNeumann_O4()
BoundaryNeumann_O4(BoundaryRegion *region, std::shared_ptr<FieldGenerator> g)
BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args)
void apply(Field2D &f)
    Apply a boundary condition on field f.
void apply(Field2D &f, BoutReal t)
void apply(Field3D &f)
void apply(Field3D &f, BoutReal t)
void apply_ddt(Field2D &f)
    Apply a boundary condition on ddt(f)
void apply_ddt(Field3D &f)
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

BoundaryNeumannPar()

BoundaryNeumannPar(BoundaryRegion *region)

BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args)

void apply(Field2D &f)
    Apply a boundary condition on field f.

void apply(Field3D &f)

class BoundaryRobin : public BoundaryOp
#include <boundary_standard.hxx> Robin (mix of Dirichlet and Neumann)

Public Functions

BoundaryRobin()

BoundaryRobin(BoundaryRegion *region, BoutReal a, BoutReal b, BoutReal g)

BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args)

void apply(Field2D &f)
    Apply a boundary condition on field f.

void apply(Field3D &f)

Private Members

BoutReal aval

BoutReal bval

BoutReal gval

class BoundaryConstGradient : public BoundaryOp
#include <boundary_standard.hxx> Constant gradient (zero second derivative)

Public Functions

BoundaryConstGradient()

BoundaryConstGradient(BoundaryRegion *region)

BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args)
void apply (Field2D &f)
  Apply a boundary condition on field f.

void apply (Field3D &f)

class BoundaryZeroLaplace : public BoundaryOp
  
  #include <boundary_standard.hxx> Zero Laplacian, decaying solution.

Public Functions

BoundaryZeroLaplace()

BoundaryZeroLaplace (BoundaryRegion *region)

  BoundaryOp *clone (BoundaryRegion *region, const std::list<std::string> &args)

void apply (Field2D &f)
  Apply a boundary condition on field f.

void apply (Field3D &f)

class BoundaryZeroLaplace2 : public BoundaryOp
  
  #include <boundary_standard.hxx> Zero Laplacian.

Public Functions

BoundaryZeroLaplace2()

BoundaryZeroLaplace2 (BoundaryRegion *region)

  BoundaryOp *clone (BoundaryRegion *region, const std::list<std::string> &args)

void apply (Field2D &f)
  Apply a boundary condition on field f.

void apply (Field3D &f)

class BoundaryConstLaplace : public BoundaryOp
  
  #include <boundary_standard.hxx> Constant Laplacian, decaying solution.

Public Functions

BoundaryConstLaplace()

BoundaryConstLaplace (BoundaryRegion *region)

  BoundaryOp *clone (BoundaryRegion *region, const std::list<std::string> &args)

void apply (Field2D &f)
  Apply a boundary condition on field f.

void apply (Field3D &f)

class BoundaryDivCurl : public BoundaryOp
  
  #include <boundary_standard.hxx> Vector boundary condition \( \text{Div}(B) = 0, \text{Curl}(B) = 0 \).
Public Functions

BoundaryDivCurl()

BoundaryDivCurl(BoundaryRegion *region)

BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args)

void apply(Field2D &f)
    Apply a boundary condition on field f.

void apply(Field3D &f)

void apply(Vector2D &f)

void apply(Vector3D &f)

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

BoundaryFree()

BoundaryFree(BoutReal setval)

BoundaryFree(BoundaryRegion *region, BoutReal setval = 0.)

BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args)

void apply(Field2D &f)
    Apply a boundary condition on field f.

void apply(Field3D &f)

void apply_ddt(Field2D &f)
    Apply a boundary condition on ddt(f)

void apply_ddt(Field3D &f)

Private Members

BoutReal val

class BoundaryFree_O2: 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

BoundaryFree_O2()

BoundaryFree_O2(BoundaryRegion *region)

BoundaryOp *clone(BoundaryRegion *region, const std::list<std::string> &args)
void apply (Field2D &f)
    Apply a boundary condition on field f.

void apply (Field3D &f)

void apply_ddt (Field2D &f)
    Apply a boundary condition on ddt(f)

void apply_ddt (Field3D &f)

class BoundaryFree_O3 : public BoundaryOp

Public Functions

BoundaryFree_O3 ()

BoundaryFree_O3 (BoundaryRegion *region)

BoundaryOp *clone (BoundaryRegion *region, const std::list<std::string> &args)

void apply (Field2D &f)
    Apply a boundary condition on field f.

void apply (Field3D &f)

void apply_ddt (Field2D &f)
    Apply a boundary condition on ddt(f)

void apply_ddt (Field3D &f)

class BoundaryRelax : public BoundaryModifier
    #include <boundary_standard.hxx> Convert a boundary condition to a relaxing one.

Public Functions

BoundaryRelax ()

BoundaryRelax (BoundaryOp *operation, BoutReal rate)

BoundaryOp *cloneMod (BoundaryOp *op, const std::list<std::string> &args)

void apply (Field2D &f)
    Apply a boundary condition on field f.

void apply (Field2D &f, BoutReal t)

void apply (Field3D &f)

void apply (Field3D &f, BoutReal t)

void apply_ddt (Field2D &f)
    Apply a boundary condition on ddt(f)

void apply_ddt (Field3D &f)
Private Members

BoutReal r

class BoundaryWidth : public BoundaryModifier
#include <boundary_standard.hxx> Increase the width of a boundary.

Public Functions

BoundaryWidth()

BoundaryWidth(BoundaryOp *operation, int wid)

BoundaryOp *cloneMod(BoundaryOp *op, const std::list<std::string> &args)

void apply(Field2D &f)
    Apply a boundary condition on field f.

void apply(Field2D &f, BoutReal t)

void apply(Field3D &f)

void apply(Field3D &f, BoutReal t)

void apply_ddt(Field2D &f)
    Apply a boundary condition on ddt(f)

void apply_ddt(Field3D &f)

Private Members

int width

class BoundaryToFieldAligned : public BoundaryModifier
#include <boundary_standard.hxx> Convert input field fromFieldAligned, apply boundary and then convert back toFieldAligned Equivalent to converting the boundary condition to “Field Aligned” from “orthogonal”

Public Functions

BoundaryToFieldAligned()

BoundaryToFieldAligned(BoundaryOp *operation)

BoundaryOp *cloneMod(BoundaryOp *op, const std::list<std::string> &args)

void apply(Field2D &f)
    Apply a boundary condition on field f.

void apply(Field2D &f, BoutReal t)

void apply(Field3D &f)

void apply(Field3D &f, BoutReal t)

void apply_ddt(Field2D &f)
    Apply a boundary condition on ddt(f)
void apply_ddt (Field3D &f)

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

BoundaryFromFieldAligned()
BoundaryFromFieldAligned (BoundaryOp *operation)

BoundaryOp *cloneMod (BoundaryOp *op, const std::list<std::string> &args)

void apply (Field2D &f)
    Apply a boundary condition on field f.

void apply (Field2D &f, BoutReal t)

void apply (Field3D &f)

void apply (Field3D &f, BoutReal t)

void apply_ddt (Field2D &f)
    Apply a boundary condition on ddt(f)

void apply_ddt (Field3D &f)

L.2.15 File bout++.cxx

Defines

GLOBALORIGIN
BOUT_NO_USING_NAMESPACE_BOUTGLOBALS
BUILDFLAG1_ (x)
BUILDFLAG (x)
INDIRECT1_BOUTMAIN (a)
INDIRECT0_BOUTMAIN (...)
STRINGIFY (a)

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.

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.

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

const char DEFAULT_DIR[] = "data"

BoutReal simtime = {0.0}
int iteration = {0}

bool user_requested_exit = false

namespace bout

    SNB model

L.2.16 File bout.hxx

Functions

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:

    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.

```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 = 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)

```c
class BoutMonitor : public Monitor
#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

```c
BoutMonitor (BoutReal timestep = -1)
```

Private Functions

```c
int call (Solver *solver, BoutReal t, int iter, int NOUT)
```

SOLUTION MONITOR FUNCTION

Called each timestep by the solver

Private Members

```c
RunMetrics run_data
```

namespace bout

SNB model

namespace experimental

Typedefs

```c
using SignalHandler = void (*)(int)
```

Function type for handling signals.
Functions

void \textbf{setupSignalHandler} (\texttt{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 \texttt{--enable-signal}.
- For floating point errors, compile with \texttt{--enable-sigfpe}

void \textbf{defaultSignalHandler} (\texttt{int sig})

The default BOUT++ signal handler: throw an exception with an appropriate message

void \textbf{setupGetText} ()

Set up the i18n environment.

auto \textbf{parseCommandLineArgs} (\texttt{int argc, char **argv})

Parse the “fixed” command line arguments, like help and -d.

void \textbf{checkDataDirectoryIsAccessible} (\texttt{const std::string &data\_dir})

Throw an exception if \texttt{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 \textbf{setupOutput} (\texttt{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 \texttt{verbosity}, disable writing to stdout for \texttt{MYPE} \neq 0

void \textbf{savePIDtoFile} (\texttt{const std::string &data\_dir, int MYPE})

Save the process ID for processor \texttt{N = MYPE} to file “.BOUT.pid.N” in \texttt{data\_dir}, so it can be shut down by user signal

Throws if it was not possible to create the file

void \textbf{printStartupHeader} (\texttt{int MYPE, int NPES})

Print the initial header.

void \textbf{printCompileTimeOptions} ()

Print the compile-time options.

void \textbf{printCommandLineArguments} (\texttt{const std::vector<std::string> &original\_argv})

Print the arguments given on the command line.

bool \textbf{setupBoutLogColor} (\texttt{bool color\_output, int MYPE})

Setup the pipe etc and run stdout through bout-log-color. Return true if it was successful

void \textbf{setRunStartInfo} (\texttt{Options &options})

Set BOUT++ version information, along with current time (as \texttt{started}), into \texttt{run} section of \texttt{options}

void \textbf{setRunFinishInfo} (\texttt{Options &options})

Set the current time (as \texttt{finished}) into \texttt{run} section of \texttt{options}

void \textbf{writeSettingsFile} (\texttt{Options &options, const std::string &data\_dir, const std::string &settings\_file})

Write options to \texttt{settings\_file} in directory \texttt{data\_dir}.

\textit{Datafile} \textbf{setupDumpFile} (\texttt{Options &options, Mesh &mesh, const std::string &data\_dir})

Setup the output dump files from \texttt{options} using the \texttt{mesh}. Files are created in the \texttt{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.

L.2.17 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
L.2.18 File bout_types.cxx

Functions

std::string toString (CELL_LOC location)

CELLO CCELLOFromString (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)

L.2.19 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:

deflt
centre
xlow
ylow
zlow
vshift

enum DIFF_METHOD
Differential methods. Both central and upwind.

Values:
deflt
u1
u2
c2
w2
w3
c4
u3
fft
split
s2

enum REGION
Specify grid region for looping.

Values:
all
nobndry
nox
noy
noz

enum DIRECTION
To identify particular directions (in index space):

• X, Y, Z are the coordinate directions
• YAligned is a special case of Y, indicating a field-aligned grid, where the x- and z- axes are not necessarily orthogonal
• 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

Values:
X
Y
Z
YAligned
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:
Standard
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:
Standard
Average

enum STAGGER
To identify valid staggering combinations.

Values:
None
C2L
L2C

enum DERIV
To identify types of derivative method combinations.

Values:
Standard
StandardSecond
StandardFourth
Upwind
Flux

Functions
std::string toString(CELL_LOC location)
CELL_LOC CELL_LOCFromString(const std::string &location_string)
std::string CELL_LOC_STRING (CELL_LOC location)
std::string toString(DIFF_METHOD location)
std::string DIFF_METHOD_STRING (DIFF_METHOD location)
std::string toString(REGION region)
std::string REGION_STRING (REGION region)
std::string toString (DIRECTION direction)
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)
std::string STAGGER_STRING (STAGGER stagger)
std::string toString (DERIV deriv)
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::xlow
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

        template<> using type = T

        Public Functions

        T lookup()

        Public Static Attributes

        const type value = val

L.2.20 File boutcomm.cxx

L.2.21 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

BoutComm *getInstance()
   Get a pointer to the only instance.

MPI_Comm get()
   Shortcut method.

void setArgs(int &c, char **&v)

void cleanup()

int rank()
   Rank: my processor number.

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

BoutComm *instance = nullptr
   The only instance of this class (Singleton)

L.2.22 File boutexception.cxx

Functions

void BoutParallelThrowRhsFail(int status, const char *message)
   Throw BoutRhsFail with message if any one process has non-zero status

L.2.23 File boutexception.hxx

Functions

void BoutParallelThrowRhsFail(int status, const char *message)
   Throw BoutRhsFail with message if any one process has non-zero status


```cpp
class BoutException : public std::exception
    Subclassed by BoutIterationFail, BoutRhsFail

Public Functions

BoutException (std::string msg)

template<class S, class ...Args>
BoutException (const S &format, const Args &... args)

~BoutException ()

const char *what () const

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

std::string message

std::string backtrace_message = {}

class BoutRhsFail : public BoutException

Public Functions

BoutRhsFail (std::string message)

template<class S, class ...Args>
BoutRhsFail (const S &format, const Args &... args)

class BoutIterationFail : public BoutException

Public Functions

BoutIterationFail (std::string message)

template<class S, class ...Args>
BoutIterationFail (const S &format, const Args &... args)
```

L.2. File list 327
L.2.24 File boutmain.hxx

Defines

GLOBAL
Main function for legacy interface

The user provides two standalone functions, physics_init and physics_run. This header defines a class Legacy-Model, which inherits from PhysicsModel. This class is used by Solver, and forwards calls to the user standalone functions.

It is recommended that users define a model by inheriting from PhysicsModel directly, rather than using this header.

Note: This header file implements a main() function, so should only be included once, and cannot be combined with a user-defined main().

Functions

int physics_init(bool restarting)
Initialise the model. Called once at the start

Return zero on success, non-zero error code

Parameters

• restarting: True if the simulation is restarting

int physics_run(BoutReal t)
Calculate the time derivative

Return zero on success, non-zero error code

Parameters

• t: Simulation time

template<class T>
void bout_solve(T &var, const char *name)
Global functions used by some legacy models.

bool bout_constrain(Field3D &var, Field3D &F_var, const char *name)
Add a constraint 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

int init (bool restarting)
    Initialise.

int rhs (BoutReal t)
    Calculate time derivatives.

L.2.25 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)

L.2.26 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 ()

int load ()
    Read in the mesh from data sources.
comm_handle send (FieldGroup &g)
    Send data between processors Does not wait for communications to complete, so wait() must be called before guard cell values are used

Example

Parameters
    • g: A group of fields to communicate
    comm_handle handle = mesh->send(group); ... mesh->wait(handle);

comm_handle sendX (FieldGroup &g, comm_handle handle = nullptr, bool disable_corners = false)
    Send only in the x-direction.

comm_handle sendY (FieldGroup &g, comm_handle handle = nullptr)
    Send only in the y-direction.

int wait (comm_handle handle)
    Wait for a send operation to complete

Parameters
    • handle: The handle returned by send()

MPI_Request sendToProc (int xproc, int yproc, BoutReal *buffer, int size, int tag)
    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: X index of processor to send to
    • yproc: Y index of processor to send to
    • buffer: A buffer of data to send
    • size: The length of buffer
    • tag: A label, must be the same at receive

comm_handle receiveFromProc (int xproc, int yproc, BoutReal *buffer, int size, int tag)
    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: X index of sending processor
    • yproc: Y index of sending processor
    • buffer: The buffer to fill with data. Must already be allocated of length size
    • size: The length of buffer
    • tag: A label, must be the same as send

int getNXPE ()
    The number of processors in the X direction.

int getNYPE ()
    The number of processors in the Y direction.
int getXProcIndex()
This processor’s index in X direction.

int getYProcIndex()
This processor’s index in Y direction.

bool firstX() const
to the left in X?
Is this processor the first in X? i.e. is there a boundary

bool lastX() const
right in X?
Is this processor last in X? i.e. is there a boundary to the

int sendXOut (BoutReal *buffer, int size, int tag)
Send a buffer of data to processor at X index +1

Parameters
• buffer: The data to send. Must be at least length size
• size: The number of BoutReals to send
• tag: A label for the communication. Must be the same at receive

int sendXIn (BoutReal *buffer, int size, int tag)
Send a buffer of data to processor at X index -1

Parameters
• buffer: The data to send. Must be at least length size
• size: The number of BoutReals to send
• tag: A label for the communication. Must be the same at receive

comm_handle irecvXOut (BoutReal *buffer, int size, int tag)
Receive a buffer of data from X index +1

Parameters
• buffer: A buffer to put the data in. Must already be allocated of length size
• size: The number of BoutReals to receive and put in buffer
• tag: A label for the communication. Must be the same as sent

comm_handle irecvXIn (BoutReal *buffer, int size, int tag)
Receive a buffer of data from X index -1

Parameters
• buffer: A buffer to put the data in. Must already be allocated of length size
• size: The number of BoutReals to receive and put in buffer
• tag: A label for the communication. Must be the same as sent
MPI_Comm **getXcomm** (int *jy*) **const**
Return communicator containing all processors in X.

MPI_Comm **getYcomm** (int *jx*) **const**
Return communicator containing all processors in Y.

bool **periodicY** (int *jx*, BoutReal &*ts*) **const**
Is local X index \( j_x \) periodic in Y?

**Parameters**
- \( j_x \): The local (on this processor) index in X
- ts: The Twist-Shift angle if periodic

bool **periodicY** (int *jx*) **const**
Is local X index \( j_x \) periodic in Y?

**Parameters**
- \( j_x \): The local (on this processor) index in X

int **numberOfYBoundaries** () **const**
Get number of boundaries in the y-direction, i.e. locations where there are boundary cells in the global grid

std::pair<bool, BoutReal> **hasBranchCutLower** (int *jx*) **const**
Is there a branch cut at this processor’s lower boundary?

**Return** 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

**Parameters**
- \( j_x \): The local (on this processor) index in X

std::pair<bool, BoutReal> **hasBranchCutUpper** (int *jx*) **const**
Is there a branch cut at this processor’s upper boundary?

**Return** 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

**Parameters**
- \( j_x \): The local (on this processor) index in X

int **ySize** (int *jx*) **const**
The number of points in Y at fixed X index \( j_x \).

bool **firstY** () **const**
Is this processor first in Y? i.e. is there a boundary at lower Y?

bool **lastY** () **const**
Is this processor last in Y? i.e. is there a boundary at upper Y?

bool **firstY** (int *xpos*) **const**
Is this processor first in Y? i.e. is there a boundary at lower Y?
bool lastY (int xpos) const
   Is this processor last in Y? i.e. is there a boundary at upper Y?

int UpXSplitIndex ()
   If the upper Y guard cells are split in two, return the X index where the split occurs.

int DownXSplitIndex ()
   If the lower Y guard cells are split in two, return the X index where the split occurs.

int sendYOutIndest (BoutReal *buffer, int size, int tag)
   Send data.

int sendYOutOutdest (BoutReal *buffer, int size, int tag)

int sendYInIndest (BoutReal *buffer, int size, int tag)

int sendYInOutdest (BoutReal *buffer, int size, int tag)

comm_handle irecvYOutIndest (BoutReal *buffer, int size, int tag)
   Non-blocking receive. Must be followed by a call to wait()

   Parameters
      • buffer: A buffer of length size which must already be allocated
      • size: The number of BoutReals expected
      • tag: The tag number of the expected message

comm_handle irecvYOutOutdest (BoutReal *buffer, int size, int tag)
   Non-blocking receive. Must be followed by a call to wait()

   Parameters
      • buffer: A buffer of length size which must already be allocated
      • size: The number of BoutReals expected
      • tag: The tag number of the expected message

comm_handle irecvYInIndest (BoutReal *buffer, int size, int tag)
   Non-blocking receive. Must be followed by a call to wait()

   Parameters
      • buffer: A buffer of length size which must already be allocated
      • size: The number of BoutReals expected
      • tag: The tag number of the expected message

comm_handle irecvYInOutdest (BoutReal *buffer, int size, int tag)
   Non-blocking receive. Must be followed by a call to wait()

   Parameters
      • buffer: A buffer of length size which must already be allocated
      • size: The number of BoutReals expected
      • tag: The tag number of the expected message
const RangeIterator iterateBndryLowerY() const
    Iterate over the lower Y boundary.

const RangeIterator iterateBndryUpperY() const
    Iterate over the upper Y boundary.

const RangeIterator iterateBndryLowerInnerY() const
const RangeIterator iterateBndryLowerOuterY() const
const RangeIterator iterateBndryUpperInnerY() const
const RangeIterator iterateBndryUpperOuterY() const

std::vector<BoundaryRegion *> getBoundaries()
    Return a vector containing all the boundary regions on this processor.

std::vector<BoundaryRegionPar *> getBoundariesPar()
    Get all the parallel (Y) boundaries on this processor.

void addBoundaryPar(BoundaryRegionPar *bndry)
    Add a parallel(Y) boundary to this processor.

const Field3D smoothSeparatrix(const Field3D &f)
    Branch-cut special handling (experimental)

int getNx() const

int getNy() const

BoutReal GlobalX(int jx) const
    Continuous X index between 0 and 1.

BoutReal GlobalY(int jy) const
    Continuous Y index (0 -> 1)

BoutReal GlobalX(BoutReal jx) const
    Continuous X index between 0 and 1.

BoutReal GlobalY(BoutReal jy) const
    Continuous Y index (0 -> 1)

BoutReal getIxseps1() const

BoutReal getIxseps2() const

void outputVars(Datafile &file)
    Add output variables to a data file These are used for post-processing

int getGlobalXIndex(int xlocal) const
    Returns a global X index given a local index. Global index includes boundary cells, local index includes boundary or guard cells.

int getGlobalXIndexNoBoundaries(int xlocal) const
    Returns a global X index given a local index. Global index excludes boundary cells, local index includes boundary or guard cells.

int getLocalXIndex(int xglobal) const
    Returns a local X index given a global index. Global index includes boundary cells, local index includes boundary or guard cells.
int getLocalXIndexNoBoundaries (int xglobal) const
  Returns a local X index given a global index. Global index excludes boundary cells, local index includes boundary or guard cells.

int getGlobalYIndex (int ylocal) const
  Returns a global Y index given a local index. Global index includes boundary cells, local index includes boundary or guard cells.

int getGlobalYIndexNoBoundaries (int ylocal) const
  Returns a global Y index given a local index. Global index excludes boundary cells, local index includes boundary or guard cells.

int getLocalYIndex (int yglobal) const
  Returns a local Y index given a global index. Global index includes boundary cells, local index includes boundary or guard cells.

int getLocalYIndexNoBoundaries (int yglobal) const
  Returns a local Y index given a global index. Global index excludes boundary cells, local index includes boundary or guard cells.

int getGlobalZIndex (int zlocal) const
  Returns a global Z index given a local index. Global index includes boundary cells, local index includes boundary or guard cells.

int getGlobalZIndexNoBoundaries (int zlocal) const
  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

int getLocalZIndex (int zglobal) const
  Returns a local Z index given a global index. Global index includes boundary cells, local index includes boundary or guard cells.

int getLocalZIndexNoBoundaries (int zglobal) const
  Returns a local Z index given a global index. Global index excludes boundary cells, local index includes boundary or guard cells.

**Protected Functions**

BoutMesh (int input_nx, int input_ny, int input_nz, int mxg, int myg, int nxpe, int nype, int pe_xind, int pe_yind)
  A constructor used when making fake meshes for testing. This will make a mesh which thinks it corresponds to the subdomain on one processor, even though it’s actually being run in serial.

void overlapHandleMemory (BoutMesh *yup, BoutMesh *ydown, BoutMesh *xin, BoutMesh *xout)
  For debugging purposes (when creating fake parallel meshes), make the send and receive buffers share memory. This allows for communications to be faked between meshes as though they were on different processors.

**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 YLOCAL (int yglo, int yproc) const
    Return the Y processor number given a global Y index.

int YPROC (int yind)
    Return the X processor number given a global X index.

void default_connections ()
    Connection initialisation: Set processors in a simple 2D grid.

void set_connection (int ypos1, int ypos2, int xge, int xlt, bool ts = false)
    Add a topology connection.
    Set ypos1 and ypos2 to be neighbours in the range xge <= x < xlt. Optional argument ts sets whether to
    use twist-shift condition

void add_target (int ypos, int xge, int xlt)
    Add a divertor target or limiter.
    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.

void topology ()

void addBoundaryRegions ()
    Adds 2D and 3D regions for boundaries.

void free_handle (CommHandle *h)

BoutMesh::CommHandle *get_handle (int xlen, int ylen)

void clear_handles ()

void post_receiveX (CommHandle &ch)
    Create the MPI requests to receive data in the x-direction. Non-blocking call.

void post_receiveY (CommHandle &ch)
    Create the MPI requests to receive data in the y-direction. Non-blocking call.

int pack_data (const std::vector<FieldData *> &var_list, int xge, int xlt, int yge, int ylt, BoutReal*
    buffer)
    Take data from objects and put into a buffer.

int unpack_data (const std::vector<FieldData *> &var_list, int xge, int xlt, int yge, int ylt, BoutReal*
    buffer)
    Copy data from a buffer back into the fields.
Private Members

std::string gridname
int nx
int ny
int nz
    Size of the grid in the input file.
int MX
int 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::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?

bool include_x_corners  
   Are corner cells included in x-communication?

bool has_y_communication  
   Is there a y-communication.

FieldGroup var_list  
   List of fields being communicated.

L.2.27 File build_config.hxx

Defines

STRINGIFY1(x)

STRINGIFY(x)

namespace bout
   SNB model

namespace build

Variables

constexpr auto check_level = BOUT_CHECK_LEVEL

constexpr auto openmp_schedule = "BOUT_OPENMP_SCHEDULE"

constexpr auto has_fftw = static_cast<bool>(BOUT_HAS_FFTW)

constexpr auto has_gettext = static_cast<bool>(BOUT_HAS_GETTEXT)
constexpr auto has_hdf5 = static_cast<bool>(BOUT_HAS_HDF5)
constexpr auto has_lapack = static_cast<bool>(BOUT_HAS_LAPACK)
constexpr auto has_netcdf = static_cast<bool>(BOUT_HAS_NETCDF)
constexpr auto has_petsc = static_cast<bool>(1)
constexpr auto has_pretty_function = static_cast<bool>(BOUT_HASPRETTY_FUNCTION)
constexpr auto has_pvode = static_cast<bool>(1)
constexpr auto has_scorep = static_cast<bool>(1)
constexpr auto has_slepc = static_cast<bool>(1)
constexpr auto has_sundials = static_cast<bool>(BOUT_HASSUNDIALS)
constexpr auto use_backtrace = static_cast<bool>(BOUT_USE_BACKTRACE)
constexpr auto use_color = static_cast<bool>(BOUT_USE_COLOR)
constexpr auto use_omp = static_cast<bool>(BOUT_USE_OPENMP)
constexpr auto use_output_debug = static_cast<bool>(BOUT_USE_OUTPUT_DEBUG)
constexpr auto use_sigfpe = static_cast<bool>(BOUT_USE_SIGFPE)
constexpr auto use_signal = static_cast<bool>(BOUT_USE_SIGNAL)
constexpr auto use_track = static_cast<bool>(BOUT_USE_TRACK)

L.2.28 File cashkarp.cxx

L.2.29 File cashkarp.hxx

class CASHKARPScheme: public RKScheme

Public Functions

CASHKARPScheme(Options *options)

L.2.30 File constants.hxx

Variables

constexpr BoutReal PI = 3.141592653589793
Mathematical constant pi.
constexpr BoutReal TWOPI = 2 * PI
Mathematical constant 2 * pi.

namespace SI
Variables

```cpp
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 BoutRealMp = 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
constexpr BoutReal M_Deuterium = 2.01410178 * amu
   // Mass of a Deuterium atom.
constexpr BoutReal M_Tritium = 3.0160492 * amu
   // Mass of a Tritium atom.
```

L.2.31 File coordinates.cxx

L.2.32 File coordinates.hxx

Defines

```cpp
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

```cpp
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
```

L.2. File list
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 g23, 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&&)

~Coordinates()

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

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.

void setParallelTransform (std::unique_ptr<ParallelTransform> pt)

Set the parallel (y) transform for this mesh. Mostly useful for tests.

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, 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")

Field2D DDX (const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)

Field2D DDX (const Field2D &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")

Field2D DDX (const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)

Field2D DDX (const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)
Field2D DDZ(const Field2D &f, CELL_LOC outloc, DIFF_METHOD 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)

Field2D Grad_par(const Field2D &f, CELL_LOC outloc, DIFF_METHOD method)

Field3D Grad_par(const Field3D &var, 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)

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")

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)

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")

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")

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")

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, const std::string &dfdy_boundary_conditions = "free_o3", const std::string &dfdy_dy_region = "")

Field3D Laplace(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &dfdy_boundary_conditions = "free_o3", const std::string &dfdy_dy_region = ")

Field2D Laplace_perpXY(const Field2D &A, const Field2D &f)
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
d pitch angle / dx. Needed for vector differentials (Curl)
Field2D IntShiftTorsion
Integrated shear (I in BOUT notation)

Private Functions

void setParallelTransform(Options *options)
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}
Handles calculation of yup and ydown.

L.2.33 File cvode.cxx

Defines

ZERO
ONE

Typedefs

using CVODEINT = bout::utils::function_traits<CVLocalFn>::arg_t<0>
Functions

```c
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 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.
```

```c
int CVSpilsSetJacTimes (void *arkode_mem, std::nullptr_t, CVSpilsJacTimesVecFn jtimes)
```

L.2.34 File cvode.hxx

class CvodeSolver : public Solver

Public Functions

```c
CvodeSolver (Options *opts = nullptr)
~CvodeSolver ()
void setJacobian (Jacobian j)
    Specify a Jacobian (optional)

BoutReal getCurrentTimestep ()
    Return the current internal timestep.

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

int run ()
    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 resetInternalFields ()
    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_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
Jacobian jacfunc = {nullptr}
bool diagnose = {false}
N_Vector uvec = {nullptr}
void *cvode_mem = {nullptr}
BoutReal pre_Wtime = {0.0}
int pre_ncalls = {0}
bool cvode_initialised = false

L.2.35 File cyclic.cxx

FFT + Tridiagonal solver in serial or parallel.
Not particularly optimised: Each y slice is solved sequentially

CHANGELOG

Jan 2014: Brendan Shanahan bws502@york.ac.uk
  • Added DST option
Copyright 2013 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/.
L.2.36 File cyclic.hxx

class InvertParCR: public InvertPar

Public Functions

InvertParCR (Options *opt, CELL_LOC location = CELL_CENTRE, Mesh *mesh_in = bout::globals::mesh)

const Field3D solve (const Field3D &f)
    Solve the system of equations
    This method must be implemented

void setCoefA (const Field2D &f)
    Set the constant coefficient A

void setCoefB (const Field2D &f)
    Set the Grad2_par2 coefficient B

void setCoefC (const Field2D &f)
    Set the D2DYDZ coefficient C

void setCoefD (const Field2D &f)
    Set the D2DZ2 coefficient D

void setCoefE (const Field2D &f)
    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

L.2.37 File cyclic_laplace.cxx

L.2.38 File cyclic_laplace.hxx

class LaplaceCyclic: public Laplacian
    #include <cyclic_laplace.hxx> Solves the 2D Laplacian equation using the CyclicReduce class.
Public Functions

LaplaceCyclic (Options *opt = nullptr, const CELL_LOC loc = CELL_CENTRE, Mesh *mesh_in = nullptr)

~LaplaceCyclic()

void setCoefA(const Field2D &val)
    Set coefficients for inversion. Re-builds matrices if necessary.

void setCoefC(const Field2D &val)

void setCoefC1 (const Field2D &val)

void setCoefC2 (const Field2D &val)

void setCoefD (const Field2D &val)

void setCoefEx (const Field2D &val)

void setCoefEz (const Field2D &val)

FieldPerp solve (const FieldPerp &b)

FieldPerp solve (const FieldPerp &b, const FieldPerp &x0)

Field3D solve (const Field3D &b)

Field3D solve (const Field3D &b, const Field3D &x0)
    Performs the laplacian inversion y-slice by y-slice

    Return x All the y-slices of x_slice in the equation A*x_slice = b_slice

    Parameters

    • b: All the y-slices of b_slice, which is the right hand side of the equation A*x_slice = b_slice
    • x0: All the y-slices of the variable eventually used to set BC

Private Members

Field2D Acoef
Field2D C1coef
Field2D C2coef
Field2D Dcoef
int nmode
int xs
int xe
Matrix<dcomplex> a
Matrix<dcomplex> b
Matrix<dcomplex> c
Matrix<dcomplex> bcmplx
Matrix\texttt{<dcomplex> xcmplx} 
bool \texttt{dst} 
CyclicReduce\texttt{<dcomplex>} *\texttt{cr} 
Tridiagonal solver.

L.2.39 File cyclic_reduction.hxx

template<class T>
class CyclicReduce

\textbf{Public Functions}

\texttt{CyclicReduce()} 
\texttt{CyclicReduce(MPI_Comm c, int size)} 
void \texttt{setup(MPI_Comm c, int size)}
  \texttt{Set parameters}
  Parameters
  \begin{itemize}
    \item \texttt{c}: The communicator of all processors involved in the solve
    \item \texttt{size}: The number of rows on this processor
  \end{itemize}
\texttt{~CyclicReduce()} 
void \texttt{setPeriodic(bool p = true)}
  Specify that the tridiagonal system is periodic By default not periodic

void \texttt{setCoefs(const Array<T> \& a, const Array<T> \& b, const Array<T> \& c)}

void \texttt{setCoefs(const Matrix<T> \& a, const Matrix<T> \& b, const Matrix<T> \& c)}
  \texttt{Set} the entries in the matrix to be inverted

Parameters
  \begin{itemize}
    \item \texttt{a}: Left diagonal. Should have size [nsys][N] where N is set in the constructor or setup
    \item \texttt{b}: Diagonal values. Should have size [nsys][N]
    \item \texttt{c}: Right diagonal. Should have size [nsys][N]
  \end{itemize}

void \texttt{solve(const Array<T> \& rhs, Array<T> \& x)}
  Solve a set of tridiagonal systems

Parameters
  \begin{itemize}
    \item \texttt{rhs}: \texttt{Array} storing Values of the rhs for a single system
    \item \texttt{x}: \texttt{Array} storing the result for a single system
  \end{itemize}

void \texttt{solve(const Matrix<T> \& rhs, Matrix<T> \& x)}
  Solve a set of tridiagonal systems

Parameters
• rhs: Matrix storing Values of the rhs for each system
• x: Matrix storing the result for each system

Private Functions

void allocMemory (int np, int nsys, int n)
Allocate memory arrays

Parameters
  • np: Number of processors
  • nsys: Number of independent systems to solve
  • n: Size of each system of equations

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 interface rows for each system, which are stored in ifc.

\[(a1 \ b1 \ c1) \ (a2 \ b2 \ c2) \ (A1 \ B1 \ C1) \ (a3 \ b3 \ c3) \Rightarrow (A2 \ B2 \ C2) \ (\ldots) \ (an \ bn \ cn)\]

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\langle T \rangle \textbf{ifcs}
    Coefficients for interface solve.
Matrix\langle T \rangle \textbf{if2x2}
    2x2 interface equations on this processor
Matrix\langle T \rangle \textbf{ifx}
    Solution of interface equations.
Array\langle T \rangle \textbf{ifp}
    Interface equations returned to processor p.
Array\langle T \rangle \textbf{x1}
Array\langle T \rangle \textbf{xn}
    Interface solutions for back-solving.

\textbf{L.2.40 File datafile.cxx}

\textbf{L.2.41 File datafile.hxx}

Data file handling object definition.
26th Sep 2009: Modified to use varargs
Author B.Dudson
Date April 2009

\textbf{Defines}

\textbf{SAVE\_ONCE1} (var)
    Write this variable once to the grid file.
\textbf{SAVE\_ONCE2} (var1, var2)
\textbf{SAVE\_ONCE3} (var1, var2, var3)
\textbf{SAVE\_ONCE4} (var1, var2, var3, var4)
\textbf{SAVE\_ONCE5} (var1, var2, var3, var4, var5)
\textbf{SAVE\_ONCE6} (var1, var2, var3, var4, var5, var6)
\textbf{SAVE\_ONCE} (...)
\textbf{SAVE\_REPEAT1} (var)
    Write this variable every timestep.
\textbf{SAVE\_REPEAT2} (var1, var2)
\textbf{SAVE\_REPEAT3} (var1, var2, var3)
\textbf{SAVE\_REPEAT4} (var1, var2, var3, var4)
\textbf{SAVE\_REPEAT5} (var1, var2, var3, var4, var5)
\textbf{SAVE\_REPEAT6} (var1, var2, var3, var4, var5, var6)
\textbf{SAVE\_REPEAT} (...)

352 Appendix L. API reference
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)
~Datafile ()
Datafile &operator=(Datafile &&rhs)
Datafile &operator=(const Datafile &rhs)

bool openr (const std::string &filename)
   Open read-only.

template<class S, class ...Args>
bool openr (const S &format, const Args&... args)

bool openw (const std::string &filename)
   Overwrites existing file.

template<class S, class ...Args>
bool openw (const S &format, const Args&... args)

bool opena (const std::string &filename)
   Appends if exists.

template<class S, class ...Args>
bool opena (const S &format, const Args&... args)

bool isValid ()

void close ()

void setLowPrecision ()
   Only output floats.

template<typename T>
void addRepeat (T &value, std::string name)

template<typename T>
void addOnce (T &value, std::string name)

void add (int &i, const char *name, bool save_repeat = false)
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.

bool write (const std::string & filename) const
    Opens, writes, closes file.

template<typename S, typename ... Args>
bool write (const S & format, const Args &... args) const

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)

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_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
std::string filename
bool writable = {false}
bool appending = {false}
bool first_time = {true}
std::vector<VarStr<int>> int_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

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.
L.2.42 File dataformat.cxx

L.2.43 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

std::unique_ptr<DataFormat> data_format(const char *filename = nullptr)

class DataFormat
    Subclassed by NcFormat, PncFormat

Public Functions

DataFormat (Mesh *mesh_in = nullptr)

virtual ~DataFormat ()

virtual bool openr(const char *name) = 0

virtual bool openr(const std::string &name)

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)

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
virtual bool setGlobalOrigin(int x = 0, int y = 0, int z = 0) = 0
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 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(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 ly = 0, 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 ly = 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(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 ly = 0, 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 ly = 0, int lz = 0) = 0
virtual void setLowPrecision()
**setAttribute**

```cpp
virtual void setAttribute(const std::string &varname, const std::string &attrname, const std::string &text) = 0
```

Sets a string attribute

**Inputs**

**Parameters**

- `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.
- `attrname`: Attribute name
- `text`: A string attribute to attach to the variable

```cpp
virtual void setAttribute(const std::string &varname, const std::string &attrname, int value) = 0
```

Sets an integer attribute

**Inputs**

**Parameters**

- `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.
- `attrname`: Attribute name
- `value`: 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`: 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`: Attribute name
- `value`: 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**

**Parameters**

- `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.
- `attrname`: Attribute name
- `text`: A string attribute of the variable
virtual bool getAttribute(const std::string &varname, const std::string &attrname, int &value) = 0

Gets an integer attribute

Inputs

Returns

Parameters

- **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.
- **attrname**: Attribute name

value An int attribute of the variable

virtual bool getAttribute(const std::string &varname, const std::string &attrname, BoutReal &value) = 0

Gets a BoutReal attribute

Inputs

Returns

Parameters

- **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.
- **attrname**: Attribute name

value A BoutReal attribute of the variable

void writeFieldAttributes(const std::string &name, const Field &f)

Write out the meta-data of a field as attributes of the variable.

void writeFieldAttributes(const std::string &name, const FieldPerp &f)

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

L.2.44 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

const dcomplex Im(0, 1)
struct fcmplx
    #include <dcomplex.hxx> Complex type for passing data to/from FORTRAN.

    Public Members

    BoutReal r
    BoutReal i

L.2.45 File deprecated.hxx

Defines

DEPRECATED(func)
Mark functions for future removal
On gcc, expands to

    func __attribute__((deprecated)) ((deprecated))

Example

    class SomeClass {
    public:
        DEPRECATED(int someFunction(const string &input));
    }

L.2.46 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
BOUT++ Documentation, Release 5.0.0-alpha

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/.

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

template<>
using standardFunc = std::function<void (const FieldType&, FieldType&, const std::string& )>

template<>
using flowFunc = std::function<void (const FieldType&, const FieldType&, const FieldType&, const std::string& )>

template<>
using upwindFunc = flowFunc

template<>
using fluxFunc = flowFunc

template<>
using storageType = std::unordered_map<K, V>

Public Functions

DerivativeStore (const DerivativeStore &junk)

bool isEmpty () const
    Report if store has any registered methods.

bool isEmpty (std::size_t key) const
    Report if store has any registered methods for specific type determined by key.

bool isEmpty (DERIV derivType, DIRECTION direction, STAGGER stagger = STAGGER::None) const
    Report if store has any registered methods for specific type.

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.
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.

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.

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>
void registerDerivative (standardFunc func, Direction direction, Stagger stagger, Method method)

Templated versions of the above registration routines.

template<typename Direction, typename Stagger, typename Method>
void registerDerivative (upwindFunc func, Direction direction, Stagger stagger, Method method)

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.

standardFunc getStandard2ndDerivative (std::string name, DIRECTION direction, STAGGER stagger = STAGGER::None) const

standardFunc getStandard4thDerivative (std::string name, DIRECTION direction, STAGGER stagger = STAGGER::None) const

flowFunc getFlowDerivative (std::string name, DIRECTION direction, STAGGER stagger = STAGGER::None, DERIV derivType = DERIV::Upwind) const

upwindFunc getUpwindDerivative (std::string name, DIRECTION direction, STAGGER stagger = STAGGER::None) const

fluxFunc getFluxDerivative (std::string name, DIRECTION direction, STAGGER stagger = STAGGER::None) const

void initialise (Options *options)

void forceDefaultMethod (std::string methodName, DERIV deriv, DIRECTION direction, STAGGER stagger = STAGGER::None)

Provide a method to override/force a specific default method.

void clear ()

Empty all member storage.

void reset ()

Reset to initial state.

Public Static Functions

static DerivativeStore &getInstance ()
Private Functions

DerivativeStore()
void setDefaults()

std::string getMethodName(std::string name, DIRECTION direction, STAGGER stagger = STAGGER::None) const

std::string nameLookup(const std::string name, const std::string defaultValue) const

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>
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!

L.2.47 File derivs.cxx

Functions

Field3D DDX (const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DIFF_DEFAULT", const std::string &region = "RGN_NOBNDRY")
Calculate first partial derivative in X
\frac{\partial}{\partial x}

Parameters

• f: The field to be differentiated
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: The field to be differentiated
- outloc: 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: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- region: What region is expected to be calculated If not given, defaults to 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: The field to be differentiated
- outloc: 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: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- region: What region is expected to be calculated If not given, defaults to 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: The field to be differentiated
- outloc: 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: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- region: What region is expected to be calculated If not given, defaults to RGN_NOBNDRY
Field3D DDZ (const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DE-FAULT", const std::string &region = "RGN_NOBNDRY")
Calculate first partial derivative in $\frac{\partial}{\partial z}$

Parameters

- $f$: The field to be differentiated
- $\text{outloc}$: 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
- $\text{method}$: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- $\text{region}$: What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field2D DDZ (const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DE-FAULT", const std::string &region = "RGN_NOBNDRY")
Calculate first partial derivative in $\frac{\partial}{\partial z}$

Parameters

- $f$: The field to be differentiated
- $\text{outloc}$: 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
- $\text{method}$: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- $\text{region}$: What region is expected to be calculated If not given, defaults to 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 $\frac{\partial}{\partial z}$

Parameters

- $f$: The field to be differentiated
- $\text{outloc}$: 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
- $\text{method}$: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- $\text{region}$: What region is expected to be calculated If not given, defaults to 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 $\frac{\partial}{\partial z}$

Parameters

- $f$: The field to be differentiated
• **outloc**: 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**: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
• **region**: What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

**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**: The field to be differentiated
• **outloc**: 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**: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
• **region**: What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

**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**: The field to be differentiated
• **outloc**: 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**: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
• **region**: What region is expected to be calculated If not given, defaults to 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
\[ \frac{\partial^2}{\partial y^2} \]

**Parameters**
• **f**: The field to be differentiated
• **outloc**: 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**: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
• **region**: What region is expected to be calculated If not given, defaults to RGN_NOBNDRY
Field2D \texttt{D2DY2} (\texttt{const Field2D} &f, \texttt{CELL\_LOC} outloc = \texttt{CELL\_DEFAULT}, \texttt{const std::string} &method = "DEFAULT", \texttt{const std::string} &region = "RGN\_NOBNDRY")

Calculate second partial derivative in Y
\( \frac{\partial^2}{\partial y^2} \)

**Parameters**

- \( f \): The field to be differentiated
- \( \texttt{outloc} \): The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to \texttt{CELL\_DEFAULT}
- \( \texttt{method} \): Differencing method to use. This overrides the default If not given, defaults to \texttt{DIFF\_DEFAULT}
- \( \texttt{region} \): What region is expected to be calculated If not given, defaults to \texttt{RGN\_NOBNDRY}

Field3D \texttt{D2DZ2} (\texttt{const Field3D} &f, \texttt{CELL\_LOC} outloc = \texttt{CELL\_DEFAULT}, \texttt{const std::string} &method = "DEFAULT", \texttt{const std::string} &region = "RGN\_NOBNDRY")

Calculate second partial derivative in Z
\( \frac{\partial^2}{\partial z^2} \)

**Parameters**

- \( f \): The field to be differentiated
- \( \texttt{outloc} \): The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to \texttt{CELL\_DEFAULT}
- \( \texttt{method} \): Differencing method to use. This overrides the default If not given, defaults to \texttt{DIFF\_DEFAULT}
- \( \texttt{region} \): What region is expected to be calculated If not given, defaults to \texttt{RGN\_NOBNDRY}

Field2D \texttt{D2DZ2} (\texttt{const Field2D} &f, \texttt{CELL\_LOC} outloc = \texttt{CELL\_DEFAULT}, \texttt{const std::string} &method = "DEFAULT", \texttt{const std::string} &region = "RGN\_NOBNDRY")

Calculate second partial derivative in Z
\( \frac{\partial^2}{\partial z^2} \)

**Parameters**

- \( f \): The field to be differentiated
- \( \texttt{outloc} \): The cell location where the result is desired. If staggered grids is not enabled then this has no effect If not given, defaults to \texttt{CELL\_DEFAULT}
- \( \texttt{method} \): Differencing method to use. This overrides the default If not given, defaults to \texttt{DIFF\_DEFAULT}
- \( \texttt{region} \): What region is expected to be calculated If not given, defaults to \texttt{RGN\_NOBNDRY}

Field3D \texttt{D4DX4} (\texttt{const Field3D} &f, \texttt{CELL\_LOC} outloc = \texttt{CELL\_DEFAULT}, \texttt{const std::string} &method = "DEFAULT", \texttt{const std::string} &region = "RGN\_NOBNDRY")

Calculate forth partial derivative in X
\( \frac{\partial^4}{\partial x^4} \)

**Parameters**

- \( f \): The field to be differentiated

L.2. File list 367
• **outloc**: 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**: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

• **region**: What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

```cpp
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**: The field to be differentiated

- **outloc**: 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**: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

- **region**: What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

```cpp
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**: The field to be differentiated

- **outloc**: 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**: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

- **region**: What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

```cpp
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**: The field to be differentiated

- **outloc**: 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**: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

- **region**: What region is expected to be calculated If not given, defaults to RGN_NOBNDRY
Field3D D4D24 (const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

Calculate forth partial derivative in Z
\[ \frac{\partial^4}{\partial z^4} \]

Parameters
- \( f \): The field to be differentiated
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field2D D4D24 (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
\[ \frac{\partial^4}{\partial z^4} \]

Parameters
- \( f \): The field to be differentiated
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): 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, const std::string &dfdy_region)

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, const std::string &dfdy_region)

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 = 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
- \( f \): The field to be differentiated
- \( \text{outloc} \): 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
\textbf{Field3D} \texttt{D2DXDZ} (const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

X-Z mixed derivative.

\textbf{Parameters}

- \textit{f}: The field to be differentiated
- \textit{outloc}: 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
- \textit{method}: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \textit{region}: What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

\textbf{Field2D} \texttt{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

\[ \frac{\partial^2}{\partial y \partial z} \]

\textbf{Parameters}

- \textit{f}: The field to be differentiated
- \textit{outloc}: 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
- \textit{method}: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \textit{region}: What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

\textbf{Field3D} \texttt{D2DYDZ} (const Field3D &f, CELL_LOC outloc, MAYBE_UNUSED) const std::string &method

Special case where both arguments are 2D. \textit{Output} location ignored for now.

\textbf{Field2D} \texttt{VDDX} (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 \( v \cdot \text{grad}(f) \)

\[ v \cdot \frac{\partial f}{\partial x} \]

\textbf{Parameters}

- \textit{v}: The velocity field
- \textit{f}: The field of the advected quantity
- \textit{outloc}: 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: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
• region: What region is expected to be calculated If not given, defaults to 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

• $v$: The velocity field
• $f$: The field of the advected quantity
• outloc: 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: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
• region: What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

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 $v \cdot \nabla f$

$v \cdot \frac{\partial f}{\partial y}$

Parameters

• $v$: The velocity field
• $f$: The field of the advected quantity
• outloc: 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: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
• region: 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 = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

For terms of form $v \cdot \nabla f$

$v \cdot \frac{\partial f}{\partial y}$

Parameters

• $v$: The velocity field
• $f$: The field of the advected quantity
• outloc: 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: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
• region: 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 = 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: The velocity field
• f: The field of the advected quantity
• outloc: 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: Differencing method to use. This overrides the default. If not given, defaults to DIFF_DEFAULT
• region: 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 = 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: The velocity field
• f: The field of the advected quantity
• outloc: 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: Differencing method to use. This overrides the default. If not given, defaults to DIFF_DEFAULT
• region: 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 = 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: The velocity field
• f: The field of the advected quantity
• outloc: 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: Differencing method to use. This overrides the default. If not given, defaults to DIFF_DEFAULT
• region: 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 = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

for terms of form \( \partial (vf) / \partial x \)

Parameters

- \( v \): The velocity field
- \( f \): The field of the advected quantity
- \( outloc \): 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
- \( method \): Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( region \): 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 = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

for terms of form \( \partial (vf) / \partial x \)

Parameters

- \( v \): The velocity field
- \( f \): The field of the advected quantity
- \( outloc \): 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
- \( method \): Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( region \): 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 = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

for terms of form \( \partial (vf) / \partial y \)

Parameters

- \( v \): The velocity field
- \( f \): The field of the advected quantity
- \( outloc \): 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
- \( method \): Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( region \): 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 = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

for terms of form \( \partial (vf) / \partial y \)
\[ \partial (vf) / \partial y \]

**Parameters**

- \( v \): The velocity field
- \( f \): The field of the advected quantity
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default. If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): What region is expected to be calculated. If not given, defaults to RGN_NOBNDRY

```cpp
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 \( \text{div}(v * f) \)

\[ \partial (vf) / \partial z \]

**Parameters**

- \( v \): The velocity field
- \( f \): The field of the advected quantity
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default. If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): What region is expected to be calculated. If not given, defaults to RGN_NOBNDRY

```cpp
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 \( \text{div}(v * f) \)

\[ \partial (vf) / \partial \zeta \]

**Parameters**

- \( v \): The velocity field
- \( f \): The field of the advected quantity
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default. If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): What region is expected to be calculated. If not given, defaults to RGN_NOBNDRY

**L.2.48 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 $\frac{\partial}{\partial x}$

**Parameters**

- f: The field to be differentiated
- outloc: 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: Differencing method to use. This overrides the default If not given, defaults to `DIFF_DEFAULT`
- region: What region is expected to be calculated If not given, defaults to `RGN_NOBNDRY`

`Field3D DDX` (const `Field3D` &f, `CELL_LOC` outloc, const std::string &method, `REGION` region)

`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 $\frac{\partial}{\partial x}$

**Parameters**

- f: The field to be differentiated
- outloc: 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: Differencing method to use. This overrides the default If not given, defaults to `DIFF_DEFAULT`
- region: 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, REGION region)
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 = "DE-FAULT", const std::string &region = "RGN_NOBNDRY")
Calculate first partial derivative in Y
\[ \frac{\partial}{\partial y} \]

Parameters
- \( f \): The field to be differentiated
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field3D DDY(const Field3D &f, CELL_LOC outloc, const std::string &method, REGION region)
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 = "DE-FAULT", const std::string &region = "RGN_NOBNDRY")
Calculate first partial derivative in Y
\[ \frac{\partial}{\partial y} \]

Parameters
- \( f \): The field to be differentiated
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field2D DDY(const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)
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 = "DE-FAULT", const std::string &region = "RGN_NOBNDRY")
Calculate first partial derivative in Z
\[ \frac{\partial}{\partial z} \]

Parameters
- \( f \): The field to be differentiated
- \( \text{outloc} \): 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**: Differencing method to use. This overrides the default. If not given, defaults to DIFF_DEFAULT

• **region**: What region is expected to be calculated. If not given, defaults to RGN_NOBNDRY

Field3D DDZ(const Field3D &f, CELL_LOC outloc, const std::string &method, REGION region)

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**: The field to be differentiated

• **outloc**: 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**: Differencing method to use. This overrides the default. If not given, defaults to DIFF_DEFAULT

• **region**: 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, REGION region)

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**: The field to be differentiated

• **outloc**: 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**: Differencing method to use. This overrides the default. If not given, defaults to DIFF_DEFAULT

• **region**: What region is expected to be calculated. If not given, defaults to RGN_NOBNDRY

Vector3D DDZ(const Vector3D &f, CELL_LOC outloc, const std::string &method, REGION region)

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**: The field to be differentiated
• outloc: 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: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
• region: What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Vector2D DDZ(const Vector2D &f, CELL_LOC outloc, const std::string &method, REGION region)
Vector2D DDZ(const Vector2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)
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: The field to be differentiated
• outloc: 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: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
• region: What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field3D D2DX2(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
Field3D D2DX2(const Field3D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)
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: The field to be differentiated
• outloc: 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: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
• region: 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, REGION region)
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
\( \frac{\partial^2}{\partial y^2} \)
Parameters

- \( f \): The field to be differentiated
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field3D \text{D2DY2} (\text{const Field3D} &f, \text{CELL_LOC} \text{outloc}, \text{const} std::string \&\text{method}, \text{REGION} \text{region})

Field3D \text{D2DY2} (\text{const Field3D} &f, \text{CELL_LOC} \text{outloc}, \text{DIFF_METHOD} \text{method}, \text{REGION} \text{region} = RGN_NOBNDRY)

Field2D \text{D2DY2} (\text{const Field2D} &f, \text{CELL_LOC} \text{outloc} = \text{CELL_DEFAULT}, \text{const} std::string \&\text{method} = "DEFAULT", \text{const} std::string \&\text{region} = "RGN_NOBNDRY")

Calculate second partial derivative in \( Y \)

\( \partial^2 / \partial y^2 \)

Parameters

- \( f \): The field to be differentiated
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field2D \text{D2DY2} (\text{const Field2D} &f, \text{CELL_LOC} \text{outloc}, \text{const} std::string \&\text{method}, \text{REGION} \text{region})

Field2D \text{D2DY2} (\text{const Field2D} &f, \text{CELL_LOC} \text{outloc}, \text{DIFF_METHOD} \text{method}, \text{REGION} \text{region} = RGN_NOBNDRY)

Field3D \text{D2DZ2} (\text{const Field3D} &f, \text{CELL_LOC} \text{outloc} = \text{CELL_DEFAULT}, \text{const} std::string \&\text{method} = "DEFAULT", \text{const} std::string \&\text{region} = "RGN_NOBNDRY")

Calculate second partial derivative in \( Z \)

\( \partial^2 / \partial z^2 \)

Parameters

- \( f \): The field to be differentiated
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field3D \text{D2DZ2} (\text{const Field3D} &f, \text{CELL_LOC} \text{outloc}, \text{const} std::string \&\text{method}, \text{REGION} \text{region})

Field3D \text{D2DZ2} (\text{const Field3D} &f, \text{CELL_LOC} \text{outloc}, \text{DIFF_METHOD} \text{method}, \text{REGION} \text{region} = RGN_NOBNDRY)
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
\[ \frac{\partial^2 f}{\partial z^2} \]

Parameters
- \( f \): The field to be differentiated
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): 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, REGION region)
Field2D D2DZ2 (const Field2D & f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)
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
\[ \frac{\partial^4 f}{\partial x^4} \]

Parameters
- \( f \): The field to be differentiated
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): 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, REGION region)
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
\[ \frac{\partial^4 f}{\partial x^4} \]

Parameters
- \( f \): The field to be differentiated
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): 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, REGION region)

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
\[ \partial^4 / \partial y^4 \]

Parameters

- **f**: The field to be differentiated
- **outloc**: 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**: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- **region**: 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, REGION region)

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
\[ \partial^4 / \partial y^4 \]

Parameters

- **f**: The field to be differentiated
- **outloc**: 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**: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- **region**: 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, REGION region)

Field2D D4DY4 (const Field2D & f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field3D D4DZ4 (const Field3D & 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**: The field to be differentiated
- **outloc**: 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**: Differencing method to use. This overrides the default. If not given, defaults to `DIFF_DEFAULT`

• **region**: 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, `REGION` region)

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

\[ \frac{\partial^4}{\partial z^4} \]

**Parameters**

- `f`: The field to be differentiated
- `outloc`: 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`: Differencing method to use. This overrides the default. If not given, defaults to `DIFF_DEFAULT`
- `region`: 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, `REGION` region)

Field2D D4DZ4 (const `Field2D` &f, `CELL_LOC` outloc, `DIFF_METHOD` method, `REGION` region = `RGN_NOBNDRY`)

Field3D VDX (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 \( v \cdot \frac{\partial f}{\partial x} \)

\[ v \cdot \frac{\partial f}{\partial x} \]

**Parameters**

- `v`: The velocity field
- `f`: The field of the advected quantity
- `outloc`: 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`: Differencing method to use. This overrides the default. If not given, defaults to `DIFF_DEFAULT`
- `region`: What region is expected to be calculated. If not given, defaults to `RGN_NOBNDRY`

### Field3D VDX

(const `Field3D` &v, const `Field3D` &f, `CELL_LOC` outloc, const std::string &method, `REGION` region)

Field3D VDX (const `Field3D` &v, const `Field3D` &f, `CELL_LOC` outloc, `DIFF_METHOD` method, `REGION` region = `RGN_NOBNDRY`)

Field2D VDX (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**

- $v$: The velocity field
- $f$: The field of the advected quantity
- `outloc`: 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`: Differencing method to use. This overrides the default If not given, defaults to `DIFF_DEFAULT`
- `region`: What region is expected to be calculated If not given, defaults to `RGN_NOBNDRY`

```cpp
Field2D VDX(const Field2D &v, const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)
```

```cpp
Field2D VDX(const Field2D &v, const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)
```

```cpp
Field3D VDY(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 $v \cdot \nabla f$

$v \cdot \frac{\partial f}{\partial y}$

**Parameters**

- $v$: The velocity field
- $f$: The field of the advected quantity
- `outloc`: 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`: Differencing method to use. This overrides the default If not given, defaults to `DIFF_DEFAULT`
- `region`: What region is expected to be calculated If not given, defaults to `RGN_NOBNDRY`

```cpp
Field3D VDY(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 $v \cdot \nabla f$

$v \cdot \frac{\partial f}{\partial y}$

**Parameters**

- $v$: The velocity field
- $f$: The field of the advected quantity
- `outloc`: 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: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT

region: What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field2D VDDY(const Field2D &v, const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)
Field2D VDDY(const Field2D &v, const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)
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 $v \cdot \frac{\partial f}{\partial z}$

Parameters

• v: The velocity field
• f: The field of the advected quantity
• outloc: 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: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
• region: 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, 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 $v \cdot \frac{\partial f}{\partial z}$

Parameters

• v: The velocity field
• f: The field of the advected quantity
• outloc: 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: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
• region: 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, REGION region)
Field2D VDDZ(const Field2D &v, const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")
Field2D `VDDZ` (const Field3D &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

- \( v \): The velocity field
- \( f \): The field of the advected quantity
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default. If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): 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, REGION region)

Field2D `VDDZ` (const Field3D &v, const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field3D `FDDX` (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 \( \text{div}(v \cdot f) \)

\[ \frac{\partial (vf)}{\partial x} \]

Parameters

- \( v \): The velocity field
- \( f \): The field of the advected quantity
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default. If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): 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, REGION region)

Field3D `FDDX` (const Field3D &v, const Field3D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field2D `FDDX` (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 \( \text{div}(v \cdot f) \)

\[ \frac{\partial (vf)}{\partial x} \]

Parameters

- \( v \): The velocity field
- \( f \): The field of the advected quantity
• **outloc**: 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**: Differencing method to use. This overrides the default. If not given, defaults to `DIFF_DEFAULT`.
• **region**: 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, REGION region)`

`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 \( \frac{\partial (vf)}{\partial y} \)

**Parameters**

• **v**: The velocity field
• **f**: The field of the advected quantity
• **outloc**: 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**: Differencing method to use. This overrides the default. If not given, defaults to `DIFF_DEFAULT`.
• **region**: 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, REGION region)`

`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 \( \frac{\partial (vf)}{\partial y} \)

**Parameters**

• **v**: The velocity field
• **f**: The field of the advected quantity
• **outloc**: 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**: Differencing method to use. This overrides the default. If not given, defaults to `DIFF_DEFAULT`.
• **region**: 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, REGION region)`

`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)
\[ \frac{\partial (vf)}{\partial z} \]

Parameters
- \( v \): The velocity field
- \( f \): The field of the advected quantity
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): 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, REGION region)

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)
\[ \frac{\partial (vf)}{\partial z} \]

Parameters
- \( v \): The velocity field
- \( f \): The field of the advected quantity
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): 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, REGION region)

Field2D FDDZ (const Field2D &v, const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

Field3D D2DXDY (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", const std::string &dfdy_region = "")

Calculate mixed partial derivative in x and y
\[ \frac{\partial^2 f}{\partial x \partial y} \]

Mixed derivative in X and Y

Parameters
- \( f \): The field to be differentiated
• **outloc**: 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**: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
• **region**: What region is expected to be calculated If not given, defaults to RGN_NOBNDRY
• **dfdy_boundary_condition**: Boundary condition to use to set the guard cells of df/dy, before calculating the x-derivative.
• **dfdy_region**: Region in which to calculate df/dy. If an empty string (default) then the same as the region for the calculation as a whole. If dfdy_region < region in size then this will cause errors.

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, REGION region, const std::string &dfdy_boundary_condition = "free_o3", const std::string &dfdy_region = ")

Field3D D2DXDY(const Field3D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY, const std::string &dfdy_boundary_condition = "free_o3", const std::string &dfdy_region = ")

Field2D D2DXDY(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", const std::string &dfdy_region = ")

Calculate mixed partial derivative in x and y
\[ \frac{\partial^2}{\partial x \partial y} \]

Mixed derivative in X and Y

** Parameters

• **f**: The field to be differentiated
• **outloc**: 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**: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
• **region**: What region is expected to be calculated If not given, defaults to RGN_NOBNDRY
• **dfdy_boundary_condition**: Boundary condition to use to set the guard cells of df/dy, before calculating the x-derivative.
• **dfdy_region**: Region in which to calculate df/dy. If an empty string (default) then the same as the region for the calculation as a whole. If dfdy_region < region in size then this will cause errors.

This first takes derivatives in Y, then in X.

** Communicates and applies boundary in X.

Field2D D2DXDY(const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region, const std::string &dfdy_boundary_condition = "free_o3", const std::string &dfdy_region = ")

Field2D D2DXDY(const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY, const std::string &dfdy_boundary_condition = "free_o3", const std::string &dfdy_region = ")
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

- \( f \): The field to be differentiated
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): 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, REGION region)
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

- \( f \): The field to be differentiated
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): What region is expected to be calculated If not given, defaults to RGN_NOBNDRY

Field2D D2DXDZ (const Field2D &f, CELL_LOC outloc, const std::string &method, REGION region)
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

- \( f \): The field to be differentiated
- \( \text{outloc} \): 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
- \( \text{method} \): Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- \( \text{region} \): What region is expected to be calculated If not given, defaults to RGN_NOBNDRY
Field3D D2DYDZ (const Field3D &f, CELL_LOC outloc, const std::string &method, REGION region)

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: The field to be differentiated
- outloc: 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: Differencing method to use. This overrides the default If not given, defaults to DIFF_DEFAULT
- region: 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, REGION region)

Field2D D2DYDZ (const Field2D &f, CELL_LOC outloc, DIFF_METHOD method, REGION region = RGN_NOBNDRY)

L.2.49 File difops.cxx

Functions

const 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: The field to be differentiated
- outloc: The cell location where the output is needed (if staggered grids is enabled)
- method: The method to use. The default is set in the options.

const Field2D Grad_par (const Field2D &var, const std::string &method, CELL_LOC outloc)
const Field3D Grad_par (const Field3D &var, CELL_LOC outloc, const std::string &method)
const Field3D Grad_par (const Field3D &var, const std::string &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**

- \( v \): The velocity in \( y \) direction
- \( f \): The scalar field to be differentiated
- \( \text{outloc} \): The cell location of the output. By default this is the same as \( f \)
- \( \text{method} \): 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)
const Field3D Vpar_Grad_par (const Field3D &v, const Field3D &f, CELL_LOC outloc, const std::string &method)
const Field3D Vpar_Grad_par (const Field3D &v, const Field3D &f, const std::string &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**

- \( f \): The component of a vector along the magnetic field
- \( \text{outloc} \): The cell location for the result. By default the same as \( f \)
- \( \text{method} \): The numerical method to use

```cpp
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_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**

- \( f \): The field to be differentiated
• `outloc`: The cell location of the result

```cpp
const Field3D Grad2_par2 (const Field3D &f, CELL_LOC outloc, const std::string &method)
```

```cpp
const Field2D Div_par_K_Grad_par (BoutReal kY, const Field2D &f, CELL_LOC outloc = CELL_DEFAULT)
```

Parallel divergence of diffusive flux, $K \cdot \nabla \cdot (b_0 \cdot \nabla f)$

**Parameters**

- `kY`: The diffusion coefficient
- `f`: The field whose gradient drives a flux

```cpp
const Field3D Div_par_K_Grad_par (BoutReal kY, const Field3D &f, CELL_LOC outloc)
```

```cpp
const Field2D Div_par_K_Grad_par (const Field2D &kY, const Field2D &f, CELL_LOC outloc)
```

```cpp
const Field3D Div_par_K_Grad_par (const Field2D &kY, const Field3D &f, CELL_LOC outloc)
```

```cpp
const Field3D Div_par_K_Grad_par (const Field3D &kY, const Field2D &f, CELL_LOC outloc)
```

```cpp
const Field3D Div_par_K_Grad_par (const Field3D &kY, const Field3D &f, CELL_LOC outloc)
```

```cpp
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`

```cpp
const Field3D Delp2 (const Field3D &f, CELL_LOC outloc, bool useFFT)
```

```cpp
const FieldPerp Delp2 (const FieldPerp &f, CELL_LOC outloc, bool useFFT)
```

```cpp
const Field2D Laplace_perp (const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &dfdy_boundary_condition = "free_o3", const std::string &dfdy_region = "")
```

Perpendicular Laplacian, keeping y derivatives

```cpp
const Field3D Laplace_perp (const Field3D &f, CELL_LOC outloc, const std::string &dfdy_boundary_condition, const std::string &dfdy_region)
```

```cpp
const Field2D Laplace_par (const Field2D &f, CELL_LOC outloc = CELL_DEFAULT)
```

Parallel Laplacian operator

```cpp
const Field3D Laplace_par (const Field3D &f, CELL_LOC outloc)
```

```cpp
const Field2D Laplace (const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &dfdy_boundary_condition = "free_o3", const std::string &dfdy_region = ")
```

Full Laplacian operator (par + perp)

```cpp
const Field3D Laplace (const Field3D &f, CELL_LOC outloc, const std::string &dfdy_boundary_condition, const std::string &dfdy_region)
```

```cpp
const Field2D Laplace_perpXY (const Field2D &A, const Field2D &f)
```

Inverse of Laplacian operator in `LaplaceXY` solver

```cpp
const Field2D b0xGrad_dot_Grad (const Field2D &phi, const Field2D &A, CELL_LOC outloc = CELL_DEFAULT)
```

Terms of form $b_0 \cdot \nabla \phi \cdot \nabla A$
const Field3D b0xGrad_dot_Grad (const Field2D &phi, const Field3D &A, CELL_LOC outloc)

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

- \( \phi \): The scalar potential
- \( A \): The field being advected
- \( \text{outloc} \): 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 = 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

- \( f \): The potential
- \( g \): The field being advected
- \( \text{method} \): The method to use
- \( \text{outloc} \): The cell location where the result is defined. Default is the same as \( g \)
- \( \text{solver} \): 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)

L.2.50 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
Enums

enum BRACKET_METHOD
   Poisson bracket methods
   Values:
   standard
   Use b0xGrad_dot_Grad.
   simple
   Keep only terms in X-Z.
   arakawa
   Arakawa method in X-Z (optimised)
   ctu
   Corner Transport Upwind (CTU) method. Explicit method only, needs the timestep from the solver
   arakawa_old
   Older version, for regression testing of optimised version.

Functions

const 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: The field to be differentiated
• outloc: The cell location where the output is needed (if staggered grids is enabled)
• method: The method to use. The default is set in the options.

const Field2D Grad_par(const Field2D &var, const std::string &method, CELL_LOC outloc = CELL_DEFAULT)
const Field2D Grad_par(const Field2D &var, CELL_LOC outloc, DIFF_METHOD method)
const Field2D Grad_par(const Field2D &var, DIFF_METHOD method, CELL_LOC outloc)
const Field3D Grad_par(const Field3D &var, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")
const Field3D Grad_par(const Field3D &var, const std::string &method, CELL_LOC outloc = CELL_DEFAULT)

const Field3D Grad_par(const Field3D &var, CELL_LOC outloc, DIFF_METHOD method)

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

\[ b_0 \cdot \nabla \cdot \left( \frac{1}{B} b_0 \times \nabla \right. \) 

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

- \( v \): The velocity in y direction
- \( f \): The scalar field to be differentiated
- \( \text{outloc} \): The cell location of the output. By default this is the same as \( f \)
- \( \text{method} \): The numerical method to use. The default is set in the options

const Field2D Vpar_Grad_par(const Field2D &v, const Field2D &f, DIFF_METHOD method)

const Field2D Vpar_Grad_par(const Field2D &v, const Field2D &f, CELL_LOC outloc, DIFF_METHOD method)

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, CELL_LOC outloc = CELL_DEFAULT)

const Field3D Vpar_Grad_par(const Field3D &v, const Field3D &f, CELL_LOC outloc, DIFF_METHOD method)

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

\[ B \partial_i \left( \frac{f}{B} \right) = B \nabla \cdot \left( \frac{bf}{B} \right) \]

Parameters

- \( f \): The component of a vector along the magnetic field
- \( \text{outloc} \): The cell location for the result. By default the same as \( f \)
• method: The numerical method to use

```cpp
const Field2D Div_par(const Field2D &f, const std::string &method, CELL_LOC outloc = CELL_DEFAULT)
const Field2D Div_par(const Field2D &f, CELL_LOC outloc, DIFF_METHOD method)
const Field2D Div_par(const Field2D &f, DIFF_METHOD method, CELL_LOC outloc)
const Field3D Div_par(const Field3D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")
const Field3D Div_par(const Field3D &f, const std::string &method, CELL_LOC outloc = CELL_DEFAULT)
const Field3D Div_par(const Field3D &f, CELL_LOC outloc, DIFF_METHOD method)
const Field3D Div_par(const Field3D &f, DIFF_METHOD method, CELL_LOC outloc)
const Field2D Div_par(const Field2D &f, const Field2D &v)
const Field3D Div_par_flux(const Field3D &v, const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")
const Field3D Div_par_flux(const Field3D &v, const Field3D &f, const std::string &method, CELL_LOC outloc = CELL_DEFAULT)
const Field3D Div_par_flux(const Field3D &v, const Field3D &f, DIFF_METHOD method)
const Field3D Div_par_flux(const Field3D &v, const Field3D &f, DIFF_METHOD method, CELL_LOC outloc = CELL_DEFAULT)
const Field2D Grad2_par2(const Field2D &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT")
second parallel derivative
(bdot\n)(bdot\n)
```

Note: For parallel Laplacian use LaplacePar

**Parameters**

- **f**: The field to be differentiated
- **outloc**: The cell location of the result

```cpp
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")
const Field3D Grad2_par2(const Field3D &f, CELL_LOC outloc, DIFF_METHOD method)
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
const Field2D Grad_par_CtoL(const Field2D &var)
const Field3D Vpar_Grad_par_LCtoC(const Field3D &v, const Field3D &f, const std::string &region = "RGN_NOBDRY")
const Field3D Vpar_Grad_par_LCtoC(const Field3D &v, const Field3D &f, REGION region = RGN_NOBDRY)
const Field3D Grad_par_LtoC(const Field3D &var)
```
const Field2D Grad_par_LtoC(const Field2D &var)
const Field3D Div_par_LtoC(const Field3D &var)
const Field2D Div_par_LtoC(const Field2D &var)
const Field3D Div_par_CtoL(const Field3D &var)
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

\[ \nabla \cdot (b_0 kY (b_0 \cdot \nabla) f) \n\]

Parameters

• kY: The diffusion coefficient
• f: 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, const std::string & dfdy_boundary_condition = "free_o3", const std::string & dfdy_region = "")
        Perpendicular Laplacian, keeping y derivatives

const Field3D Laplace_perp(const Field3D & f, CELL_LOC outloc = CELL_DEFAULT, const std::string & dfdy_boundary_condition = "free_o3", const std::string & dfdy_region = "")
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, const std::string & dfdy_boundary_condition = "free_o3", const std::string & dfdy_region = "")

Full Laplacian operator (par + perp)

const Field3D Laplace (const Field3D & f, CELL_LOC outloc = CELL_DEFAULT, const std::string & dfdy_boundary_condition = "free_o3", const std::string & dfdy_region = "")

const Field2D Laplace_perpXY (const Field2D & A, const Field2D & f)

Inverse of Laplacian operator in LaplaceXY solver

const Field3D 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

- \( \phi \): The scalar potential
- \( A \): The field being advected
- \( \text{outloc} \): 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 perp terms, which can be cast as antisymmetric Poisson brackets

\[ [f, g] = (1/B)b_0 \times \nabla f \cdot \nabla g \]

Parameters

- \( f \): The potential
- \( g \): The field being advected
- \( \text{method} \): The method to use
- \( \text{outloc} \): The cell location where the result is defined. Default is the same as \( g \)
- \( \text{solver} \): 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 Field2D & 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

L.2.51 File emptyformat.hxx

class EmptyFormat

    Private Functions

    EmptyFormat ()
    bool openr (const std::string &name)
    bool openw (const std::string &name, bool append)
    bool is_valid ()
    void close ()
    const std::vector<int> getSize (const char *var)
    const std::vector<int> getSize (const std::string &var)
    bool setOrigin (int x = 0, int y = 0, int z = 0)
    bool setRecord (int t)
    bool read (int *var, const char *name, int lx = 1, int ly = 0, int lz = 0)
    bool read (int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0)
    bool read (BoutReal *var, const char *name, int lx = 1, int ly = 0, int lz = 0)
    bool read (BoutReal *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0)
    bool write (int *var, const char *name, int lx = 0, int ly = 0, int lz = 0)
    bool write (int *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0)
    bool write (BoutReal *var, const char *name, int lx = 0, int ly = 0, int lz = 0)
    bool write (BoutReal *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0)
    bool read_rec (int *var, const char *name, int lx = 1, int ly = 0, int lz = 0)
    bool read_rec (int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0)
bool read_rec (int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0)
bool read_rec (BoutReal *var, const char *name, int lx = 1, int ly = 0, int lz = 0)
bool read_rec (BoutReal *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0)
bool write_rec (int *var, const char *name, int lx = 0, int ly = 0, int lz = 0)
bool write_rec (int *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0)
bool write_rec (BoutReal *var, const char *name, int lx = 0, int ly = 0, int lz = 0)
bool write_rec (BoutReal *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0)

L.2.52 File euler.cxx

L.2.53 File euler.hxx

class EulerSolver : public Solver

Public Functions

EulerSolver (Options *options)

~EulerSolver ()

void setMaxTimestep (BoutReal dt)

Set a maximum internal timestep (only for explicit schemes)

BoutReal getCurrentTimestep ()

Return the current internal timestep.

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

int run ()

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

L.2.54 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>::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

    Literal (BoutReal v)

    ~Literal ()

    BoutReal operator() (int x, int y, int z) const

Private Members

    const BoutReal val

class Field3DEexpr
Public Types

using type = Field3D

Public Functions

Field3DExpr(const Field3D &f)
const BoutReal &operator() (int x, int y, int z) const

Private Members

const BoutReal *data

class Field2DExpr

Public Types

using type = Field2D

Public Functions

Field2DExpr(const Field2D &f)
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

template<>
using expr_type = ExprT

template<>
struct exprTraits<double>

Public Types

template<>
using expr_type = Literal

template<>
struct exprTraits<float>
Public Types

template<>
using expr_type = Literal

template<>
struct exprTraits<int>

Public Types

template<>
using expr_type = Literal

template<typename T>
struct asExpr

Public Types

template<>
using type = T

Public Static Functions

static const T &getExpr(const T &x)

template<>
struct asExpr<int>

Public Types

template<>
using type = Literal

Public Static Functions

static const Literal getExpr(const int &x)

template<>
struct asExpr<double>

Public Types

template<>
using type = Literal

Public Static Functions

static const Literal getExpr(const double &x)

template<>
struct asExpr<float>
Public Types

```cpp
template<>
using type = Literal
```

Public Static Functions

```cpp
static const Literal getExpr(const float &x)
```

```cpp
struct asExpr<Field3D>
```

Public Types

```cpp
template<>
using type = Field3DEexpr
```

Public Static Functions

```cpp
static const Field3DEexpr getExpr(const Field3D &x)
```

```cpp
struct PromoteType
```

Public Types

```cpp
template<>
using type = Field3D
```

```cpp
template<class ExprT1, class ExprT2, class BinOp>
class BinaryExpr
```

```cpp
using ltype = typename exprTraits<ExprT1>::expr_type
```

```cpp
using rtype = typename exprTraits<ExprT2>::expr_type
```

```cpp
using type = typename PromoteType::type
```

Type of the resulting expression.

Public Functions

```cpp
BinaryExpr(const ExprT1 &e1, const ExprT2 &e2)
```

```cpp
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

template<>
using arg1 = typename asExpr<ExprT1>::type
template<>
using arg2 = typename asExpr<ExprT2>::type
template<>
using type = BinaryExpr<arg1, arg2, name>

struct Add

Public Static Functions

template<typename T>
static T apply(T a, T b)

struct Subtract

Public Static Functions

template<typename T>
static T apply(T a, T b)

struct Multiply

Public Static Functions

template<typename T>
static T apply(T a, T b)

struct Divide

Public Static Functions

template<typename T>
static T apply(T a, T b)

struct Power

Public Static Functions

template<typename T>
static T apply(T a, T b)
L.2.55 File expressionparser.cxx

L.2.56 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 FieldATan, FieldBallooning, FieldBinary, FieldClamp, FieldFunction, FieldGaussian, FieldGenOneArg< Op >, FieldGenTwoArg< Op >, FieldHeaviside, FieldMax, FieldMin, FieldMixmode, FieldNull, FieldRound, FieldTanhHat, FieldValue, FieldValuePtr, FieldWhere

Public Functions

  virtual ~FieldGenerator()  // Virtual destructor

  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: A (possibly empty) list of arguments to the generator function

  virtual double generate(BoutReal x, BoutReal y, BoutReal z, BoutReal t)  // Note: This will be removed in a future version. Implementations should override the Context version of this function.

  double generate(const bout::generator::Context &ctx)  // Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs.

  Note: The default implementations of generate call each other; the implementor of a FieldGenerator type must implement one of them or an infinite recursion results. This is for backward compatibility for users and implementors. In a future version this function will be made pure virtual.

```
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()

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: The name to be recognised in expressions. This should start with a letter and contain no whitespace, only alphanumeric letters and underscores.
• g: 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 +,-,*,/,

Parameters

• sym: The operator symbol. This must be a single character
• b: The FieldGenerator to use. When the symbol is recognised, b->clone() will be called with two input arguments
• precedence: 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 *, / precedence = 20 ^ precedence = 30

Protected Functions

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 without escaping; 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 parseContextExpr (LexInfo &lex) const
    Context definition
    Returns a pointer to a FieldContext object.
    Matches [ symbol = expression , symbol = expression . . . ] ( expression )

FieldGeneratorPtr parsePrimary (LexInfo &lex) const
    Parse a primary expression, one of:
    • number
    • identifier
    • ( . . . ) parenexpr
    • [ . . . ]() context
    • 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 symbol, -3 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

**FieldBinary** (*FieldGeneratorPtr l, FieldGeneratorPtr r, char o*)

**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**: A (possibly empty) list of arguments to the generator function

**BoutReal generate(const bout::generator::Context &ctx)**
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

Note: The default implementations of generate call each other; the implementor of a **FieldGenerator** type must implement one of them or an infinite recursion results. This is for backward compatibility for users and implementors. In a future version this function will be made pure virtual.

std::string **str()** const
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

**FieldValue** (double *val*)

**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: A (possibly empty) list of arguments to the generator function

```cpp
double generate(const bout::generator::Context &ctx)
```
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same
value given the same inputs

Note: The default implementations of generate call each other; the implementor of a FieldGenerator type
must implement one of them or an infinite recursion results. This is for backward compatibility for users
and implementors. In a future version this function will be made pure virtual.

```cpp
std::string str() const
```
Create a string representation of the generator, for debugging output.

**Private Members**

double value

class ParseException : public std::exception

**Public Functions**

```cpp
ParseException(const std::string &message_)
```
template<class S, class ...Args>
ParseException(const S &format, const Args&... args)

```cpp
~ParseException() const
```

```cpp
const char *what() const
```

**Private Members**

std::string message

### L.2.57 File fci.cxx

### L.2.58 File fci.hxx

```cpp
class FCIMap
```
#include <fci.hxx> Field line map - contains the coefficients for interpolation.

**Public Functions**

```cpp
FCIMap()
```

```cpp
FCIMap(Mesh &mesh, Options &options, int offset, BoundaryRegionPar *boundary, bool zperiodic)
```

```cpp
Field3D interpolate(Field3D &f) const
```

```cpp
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<XZInterpolation> interp
   Interpolation objects.
std::unique_ptr<XZInterpolation> interp_corner

class FCITransform: public ParallelTransform
   #include <fci.hxx> Flux Coordinate Independent method for parallel derivatives.

Public Functions

FCITransform()
FCITransform(Mesh &mesh, bool zperiodic = true, Options *opt = nullptr)
void calcParallelSlices(Field3D &f)
   Given a 3D field, calculate and set the Y up down fields.
void integrateParallelSlices(Field3D &f)
   Calculate Yup and Ydown fields by integrating over mapped points. This should be used for parallel divergence operators.
const Field3D toFieldAligned(const Field3D &f, const std::string &region = "RGN_ALL")
   Convert a field into field-aligned coordinates so that the y index is along the magnetic field.
const FieldPerp toFieldAligned(const FieldPerp &f, const std::string &region = "RGN_ALL")
const Field3D fromFieldAligned(const Field3D &f, const std::string &region = "RGN_ALL")
   Convert back from field-aligned coordinates into standard form.
const FieldPerp fromFieldAligned(const FieldPerp &f, const std::string &region = "RGN_ALL")
bool canToFromFieldAligned()
bool requiresTwistShift(bool twist_shift_enabled, MAYBE_UNUSED YDirectionType ytype

Protected Functions

void checkInputGrid()
   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**

\[ \text{std::vector<FCIMap> field_line_maps} \]
FCI maps for each of the parallel slices.

---

**L.2.59 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
{
    estimate,
    measure,
    exhaustive,
};
```

---

**Functions**

```
std::string toString(const FFT_MEASUREMENT_FLAG e)

FFT_MEASUREMENT_FLAG FFT_MEASUREMENT_FLAGFromString(const std::string &s)

std::ostream & operator<< (std::ostream &out, const FFT_MEASUREMENT_FLAG &e)

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 \( \text{IDFT} \) is the inverse fourier transform. See the the fftw user manual for details.

**Parameters**

- \text{in}: Pointer to the 1D array to take the fourier transform of
- \text{length}: Number of points in the input array
- \text{out}: Pointer to the complex 1D array which is the FFT of \text{in}
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 out_k = sum_{j=0}^{length-1} in_j*exp(2*pi*j*k*sqrt(-1)/length)
    See the the fftw user manual for details.

    Parameters
    • in: Pointer to the 1D array to take the inverse fourier transform of
    • length: Number of points in the input array
    • out: Pointer to the complex 1D array which is IFFTed

void DST (const BoutReal *in, int length, dcomplex *out)
    Discrete Sine Transform
    in and out arrays must both be of the same length

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

void rfft (const BoutReal *in, int length, dcomplex *out)
    Returns the fft of a real signal using fftw_forward
    
    The fftw_forward returns out_k = sum_{j=0}^{length-1} in_j*exp(-2*pi*j*k*sqrt(-1)/length)
    Thus, out_k must be divided by ‘length’ in order for DFT[IDFT[in]] = in where IDFT is the inverse fourier transform. See the the fftw user manual for details.

    Parameters
    • in: Pointer to the 1D array to take the fourier transform of
    • length: Number of points in the input array
    • out: Pointer to the complex 1D array which is the FFT of in

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 out_k = sum_{j=0}^{length-1} in_j*exp(2*pi*j*k*sqrt(-1)/length)
    See the the fftw user manual for details.

    Parameters
    • in: Pointer to the 1D array to take the inverse fourier transform of
    • length: Number of points in the input array
    • out: Pointer to the complex 1D array which is IFFTed
void DST (const BoutReal *in, int length, dcomplex *out)
Discrete Sine Transform
in and out arrays must both be of the same length

void DST_rev (dcomplex *in, int length, BoutReal *out)
Inverse Discrete Sine Transform
in and out arrays must both be of the same 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 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 in using fftw_forward.

Array<BoutReal> irfft (const Array<dcomplex> &in, int length)
Take the inverse fft of signal in where the outputs are only reals. Requires the length of the original real signal
length is required because input signals to the forward transform of length n and n + 1 both produce ffts of length \( \frac{n}{2} + 1 \) i.e. it’s not possible to recover the length of the original signal from the fft alone.

Expects that in.size() == (length / 2) + 1

L.2.60 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_UNUSEDInt length, MAYBE_UNUSEDdcomplex *out

void irfft (MAYBE_UNUSED) const dcomplex *in
  , MAYBE_UNUSEDInt length, MAYBE_UNUSEDBoutReal *out

void DST (MAYBE_UNUSED) const BoutReal *in
  , MAYBE_UNUSEDInt length, MAYBE_UNUSEDdcomplex *out

void DST_rev (MAYBE_UNUSED) dcomplex *in
  , MAYBE_UNUSEDInt length, MAYBE_UNUSEDBoutReal *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?

L.2.61 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 = "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

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)

**L.2.62 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/](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.

Field2D toFieldAligned(const Field2D &f, const std::string &region = "RGN_ALL")
Field2D toFieldAligned(const Field2D &f, REGION region)
Field2D fromFieldAligned(const Field2D &f, const std::string &region = "RGN_ALL")
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

void checkData (const Field2D &f, REGION region)

void invalidateGuards (Field2D &var)
    Force guard cells of passed field var to NaN.

Field2D &ddt (Field2D &f)
    Returns a reference to the time-derivative of a field f

    Wrapper around member function f.timeDeriv()

template<>
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 ind_type = Ind2D

using value_type = BoutReal
    Data type.

Public Functions

Field2D::Field2D (Mesh * localmesh = nullptr, CELL_LOC location_in = CELL_CENTRE, DirectionTypes directions_in = {YDirectionType::Standard, ZDirectionType::Average})
    Constructor, taking an optional mesh pointer This mesh pointer is not used until the data is allocated, since
    Field2D objects can be globals, created before a mesh has been created.

    By default the global Mesh pointer (mesh) is used.

    Parameters

      • localmesh: The mesh which defines the field size.

Field2D (const Field2D &f)
    Copy constructor. After this both fields will share the same underlying data.
Field2D (Field2D &f)
   Move constructor

Field2D (BoutReal val, Mesh *localmesh = nullptr)
   Constructor. This creates a Field2D using the global Mesh pointer (mesh) allocates data, and assigns the value val to all points including boundary cells.

Field2D : Field2D (Array < BoutReal > data, Mesh * localmesh, CELL_LOC location = CELL_CENTRE)
   Constructor from Array and Mesh.

~Field2D ()
   Destructor

Field2D &allocate ()
   Ensure data is allocated.

bool isAllocated () const
   Test if data is allocated.

Field2D *timeDeriv ()
   Return a pointer to the time-derivative field.

int getNx () const
   Return the number of nx points

int getNy () const
   Return the number of ny points

int getNz () const
   Return the number of nz points

Field2D &setLocation (CELL_LOC new_location)

Field2D &setDirectionY (YDirectionType d)

bool hasParallelSlices () const
   Check if this field has yup and ydown fields.

bool hasYupYdown () const

Field2D &yup (std::vector<Field2D>::size_type index = 0)

const Field2D &yup (std::vector<Field2D>::size_type index = 0) const

Field2D &ydown (std::vector<Field2D>::size_type index = 0)

const Field2D &ydown (std::vector<Field2D>::size_type index = 0) const

Field2D &ynext (int dir)

const 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
Region<Ind2D>::RegionIndices::const_iterator begin () const
Region<Ind2D>::RegionIndices::const_iterator end () const

BoutReal &operator[] (const Ind2D &d)
const BoutReal &operator[] (const Ind2D &d) const

BoutReal &operator[] (const Ind3D &d)
const BoutReal &operator[] (const Ind3D &d) const

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.

const BoutReal &operator() (int jx, int jy) const
BoutReal &operator() (int jx, int jy, int jz)
Direct access to underlying array. This version is for compatibility with Field3D objects

const BoutReal &operator() (int jx, int jy, int jz) const
Field2D &operator+= (const Field2D &rhs)
In-place addition. Copy-on-write used if data is shared.

Field2D &operator+= (BoutReal rhs)
In-place addition. Copy-on-write used if data is shared.

Field2D &operator-= (const Field2D &rhs)
In-place subtraction. Copy-on-write used if data is shared.

Field2D &operator-= (BoutReal rhs)
In-place subtraction. Copy-on-write used if data is shared.

Field2D &operator* (const Field2D &rhs)
In-place multiplication. Copy-on-write used if data is shared.

Field2D &operator* (BoutReal rhs)
In-place multiplication. Copy-on-write used if data is shared.

Field2D &operator/= (const Field2D &rhs)
In-place division. Copy-on-write used if data is shared.

Field2D &operator/= (BoutReal rhs)
In-place division. Copy-on-write used if data is shared.

void accept (FieldVisitor &v)
Visitor pattern support.

bool isReal () const
Returns true if field consists of BoutReal values.
bool is3D() const
    True if variable is 3D.

int byteSize() const
    Number of bytes for a single point.

int BoutRealSize() const
    Number of BoutReals (not implemented if not BoutReal)

void doneComms()

void applyBoundary(bool init = false)
void applyBoundary(BoutReal time)
void applyBoundary(const std::string &condition)
void applyBoundary(const char *condition)
void applyBoundary(const std::string &region, const std::string &condition)
void applyTDerivBoundary()
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 Vector2D

void swap(Field2D &first, Field2D &second)

L.2.63  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 = "RGN_ALL")**

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 = "RGN_ALL")**

Fourier filtering, removes all except one mode

**Parameters**

- var: Variable to apply filter to
- N0: The component to keep
- rgn: The region to calculate the result over

**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: Variable to apply filter to
- zmax: Maximum mode in Z
- keep_zonal: Keep the zonal component if true
- rgn: 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: The variable to be modified in-place
- jx: X index
- jy: Y index
- zangle: 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: The variable to modify in-place
- zangle: The angle to shift by in Z
- rgn: The region to calculate the result over

void checkData (const Field3D &f, const std::string &region = "RGN_NOBNDRY")
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 = "RGN_ALL")
Average in the Z direction

Parameters

- f: Variable to average
- rgn: 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.

L.2.64 File field3d.hxx

Defines

__FIELD3D_H__

Functions

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)
Field3D operator+ (const Field3D &lhs, BoutReal rhs)
Field3D operator- (const Field3D &lhs, BoutReal rhs)
Field3D operator* (const Field3D &lhs, BoutReal rhs)
Field3D operator/ (const Field3D &lhs, BoutReal rhs)
Field3D operator+ (BoutReal lhs, const Field3D &rhs)
Field3D operator- (BoutReal lhs, const Field3D &rhs)
Field3D operator* (BoutReal lhs, const Field3D &rhs)
Field3D operator/ (BoutReal lhs, const Field3D &rhs)
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 = "RGN_ALL")
  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

Field3D pow (const Field3D &lhs, const Field2D &rhs, REGION rgn)
FieldPerp pow (const Field3D &lhs, const FieldPerp &rhs, const std::string &rgn = "RGN_ALL")
FieldPerp pow (const Field3D &lhs, const FieldPerp &rhs, REGION rgn)

void checkData (const Field3D &f, const std::string &rgn = "RGN_NOBNDRY")
  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 checkData (const Field3D &f, REGION region)
Field3D filter (const Field3D &var, int N0, const std::string &rgn = "RGN_ALL")
  Fourier filtering, removes all except one mode

Parameters
  • var: Variable to apply filter to
  • N0: The component to keep
  • rgn: The region to calculate the result over

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: Variable to apply filter to
  • zmax: Maximum mode in Z
  • keep_zonal: Keep the zonal component if true
  • rgn: The region to calculate the result over

Field3D lowPass (const Field3D &var, int zmax, bool keep_zonal, REGION rgn)
Field3D lowPass (const Field3D &var, int zmax, int keep_zonal, REGION rgn = RGN_ALL)
The argument keep_zonal used to be integer “zmin” this was a misnomer. Please use the version above which uses a bool instead.

Fourier low pass filtering. Removes modes higher than zmax

Parameters
- *var*: Variable to apply filter to
- *zmax*: Maximum mode in Z
- *rgn*: The region to calculate the result over

Field3D lowPass (const Field3D &var, int zmax, const std::string &rgn = "RGN_ALL")

Parameters
- *var*: Variable to apply filter to
- *zmax*: Maximum mode in Z
- *rgn*: The region to calculate the result over

Field3D lowPass (const Field3D &var, int zmax, REGION rgn)

void shiftZ (Field3D &var, int jx, int jy, double zangle)
Perform a shift by a given angle in Z

Parameters
- *var*: The variable to be modified in-place
- *jx*: X index
- *jy*: Y index
- *zangle*: 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*: The variable to modify in-place
- *zangle*: The angle to shift by in Z
- *rgn*: The region to calculate the result over

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*: Variable to average
- *rgn*: The region to calculate the result over

Field2D DC (const Field3D &f, REGION rgn)

void invalidateGuards (Field3D &var)
Force guard cells of passed field var to NaN.

Field3D &ddt (Field3D &f)
Returns a reference to the time-derivative of a field f
Wrapper around member function f.timeDeriv()
```cpp
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.

```cpp
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(0.0); // 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.

```cpp
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:

```cpp
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.

```cpp
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:

```cpp
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
```

To have separate fields for yup and ydown, first call

```
f.splitParallelSlices(); // f.yup() and f.ydown() separate
f.yup(); // ok
f.yup()(0,1,0) // error; f.yup not allocated
f.yup() = 1.0; // Set f.yup() field to 1.0
f.yup()(0,1,0) // ok
```

**Unnamed Group**

```
Field3D &operator=(const Field3D &rhs)
Assignment operators

Field3D &operator=(Field3D &rhs)

Field3D &operator=(const Field2D &rhs)
```

426 Appendix L. API reference
void operator= (const FieldPerp &rhs)
    return void, as only part initialised

Field3D &operator= (BoutReal val)
Public Functions

Field3D::Field3D(Mesh * localmesh = nullptr, CELL_LOC location_in = CELL_CENTRE, DirectionTypes directions_in = {YDirectionType::Standard, ZDirectionType::Standard})
Constructor.

Constructor

Note: the global “mesh” can’t be passed here because fields may be created before the mesh is.

Field3D (const Field3D &f)
Copy constructor

Doesn’t copy any data, just create a new reference to the same data (copy on change later)

Field3D (Field3D &&f)
Move constructor.

Field3D (const Field2D &f)
Constructor from 2D field.

Field3D (BoutReal val, Mesh * localmesh = nullptr)
Constructor from value.

Field3D::Field3D(Array < BoutReal > data, Mesh * localmesh, CELL_LOC location = CELL_CENTRE, DirectionTypes directions_in = {YDirectionType::Standard, ZDirectionType::Standard})
Constructor from Array and Mesh.

~Field3D ()
Destructor.

Field3D &allocate ()
Ensures that memory is allocated and unique

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

int getNx () const
Return the number of nx points

int getNy () const
Return the number of ny points

int getNz () const
Return the number of nz points

Field3D &setLocation (CELL_LOC new_location)

Field3D &setDirectionY (YDirectionType d)

void splitParallelSlices ()
Ensure that this field has separate fields for yup and ydown.

void splitYupYdown ()

void clearParallelSlices ()
Clear the parallel slices, yup and ydown
void mergeYupYdown()

bool hasParallelSlices () const
    Check if this field has yup and ydown fields.

bool hasYupYdown () const

const Field3D & yup (std::vector<Field3D>::size_type index = 0)
    Check if this field has yup and ydown fields Return reference to yup field

const Field3D & yup (std::vector<Field3D>::size_type index = 0) const
    Return const reference to yup field.

Field3D & ydown (std::vector<Field3D>::size_type index = 0)
    Return reference to ydown field.

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 Field3D &ynext (int offset) const

bool requiresTwistShift (bool twist_shift_enabled)
    If twist_shift_enabled is true, does this Field3D require a twist-shift at branch cuts on closed field lines?

const Region<Ind3D> & getRegion (REGION region) const
    Return a Region<Ind3D> reference to use to iterate over this field

    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 x (i.xp()) and one index down in x (i.xm()), putting the result into a different field ‘g’

    for(const auto &i : f.getRegion(RGN_NOBNDRY)) { g[i] = f[i.xp()] - f[i.xm()]; }

const Region<Ind3D> & getRegion (const std::string &region_name) const

const Region<Ind2D> & getRegion2D (REGION region) const
    Return a Region<Ind2D> reference to use to iterate over the x- and y-indices of this field

const Region<Ind2D> & getRegion2D (const std::string &region_name) const

Region<Ind3D>::RegionIndices::const_iterator begin () const

Region<Ind3D>::RegionIndices::const_iterator end () const

BoutReal & operator[] (const Ind3D &d)

const BoutReal & operator[] (const Ind3D &d) const

BoutReal & operator() (const IndPerp &d, int jy)

const BoutReal & operator() (const IndPerp &d, int jy) const

BoutReal & operator() (const Ind2D &d, int jz)
const BoutReal &operator() (const Ind2D &d, int jz) const

BoutReal &operator() (int jx, int jy, int jz)
Direct access to the underlying data array
If CHECK > 2 then bounds checking is performed
If CHECK <= 2 then no checks are performed, to allow inlining and optimisation of inner loops

const BoutReal &operator() (int jx, int jy, int jz) const

const BoutReal *operator() (int jx, int jy) const
Direct access to the underlying data array
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

BoutReal *operator() (int jx, int jy)

bool isReal () const
Returns true if field consists of BoutReal values.

bool is3D () const
True if variable is 3D.

int byteSize () const
Number of bytes for a single point.

int BoutRealSize () const
Number of BoutReals (not implemented if not BoutReal)

void accept (FieldVisitor &v)
Visitor pattern support.

void doneComms ()

Field3D &calcParallelSlices ()

void applyBoundary (bool init = false)

void applyBoundary (BoutReal t)

void applyBoundary (const std::string &condition)

void applyBoundary (const char *condition)

void applyBoundary (const std::string &region, const std::string &condition)

void applyTDerivBoundary ()

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.

void applyParallelBoundary ()

void applyParallelBoundary (BoutReal t)

void applyParallelBoundary (const std::string &condition)
void applyParallelBoundary(const char *condition)
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 Vector3D

void swap(Field3D &first, Field3D &second)

L.2.65 File field.cxx

L.2.66 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

**ASSERT1_FIELDS_COMPATIBLE** (field1, field2)

**FIELD_FUNC** (name, func)
   This macro takes a function `func`, which is assumed to operate on a single BoutReal and return a single BoutReal, and wraps it up into a function of a Field called `name`.
   
   If CHECK >= 1, checks if the Field is allocated

   **Parameters**
   - `name`: The name of the function to define
   - `func`: The function to apply to each value

   Loops over the entire domain, applies function, and uses checkData() to, if CHECK >= 3, check result for non-finite numbers

Functions

bool **areFieldsCompatible** (const Field &field1, const Field &field2)
   Check if Fields have compatible meta-data.

template<typename T>
T **emptyFrom** (const T &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.

template<typename T>
T **zeroFrom** (const T &f)
   Return a field of some type derived from Field, with metadata copied from another field and a data array allocated and initialised to zero.

template<typename T>
T **filledFrom** (const T &f, BoutReal fill_value)
   Return a field of some type derived from Field, with metadata copied from another field and a data array allocated and filled with the given value.

template<typename T, typename Function, typename = decltype(std::declval<Function&>()(std::declval<typename T::ind_type&>()))>
T **filledFrom** (const T &f, Function func, std::string region_string = "RGN_ALL")
   Return a field of some type derived from Field, with metadata copied from another field and a data array allocated and filled using a callable e.g. lambda function

   e.g. Field3D result = filledFrom(field, [&](const auto& index) { return ... });

   An optional third argument is the region string

template<typename T, typename = bout::utils::EnableIfField<T>>
T **operator+** (const T &f)
   Unary + operator. This doesn’t do anything.

template<typename T>
T **toFieldAligned** (const T &f, const std::string &region = "RGN_ALL")

template<typename T>
T **toFieldAligned** (const T &f, REGION region)

template<typename T>
T **fromFieldAligned** (const T &f, const std::string &region = "RGN_ALL")

template<typename T>
\[ T \text{ fromFieldAligned}(\text{const } T & f, \text{REGION region}) \]

template<typename T, typename = bout::utils::EnableIfField<T>>
\[ \text{BoutReal min}(\text{const } T & f, \text{bool allpe = false, const std::string & rgn = "RGN_NOBNDRY"}) \]

template<typename T, typename = bout::utils::EnableIfField<T>>
\[ \text{BoutReal max}(\text{const } T & f, \text{bool allpe = false, const std::string & rgn = "RGN_NOBNDRY"}) \]

template<typename T, typename = bout::utils::EnableIfField<T>>
\[ \text{BoutReal mean}(\text{const } T & f, \text{bool allpe, REGION rgn}) \]

template<typename T, typename = bout::utils::EnableIfField<T>>
\[ T \text{ pow}(\text{const } T & lhs, \text{const } T & rhs, \text{const std::string & rgn = "RGN_ALL"}) \]

\begin{itemize}
  \item Exponent: pow(lhs, rhs) is \( f \) raised to the power of \( rhs \)
  \item This loops over the entire domain, including guard/boundary cells by default (can be changed using the \( rgn \) argument). If CHECK \( \geq 3 \) then the result will be checked for non-finite numbers
\end{itemize}

template<typename T, typename = bout::utils::EnableIfField<T>>
\[ T \text{ pow}(\text{const } T & lhs, \text{const } T & rhs, \text{REGION rgn}) \]

template<typename T, typename = bout::utils::EnableIfField<T>>
\[ T \text{ pow}(\text{const } T & lhs, \text{BoutReal rhs, const std::string & rgn = "RGN_ALL"}) \]

template<typename T, typename = bout::utils::EnableIfField<T>>
\[ T \text{ pow}(\text{const } T & lhs, \text{BoutReal rhs, REGION rgn}) \]

template<typename T, typename = bout::utils::EnableIfField<T>>
\[ T \text{ pow}(\text{BoutReal lhs, const } T & rhs, \text{const std::string & rgn = "RGN_ALL"}) \]

template<typename T, typename = bout::utils::EnableIfField<T>>
\[ T \text{ pow}(\text{BoutReal lhs, const } T & rhs, \text{REGION rgn}) \]

template<typename T, typename = bout::utils::EnableIfField<T>>
\[ T \text{ sqrt}(\text{const } T & f, \text{const std::string & rgn = "RGN_ALL"}) \]

\begin{itemize}
  \item Square root of \( f \) over region \( rgn \)
  \item This loops over the entire domain, including guard/boundary cells by default (can be changed using the \( rgn \) argument). If CHECK \( \geq 3 \) then the result will be checked for non-finite numbers
\end{itemize}

template<typename T, typename = bout::utils::EnableIfField<T>>
\[ T \text{ sqrt}(\text{const } T & f, \text{REGION region}) \]

template<typename T, typename = bout::utils::EnableIfField<T>>
\[ T \text{ abs}(\text{const } T & f, \text{const std::string & rgn = "RGN_ALL"}) \]

\begin{itemize}
  \item Absolute value (modulus, \( f \) of \( f \) over region \( rgn \)
  \item This loops over the entire domain, including guard/boundary cells by default (can be changed using the \( rgn \) argument). If CHECK \( \geq 3 \) then the result will be checked for non-finite numbers
\end{itemize}

template<typename T, typename = bout::utils::EnableIfField<T>>
\[ T \text{ abs}(\text{const } T & f, \text{REGION region}) \]

template<typename T, typename = bout::utils::EnableIfField<T>>
\[ \text{fromFieldAligned}(\text{const } T & f, \text{REGION region}) \]
\( T \exp (\text{const } T &f, \text{const } \text{std::string } &rgn = \"RGN\_ALL\") \)

Exponential: \( \exp(f) \) is e to the power of \( f \), over region \( rgn \).  
This loops over the entire domain, including guard/boundary cells by default (can be changed using the \( rgn \) argument). If CHECK \( \geq 3 \) then the result will be checked for non-finite numbers.

```cpp
template<typename T, typename = bout::utils::EnableIfField<T>>
T exp (\text{const } T &f, \text{const } REGION &region)
```

\( T \log (\text{const } T &f, \text{const } \text{std::string } &rgn = \"RGN\_ALL\") \)

Natural logarithm of \( f \) over region \( rgn \), inverse of exponential  
\( \ln(\exp(f)) = f \)
This loops over the entire domain, including guard/boundary cells by default (can be changed using the \( rgn \) argument). If CHECK \( \geq 3 \) then the result will be checked for non-finite numbers.

```
template<typename T, typename = bout::utils::EnableIfField<T>>
T log (\text{const } T &f, \text{REGION region})
```

\( T \sin (\text{const } T &f, \text{const } \text{std::string } &rgn = \"RGN\_ALL\") \)

Sine trigonometric function.

This loops over the entire domain, including guard/boundary cells by default (can be changed using the \( rgn \) argument). If CHECK \( \geq 3 \) then the result will be checked for non-finite numbers.

```
template<typename T, typename = bout::utils::EnableIfField<T>>
T sin (\text{const } T &f, \text{const } REGION region)
```

\( T \cos (\text{const } T &f, \text{const } \text{std::string } &rgn = \"RGN\_ALL\") \)

Cosine trigonometric function.

This loops over the entire domain, including guard/boundary cells by default (can be changed using the \( rgn \) argument). If CHECK \( \geq 3 \) then the result will be checked for non-finite numbers.

```
template<typename T, typename =bout::utils::EnableIfField<T>>
T cos (\text{const } T &f, \text{REGION region})
```

\( T \tan (\text{const } T &f, \text{const } \text{std::string } &rgn = \"RGN\_ALL\") \)

Tangent trigonometric function.

This loops over the entire domain, including guard/boundary cells by default (can be changed using the \( rgn \) argument). If CHECK \( \geq 3 \) then the result will be checked for non-finite numbers.

```
template<typename T, typename =bout::utils::EnableIfField<T>>
T tan (\text{const } T &f, \text{REGION region})
```

Parameters

- \( f \): Angle in radians
- \( rgn \): The region to calculate the result over
template<typename T, typename = bout::utils::EnableIfField<T>>
T tan (const T &f, REGION region)

template<typename T, typename = bout::utils::EnableIfField<T>>
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: Angle in radians
  • rgn: The region to calculate the result over

template<typename T, typename = bout::utils::EnableIfField<T>>
T cosh (const T &f, REGION region)

template<typename T, typename = bout::utils::EnableIfField<T>>
T tanh (const T &f, const std::string &rgn = "RGN_ALL")
  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: Angle in radians
  • rgn: The region to calculate the result over

template<typename T, typename = bout::utils::EnableIfField<T>>
T tanh (const T &f, REGION region)

template<typename T, typename = bout::utils::EnableIfField<T>>
bool finite (const T &f, const std::string &rgn = "RGN_ALL")
  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)

template<typename T, typename = bout::utils::EnableIfField<T>>
bool finite (const T &f, REGION rgn)

template<typename T, typename = bout::utils::EnableIfField<T>>
T copy (const T &f)
  Makes a copy of a field f, ensuring that the underlying data is not shared.

template<typename T, typename = bout::utils::EnableIfField<T>>
Apply a floor value \( f \) to a field \( \text{var} \). Any value lower than the floor is set to the floor.

**Parameters**

- \( \text{var} \): Variable to apply floor to
- \( f \): The floor value
- \( \text{rgn} \): The region to calculate the result over

```cpp
template<typename T, typename = bout::utils::EnableIfField<T>>
T floor(const T &var, BoutReal f, const std::string &rgn = "RGN_ALL")
```

### Public Functions

- **Field()**
- **Field(const Field &other)**
- **Field(Field &&other)**
- **Field &operator=(const Field &other)**
- **Field &operator=(Field &&other)**
- **virtual ~Field()**
- **Field(Mesh *localmesh, CELL_LOC location_in, DirectionTypes directions_in)**
- **void setLocation(CELL_LOC new_location)**
  
  Set variable location for staggered grids to

  **Parameters**

  - \( \text{new\_location} \): Throws BoutException if \( \text{new\_location} \) 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.

  ```cpp
  CELL_LOC getLocation() const
  ```

  Get variable location.

  ```cpp
  DirectionTypes getDirections() const
  ```

  Getters for DIRECTION types.

  ```cpp
  YDirectionType getDirectionY() const
  ```

  ```cpp
  ZDirectionType getDirectionZ() const
  ```

  ```cpp
  void setDirectionY(YDirectionType y_type)
  ```

  Setters for *DirectionType.

  ```cpp
  void setDirectionZ(ZDirectionType z_type)
  ```
virtual bool bndryValid()

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 *getCoordinates (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

int getNx () const
    Return the number of nx points

int getNy () const
    Return the number of ny points

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

void copyFieldMembers (const Field &f)
    Copy the members from another Field.

Protected Attributes

Mesh *fieldmesh = {nullptr}

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

void swap (Field &first, Field &second)

namespace bout
    SNB model
### Functions

```cpp
template<typename T>
void checkFinite(const T &f, const std::string &name = "field", const std::string &rgn = "RGN_ALL")
{
    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) 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>
void checkPositive(const T &f, const std::string &name = "field", const std::string &rgn = "RGN_ALL")
{
    Check if all values of a field var are positive. Loops over all points including the boundaries by default (can be changed using the 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.
}
```

### L.2.67 File field_data.cxx

### L.2.68 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

```cpp
#define FIELD_DATA_H
```

```cpp
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()
~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.

virtual int BoutRealSize() const
    Number of BoutReals (not implemented if not BoutReal)

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.

virtual void applyBoundary(bool init = false)

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
L.2.69 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.

L.2.70 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()

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

*FieldFactory* `get()`
Get the Singleton object.

### Protected Functions

*FieldGeneratorPtr* `resolve` (std::string &name) `const`
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)?

int `max_recursion_depth` = {0}

*const Options* `options`
The default options used in `resolve()`, can be temporarily overridden in `parse()/create2D()/create3D()`.

std::list<std::string> `lookup`
Names currently being parsed.

std::map<std::string, FieldGeneratorPtr> `cache`
Cache parsed strings so repeated evaluations don’t result in allocating more generators.

### Public Functions

**FieldFunction**

FieldFunction()
FieldFunction(FuncPtr userfunc)

*BoutReal* `generate` (const bout::generator::Context &ctx)
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

Note: The default implementations of `generate` call each other; the implementor of a *FieldGenerator* type must implement one of them or an infinite recursion results. This is for backward compatibility for users and implementors. In a future version this function will be made pure virtual.
Private Members

`FuncPtr func`

`class FieldNull: public FieldGenerator`

Public Functions

FieldNull()

`BoutReal generate(const bout::generator::Context &ctx)`
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

Note: The default implementations of generate call each other; the implementor of a `FieldGenerator` type must implement one of them or an infinite recursion results. This is for backward compatibility for users and implementors. In a future version this function will be made pure 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`: A (possibly empty) list of arguments to the generator function

Public Static Functions

`static FieldGeneratorPtr get()`
Singleton.

L.2.71 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`

`virtual ~FieldVisitor()`
These classes are used by FieldFactory.

**Typedefs**

```cpp
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
FieldValuePtr (BoutReal *val)
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: A (possibly empty) list of arguments to the generator function

BoutReal generate (const bout::generator::Context &ctx)
  // Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same
  // value given the same inputs

Note: The default implementations of generate call each other; the implementor of a FieldGenerator type
must implement one of them or an infinite recursion results. This is for backward compatibility for users
and implementors. In a future version this function will be made pure virtual.
```

**Private Members**

```cpp
BoutReal *ptr
```

```cpp
template <single_arg_op Op>
class FieldGenOneArg : public FieldGenerator
```

**Public Functions**

```cpp
FieldGenOneArg (FieldGeneratorPtr g, const std::string &name = "function")
  // Template for single-argument function
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

- \texttt{args}: A (possibly empty) list of arguments to the generator function

\texttt{BoutReal generate (const bout::generator::Context \&ctx)}

Generate a value at the given coordinates \((x,y,z,t)\) This should be deterministic, always returning the same value given the same inputs

Note: The default implementations of generate call each other; the implementor of a \texttt{FieldGenerator} type must implement one of them or an infinite recursion results. This is for backward compatibility for users and implementors. In a future version this function will be made pure virtual.

\texttt{std::string str() const}

Create a string representation of the generator, for debugging output.

Private Members

\texttt{FieldGeneratorPtr gen}

\texttt{std::string name}

A string describing the function, to be printed in error messages.

\texttt{template<\texttt{double_arg_op Op}> class FieldGenTwoArg : public FieldGenerator}

Public Functions

\texttt{FieldGenTwoArg (FieldGeneratorPtr a, FieldGeneratorPtr b, const std::string \&name = \texttt{"function"})}

< Template for two-argument function

\texttt{FieldGeneratorPtr clone (const std::list<FieldGeneratorPtr> \&args)}

Virtual constructor. Makes a copy of this \texttt{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

- \texttt{args}: A (possibly empty) list of arguments to the generator function

\texttt{BoutReal generate (const bout::generator::Context \&ctx)}

Generate a value at the given coordinates \((x,y,z,t)\) This should be deterministic, always returning the same value given the same inputs

Note: The default implementations of generate call each other; the implementor of a \texttt{FieldGenerator} type must implement one of them or an infinite recursion results. This is for backward compatibility for users and implementors. In a future version this function will be made pure virtual.

\texttt{std::string str() const}

Create a string representation of the generator, for debugging output.

Private Members

\texttt{FieldGeneratorPtr A}

\texttt{FieldGeneratorPtr B}
std::string name
    The name of the function, to be printed in error messages.

class FieldATan : public FieldGenerator
    #include <fieldgenerators.hxx> Arc (Inverse) tangent. Either one or two argument versions.

Public Functions

FieldATan (FieldGeneratorPtr a, FieldGeneratorPtr b = nullptr)

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: A (possibly empty) list of arguments to the generator function

BoutReal generate (const bout::generator::Context &ctx)
    Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same
    value given the same inputs

    Note: The default implementations of generate call each other; the implementor of a FieldGenerator type
    must implement one of them or an infinite recursion results. This is for backward compatibility for users
    and implementors. In a future version this function will be made pure virtual.

Private Members

    FieldGeneratorPtr A
    FieldGeneratorPtr B

class FieldGaussian : public FieldGenerator
    #include <fieldgenerators.hxx> Gaussian distribution, taking mean and width arguments.

Public Functions

FieldGaussian (FieldGeneratorPtr xin, FieldGeneratorPtr sin)

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: A (possibly empty) list of arguments to the generator function

BoutReal generate (const bout::generator::Context &ctx)
    Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same
    value given the same inputs

    Note: The default implementations of generate call each other; the implementor of a FieldGenerator type
    must implement one of them or an infinite recursion results. This is for backward compatibility for users
    and implementors. In a future version this function will be made pure virtual.
Private Members

FieldGeneratorPtr X
FieldGeneratorPtr s

class FieldHeaviside: public FieldGenerator
#include <fieldgenerators.hxx> Heaviside function, switches between 0 and 1.

Public Functions

FieldHeaviside(FieldGeneratorPtr g)

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: A (possibly empty) list of arguments to the generator function

BoutReal generate(const bout::generator::Context &ctx)
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

Note: The default implementations of generate call each other; the implementor of a FieldGenerator type must implement one of them or an infinite recursion results. This is for backward compatibility for users and implementors. In a future version this function will be made pure virtual.

std::string str() const
Create a string representation of the generator, for debugging output.

Private Members

FieldGeneratorPtr gen

class FieldMin: public FieldGenerator
#include <fieldgenerators.hxx> Minimum.

Public Functions

FieldMin()

FieldMin(const std::list<FieldGeneratorPtr> args)

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: A (possibly empty) list of arguments to the generator function
BoutReal generate (const bout::generator::Context &ctx)
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

Note: The default implementations of generate call each other; the implementor of a FieldGenerator type must implement one of them or an infinite recursion results. This is for backward compatibility for users and implementors. In a future version this function will be made pure virtual.

Private Members

std::list<FieldGeneratorPtr> input

class FieldMax: public FieldGenerator
#include <fieldgenerators.hxx> Maximum.

Public Functions

FieldMax ()
FieldMax (const std::list<FieldGeneratorPtr> args)

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: A (possibly empty) list of arguments to the generator function

BoutReal generate (const bout::generator::Context &ctx)
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

Note: The default implementations of generate call each other; the implementor of a FieldGenerator type must implement one of them or an infinite recursion results. This is for backward compatibility for users and implementors. In a future version this function will be made pure virtual.

Private Members

std::list<FieldGeneratorPtr> input

class FieldClamp: public FieldGenerator
#include <fieldgenerators.hxx> Clamp

Force a value to be in specified range Also called clip e.g. in NumPy, but in C++17 std::clamp is in <algorithm>

Note that the result is not well defined if low > high

Public Functions

FieldClamp ()
FieldClamp (FieldGeneratorPtr value, FieldGeneratorPtr low, FieldGeneratorPtr high)
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: A (possibly empty) list of arguments to the generator function

BoutReal generate(const bout::generator::Context &ctx)
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

Note: The default implementations of generate call each other; the implementor of a FieldGenerator type must implement one of them or an infinite recursion results. This is for backward compatibility for users and implementors. In a future version this function will be made pure virtual.

Private Members

FieldGeneratorPtr value
The value to be clamped.

FieldGeneratorPtr low

FieldGeneratorPtr high
The range within which the result will be.

class FieldRound: public FieldGenerator
#include <fieldgenerators.hxx> Generator to round to the nearest integer.

Public Functions

FieldRound(FieldGeneratorPtr g)

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: A (possibly empty) list of arguments to the generator function

BoutReal generate(const bout::generator::Context &ctx)
Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

Note: The default implementations of generate call each other; the implementor of a FieldGenerator type must implement one of them or an infinite recursion results. This is for backward compatibility for users and implementors. In a future version this function will be made pure virtual.

Private Members

FieldGeneratorPtr gen

class FieldBallooning: public FieldGenerator
Public Functions

FieldBallooning (Mesh *m, FieldGeneratorPtr a = nullptr, int n = 3)

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: A (possibly empty) list of arguments to the generator function

BoutReal generate (const bout::generator::Context &ctx)
   Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

Note: The default implementations of generate call each other; the implementor of a FieldGenerator type must implement one of them or an infinite recursion results. This is for backward compatibility for users and implementors. In a future version this function will be made pure virtual.

Private Members

Mesh *mesh

FieldGeneratorPtr arg

int ball_n

class FieldMixmode : public FieldGenerator

Public Functions

FieldMixmode (FieldGeneratorPtr a = nullptr, BoutReal seed = 0.5)

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: A (possibly empty) list of arguments to the generator function

BoutReal generate (const bout::generator::Context &ctx)
   Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

Note: The default implementations of generate call each other; the implementor of a FieldGenerator type must implement one of them or an infinite recursion results. This is for backward compatibility for users and implementors. In a future version this function will be made pure virtual.

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

class FieldTanhHat : public FieldGenerator

Public Functions

FieldTanhHat (FieldGeneratorPtr xin, FieldGeneratorPtr widthin, FieldGeneratorPtr centerin, FieldGeneratorPtr steepnessin)

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: A (possibly empty) list of arguments to the generator function

BoutReal generate (const bout::generator::Context &ctx)

Generate a value at the given coordinates (x,y,z,t) This should be deterministic, always returning the same value given the same inputs

Note: The default implementations of generate call each other; the implementor of a FieldGenerator type must implement one of them or an infinite recursion results. This is for backward compatibility for users and implementors. In a future version this function will be made pure virtual.

Private Members

FieldGeneratorPtr X

FieldGeneratorPtr width

FieldGeneratorPtr center

FieldGeneratorPtr steepness

class FieldWhere : public FieldGenerator

Public Functions

FieldWhere (FieldGeneratorPtr test, FieldGeneratorPtr g0, FieldGeneratorPtr l0)

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: A (possibly empty) list of arguments to the generator function
double \texttt{generate} (\texttt{const bout::generator::Context} \&\texttt{ctx})
Generate a value at the given coordinates \((x,y,z,t)\) This should be deterministic, always returning the same value given the same inputs

Note: The default implementations of \texttt{generate} call each other; the implementor of a \texttt{FieldGenerator} type must implement one of them or an infinite recursion results. This is for backward compatibility for users and implementors. In a future version this function will be made pure virtual.

\texttt{std::string \texttt{str} () const}
Create a string representation of the generator, for debugging output.

\textbf{Private Members}

\begin{itemize}
\item \texttt{FieldGeneratorPtr \texttt{test}}
\item \texttt{FieldGeneratorPtr gt0}
\item \texttt{FieldGeneratorPtr lt0}
\end{itemize}

\textbf{L.2.74 File fieldgroup.cxx}

\textbf{Functions}

\begin{itemize}
\item \texttt{FieldGroup operator+ (\texttt{const FieldGroup} \&\texttt{lhs, const FieldGroup} \&\texttt{rhs})}
Combine two FieldGroups.
\end{itemize}

\textbf{L.2.75 File fieldgroup.hxx}

\textbf{Functions}

\begin{itemize}
\item \texttt{FieldGroup operator+ (\texttt{const FieldGroup} \&\texttt{lhs, const FieldGroup} \&\texttt{rhs})}
Combine two FieldGroups.
\end{itemize}

\textbf{class FieldGroup}

\begin{itemize}
\item \texttt{#include} \texttt{<fieldgroup.hxx>} Group together fields for easier communication
\end{itemize}

Note: The \texttt{FieldData} class is used as a base class, which is inherited by \texttt{Field2D}, \texttt{Field3D}, \texttt{Vector2D} and \texttt{Vector3D} however \texttt{Vector2D} and \texttt{Vector3D} are stored by reference to their components \((x,y,z)\) as \texttt{Field2D} or \texttt{Field3D} objects.

\textbf{Public Types}

\begin{itemize}
\item \texttt{using \texttt{iterator} = std::vector::iterator}
Iteration over all fields.
\item \texttt{using \texttt{const_iterator} = std::vector::const_iterator}
Const iteration over all fields.
\end{itemize}

\textbf{Public Functions}

\begin{itemize}
\item \texttt{FieldGroup ()}
\item \texttt{FieldGroup (\texttt{const FieldGroup} \&\texttt{other})}
\end{itemize}
FieldGroup (FieldGroup &&other)

FieldGroup &operator= (const FieldGroup &other)

FieldGroup &operator= (FieldGroup &&other)

FieldGroup (FieldData &f)
  Constructor with a single FieldData f.

FieldGroup (Field3D &f)
  Constructor with a single Field3D f.

FieldGroup (Vector2D &v)
  Constructor with a single Vector2D v
  This is needed so that fvec only contains Field2D or Field3D

FieldGroup (Vector3D &v)
  Constructor with a single Vector3D v
  This is needed so that fvec only contains Field2D or Field3D

template<typename ...Ts>
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.

void add ( const FieldGroup &other)
  Copy contents of another FieldGroup other into this group.

FieldGroup &operator+= (const FieldGroup &other)
  Add the contents of other to this.

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.

void add (Field3D &f)

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.

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>
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>
void add (Field3D &t, Ts&... ts)

template<typename ...Ts>
void add (Vector3D &t, Ts&... ts)

template<typename ...Ts>
void add (Vector2D &t, Ts&... ts)

int size () const
    Return number of fields.

int size_field3d () const
    Return number of Field3Ds.

bool empty () const
    Test whether this group is empty.

void clear ()
    Remove all fields from this group.

iterator begin ()

iterator end ()

const_iterator begin () const

const_iterator end () const

const std::vector<FieldData *> &get () const

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

L.2.76 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.

L.2.77 File fieldperp.hxx

Functions

FieldPerp operator+ (const FieldPerp &lhs, const FieldPerp &rhs)
FieldPerp operator+ (const FieldPerp &lhs, const Field3D &rhs)
FieldPerp operator+ (const FieldPerp &lhs, const Field2D &rhs)
FieldPerp operator+ (const FieldPerp &lhs, BoutReal rhs)
FieldPerp operator+ (BoutReal lhs, const FieldPerp &rhs)
FieldPerp operator- (const FieldPerp &lhs, const FieldPerp &rhs)
FieldPerp operator- (const FieldPerp &lhs, const Field3D &rhs)
FieldPerp operator- (const FieldPerp &lhs, const Field2D &rhs)
FieldPerp operator- (const FieldPerp &lhs, BoutReal rhs)
FieldPerp operator- (BoutReal lhs, const FieldPerp &rhs)
FieldPerp operator* (const FieldPerp &lhs, const FieldPerp &rhs)
FieldPerp operator* (const FieldPerp &lhs, const Field3D &rhs)
FieldPerp operator* (const FieldPerp &lhs, const Field2D &rhs)
FieldPerp operator* (const FieldPerp &lhs, BoutReal rhs)
FieldPerp operator* (BoutReal lhs, const FieldPerp &rhs)
FieldPerp operator/ (const FieldPerp &lhs, const FieldPerp &rhs)
FieldPerp operator/ (const FieldPerp &lhs, const Field3D &rhs)
FieldPerp operator/ (const FieldPerp &lhs, const Field2D &rhs)
FieldPerp operator/ (const FieldPerp &lhs, BoutReal rhs)
FieldPerp operator/ (BoutReal lhs, const 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<>
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.

void checkData (const FieldPerp &f, const std::string &region = "RGN_NOX")
  Check if the data is valid.

void checkData (const FieldPerp &f, REGION region)
void invalidateGuards(FieldPerp &var)
    Force guard cells of passed field var to NaN.

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::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)
    Copy constructor. After this the data will be shared (non unique)

FieldPerp(FieldPerp &&rhs)
    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()

FieldPerp &operator=(const FieldPerp &rhs)
    Assignment operators

FieldPerp &operator=(FieldPerp &&rhs)

FieldPerp &operator=(BoutReal rhs)

const Region<IndPerp> &getRegion(REGION region) const
    Return a Region<IndPerp> reference to use to iterate over this field.

const Region<IndPerp> &getRegion(const std::string &region_name) const

Region<IndPerp>::RegionIndices::const_iterator begin() const

Region<IndPerp>::RegionIndices::const_iterator end() const

BoutReal &operator[](const IndPerp &d)

const BoutReal &operator[](const IndPerp &d) const

BoutReal &operator[](const Ind3D &d)

const BoutReal &operator[](const Ind3D &d) const

int getIndex() const
    Returns the y index at which this field is defined
FieldPerp &setIndex (int y)
    Sets the y index at which this field is defined
    This is used in arithmetic operations

FieldPerp &setLocation (CELL_LOC new_location)

FieldPerp &setDirectionY (YDirectionType d)

FieldPerp &allocate()
    Ensure that data array is allocated and unique

bool isAllocated() const
    True if the underlying data array is allocated.

const BoutReal *operator[] (int jx) const

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

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

const BoutReal &operator() (int jx, int jz) const
    Const (read-only) access to the underlying data array.

BoutReal &operator() (int jx, int jy, int jz)
    Access to the underlying data array. (X,Y,Z) indices for consistency with other field types

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)
Division, modifying in place. This loops over the entire domain, including guard/boundary cells

```
FieldPerp &operator/= (const FieldPerp &rhs)
```

```
FieldPerp &operator/= (const Field3D &rhs)
```

```
FieldPerp &operator/= (const Field2D &rhs)
```

```
FieldPerp &operator/= (BoutReal rhs)
```

```
int getNx () const
    Return the number of nx points
```

```
int getNy () const
    Return the number of ny points
```

```
int getNz () const
    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.
```

### L.2.78 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.

### L.2.79 File formatfactory.cxx

### L.2.80 File formatfactory.hxx

class FormatFactory : Uncopyable

**Public Functions**

```
std::unique_ptr<DataFormat> createDataFormat (const char *filename = nullptr, bool parallel = true, Mesh *mesh_in = nullptr)
```
Public Static Functions

`FormatFactory *getInstance()`  
Return a pointer to the only instance.

Private Functions

`int matchString(const char *str, int n, const char **match)`

Private Static Attributes

`FormatFactory *instance = nullptr`  
The only instance of this class (Singleton)

L.2.81 File fv_ops.cxx

L.2.82 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:  
d3f/dx3 ~= 16/5 f_b - 6 f_0 + 4 f_1 - 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  
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:  
d3f/dx3 ~= 16/5 f_b - 6 f_0 + 4 f_1 - 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**
- **f_in**: The field being advected. This will be reconstructed at cell faces using the given CellEdges method
- **v_in**: The advection velocity. This will be interpolated to cell boundaries using linear interpolation
- **wave_speed_in**: Local maximum speed of all waves in the system at each
- **fixflux**: 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)
```

**Parameters**
- **n_in**: The field being advected. This will be reconstructed at cell faces using the given CellEdges method
- **v**: The advection velocity. This will be interpolated to cell boundaries using linear interpolation

**Public Functions**

```
void operator() (Stencil1D &n)
```

```
struct Fromm
#include <fv_ops.hxx> Fromm method
```

```
Public Functions
void operator() (Stencil1D &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
void operator() (Stencil1D &n)
```

```
Private Functions
BoutReal minmod(BoutReal a, BoutReal b, BoutReal c)
```


```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

    void operator() (Stencil1D &n)

Private Functions

    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

    BoutReal c
    BoutReal m
    BoutReal p
    BoutReal mm
    BoutReal pp
    BoutReal L
    BoutReal R

struct Upwind
    #include <fv_ops.hxx> First order upwind for testing

Public Functions

    void operator() (Stencil1D &n)

L.2.83 File gen_fieldops.py

namespace gen_fieldops

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
(continues on next page)
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

```
smart_open(filename filename, mode = 'r')
```

From: http://stackoverflow.com/a/29824059/2043465

```
returnType(f1 f1, f2 f2) # Return type, by seeing which field is 'larger'.
```

Variables

```
gen_fieldops.operators = OrderedDict([(' *', 'multiplication'), (' /', 'division')])
gen_fieldops.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>'''
gен_fieldops.parser = argparse.ArgumentParser(description="Generate code for the Field arithmetic operators")
gен_fieldops.args = parser.parse_args()
gен_fieldops.index_var = 'index'
gен_fieldops.jz_var = 'jz'
gен_fieldops.mixed_base_ind_var = 'base_ind'
gен_fieldops.region_name = 'RGN_ALL'
gен_fieldops.region_loop = 'BOUT_FOR_SERIAL'
gen_fieldops.field3D = Field('Field3D', ['x', 'y', 'z'], index_var=index_var, jz_var = jz_var, mixed_base_ind_var = mixed_base_ind_var)
gen_fieldops.field2D = Field('Field2D', ['x', 'y'], index_var=index_var, jz_var = jz_var, mixed_base_ind_var = mixed_base_ind_var)
gen_fieldops.fieldPerp = Field('FieldPerp', ['x', 'z'], index_var=index_var, jz_var = jz_var, mixed_base_ind_var = mixed_base_ind_var)
gen_fieldops.boutreal = Field('BoutReal', [], index_var=index_var, jz_var = jz_var, mixed_base_ind_var = mixed_base_ind_var)
list gen_fieldops.fields = [field3D, field2D, fieldPerp, boutreal]
gen_fieldops.env = jinja2.Environment(loader=jinja2.FileSystemLoader('.'), trim_blocks=True)
gen_fieldops.template = env.get_template("gen_fieldops.jinja")
gen_fieldops.rhs = copy(rhs)
gen_fieldops.lhs = copy(lhs)
gen_fieldops.out = returnType(rhs, lhs)
```
class Field

Abstracts over BoutReals and Field2D/3D/Perps

Provides some helper functions for writing function signatures and passing data

**Public Functions**

```python
__init__(self, field_type: str, dimensions: int, name: str = None, index_var: str = None, jz_var: str = 'jz', mixed_base_ind_var: str = 'base_ind')

passByReference(self) except if field_type is BoutReal, in which case just returns "Type name"

index(self)[index_var]", except if field_type is BoutReal, in which case just returns ""

mixed_index(self) + {jz_var}]" if field_type is Field3D, self.index if Field2D or just returns "" for BoutReal

base_index(self)[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`

L.2.84 File generated_fieldops.cxx

**Functions**

```c
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 Field3D &rhs)
FieldPerp operator*(const Field3D &lhs, const Field2D &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)
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 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 Field3D &rhs)
Field2D operator+(const Field2D &lhs, const Field3D &rhs)
Field2D operator-(const Field2D &lhs, const Field3D &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)
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 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 BoutReal lhs, const FieldPerp &rhs)
FieldPerp operator/ (const BoutReal lhs, const FieldPerp &rhs)
FieldPerp operator+ (const BoutReal lhs, const FieldPerp &rhs)
FieldPerp operator- (const Field3D &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)
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)

L.2.85 File generator_context.cxx

namespace bout
    SNB model

L.2.86 File generator_context.hxx

namespace bout
    SNB model

    namespace generator

        class Context
Public Functions

```cpp
template<IND_TYPE N>
Context (const SpecificInd<N> &i, CELL_LOC loc, Mesh *msh, BoutReal t)
```
Set using an index. Can be Ind2D, Ind3D or IndPerp.

```cpp
Context (const Ind2D &i, CELL_LOC loc, Mesh *msh, BoutReal t)
```

```cpp
Context (int ix, int iy, int iz, CELL_LOC loc, Mesh *msh, BoutReal t)
```
Specify a cell index, together with the cell location, mesh and time

```cpp
Context ()
```
If constructed without parameters, contains no values (null). Requesting x,y,z or t should throw an exception

NOTE: For backward compatibility, all locations are set to zero. This should be changed in a future release.

```cpp
Context (const BoundaryRegion *bndry, int iz, CELL_LOC loc, BoutReal t, Mesh *msh)
```
The location on the boundary.

```cpp
Context (const BoundaryRegion *bndry, CELL_LOC loc, BoutReal t, Mesh *msh)
```

```cpp
BoutReal x() const
```

```cpp
BoutReal y() const
```

```cpp
BoutReal z() const
```

```cpp
BoutReal t() const
```

```cpp
Context &set (const std::string &name, BoutReal value)
```
Set the value of a parameter with given name.

```cpp
template<typename ...Args>
Context &set (const std::string &name, BoutReal value, Args... args)
```
Set multiple values, by passing alternating strings and values

eg. set("x", 1, "y", 2)

```cpp
BoutReal get (const std::string &name) const
```
Retrieve a value previously set.

```cpp
Mesh *getMesh () const
```
Get the mesh for this context (position) If the mesh is null this will throw a BoutException (if CHECK >= 1)

Private Members

```cpp
Mesh *localmesh = {nullptr}
```
The mesh on which the position is defined.

```cpp
std::map<std::string, BoutReal> bout::generator::Context::parameters{ "x", }
```
Contains user-set values which can be set and retrieved.
L.2.87 File generic_factory.hxx

Defines

__BOUT_GENERIC_FACTORY_H__

Base type for factories.

template<class BaseType, class DerivedFactory, class TypeCreator = std::function<std::unique_ptr<BaseType>(Options*)> 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. By default assumes the type can be created with an Options*

Uses static polymorphism (via CRTP) to overload static data. This is done by inheriting from this class and templating on the inherited class, and then providing four public data members:

Example:

class Base {
public:
    Base(Options*) {};
};
class Derived : public Base {
public:
    Derived(Options*) : Base({}) {};
};
class MyFactory : public Factory<Base, Derived> {
public:
    static constexpr auto type_name = "Base";
    static constexpr auto section_name = "base";
    static constexpr auto option_name = "type";
    static constexpr auto default_type = "derived_type";
};

RegisterInFactory<Base, Derived, MyFactory> register("derived_type");
auto foo = MyFactory::getInstance().create("derived_type");

In a .cxx file the static members should be declared:

cnstexpr decltype(MyFactory::type_name) MyFactory::type_name;
cnstexpr decltype(MyFactory::section_name) MyFactory::section_name;
cnstexpr decltype(MyFactory::option_name) MyFactory::option_name;

MIT Licence

Template Parameters

- BaseType: The base class that this factory creates
- DerivedFactory: The derived factory inheriting from this class
- TypeCreator: The function signature for creating a new BaseType

Public Functions

virtual ~Factory()
virtual bool add (const std::string & name, TypeCreator creator)  

Add a new type name to the factory  

Return true if the type was successfully added  

Parameters  

• name: An identifier for this type  

• creator: A function for creating this type  

virtual bool remove (const std::string & name)  

Remove a type name from the factory  

Return true if the type was successfully removed  

Parameters  

• name: The identifier for the type to be removed  

std::string getType (Options *options = nullptr) const  

Get the name of the type to create  

Looks in options first, then the root Options  

Return the name of the type to create  

Parameters  

• options: Options section to look for type name  

ReturnType create (Options *options = nullptr) const  

Create a new object using the type set in options.  

ReturnType create (const std::string & name) const  

Create a new object of type name using the root Options.  

template<typename ...Args>  

ReturnType create (const std::string & name, Args&&... args) const  

Create a new object of type name  

Return the new object  

Parameters  

• name: The identifier for the type to be created  

template<typename ...Args>  

ReturnType create (Options *options, Args&&... args) const  

Create a new object of the type given in options[“type”]  

Return the new object  

Parameters  

• options: The Options object to get the type to be created from  

std::vector<std::string> listAvailable () const  

List available types that can be created  

L.2. File list 467
Public Static Functions

static DerivedFactory & getInstance()  
Get the singleton instance.

static constexpr auto getDefaultType()  
Return the name of the default type to create.

static constexpr auto getSectionName()  
Return the name of the section to get from the root Options.

static constexpr auto getOptionName()  
Return the name of the Option value that sets the type.

Protected Types

template<>  
using ReturnType = typename TypeCreator::result_type

Protected Functions

Factory()  

Options *optionsOrDefaultSection(Options *options) const  
Return either options or the section from root.

Protected Attributes

std::map<std::string, TypeCreator> type_map  
Storage of the creation functions.

Protected Static Functions

static void ensureRegistered()  
Disgusting hack to defeat linker throwing out the registration symbols. If necessary, override this and put
the (empty) implementation in the same TU as the registration symbols

template<class BaseType, class DerivedType, class DerivedFactory>
class RegisterInFactory  
#include <generic_factory.hxx> Helper class for adding new types to Factory

See Factory for example

Adapted from http://www.drdobbs.com/conversations-abstract-factory-template/184403786

Template Parameters

- BaseType: Which factory to add DerivedType to
- DerivedType: The new type to add to Factory<BaseType>
**Public Functions**

`RegisterInFactory(const std::string &name)`

**L.2.88 File gettext.hxx**

Defines

```c
gettext(_)(string)
```

Support for i18n using GNU gettext.

**L.2.89 File globalfield.cxx**

**L.2.90 File globalfield.hxx**

```c
class GlobalField
#ifndef <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
#endif
```

**Public Functions**

```c
GlobalField()
```

```c
virtual ~GlobalField()
```

```c
virtual bool valid() const = 0
```

Is the data valid on any processor?

```c
bool dataIsLocal() const
```

Data is on this processor.

```c
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()`

```c
const BoutReal &operator()(int jx, int jy, int jz) const
```

Const data access by index.

```c
int xSize() const
```

Size of the field in X.

```c
int ySize() const
```

Size of the field in Y.

```c
int zSize() const
```

Size of the field in Z.

```c
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

```cpp
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:

```cpp
GlobalField3D g2d(mesh, 1); // Gather onto processor 1
```

Gather and scatter methods operate on **Field2D** objects:

```cpp
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:

```cpp
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.

```cpp
for (int x=0; x<g2d.xSize(); x++)
  for (int y=0; y<g2d.ySize(); y++)
    output.write(" Value at ({:d},{:d}) is {:e}\n",
               x, y, g2d(x, y));
```

### Public Functions

**GlobalField2D()**

Can’t be constructed without args.

**GlobalField2D(Mesh *mesh, int proc = 0)**

Construct, giving a mesh and an optional processor

- **Parameters**
  - `mesh`: The mesh to gather over
  - `proc`: The processor index where everything will be gathered/scattered to/from

**~GlobalField2D()**

Destructor.

**bool valid() const**

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.

**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`.

**BoutReal &operator() (int jx, int jy)**

Data access by global index.

**const BoutReal &operator() (int jx, int jy) const**
Private Functions

int msg_len (int proc) const
The length of message (in BoutReals) to be sent to or from processor proc

Parameters
• proc: 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?

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

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:

GlobalField3D g3d(mesh, 1);  // Gather onto processor 1

Gather and scatter methods operate on Field3D objects:

Field3D localdata;
g3d.gather(localdata);  // Gather onto one processor

To scatter data back, use the scatter method:

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:

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.

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));
Public Functions

**GlobalField3D()**
  Can’t be constructed without args.

**GlobalField3D(Mesh *mesh, int proc = 0)**
  Construct, giving a mesh and an optional processor
  
  **Parameters**
  
  • *mesh*: The mesh to gather over
  • *proc*: The processor index where everything will be gathered/scattered to/from

~GlobalField3D()
  Destructor.

**bool valid() const**
  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.

**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*: 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?

L.2.91 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

MpiWrapper *mpi
The MPI wrapper object.

Datafile dump
Dump file object.

L.2.92 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
Values:

X
Y
Z

Public Functions

GridDataSource (const bool source_is_file = false)

virtual ~GridDataSource ()
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, const std::string &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

constexpr Direction X = Direction::X
constexpr Direction Y = Direction::Y
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()

GridFile(std::unique_ptr<DataFormat> format, std::string gridfilename)
    Creates a GridFile object
    format Pointer to DataFormat. This will be deleted in destructor
~GridFile()

bool hasVar(const std::string &name)
Tests whether a variable exists in the file
Currently this is done by getting the variable’s size, and testing for zero size.

bool get (Mesh *m, std::string &sval, const std::string &name, const std::string &def = "")
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.

bool get (Mesh *m, int &ival, const std::string &name, int def = 0)
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
Returns
Boolean. True on success.

bool get (Mesh *m, BoutReal &rval, const std::string &name, BoutReal def = 0.0)
Get a BoutReal number.

bool get (Mesh *m, Field2D &var, const std::string &name, BoutReal def = 0.0)
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.

bool get (Mesh *m, Field3D &var, const std::string &name, BoutReal def = 0.0)

bool get (Mesh *m, FieldPerp &var, const std::string &name, BoutReal def = 0.0)

bool get (Mesh *m, std::vector<int> &var, const std::string &name, int len, int offset = 0, GridDataSource::Direction dir = GridDataSource::X)

bool get (Mesh *m, std::vector<BoutReal> &var, const std::string &name, int len, int offset = 0, GridDataSource::Direction dir = GridDataSource::X)

bool hasXBoundaryGuards (Mesh *m)
Are x-boundary guard cells read from the source?

bool hasYBoundaryGuards ()
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 **xd**, 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

**GridFromOptions** (**Options** *opt* = nullptr)

Constructor, passing optional **Options** object

Parameters

• **opt**: **Options** section to use as input. By default the “mesh” section under root will be used.

bool **hasVar** (**const** std::string &**name**)

Checks if the options has a given variable

bool **get** (**Mesh** *mesh*, std::string &**val**, **const** std::string &**name**, **const** std::string &**def** = "")

Reads strings from options. Uses **Options::get** to handle expressions
Return  True if option is set, false if ival is default (0)

Parameters

- mesh: Not used
- ival: The variable which will be set
- name: Name of variable
- def: Default value to use if option not found

bool get (Mesh *mesh, int &ival, const std::string &name, int def = 0)
Reads integers from options. Uses Options::get to handle expressions

Return  True if option is set, false if ival is default (0)

Parameters

- mesh: Not used
- ival: The variable which will be set
- name: Name of variable
- def: Default value to use if option not found

bool get (Mesh *mesh, BoutReal &rval, const std::string &name, BoutReal def = 0.0)
Reads BoutReal from options. Uses Options::get to handle expressions

Return  True if option is set, false if ival is default (0)

Parameters

- mesh: Not used
- name: Name of variable
- rval: Always given a value, defaults to 0
- def: Default value to use if option not found

bool get (Mesh *mesh, Field2D &var, const std::string &name, BoutReal def = 0.0)
Get a Field2D object by finding the option with the given name, and passing the string to FieldFactory

Parameters

- mesh: The Mesh object over which the field is defined
- var: The variable which will be set
- name: The name in the options. Not case sensitive
- def: Default value to use if option not found

bool get (Mesh *mesh, Field3D &var, const std::string &name, BoutReal def = 0.0)
Get a Field3D object by finding the option with the given name, and passing the string to FieldFactory

Parameters

- mesh: The Mesh object over which the field is defined
- var: The variable which will be set
- name: The name in the options. Not case sensitive
- def: Default value to use if option not found
bool get (Mesh *mesh, FieldPerp &var, const std::string &name, BoutReal def = 0.0)
Get a FieldPerp object by finding the option with the given name, and passing the string to FieldFactory

Parameters
• mesh: The Mesh object over which the field is defined
• var: The variable which will be set
• name: The name in the options. Not case sensitive
• def: Default value to use if option not found

bool get (Mesh *mesh, std::vector<int> &var, const std::string &name, int len, int offset = 0, GridDataSource::Direction dir = GridDataSource::X)
Get an array of integers. Currently reads a single integer, and sets the whole array to the same value

Parameters
• mesh: Mesh object
• var: A vector which will be resized to length len
• name: The name of the option
• len: The length of the vector
• offset: Not currently used
• dir: The direction (X,Y,Z) of the array

bool get (Mesh *mesh, std::vector<BoutReal> &var, const std::string &name, int len, int offset = 0, GridDataSource::Direction dir = GridDataSource::X)
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: Mesh object
• var: A vector which will be resized to length len
• name: The name of the option
• len: The length of the vector
• offset: The index where this vector starts i.e. var[0] is at x=offset if dir is X.
• dir: The direction (X,Y,Z) of the array

bool hasXBoundaryGuards (Mesh *m)
Are x-boundary guard cells read from the source?

bool hasYBoundaryGuards ()
Are y-boundary guard cells read from the source?

Private Members

Options *options
The options section to use. Could be nullptr.
Field3D gyroTaylor0 (const Field3D &f, const Field3D &rho)
Gyro-average using Taylor series approximation
\[ \Gamma(f) = f + \rho^2 \nabla^2 \perp(f) \]
Note: Faster, but less robust than Pade approximations

Parameters
- \( f \): The field to gyro-average
- \( \rho \): Gyro-radius

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 = GYRO_FLAGS, int outer_boundary_flags = GYRO_FLAGS)
Gyro-average using Pade approximation
\[ \Gamma_0 = (1 - \frac{1}{2} \rho^2 \nabla_\perp^2) g = f \]
NOTE: Uses Z average of rho for efficient inversion

Parameters
- \( f \): The field to gyro-average
- \( \rho \): Gyro-radius
- \( \text{inner}_\text{boundary}_\text{flags} \): Flags for the inner boundary to be passed to the Laplacian inversion operator
- \( \text{outer}_\text{boundary}_\text{flags} \): Flags for the outer boundary to be passed to the Laplacian inversion operator

Field3D gyroPadel1 (const Field3D &f, BoutReal rho, int inner_boundary_flags, int outer_boundary_flags)
Field3D gyroPadel1 (const Field3D &f, const Field2D &rho, int inner_boundary_flags, int outer_boundary_flags)
Field3D gyroPadel1 (const Field3D &f, const Field3D &rho, int inner_boundary_flags = GYRO_FLAGS, int outer_boundary_flags = GYRO_FLAGS)
Pade approximation \( \Gamma_{a1} = (1 - \frac{1}{2} \rho^2 \nabla_\perp^2) g = f \)
Note: Have to use Z average of rho for efficient inversion

Parameters
- \( f \): The field to gyro-average
• \( \rho \): Gyro-radius

• \texttt{inner\_boundary\_flags}: Flags for the inner boundary to be passed to the \textit{Laplacian} inversion operator

• \texttt{outer\_boundary\_flags}: Flags for the outer boundary to be passed to the \textit{Laplacian} inversion operator

\texttt{Field2D gyroPade1(const Field2D &f, const Field2D &rho, int inner\_boundary\_flags, int outer\_boundary\_flags)}

\texttt{Field3D gyroPade2(const Field3D &f, BoutReal rho, int inner\_boundary\_flags, int outer\_boundary\_flags)}

\texttt{Field3D gyroPade2(const Field3D &f, const Field2D &rho, int inner\_boundary\_flags, int outer\_boundary\_flags)}

\texttt{Field3D gyroPade2(const Field3D &f, const Field3D &rho, int inner\_boundary\_flags = GYRO\_FLAGS, int outer\_boundary\_flags = GYRO\_FLAGS)}

Pade approximation

\[ \Gamma_2(f) = \frac{1}{2} \rho^2 \nabla^2 \perp (1 - \frac{1}{2} \rho^2 \nabla^2)^{-1} \Gamma_1(f) \]

Note: Have to use Z average of rho for efficient inversion

**Parameters**

• \( f \): The field to gyro-average

**L.2.96 File gyro\_average.hxx**

Gyro-averaging operators

2010-09-03 Ben Dudson bd512@york.ac.uk

• Initial version, simple averaging operator

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/}. 

**L.2. File list**
Functions

Field3D gyroTaylor0 (const Field3D &f, const Field3D &rho)
Gyro-average using Taylor series approximation
\[ \Gamma(f) = f + \rho^2 \nabla^2_{\perp}(f) \]
Note: Faster, but less robust than Pade approximations

Parameters
- \(f\): The field to gyro-average
- \(\rho\): Gyro-radius

Field3D gyroPade0 (const Field3D &f, const Field3D &rho, int inner_boundary_flags = GYRO_FLAGS, int outer_boundary_flags = GYRO_FLAGS)
Gyro-average using Pade approximation
\[ \Gamma_0 = (1 - \frac{1}{2} \rho^2 \nabla^2_{\perp})g = f \]
NOTE: Uses Z average of rho for efficient inversion

Parameters
- \(f\): The field to gyro-average
- \(\rho\): Gyro-radius
- \textit{inner\_boundary\_flags}: Flags for the inner boundary to be passed to the Laplacian inversion operator
- \textit{outer\_boundary\_flags}: Flags for the outer boundary to be passed to the Laplacian inversion operator

Field3D gyroPade0 (const Field3D &f, const Field2D &rho, int inner_boundary_flags = GYRO_FLAGS, int outer_boundary_flags = GYRO_FLAGS)
Field3D gyroPade0 (const Field3D &f, BoutReal rho, int inner_boundary_flags = GYRO_FLAGS, int outer_boundary_flags = GYRO_FLAGS)
Field3D gyroPade1 (const Field3D &f, const Field3D &rho, int inner_boundary_flags = GYRO_FLAGS, int outer_boundary_flags = GYRO_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
- \(f\): The field to gyro-average
- \(\rho\): Gyro-radius
- \textit{inner\_boundary\_flags}: Flags for the inner boundary to be passed to the Laplacian inversion operator
- \textit{outer\_boundary\_flags}: Flags for the outer boundary to be passed to the Laplacian inversion operator

Field3D gyroPade1 (const Field3D &f, const Field2D &rho, int inner_boundary_flags = GYRO_FLAGS, int outer_boundary_flags = GYRO_FLAGS)
Field3D gyroPade1 (const Field3D &f, BoutReal rho, int inner_boundary_flags = GYRO_FLAGS, int outer_boundary_flags = GYRO_FLAGS)
Field2D gyroPade1 (const Field2D &f, const Field2D &rho, int inner_boundary_flags = GYRO_FLAGS, int outer_boundary_flags = GYRO_FLAGS)
Field3D gyroPade2 (const Field3D &f, const Field3D &rho, int inner_boundary_flags = GYRO_FLAGS, int outer_boundary_flags = GYRO_FLAGS)

Pade approximation

\[ \Gamma_2(f) = \frac{1}{2} \rho^2 \nabla^2_{\perp} (1 - \frac{1}{2} \rho^2 \nabla^2_{\perp})^{-1} \Gamma_1(f) \]

Note: Have to use Z average of rho for efficient inversion

Parameters

- \( f \): The field to gyro-average

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

L.2.97 File h5_format.cxx

L.2.98 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/.
Typedefs

using H5Format = EmptyFormat

L.2.99 File hermite_spline_xz.hxx

L.2.100 File hermite_spline_z.hxx

L.2.101 File ida.hxx

Defines

ZERO
ONE

Typedefs

using IDAINT = bout::utils::function_traits<IDABBDLocalFn>::arg_t<0>

Functions

static int idares (BoutReal t, N_Vector u, N_Vector du, N_Vector rr, void *user_data)
static int ida_bbd_res (IDAINT Nlocal, BoutReal t, N_Vector u, N_Vector du, N_Vector rr, void *user_data)
    Residual function for BBD preconditioner.
static int ida_pre (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 int ida_pre_shim (BoutReal t, N_Vector yy, N_Vector yp, N_Vector rr, N_Vector rvec, N_Vector zvec,
        BoutReal cj, BoutReal delta, void *user_data, N_Vector tmp)

L.2.102 File ida.hxx

class IdaSolver : public Solver

Public Functions

IdaSolver (Options *opts = nullptr)

~IdaSolver ()

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

int run ()
        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 \texttt{run} (BoutReal \texttt{tout})

void \texttt{res} (BoutReal \texttt{t}, BoutReal *\texttt{udata}, BoutReal *\texttt{ddata}, BoutReal *\texttt{rdata})

void \texttt{pre} (BoutReal \texttt{t}, BoutReal \texttt{cj}, BoutReal \texttt{delta}, BoutReal *\texttt{udata}, BoutReal *\texttt{rvec}, BoutReal *\texttt{zvec})

\textbf{Private Members}

\begin{verbatim}
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}
\end{verbatim}

\textbf{L.2.103 File identity.cxx}

\textbf{L.2.104 File imex-bdf2.cxx}

2nd order IMEX-BDF scheme

Scheme taken from this paper: \url{http://homepages.cwi.nl/~willem/DOCART/JCP07.pdf} W.Hundsdorfer, S.J.Ruuth
“IMEX extensions of linear multistep methods with general
monotonicity and boundedness properties” JCP 225 (2007) 2016-2042

Uses PETSc for the SNES interface

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/}.

\textbf{Defines}

\begin{verbatim}
#define \_FUNCT_
\end{verbatim}

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 \texttt{IMEXBDF2} object.
PETSc callback function for forming Jacobian
This function can be a linearised form of FormFunction
SNES callback for forming Jacobian with coloring
This can be a linearised and simplified form of FormFunction

___FUNCTION___

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.
PETSc callback function for forming Jacobian
This function can be a linearised form of FormFunction
SNES callback for forming Jacobian with coloring
This can be a linearised and simplified form of FormFunction

___FUNCTION___

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.
PETSc callback function for forming Jacobian
This function can be a linearised form of FormFunction
SNES callback for forming Jacobian with coloring
This can be a linearised and simplified form of FormFunction

___FUNCTION___

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.
PETSc callback function for forming Jacobian
This function can be a linearised form of FormFunction
SNES callback for forming Jacobian with coloring
This can be a linearised and simplified form of FormFunction

Functions

static PetscErrorCode FormFunction (SNES snes, Vec x, Vec f, void *ctx)
static PetscErrorCode FormFunctionForDifferencing (void *ctx, Vec x, Vec f)
static PetscErrorCode FormFunctionForColoring (SNES snes, Vec x, Vec f, void *ctx)
static PetscErrorCode imexbdf2PCapply (PC pc, Vec x, Vec y)

class SaveVarOp

Public Functions

SaveVarOp (Field2D *var, Field2D *F_var)
SaveVarOp (Field3D *var, Field3D *F_var)
void `run` (int `jx`, int `jy`, `BoutReal *u`)

void `run` (int `jx`, int `jy`, int `jz`, `BoutReal *u`)

**Private Members**

`Field2D *var2D`

`Field3D *var3D`

class `LoadVarOp`

**Public Functions**

`LoadVarOp (Field2D *var, Field2D *F_var)`

`LoadVarOp (Field3D *var, Field3D *F_var)`

void `run` (int `jx`, int `jy`, `BoutReal *u`)

void `run` (int `jx`, int `jy`, int `jz`, `BoutReal *u`)

**Private Members**

`Field2D *var2D`

`Field3D *var3D`

class `SaveDerivsOp`

**Public Functions**

`SaveDerivsOp (Field2D *var, Field2D *F_var)`

`SaveDerivsOp (Field3D *var, Field3D *F_var)`

void `run` (int `jx`, int `jy`, `BoutReal *u`)

void `run` (int `jx`, int `jy`, int `jz`, `BoutReal *u`)

**Private Members**

`Field2D *F_var2D`

`Field3D *F_var3D`
L.2.105 File imex-bdf2.hxx

class IMEXBDF2: public Solver
#include <imex-bdf2.hxx> IMEX-BDF2 time integration solver

Scheme taken from this paper: http://homepages.cwi.nl/~willem/DOCART/JCP07.pdf W.Hundsdorfer,
S.J.Ruuth “IMEX extensions of linear multistep methods with general
monotonicity and boundedness properties” JCP 225 (2007) 2016-2042
The method has been extended to variable order, variable timestep, and includes some adaptive capabilities

Public Functions

IMEXBDF2 (Options *opt = nullptr)

~IMEXBDF2 ()

BoutReal getCurrentTimestep ()
Returns the current internal timestep.

int init (int nout, BoutReal tstep)
Initialise solver. Must be called once and only once
Initialisation routine. Called once before solve.

Parameters
• nout: Number of outputs
• tstep: Time between outputs. NB: Not internal timestep

int run ()
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:

\[ f = (x - \gamma \cdot G(x)) - \text{rhs} \]

Parameters
• x: The state vector
• f: The vector for the result f(x)
• linear: 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: The vector to be operated on
• f: The result of the operation
Private Functions

```cpp
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).

Parameters
    • curtime: The current simulation time
    • dt: The time step to take
    • order: The order of accuracy

Inputs: u* - Solution history f* - Non-stiff component history
Outputs: u - Latest Solution f1 - Non-stiff time derivative at current time
```
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.

std::vector<BoutReal> uFac
std::vector<BoutReal> fFac
std::vector<BoutReal> gFac

BoutReal dtImp

int nlocal

int neq
    Number of variables on local processor and in total.

Array<BoutReal> u
    System state at current time.

std::vector<Array<BoutReal>> uV
    The solution history.

std::vector<Array<BoutReal>> fV
    The non-stiff solution history.

std::vector<BoutReal> timesteps
    Timestep history.

Array<BoutReal> rhs

Array<BoutReal> err

BoutReal implicit_gamma
**BoutReal** implicit_curtime

int predictor
    Predictor method.

**PetscLib** lib
    Handles initialising, finalising PETSc.

Vec snes_f
    Used by SNES to store function.

Vec snes_x
    Result of SNES.

SNES snes
    SNES context.

SNES snesAlt
    Alternative SNES object for adaptive checks.

SNES snesUse
    The snes object to use in solve stage. Allows easy switching.

Mat Jmf
    Matrix-free Jacobian.

bool 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**

constexpr int MAX_SUPPORTED_ORDER = 4

L.2.106 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::StandardFourth)
    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
    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_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), ... TypeContainer<Field2D> > 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<\textit{DIRECTION} direction, \textit{STAGGER} stagger, \textit{int} nGuards, \textit{typename} \textit{T}>
void standard(const \textit{T} &, \textit{T} &, const std::string) const

template<\textit{DIRECTION} direction, \textit{STAGGER} stagger, \textit{int} nGuards, \textit{typename} \textit{T}>
void upwindOrFlux (const \textit{T} &vel, const \textit{T} &var, \textit{T} &result, const std::string region) const

Public Static Attributes

const expr \textit{metaData} meta = {"SPLIT", 2, DERIV::Flux}

L.2.107 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.

L.2. File list 493
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**

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<typename Direction, typename Stagger, typename FieldTypeContainer, typename Method>
void operator() (Direction, Stagger, FieldTypeContainer, Method)

BoutReal apply (const stencil &f) const
BoutReal apply (BoutReal v, const stencil &f) const
BoutReal apply (const stencil &v, const stencil &f) const

Public Static Attributes

constexpr FF func = {}
constexpr metaData meta = {FF::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

L.2.108 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")

template<typename T>
T VDDX (const T &vel, 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 $[\text{\textit{\textbf{}}}]$ direction

$$v \frac{d}{dt} f$$

Parameters
- $v$: The velocity in the $Y$ direction
- $f$: The field being advected
- outloc: The cell location where the result is desired. The default is the same as $f$
- method: The differencing method to use
- region: The region of the grid for which the result is calculated.

template<typename T>
T FDDX (const T &vel, const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

template<typename T>
T VDDY (const T &vel, const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

template<typename T>
T FDDY (const T &vel, const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

template<typename T>
T VDDZ (const T &vel, const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

template<typename T>
T FDDZ (const T &vel, const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", const std::string &region = "RGN_NOBNDRY")

L.2.109 File initialprofiles.cxx

Functions

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.

**Parameters**

- **name**: The name of the field. This will be used as the section name in the options
- **var**: The field, which will be set to a value depending on the options

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)

```cpp
void initial_profile(const std::string &name, Field2D &var)
Set a Field2D from options
```

**Parameters**

- **name**: The name of the field, used as a section name
- **var**: 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.
```

**L.2.110 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.

**Parameters**

- **name**: The name of the field. This will be used as the section name in the options

```cpp
```
• var: The field, which will be set to a value depending on the options

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)

[p pressure]

[density] scale = 0.2

[vorticity] function = \cos(y)

initial_profile would generate:

- pressure -> \sin(y)
- density -> 0.2*\sin(y)
- vorticity -> \cos(y)

```cpp
void initial_profile(const std::string &name, Field2D &var)
```

Set a Field2D from options

Parameters

• name: The name of the field, used as a section name
• var: 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.

### L.2.111 File interpolation.hxx

#### Functions

**BoutReal interp(const stencil &s)**

Perform interpolation between centre -> shifted or vice-versa.

Interpolate using 4th-order staggered formula

Parameters

- s: Input stencil. mm -> -3/2, m -> -1/2, p -> +1/2, pp -> +3/2

```cpp
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. 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: Input variable
- loc: Location of output values
- region: Region where output will be calculated

```cpp
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)
```

```cpp
const char *strLocation (CELL_LOC loc)
```

### L.2.112 File interpolation_xz.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)

### L.2.113 File interpolation_xz.hxx

#### Functions

- 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)

#### Public Functions

- XZInterpolation (int y_offset = 0, Mesh *localmeshIn = nullptr)
- XZInterpolation (const BoutMask &mask, int y_offset = 0, Mesh *mesh = nullptr)
virtual ~XZInterpolation()

void setMask (const BoutMask &mask)

virtual void calcWeights (const Field3D &delta_x, const Field3D &delta_z, const std::string &region = "RGN_NOBNDRY") = 0

virtual void calcWeights (const Field3D &delta_x, const Field3D &delta_z, const BoutMask &mask, const std::string &region = "RGN_NOBNDRY") = 0

virtual Field3D interpolate (const Field3D &f, const std::string &region = "RGN_NOBNDRY") const = 0

virtual Field3D interpolate (const Field3D &f, const Field3D &delta_x, const Field3D &delta_z, const std::string &region = "RGN_NOBNDRY") = 0

virtual Field3D interpolate (const Field3D &f, const Field3D &delta_x, const Field3D &delta_z, const BoutMask &mask, const std::string &region = "RGN_NOBNDRY") = 0

void setYOffset (int offset)

virtual std::vector<ParallelTransform::PositionsAndWeights> getWeightsForYUpApproximation (int i, int j, int k)

virtual std::vector<ParallelTransform::PositionsAndWeights> getWeightsForYDownApproximation (int i, int j, int k)

virtual std::vector<ParallelTransform::PositionsAndWeights> getWeightsForYApproximation (int i, int j, int k, int yoffset)

**Public Members**

int y_offset

**Protected Attributes**

Mesh *localmesh = {nullptr}

BoutMask skip_mask

class XZHermiteSpline : public XZInterpolation
  Subclassed by XZMonotonicHermiteSpline
Public Functions

XZHermiteSpline (Mesh *mesh = nullptr)

XZHermiteSpline (int y_offset = 0, Mesh *mesh = nullptr)

XZHermiteSpline (const BoutMask &mask, int y_offset = 0, Mesh *mesh = nullptr)

void calcWeights (const Field3D &delta_x, const Field3D &delta_z, const std::string &region = "RGN_NOBNDRY")

void calcWeights (const Field3D &delta_x, const Field3D &delta_z, const BoutMask &mask, const std::string &region = "RGN_NOBNDRY")

Field3D interpolate (const Field3D &f, const std::string &region = "RGN_NOBNDRY")

const Field3D interpolate (const Field3D &f, const Field3D &delta_x, const Field3D &delta_z, const std::string &region = "RGN_NOBNDRY")

const Field3D interpolate (const Field3D &f, const Field3D &delta_x, const Field3D &delta_z, const BoutMask &mask, const std::string &region = "RGN_NOBNDRY")

std::vector<ParallelTransform::PositionsAndWeights> getWeightsForYApproximation (int i, int j, int k, int y_offset)

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 XZMonotonicHermiteSpline : public XZHermiteSpline
#include <interpolation_xz.hxx> Monotonic Hermite spline interpolator

Similar to XZHermiteSpline, 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**

XZMonotonicHermiteSpline (Mesh *mesh = nullptr)
XZMonotonicHermiteSpline (int y_offset = 0, Mesh *mesh = nullptr)
XZMonotonicHermiteSpline (const BoutMask &mask, int y_offset = 0, Mesh *mesh = nullptr)

Field3D interpolate (const Field3D &f, const std::string &region = "RGN_NOBNDRY")

Interpolate using precalculated weights. This function is called by the other interpolate functions in the base class XZHermiteSpline.

class XZLagrange4pt : public XZInterpolation

**Public Functions**

XZLagrange4pt (Mesh *mesh = nullptr)
XZLagrange4pt (int y_offset = 0, Mesh *mesh = nullptr)
XZLagrange4pt (const BoutMask &mask, int y_offset = 0, Mesh *mesh = nullptr)

void calcWeights (const Field3D &delta_x, const Field3D &delta_z, const std::string &region = "RGN_NOBNDRY")
void calcWeights (const Field3D &delta_x, const Field3D &delta_z, const BoutMask &mask, const std::string &region = "RGN_NOBNDRY")

Field3D interpolate (const Field3D &f, const std::string &region = "RGN_NOBNDRY")

Field3D interpolate (const Field3D &f, const Field3D &delta_x, const Field3D &delta_z, const std::string &region = "RGN_NOBNDRY")

Field3D interpolate (const Field3D &f, const Field3D &delta_x, const Field3D &delta_z, const BoutMask &mask, const std::string &region = "RGN_NOBNDRY")

BoutReal lagrange_4pt (BoutReal v2m, BoutReal vm, BoutReal vp, BoutReal v2p, BoutReal offset)
BoutReal lagrange_4pt (const BoutReal v[], BoutReal offset) const

**Private Members**

Tensor<int> i_corner
Tensor<int> k_corner
Field3D t_x
Field3D t_z
class XZBilinear : public XZInterpolation

Public Functions

XZBilinear (Mesh *mesh = nullptr)
XZBilinear (int y_offset = 0, Mesh *mesh = nullptr)
XZBilinear (const BoutMask &mask, int y_offset = 0, Mesh *mesh = nullptr)

void calcWeights (const Field3D &delta_x, const Field3D &delta_z, const std::string &region = "RGN_NOBNDRY")
void calcWeights (const Field3D &delta_x, const Field3D &delta_z, const BoutMask &mask, const std::string &region = "RGN_NOBNDRY")

Field3D interpolate (const Field3D &f, const std::string &region = "RGN_NOBNDRY")
const
Field3D interpolate (const Field3D &f, const Field3D &delta_x, const Field3D &delta_z, const std::string &region = "RGN_NOBNDRY")
Field3D interpolate (const Field3D &f, const Field3D &delta_x, const Field3D &delta_z, const BoutMask &mask, const std::string &region = "RGN_NOBNDRY")

Private Members

Tensor<int> i_corner
Tensor<int> k_corner
Field3D w0
Field3D w1
Field3D w2
Field3D w3

class XZInterpolationFactory : public Factory<XZInterpolation, XZInterpolationFactory, std::function<unique_ptr<XZInterpolation>(Mesh*)>
Public Static Attributes

```cpp
constexpr auto type_name = "XZInterpolation"
constexpr auto section_name = "xzinterpolation"
constexpr auto option_name = "type"
constexpr auto default_type = "hermitespline"
```

template<class DerivedType>
class RegisterXZInterpolation

Public Functions

```cpp
RegisterXZInterpolation (const std::string &name)
```

L.2.114 File interpolation_z.cxx

L.2.115 File interpolation_z.hxx

class ZInterpolation
    Subclassed by ZHermiteSpline

Public Functions

```cpp
ZInterpolation (int y_offset = 0, Mesh *mesh = nullptr, Region<Ind3D> region_in = {})
virtual ~ZInterpolation ()
virtual void calcWeights (const Field3D &delta_z) = 0
virtual Field3D interpolate (const Field3D &f, const std::string &region_str = "DEFAULT") const = 0
virtual Field3D interpolate (const Field3D &f, const Field3D &delta_z, const std::string &region_str = "DEFAULT") = 0
void setRegion (Region<Ind3D> new_region)
virtual std::vector<ParallelTransform::PositionsAndWeights> getWeightsForYUpApproximation (int i, int j, int k) const
virtual std::vector<ParallelTransform::PositionsAndWeights> getWeightsForYDownApproximation (int i, int j, int k) const
```
virtual std::vector<ParallelTransform::PositionsAndWeights> getWeightsForYApproximation (int i, int j, int k, int yoffset) const

**Public Members**

const int y_offset

**Protected Attributes**

Mesh *localmesh = {nullptr}
Region<Ind3D> region

**class ZInterpolationFactory : public Factory<ZInterpolation, ZInterpolationFactory, std::function<std::unique_ptr<ZInterpolation>(int, Mesh *, Region<Ind3D> *region_in)>>**

**Public Functions**

ReturnType create (Options *options, int y_offset = 0, Mesh *mesh = nullptr, Region<Ind3D> region_in = { })
ReturnType create (int y_offset = 0, Mesh *mesh = nullptr, Region<Ind3D> region_in = { })

**Public Static Functions**

void ensureRegistered ()

**Public Static Attributes**

constexpr auto type_name = "ZInterpolation"
constexpr auto section_name = "zinterpolation"
constexpr auto option_name = "type"
constexpr auto default_type = "hermitespline"

template<class DerivedType>
class RegisterZInterpolation
Public Functions

RegisterZInterpolation (const std::string &name)

class ZHermiteSpline: public ZInterpolation

Public Functions

ZHermiteSpline (int y_offset = 0, Mesh *mesh = nullptr, Region<Ind3D> region_in = {})

void calcWeights (const Field3D &delta_z)

Field3D interpolate (const Field3D &f, const std::string &region_str = "DEFAULT") const

Field3D interpolate (const Field3D &f, const Field3D &delta_z, const std::string &region_str = "DEFAULT")

std::vector<ParallelTransform::PositionsAndWeights> getWeightsForYApproximation (int i, int j, int k, int yoffset)

const

Return position and weight of points needed to approximate the function value at the point that the field line through (i,j,k) meets the (j+1)-plane. For the case where only the z-direction is not aligned to grid points, the approximation is: f(i,j+1,k*) = h00 * f(i,j+1,k) + h01 * f(i,j+1,k+1) + h10 * dfdz(i,j+1,k) + h11 * dfdz(i,j+1,k+1) = h00 * f(i,j+1,k) + h01 * f(i,j+1,k+1) + h10 * ( f(i,j+1,k+1) - f(i,j+1,k-1) ) / 2 + h11 * ( f(i,j+1,k+2) - f(i,j+1,k) ) / 2 for k* a point between k and k+1. Therefore, this function returns position weight h11 / 2

Private Members

const std::string fz_region

Array<Ind3D> k_corner

Field3D h00

Field3D h01

Field3D h10

Field3D h11

L.2.116 File invert_laplace.cxx

Perpendicular Laplacian inversion using FFT and Tridiagonal solver.

Equation solved is d * \nabla^2 x + (1./c)\nabla \cdot \text{erpc} \cdot \nabla x + ax = b, where x and \nabla 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
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.

L.2.117 File invert_laplace.hxx

Perpendicular Laplacian inversion using FFT and Tridiagonal solver

Equation solved is: \( d \cdot \nabla^2 \perp x + \frac{1}{c} \nabla_p e r p c \cdot \nabla_\perp x + ax = b \)

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++.

This file 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

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.
Variables

```cpp
constexpr auto LAPLACE_SPT = "spt"
constexpr auto LAPLACE_PDD = "pdd"
constexpr auto LAPLACE_TRI = "tri"
constexpr auto LAPLACE_BAND = "band"
constexpr auto LAPLACE_PETSC = "petsc"
constexpr auto LAPLACE_PETSCAMG = "petscamg"
constexpr auto LAPLACE_PETSC3DAMG = "petsc3damg"
constexpr auto LAPLACE_CYCLIC = "cyclic"
constexpr auto LAPLACE_SHOOT = "shoot"
constexpr auto LAPLACE_MULTIGRID = "multigrid"
constexpr auto LAPLACE_NAULIN = "naulin"
constexpr int INVERT_DC_GRAD = 1
    Zero-gradient for DC (constant in Z) component. Default is zero value.
constexpr int INVERT_AC_GRAD = 2
    Zero-gradient for AC (non-constant in Z) component. Default is zero value.
constexpr int INVERT_AC_LAP = 4
    Use zero-laplacian (decaying solution) to AC component.
constexpr int INVERT_SYM = 8
    Use symmetry to enforce either zero-value or zero-gradient.
constexpr int INVERT_SET = 16
    Set boundary to value.
constexpr int INVERT_RHS = 32
    Use input value in RHS boundary.
constexpr int INVERT_DC_LAP = 64
    Use zero-laplacian solution for DC component.
constexpr int INVERT_BNDRY_ONE = 128
    Only use one boundary point.
constexpr int INVERT_DC_GRADPAR = 256
constexpr int INVERT_DC_GRADPARINV = 512
constexpr int INVERT_IN_CYLINDER = 1024
    For use in cylindrical coordinate system.
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 LaplaceFactory : public Factory<Laplacian, LaplaceFactory, std::function<std::unique_ptr<Laplacian>(Options*, CELL_LOC, Mesh*)>>

Public Functions
ReturnType create(Options *options = nullptr, CELL_LOC loc = CELL_CENTRE, Mesh *mesh = nullptr)

Public Static Attributes
constexpr auto type_name = "Laplacian"
constexpr auto section_name = "laplace"
constexpr auto option_name = "type"
constexpr auto default_type = LAPLACE_CYCLIC

template<class DerivedType>
class RegisterLaplace
#include <invert_laplace.hxx> Simpler name for Factory registration helper class

Usage:
#include <bout/laplacefactory.hxx>
namespace {
    RegisterLaplace<MyLaplace> registerlaplacemine("mylaplace");
}

Public Functions

RegisterLaplace(const std::string &name)

class Laplacian
#include <invert_laplace.hxx> Base class for Laplacian inversion.

Subclassed by LaplaceCyclic, LaplaceMultigrid, LaplaceNaulin, LaplacePDD, LaplacePetsc, LaplacePetsc3dAmg, LaplaceSerialBand, LaplaceSerialTri, LaplaceShoot, LaplaceSPT

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()

virtual void setCoefA(const Field2D &val) = 0
Set coefficients for inversion. Re-builds matrices if necessary.
virtual void setCoefA(const Field3D &val)
virtual void setCoefA(BoutReal r)
virtual void setCoefC(const Field2D &val) = 0
virtual void setCoefC(const Field3D &val)
virtual void setCoefC(BoutReal r)
virtual void setCoefC1(const Field2D &val)
virtual void setCoefC1(const Field3D &val)
virtual void setCoefC1(BoutReal r)
virtual void setCoefC2(const Field2D &val)
virtual void setCoefC2(const Field3D &val)
virtual void setCoefC2(BoutReal r)
virtual void setCoefD(const Field2D &val) = 0
virtual void setCoefD(const Field3D &val)
virtual void setCoefD(BoutReal r)
virtual void setCoefEx(const Field2D &val) = 0
virtual void setCoefEx(const Field3D &val)
virtual void setCoefEx(BoutReal r)
virtual void setCoefEz(const Field2D &val) = 0
virtual void setCoefEz(const Field3D &val)
virtual void setCoefEz(BoutReal r)
virtual void setGlobalFlags(int f)
virtual void setInnerBoundaryFlags(int f)
virtual void setOuterBoundaryFlags(int f)
virtual bool uses3DCoefs() const
  Does this solver use Field3D coefficients (true) or only their DC component (false)

virtual FieldPerp solve(const FieldPerp &b) = 0
Field3D solve(const Field3D &b)
Field2D solve(const Field2D &b)
virtual FieldPerp solve(const FieldPerp &b, const FieldPerp &x0)
Field3D solve(const Field3D &b, const Field3D &x0)
  Performs the laplacian inversion y-slice by y-slice

  Return x All the y-slices of x_slice in the equation A*x_slice = b_slice
Parameters

- **b**: All the y-slices of b_slice, which is the right hand side of the equation A*x_slice = b_slice
- **x0**: All the y-slices of the variable eventually used to set BC

Field2D solve(const Field2D &b, const Field2D &x0)

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 std::unique_ptr<Laplacian> create(Options *opts = nullptr, const CELL_LOC location = CELL Centre, Mesh *mesh_in = nullptr)

Create a new Laplacian solver

Parameters

- **opt**: The options section to use. By default “laplace” will be used

Laplacian *defaultInstance ()

Return pointer to global singleton.

void cleanup ()

Frees all memory.

Protected Functions

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 *c2coef, const Field2D *d, CELL_LOC loc = CELL_DEFAULT)

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 *ccoef, 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)

Set the matrix components of A in Ax=b

This function will

1. Calling tridagCoef, solving
   
   \[
   D \cdot \text{Laplace}_\text{perp}(x) + \frac{1}{C1} \cdot \text{Grad}_\text{perp}(C2) \cdot \text{Grad}_\text{perp}(x) + Ax = B
   \]

   for each fourier component
2. Set the boundary conditions by setting the first and last rows properly

**Parameters**

- *avec*: Lower diagonal of the tridiagonal matrix. DO NOT CONFUSE WITH “A”
- *bvec*: The main diagonal
- *cvec*: The upper diagonal. DO NOT CONFUSE WITH “C” (called ccoef here)
- *bk*: The b in Ax = b
- *jy*: Index of the current y-slice
- *kz*: The mode number index
- *kwave*: 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*: Global flags of the inversion
- *inner_boundary_flags*: Flags used to set the inner boundary
- *outer_boundary_flags*: Flags used to set the outer boundary
- *a*: A in the equation above. DO NOT CONFUSE WITH avec
- *c1coef*: C1 in the equation above. DO NOT CONFUSE WITH cvec
- *c2coef*: C2 in the equation above. DO NOT CONFUSE WITH cvec
- *d*: D in the equation above
- *includeguards*: Whether or not the guard points in x should be used
- *avec*: Lower diagonal of the tridiagonal matrix. DO NOT CONFUSE WITH “A”
- *bvec*: The main diagonal
- *cvec*: The upper diagonal. DO NOT CONFUSE WITH “C” (called ccoef here)

**Protected Attributes**

- **bool async_send**
  - If true, use asyncronous 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

std::unique_ptr<Laplacian> instance = nullptr
    Singleton instance.

L.2.118 File invert_parderiv.cxx

L.2.119 File invert_parderiv.hxx

Variables

constexpr auto PARDERIVCYCLIC = "cyclic"

class InvertParFactory : public Factory<InvertPar, InvertParFactory, std::function<std::unique_ptr<InvertPar> (Options *, CELL_LOC, Mesh *)>>

Public Functions

ReturnType create (Options *options = nullptr, CELL_LOC location = CELL_CENTRE, Mesh *mesh = nullptr)

Public Static Functions

void ensureRegistered ()
Public Static Attributes

constexpr auto type_name = "InvertPar"
constexpr auto section_name = "parderiv"
constexpr auto option_name = "type"
constexpr auto default_type = PARDERIVCYCLIC

template<class DerivedType>
class RegisterInvertPar

Public Functions

RegisterInvertPar (const std::string &name)

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
auto inv = InvertPar::Create(); inv->setCoefA(1.0); inv->setCoefB(-0.1);
Field3D result = inv->solve(rhs);

Subclassed by InvertParCR

Public Functions

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 ()

cost 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

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

virtual const Field3D solve (const Field3D &f, const Field3D &start)

virtual void setCoefA (const Field2D &f) = 0
Set the constant coefficient A

virtual void setCoefA (const Field3D &f)

virtual void setCoefA (BoutReal f)
virtual void setCoefB(const Field2D &f) = 0
    Set the Grad2_par2 coefficient B

virtual void setCoefB(const Field3D &f)

virtual void setCoefB(BoutReal f)

virtual void setCoefC(const Field2D &f) = 0
    Set the D2DYDZ coefficient C

virtual void setCoefC(const Field3D &f)

virtual void setCoefC(BoutReal f)

virtual void setCoefD(const Field2D &f) = 0
    Set the D2DZ2 coefficient D

virtual void setCoefD(const Field3D &f)

virtual void setCoefD(BoutReal f)

virtual void setCoefE(const Field2D &f) = 0
    Set the DDY coefficient E

virtual void setCoefE(const Field3D &f)

virtual void setCoefE(BoutReal f)

Public Static Functions

static std::unique_ptr<InvertPar> create(Options *opt_in = nullptr, CELL_LOC location_in = CELL_CENTRE, Mesh *mesh_in = nullptr)
    Create an instance of InvertPar

Protected Attributes

CELL_LOC location

Mesh *localmesh
    Mesh object for this solver.

L.2.120 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 $A \cdot x = b$

**Public Types**

template<>
using data_type = T
    What type of field does the operator take?

template<>
using function_signature = std::function<T(const T &)> 
    The signature of the functor that applies the operator.

**Public Functions**

InvertableOperator (const function_signature &func = identity<T>, Options *optIn = nullptr, Mesh *localmeshIn = nullptr)
    Almost empty constructor currently don’t actually use Options for anything.

~InvertableOperator()
    Destructor just has to cleanup the PETSc owned objects.

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.

void setPreconditionerFunction (const function_signature &func)
    Allow the user to override the existing preconditioner function.

T operator() (const T &input)
    Provide a way to apply the operator to a Field.

T apply (const T &input)
    Provide a synonym for applying the operator to a Field.

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).

T invert (const T &rhsField, const T &guess)
T invert (const T &rhsField)
    Triggers the solve of $A \cdot x = b$ 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.
bool verify(const T &rhsIn, BoutReal tol = 1.0e-5)
With checks enabled provides a convenience routine to check that applying the registered function on the calculated inverse gives back the initial values.

Public Static Functions

static 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

Options *opt = nullptr
Mesh *localmesh = nullptr
bool doneSetup = false
PetscLib lib

Private Static Functions

static 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 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.

L.2.121 File karniadakis.cxx

L.2.122 File karniadakis.hxx

class KarniadakisSolver : public Solver
**Public Functions**

*KarniadakisSolver (Options *options)*

~KarniadakisSolver()

*BoutReal getCurrentTimestep ()*

Return the current internal timestep.

```cpp
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
```

```cpp
int run ()
    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.
```

void resetInternalFields ()

    Should wipe out internal field vector and reset from current field object data.

**Private Functions**

```cpp
void take_step (BoutReal dt)
```

**Private Members**

```cpp
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
```
L.2.123 File lagrange_4pt_xz.cxx

L.2.124 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 \(a[0\ldots(n-1)][0\ldots(m1+m2)]\) and the rhs vector \(b[0\ldots(n-1)]\)
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

```c
int tridag(const dcomplex *, const dcomplex *, const dcomplex *, const dcomplex *, dcomplex *, int)
Tri-diagonal complex matrix inversion.

bool tridag(const BoutReal *, const BoutReal *, const BoutReal *, const BoutReal *, BoutReal *, int)
Tri-diagonal matrix inversion (BoutReal)

void cyclic_tridag(BoutReal *, BoutReal *, BoutReal *, BoutReal *, BoutReal *, int)
Solve a cyclic tridiagonal matrix.

void cband_solve (Matrix<dcomplex>&, int, int, int, Array<dcomplex>&)
Complex band matrix solver.

void cyclic_tridag(dcomplex *a, dcomplex *b, dcomplex *c, dcomplex *r, dcomplex *x, int n)
Solve a cyclic tridiagonal matrix.
```

L.2.125 File lapack_routines.hxx

Functions

```c
int tridag (const dcomplex *, const dcomplex *, const dcomplex *, const dcomplex *, dcomplex *, int)
Tri-diagonal complex matrix inversion.

bool tridag (const BoutReal *, const BoutReal *, const BoutReal *, const BoutReal *, BoutReal *, int)
Tri-diagonal matrix inversion (BoutReal)
```
void cyclic_tridag(BoutReal *a, BoutReal *b, BoutReal *c, BoutReal *r, BoutReal *x, int n)
    Solve a cyclic tridiagonal matrix.

void cyclic_tridag(dcomplex *a, dcomplex *b, dcomplex *c, dcomplex *r, dcomplex *x, int n)
    Solve a cyclic tridiagonal matrix.

cband_solve(Matrix<dcomplex> &a, int n, int m1, int m2, Array<dcomplex> &b)
    Complex band matrix solver.

L.2.126 File laplacexy.cxx

 Defines

 static PetscErrorCode laplacePCapply(PC pc, Vec x, Vec y)

L.2.127 File laplacexy.hxx

 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 generality, this routine does use globalIndex() in the inner loop, although this may be slightly less efficient than incrementing an integer for the global index, the finite-volume and finite-difference implementations have slightly different indexing patterns, so incrementing an integer would be tricky.
void **savePerformance**(Datafile &outputfile, Solver &solver, std::string name = "")

If this method is called, save some performance monitoring information

**Private Functions**

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.

void **setPreallocationFiniteVolume**(PetscInt *d_nnz, PetscInt *o_nnz)

void **setPreallocationFiniteDifference**(PetscInt *d_nnz, PetscInt *o_nnz)

void **setMatrixElementsFiniteVolume**(const Field2D &A, const Field2D &B)

void **setMatrixElementsFiniteDifference**(const Field2D &A, const Field2D &B)

void **solveFiniteVolume**(const Field2D &x0)

void **solveFiniteDifference**(const Field2D &x0)

**Private Members**

*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.

std::string **default_prefix**

default prefix for writing performance logging variables

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

Friends

friend LaplaceXYMonitor

class LaplaceXYMonitor : public Monitor

Public Functions

LaplaceXYMonitor (LaplaceXY &owner)

int call (Solver *, BoutReal, int, int)

Private Members

LaplaceXY &laplacey
L.2.128 File laplacexz-cyclic.hxx

class LaplaceXZcyclic: public LaplaceXZ

Public Functions

LaplaceXZcyclic (Mesh *m = nullptr, Options *options = nullptr, const CELL_LOC loc = CELL_CENTRE)

LaplaceXZcyclic ()

void setCoefs (const Field2D &A, const Field2D &B)

Field3D solve (const Field3D &b, const Field3D &x0)

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> k1d
Array<dcomplex> k1d_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.

L.2.130 File laplacexz-petsc.hxx

L.2.131 File laplacexz-petsc.hxx

class LaplaceXZpetsc: public LaplaceXZ
Public Functions

LaplaceXZpetsc(Mesh *m = nullptr, Options *options = nullptr, const CELL_LOC loc = CELL_CENTRE)

~LaplaceXZpetsc()

void setCoefs(const Field3D &A, const Field3D &B)
void setCoefs(const Field2D &A, const Field2D &B)
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

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.

L.2.132 File laplacexz.cxx

L.2.133 File laplacexz.hxx

class LaplaceXZFactory : public Factory<LaplaceXZ, LaplaceXZFactory, std::function<std::unique_ptr<LaplaceXZ>(Mesh *, Options *, CELL_LOC)>>

Public Functions
ReturnType create(Mesh *mesh = nullptr, Options *options = nullptr, CELL_LOC loc = CELL_CENTRE)

Public Static Functions
void ensureRegistered()

Public Static Attributes
constexpr auto type_name = "LaplaceXZ"
constexpr auto section_name = "laplacexz"
constexpr auto option_name = "type"
constexpr auto default_type = "cyclic"

template<class DerivedType>
class RegisterLaplaceXZ

Public Functions
RegisterLaplaceXZ(const std::string &name)

class LaplaceXZ
Subclassed by LaplaceXZcyclic, LaplaceXZpetsc

Public Functions
LaplaceXZ(Mesh *m = nullptr, Options *options = nullptr, const CELL_LOC loc = CELL_CENTRE)
virtual ~LaplaceXZ()

virtual void setCoefs(const Field2D &A, const Field2D &B) = 0
virtual void setCoefs(const Field3D &A, const Field3D &B)
virtual Field3D solve(const Field3D &b, const Field3D &x0) = 0

Public Static Functions

static std::unique_ptr<LaplaceXZ> create(Mesh *m = nullptr, Options *opt = nullptr, CELL_LOC loc = CELL_CENTRE)

Protected Attributes

Mesh *localmesh
The mesh this operates on, provides metrics and communication.

CELL_LOC location

Protected Static Attributes

const int INVERT_DC_GRAD = 1
const int INVERT_AC_GRAD = 2
const int INVERT_SET = 16
const int INVERT_RHS = 32
When called with $\text{VA_ARGS}$ first, this evaluates to an argument which depends on the length of $\text{VA_ARGS}$. This is used to find the appropriate macro to begin the expansion.

\textbf{MACRO\_FOR\_EACH} (mac, ...)

Apply a macro (first argument) to each of the following arguments. Currently supports up to 10 arguments.

Example:

\texttt{MACRO\_FOR\_EACH(test, a, b, c)}

expands to

\texttt{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.

\textbf{MACRO\_FOR\_EACH\_FN} (fn, ...)

Apply a function (first argument) to each of the following arguments. Currently supports up to 10 arguments.

Example:

\texttt{MACRO\_FOR\_EACH\_FN(test, a, b, c)}

expands to

\texttt{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.

\textbf{L.2.135 File mask.hxx}

\texttt{class BoutMask}

\texttt{\#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
// 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

BoutMask (int nx, int ny, int nz, bool value = false)
BoutMask (const Mesh &mesh, bool value = false)
BoutMask (const Mesh *mesh = nullptr, bool value = false)
BoutMask &operator= (bool value)
bool &operator() (int jx, int jy, int jz)
const bool &operator() (int jx, int jy, int jz) const

Private Members

Tensor<bool> mask

L.2.136 File mesh.cxx

L.2.137 File mesh.hxx

Interface for mesh classes. Contains standard variables and useful routines.

Changelog
2014-12 Ben Dudson bd512@york.ac.uk
• Removing coordinate system into separate Coordinates class
• Adding index derivative functions from derivs.cxx
2010-06 Ben Dudson, Sean Farley
• Initial version, adapted from GridData class
• Incorporates code from topology.cpp and Communicator

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++ Documentation, Release 5.0.0-alpha

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

```cpp
using comm_handle = void *
// Type used to return pointers to handles.
```

```cpp
class MeshFactory : public Factory<Mesh, MeshFactory, std::function<std::unique_ptr<Mesh>(GridDataSource *, Options *)>>
```

**Public Functions**

```cpp
MeshFactory::ReturnType create(Options *options = nullptr, GridDataSource *source = nullptr)
```

**Public Static Attributes**

```cpp
constexpr auto type_name = "Mesh"
constexpr auto section_name = "mesh"
constexpr auto option_name = "type"
constexpr auto default_type = "bout"
```

```cpp
template<class DerivedType>
class RegisterMesh
```

**Public Functions**

```cpp
RegisterMesh(const std::string &name)
```

**class Mesh**

Subclassed by *BoutMesh*

**Public Functions**

```cpp
Mesh()
// Constructor for a “bare”, uninitialised Mesh Only useful for testing
```

```cpp
Mesh(GridDataSource *s, Options *options)
// Constructor
```

**Parameters**

- *s*: The source to be used for loading variables
- *options*: The options section for settings
~Mesh()
   Destructor.

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

virtual void outputVars(Datafile &file)
   Add output variables to a data file These are used for post-processing

int get (std::string &sval, const std::string &name, const std::string &def = "")
   Get a string from the input source

   Return zero if successful, non-zero on failure

   Parameters
      • sval: The value will be put into this variable
      • name: The name of the variable to read
      • def: The default value if not found

int get (int &ival, const std::string &name, int def = 0)
   Get an integer from the input source

   Return zero if successful, non-zero on failure

   Parameters
      • ival: The value will be put into this variable
      • name: The name of the variable to read
      • def: The default value if not found

int get (BoutReal &rval, const std::string &name, BoutReal def = 0.0)
   Get a BoutReal from the input source

   Return zero if successful, non-zero on failure

   Parameters
      • rval: The value will be put into this variable
      • name: The name of the variable to read
      • def: The default value if not found

int get (bool &bval, const std::string &name, bool def = false)
   Get a bool from the input source

   Return zero if successful, non-zero on failure

   Parameters
      • bval: The value will be put into this variable
      • name: The name of the variable to read
int \texttt{get} (\texttt{Field2D} &\texttt{var}, \texttt{const} std::string &\texttt{name}, \texttt{BoutReal} \texttt{def} = 0.0)

Get a Field2D from the input source including communicating guard cells

\textbf{Return} zero if successful, non-zero on failure

\textbf{Parameters}

- var: This will be set to the value. Will be allocated if needed
- name: Name of the variable to read
- def: The default value if not found

int \texttt{get} (\texttt{Field3D} &\texttt{var}, \texttt{const} std::string &\texttt{name}, \texttt{BoutReal} \texttt{def} = 0.0, \texttt{bool} \texttt{communicate} = \texttt{true})

Get a Field3D from the input source

\textbf{Return} zero if successful, non-zero on failure

\textbf{Parameters}

- var: This will be set to the value. Will be allocated if needed
- name: Name of the variable to read
- def: The default value if not found
- communicate: Should the field be communicated to fill guard cells?

int \texttt{get} (\texttt{FieldPerp} &\texttt{var}, \texttt{const} std::string &\texttt{name}, \texttt{BoutReal} \texttt{def} = 0.0, \texttt{bool} \texttt{communicate} = \texttt{true})

Get a FieldPerp from the input source

\textbf{Return} zero if successful, non-zero on failure

\textbf{Parameters}

- var: This will be set to the value. Will be allocated if needed
- name: Name of the variable to read
- def: The default value if not found
- communicate: Should the field be communicated to fill guard cells?

int \texttt{get} (\texttt{Vector2D} &\texttt{var}, \texttt{const} std::string &\texttt{name}, \texttt{BoutReal} \texttt{def} = 0.0)

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

\textbf{Return} zero always.

\textbf{Parameters}

- var: This will be set to the value read
- name: The name of the vector. Individual fields are read based on this name by appending. See above
- def: The default value if not found (used for all the components)
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.

Return  zero always.

Parameters

• var: This will be set to the value read
• name: The name of the vector. Individual fields are read based on this name by appending. See above
• def: The default value if not found (used for all the components)

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>
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>
void communicateXZ (Ts&... ts)

template<typename ...Ts>
void communicateYZ (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 communicateYZ (FieldGroup &g)
Communicate guard cells in YZ only i.e. no X 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>
comm_handle send (Ts... ts)
Send a list of FieldData objects. Packs arguments into a FieldGroup and passes to send(FieldGroup&).

template<typename ...Ts>
comm_handle sendX (Ts... ts)
Send guard cells from a list of FieldData objects in the x-direction. Packs arguments into a FieldGroup and passes to send(FieldGroup&).

template<typename ...Ts>
comm_handle sendY (Ts... ts)
Send guard cells from a list of FieldData objects in the y-direction. 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.

Return handle to be used as input to wait()

Parameters
• g: Group of fields to communicate

virtual comm_handle sendX (FieldGroup &g, comm_handle handle = nullptr, bool disable_corners = false) = 0
Send only the x-guard cells.

virtual comm_handle sendY (FieldGroup &g, comm_handle handle = nullptr) = 0
Send only the y-guard cells.

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: X index of processor to send to
• yproc: Y index of processor to send to
• buffer: A buffer of data to send
• size: The length of buffer
• tag: 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: X index of sending processor
• yproc: Y index of sending processor
• buffer: The buffer to fill with data. Must already be allocated of length size
• size: The length of buffer
• tag: 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() const = 0
Is this processor first in X? i.e. is there a boundary to the left in X?

virtual bool lastX() const = 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: The data to send. Must be at least length size
• size: The number of BoutReals to send
• tag: 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: The data to send. Must be at least length size
• size: The number of BoutReals to send
• tag: A label for the communication. Must be the same at receive

virtual comm_handle irecvXOut(BoutReal *buffer, int size, int tag) = 0
Receive a buffer of data from X index +1

Parameters

• buffer: A buffer to put the data in. Must already be allocated of length size
• size: The number of BoutReals to receive and put in buffer
• tag: A label for the communication. Must be the same as sent
virtual comm_handle irecvXIn (BoutReal *buffer, int size, int tag) = 0
Receive a buffer of data from X index -1

Parameters
• buffer: A buffer to put the data in. Must already be allocated of length size
• size: The number of BoutReals to receive and put in buffer
• tag: A label for the communication. Must be the same as sent

MPI_Comm getXcomm ()
Return communicator containing all processors in X.

virtual MPI_Comm getXcomm (int jy) const = 0
Return X communicator.

virtual MPI_Comm getYcomm (int jx) const = 0
Return Y communicator.

MpiWrapper &getMpi ()
Return pointer to the mesh’s MPI Wrapper object.

bool periodicY (int jx) const
Is local X index jx periodic in Y?

Parameters
• jx: The local (on this processor) index in X

virtual bool periodicY (int jx, BoutReal &ts) const = 0
Is local X index jx periodic in Y?

Parameters
• jx: The local (on this processor) index in X
• ts: The Twist-Shift angle if periodic

virtual int numberOfYBoundaries () const = 0
Get number of boundaries in the y-direction, i.e. locations where there are boundary cells in the global grid

virtual std::pair<bool, BoutReal> hasBranchCutLower (int jx) const = 0
Is there a branch cut at this processor’s lower y-boundary?

Return 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

Parameters
• jx: The local (on this processor) index in X

virtual std::pair<bool, BoutReal> hasBranchCutUpper (int jx) const = 0
Is there a branch cut at this processor’s upper y-boundary?

Return 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
Parameters

- \(jx\): The local (on this processor) index in X

```cpp
type ySize(int jx) const
  The number of points in Y at fixed X index \(jx\).
```

```cpp
virtual bool firstY() const = 0
  Is this processor first in Y? i.e. is there a boundary at lower Y?
```

```cpp
virtual bool lastY() const = 0
  Is this processor last in Y? i.e. is there a boundary at upper Y?
```

```cpp
virtual bool firstY(int xpos) const = 0
  Is this processor first in Y? i.e. is there a boundary at lower Y?
```

```cpp
virtual bool lastY(int xpos) const = 0
  Is this processor last in Y? i.e. is there a boundary at upper Y?
```

```cpp
virtual int UpXSplitIndex() = 0
  If the upper Y guard cells are split in two, return the X index where the split occurs.
```

```cpp
virtual int DownXSplitIndex() = 0
  If the lower Y guard cells are split in two, return the X index where the split occurs.
```

```cpp
virtual int sendYOutIndest(BoutReal *buffer, int size, int tag) = 0
  Send data.
```

```cpp
virtual int sendYOutOutdest(BoutReal *buffer, int size, int tag) = 0
```

```cpp
virtual int sendYInIndest(BoutReal *buffer, int size, int tag) = 0
```

```cpp
virtual int sendYInOutdest(BoutReal *buffer, int size, int tag) = 0
```

```cpp
virtual comm_handle irecvYOutIndest(BoutReal *buffer, int size, int tag) = 0
  Non-blocking receive. Must be followed by a call to `wait()`
```

Parameters

- \(buffer\): A buffer of length \(size\) which must already be allocated
- \(size\): The number of BoutReals expected
- \(tag\): The tag number of the expected message

```cpp
virtual comm_handle irecvYOutOutdest(BoutReal *buffer, int size, int tag) = 0
  Non-blocking receive. Must be followed by a call to `wait()`
```

Parameters

- \(buffer\): A buffer of length \(size\) which must already be allocated
- \(size\): The number of BoutReals expected
- \(tag\): The tag number of the expected message

```cpp
virtual comm_handle irecvYInIndest(BoutReal *buffer, int size, int tag) = 0
  Non-blocking receive. Must be followed by a call to `wait()`
```

Parameters
virtual comm_handle irecvYInOutdest (BoutReal *buffer, int size, int tag) = 0
Non-blocking receive. Must be followed by a call to wait()

Parameters

buffer: A buffer of length size which must already be allocated
size: The number of BoutReals expected
tag: The tag number of the expected message

virtual const RangeIterator iterateBndryLowerY () const = 0
Iterate over the lower Y boundary.

virtual const RangeIterator iterateBndryUpperY () const = 0
Iterate over the upper Y boundary.

virtual const RangeIterator iterateBndryLowerOuterY () const = 0
virtual const RangeIterator iterateBndryLowerInnerY () const = 0
virtual const RangeIterator iterateBndryUpperOuterY () const = 0
virtual const RangeIterator 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.

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.

virtual void addBoundaryPar (BoundaryRegionPar *bndry)
Add a parallel(Y) boundary to this processor.

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)

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.

int YGLOBAL (int yloc) const
    Returns the global Y index given a local index if the local index must include the boundary, the global index does not.

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.

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.

Coordinates *getCoordinates(const CELL_LOC location = CELL_CENTRE)
Coordinate system.

std::shared_ptr<Coordinates> getCoordinatesSmart(const CELL_LOC location = CELL_CENTRE)

CELL_LOC getAllowedStaggerLoc(DIRECTION direction) const
Returns the non-CELL_CENTRE location allowed as a staggered location

int getNpoints(DIRECTION direction) const
Returns the number of grid points in the particular direction

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>
T indexDDX(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

template<typename T>
T indexD2DX2(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

template<typename T>
T indexD4DX4(const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

540 Appendix L. API reference
template<typename T>
T indexDDZ (const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

template<typename T>
T indexD2DZ2 (const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

template<typename T>
T indexD4DZ4 (const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

template<typename T>
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 \( \mathbb{D} \) direction

\[
\frac{d}{dt} f
\]

Parameters

- \( v \): The velocity in the Y direction
- \( f \): The field being advected
- \( \text{outloc} \): The cell location where the result is desired. The default is the same as \( f \)
- \( \text{method} \): The differencing method to use
- \( \text{region} \): The region of the grid for which the result is calculated.

template<typename T>
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>
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>
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>
T indexVDDZ (const T &vel, const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

template<typename T>
T indexFDDZ (const T &vel, const T &f, CELL_LOC outloc = CELL_DEFAULT, const std::string &method = "DEFAULT", REGION region = RGN_NOBNDRY) const

const Field3D toFieldAligned (const Field3D &f, const REGION region = RGN_ALL)

const Field3D fromFieldAligned (const Field3D &f, const REGION region = RGN_ALL)

const Field2D toFieldAligned (const Field2D &f, const REGION region = RGN_ALL)

const Field2D fromFieldAligned (const Field2D &f, const REGION region = RGN_ALL)
bool canToFromFieldAligned()

void setParallelTransform(std::unique_ptr<ParallelTransform> pt)

void setParallelTransform()

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

const Region &getRegion(const std::string &region_name) const

const Region &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

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

void addRegion(const std::string &region_name, const Region<Ind2D> &region)

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)

Ind3D ind2Dto3D(const Ind2D &ind2D, int jz = 0)

Converts an Ind2D to an Ind3D using calculation.

Ind2D ind3Dto2D(const Ind3D &ind3D)

Converts an Ind3D to an Ind2D using calculation.

IndPerp ind3DtoPerp(const Ind3D &ind3D)

Converts an Ind3D to an IndPerp using calculation.

Ind3D indPerpTo3D(const IndPerp &indPerp, int jy = 0)

Converts an IndPerp to an Ind3D using calculation.

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<>
    const Region<Ind3D> &getRegion (const std::string &region_name) const

    template<>
    const Region<Ind2D> &getRegion (const std::string &region_name) const

    template<>
    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 GlobalNxNoBoundaries
int GlobalNyNoBoundaries
int GlobalNzNoBoundaries
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
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

const bool include_corner_cells

Public Static Functions

Mesh *create (GridDataSource *source, Options *opt = nullptr)
   Create a Mesh object

   Parameters
   • source: The data source to use for loading variables
   • opt: The option section. By default this is “mesh”

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: 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)

MpiWrapper *mpi = nullptr

Pointer to the global MPI wrapper, for convenience.

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
int localNumCells3D = -1
int localNumCells2D = -1
int localNumCellsPerp = -1

L.2.138 File monitor.hxx

Functions

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

Monitor (BoutReal timestep_ = -1)

A timestep_ of -1 defaults to the the frequency of the BOUT++ output monitor

virtual ~Monitor()
virtual int call(Solver *solver, BoutReal time, int iter, int nout) = 0
Callback function for the solver, called after timestep_ has passed.

Return non-zero if simulation should be stopped.

Parameters
- solver: The solver calling this monitor
- time: The current simulation time
- iter: The current simulation iteration
- nout: The total number of iterations for this simulation

virtual void cleanup()
Callback function for when a clean shutdown is initiated.

Protected Functions

BoutReal getTimestep() const
Get the currently set timestep for this monitor.

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 Solver
needs access to timestep and freq

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

\textit{BoutReal} \texttt{t\_elapsed} = 0
  cumulative wall clock time in seconds
\textit{BoutReal} \texttt{wtime} = 0
  time step’s wall clock time in seconds
\texttt{int} \texttt{ncalls} = 0
  number of RHS calls
\texttt{int} \texttt{ncalls\_e} = 0
  number of RHS calls for fast timescale
\texttt{int} \texttt{ncalls\_i} = 0
  number of RHS calls for slow timescale
\textit{BoutReal} \texttt{wtime\_rhs} = 0
  wall time spent calculating RHS
\textit{BoutReal} \texttt{wtime\_invert} = 0
  wall time spent inverting \textit{Laplacian}
\textit{BoutReal} \texttt{wtime\_comms} = 0
  wall time spent communicating (part of RHS)
\textit{BoutReal} \texttt{wtime\_io} = 0
  wall time spent on I/O
\textit{BoutReal} \texttt{wtime\_per\_rhs} = 0
  wall time per RHS evaluation
\textit{BoutReal} \texttt{wtime\_per\_rhs\_e} = 0
  wall time per fast timescale RHS evaluation
\textit{BoutReal} \texttt{wtime\_per\_rhs\_i} = 0
  wall time per slow timescale RHS evaluation

L.2.139 File \textit{monotonic\_hermite\_spline\_xz.cxx}

L.2.140 File \textit{mpi\_wrapper.hxx}

class MpiWrapper
  \texttt{#include <mpi\_wrapper.hxx>} Provides wrappers around MPI functions, taking the same names. These can then be overloaded for testing purposes.

Public Functions

\texttt{virtual \sim MpiWrapper()} 

\texttt{virtual int MPI\_Abort (MPI\_Comm comm, int errorcode)} 

\texttt{virtual int MPI\_Allreduce(const void \*sendbuf, void \*recvbuf, int count, MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm)} 

\texttt{virtual int MPI\_Barrier (MPI\_Comm comm)} 

\texttt{virtual int MPI\_Comm\_create (MPI\_Comm comm, MPI\_Group group, MPI\_Comm \*newcomm)
virtual int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
virtual int MPI_Comm_free(MPI_Comm *comm)
virtual int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
virtual int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)
virtual int MPI_Comm_rank(MPI_Comm comm, int *rank)
virtual int MPI_Comm_size(MPI_Comm comm, int *size)
virtual int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
virtual int MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int *recvcouents, const int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)
virtual int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
virtual int MPI_Group_free(MPI_Group *group)
virtual int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)
virtual int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
virtual int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
virtual int MPI_Scan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
virtual int MPI_Scatterv(const void *sendbuf, const int *sendcounts, const int *displs, MPI_Datatype sendtype, void *recvbuf, const int *recvcouents, MPI_Datatype recvtype, int root, MPI_Comm comm)
virtual int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
virtual int MPI_Type_commit(MPI_Datatype *datatype)
virtual int MPI_Type_free(MPI_Datatype *datatype)
virtual int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
virtual int MPI_Wait(MPI_Request *request, MPI_Status *status)
virtual int MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status array_of_statuses[])
virtual int MPI_Waitany(int count, MPI_Request array_of_requests[], int *indx, MPI_Status *status)
virtual double MPI_Wtime()
L.2.141 File msg_stack.cxx

L.2.142 File msg_stack.hxx

Defines

__thefunc__

The PRETTY_FUNCTION variable is defined by GCC (and some other families) but is not a part of the standard. The 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 PRETTY_FUNCTION is defined or not. Instead we need to say if we support this or not by defining BOUT_HASPRETTY_FUNCTION (to be implemented in configure)

GLOBAL

This is a way to define a global object, so that it is declared extern in all files except one where GLOBALORIGIN is defined.

CONCATENATE_DIRECT (s1, s2)

To concatenate strings for a variable name.

CONCATENATE (s1, s2)

Need to use two levels due to macro strangeness.

TRACE (...)

The TRACE macro provides a convenient way to put messages onto the msg_stack It pushes a message onto the stack, and pops it when the scope ends

Example

{ TRACE(“Starting calculation”)
 } // Scope ends, message popped

AUTO_TRACE ()

The AUTO_TRACE macro provides a convenient way to put messages onto the 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 PRETTY_FUNCTION if available else it will be the mangled form.

This is implemented as a use of the TRACE macro with specific arguments.

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()`

`~MsgStack()`

`int push (std::string message)`

`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 push ()`

`int push (std::nullptr_t)`

`template<class S, class ... Args>`

`int push (const S &format, const Args&... args)`

`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

`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

MsgStackItem(std::string message)
MsgStackItem(const std::string &message, const char * file, int line)
MsgStackItem(const std::string &file, int line, const char *msg)
template<class S, class ...Args>
MsgStackItem(const std::string &file, int line, const S &msg, const Args&... args)
~MsgStackItem()

Private Members

int point

L.2.143 File multigrid_alg.cxx

L.2.144 File multigrid_laplace.cxx

Variables

BoutReal soltime = 0.0
BoutReal settime = 0.0

L.2.145 File multigrid_laplace.hxx

Defines

MAXGM

class MultigridAlg
    Subclassed by Multigrid1D, Multigrid2DPf1D, MultigridSerial

Public Functions

    MultigridAlg(int level, int lx, int lz, int gx, int gz, MPI_CComm comm, int check)
    ~MultigridAlg()
    void setMultigridC(int plag)
    void getSolution(BoutReal *x, BoutReal *b, int flag)
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 communications (BoutReal *x, int level)
void setMatrixC (int level)
void cycleMG (int level, BoutReal *sol, BoutReal *rhs)
void smoothings (int level, BoutReal *x, BoutReal *b)
void projection (int level, BoutReal *r, BoutReal *pr)
void prolongation (int level, BoutReal *x, BoutReal *ix)
void pGMRES (BoutReal *sol, BoutReal *rhs, int level, int iplag)
void solveMG (BoutReal *sol, BoutReal *rhs, int level)
void multiAVec (int level, BoutReal *x, BoutReal *b)
void residualVec (int level, BoutReal *x, BoutReal *b, BoutReal *r)
BoutReal vectorProd (int level, BoutReal *x, BoutReal *y)
void lowestSolver (BoutReal *x, BoutReal *b, int plag)
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)
~MultigridSerial ()
void convertMatrixF (BoutReal *)

class Multigrid2DPf1D : public MultigridAlg

Public Functions

Multigrid2DPf1D (int level, int lx, int lz, int gx, int gz, int dl, int px, int pz, MPI_Comm comm, int check)
~Multigrid2DPf1D ()
void setMultigridC (int plag)
void setPcheck (int check)
void setValueS ()

Public Members

int kflag

Private Functions

void convertMatrixFS (int level)
void lowestSolver (BoutReal *x, BoutReal *b, int plag)
Private Members

std::unique_ptr<MultigridSerial> sMG

class Multigrid1DP : public MultigridAlg

Public Functions

Multigrid1DP (int level, int lx, int lz, int gx, int dl, int merge, MPI_Comm comm, int check)

~Multigrid1DP()

void setMultigridC (int plag)

void setPcheck (int check)

void setValueS ()

Public Members

int kflag

Private Functions

void convertMatrixF2D (int level)

void convertMatrixFS (int level)

void lowestSolver (BoutReal *, BoutReal *, int plag)

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)

~LaplaceMultigrid ()

void setCoefA (const Field2D &val)

Set coefficients for inversion. Re-builds matrices if necessary.

void setCoefC (const Field2D &val)

void setCoefC1 (const Field2D &val)

void setCoefC2 (const Field2D &val)
void setCoefD(const Field2D &val)
void setCoefEx(const Field2D &val)
void setCoefEz(const Field2D &val)
void setCoefA(const Field3D &val)
void setCoefC(const Field3D &val)
void setCoefC1(const Field3D &val)
void setCoefC2(const Field3D &val)
void setCoefD(const Field3D &val)

bool uses3DCoefs() const
    Does this solver use Field3D coefficients (true) or only their DC component (false)

FieldPerp solve(const FieldPerp &b)
FieldPerp solve(const FieldPerp &b_in, const FieldPerp &x0)

Private Functions

void generateMatrixF(int level)

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<MG1DP> kMG
int mglvlel
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

L.2.146 File multigrid_solver.cxx

L.2.147 File multiostream.hxx

Typedefs

using cmultiostream = multiostream<char>
using wmultiostream = multiostream<wchar_t>

template<typename char_type, typename traits = std::char_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

void add (std::basic_ostream<char_type, traits> &str)
void remove (std::basic_ostream<char_type, traits> &str)

Protected Functions

std::streamsize xputn (const char_type *sequence, std::streamsize num)
int overflow (int c)

Private Types

template<>
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

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 : multioutbuf_init<char_type, traits>, public std::basic_ostream<char_type, traits>

Public Functions

multiostream()

void add (std::basic_ostream<char_type, traits> &str)

void remove (std::basic_ostream<char_type, traits> &str)

Private Types

template<>
using multioutbuf_init = multioutbuf_init<char_type, traits>

L.2.148 File naulin_laplace.cxx

Iterative solver to handle non-constant-in-z coefficients.

Scheme suggested by Volker Naulin: solve Delp2(phi[i+1]) + 1/DC(C1*D)*Grad_perp(DC(C2))*Grad_perp(phi[i+1]) + DC(A/D)*phi[i+1] = rhs(phi[i]) + 1/DC(C1*D)*Grad_perp(DC(C2))*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)!=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++ Documentation, Release 5.0.0-alpha

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:

\[
\begin{align*}
\Omega^D &= \nabla \cdot (n \nabla_\perp \phi) \\
&= n \nabla_\perp^2 \phi + \nabla n \cdot \nabla_\perp \phi \\
&= n \Omega + \nabla n \cdot \nabla_\perp \phi \\
&= n \Omega + \nabla_\perp n \cdot \nabla_\perp \phi \quad (12.1)
\end{align*}
\]

Rearranging gives

\[
\begin{align*}
\nabla \cdot (n \nabla_\perp \phi) &= n \nabla_\perp^2 \phi + \nabla n \cdot \nabla_\perp \phi \\
&= n \Omega + \nabla n \cdot \nabla_\perp \phi \\
&= n \Omega + \nabla_\perp n \cdot \nabla_\perp \phi
\end{align*}
\]
In fact we allow for the slightly more general form

\[ \nabla^2 \phi + \langle A_D \rangle \phi = \frac{\text{rhs}}{D} - \frac{1}{D C_1} \nabla_{\perp} C_2 \cdot \nabla_{\perp} \phi - \langle \frac{A}{D} < \frac{A}{D} > \rangle \ast \phi \] (12.1)

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

   \[
   \text{vort} = \frac{\text{vortD}}{n} - \text{grad_perp}(\ln_n) \times \text{grad_perp}(\phi_{\text{Cur}})
   \]

   

   \[
   [D \cdot \text{Del}_2(\phi_{\text{Next}}) + 1/DC(C2) \times \text{grad_perp}(DC(C2)) \times \text{grad_perp}(\phi_{\text{Next}}) + DC(A/D) \times \phi_{\text{Next}}]
   \]

   

   \[
   = b(\phi_{\text{Cur}})
   \]

   

   \[
   = (\text{rhs}/D) - (1/C1 \times \text{grad_perp}(C2) \times \text{grad_perp}(\phi_{\text{Cur}}) - 1/DC(C2) \times \text{grad_perp}(DC(C2)) \times \text{grad_perp}(\phi_{\text{Next}}) - (A/D - DC(A/D)) \times \phi_{\text{Cur}})
   \]

   

   where \( \phi_{\text{Cur}} \) is \( \phi \) of the current iteration [and DC(f) is the constant-in-z component of f]

2. Invert \( \phi \) to find the vorticity using

   \[
   \phi_{\text{Next}} = \text{invert_laplace_perp}(\text{vort})
   \]

   

   \[
   \text{set Acoef of laplace_perp solver to DC(A/D)}
   \]

   

   \[
   \text{and C1coef of laplace_perp solver to DC(C1/D)}
   \]

   

   \[
   \text{and C2coef of laplace_perp solver to DC(C2)}
   \]

   

   \[
   \text{then } \phi_{\text{Next}} = \text{invert_laplace_perp(underrelax_factor \times b(\phi_{\text{Cur}}) - (1-underrelax_factor) \times b(\phi_{\text{Prev}}))}
   \]

   

   where \( b(\phi_{\text{Prev}}) \) is the previous rhs value, which (up to rounding errors) is the same as the lhs of the direct solver applied to \( \phi_{\text{Cur}} \).

3. Calculate the error at \( \phi=\phi_{\text{Next}} \)

   \[
   \text{error3D} = D \cdot \text{Del}_2(\phi_{\text{Next}}) + 1/DC(C2) \times \text{grad_perp}(DC(C2)) \times \text{grad_perp}(\phi_{\text{Next}}) + DC(A/D) \times \phi_{\text{Next}} - \text{rhs}/D
   \]

   

   as \( b(\phi_{\text{Cur}}) = D \cdot \text{Del}_2(\phi_{\text{Next}}) + 1/DC(C2) \times \text{grad_perp}(DC(C2)) \times \text{grad_perp}(\phi_{\text{Next}}) + DC(A/D) \times \phi_{\text{Next}} \)

   

   up to rounding errors

4. Calculate the infinity norms of the error

   \[
   E\text{AbsLInf} = \max(\text{error3D})
   \]

   

   E\text{RelLInf} = E\text{AbsLInf}/\sqrt{\max((\text{rhs}/D)^2)}

5. Check whether

   \[
   E\text{AbsLInf} > \text{atol}
   \]

   

   • If yes

     \[
     \text{Check whether}
     \]

     \[
     E\text{RelLInf} > \text{rtol}
     \]

     

   • If yes

     \[
     \text{Check whether}
     \]

     \[
     E\text{AbsLInf} > E\text{AbsLInf(\text{previous step})}
     \]

     

   • 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

L.2.149 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 ()

void setCoefA (const Field2D &val)
  Set coefficients for inversion. Re-builds matrices if necessary.

void setCoefA (const Field3D &val)
void setCoefC (const Field2D &val)
void setCoefC (const Field3D &val)
void setCoefC1 (const Field3D &val)
void setCoefC1 (const Field2D &val)
void setCoefC2 (const Field3D &val)
void setCoefC2 (const Field2D &val)
void setCoefD (const Field3D &val)
void setCoefD (const Field2D &val)
void setCoefEx (const Field2D &val)
void setCoefEz (const Field2D &val)
bool uses3DCoefs() const

Does this solver use Field3D coefficients (true) or only their DC component (false)

FieldPerp solve(const FieldPerp &b)

FieldPerp solve(const FieldPerp &b, const FieldPerp &x0)

Field3D solve(const Field3D &b)

Field3D solve(const Field3D &b, const Field3D &x0)

Performs the laplacian inversion y-slice by y-slice

Return  x  All the y-slices of x_slice in the equation A*x_slice = b_slice

Parameters

•  b: All the y-slices of b_slice, which is the right hand side of the equation A*x_slice = b_slice
•  x0: All the y-slices of the variable eventually used to set BC

Field3D solve(const Field3D &b)

void setGlobalFlags(int f)

void setInnerBoundaryFlags(int f)

void setOuterBoundaryFlags(int f)

BoutReal getMeanIterations() const

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

std::unique_ptr<Laplacian> delp2solver = {nullptr}

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 \texttt{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 \texttt{naulinsolver\_mean\_its}

Mean number of iterations taken by the solver.

BoutReal \texttt{naulinsolver\_mean\_underrelax\_counts} = \{0.\}

Mean number of times the underrelaxation factor is reduced.

\texttt{int ncalls}

Counter for the number of times the solver has been called.

L.2.150 File nc\_format.cxx

L.2.151 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)

NcFormat (const std::string &name, Mesh *mesh\_in = nullptr)

~NcFormat ()

bool openr (const char *name)

bool openw (const char *name, bool append = false)
bool is_valid()
void close()
void flush()

const char *filename()

const vector<int> getSize(const char *var)
const vector<int> getSize(const std::string &var)

bool setGlobalOrigin(int x = 0, int y = 0, int z = 0)
bool setLocalOrigin(int x = 0, int y = 0, int z = 0, int offset_x = 0, int offset_y = 0, int offset_z = 0)
bool setRecord(int t)

bool addVarInt(const std::string &name, bool repeat)
bool addVarBoutReal(const std::string &name, bool repeat)
bool addVarField2D(const std::string &name, bool repeat)
bool addVarField3D(const std::string &name, bool repeat)
bool addVarFieldPerp(const std::string &name, bool repeat)

bool read(int *var, const char *name, int lx = 1, int ly = 0, int lz = 0)
bool read(int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0)
bool read(BoutReal *var, const char *name, int lx = 1, int ly = 0, int lz = 0)
bool read(BoutReal *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0)
bool read_perp(BoutReal *var, const std::string &name, int lx = 1, int lz = 0)
bool write(int *var, const char *name, int lx = 0, int ly = 0, int lz = 0)
bool write(int *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0)
bool write(BoutReal *var, const char *name, int lx = 0, int ly = 0, int lz = 0)
bool write(BoutReal *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0)
bool write_perp(BoutReal *var, const std::string &name, int lx = 0, int lz = 0)
bool write_rec(int *var, const char *name, int lx = 1, int ly = 0, int lz = 0)
bool write_rec(int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0)
bool write_rec(BoutReal *var, const char *name, int lx = 1, int ly = 0, int lz = 0)
bool write_rec(BoutReal *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0)
bool write_rec_perp(BoutReal *var, const std::string &name, int lx = 1, int lz = 0)
bool write_rec(int *var, const char *name, int lx = 0, int ly = 0, int lz = 0)
bool write_rec(int *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0)
bool write_rec (BoutReal *var, const char *name, int lx = 0, int ly = 0, int lz = 0)

bool write_rec (BoutReal *var, const std::string &name, int lx = 0, int ly = 0, int lz = 0)

bool write_rec_perp (BoutReal *var, const std::string &name, int lx = 0, int lz = 0)

void setLowPrecision ()

void setAttribute (const std::string &varname, const std::string &attrname, const std::string &text)

Sets a string attribute

    Inputs

    Parameters

    • 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.

    • attrname: Attribute name

    • text: A string attribute to attach to the variable

void setAttribute (const std::string &varname, const std::string &attrname, int value)

Sets an integer attribute

    Inputs

    Parameters

    • 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.

    • attrname: Attribute name

    • value: An int attribute to attach to the variable

void setAttribute (const std::string &varname, const std::string &attrname, BoutReal value)

Sets a BoutReal attribute

    Inputs

    Parameters

    • 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.

    • attrname: Attribute name

    • value: A BoutReal attribute to attach to the variable

bool getAttribute (const std::string &varname, const std::string &attrname, std::string &text)

Gets a string attribute

    Inputs

    Returns

    Parameters

    • 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.
• attrname: Attribute name
  text A string attribute of the variable

bool getAttribute(const std::string &varname, const std::string &attrname, int &value)
  Gets an integer attribute

Inputs
Returns
Parameters
  • 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.
  • attrname: Attribute name
  value An int attribute of the variable

bool getAttribute(const std::string &varname, const std::string &attrname, BoutReal &value)
  Gets a BoutReal attribute

Inputs
Returns
Parameters
  • 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.
  • attrname: Attribute name
  value A BoutReal attribute of the variable

Private Functions

void checkName(const char *name)
  Check if a name contains invalid characters.

Private Members

cchar *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 NcDim **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

L.2.152 File ncxx4.cxx

L.2.153 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/.

Typedefs

using Ncxx4 = EmptyFormat

L.2.154 File openmpwrap.hxx

Defines

INDIRECT0 (a)
    Openmp utility wrappers.

Copyright 2017
L.2.155 File operatorstencil.hxx

Classes describing the geometry of stencils used for differentiation operators. These can be used to determine how much memory to preallocate when constructing a sparse matrix to represent the operator.

Copyright 2019 C. MacMackin

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 OffsetInd3D = IndexOffset<Ind3D>

Using OffsetInd2D = IndexOffset<Ind2D>

Using OffsetIndPerp = IndexOffset<IndPerp>

### Functions

Template<class T>

bool operator==(const IndexOffset<T> &lhs, const IndexOffset<T> &rhs)

Template<class T>

bool operator!=(const IndexOffset<T> &lhs, const IndexOffset<T> &rhs)

Template<class T>


bool operator< (const[IndexOffset]<T> &lhs, const[IndexOffset]<T> &rhs)

template<class T>
const[IndexOffset]<T> operator+ (IndexOffset<T> lhs, const[IndexOffset]<T> &rhs)

template<class T>
const[IndexOffset]<T> operator- (IndexOffset<T> lhs, const[IndexOffset]<T> &rhs)

template<class T>
const T operator+ (const T &lhs, const[IndexOffset]<T> &rhs)

template<class T>
const T operator+ (const[IndexOffset]<T> &lhs, const T &rhs)

template<class T>
const T operator- (const T &lhs, const[IndexOffset]<T> &rhs)

template<class T>
struct IndexOffset
    #include <operatorstencil.hxx> A representation of offsets for indices, which can be added and subtracted from them.

    Public Functions

    const[IndexOffset] xp (int delta_x = 1) const
    const[IndexOffset] xm (int delta_x = 1) const
    const[IndexOffset] yp (int delta_y = 1) const
    const[IndexOffset] ym (int delta_y = 1) const
    const[IndexOffset] zp (int delta_z = 1) const
    const[IndexOffset] zm (int delta_z = 1) const
    IndexOffset &operator+=(const[IndexOffset] &n)
    IndexOffset &operator-=(const[IndexOffset] &n)

    Public Members

    int dx = 0
    int dy = 0
    int dz = 0

template<class T>
class OperatorStencil
    #include <operatorstencil.hxx> A class which can be used to represent the shape of a stencil used to perform some operation. A stencil is made up of pairs of stencil-parts and stencil-tests. A stencil-part is a vector of OffsetIndices. When added to another index, the result is part of the stencil. A stencil-test indicates whether a particular stencil-part should be applied at a given index.

    When trying to get the stencil part for an index, this class will iterate through the part/test pairs in the order which they were added, returning the first stencil-part with a test that passes. If no stencil-part is found with a passing test, then an error is thrown.
Public Types

```cpp
template<>
using offset = IndexOffset<T>

template<>
using stencil_part = std::vector<offset>

template<>
using stencil_test = std::function<bool (T)>

template<>
using iterator = typename std::vector::iterator
    Iterators for the underlying vector data type.

template<>
using const_iterator = typename std::vector::const_iterator

template<>
using reverse_iterator = typename std::vector::reverse_iterator

template<>
using const_reverse_iterator = typename std::vector::const_reverse_iterator
```

Public Functions

```cpp
void add(stencil_test test, stencil_part stencil)
    Add a stencil test/part pair.

const stencil_part &getStencilPart(int i) const
    Get the ith stencil-part to have been added.

const stencil_part &getStencilPart(const T &i) const
    Get the stencil-part to be used on this index. The method will iterate through the part/test pairs in the
    order which they were added, returning the first stencil-part with a test that passes. If no stencil-part is
    found with a passing test, then an error is thrown.

int getStencilSize(int i) const
    Get the number of elements in the ith stencil-part.

int getStencilSize(const T &i) const
    Get the number of elements in the stencil part to be used at this index.

int getNumParts() const
    Get the number of stencil-parts to have been added.

const std::vector<T> getIndicesWithStencilIncluding(const T &i) const
    Returns a list of indices for which the stencils contain the argument

iterator begin()

const_iterator begin() const

const_iterator cbegin() const

iterator end()

const_iterator end() const
```
Private Functions

int getStencilNumber (const T &i) const
    Returns the position of the first passing stencil test for this index, or -1 if no test passes.

Private Members

std::vector<Stencil> stencils = {}

struct Stencil

Public Members

Public Stencil

template<>
 stencil_test test

template<>
 stencil_part part

L.2.156 File optionparser.hxx

class OptionParser
    #include <optionparser.hxx> Base class for input file types.

    Subclassed by OptionINI

Public Functions

OptionParser()

virtual ~OptionParser()

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.
L.2.157 File options.cxx

Functions

template<>  
std::string as<>::string (const std::string &similar_to) const  
   Specialised as routines.

L.2.158 File options.hxx

Defines

__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<>::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.

Options options;

   // Set values
   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

(continues on next page)
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, 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()  
Constructor. This is called to create the root object.
```

```
Options(Options *parent_instance, std::string full_name)  
Constructor used to create non-root objects
```

Parameters

- `parent`: Parent object
- `sectionName`: Name of the section, including path from the root

```
template<typename T> Options(T value)  
Initialise with a value These enable Options to be constructed using initializer lists
```

```
Options(std::initializer_list<std::pair<std::string, Options>> values)  
Construct with a nested initializer list This allows Options trees to be constructed, using a mix of types.
```

Example: `{ {"key1", 42}, {"key2", field} }`

```
Options(const Options &other)  
Copy constructor.
```

```
~Options()  
```

```
Options &operator[](const std::string &name)  
Get a sub-section or value
```

Example:

```
Options parent; auto child = parent["child"];  
parent is now a section.
```

```
Options &operator[](const char *name)  
```
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

const Options &operator[](const char *name) const

template<typename T>
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.

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;

template<typename T>
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:
Options option; option["test"].assign(42, “some source”);
Note: Specialised versions for types stored in ValueType

template<typename T>
void force(T val, const std::string source = "")
Force to a value Overwrites any existing setting

bool isset() const
Test if a key is set by the user. Values set via default values are ignored.

template<typename T>
operator T() const
Cast operator, which allows this class to be assigned to type T
Example:
Options option; option[“test”] = 2.0; int value = option[“test”];

template<typename T>
T as(const T &similar_to = {}) const
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:
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.

template<typename T>
T withDefault(T def)
Get the value of this option. If not found, set to the default value
std::string withDefault (const char *def)
    Overloaded version for const char* Note: Different from template since return type is different to input

Options &withDefault (const Options &def)
    Overloaded version to copy from another option.

template<typename T>
T withDefault (T def) const
    Get the value of this option. If not found, return the default value but do not set

template<typename T>
T overrideDefault (T def)
    Allow the user to override defaults set later, also used by the BOUT_OVERRIDE_DEFAULT_OPTION.

std::string overrideDefault (const char *def)
    Overloaded version for const char* Note: Different from template since return type is different to input

Options &parent ()
    Get the parent Options object.

template<typename T>
bool operator== (const T &other) const
    Equality operator Converts to the same type and compares This conversion may fail, throwing std::bad_cast

bool operator== (const char *other) const
    Overloaded equality operator for literal strings.

template<typename T>
bool operator< (const T &other) const
    Comparison operator.

bool operator< (const char *other) const
    Overloaded comparison operator for literal strings.

template<typename T>
void set (const std::string &key, T val, const std::string &source = "", bool force = false)

template<typename T>
void forceSet (const std::string &key, T t, const std::string &source = "")

bool isSet (const std::string &key) const
    Test if a key is set by the user. Values set via default values are ignored.

template<typename T, typename U>
void get (const std::string &key, T &val, U def, bool log = false)
    Get options, passing in a reference to a variable which will be set to the requested value.

template<typename T, typename U>
void get (const std::string &key, T &val, U def, bool log = false) const

Options *getSection (const std::string &name)
    Creates new section if doesn’t exist.

const Options *getSection (const std::string &name) const

Options *getParent () const
std::string str() const
  Print string representation of this object and all sections in a tree structure

std::string name() const
  Print just the name of this object without parent sections.

void printUnused() const
  Print the options which haven’t been used.

std::map<std::string, Options::OptionValue> values() const

std::map<std::string, const Options *> subsections() const

const std::map<std::string, Options> &getChildren() const

bool isValue() const

bool isSection(const std::string &name = "") const

bool valueUsed() const
  If the option value has been used anywhere.

Options &doc(const std::string &docstring)
  Set a documentation string as an attribute “doc” Returns a reference to this, to allow chaining

template<> void assign(bool val, const std::string source)

template<> void assign(int val, const std::string source)

template<> void assign(BoutReal val, const std::string source)

template<> void assign(std::string val, const std::string source)

template<> void assign(const char *val, const std::string source)

template<> void assign(Field2D val, const std::string source)

template<> void assign(Field3D val, const std::string source)

template<> 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)

int as(const int &similar_to) const
**BoutReal as (const BoutReal &similar_to) const**

```cpp
template<>
bool as (const bool &similar_to) const
```

```cpp
template<>
Field2D as (const Field2D &similar_to) const
```

```cpp
template<>
Field3D as (const Field3D &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

**Options &root ()**
- Get a reference to the only root instance.

**void cleanup ()**
- Free all memory.

**static Options *getRoot ()**
- Get a pointer to the only root instance (singleton)

**void cleanCache ()**
- Clean the cache of parsed options

### Private Functions

**template<typename T> void _set (T val, std::string source, bool force)**

**template<typename T> bool similar (T a, T b) const**
- Tests if two values are similar.

**template<>**

**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?

bool `value_used` = false
   Record whether this value is used.

**Private Static Attributes**

const std::string `DEFAULT_SOURCE` = {"default")
   The source label given to default values.

*Options* `root_instance` = {nullptr}
   Only instance of the root section.

**Public Types**

```cpp
using Base = bout::utils::variant<bool, int, BoutReal, std::string>
```

**Public Functions**

`AttributeType()`
   Constructor.

`AttributeType(const AttributeType &other)`
   Copy constructor.

`AttributeType(AttributeType &&other)`
   Move constructor.

`~AttributeType()`
   Destructor.

`AttributeType &operator=(const char *str)`
   Assignment from const char*.

```cpp
template<typename T>
```


operator \textbf{\textit{T}}() \textbf{\textit{const}}

Cast operator, which allows this class to be assigned to type T. This will throw \texttt{std::bad\_cast} if it can’t be done

\textbf{\textit{template<typename T>}}

\textbf{\textit{T}} \textbf{\textit{as}}() \textbf{\textit{const}}

Get the value as a specified type. This will throw \texttt{std::bad\_cast} if it can’t be done

\textbf{\textit{struct OptionValue}}

\texttt{\#include <options.hxx>} Class used to store values, together with information about their origin and usage

\textbf{\textit{Public Functions}}

\textbf{\textit{OptionValue}} (\texttt{std::string value, std::string source, bool used})

This constructor needed for \texttt{map::emplace}. Can be removed in C++17 with \texttt{map::insert} and brace initialisation

\textbf{\textit{Public Members}}

\texttt{std::string value}

\texttt{std::string source}

\texttt{bool used} = \texttt{false}

L.2.159 File options\_ini.cxx

L.2.160 File options\_ini.hxx

class \textbf{\textit{OptionINI}} : \textbf{\textit{public OptionParser}}

\texttt{\#include <options\_ini.hxx>} Class for reading INI style configuration files.

\textbf{\textit{Public Functions}}

\textbf{\textit{void read (Options *options, const std::string &filename)}}

Read options from file.

\textbf{\textit{void write (Options *options, const std::string &filename)}}

Write options to file.

\textbf{\textit{Private Functions}}

\textbf{\textit{void parse (const std::string &buffer, std::string &key, std::string &value)}}

\textbf{\textit{string getNextLine (std::ifstream &fin)}}

\textbf{\textit{void writeSection (const Options *options, std::ofstream &fout)}}
Defines

```cpp
__OPTIONS_NETCDF_H__
```

```cpp
namespace bout
    SNB model
```

```cpp
namespace experimental
```

```cpp
class OptionsNetCDF
```

**Public Types**

```cpp
enum FileMode

Values:
```

```cpp
replace
    Overwrite file when writing.
```

```cpp
append
    Append to file when writing.
```

**Public Functions**

```cpp
OptionsNetCDF (const std::string &filename, FileMode mode = FileMode::replace)
```

```cpp
Options read()
    Read options from file.
```

```cpp
void write(const Options &options)
    Write options to file.
```

L.2.163 File optionsreader.cxx

L.2.164 File optionsreader.hxx

```cpp
class OptionsReader
```

```cpp
#include <optionsreader.hxx> Class to handle reading options from file
```

**Example**

```cpp
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 std::string &filename)
Read the given file, parse options into the options tree.

Parameters

- **options**: The options section to insert values and subsections into
- **file**: The name of the file. printf style arguments can be used to create the file name.

template<class S, class ...Args>
void **read**(Options *options, const S &format, const Args&... args)

void **write**(Options *options, const std::string &filename)
Write options to file

Parameters

- **options**: The options tree to be written
- **file**: The name of the file to (over)write

template<class S, class ...Args>
void **write**(Options *options, const S &format, const Args&... args)

void **parseCommandLine**(Options *options, int argc, char **argv)
Parse options from the command line

Example

Parameters

- **options**: The options section to insert values and subsections into
- **argc**: The number of command-line arguments
- **argv**: The command line arguments

int main(int argc, char **argv) {
  Options opt; OptionsReader::getInstance()->read(&opt, argc, argv); . . .
  return 0; }

Public Static Functions

**OptionsReader** *getInstance*
Return a pointer to the instance singleton.

static void **cleanup**
Delete the instance.

Private Static Attributes

**OptionsReader** *instance* = nullptr
The instance of this singleton.
L.2.165 File output.cxx

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.

L.2.166 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 BOUT_USE_OUTPUT_DEBUG is false

template<typename T>
DummyOutput &operator<<(DummyOutput &out, const T *t)

DummyOutput &operator<<(DummyOutput &out, stream_manipulator pf)

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 : `multiooutbuf init`<char, std::char_traits<char>>, public std::basic_ostream<char, std::char_traits<char>>

#include `<output.hxx>` Class for text output to stdout and/or log file.

This class can be used to output either in fmt format:

output.write(“A string {:s} and number {:d}\n”, str, i);

or as a C++ stream buffer:

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

Output()

Output(const std::string &filename)
Specify a log file to open.

template<class S, class ...Args>
Output(const S &format, const Args&... args)

~Output()

void enable()
Enables writing to stdout (default)

void disable()
Disables stdout.

int open(const std::string &filename)
Open an output log file.

template<class S, class ...Args>
int open (const S &format, const Args&... args)

void close ()
    Close the log file.

void write (const std::string &message)
    Write a string using fmt format.

template<class S, class ...Args>
void write (const S &format, const Args&... args)

void print (const std::string &message)
    Same as write, but only to screen.

template<class S, class ...Args>
void print (const S &format, const Args&... args)

void add (std::basic_ostream<char, _Tr> &str)
    Add an output stream. All output will be sent to all streams.

void remove (std::basic_ostream<char, _Tr> &str)
    Remove an output stream.

Public Static Functions

Output *getInstance ()
    Return pointer to instance.

Protected Functions

virtual Output *getBase ()

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.

bool enabled
    Whether output to stdout is enabled.

Friends

friend 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

template<class S, class ...Args>
void write (const S&, const Args&...)

template<class S, class ...Args>
void print (const S&, const Args&...)

void enable ()
   Enables writing to stdout (default)

disable ()
   Disables stdout.

void enable (bool enable)

bool isEnabled ()

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

ConditionalOutput (Output *base, bool enabled = true)
   Parameters
      • base: The Output object which will be written to if enabled
      • enabled: Should this be enabled by default?

ConditionalOutput (ConditionalOutput *base)
   Constructor taking ConditionalOutput. This allows several layers of conditions

   Parameters
      • base: A ConditionalOutput which will be written to if enabled

void write (const std::string &message)
   If enabled, writes a string using fmt formatting by calling base->write This string is then sent to log file and stdout (on processor 0)

template<class S, class ...Args>
void write (const S &format, const Args&... args)

void print (const std::string &message)
   If enabled, print a string to stdout using fmt formatting note: unlike write, this is not also sent to log files

template<class S, class ...Args>
void print (const S &format, const Args&... args)

Output *getBase ()
   Get the lowest-level Output object which is the base of this ConditionalOutput.
void enable (bool enable_)
   Set whether this ConditionalOutput is enabled. If set to false (disabled), then all print and write calls do nothing.

void enable ()
   Turn on outputs through calls to print and write.

void disable ()
   Turn off outputs through calls to print and write. This includes log files and stdout.

bool isEnabled ()
   Check if output is enabled.

Private Members

Output *base
   The lower-level Output to send output to.

bool enabled
   Does this instance output anything?

L.2.167 File parallel_boundary_op.cxx

L.2.168 File parallel_boundary_op.hxx

class BoundaryOpPar : public BoundaryOpBase
   Subclassed by BoundaryOpPar_dirichlet, BoundaryOpPar_dirichlet_interp, BoundaryOpPar_dirichlet_O3, BoundaryOpPar_neumann

Public Functions

BoundaryOpPar ()

BoundaryOpPar (BoundaryRegionPar *region, std::shared_ptr<FieldGenerator> value)

BoundaryOpPar (BoundaryRegionPar *region, Field3D *value)

BoundaryOpPar (BoundaryRegionPar *region, BoutReal value)

~BoundaryOpPar ()

virtual BoundaryOpPar *clone (BoundaryRegionPar *region, const std::list<std::string> &args)

virtual BoundaryOpPar *clone (BoundaryRegionPar *region, Field3D *f)

virtual BoundaryOpPar *clone (BoundaryRegionPar *region, const std::map<std::string, std::string> &keywords, const std::map<std::string, std::string> &keywords)

void apply (Field2D &f)
   Apply a boundary condition on field f.

void apply (Field2D &f, BoutReal t)
Public Members

*BoundaryRegionPar* \texttt{bndry} = \texttt{nullptr}

Protected Types

\begin{verbatim}
enum ValueType
    Where to take boundary values from - the generator, field or BoutReal.
    Values:
    GEN
    FIELD
    REAL
\end{verbatim}

Protected Functions

\begin{verbatim}
BoutReal getValue (int x, int y, int z, BoutReal t)
BoutReal getValue (const BoundaryRegionPar \&bndry, BoutReal t)
\end{verbatim}

Protected Attributes

\begin{verbatim}
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\}
\end{verbatim}

class BoundaryOpPar_dirichlet : public BoundaryOpPar

Public Functions

BoundaryOpPar_dirichlet ()

BoundaryOpPar_dirichlet (BoundaryRegionPar \*region)

BoundaryOpPar_dirichlet (BoundaryRegionPar \*region, std::shared_ptr<FieldGenerator> value)

BoundaryOpPar_dirichlet (BoundaryRegionPar \*region, Field3D \*value)

BoundaryOpPar_dirichlet (BoundaryRegionPar \*region, BoutReal value)

BoundaryOpPar \*clone (BoundaryRegionPar \*region, const std::list\&string \&args)

BoundaryOpPar \*clone (BoundaryRegionPar \*region, Field3D \*f)

void apply (Field3D \&f)

void apply (Field3D \&f, BoutReal t)

class BoundaryOpPar_dirichlet_03 : public BoundaryOpPar
Public Functions

BoundaryOpPar_dirichlet_03()

BoundaryOpPar_dirichlet_03(BoundaryRegionPar *region)

BoundaryOpPar_dirichlet_03(BoundaryRegionPar *region, std::shared_ptr<FieldGenerator> value)

BoundaryOpPar_dirichlet_03(BoundaryRegionPar *region, Field3D *value)

BoundaryOpPar_dirichlet_03(BoundaryRegionPar *region, BoutReal value)

BoundaryOpPar *clone(BoundaryRegionPar *region, const std::list<std::string> &args)

BoundaryOpPar *clone(BoundaryRegionPar *region, Field3D *f)

void apply(Field3D &f)

void apply(Field3D &f, BoutReal t)

class BoundaryOpPar_dirichlet_interp: public BoundaryOpPar

Public Functions

BoundaryOpPar_dirichlet_interp()

BoundaryOpPar_dirichlet_interp(BoundaryRegionPar *region)

BoundaryOpPar_dirichlet_interp(BoundaryRegionPar *region, std::shared_ptr<FieldGenerator> value)

BoundaryOpPar_dirichlet_interp(BoundaryRegionPar *region, Field3D *value)

BoundaryOpPar_dirichlet_interp(BoundaryRegionPar *region, BoutReal value)

BoundaryOpPar *clone(BoundaryRegionPar *region, const std::list<std::string> &args)

BoundaryOpPar *clone(BoundaryRegionPar *region, Field3D *f)

void apply(Field3D &f)

void apply(Field3D &f, BoutReal t)

class BoundaryOpPar_neumann: public BoundaryOpPar

Public Functions

BoundaryOpPar_neumann()

BoundaryOpPar_neumann(BoundaryRegionPar *region)

BoundaryOpPar_neumann(BoundaryRegionPar *region, std::shared_ptr<FieldGenerator> value)

BoundaryOpPar_neumann(BoundaryRegionPar *region, Field3D *value)

BoundaryOpPar_neumann(BoundaryRegionPar *region, BoutReal value)

BoundaryOpPar *clone(BoundaryRegionPar *region, const std::list<std::string> &args)
**BoundaryOpPar** *

```cpp
BoundaryOpPar *clone (BoundaryRegionPar *region, Field3D *f)
```

```cpp
void apply (Field3D &f)
```

```cpp
void apply (Field3D &f, BoutReal t)
```

**L.2.169 File parallel_boundary_region.cxx**

**L.2.170 File parallel_boundary_region.hxx**

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**

```cpp
BoundaryRegionPar (const std::string &name, int dir, Mesh *passmesh)
```

```cpp
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.

void **first** ()

Move the region iterator to the start.

void **next** ()

Get the next element in the loop over every element from inside out (in X or Y first)

bool **isDone** ()

Returns true if outside domain. Can use this with nested nextX, nextY.

**Public Members**

```cpp
int x
```

Index of the point in the boundary.

```cpp
int y
```

```cpp
int z
```

```cpp
BoutReal s_x
```

```cpp
BoutReal s_y
```

```cpp
BoutReal s_z
```

```cpp
BoutReal length
```

```cpp
BoutReal angle
```

```cpp
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

L.2.171 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, ShiftedMetricInterp
Public Functions

ParallelTransform (Mesh &mesh_in, Options *opt = nullptr)

virtual ~ParallelTransform ()

virtual void calcParallelSlices (Field3D &f) = 0
Given a 3D field, calculate and set the Y up down fields.

void calcYupYdown (Field3D &f)

virtual void integrateParallelSlices (Field3D &f)
Calculate Yup and Ydown fields by integrating over mapped points. This should be used for parallel divergence operators.

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.

const Field3D toFieldAligned (const Field3D &f, REGION region)

virtual const FieldPerp toFieldAligned (const FieldPerp &f, const std::string &region = "RGN_ALL") = 0

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.

const Field3D fromFieldAligned (const Field3D &f, REGION region)

virtual const FieldPerp fromFieldAligned (const FieldPerp &f, const std::string &region = "RGN_ALL") = 0

const FieldPerp fromFieldAligned (const FieldPerp &f, REGION region)

virtual bool canToFromFieldAligned () = 0

virtual std::vector<PositionsAndWeights> getWeightsForYUpApproximation (int i, int j, int k)

virtual std::vector<PositionsAndWeights> getWeightsForYDownApproximation (int i, int j, int k)

virtual std::vector<PositionsAndWeights> getWeightsForYApproximation (int i, int j, int k, int yoffset)

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.

Options &options
Options for this ParallelTransform.

struct PositionsAndWeights

Public Members

int i
int j
int k
BoutReal weight

class ParallelTransformIdentity : public ParallelTransform
#include <paralleltransform.hxx> This class implements the simplest form of ParallelTransform where the domain is a logically rectangular domain, and yup() and ydown() refer to the same field.

Public Functions

ParallelTransformIdentity (Mesh &mesh_in, Options *opt = nullptr)
void calcParallelSlices (Field3D &f)
Merges the yup and ydown() fields of f, so that f.yup() = f.ydown() = f

const Field3D toFieldAligned (const Field3D &f, const std::string &region = "RGN_ALL")
The field is already aligned in Y, so this does nothing

const FieldPerp toFieldAligned (const FieldPerp &f, const std::string &region = "RGN_ALL")

const Field3D fromFieldAligned (const Field3D &f, const std::string &region = "RGN_ALL")
The field is already aligned in Y, so this does nothing

const FieldPerp fromFieldAligned (const FieldPerp &f, const std::string &region = "RGN_ALL")

virtual std::vector<PositionsAndWeights> getWeightsForYApproximation (int i, int j, int k, int yoffset)

bool canToFromFieldAligned ()

bool requiresTwistShift (bool twist_shift_enabled, YDirectionType ytype)
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**

void checkInputGrid()
   This method should be called in the constructor to check that if the grid has a ‘parallel_transform’ variable, it has the correct value

**Public Functions**

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()

ShiftedMetric (Mesh &mesh, CELL_LOC location, Field2D zShift, BoutReal zlength_in, Options *opt = nullptr)

void calcParallelSlices (Field3D &f)
   Calculates the yup() and ydown() fields of f by taking FFTs in Z and applying a phase shift.

const Field3D toFieldAligned (const Field3D &f, const std::string &region = "RGN_ALL")
   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.

const FieldPerp toFieldAligned (const FieldPerp &f, const std::string &region = "RGN_ALL")

const Field3D fromFieldAligned (const Field3D &f, const std::string &region = "RGN_ALL")
   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.

const FieldPerp fromFieldAligned (const FieldPerp &f, const std::string &region = "RGN_ALL")

bool canToFromFieldAligned()

void outputVars (Datafile &file)
   Save zShift to the output.

bool requiresTwistShift (bool twist_shift_enabled, YDirectionType ytype)
   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**

void checkInputGrid()
   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

\begin{verbatim}
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

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
- \( f \): The field to shift
- \( zangle \): Toroidal angle (z)

const Field3D shiftZ(const Field3D &f, const Field2D &zangle, REGION region) const

const Field3D shiftZ(const Field3D &f, const Tensor<dcomplex> &phs, const YDirection-Type 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
- \( f \): The field to shift
- \( phs \): The phase to shift by
- \( y_direction_out \): The value to set yDirectionType of the result to

const FieldPerp shiftZ(const FieldPerp &f, const Tensor<dcomplex> &phs, const YDirection-Type 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
- \( in \): A 1D array of length \( len \)
- \( len \): Length of the in and out arrays
- \( zangle \): The angle (z coordinate) to shift by
- \( out \): 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
- \( in \): A 1D array of length mesh.LocalNz
- \( phs \): Phase shift, assumed to have length (mesh.LocalNz/2 + 1) i.e. the number of modes
- \( out \): A 1D array of length mesh.LocalNz, already allocated
\end{verbatim}
void cachePhases()
Calculate and store the phases for to/from field aligned and for the parallel slices using zShift

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

Return The shifted parallel slices

Parameters

• f: The field to shift
• phases: The phase and offset information for each parallel slice

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

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[n + i] stores offset -(i+1) where i goes from 0 to (n-1), with n the number of y guard cells

struct ParallelSlicePhase
coordinates
Helper POD for parallel slice phase shifts

Public Members

Tensor<dcomplex> phase_shift

int y_offset

L.2.172 File pdd.cxx

L.2.173 File pdd.hxx

class LaplacePDD : public Laplacian
**Public Functions**

`LaplacePDD (Options *opt = nullptr, const CELL_LOC loc = CELL_CENTRE, Mesh *mesh_in = nullptr)`

`~LaplacePDD ()`

void `setCoefA(const Field2D &val)`

Set coefficients for inversion. Re-builds matrices if necessary.

void `setCoefC(const Field2D &val)`

void `setCoefD(const Field2D &val)`

void `setCoefEx(const Field2D &val)`

void `setCoefEz(const Field2D &val)`

`FieldPerp solve(const FieldPerp &b)`

`Field3D solve(const Field3D &b)`

**Private Functions**

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`: RHS values (Ax = b)
- `data`: Internal data used for multiple calls in parallel mode

void `next(PDD_data &data)`

Middle part of the PDD algorithm.

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<dcomplex>v
Matrix<dcomplex>w
Array<BoutReal> snd
Array<BoutReal> rcv
   comm_handle recv_handle
Array<dcomplex>y2i

L.2.174 File petsc3damg.cxx
L.2.175 File petsc3damg.hxx

class LaplacePetsc3dAmg : public Laplacian

Public Functions

LaplacePetsc3dAmg (Options *opt = nullptr, const CELL_LOC loc = CELL_CENTRE, Mesh
   *mesh_in = nullptr)

~LaplacePetsc3dAmg ()

void setCoefA (const Field2D &val)
   Set coefficients for inversion. Re-builds matrices if necessary.

void setCoefC (const Field2D &val)

void setCoefC1 (const Field2D &val)

void setCoefC2 (const Field2D &val)

void setCoefD (const Field2D &val)

void setCoefEx (const Field2D &val)

void setCoefEz (const Field2D &val)

void setCoefA (const Field3D &val)

void setCoefC (const Field3D &val)
void setCoefC1(const Field3D &val)
void setCoefC2(const Field3D &val)
void setCoefD(const Field3D &val)
void setCoefEx(const Field3D &val)
void setCoefEz(const Field3D &val)

PetscMatrix<Field3D> &getMatrix3D()

IndexerPtr<Field3D> getIndexer()

Field2D solve(const Field2D &b)

virtual Field3D solve(const Field3D &b)

Field3D solve(const Field3D &b, const Field3D &x0)
Perform the laplacian inversion y-slice by y-slice

Return x All the y-slices of x_slice in the equation A*x_slice = b_slice

Parameters
  • b: All the y-slices of b_slice, which is the right hand side of the equation A*x_slice = b_slice
  • x0: All the y-slices of the variable eventually used to set BC

virtual FieldPerp solve(const FieldPerp &b)

Private Functions

void updateMatrix3D()

Private Members

Field3D A
Field3D C1
Field3D C2
Field3D D
Field3D Ex
Field3D Ez
bool issetD = false
bool issetC = false
bool issetE = false
bool updateRequired = true
int lastflag
int lower_boundary_flags
int upper_boundary_flags
int meshx
int meshz
int size
int localN

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

RangeIterator lowerY

RangeIterator upperY

IndexerPtr<Indexer> indexer

PetscMatrix<Field3D> operator3D

KSP ksp

bool kspInitialised

PetscLib lib

bool use_precon

bool rightprec

Private Static Functions

OperatorStencil<Ind3D> getStencil (Mesh *localmesh, RangeIterator lowerYBound, RangeIterator upperYBound)

Private Static Attributes

constexpr int implemented_flags = INVERT_START_NEW

constexpr int implemented_boundary_flags = INVERT_AC_GRAD + INVERT_SET + INVERT_RHS
L.2.176 File petsc.cxx

Defines

__FUNCT__
__FUNCT__
__FUNCT__
__FUNCT__
__FUNCT__
__FUNCT__
__FUNCT__
__FUNCT__
__FUNCT__
__FUNCT__
__FUNCT__
__FUNCT__

Functions

PetscErrorCode solver_f (TS ts, BoutReal t, Vec globalin, Vec globalout, void *f_data)
PetscErrorCode solver_rhsjacobian (TS ts, BoutReal t, Vec globalin, Mat *J, Mat *Jpre, MatStructure *str, void *f_data)
PetscErrorCode solver_ijacobianfd (TS, PetscReal, Vec, Vec, PetscReal, Mat *, Mat *, MatStructure *, void *)
PetscErrorCode solver_if (TS ts, BoutReal t, Vec globalin, Vec globalindot, Vec globalout, void *f_data)
PetscErrorCode PhysicsPCApply (PC pc, Vec x, Vec y)

KSP preconditioner PChell routines for physics preconditioners.

PetscErrorCode PhysicsJacobianApply (Mat J, Vec x, Vec y)

PetscErrorCode solver_rhsjacobian (MAYBE_UNUSED) TS ts
    , MAYBE_UNUSED BoutReal t, MAYBE_UNUSED Vec globalin, Mat *J, Mat *Jpre,
    MAYBE_UNUSED MatStructure *str, MAYBE_UNUSED void *f_data
PetscErrorCode solver_ijacobian (TS ts, BoutReal t, Vec globalin, MAYBE_UNUSED) Vec globalindot
    , MAYBE_UNUSED PetscReal a, Mat *J, Mat *Jpre, MatStructure *str, void *f_data
PetscErrorCode solver_ijacobianfd (TS ts, BoutReal t, Vec globalin, Vec globalindot, PetscReal a, Mat *J, Mat *Jpre, MatStructure *str, void *f_data)

PetscErrorCode PetscMonitor (TS ts, PetscInt step, PetscReal t, Vec X, void *ctx)

Monitor function called on every internal timestep.

PetscErrorCode PetscSNESMonitor (SNES snes, PetscInt its, PetscReal norm, void *ctx)

Monitor function for SNES.
L.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 ()
void setPrecon (PhysicsPrecon f)
void setJacobian (Jacobian j)
  Specify a Jacobian (optional)
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

PetscErrorCode run ()
    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

PetscLogEvent solver_event
PetscLogEvent loop_event
PetscLogEvent init_event

Private Members

PhysicsPrecon prefunc
    Preconditioner.

Jacobian jacfunc
    Jacobian - vector function.

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 Jmf
    RHS Jacobian.

MatFDColoring matfdcoloring

int nout
    The number of outputs.
BoutReal tstep
  Time between outputs.

bool diagnose
  If true, print some information about current stage.

BoutReal next_output
  When the monitor should be called next.

PetscBool interpolate
  Whether to interpolate or not.

cchar output_name[PETSC_MAX_PATH_LEN]

PetscBool output_flag

PetscInt prev_linear_its

BoutReal bout_snes_time

std::vector<snes_info> snes_list

bool adaptive
  Use adaptive timestepping.

Friends

PetscErrorCode PetscMonitor(TS ts, PetscInt step, BoutReal t, Vec X, void *ctx)
  Monitor function called on every internal timestep.

PetscErrorCode PetscSNESMonitor(SNES snes, PetscInt its, BoutReal norm, void *ctx)
  Monitor function for SNES.

PetscErrorCode solver_ijacobian(TS, BoutReal, Vec, Vec, PetscReal, Mat *, Mat *, MatStructure *, void *)
  Compute IJacobian = dF/dU + a dF/dUdot - a dummy matrix used for pc=none.

L.2.178 File petsc_interface.hxx

Classes to wrap PETSc matrices and vectors, providing a convenient interface to them. In particular, they will internally convert between BOUT++ indices and PETSc ones, making it far easier to set up a linear system.

Copyright 2019 C. MacMackin

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 IndexerPtr = std::shared_ptr<GlobalIndexer<T>>
using InterpolationWeights = std::vector<ParallelTransform::PositionsAndWeights>

Functions

template<class T>
void swap(PetscVector<T> &first, PetscVector<T> &second)
    Move reference to one Vec from first to second and vice versa.

template<class T>
void swap(PetscMatrix<T> &first, PetscMatrix<T> &second)
    Move reference to one Mat from first to second and vice versa.

template<class T>
PetscVector<T> operator*(const PetscMatrix<T> &mat, const PetscVector<T> &vec)
    Performs matrix-multiplication on the supplied vector

class GlobalIndexer
#include <petsc_interface.hxx> A singleton which accepts index objects produced by iterating over fields and returns a global index. This index can be used when constructing PETSc arrays. Guard regions used for communication between processes will have the indices of the part of the interior region they are mirroring. Boundaries required by the stencil will have unique indices. If no stencil is provided then only the interior region will be assigned global indices. By default, the indexer is fully initialised so that guard cells are communicated (ensuring they hold the appropriate global indices). However, by passing autoInitialise = false this behaviour can be turned off and the user can then manually call the initialise() method later. This can be useful for mocking/faking the class when testing.

Public Types

template<>
using ind_type = typename T::ind_type

Public Functions

GlobalIndexer()

GlobalIndexer(Mesh *localmesh, OperatorStencil<ind_type> stencil = OperatorStencil<ind_type>(), bool autoInitialise = true)

virtual ~GlobalIndexer()

void initialiseTest()
    Call this immediately after construction when running unit tests.

void initialise()
    Finish setting up the indexer, communicating indices across processes and, if possible, calculating the sparsity pattern of any matrices.

Mesh *getMesh() const

PetscInt getGlobal(const ind_type &ind) const
    Convert the local index object to a global index which can be used in PETSc vectors and matrices.
bool isLocal (const ind_type &ind) const
    Check whether the local index corresponds to an element which is stored locally.
PetscInt getGlobalStart () const
const Region<ind_type> &getRegionAll () const
const Region<ind_type> &getRegionNobdry () const
const Region<ind_type> &getRegionBndry () const
const Region<ind_type> &getRegionLowerY () const
const Region<ind_type> &getRegionUpperY () const
const Region<ind_type> &getRegionInnerX () const
const Region<ind_type> &getRegionOuterX () const
bool sparsityPatternAvailable () const
const std::vector<int> &getNumDiagonal () const
const std::vector<int> &getNumOffDiagonal () const
int size () const

Protected Functions

T &getIndices ()

Private Functions

virtual void registerFieldForTest (T &f)
    This gets called by initialiseTest and is used to register fields with fake parallel meshes.

void calculateSparsity () const

Private Members

Mesh *fieldmesh
T indices
    Fields containing the indices for each element (as reals)
PetscInt globalStart
    The first and last global index on this processor (inclusive in both cases)
PetscInt globalEnd
OperatorStencil<ind_type> stencils
    Stencil for which this indexer has been configured.
Region<ind_type> regionAll
    Regions containing the elements for which there are global indices.
Region<ind_type> regionLowerY
Region<ind_type> regionUpperY
Region<ind_type> regionInnerX
Region<ind_type> regionOuterX
Region<ind_type> regionBndry
bool sparsityCalculated = false
std::vector<PetscInt> numDiagonal
std::vector<PetscInt> numOffDiagonal

Private Static Functions

static void insertIndex(const ind_type i, std::set<ind_type> &allInds, std::set<ind_type> &newInds)

template<class T>
class GlobalIndexer
#include <petsc_interface.hxx> A singleton which accepts index objects produced by iterating over fields and returns a global index. This index can be used when constructing PETSc arrays. Guard regions used for communication between processes will have the indices of the part of the interior region they are mirroring. Boundaries required by the stencil will have unique indices. If no stencil is provided then only the interior region will be assigned global indices. By default, the indexer is fully initialised so that guard cells are communicated (ensuring they hold the appropriate global indices). However, by passing autoInitialise = false this behaviour can be turned off and the user can then manually call the initialise() method later. This can be useful for mocking/faking the class when testing.

Public Types

template<>
using ind_type = typename T::ind_type

Public Functions

GlobalIndexer()

GlobalIndexer(Mesh *localmesh, OperatorStencil<ind_type> stencil = OperatorStencil<ind_type>(), bool autoInitialise = true)

virtual ~GlobalIndexer()

void initialiseTest()
Call this immediately after construction when running unit tests.

void initialise()
Finish setting up the indexer, communicating indices across processes and, if possible, calculating the sparsity pattern of any matrices.

Mesh *getMesh() const

PetscInt getGlobal(const ind_type &ind) const
Convert the local index object to a global index which can be used in PETSc vectors and matrices.
bool isLocal (const ind_type &ind) const
  Check whether the local index corresponds to an element which is stored locally.

PetscInt getGlobalStart () const

const Region<ind_type> &getRegionAll () const
const Region<ind_type> &getRegionNobndry () const
const Region<ind_type> &getRegionBndry () const
const Region<ind_type> &getRegionLowerY () const
const Region<ind_type> &getRegionUpperY () const
const Region<ind_type> &getRegionInnerX () const
const Region<ind_type> &getRegionOuterX () const
bool sparsityPatternAvailable () const

const std::vector<int> &getNumDiagonal () const
const std::vector<int> &getNumOffDiagonal () const

int size () const

Protected Functions

T &getIndices ()

Private Functions

virtual void registerFieldForTest (T &f)  
  This gets called by initialiseTest and is used to register fields with fake parallel meshes.

void calculateSparsity () const

Private Members

Mesh *fieldmesh
T indices  
  Fields containing the indices for each element (as reals)
PetscInt globalStart  
  The first and last global index on this processor (inclusive in both cases)
PetscInt globalEnd
OperatorStencil<ind_type> stencils  
  Stencil for which this indexer has been configured.
Region<ind_type> regionAll  
  Regions containing the elements for which there are global indices.
Region<ind_type> regionLowerY
Region<ind_type> regionUpperY
Region<ind_type> regionInnerX
Region<ind_type> regionOuterX
Region<ind_type> regionBndry
bool sparsityCalculated = false
std::vector<PetscInt> numDiagonal
std::vector<PetscInt> numOffDiagonal

Private Static Functions

static void insertIndex(const ind_type i, std::set<ind_type> &allInds, std::set<ind_type> &newInds)

template<class T>
class PetscVector
#include <petsc_interface.hxx> A class which wraps PETSc vector objects, allowing them to be indexed using the BOUT++ scheme. Note that boundaries are only included in the vector to a depth of 1.

Public Types

template<>
using ind_type = typename T::ind_type

Public Functions

PetscVector() Default constructor does nothing.

PetscVector(const PetscVector<T> &v) Copy constructor.

PetscVector(PetscVector<T> &&v) Move constructor.

PetscVector(const T &f, IndexerPtr<T> indConverter) Construct from a field, copying over the field values.

PetscVector(const PetscVector<T> &v, Vec *vec) Construct a vector like v, but using data from a raw PETSc Vec. That Vec (not a copy) will then be owned by the new object.

PetscVector<T> &operator= (const PetscVector<T> &rhs) Copy assignment.

PetscVector<T> &operator= (PetscVector<T> &&rhs) Move assignment.

PetscVector<T> &operator= (const T &f)

Element operator() (const ind_type &index)

BoutReal operator() (const ind_type &index) const
void assemble()
void destroy()

T toField() const
Returns a field constructed from the contents of this vector.

Vec *get() const
Provides a reference to the raw PETSc Vec object.

const Vec *get() const

Private Members

PetscLib lib
std::unique_ptr<Vec, VectorDeleter> vector = nullptr

IndexerPtr<T> indexConverter

CELL_LOC location

bool initialised = false

Friends

void swap(PetscVector<T> &first, PetscVector<T> &second)
Move reference to one Vec from first to second and vice versa.

class Element
#include <petsc_interface.hxx> A class which is used to assign to a particular element of a PETSc vector.
It is meant to be transient and will be destroyed immediately after use. In general you should not try to
assign an instance to a variable.

The Element object will store a copy of the value which has been assigned to it. This is because mixing
calls to the PETSc function VecGetValues() and VecSetValue() without intermediate vector assembly
will cause errors. Thus, if the user wishes to get the value of the vector, it must be stored here.

Public Functions

template<>
 Element()

template<>
 Element(const Element &other)

template<>
 Element(Vec *vector, int index)

template<>
 Element &operator=(Element &other)

template<>
 Element &operator=(BoutReal val)

template<>
Element &operator+=(BoutReal val)

template<>
operator BoutReal () const

Private Members

template<>
Vec *petscVector = nullptr

template<>
PetscInt petscIndex

template<>
PetscScalar value

struct VectorDeleter

Public Functions

template<>
void operator() (Vec *v) const

template<class T>

class PetscVector

#include <petsc_interface.hxx> A class which wraps PETSc vector objects, allowing them to be indexed using the BOUT++ scheme. Note that boundaries are only included in the vector to a depth of 1.

Public Types

template<>
using ind_type = typename T::ind_type

Public Functions

PetscVector ()
Default constructor does nothing.

PetscVector (const PetscVector<T> &v)
Copy constructor.

PetscVector (PetscVector<T> &&v)
Move constructor.

PetscVector (const T &f, IndexerPtr<T> indConverter)
Construct from a field, copying over the field values.

PetscVector (const PetscVector<T> &v, Vec *vec)
Construct a vector like v, but using data from a raw PETSc Vec. That Vec (not a copy) will then be owned by the new object.

PetscVector<T> &operator= (const PetscVector<T> &rhs)
Copy assignment.
PetscVector<T> &\texttt{operator=} (PetscVector<T> &&rhs)
Move assignment.

PetscVector<T> &\texttt{operator=} (\texttt{const T} &f)

\texttt{Element operator() (const ind_type &index)}

\texttt{BoutReal operator() (const ind_type &index) const}

void \texttt{assemble()}

void \texttt{destroy()}

\texttt{T toField() const}
Returns a field constructed from the contents of this vector.

Vec *\texttt{get()}
Provides a reference to the raw PETSc Vec object.

\texttt{const Vec *get() const}

\textbf{Private Members}

\textit{PetscLib lib}
std::unique_ptr<Vec, VectorDeleter> \texttt{vector} = nullptr

\textit{IndexerPtr<T> indexConverter}

\textit{CELL_LOC location}

bool \texttt{initialised} = false

\textbf{Friends}

void \texttt{swap (PetscVector<T> &first, PetscVector<T> &second)}
Move reference to one Vec from \texttt{first} to \texttt{second} and vice versa.

class \texttt{Element}

\texttt{#include <petsc_interface.hxx> A class which is used to assign to a particular element of a PETSc vector. It is meant to be transient and will be destroyed immediately after use. In general you should not try to assign an instance to a variable.}

The Element object will store a copy of the value which has been assigned to it. This is because mixing calls to the PETSc function VecGetValues() and VecSetValues() without intermediate vector assembly will cause errors. Thus, if the user wishes to get the value of the vector, it must be stored here.

\textbf{Public Functions}

template<>
\texttt{Element ()}

template<>
\texttt{Element (const Element &other)}

template<>
\texttt{Element (Vec *vector, int index)}
template<> Element &operator= (Element &other)  

template<> Element &operator= (BoutReal val)  

template<> Element &operator+= (BoutReal val)  

operator BoutReal () const

**Private Members**

template<>  
Vec *petscVector = nullptr  

template<>  
PetscInt petscIndex  

template<>  
PetscScalar value

**struct VectorDeleter**

**Public Functions**

template<>  
void operator() (Vec *v) const

**template<class T>**

**class PetscMatrix**

#include <petsc_interface.hxx> A class which wraps PETSc vector objects, allowing them to be indexed using the BOUT++ scheme. It provides the option of setting a y-offset that interpolates onto field lines.

**Public Types**

template<>
using ind_type = typename T::ind_type

**Public Functions**

**PetscMatrix ()**  
Default constructor does nothing.

**PetscMatrix (const PetscMatrix<T> &m)**  
Copy constructor.

**PetscMatrix (PetscMatrix<T> &&m)**  
Move constructor.

**PetscMatrix (IndexerPtr<T> indConverter, bool preallocate = true)**
PetscMatrix<T> &operator= (PetscMatrix<T> rhs)
    Copy assignment.

PetscMatrix<T> &operator= (PetscMatrix<T> &&rhs)
    Move assignment.

Element operator() (const ind_type &index1, const ind_type &index2)

BoutReal operator() (const ind_type &index1, const ind_type &index2) const

void assemble ()

void partialAssemble ()

void destroy ()

PetscMatrix<T> yup (int index = 0)

PetscMatrix<T> ydown (int index = 0)

PetscMatrix<T> ynext (int dir)

Mat *get ()
    Provides a reference to the raw PETSc Mat object.

    const Mat *get () const

Private Members

PetscLib lib

std::shared_ptr<Mat> matrix = nullptr

IndexerPtr<T> indexConverter

ParallelTransform *pt

int yoffset = 0

bool initialised = false

Friends

void swap (PetscMatrix<T> &first, PetscMatrix<T> &second)
    Move reference to one Mat from first to second and vice versa.

class Element
    
#include <petsc_interface.hxx> A class which is used to assign to a particular element of a PETSc matrix, potentially with a y-offset. It is meant to be transient and will be destroyed immediately after use. In general you should not try to assign an instance to a variable.

    The Element object will store a copy of the value which has been assigned to it. This is because mixing calls to the PETSc function MatGetValues() and MatSetValue() without intermediate matrix assembly will cause errors. Thus, if the user wishes to get the value of the matrix, it must be stored here.
Public Functions

template<>
Element ()

template<>
Element (const Element &other)

template<>
Element (Mat *matrix, PetscInt row, PetscInt col, std::vector<PetscInt> p = {},
        std::vector<BoutReal> w = {})

template<>
Element & operator= (Element &other)

template<>
Element & operator= (BoutReal val)

template<>
Element & operator+= (BoutReal val)

operator BoutReal () const

Private Functions

template<>
void setValue (BoutReal val, InsertMode mode)

Private Members

Mat *petscMatrix

PetscInt petscRow

PetscInt petscCol

PetscScalar value

std::vector<PetscInt> positions

std::vector<BoutReal> weights

struct MatrixDeleter

Public Functions

template<>
void operator () (Mat *m) const
class PetscMatrix
#include <petsc_interface.hxx> A class which wraps PETSc vector objects, allowing them to be indexed using the BOUT++ scheme. It provides the option of setting a y-offset that interpolates onto field lines.

Public Types

template<>
using ind_type = typename T::ind_type

Public Functions

PetscMatrix() Default constructor does nothing.

PetscMatrix(const PetscMatrix<T> &m) Copy constructor.

PetscMatrix(PetscMatrix<T> &&m) Move constructur.

PetscMatrix(IndexerPtr<T> indConverter, bool preallocate = true)

PetscMatrix<T> &operator=(PetscMatrix<T> rhs) Copy assignment.

PetscMatrix<T> &operator=(PetscMatrix<T> &&rhs) Move assignment.

Element operator() (const ind_type &index1, const ind_type &index2)

BoutReal operator() (const ind_type &index1, const ind_type &index2) const

void assemble()

void partialAssemble()

void destroy()

PetscMatrix<T> yup (int index = 0)

PetscMatrix<T> ydown (int index = 0)

PetscMatrix<T> ynext (int dir)

Mat *get ()
    Provides a reference to the raw PETSc Mat object.

    const Mat *get () const

Private Members

PetscLib lib
std::shared_ptr<Mat> matrix = nullptr

IndexerPtr<T> indexConverter
ParallelTransform *pt

int yoffset = 0
bool initialised = false

Friends

void swap (PetscMatrix<T> &first, PetscMatrix<T> &second)
    Move reference to one Mat from first to second and vice versa.

class Element
    #include <petsc_interface.hxx> A class which is used to assign to a particular element of a PETSc matrix, potentially with a y-offset. It is meant to be transient and will be destroyed immediately after use. In general you should not try to assign an instance to a variable.

    The Element object will store a copy of the value which has been assigned to it. This is because mixing calls to the PETSc function MatGetValues() and MatSetValues() without intermediate matrix assembly will cause errors. Thus, if the user wishes to get the value of the matrix, it must be stored here.

Public Functions

    template<>
    Element ()

    template<>
    Element (const Element &other)

    template<>
    Element (Mat *matrix, PetscInt row, PetscInt col, std::vector<PetscInt> p = {}, std::vector<BoutReal> w = {})

    template<>
    Element &(operator= (Element &other)

    template<>
    Element &(operator= (BoutReal val)

    template<>
    Element &(operator+= (BoutReal val)

    template<>
    operator BoutReal () const

Private Functions

    template<>
    void setValues (BoutReal val, InsertMode mode)

Private Members

    template<>
    Mat *petscMatrix
PetscInt `petscRow`

template<>
PetscInt `petscCol`

template<>
PetscScalar `value`

template<>
std::vector<PetscInt> `positions`

template<>
std::vector<BoutReal> `weights`

```cpp
struct MatrixDeleter

Public Functions

template<>
void operator() (Mat *m) const
```

**L.2.179 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`
- `__FUNCT__`

**Functions**

```cpp
static PetscErrorCode laplacePCapply (PC pc, Vec x, Vec y)
```

**L.2.180 File petsc_laplace.hxx**

```cpp
class LaplacePetsc: public Laplacian
```
Public Functions

**LaplacePetsc** *(Options *opt = nullptr, const CELL_LOC loc = CELL_CENTRE, Mesh *mesh_in = nullptr)*

~LaplacePetsc()  
void setCoefA(const Field2D &val)  

Set coefficients for inversion. Re-builds matrices if necessary.

void setCoefC(const Field2D &val)  
void setCoefC1(const Field2D &val)  
void setCoefC2(const Field2D &val)  
void setCoefD(const Field2D &val)  
void setCoefEx(const Field2D &val)  
void setCoefEz(const Field2D &val)  
void setCoefA(const Field3D &val)  
void setCoefC(const Field3D &val)  
void setCoefC1(const Field3D &val)  
void setCoefC2(const Field3D &val)  
void setCoefD(const Field3D &val)  
void setCoefEx(const Field3D &val)  
void setCoefEz(const Field3D &val)

**FieldPerp solve** *(const FieldPerp &b)*

**FieldPerp solve** *(const FieldPerp &b, const FieldPerp &x0)*  
Solves Ax=b for x given a b and an initial guess for x (x0)

This function will:

1. Set the matrix element of the matrix A, used to solve Ax=b (this includes setting the values for the boundary condition)
2. Solve the matrix Ax = b

**Return**  
sol The solution x of the problem Ax=b.

**Parameters**

- b: 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: The initial guess for the solver. May also contain the boundary condition if flag 32 - INVERT_SET is set

**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 &\(\text{MatA}\))

Sets the elements of the matrix \(A\), which is used to solve the problem \(Ax=b\).

**Parameters**

- \(i\): The row of the PETSc matrix
- \(x\): Local \(x\) index of the mesh
- \(z\): Local \(z\) index of the mesh
- \(xshift\): The shift in rows from the index \(x\)
- \(zshift\): The shift in columns from the index \(z\)
- \(ele\): Value of the element
- \(\text{MatA}\): The matrix \(A\) used in the inversion
- \(\text{MatA}\): The matrix \(A\) used in the inversion

void **Coeffs** (int \(x\), int \(y\), int \(z\), BoutReal &\(A1\), BoutReal &\(A2\), BoutReal &\(A3\), BoutReal &\(A4\), BoutReal &\(A5\))

Sets the matrix components of \(A\) in \(Ax=b\), solving \(D\cdot\text{Laplace}_\text{perp}(x) + \left(\frac{1}{C1}\right)\text{Grad}_\text{perp}(C2)\cdot\text{Grad}_\text{perp}(x) + Ax = B\)

**Note** "\(A\)" in the equation above is not added here. For calculations of the coefficients, please refer to the user manual.

**Parameters**

- \(x\): The current \(x\) index
- \(y\): The current \(y\) index
- \(z\): The current \(z\) index
- \(\text{coef1}\): Placeholder for convenient variable used to set matrix (see manual for details)
- \(\text{coef2}\): Convenient variable used to set matrix (see manual for details)
- \(\text{coef3}\): Placeholder for convenient variable used to set matrix (see manual for details)
- \(\text{coef4}\): Placeholder for convenient variable used to set matrix (see manual for details)
- \(\text{coef5}\): Placeholder for convenient variable used to set matrix (see manual for details)
- \(\text{coef1}\): Convenient variable used to set matrix (see manual for details)
- \(\text{coef2}\): Convenient variable used to set matrix (see manual for details)
- \(\text{coef3}\): Convenient variable used to set matrix (see manual for details)
- \(\text{coef4}\): Convenient variable used to set matrix (see manual for details)
- \(\text{coef5}\): 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
std::unique_ptr<Laplacian> pcsolve
int implemented_flags
int implemented_boundary_flags

L.2.181 File petsclib.cxx

L.2.182 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

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.

Public Static Functions

static 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()

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

int count = 0
   How many instances?
char help = "BOUT++: Uses finite difference methods to solve plasma fluid problems in curvilinear coordinates"
   Help string.
int *pargc = nullptr
char ***pargv = nullptr
PetscLogEvent USER_EVENT = 0

L.2.183 File physicsmodel.cxx

Defines

BOUT_NO_USING_NAMESPACE_BOUTGLOBALS

L.2.184 File physicsmodel.hxx

Base class for Physics Models.
Changelog:
2013-08 Ben Dudson benjamin.dudson@york.ac.uk
   • Initial version
Copyright 2013 B.D.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

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

L.2. File list
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

class PhysicsModel
    #include <physicsmodel.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()
        void initialise (Solver *)
            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
            Parameters
                • time: The simulation time

            The time derivatives will be put in the ddt() variables
            Returns a flag: 0 indicates success, non-zero an error flag
        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

int runTimestepMonitor (BoutReal simtime, BoutReal dt)

Public Members

Mesh *mesh = {nullptr}
Datfile &dump

Protected Functions

virtual int init (bool restarting) = 0
This function is called once by the solver at the start of a simulation.
A valid PhysicsModel must implement this function
Variables should be read from the inputs, and the variables to be evolved should be specified.

int postInit (bool restarting)
Post-initialise. This reads the restart file

Parameters

• restarting: If true, will load state from restart file

virtual int rhs (BoutReal 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.

virtual int convective (BoutReal t)

virtual int diffusive (BoutReal t)
virtual int **diffusive**( BoutReal t, bool linear)**

virtual int **outputMonitor**( BoutReal simtime, int iter, int NOUT)**
    Implemented by user code to monitor solution at output times

virtual int **timestepMonitor**( BoutReal simtime, BoutReal dt)**
    Timestep monitor. If enabled by setting solver:monitor_timestep=true then this function is called every internal timestep.

void **setSplitOperator**( bool split = true)**
    Specify that this model is split into a convective and diffusive part.

void **setPrecon**( preconfunc pset)**
    Specify a preconditioner function.

void **setJacobian**( jacobianfunc jset)**
    Specify a Jacobian-vector multiply function.

void **bout_solve**( Field2D &var, const char *name)**
    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 var is stored in the solver.

Parameters

• var: The variable to evolve
• name: The name to use for variable initialisation and output

To evolve the state, the solver will set var, and the user-supplied rhs() function should calculate ddt(var).

void **bout_solve**( Field3D &var, const char *name)**

void **bout_solve**( Vector2D &var, const char *name)**

void **bout_solve**( Vector3D &var, const char *name)**

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: The variable the solver should modify
• F_var: The control variable, which the user will set
• name: 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 \texttt{splitop} = \{false\}
Split operator model?

\texttt{preconfunc userprecon} = \{nullptr\}
Pointer to user-supplied preconditioner function.

\texttt{jacobianfunc userjacobian} = \{nullptr\}
Pointer to user-supplied Jacobian-vector multiply function.

bool \texttt{initialised} = \{false\}
True if model already initialised.

\textbf{class} \texttt{PhysicsModelMonitor} : public \texttt{Monitor}

\#include <physicsmodel.hxx> \texttt{Monitor} class for \texttt{PhysicsModel}

**Public Functions**

\texttt{PhysicsModelMonitor()}  
\texttt{PhysicsModelMonitor(PhysicsModel *model)}

\textbf{int call} (Solver *\texttt{solver}, BoutReal \texttt{time}, int \texttt{iter}, int \texttt{nout})
Callback function for the solver, called after timestep_ has passed

\textbf{Return} non-zero if simulation should be stopped

\textbf{Parameters}

- solver: The solver calling this monitor
- \texttt{time}: The current simulation time
- \texttt{iter}: The current simulation iteration
- \texttt{nout}: The total number of iterations for this simulation

**Private Members**

\texttt{PhysicsModel *model}

L.2.185 File pnetcdf.cxx

**Defines**

\texttt{CHKERR(ret)}

**Functions**

int \texttt{pnc_get_var_all} (int \texttt{ncfile}, int \texttt{var}, double *\texttt{data})
int \texttt{pnc_get_var_all} (int \texttt{ncfile}, int \texttt{var}, float *\texttt{data})
int \texttt{pnc_get_vara_all} (int \texttt{ncfile}, int \texttt{var}, MPI_Offset *\texttt{start}, MPI_Offset *\texttt{count}, double *\texttt{data})
int \texttt{pnc_get_vara_all} (int \texttt{ncfile}, int \texttt{var}, MPI_Offset *\texttt{start}, MPI_Offset *\texttt{count}, float *\texttt{data})
int pnc_put_var_all (int ncfile, int var, double *data)
int pnc_put_var_all (int ncfile, int var, float *data)
int pnc_put_vara_all (int ncfile, 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)

L.2.186 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/.

class PncFormat : public DataFormat

Public Functions

PncFormat (Mesh *mesh_in = nullptr)
PncFormat (const char *name, Mesh *mesh_in = nullptr)
PncFormat (const std::string &name, Mesh *mesh_in = nullptr)
~PncFormat ()
bool openr (const char *name)
bool openr (const std::string &name, int mype)
bool openw (const char *name, bool append = false)
bool openw (const std::string &name, int mype, bool append = false)
bool is_valid ()
void close ()
void flush()

const char *filename()

class BOUT++:

const vector<int> getSize(const char *var)
const vector<int> getSize(const std::string &var)

bool setGlobalOrigin(int x = 0, int y = 0, int z = 0)

bool setRecord(int t)

bool read(int *var, const char *name, int lx = 1, int ly = 0, int lz = 0)
bool read(int *var, const std::string &name, int lx = 1, int ly = 0, int lz = 0)
bool read(BoutReal *var, const char *name, int lx = 1, int ly = 0, int lz = 0)
bool read(BoutReal *var, const std::string &name, int lx = 1, int ly = 0, intlz = 0)

bool write(int *var, const char *name, int lx = 0, int ly = 0, intlz = 0)
bool write(int *var, const std::string &name, int lx = 0, int ly = 0, intlz = 0)
bool write(BoutReal *var, const char *name, int lx = 0, int ly = 0, intlz = 0)
bool write(BoutReal *var, const std::string &name, intlx = 0, intly = 0, intlz = 0)

bool read_rec(int *var, const char *name, int lx = 1, int ly = 0, intlz = 0)
bool read_rec(int *var, const std::string &name, int lx = 1, int ly = 0, intlz = 0)
bool read_rec(BoutReal *var, const char *name, int lx = 1, int ly = 0, intlz = 0)
bool read_rec(BoutReal *var, const std::string &name, int lx = 1, int ly = 0, intlz = 0)

bool write_rec(int *var, const char *name, int lx = 0, int ly = 0, intlz = 0)
bool write_rec(int *var, const std::string &name, int lx = 0, int ly = 0, intlz = 0)
bool write_rec(BoutReal *var, const char *name, int lx = 0, int ly = 0, intlz = 0)
bool write_rec(BoutReal *var, const std::string &name, int lx = 0, int ly = 0, intlz = 0)

void setLowPrecision()

Private Members:

char *fname
    Current file name.

int ncf
    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

L.2.187 File power.cxx
L.2.188 File power.hxx

class PowerSolver : public Solver

    Public Functions

    PowerSolver()
    PowerSolver(Options *)
    ~PowerSolver()
    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
    int run()
        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.
    void outputVars (Datafile &outputfile, bool save_repeat = true)
        Add evolving variables to output (dump) file or restart file

        Parameters
        • outputfile: The file to add variable to
        • save_repeat: If true, add variables with time dimension
Private Functions

\texttt{BoutReal norm} (Array\texttt{<BoutReal>} &\texttt{state})

\textbf{void} divide (Array\texttt{<BoutReal>} &\texttt{in}, BoutReal \texttt{value})

Private Members

\texttt{BoutReal curtime}
\texttt{BoutReal eigenvalue}
int nlocal
int nglobal
Array\texttt{<BoutReal>} \texttt{f0}
int nsteps

L.2.189 File pvode.cxx

Functions

\textbf{void} solver\_f (integer \texttt{N}, BoutReal \texttt{t}, N\texttt{-Vector} \texttt{u}, N\texttt{-Vector} \texttt{udot}, void \texttt{*f\_data})

\textbf{void} solver\_gloc (integer \texttt{N}, BoutReal \texttt{t}, BoutReal \texttt{*u}, BoutReal *\texttt{udot}, void \texttt{*f\_data})

\textbf{void} solver\_cfn (integer \texttt{N}, BoutReal \texttt{t}, N\texttt{-Vector} \texttt{u}, void \texttt{*f\_data})

Variables

\texttt{const BoutReal ZERO = 0.0}

long int \texttt{iopt[OPT\_SIZE]}

BoutReal \texttt{ropt[OPT\_SIZE]}

L.2.190 File pvode.hxx

class PvodeSolver : public Solver

Public Functions

\texttt{PvodeSolver (Options *opts)}

\texttt{~PvodeSolver ()}

\texttt{BoutReal getCurrentTimestep ()}

\hspace{1cm} Return the current internal timestep.

\texttt{int init (int nout, BoutReal tstep)}

\hspace{1cm} Initialise the solver NOTE: nout and tstep should be passed to run, not init. Needed because of how the PETSc TS code works
int run()

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::PVBBDDa pdata
bool pvode_initialised = false

L.2.191 File range.cxx

L.2.192 File range.hxx

class RangeIterator

Public Functions

RangeIterator ()

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
int operator*()  

RangeIterator &operator++()  

RangeIterator operator++(int)  

bool operator==(const RangeIterator &) const  

bool operator!=(const RangeIterator &) const  

bool intersects(const RangeIterator &other, bool all = true) const  

bool intersects(int ind, bool all = true) const  

RangeIterator &operator=(const RangeIterator &)  

RangeIterator &operator+=(const RangeIterator &)  

RangeIterator &operator-=(const RangeIterator &)

int min() const  

int max() const  

RangeIterator *nextRange() const

Public Members  

int ind

Public Static Functions  

static RangeIterator end()  

Private Members  

int is = {1}  

int ie = {0}  

RangeIterator *n = {nullptr}  

RangeIterator *cur = {nullptr}  

int curend  

bool delete_next = false

L.2.193 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.

Parameters

- index: The name of the index variable to use in the loop
- region: An already existing Region

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];
}
```

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:
  IND_3D = 0
  IND_2D = 1
  IND_PERP = 2

Functions

template<IND_TYPE N>
bool operator==(const SpecificInd<N> &lhs, const SpecificInd<N> &rhs)
  Relational operators.

template<IND_TYPE N>
bool operator!=(const SpecificInd<N> &lhs, const SpecificInd<N> &rhs)

template<IND_TYPE N>
bool operator<(const SpecificInd<N> &lhs, const SpecificInd<N> &rhs)

template<IND_TYPE N>
bool operator>(const SpecificInd<N> &lhs, const SpecificInd<N> &rhs)

template<IND_TYPE N>
bool operator<=(const SpecificInd<N> &lhs, const SpecificInd<N> &rhs)

template<IND_TYPE N>
bool operator>=(const SpecificInd<N> &lhs, const SpecificInd<N> &rhs)

template<IND_TYPE N>
SpecificInd<N> operator+ (SpecificInd<N> lhs, const SpecificInd<N> &rhs)
  Arithmetic operators with integers.

template<IND_TYPE N>
SpecificInd<N> operator+ (SpecificInd<N> lhs, int n)

template<IND_TYPE N>
SpecificInd<N> operator+ (int n, SpecificInd<N> rhs)

const std::string toString (const Ind3D &i)
  Get string representation of Ind3D.

  Get string representation of IndPerp.
Get string representation of Ind2D.

```cpp
std::ostream &operator<<(std::ostream &out, const RegionStats &stats)
    Provide an easy way to report a Region’s statistics.
```

```cpp
template<typename T>
Region<T> sort (Region<T> &region)
    Return a new region with sorted indices.
```

```cpp
template<typename T>
Region<T> unique (Region<T> &region)
    Return a new region with unique indices.
```

```cpp
template<typename T>
Region<T> mask (const Region<T> &region, const Region<T> &mask)
    Return a masked version of a region.
```

```cpp
template<typename T>
Region<T> getUnion (const Region<T> &region, const Region<T> &otherRegion)
    Return the union of two regions.
```

```cpp
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.

```cpp
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
```

```cpp
unsigned int size (const Region<T> &region)
    Return the number of indices in a Region.
```

```cpp
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
```

```cpp
Examples
Field3D field, result;
auto index = std::begin(region);
result = field[index->yp()] - field[index->ym()];
```
Public Functions

SpecificInd()
SpecificInd (int i, int ny, int nz)
SpecificInd (int i)
SpecificInd &operator++ ()
    Pre-increment operator.
SpecificInd operator++ (int)
    Post-increment operator.
SpecificInd &operator-- ()
    Pre-decrement operator.
SpecificInd operator-- (int)
    Post-decrement operator.
SpecificInd &operator+=(SpecificInd n)
    In-place addition.
SpecificInd &operator+=(int n)
SpecificInd &operator-= (SpecificInd n)
    In-place subtraction.
SpecificInd &operator-= (int n)
SpecificInd operator% (int n)
    Modulus operator.

int x () const
    Convenience functions for converting to (x, y, z)
int y () const
int z () const

template<int dd, DIRECTION dir>
const SpecificInd plus () const
    Templatized routine to return index.?p(offset), where ? is one of {x,y,z} and is determined by the dir template argument. The offset corresponds to the dd template argument.

template<int dd, DIRECTION dir>
const SpecificInd minus () const
    Templatized routine to return index.?m(offset), where ? is one of {x,y,z} and is determined by the dir template argument. The offset corresponds to the dd template argument.

const SpecificInd xp (int dx = 1) const
const SpecificInd xm (int dx = 1) const
    The index one point -1 in x.
const SpecificInd yp (int dy = 1) const
    The index one point +1 in y.
const SpecificInd ym (int dy = 1) const
    The index one point -1 in y.

const SpecificInd zp (int dz = 1) const
    The index one point +1 in z. Wraps around zend to zstart An alternative, non-branching calculation is :
    ind + dz - nz * ((ind + dz) / nz - ind / nz) but this appears no faster (and perhaps slower).

const SpecificInd zm (int dz = 1) const
    The index one point -1 in z. Wraps around zstart to zend An alternative, non-branching calculation is :
    ind - dz + nz * ((nz + ind) / nz - (nz + ind - dz) / nz) but this appears no faster (and perhaps slower).

const SpecificInd xpp () const
const SpecificInd xmm () const
const SpecificInd ypp () const
const SpecificInd ymm () const
const SpecificInd zpp () const
const SpecificInd zmm () const
const SpecificInd offset (int dx, int dy, int dz) const
    Generic offset of index in multiple directions simultaneously.

Public Members

int ind = -1

Private Members

int ny = -1
int nz = -1

struct RegionStats
    #include <region.hxx> Structure to hold various derived “statistics” from a particular region.

Public Members

int numBlocks = 0
    How many blocks.
int minBlockSize = 0
    Size of smallest block.
int numMinBlocks = 0
    Number of blocks with min size.
int maxBlockSize = 0
    Size of largest block.
int numMaxBlocks = 0
    Number of blocks with max size.
int numSmallBlocks = 0
    Number of “small” blocks, for definition see Region::getStats.
**BoutReal** \texttt{maxImbalance} = 0  
Ratio of largest block to smallest.

template<typename T = Ind3D>

\texttt{class 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 \texttt{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.  

\textit{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 \texttt{begin()} and \texttt{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 \texttt{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;
}

\textbf{Public Types}

\begin{verbatim}
template<>  
using data_type = T
\end{verbatim}
template<>  
using RegionIndices = std::vector<T>  
    Indices to iterate over.

template<>  
using ContiguousBlock = std::pair<T, T>  
    Start and end of contiguous region. This describes a range [block.first,block.second)

template<>  
using ContiguousBlocks = std::vector<ContiguousBlock>  
    Collection of contiguous regions.

template<>  
using value_type = T

template<>  
using reference = value_type&

template<>  
using const_reference = const value_type&

template<>  
using size_type = typename RegionIndices::size_type

template<>  
using iterator = typename RegionIndices::iterator

template<>  
using const_iterator = typename RegionIndices::const_iterator

Public Functions

Region()  
Region(int xstart, int xend, int ystart, int yend, int zstart, int zend, int ny, int nz, int maxregionblocksize = 64)  
Region(RegionIndices &indices, int maxregionblocksize = 64)  
Region(ContiguousBlocks &blocks)  
~Region()  
    Destructor.

RegionIndices::iterator begin()  
    Expose 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.
RegionIndices::const_iterator begin() const
RegionIndices::const_iterator cbegin() const
RegionIndices::iterator end()  
RegionIndices::const_iterator end() const
RegionIndices::const_iterator cend() const
    const ContiguousBlocks &getBlocks() const
const RegionIndices &getIndices () const

void setIndices (RegionIndices &indicesIn, int maxregionblocksize = 64)  
     Set the indices and ensure blocks updated.

void setBlocks (ContiguousBlocks &blocksIn)  
     Set the blocks and ensure indices updated.

Region<T> asSorted ()  
     Return a new Region that has the same indices as this one but ensures the indices are sorted.

Region<T> &sort ()  
     Sort this Region in place.

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.

Region<T> &unique ()  
     Make this Region unique in-place.

Region<T> mask (const Region<T> &maskRegion)  
     Return a new region equivalent to *this but with indices contained in mask Region removed

Region<T> getUnion (const Region<T> &otherRegion)  
     Returns a new region including only indices contained in both this region and the other.

Region<T> &operator+=(const Region<T> &rhs)  
     Accumulate operator.

Region<T> &offset (int offset)  
     Offset all indices by fixed value.

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.

unsigned int size () const  
     Number of indices (possibly repeated)

RegionStats getStats () const  
     Returns a RegionStats struct describing the region.

Private Functions

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

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.

RegionIndices getRegionIndices ()  
     Constructs the vector of indices from the stored blocks information.
Private Members

RegionIndices indices
ContiguousBlocks blocks
int ny = -1
int nz = -1

L.2.194 File rk3-ssp.cxx

L.2.195 File rk3-ssp.hxx

class RK3SSP : public Solver

Public Functions

RK3SSP (Options *opt = nullptr)
~RK3SSP ()
voidsetMaxTimestep (BoutReal dt)
Set a maximum internal timestep (only for explicit schemes)
BoutReal getCurrentTimestep ()
Return the current internal timestep.
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
int run ()
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
Array<BoutReal> L

L.2.196 File rk4.cxx
L.2.197 File rk4.hxx

class RK4Solver: public Solver

Public Functions

RK4Solver (Options *options)

void resetInternalFields ()
   Should wipe out internal field vector and reset from current field object data.

void setMaxTimestep (BoutReal dt)
   Set a maximum internal timestep (only for explicit schemes)

BoutReal getCurrentTimestep ()
   Return the current internal timestep.

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

int run ()
   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 atol
BoutReal rtol
BoutReal max_timestep
int mxstep
Array<BoutReal> f0
Array<
BoutReal>
\texttt{f1}
Array<
BoutReal>
\texttt{f2}
\texttt{BoutReal} \texttt{out\_timestep}
\text{int} \texttt{nsteps}
\texttt{BoutReal} \texttt{timestep}
\text{bool} \texttt{adaptive}
\text{int} \texttt{nlocal}
\text{int} \texttt{neq}
Array<
BoutReal>
\texttt{k1}
Array<
BoutReal>
\texttt{k2}
Array<
BoutReal>
\texttt{k3}
Array<
BoutReal>
\texttt{k4}
Array<
BoutReal>
\texttt{k5}

L.2.198 File rk4simple.cxx

L.2.199 File rk4simple.hxx

class \texttt{RK4SIMPLEScheme} : public \texttt{RKScheme}

\textbf{Public Functions}

\texttt{RK4SIMPLEScheme (Options *options)}

\texttt{BoutReal setOutputStates (const Array<
BoutReal> &start, BoutReal dt, Array<
BoutReal> &resultFollow)}

L.2.200 File rkf34.cxx

L.2.201 File rkf34.hxx

class \texttt{RKF34Scheme} : public \texttt{RKScheme}

\textbf{Public Functions}

\texttt{RKF34Scheme (Options *options)}

L.2.202 File rkf45.cxx

L.2.203 File rkf45.hxx

class \texttt{RKF45Scheme} : public \texttt{RKScheme}
Public Functions

RKF45Scheme (Options *options)

L.2.204 File rkgeneric.cxx

L.2.205 File rkgeneric.hxx

class RKGenericSolver : public Solver

Public Functions

RKGenericSolver (Options *options)

~RKGenericSolver ()

void resetInternalFields ()
    Should wipe out internal field vector and reset from current field object data.

void setMaxTimestep (BoutReal dt)
    Set a maximum internal timestep (only for explicit schemes)

BoutReal getCurrentTimestep ()
    Return the current internal timestep.

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

int run ()
    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

BoutReal take_step (BoutReal timeIn, BoutReal dt, const Array<BoutReal> &start, Array<BoutReal> &resultFollow)

Private Members

Array<BoutReal> f0
Array<BoutReal> f2
Array<BoutReal> tmpState
BoutReal atol
BoutReal rtol
BoutReal max_timestep
int mxstep

L.2. File list 645
BOOL adaptive

boutReal out_timestep
int nsteps
boutReal timestep
int mlocal
int neq

std::unique_ptr<RKScheme> scheme = {nullptr}

L.2.206 File rkscheme.cxx

L.2.207 File rkscheme.hxx

Typedefs

using RegisterRKScheme = RegisterInFactory<RKScheme, DerivedType, RKSchemeFactory>

Simpler name for Factory registration helper class

Usage:

```
#include <bout/rkschemefactory.hxx>
namespace {
  RegisterRKScheme<MyRKScheme> registrkschememine("myrkscheme");
}
```

Variables

constexpr auto RKSCHEME_RKF45 = "rkf45"
constexpr auto RKSCHEME_CASHKARP = "cashkarp"
constexpr auto RKSCHEME_RK4 = "rk4"
constexpr auto RKSCHEME_RKF34 = "rkf34"

class RKSchemeFactory : public Factory<RKScheme, RKSchemeFactory>

Public Static Attributes

constexpr auto type_name = "RKScheme"
constexpr auto section_name = "solver"
constexpr auto option_name = "scheme"
constexpr auto default_type = RKSCHEME_RKF45

class RKScheme
  Subclassed by CASHKARPScheme, RK4SIMPLEScheme, RKF34Scheme, RKF45Scheme
**Public Functions**

**RKScheme** *(Options *opts = nullptr)*  

virtual ~RKScheme()

void init (int nlocalIn, int neqIn, bool adaptiveIn, BoutReal atolIn, BoutReal rtolIn, Options *options = nullptr)

BoutReal setCurTime (BoutReal timeIn, BoutReal dt, int curStage)

void setCurState (const Array<BoutReal> &start, Array<BoutReal> &out, int curStage, BoutReal dt)

BoutReal setOutputStates (const Array<BoutReal> &start, BoutReal dt, Array<BoutReal> &resultFollow)

BoutReal updateTimestep (BoutReal dt, BoutReal err)

virtual std::string getType()

int getStageCount()

int getNumOrders()

**Public Members**

Matrix<BoutReal> steps

**Protected Functions**

BoutReal getErr (Array<BoutReal> &solA, Array<BoutReal> &solB)

void constructOutput (const Array<BoutReal> &start, BoutReal dt, int index, Array<BoutReal> &sol)

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<BoutReal> stageCoeffs

Matrix<BoutReal> resultCoeffs

Array<BoutReal> timeCoeffs

Array<BoutReal> resultAlt

int nlocal
int neq
    BoutReal atol
    BoutReal rtol
    bool adaptive
    BoutReal dtfac

Private Functions

void verifyCoeffs ()
void printButcherTableau ()
void zeroSteps ()

L.2.208 File rvec.hxx

Defines

__RVEC_H__

Typedefs

using rvec = std::vector<BoutReal>

L.2.209 File scorepwrapper.hxx

Defines

SCOREPLVL
SCOREP_BASE_CALL (...)  
    Instrument a 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
SCOREP0 (...)  
    This is always defined.
SCOREP1 (...)  
SCOREP2 (...)  
SCOREP3 (...)  
BOUN_SCOREP_REGION (...)  
    Instrument a region with scorep.
L.2.210 File serial_band.cxx

L.2.211 File serial_band.hxx

class LaplaceSerialBand : public Laplacian

Public Functions

LaplaceSerialBand (Options *opt = nullptr, const CELL_LOC loc = CELL_CENTRE, Mesh *mesh_in = nullptr)
~LaplaceSerialBand ()
void setCoefA (const Field2D &val)
void setCoefC (const Field2D &val)
void setCoefD (const Field2D &val)
void setCoefEx (const Field2D &val)
void setCoefEz (const Field2D &val)
FieldPerp solve (const FieldPerp &)
FieldPerp solve (const FieldPerp &, const FieldPerp &x0)

Private Members

Field2D Acoef
Field2D Ccoef
Field2D Dcoef
Matrix<dcomplex> bk
Matrix<dcomplex> xk
Matrix<dcomplex> A
Array<dcomplex> bk1d
Array<dcomplex> xk1d

L.2.212 File serial_tri.cxx

L.2.213 File serial_tri.hxx

class LaplaceSerialTri : public Laplacian
Public Functions

LaplaceSerialTri (Options *opt = nullptr, const CELL_LOC loc = CELL_CENTRE, Mesh *mesh_in = nullptr)
~LaplaceSerialTri ()

void setCoefA (const Field2D &val)
  Set coefficients for inversion. Re-builds matrices if necessary.

void setCoefC (const Field2D &val)
void setCoefD (const Field2D &val)
void setCoefEx (const Field2D &val)
void setCoefEz (const Field2D &val)

FieldPerp solve (const FieldPerp &b)
FieldPerp solve (const FieldPerp &b, const FieldPerp &x0)
  Solve Ax=b for x given b
  
  This function will
  1. Take the fourier transform of the y-slice given in the input
  2. For each fourier mode a) Set up the tridiagonal matrix b) Call the solver which inverts the matrix
     Ax_mode = b_mode
  3. Collect all the modes in a 2D array
  4. Back transform the y-slice
  
  Input:
  
  Return  The inverted variable.

Parameters

• b: 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: Variable used to set BC (if the right flags are set, see the user manual)

Private Members

Field2D A
Field2D C
Field2D D

L.2.214 File shiftedmetric.cxx
L.2.215 File shiftedmetricinterp.cxx
L.2.216 File shiftedmetricinterp.hxx

class ShiftedMetricInterp : public ParallelTransform
   #include <shiftedmetricinterp.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 \texttt{ZInterpolation} objects

**Public Functions**

\texttt{ShiftedMetricInterp()}  
\texttt{ShiftedMetricInterp(Mesh \&mesh, CELL\_LOC location\_in, Field2D \&\texttt{zShift\_in}, Options \*opt = nullptr)}  

\texttt{void calcParallelSlices(Field3D \&f)}  
Calculate the \texttt{yup()} and \texttt{ydown()} fields of \texttt{f} by interpolating \texttt{f} through a toroidal shift angle

\texttt{const Field3D toFieldAligned(const Field3D \&f, const std::string \&\texttt{region = "RGN\_ALL"})}  
Uses interpolation of \texttt{f} through a toroidal shift angle 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.

\texttt{const FieldPerp toFieldAligned(const FieldPerp \&f, const std::string \&\texttt{region = "RGN\_ALL"})}  
\texttt{const Field3D fromFieldAligned(const Field3D \&f, const std::string \&\texttt{region = "RGN\_ALL"})}  
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.

\texttt{const FieldPerp fromFieldAligned(const FieldPerp \&f, const std::string \&\texttt{region = "RGN\_ALL"})}  
\texttt{bool canToFromFieldAligned()}  

\texttt{std::vector<ParallelTransform::PositionsAndWeights> getWeightsForYUpApproximation(int i, int j, int k)}  

\texttt{std::vector<ParallelTransform::PositionsAndWeights> getWeightsForYDownApproximation(int i, int j, int k)}  

\texttt{bool requiresTwistShift(bool twist\_shift\_enabled, YDirectionType \texttt{ytype})}  
If \texttt{twist\_shift\_enabled} is true, does a Field3D with Y direction \texttt{ytype} require a twist-shift at branch cuts on closed field lines?
Protected Functions

void checkInputGrid()
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

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.

std::vector<std::unique_ptr<ZInterpolation>> parallel_slice_interpolators
Cache of interpolators for the parallel slices. Slices are stored in the following order: {+1, ..., +n, -1, ..., -n} parallel_slice_interpolator[i] stores interpolator for slice i+1 parallel_slice_interpolator[n + i] stores offset -(i+1) where i goes from 0 to (n-1), with n the number of y guard cells.

std::unique_ptr<ZInterpolation> interp_to_aligned ZInterpolation objects for shifting to and from field-aligned coordinates.

std::unique_ptr<ZInterpolation> interp_from_aligned

const std::size_t ydown_index

Private Static Attributes

constexpr std::size_t yup_index = 0

L.2.217 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/.
L.2.218 File shoot_laplace.hxx

class LaplaceShoot : public Laplacian

Public Functions

LaplaceShoot (Options *opt = nullptr, const CELL_LOC loc = CELL Centre, Mesh *mesh_in = nullptr)
~LaplaceShoot ()
void setCoefA (const Field2D &val)
    Set coefficients for inversion. Re-builds matrices if necessary.
void setCoefC (const Field2D &val)
void setCoefD (const Field2D &val)
void setCoefEx (const Field2D &val)
void setCoefHz (const Field2D &val)
FieldPerp solve (const FieldPerp &b)
FieldPerp solve (const FieldPerp &b, const FieldPerp &x0)

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

L.2.219 File slepc.cxx

Functions

std::string formatEig (BoutReal reEig, BoutReal imEig)
PetscErrorCode advanceStepWrapper (Mat matOperator, Vec inData, Vec outData)
PetscErrorCode compareEigsWrapper (PetscScalar ar, PeteSCALar ai, PeteSCALar br, PeteSCALar bi,
    PeteSCALar *res, void *ctx)
PetscErrorCode monitorWrapper (EPS eps, PeteSCALar its, PeteSCALar nconv, PeteSCALar *eigr, PeteSCALar *eigi,
    PeteSCALar *errest, PeteSCALar next, void *ctx)
PetscErrorCode stApplyWrapper (ST st, Vec vecIn, Vec vecOut)

PetscErrorCode stBackTransformWrapper (ST st, PetscInt nEig, PetscScalar *eigr, PetscScalar *eigi)

L.2.220 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 eigr[], PetscScalar eigi[], PetscReal errest[], PetscInt nest)

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

int run ()
        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.

void setModel (PhysicsModel *model)
        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

void setRHS (rhsfunc f)
        Set the RHS function.

void add (Field2D &v, const std::string &name)
        Add a variable to be solved. This must be done in the initialisation stage, before the simulation starts.

void add (Field3D &v, const std::string &name)

void add (Vector2D &v, const std::string &name)

void add (Vector3D &v, const std::string &name)

void setJacobian (Jacobian j)
        Specify a Jacobian (optional)

void setSplitOperator (rhsfunc fC, rhsfunc fD)
        Split operator solves.
bool constraints()
    Returns true if constraints available.

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.

void constraint (Field3D &v, Field3D &C_v, std::string name)
void constraint (Vector2D &v, Vector2D &C_v, std::string name)
void constraint (Vector3D &v, Vector3D &C_v, std::string name)

int n2Dvars () const
    Number of 2D variables. Vectors count as 3.

int n3Dvars () const
    Number of 3D variables. Vectors count as 3.

void setMaxTimestep (BoutReal dt)
    Set a maximum internal timestep (only for explicit schemes)

BoutReal getCurrentTimestep()
    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 &reEigOut, PetscScalar &imEigIn, PetscScalar &imEigOut, bool force = false)

Private Members

MPI_Comm comm
EPS eps
ST st
PetscBool stIsShell
std::unique_ptr<Solver> advanceSolver = {nullptr}
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

L.2.221 File slepclib.cxx

L.2.222 File slepclib.hxx

class SlepcLib

Public Functions

SlepcLib()
~SlepcLib()

Public Static Functions

static void setArgs (int &c, char **&v)
void cleanup()
Private Static Attributes

int count = 0
char help = "BOUT++: Uses finite difference methods to solve plasma fluid problems in curvilinear coordinates"
int *pargc = nullptr
char ***pargv = nullptr
PetscLogEvent USER_EVENT = 0

L.2.223 File smoothing.cxx

Functions

const Field3D smooth_x(const Field3D &f)
  Smooth in X using simple 1-2-1 filter.

const Field3D smooth_y(const Field3D &f)
  Smooth in Y using 1-2-1 filter.

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

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

const 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

const Field3D averageY(const Field3D &f)
  Average in Y
  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

const Field3D smoothXY(const Field3D &f)
Smooth using a stencil in X and Y.

void nl_filter(rvec &f, BoutReal w)

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”

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”

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”

L.2.224 File smoothing.hxx

Functions

const Field3D smooth_x(const Field3D &f)
Smooth in X using simple 1-2-1 filter.

const Field3D smooth_y(const Field3D &f)
Smooth in Y using 1-2-1 filter.

const Field3D smoothXY(const Field3D &f)
Smooth using a stencil in X and Y.
const 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

const Field3D averageY (const Field3D &f)
  Average in Y

  Issues
  Important: Only works if there are no branch cuts
  Creates static arrays
  Not thread safe
  Assumes every processor has the same domain shape

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

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

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

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”

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”

\texttt{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”

\texttt{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”

\textbf{L.2.225 File snb.cxx}

\begin{verbatim}
namespace bout
    SNB model
\end{verbatim}

\textbf{L.2.226 File snb.hxx}

\begin{verbatim}
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

    HeatFluxSNB ()
        Construct using the options in the “snb” section.

    HeatFluxSNB (Options \& options)
        Construct using options in given section.

    ~HeatFluxSNB ()

    HeatFluxSNB (HeatFluxSNB\&\&)

    HeatFluxSNB &operator= (HeatFluxSNB\&\&)

    HeatFluxSNB (const HeatFluxSNB\&)

    HeatFluxSNB &operator= (const HeatFluxSNB\&)
\end{verbatim}
**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=1.602e-19 to get Watts per cubic meter.

**Private Functions**

*BoutReal int_beta4_exp (BoutReal beta)*

Indefinite integral of beta^4 * exp(-beta) with constant set to zero

*BoutReal groupWeight (BoutReal beta_min, BoutReal beta_max)*

(1/24) * Integral of beta^4 * exp(-beta) from beta_min to beta_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.

### L.2.227 File snes.cxx

**Functions**

static PetscErrorCode FormFunction (SNES snes, Vec x, Vec f, void *ctx)

### L.2.228 File snes.hxx

**class SNESSolver : public Solver**

#include <snes.hxx> Uses PETSc’s SNES interface to find a steady state solution to a nonlinear ODE

**Public Functions**

SNESSolver (Options *opt = nullptr)

~SNESSolver ()

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
int \textbf{run} ()
    Run the solver, calling monitors nout times, at intervals of tstep. This function is called by \textit{solve()}, and is specific to each solver type

    This should probably be protected, since it shouldn't be called by users.

\begin{verbatim}
PetscErrorCode \textbf{snes\_function} (Vec x, Vec f)
    Nonlinear function.
\end{verbatim}

\subsection*{Private Members}

int \textbf{mxstep}
    Maximum number of internal steps between outputs.

int \textbf{nlocal}
    Number of variables on local processor.

int \textbf{neq}
    Number of variables in total.

\begin{verbatim}
\textbf{PetscLib lib}
    Handles initialising, finalising PETSc.
\end{verbatim}

Vec \textbf{snes\_f}
    Used by SNES to store function.

Vec \textbf{snes\_x}
    Result of SNES.

SNES \textbf{snes}
    SNES context.

Mat \textbf{Jmf}
    Matrix-free Jacobian.

\section*{L.2.229 File solver.cxx}

\subsection*{Functions}

template<class T>
int \textbf{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

    \textbf{FIXME}: This could be a lambda local to getLocalN with an auto argument in C++14

\subsection*{Variables}

bool \textbf{user\_requested\_exit}

\section*{L.2.230 File solver.hxx}

\subsection*{Defines}

\begin{verbatim}
BOUT\_NO\_USING\_NAMESPACE\_BOUTGLOBALS
\end{verbatim}
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`

Using `RegisterSolver` = `RegisterInFactory<Solver, DerivedType, SolverFactory>`
Simpler name for `Factory` registration helper class

Usage:
```
#include <bout/solverfactory.hxx>
namespace {
    RegisterSolver<MySolver> registersolvermine("mysolver");
}
```

Enums

Enum `SOLVER_VAR_OP`
Values:
- `LOAD_VARS`
- `LOAD_DERIVS`
- `SET_ID`
- `SAVE_VARS`
- `SAVE_DERIVS`

Enum `MonitorPosition`:
A type to set where in the list monitors are added.
Values:
- `BACK`
- `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"`
```cpp
constexpr auto SOLVERRK4 = "rk4"
constexpr auto SOLVERSEULER = "euler"
constexpr auto SOLVERRK3SSP = "rk3ssp"
constexpr auto SOLVERPOWER = "power"
constexpr auto SOLVERARKODE = "arkode"
constexpr auto SOLVERIMEXBDF2 = "imexbdf2"
constexpr auto SOLVERSNES = "snes"
constexpr auto SOLVERRKGeneric = "rkgeneric"

class SolverFactory: public Factory<Solver, SolverFactory>

Public Static Attributes

constexpr auto type_name = "Solver"
constexpr auto section_name = "solver"
constexpr auto option_name = "type"
constexpr auto default_type = SOLVERCVODE

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:

auto 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");
auto 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");
auto 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

solver->solve();

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:

solver->solve(NOUT, TIMESTEP);


Public Functions

Solver (Options *opts = nullptr)

virtual ~Solver()

void setModel (PhysicsModel *model)
  Specify physics model to solve. Currently only one model can be evolved by a Solver.

virtual void setRHS (rhsfunc f)
  Set the RHS function.

void setPrecon (PhysicsPrecon f)
  Specify a preconditioner (optional)

virtual void setJacobian (Jacobian f)
  Specify a Jacobian (optional)

void setSplitOperator (rhsfunc fC, rhsfunc fD)
  Split operator solves.
void addMonitor (Monitor *monitor, MonitorPosition pos = MonitorPosition::FRONT)

Add a monitor to be called regularly.

The frequency at which the monitor is called is set by Monitor::timestep. By default this is every output timestep. When a new Monitor with a smaller timestep is added, the solver attempts to adjust its internal timestep to match, as well as adjusting the timesteps of the current set of Monitors to be multiples of the new timestep. If this is not possible, addMonitor will throw an exception.

Adding new Monitors after the Solver has been initialised is only possible if their timestep is a multiple of the Solver’s timestep. Smaller timesteps will throw an exception.

void removeMonitor (Monitor *monitor)

Remove a monitor function previously added.

void addTimestepMonitor (TimestepMonitorFunc monitor)

Add a monitor function to be called every timestep.

void removeTimestepMonitor (TimestepMonitorFunc monitor)

Remove a previously added timestep monitor.

void add (Field2D &v, const std::string &name)

Add a variable to be solved. This must be done in the initialisation stage, before the simulation starts.

void add (Field3D &v, const std::string &name)

void add (Vector2D &v, const std::string &name)

void add (Vector3D &v, const std::string &name)

virtual bool constraints ()

Returns true if constraints available.

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.

void constraint (Field3D &v, Field3D &C_v, std::string name)

void constraint (Vector2D &v, Vector2D &C_v, std::string name)

void constraint (Vector3D &v, Vector3D &C_v, std::string name)

virtual void setMaxTimestep (BoutReal dt)

Set a maximum internal timestep (only for explicit schemes).

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: Number of output timesteps
• dt: The time between outputs
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.

virtual void resetInternalFields ()
Should wipe out internal field vector and reset from current field object data.

virtual int n2Dvars () const
Number of 2D variables. Vectors count as 3.

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.

bool splitOperator ()
Test if this solver supports split operators (e.g. implicit/explicit)

void outputVars (Datafile &outputfile, bool save_repeat = true)
Add evolving variables to output (dump) file or restart file

Parameters

• outputfile: The file to add variable to
• save_repeat: If true, add variables with time dimension

Public Members

bool canReset = {false}

Public Static Functions

std::unique_ptr<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.

std::unique_ptr<Solver> create (const SolverType &type, Options *opts = nullptr)
Create a Solver object, specifying the type.

static 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

```cpp
constexpr MonitorPosition BACK = MonitorPosition::BACK
constexpr MonitorPosition FRONT = MonitorPosition::FRONT
```

Protected Functions

```cpp
int getLocalN()
    Calculate the number of evolving variables on this processor.

template<class T>
bool contains(const std::vector<VarStr<T>> &vars, const std::string &name)
    Does vars contain a field with name?

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:

    ```cpp
    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 have_user_precon()
    Do we have a user preconditioner?

int run_precon (BoutReal t, BoutReal gamma, BoutReal delta)

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.

auto getMonitors () const
   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

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

int *pargc = nullptr
   Number of command-line arguments.

char ***pargv = nullptr
   Command-line arguments.

Private Functions

void add_mms_sources (BoutReal t)

void calculate_mms_error (BoutReal t)
void \texttt{pre\_rhs}(\textit{BoutReal }t) \\
Should be run before user RHS is called.

void \texttt{post\_rhs}(\textit{BoutReal }t) \\
Should be run after user RHS is called.

void \texttt{loop\_vars\_op}(\textit{Ind2D} i2d, \textit{BoutReal *}udata, int &p, \textit{SOLVER\_VAR\_OP} op, bool \textit{bdry}) \\
Loading data from BOUT++ to/from solver.

Perform an operation at a given \textit{Ind2D} (jx,jy) location, moving data between BOUT++ and CVODE.

void \texttt{loop\_vars}(\textit{BoutReal *}udata, \textit{SOLVER\_VAR\_OP} op) \\
Loop over variables and domain. Used for all data operations for consistency.

bool \texttt{var\_Added}(\textit{const std::string &name}) \\
Check if a variable has already been added.

\textit{BoutReal} \texttt{adjust\_Monitor\_Periods}(\textit{Monitor *}\textit{monitor}) \\
(Possibly) adjust the periods of \textit{monitor}, and the monitors timesteps, returning the new \textit{Solver} timestep

void \texttt{finalise\_Monitor\_Periods}(int &\textit{NOUT}, \textit{BoutReal} &\textit{output\_timestep}) \\
Fix all the monitor periods based on \textit{output\_timestep}, as well as adjusting \textit{NOUT} and \textit{output\_timestep} to be consistent

\textbf{Private Members}

int \texttt{rhs\_ncalls} = \{0\} \\
Number of calls to the RHS function.

int \texttt{rhs\_ncalls\_e} = \{0\} \\
Number of calls to the explicit (convective) RHS function.

int \texttt{rhs\_ncalls\_i} = \{0\} \\
Number of calls to the implicit (diffusive) RHS function.

int \texttt{default\_monitor\_period} = \{1\} \\
Default sampling rate at which to call monitors - same as output to screen.

\textit{BoutReal} \texttt{internal\_timestep} = \{-1\} \\
timestep - shouldn’t be changed after init is called.

\textit{PhysicsModel *}\texttt{model} = \{nullptr\} \\
Physics model being evolved.

\textit{rhsfunc} \texttt{phys\_run} = \{nullptr\} \\
The user’s RHS function.

\textit{PhysicsPrecon} \texttt{prefunc} = \{nullptr\} \\
The user’s preconditioner function.

bool \texttt{split\_operator} = \{false\} \\
Is the physics model using separate convective (explicit) and diffusive (implicit) RHS functions?

\textit{rhsfunc} \texttt{phys\_conv} = \{nullptr\} \\
Convective part (if split operator)

\textit{rhsfunc} \texttt{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.

Friends

template<class T>
bool operator==(const VarStr<T> &var, const std::string &name)
    Does var represent field name?

template<class T>
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?
L.2.231 File solverfactory.hxx

L.2.232 File sourcex.cxx

Functions

BoutReal TanH (BoutReal a)

const Field2D source_tanh (const Field2D &f, BoutReal swidth, BoutReal slength)
const Field2D source_exp (const Field2D &f, BoutReal swidth, BoutReal slength)
const Field3D sink_tanh (const Field2D &f0, const Field3D &f, BoutReal swidth, BoutReal slength, bool BoutRealspace)
const Field3D mask_x (const Field3D &f, bool BoutRealspace)
const Field3D sink_tanhx (const Field2D &f0, const Field3D &f, BoutReal swidth, BoutReal slength, bool BoutRealspace)
const Field3D sink_tanhxl (const Field2D &f0, const Field3D &f, BoutReal swidth, BoutReal slength)
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)

L.2.233 File sourcex.hxx

Functions

const Field3D mask_x (const Field3D &f, bool BoutRealspace = true)
const Field2D source_tanh (const Field2D &f, BoutReal swidth, BoutReal slength)
const Field2D source_exp (const Field2D &f, BoutReal swidth, BoutReal slength)
const Field3D sink_tanh (const Field2D &f0, const Field3D &f, BoutReal swidth, BoutReal slength, bool BoutRealspace = true)
const Field3D sink_tanhx (const Field2D &f0, const Field3D &f, BoutReal swidth, BoutReal slength)
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)

L.2.234 File split-rk.cxx

L.2.235 File split-rk.hxx

Defines

SPLITRK_HXX

class SplitRK : public Solver
**Public Functions**

`SplitRK(Options *opt = nullptr)`

`~SplitRK()`

`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

`int run()`

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.
\textbf{BoutReal} $\text{max\_timestep} = \{1.0\}$

Maximum timestep.

\textbf{BoutReal} $\text{max\_timestep\_change} = \{2.0\}$

Maximum factor by which the timestep should be changed.

\texttt{int mxstep} $= \{1000\}$

Maximum number of internal steps between outputs.

\texttt{int adapt\_period} $= \{1\}$

Number of steps between checks.

\texttt{bool diagnose} $= \{false\}$

Turn on diagnostic output.

\texttt{int nlocal} $= \{0\}$

Number of variables on local processor and in total.

\texttt{Array<BoutReal> state}

System state.

\texttt{Array<BoutReal> u1}

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> statel1}

Arrays used for adaptive timesteping.

\texttt{Array<BoutReal> state2}

\section*{L.2.236 File spt.cxx}

Simple Parallel Tridiagonal solver.

\textit{Changelog}

2014-06 Ben Dudson benjamin.dudson@york.ac.uk

- 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 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/}. 
L.2.237 File spt.hxx

```cpp
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.

Parameters

- b: RHS values (Ax = b)
- flags: Inversion settings (see boundary.h for values)
- a: This is a 2D matrix which allows solution of A = Delp2 + a
- data: Structure containing data needed for second half of inversion
- ccoef: Optional coefficient for first-order derivative
- d: Optional factor to multiply the Delp2 operator

Public Functions

LaplaceSPT(Options *opt = nullptr, const CELL_LOC loc = CELL_CENTRE, Mesh *mesh_in = nullptr)

~LaplaceSPT()

void setCoefA(const Field2D &val)
    Set coefficients for inversion. Re-builds matrices if necessary.

void setCoefC(const Field2D &val)

void setCoefD(const Field2D &val)

void setCoefEx(const Field2D &val)

void setCoefEz(const Field2D &val)

FieldPerp solve(const FieldPerp &b)

FieldPerp solve(const FieldPerp &b, const FieldPerp &x0)

Field3D solve(const Field3D &b)
    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

Field3D solve(const Field3D &b, const Field3D &x0)
    Performs the laplacian inversion y-slice by y-slice
```
Return  x All the y-slices of x_slice in the equation A*x_slice = b_slice

Parameters

- b: All the y-slices of b_slice, which is the right hand side of the equation A*x_slice = b_slice
- x0: All the y-slices of the variable eventually used to set BC

Private Types

enum [anonymous]
Values:

SPT_DATA = 1123

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: Vector of matrix coefficients (Left of diagonal)
- b: Vector of matrix coefficients (Diagonal)
- c: Vector of matrix coefficients (Right of diagonal)
- r: RHS vector
- u: Result vector (Au = r)
- n: Size of the matrix
- gam: Intermediate values used for backsolve stage
- bet:
- um:
- start:

void tridagBack(dcomplex *u, int n, dcomplex *gam, dcomplex &gp, dcomplex &up)

Second (backsolve) part of the Thomas algorithm.

Parameters

- u: Result to be solved (Au = r)
- n: Size of the problem
- gam: Intermediate values produced by the forward part
- gp: gam from the processor localmesh->PE_XIND + 1, and returned to localmesh->PE_XIND - 1
- up: u from processor localmesh->PE_XIND + 1, and returned to localmesh->PE_XIND - 1
int \texttt{start} (\texttt{const 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 \(Y=0\), 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.

**Parameters**

- \texttt{b}: RHS values (Ax = b)
- \texttt{data}: Structure containing data needed for second half of inversion

int \texttt{next} (\texttt{SPT\_data &data})

Shifts the parallelised Thomas algorithm along one processor.

Returns non-zero when the calculation is complete.

**Parameters**

- \texttt{data}: Structure which keeps track of the calculation

void \texttt{finish} (\texttt{SPT\_data &data, FieldPerp \&x})

Finishes the parallelised Thomas algorithm

**Parameters**

- \texttt{data}: Structure keeping track of calculation
- \texttt{x}: The result

**Private Members**

\texttt{Field2D Ac0ef}

\texttt{Field2D C0ef}

\texttt{Field2D D0ef}

int \texttt{ys}

int \texttt{ye}

\texttt{SPT\_data slicedata}

\texttt{SPT\_data *alldata}

\texttt{Array\langle d\text{complex}\rangle dc1d}

1D in Z for taking FFTs

\texttt{struct SPT\_data}

Data structure for SPT algorithm.
Public Functions

**SPT_data()**

void **allocate**(int *mm*, int *nx*)

**~SPT_data()**

Public Members

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*

---

**L.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/](http://www.gnu.org/licenses/).
Functions

template<
    DIRECTION direction, STAGGER stagger = STAGGER::None, int nGuard = 1, typename FieldType>
void populateStencil(stencil &, const FieldType &f, const typename FieldType::ind_type)

template<
    DIRECTION direction, STAGGER stagger = STAGGER::None, int nGuard = 1, typename FieldType>
stencil populateStencil(const FieldType &f, const typename FieldType::ind_type)

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?G > 2 is not supported

BoutReal m = BoutNaN

BoutReal c = BoutNaN

BoutReal p = BoutNaN

BoutReal pp = BoutNaN

L.2.239 File surfaceiter.cxx

L.2.240 File surfaceiter.hxx

Defines a class for iterating over flux surfaces (surfaces of constant x)

class SurfaceIter
    #include <surfaceiter.hxx> Iterates over Y-Z surfaces, optionally distributing work between processors

Example

SurfaceIter si(mesh);

for( si.first(); !si.isDone(); si.next() ) { // Perform operation at x = si.xpos if(si.closed()) { // A closed flux surface
(a 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

SurfaceIter(Mesh *mesh, bool include_guards = false)
    Constructor, needs a mesh to iterate over

Parameters

• mesh: 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: 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**

int xpos
    X position where iteration is currently at.

**Private Members**

Mesh *m
    The mesh being iterated over.

    const int firstpos
    const int lastpos

L.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... other)

Here we use type inference to allow us to refer to the firstItem in the first Set and the otherItems in this Set. We use this to pass the firstItem 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**

template<>
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**

    **DeferredFunction** (currentFunction f)

    template<typename ...templatePack>
    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>
    produceCombinations (theFunction func)

    L.2.242 File timer.cxx

    L.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

        ```
        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

```cpp
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(""")`

**Timer(const std::string &label)**

Create a timer, continuing from last time if the same label has already been used

**~Timer()**

Stop the timer

**double getTime()**

Get the time in seconds for time particular `Timer` object

```cpp
Timer timer("test");
// Some calculation
output << timer.getTime();
// timer still counting
```

**double getTotalTime()**

Get the total time in seconds since the very first initialisation.

**double resetTime()**

Get the time in seconds, reset timer to zero

### Public Static Functions

**static double getTime(const std::string &label)**

The total time in seconds

**static double getTotalTime(const std::string &label)**

Total time elapsed since the very first initialisation.

**static double resetTime(const std::string &label)**

The total time in seconds, resets the timer to zero

**void cleanup()**

Clears all timers, freeing memory
```cpp
static std::map<std::string, timer_info> getAllInfo ()
    Return the map of all the individual timers.

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

Timer::timer_info &getInfo (const std::string &label)
    Get a timing info object by name or return a new instance.

double getTime (const timer_info &info)
    Get the elapsed time in seconds for timing info.

double getTotalTime (const timer_info &info)
    Get the total elapsed time in seconds since the first initialisation.

double resetTime (timer_info &info)
    Get the elapsed time, reset timing info to zero.

Private Static Attributes

std::map<std::string, Timer::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.
```
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:

```cpp
template <class T>
constexpr bool is_field_v = is_field<T>::value;
```

Examples

```cpp
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 T's 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:

1. 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

2. 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:
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<std::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<std::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<std::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

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)

L.2.245 File type_name.cxx

namespace bout
   SNB model

L.2.246 File type_name.hxx

Defines

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> ()

L.2.247 File uncopyable.hxx

class Uncopyable
   
#include <uncopyable.hxx> Inherit from this class (private) to prevent copying.
   
Subclassed by FormatFactory

Public Functions

Uncopyable (const Uncopyable &)

Uncopyable &operator=(const Uncopyable &)

Protected Functions

Uncopyable ()

~Uncopyable ()

L.2.248 File unused.hxx

Defines

UNUSED (x) Mark a function parameter as unused in the function body

For GCC, expands to
Macro taken from [http://stackoverflow.com/q/7090998/2043465](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**

```c
void someFunction(int UNUSED(x)) {};
```

**MAYBE_UNUSED**

Mark a function parameter as possibly unused in the function body

Unlike `UNUSED`, this has to go around the type as well:

```c
MAYBE_UNUSED(int foo);
```

### L.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`: The string to split (not modified by call)
- `delim`: The delimiter to split on (single char)
- `elems`: 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**: The string to split (not modified by call)
- **delim**: The delimiter to split on (single char)

```cpp
std::string trim(const std::string &s)
```
Strips leading and trailing spaces from a string

Parameters
- **s**: The string to trim (not modified)
- **c**: Collection of characters to remove

```cpp
std::string trimRight (const std::string &s)
```
Strips leading spaces from a string

Parameters
- **s**: The string to trim (not modified)
- **c**: Collection of characters to remove

```cpp
std::string trimLeft (const std::string &s)
```
Strips leading spaces from a string

Parameters
- **s**: The string to trim (not modified)
- **c**: Collection of characters to remove

```cpp
std::string trimComments (const std::string &s, const std::string &c = "#;")
```
Strips the comments from a string

Parameters
- **s**: The string to trim (not modified)
- **c**: Collection of characters to remove

```cpp
std::string toString (const time_t &time)
```
Convert a time stamp to a string This uses std::localtime and std::put_time

---

**L.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.

**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.

*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

int `ROUND` (BoutReal x)

Round x to the nearest integer

template<typename T>

T `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)

template<typename T>

T `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)

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

*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 \( b \)

```cpp
void checkData(BoutReal f)
    Throw an exception if \( f \) is not finite.
```

```cpp
char *copy_string(const char *s)
    Allocate memory and copy string \( s \)
```

```cpp
std::string toString(const BoutReal &val)
    Convert a value to a string by writing to a stringstream
```

```cpp
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.
```

```cpp
template<class T>
std::string toString(const T &val)
    Convert a value to a string by writing to a stringstream
```

```cpp
std::string toString(const Array<BoutReal> &val)
```

```cpp
std::string toString(const Matrix<BoutReal> &val)
```

```cpp
std::string toString(const Tensor<BoutReal> &val)
```

```cpp
std::string toString(const bool &val)
    Convert a bool to “true” or “false”.
```

```cpp
std::string toString(const DirectionTypes &dir)
```

```cpp
std::string toString(const time_t &time)
    Convert a time stamp to a string This uses std::localtime and std::put_time
```

```cpp
const std::string lowercase(const std::string &str)
    Convert a string to lower case
```

```cpp
const std::string uppercase(const std::string &str)
    Convert a string to upper case
```

```cpp
const std::string lowercasequote(const std::string &str)
    Convert to lower case, except inside quotes (" or ‘)
```

```cpp
BoutReal stringToReal(const std::string &s)
    Convert a string to a BoutReal Throws BoutException if can’t be done
```

```cpp
int stringToInt(const std::string &s)
    Convert a string to an int
```

```cpp
std::list<std::string> &strsplit(const std::string &s, char delim, std::list<std::string> &elems)
    Split a string on a given delimiter
```

**Parameters**

- \( s \): The string to split (not modified by call)
- \( \text{delim} \): The delimiter to split on (single char)
- \( \text{elems} \): 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: The string to split (not modified by call)
    • delim: The delimiter to split on (single char)

std::string trim (const std::string &s)
    Strips leading and trailing spaces from a string

    Parameters
    • s: The string to trim (not modified)
    • c: Collection of characters to remove

std::string trimLeft (const std::string &s)
    Strips leading spaces from a string

    Parameters
    • s: The string to trim (not modified)
    • c: Collection of characters to remove

std::string trimRight (const std::string &s)
    Strips leading spaces from a string

    Parameters
    • s: The string to trim (not modified)
    • c: Collection of characters to remove

std::string trimComments (const std::string &s, const std::string &c = "#;")
    Strips the comments from a string

    Parameters
    • s: The string to trim (not modified)
    • c: Collection of characters to remove

template<typename T>
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

template<>
using data_type = T
template<>  
using size_type = int

Public Functions

Matrix()
Matrix(size_type n1, size_type n2)
Matrix(const Matrix &other)

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!
Matrix &operator=(const Matrix &other)
T &operator() (size_type i1, size_type i2)
const T &operator() (size_type i1, size_type i2) const
Matrix &operator=(const T &val)
T *begin()
const T *begin() const
T *end()
const T *end() const
std::tuple<size_type, size_type> shape() const
bool empty() const

void ensureUnique()
    Ensures that this Matrix does not share data with another This should be called before performing any
    write operations on the data.
Array<T> &getData()  
    Access the underlying storage.
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**

template<>
using data_type = T

template<>
using size_type = int

**Public Functions**

Tensor()

Tensor(size_type n1, size_type n2, size_type n3)

Tensor(const Tensor &other)

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!

Tensor &operator=(const Tensor &other)

T &operator()(size_type i1, size_type i2, size_type i3)

const T &operator()(size_type i1, size_type i2, size_type i3) const

Tensor &operator=(const T &val)

T *begin()

const T *begin() const

T *end()

const T *end() const

std::tuple<size_type, size_type, size_type> shape() const

bool empty() const

void ensureUnique()

Ensures that this Tensor does not share data with another. This should be called before performing any write operations on the data.

Array<T> &getData()

Access the underlying storage.

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

template<class T, class ...Args>
_Unique_if<T>::_Single_object make_unique (Args&&... args)

template<class T>
_Unique_if<T>::_Unknown_bound make_unique (size_t n)

template<class T, class ...Args>
_Unique_if<T>::_Known_bound make_unique (Args&&...)

template<class T>
struct _Unique_if

Public Types

template<>
using _Single_object = std::unique_ptr<T>

template<class T>
struct _Unique_if<T[]>

Public Types

template<>
using _Unknown_bound = std::unique_ptr<T[]>

template<class T, size_t N>
struct _Unique_if<T[N]>

Public Types

template<>
using _Known_bound = void

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

template<>
using result_type = R

template<>
using arg_t = typename arg<i>::type

Public Static Attributes

constexpr size_t nargs = sizeof...(Args)
Total number of arguments.

template<size_t i>
struct arg

Public Types

template<>
template<>
using type = typename std::tuple_element::type

L.2.251 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

```cpp
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 v 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 v cannot hold in which case std::bad_cast will be thrown at runtime.

template<typename Variant>
std::string variantToString(const Variant &v)
```
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>
Target operator() (Source &&source) const

template<typename Source>
Target operator() (Source &&source, std::false_type)

template<typename Source>
Target operator() (Source &&source, std::true_type)

Public Functions

template<typename T>
std::string operator() (T &&val)

L.2.252 File vecops.cxx

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: The field to differentiate
• outloc: The location where the result is desired By default this is the same location as the input f
• method: The method to use. The default is set in the options.

const Vector3D Grad(const Field3D &f, CELL_LOC outloc, const std::string &method)

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

- \( f \): The field to differentiate
- \( \text{outloc} \): The cell location where the result is desired
- \( \text{method} \): The method to use. The default is set in the options.

```cpp
const Vector2D Grad_perp(const Field2D & f, CELL_LOC outloc, const std::string & method)
```

```cpp
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 \( \text{outloc} \) must be CELL_CENTRE

Parameters

- \( v \): The vector to differentiate
- \( \text{outloc} \): The cell location where the result is desired
- \( \text{method} \): The method to use. The default is set in the options.

```cpp
const Field3D Div(const Vector3D & v, CELL_LOC outloc, const std::string & method)
```

```cpp
const Field2D Div(const Vector2D & v, const Field2D & f, CELL_LOC outloc, const std::string & method)
```

```cpp
const Field3D Div(const Vector3D & v, const Field3D & f, CELL_LOC outloc, const std::string & method)
```

```cpp
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

- \( v \): 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
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)
```

Advection of a scalar field \( a \) by a velocity vector \( v \)

Both vectors must be at the same location, which cannot be CELL_VSHIFT

```cpp
const Vector2D V_dot_Grad(const Vector2D & v, const Vector2D & a)
```

```cpp
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)

L.2.253 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/.

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: The field to differentiate
• outloc: The location where the result is desired By default this is the same location as the input f
• method: 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: The field to differentiate
• outloc: The cell location where the result is desired
• method: The method to use. The default is set in the options.

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: The vector to differentiate
• outloc: The cell location where the result is desired
• method: 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")

const Field3D Div(const Vector3D &v, const Field3D &f, DIFF_METHOD method = DIFF_DEFAULT)

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: 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 Field3D &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)

L.2.254 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 = "RGN_ALL")

Absolute value (Modulus) of given vector v

|v| = sqrt( v dot v )

L.2.255 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* (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 Vector2D cross (const Vector2D &lhs, const Vector2D &rhs)

Cross product.
const Vector3D cross (const Vector2D &lhs, const Vector3D &rhs)
Cross product.

const Field2D abs (const Vector2D &v, const std::string &region = "RGN_ALL")
Absolute value (Modulus) of given vector v
lvl = sqrt( v dot v )

const Field2D abs (const Vector2D &v, REGION region)

Vector2D toFieldAligned (Vector2D v, const std::string &region = "RGN_ALL")
Transform to and from field-aligned coordinates.

Vector2D fromFieldAligned (Vector2D v, const std::string &region = "RGN_ALL")

Vector2D emptyFrom (const Vector2D &v)
Create new Vector2D with same attributes as the argument, but uninitialised components.

Vector2D zeroFrom (const Vector2D &v)
Create new Vector2D with same attributes as the argument, and zero-initialised components.

Vector2D &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 ()

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`

void `accept(FieldVisitor &v)`
Visitor pattern support.

bool `isReal() const`
Returns true if field consists of BoutReal values.

bool `is3D() const`
True if variable is 3D.

int `byteSize() const`
Number of bytes for a single point.

int `BoutRealSize() const`
Number of BoutReals (not implemented if not BoutReal)

void `applyBoundary(bool init = false)`
Apply boundary condition to all fields.

void `applyBoundary(const std::string &condition)`

void `applyBoundary(const char *condition)`

void `applyTDerivBoundary()`

**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.

**L.2.256 File vector3d.cxx**

**Defines**

`CROSS(v0, v1, v2)`
Functions

```cpp
const Vector3D cross (const Vector3D &lhs, const Vector3D &rhs)
Cross-product of two vectors.
```

```cpp
const Vector3D cross (const Vector3D &lhs, const Vector2D &rhs)
Cross-product of two vectors.
```

```cpp
const Vector3D cross (const Vector2D &lhs, const Vector3D &rhs)
Cross product.
```

```cpp
const Vector2D cross (const Vector2D &lhs, const Vector2D &rhs)
Cross product.
```

```cpp
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)
```

```cpp
const Field3D abs (const Vector3D &v, const std::string &region = "RGN_ALL")
Absolute magnitude (modulus) of a vector |v|
```

```cpp
sqrt( v.x^2 + v.y^2 + v.z^2 )
```

```cpp
Vector3D toFieldAligned (const Vector3D &v, const std::string &region)
Transform to and from field-aligned coordinates.
```

```cpp
Vector3D fromFieldAligned (const Vector3D &v, const std::string &region)
```

L.2.257 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

```cpp
__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}

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")

Vector3D emptyFrom (const Vector3D &v)
Create new Vector3D with same attributes as the argument, but uninitialised components.

Vector3D zeroFrom (const Vector3D &v)
Create new Vector3D with same attributes as the argument, and zero-initialised components.

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 ()
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

void accept (FieldVisitor &v)
Visitor pattern support.

bool isReal () const
Returns true if field consists of BoutReal values.

bool is3D () const
True if variable is 3D.

int byteSize () const
Number of bytes for a single point.

int BoutRealSize () const
Number of BoutReals (not implemented if not BoutReal)

void applyBoundary (bool init = false)

void applyBoundary (const std::string &condition)

void applyBoundary (const char *condition)

void applyTDerivBoundary ()

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**

\[ \text{Vector3D } *\text{deriv} = \text{nullptr}\]
Time-derivative, can be NULL.

\[ \text{CELL\_LOC } \text{location} = \text{CELL\_CENTRE}\]
Location of the variable in the cell.

**L.2.258 File where.cxx**

**L.2.259 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/.

**Functions**

\[
\text{template-class } T, \text{ class } U, \text{ class } V, \text{ class ResultType } = \text{typename bout::utils::EnableIfField}<T, U, V> >
\]
\[
\text{auto } \text{where (const } T &\text{test, const } U &gt0, \text{ const } V &le0) }
\]

For each point, choose between two inputs based on a third input

**Parameters**

- test: The value which determines which input to use
- gt0: Uses this value if test > 0.0
- le0: Uses this value if test <= 0.0

\[
\text{template-class } T, \text{ class } U, \text{ class ResultType } = \text{typename bout::utils::EnableIfField}<T, U> >
\]
\[
\text{auto } \text{where (const } T &\text{test, const } U &gt0, \text{ BoutReal le0) }
\]

\[
\text{template-class } T, \text{ class } V, \text{ class ResultType } = \text{typename bout::utils::EnableIfField}<T, V> >
\]
\[
\text{auto } \text{where (const } T &\text{test, BoutReal gt0, const } V &le0) }
\]

\[
\text{template-class } T, \text{ class ResultType } = T >
\]
\[
\text{auto } \text{where (const } T &\text{test, BoutReal gt0, BoutReal le0) }
\]
L.3 Python routines

L.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().

Generic routines, useful for all data

L.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.

```python
from boutdata.collect import collect
t = collect("t_array")  # Collect the time values
```

- **pol_slice()** takes a 3 or 4-D data set for a toroidal equilibrium, and calculates a slice through it at fixed toroidal angle.

- **gen_surface()** is a generator for iterating over flux surfaces

Routines for exchanging data to/from BOUT++

boutdata.attributes (varname, path='.', prefix='BOUT.dmp')
Return a dictionary of variable attributes in an output file

**Parameters**

- varname (str) – Name of the variable
- path (str, optional) – Path to data files (default: “.”)
- prefix (str, optional) – File prefix (default: “BOUT.dmp”)

**Returns**  A dictionary of attributes of varname

**Return type**  dict
boutdata.collect(varname, xind=None, yind=None, zind=None, tind=None, path='.', yguards=False, xguards=True, info=True, prefix='BOUT.dmp', strict=False, tind_auto=False, datafile_cache=None)

Collect a variable from a set of BOUT++ outputs.

**Parameters**

- **varname** *(str)* – Name of the variable
- **xind, yind, zind, tind** *(int, slice or list of int, optional)* – Range of X, Y, Z or time indices to collect. Either a single index to collect, a list containing [start, end] (inclusive end), or a slice object (usual python indexing). Default is to fetch all indices
- **path** *(str, optional)* – Path to data files (default: ".")
- **prefix** *(str, optional)* – File prefix (default: “BOUT.dmp”)
- **yguards** *(bool or “include_upper”, optional)* – Collect Y boundary guard cells? (default: False) If yguards==”include_upper” the y-boundary cells from the upper (second) target are also included.
- **xguards** *(bool, optional)* – Collect X boundary guard cells? (default: True) (Set to True to be consistent with the definition of nx)
- **info** *(bool, optional)* – Print information about collect? (default: True)
- **strict** *(bool, optional)* – Fail if the exact variable name is not found? (default: False)
- **tind_auto** *(bool, optional)* – Read all files, to get the shortest length of time_indices. Useful if writing got interrupted (default: False)
- **datafile_cache** *(datafile_cache_tuple, optional)* – Optional cache of open DataFile instances: namedtuple as returned by create_cache. Used by BoutOutputs to pass in a cache so that we do not have to re-open the dump files to read another variable (default: None)

**Examples**

```python
>>> collect(name)
BoutArray([[[[[...]]]]])
```

### L.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. 12.1 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. 12.1: Longest possible folder tree made by the `self.execute_runs()` function.](image)

# L.4 boutdata package

## L.4.1 Module contents

Routines for exchanging data to/from BOUT++

```python
boutdata.attributes(varname, path='.', prefix='BOUT.dmp')
```

Return a dictionary of variable attributes in an output file

**Parameters**

- `varname` (**str**) – Name of the variable
- `path` (**str**, optional) – Path to data files (default: “.”)
- `prefix` (**str**, optional) – File prefix (default: “BOUT.dmp”)

**Returns**

A dictionary of attributes of `varname`

**Return type**

```
boutdata.collect(varname, xind=None, yind=None, zind=None, tind=None, path='.', xguards=False, yguards=True, info=True, prefix='BOUT.dmp', strict=False, tind_auto=False, datafile_cache=None)
```

Collect a variable from a set of BOUT++ outputs.

**Parameters**

- `varname` (**str**) – Name of the variable
• xind, yind, zind, tind (int, slice or list of int, optional) – Range of X, Y, Z or time indices to collect. Either a single index to collect, a list containing [start, end] (inclusive end), or a slice object (usual python indexing). Default is to fetch all indices

• path (str, optional) – Path to data files (default: “.”)
• prefix (str, optional) – File prefix (default: “BOUT.dmp”)
• yguards (bool or “include_upper”, optional) – Collect Y boundary guard cells? (default: False) If yguards==”include_upper” the y-boundary cells from the upper (second) target are also included.
• xguards (bool, optional) – Collect X boundary guard cells? (default: True) (Set to True to be consistent with the definition of nx)
• info (bool, optional) – Print information about collect? (default: True)
• strict (bool, optional) – Fail if the exact variable name is not found? (default: False)
• tind_auto (bool, optional) – Read all files, to get the shortest length of time_indices. Useful if writing got interrupted (default: False)
• datafile_cache (datafile_cache_tuple, optional) – Optional cache of open DataFile instances: namedtuple as returned by create_cache. Used by BoutOutputs to pass in a cache so that we do not have to re-open the dump files to read another variable (default: None)

Examples

```python
>>> collect(name)
BoutArray([[[[...]]]])
```

L.4.2 Submodules

L.4.3 boutdata.cbdtoeqdsk module

```python
class boutdata.cbdtoeqdsk.Bunch
boutdata.cbdtoeqdsk.cbmtogeqdsk(g)
```

L.4.4 boutdata.collect module

```python
boutdata.collect.attributes(varname, path='.', prefix='BOUT.dmp')
```

Return a dictionary of variable attributes in an output file

Parameters

• varname (str) – Name of the variable
• path (str, optional) – Path to data files (default: “.”)
• prefix (str, optional) – File prefix (default: “BOUT.dmp”)

Returns A dictionary of attributes of varname

Return type dict
boutdata.collect.collect(varname, xind=None, yind=None, zind=None, tind=None, path='.', yguards=False, xguards=True, info=True, prefix='BOUT.dmp', strict=False, tind_auto=False, datafile_cache=None)

Collect a variable from a set of BOUT++ outputs.

**Parameters**

- **varname** *(str)* – Name of the variable
- **xind, yind, zind, tind** *(int, slice or list of int, optional)* – Range of X, Y, Z or time indices to collect. Either a single index to collect, a list containing [start, end] (inclusive end), or a slice object (usual python indexing). Default is to fetch all indices
- **path** *(str, optional)* – Path to data files (default: ".")
- **prefix** *(str, optional)* – File prefix (default: "BOUT.dmp")
- **yguards** *(bool or “include_upper”, optional)* – Collect Y boundary guard cells? (default: False) If yguards="’include_upper’" the y-boundary cells from the upper (second) target are also included.
- **xguards** *(bool, optional)* – Collect X boundary guard cells? (default: True) (Set to True to be consistent with the definition of nx)
- **info** *(bool, optional)* – Print information about collect? (default: True)
- **strict** *(bool, optional)* – Fail if the exact variable name is not found? (default: False)
- **tind_auto** *(bool, optional)* – Read all files, to get the shortest length of time_indices. Useful if writing got interrupted (default: False)
- **datafile_cache** *(datafile_cache_tuple, optional)* – Optional cache of open DataFile instances: namedtuple as returned by create_cache. Used by BoutOutputs to pass in a cache so that we do not have to re-open the dump files to read another variable (default: None)

**Examples**

```python
>>> collect(name)
BoutArray([[[[...]]]])
```

boutdata.collect.create_cache(path, prefix)

Create a list of DataFile objects to be passed repeatedly to collect.

**Parameters**

- **path** *(str)* – Path to data files
- **prefix** *(str)* – File prefix

**Returns** namedtuple – The cache of DataFiles in a namedtuple along with the file_list, and parallel and suffix attributes

**Return type** *(list of str, bool, str, list of DataFile)*

boutdata.collect.dimensions(varname, path='.', prefix='BOUT.dmp')

Return the names of dimensions of a variable in an output file

**Parameters**

- **varname** *(str)* – Name of the variable
- **path** *(str, optional)* – Path to data files (default: ".")
BOUT++ Documentation, Release 5.0.0-alpha

- **prefix** *(str, optional)* – File prefix (default: “BOUT.dmp”)

**Returns** The elements of the tuple give the names of corresponding variable dimensions

**Return type** tuple of strs

*boutdata.collect.findFiles(path, prefix)*

Find files matching prefix in path.

Netcdf (“.nc”, “.ncdf”, “.cdl”) and HDF5 (“.h5”, “.hdf5”, “.hdf”) files are searched.

**Parameters**

- **path** *(str)* – Path to data files
- **prefix** *(str)* – File prefix

**Returns** tuple – The first element of the tuple is the list of files, the second is whether the files are a parallel dump file and the last element is the file suffix.

**Return type** (list of str, bool, str)

*boutdata.collect.findVar(varname, varlist)*

Find variable name in a list

First does case insensitive comparison, then checks for abbreviations.

Returns the matched string, or raises a ValueError

**Parameters**

- **varname** *(str)* – Variable name to look for
- **varlist** *(list of str)* – List of possible variable names

**Returns** The closest match to varname in varlist

**Return type** str

**L.4.5 boutdata.data module**

Provides a class BoutData which makes access to code inputs and outputs easier. Creates a tree of maps, inspired by approach used in OMFIT

*boutdata.data.BoutData(path='.', prefix='BOUT.dmp', caching=False, **kwargs)*

Returns a dictionary, containing the contents of a BOUT++ output directory.

Does not allow writing, only reading of data. By default there is no cache, so each time a variable is read it is collected; if caching is set to True variables are stored once they are read.

**Parameters**

- **path** *(str, optional)* – Path to data files (default: “.”)
- **prefix** *(str, optional)* – File prefix (default: “BOUT.dmp”)
- **caching** *(bool, float, optional)* – Switches on caching of data, so it is only read into memory when first accessed (default False) If caching is set to a number, it gives the maximum size of the cache in GB, after which entries will be discarded in first-in-first-out order to prevent the cache getting too big. If the variable being returned is bigger than the maximum cache size, then the variable will be returned without being added to the cache, and the rest of the cache will be left (default: False)
• **DataFileCaching** (*bool, optional*) – Switch for creation of a cache of DataFile objects to be passed to collect so that DataFiles do not need to be re-opened to read each variable (default: True)

• **kwargs** – Keyword arguments that are passed through to collect()

**Returns** Contents of a BOUT++ output directory, including options and output files

**Return type** dict

### Examples

```python
>>> d = BoutData(".")  # Current directory

>>> d.keys()          # List all valid keys

>>> print(d["options"])) # Prints tree of options

>>> d["options"]["nout"] # Value of nout in BOUT.inp file

>>> print(d["outputs"])) # Print available outputs

>>> d["outputs"]["ne"] # Read "ne" from data files

>>> d = BoutData(".", prefix="BOUT.dmp", caching=True) # Turn on caching
```

```python
class boutdata.data.BoutOptions( name='root', parent=None)
This class represents a tree structure. Each node (BoutOptions object) can have several sub-nodes (sections), and several key-value pairs.

**Parameters**

• **name** (*str, optional*) – Name of the root section (default: “root”)

• **parent** (*BoutOptions, optional*) – A parent BoutOptions object (default: None)

**Examples**

```python
>>> optRoot = BoutOptions() # Create a root

Specify value of a key in a section “test” If the section does not exist then it is created

```python
>>> optRoot.getSection("test")["key"] = 4

Get the value of a key in a section “test” If the section does not exist then a KeyError is raised

```python
>>> print(optRoot["test"]["key"])
4

To pretty print the options

```python
>>> print(optRoot)
[test]
key = 4
```
as_dict()
Return a nested dictionary of all the options.

as_tree(indent="")
Return a string formatted as a pretty version of the options tree

evaluate_scalar(name)
Evaluate (recursively) scalar expressions

getSection(name)
Return a section object. If the section does not exist then it is created

Parameters name (str) – Name of the section to get/create

Returns A new section with the original object as the parent

Return type BoutOptions

keys()
Returns all keys, including sections and values

path()
Returns the path of this section, joining together names of parents

pop(k[, d]) → v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised.

rename(old_name, new_name)
Rename old_name to new_name

sections()
Return a list of sub-sections

values()
Return a list of values

class boutdata.data.BoutOptionsFile (filename=’BOUT.inp’, name=’root’, gridfilename=None, nx=None, ny=None, nz=None)
Parses a BOUT.inp configuration file, producing a tree of BoutOptions.

Slight differences from ConfigParser, including allowing values before the first section header.

Parameters

• filename (str, optional) – Path to file to read

• name (str, optional) – Name of root section (default: “root”)

• gridfilename (str, optional) – If present, path to gridfile from which to read grid sizes (nx, ny, nz)

• nx, ny (int, optional) –
  – Specify sizes of grid, used when evaluating option strings
  – Cannot be given if gridfilename is specified
  – Must both be given if either is
  – If neither gridfilename nor nx, ny are given then will try to find nx, ny from (in order) the ‘mesh’ section of options, outputfiles in the same directory is the input file, or the grid file specified in the options file (used as a path relative to the current directory)

• nz (int, optional) – Use this value for nz when evaluating option expressions, if given. Overrides values found from input file, output files or grid files
Examples

```python
>>> opts = BoutOptionsFile("BOUT.inp")
>>> print(opts)  # Print all options in a tree
root
  |- nout = 100
  |- timestep = 2
  ...

>>> opts["All"]["scale"]  # Value "scale" in section "All"
1.0
```

```python
COMMENT_REGEX = re.compile('(.*?)(\s*)([#;].*)')
VALID_COMMENTS = ('#', ';')
evaluate(name)
    Evaluate (recursively) expressions
    Sections and subsections must be given as part of ‘name’, separated by colons
Parameters name (str) – Name of variable to evaluate, including sections and subsections
```

```python
recalculate_xyz(*, nx=0, ny=0, nz=0)
    Recalculate the x, y and z arrays used to evaluate expressions
```

```python
write(filename=None, overwrite=False)
    Write to BOUT++ options file
    This method will throw an error rather than overwriting an existing file unless the overwrite argument is
    set to true. Note, no comments from the original input file are transferred to the new one.
Parameters
  • filename (str) – Path of the file to write (defaults to path of the file that was read in)
  • overwrite (bool) – If False then throw an exception if ‘filename’ already exists. Otherwise, just overwrite without asking. (default False)
```

```python
class boutdata.data.BoutOutputs(path='.', prefix='BOUT.dmp', suffix=None, caching=False, DataFileCaching=True, **kwargs)
    Emulates a map class, represents the contents of a BOUT++ dmp files. Does not allow writing, only reading of
data. By default there is no cache, so each time a variable is read it is collected; if caching is set to True variables
are stored once they are read. Extra keyword arguments are passed through to collect.
Parameters
  • path (str, optional) – Path to data files (default: ‘.’)
  • prefix (str, optional) – File prefix (default: “BOUT.dmp”)
  • suffix (str, optional) – File suffix (default: None, searches all file extensions)
  • caching (bool, float, optional) – Switches on caching of data, so it is only read into memory when first accessed (default False) If caching is set to a number, it gives the maximum size of the cache in GB, after which entries will be discarded in first-in-first-out order to prevent the cache getting too big. If the variable being returned is bigger than the maximum cache size, then the variable will be returned without being added to the cache, and the rest of the cache will be left (default: False)
  • DataFileCaching (bool, optional) – Switch for creation of a cache of DataFile objects to be passed to collect so that DataFiles do not need to be re-opened to read each variable (default: True)
```

L.4. boutdata package
**kwargs – keyword arguments that are passed through to _caching_collect()

### Examples

```python
>>> d = BoutOutputs(".") # Current directory
```
```
>>> d.keys() # List all valid keys
['iteration',
 'zperiod',
 'MYSUB',
 ... ]
```
```
>>> d.dimensions['ne'] # Get the dimensions of the field ne
('t', 'x', 'y', 'z')
```
```
>>> d['ne'] # Read "ne" from data files
BoutArray([[...]])
```
```
>>> d = BoutOutputs(".", prefix="BOUT.dmp", caching=True) # Turn on caching
evolvingVariables()
    Return a list of names of time-evolving variables
keys()
    Return a list of available variable names

redistribute (npes, nxpe=None, mxg=2, myg=2, include_restarts=True)
    Create a new set of dump files for npes processors.
    Useful for restarting simulations using more or fewer processors.
    Existing data and restart files are kept in the directory "redistribution_backups". redistribute() will fail if
    this directory already exists, to avoid overwriting anything

Parameters
```
- **npes** (int) – Number of new files to create
- **nxpe** (int, optional) – If nxpe is None (the default), then an ‘optimal’ number will be
  selected automatically
- **mxg, myg** (int, optional) – Number of guard cells in x, y (default: 2)
- **include_restarts** (bool, optional) – If True, then restart.redistribute will be used to
  redistribute the restart files also (default: True)
```

```python
class boutdata.data.CaseInsensitiveDict(**kwargs)
```

### L.4.6 boutdata.gen_surface module

Flux surface generator for tokamak grid files

boutdata.gen_surface.gen_surface(grid)
    Generator for iterating over flux surfaces

Parameters grid (DataFile) – An input grid file to read to find flux surfaces

Yields tuple ((int, list of int, bool)) – A tuple containing the x index, list of y indices and whether
    this flux surface is periodic
L.4.7 boutdata.griddata module

Routines for manipulating grid files

boutdata.griddata.bout2sonnet(grdname, outf)

Creates a Sonnet format grid from a BOUT++ grid.

NOTE: Branch cuts are not yet supported

Parameters

- **grdname** (str) – Filename of BOUT++ grid file
- **outf** (File) – The file-like object to write to

Examples

```python
>>> with open("output.sonnet", "w") as f:
...     bout2sonnet("BOUT.grd.nc", f)
```

boutdata.griddata.gridcontourf(grid, data2d, nlevel=31, show=True, mind=None, maxd=None, symmetric=False, cmap=None, ax=None, xlabel='Major radius [m]', ylabel='Height [m]', separatrix=False)

Plots a 2D contour plot, taking into account branch cuts (X-points).

Parameters

- **grid** (DataFile) – A DataFile object
- **data2d** (array_like) – A 2D (x,y) NumPy array of data to plot
- **nlevel** (int, optional) – Number of levels in the contour plot
- **show** (bool, optional) – If True, will immediately show the plot
- **mind** (float, optional) – Minimum data level
- **maxd** (float, optional) – Maximum data level
- **symmetric** (bool, optional) – Make mind, maxd symmetric about zero
- **cmap** (Colormap, optional) – A matplotlib colormap to use. If None, use the current default
- **ax** (Axes, optional) – A matplotlib axes instance to plot to. If None, create a new figure and axes, and plot to that
- **xlabel, ylabel** (str, optional) – Labels for the x/y axes
- **separatrix** (bool, optional) – Add separatrix

Returns

The contourf instance

Return type

con
>>> fig, axis = plt.subplots()
>>> c = gridcontourf(grid, data, show=False, ax=axis)
>>> fig.colorbar(c, ax=axis)
>>> plt.show()

boutdata.griddata.rotate(gridfile, yshift, output=None)
Shifts a grid file by the specified number of points in y
This moves the branch cut around, and can be used to change the limiter location

Parameters
- gridfile (str) – Name of DataFile to rotate
- yshift (int) – Number of points in y to shift by
- output (str, optional) – Name of DataFile to write to. If None, will write to a new file with the same name as gridfile + '_rot'

boutdata.griddata.slice(infile, outfile, region=None, xind=None, yind=None)
Copy an X-Y slice from one DataFile to another

Parameters
- infile (str) – Name of DataFile to read slice from
- outfile (str) – Name of DataFile to write slice to. File will be created, and will be overwritten if it already exists
- region (0, 1, 2, 3, 4, 5, None, optional) – Copy a whole region. The available regions are:
  - 0: Lower inner leg
  - 1: Inner core
  - 2: Upper inner leg
  - 3: Upper outer leg
  - 4: Outer core
  - 5: Lower outer leg
- xind, yind ((int, int), optional) – Index ranges for x and y. Range includes first point, but not last point

L.4.8 boutdata.input module

Fourier transform data for input to BOUT++

boutdata.input.transform3D(arr)
Fourier transforms a 3D array in the Z dimension

BOUT++ can take 3D inputs to be Fourier transformed in the Z direction.

Parameters arr (array_like) – Input 3-D array

Returns
A 3D array [x,y,kz] where kz is organised in the standard FFT order, with constant (DC, kz=0) component first, followed by real/imaginary pairs.

kz = [0, (real, imag), (real, imag), ...]
Return type  array_like

L.4.9  boutdata.mayavi2 module

L.4.10  boutdata.mms module

L.4.11  boutdata.pol_slice module

\[ \text{boutdata.pol_slice.pol_slice}(\text{var3d}, \text{gridfile}, n=1, \text{zangle}=0.0, \text{nyInterp}=\text{None}) \]

Takes a 3D variable, and returns a 2D slice at fixed toroidal angle

Parameters

- `var3d` (array_like) – The input array. Should be 3D
- `gridfile` (str) – The gridfile containing the coordinate system to used
- `n` (int, optional) – The number of times the data must be repeated for a full torus, e.g. n=2 is half a torus
- `zangle` (float, optional) – The (real) toroidal angle of the result
- `nyInterp` (int, optional) – The number of y (theta) points to use in the final result.

Returns  A 2D-slice of var3d interpolated at a fixed toroidal angle

Return type  array

L.4.12  boutdata.processor_rearrange module

Routines for redistributing files over different numbers of processors

\[ \text{boutdata.processor_rearrange.create_processor_layout}(\text{old\_processor\_layout}, \text{npes}, \text{nxpe=\text{None}}) \]

Convert one processor layout into another one with a different total number of processors

If nxpe is None, use algorithm from BoutMesh to select optimal nxpe. Otherwise, check nxpe is valid (divides npes)

Parameters

- `old_processor_layout` (processor_layout) – The processor layout to convert
- `npes` (int) – The new total number of processors
- `nxpe` (int, optional) – The number of processors in x to use

Returns  A description of the processor layout and grid sizes

Return type  processor_layout

\[ \text{boutdata.processor_rearrange.get_processor_layout}(\text{boutfile}, \text{has\_t\_dimension}=\text{True}, \text{mxg}=2, \text{myg}=2) \]

Given a BOUT.restart.* or BOUT.dmp.* file (as a DataFile object), return the processor layout for its data

Parameters

- `boutfile` (DataFile) – Restart or dump file to read
- `has_t_dimension` (bool, optional) – Does this file have a time dimension?
- `mxg, myg` (int, optional) – Number of x, y guard cells
Returns  A description of the processor layout and grid sizes

Return type  processor_layout

class boutdata.processor_rearrange.processor_layout
A namedtuple describing the processor layout, including grid sizes and guard cells

Parameters

- **nxpe, nype** \((\text{int})\) – The number of processors in x and y
- **npes** \((\text{int})\) – The total number of processors
- **mxsub, mysub** \((\text{int})\) – The size of the grid in x and y on a single processor
- **nx, ny, mz** \((\text{int})\) – The total size of the grid in x, y and z
- **mxg** \((\text{int})\) – The number of guard cells in x and y

boutdata.processor_rearrange.processor_layout_
alias of boutdata.processor_rearrange.BOUT_processor_layout

L.4.13 boutdata.restart module

Routines for manipulating restart files

boutdata.restart.addnoise(path='.', var=None, scale=1e-05)
Add random noise to restart files

**Warning:** Modifies restart files in place! This is in contrast to most of the functions in this module!

Parameters

- **path** \((\text{str, optional})\) – Path to restart files (default: current directory)
- **var** \((\text{str, optional})\) – The variable to modify. By default all 3D variables are modified
- **scale** \((\text{float})\) – Amplitude of the noise. Gaussian noise is used, with zero mean and this parameter as the standard deviation

boutdata.restart.addvar(var, value, path='.
Adds a variable with constant value to all restart files.

**Warning:** Modifies restart files in place! This is in contrast to most of the functions in this module!

This is useful for restarting simulations whilst turning on new equations. By default BOUT++ throws an error if an evolving variable is not in the restart file. By setting an option the variable can be set to zero. This allows it to start with a non-zero value.

Parameters

- **var** \((\text{str})\) – The name of the variable to add
- **value** \((\text{float})\) – Constant value for the variable
- **path** \((\text{str, optional})\) – Input path to data files (default: current directory)
boutdata.restart.create(averagelast=1, final=-1, path='data', output='./', informat='nc', outformat=None)

Create restart files from data (dmp) files.

Parameters

- **averagelast** *(int, optional)* – Number of time points (counting from *final*, inclusive) to average over (default is 1 i.e. just take last time-point)
- **final** *(int, optional)* – The last time point to use (default is last, -1)
- **path** *(str, optional)* – Path to original restart files (default: “data”)
- **output** *(str, optional)* – Path to write new restart files (default: current directory)
- **informat** *(str, optional)* – File extension of original files (default: “nc”)
- **outformat** *(str, optional)* – File extension of new files (default: use the same as *informat*)

boutdata.restart.normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently\(^2\), is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution\(^2\).

**Note:** New code should use the *normal* method of a *default_rng()* instance instead; please see the random-quick-start.

Parameters

- **loc** *(float or array_like of floats)* – Mean (“centre”) of the distribution.
- **scale** *(float or array_like of floats)* – Standard deviation (spread or “width”) of the distribution. Must be non-negative.
- **size** *(int or tuple of ints, optional)* – Output shape. If the given shape is, e.g., *(m, n, k)*, then *m* * * *n* * *k* samples are drawn. If size is *None* (default), a single value is returned if *loc* and *scale* are both scalars. Otherwise, *np.broadcast(*loc*, *scale).*size samples are drawn.

**Returns** *out* – Drawn samples from the parameterized normal distribution.

**Return type** ndarray or scalar

See also:

- *scipy.stats.norm()* probability density function, distribution or cumulative density function, etc.
- *Generator.normal()* which should be used for new code.

Notes

The probability density for the Gaussian distribution is

\[ p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \]

where \( \mu \) is the mean and \( \sigma \) the standard deviation. The square of the standard deviation, \( \sigma^2 \), is called the variance.

The function has its peak at the mean, and its “spread” increases with the standard deviation (the function reaches 0.607 times its maximum at \( x+\sigma \) and \( x-\sigma \)). This implies that normal is more likely to return samples lying close to the mean, rather than those far away.

References

Examples

Draw samples from the distribution:

```python
>>> mu, sigma = 0, 0.1  # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```python
>>> abs(mu - np.mean(s))
0.0  # may vary

>>> abs(sigma - np.std(s, ddof=1))
0.1  # may vary
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) * 
... np.exp( - (bins - mu)**2 / (2 * sigma**2) ), 
... linewidth=2, color='r')
>>> plt.show()
```

Two-by-four array of samples from N(3, 6.25):

```python
>>> np.random.normal(3, 2.5, size=(2, 4))
array([[[-4.49401501, 4.00950034, -1.81814867, 7.29718677],
        [0.39924804, 4.68456316, 4.99394529, 4.84057254]])  # random
```

boutdata.restart.redistribute(npes, path='data', nxpe=None, output='.', informat=None, outformat=None, mxg=2, myg=2)

Resize restart files across NPES processors.

Does not check if new processor arrangement is compatible with the branch cuts. In this respect restart.split() is safer. However, BOUT++ checks the topology during initialisation anyway so this is not too serious.

Parameters

- **npes (int)** – Number of processors for the new restart files
• **path** *(str, optional)* – Path to original restart files (default: “data”)

• **nxpe** *(int, optional)* – Number of processors to use in the x-direction (determines split: \( npes = nxpe \times nype \)). Default is None which uses the same algorithm as BoutMesh (but without topology information) to determine a suitable value for nxpe.

• **output** *(str, optional)* – Location to save new restart files (default: current directory)

• **informat** *(str, optional)* – Specify file format of old restart files (must be a suffix understood by DataFile, e.g. ‘nc’). Default uses the format of the first ‘BOUT.restart.*’ file listed by glob.glob.

• **outformat** *(str, optional)* – Specify file format of new restart files (must be a suffix understood by DataFile, e.g. ‘nc’). Default is to use the same as informat.

**Returns**

**Return type** True on success

```python
boutdata.restart.resize(newNx, newNy, newNz, mxg=2, myg=2, path='data', output='./', 
informat='nc', outformat=None, method='linear', maxProc=None, mute=False)
```

Increase/decrease the number of points in restart files.

**NOTE:** Can’t overwrite WARNING: Currently only implemented with uniform BOUT++ grid

**Parameters**

• **newNx, newNy, newNz** *(int)* – nx, ny, nz for the new file (including ghost points)

• **mxg, myg** *(int, optional)* – Number of ghost points in x, y (default: 2)

• **path** *(str, optional)* – Input path to data files

• **output** *(str, optional)* – Path to write new files

• **informat** *(str, optional)* – File extension of input

• **outformat** *(str, optional)* – File extension of output (default: use the same as informat)

• **method** *(‘linear’, ‘nearest’)* – What interpolation method to be used

• **maxProc** *(int, optional)* – Limits maximum processors to use when interpolating if set

• **mute** *(bool, optional)* – Whether or not output should be printed from this function

**Returns** **return** – True on success, else False

**Return type** bool

```python
boutdata.restart.resize3DField(var, data, coordsAndSizesTuple, method, mute)
```

Resize 3D fields

To be called by resize.

Written as a function in order to call it using multiprocess. Must be defined as a top level function in order to be pickable by the multiprocess.

See the function resize for details

```python
boutdata.restart.resizeY(newy, path='data', output='./', informat='nc', outformat=None, myg=2)
```

Increase the number of Y points in restart files

**Note:**
• Can’t overwrite

Parameters

- **newy (int)** – ny for the new file
- **path (str, optional)** – Path to original restart files (default: “data”)
- **output (str, optional)** – Path to write new restart files (default: current directory)
- **informat (str, optional)** – File extension of original files (default: “nc”)
- **outformat (str, optional)** – File extension of new files (default: use the same as informat)
- **myg (int, optional)** – Number of ghost points in y (default: 2)

Returns

Return type: True on success, else False

```python
boutdata.restart.resizeZ(newNz, path='data', output='./', informat='nc', outformat=None)
```

Increase the number of Z points in restart files

Note:

- Can’t overwrite
- Will not yield a result close to the original if there are asymmetries in the z-direction

Parameters

- **newNz (int)** – nz for the new file
- **path (str, optional)** – Path to original restart files (default: “data”)
- **output (str, optional)** – Path to write new restart files (default: current directory)
- **informat (str, optional)** – File extension of original files (default: “nc”)
- **outformat (str, optional)** – File extension of new files (default: use the same as informat)

Returns

Return type: True on success, else False

```python
boutdata.restart.scalevar(var, factor, path='')
```

Scales a variable by a given factor, modifying restart files in place

**Warning:** Modifies restart files in place! This is in contrast to most of the functions in this module!

Parameters

- **var (str)** – Name of the variable
- **factor (float)** – Factor to multiply
- **path (str, optional)** – Path to the restart files (default: current directory)
L.4.14 boutdata.settings module

Parse BOUT.inp settings file

boutdata.settings.get(filename, name, section=None)
    Find and return a single value from a BOUT.inp settings file

Parameters

- filename (str) – Name of the settings file
- name (str) – The name of the setting
- section (str, optional) – The section to look in (default: the global section)

Note that names and sections are case insensitive

Returns Value of the setting. If not found, raises a ValueError

Return type str

Examples

```python
>>> settings.get("BOUT.inp", "nout")
'100'
```
```
>>> settings.get("BOUT.inp", "compress", section="highbeta")
'true'
```

L.4.15 boutdata.shiftz module

boutdata.shiftz.shift(var, zangle, zperiod=1.0)
    Shift a variable in Z, changing between field-aligned and orthogonal X-Z coordinates. This mainly used for
tokamak simulations in field-aligned coordinates.

Parameters

- var (array_like) – Data to be shifted 4D [t,x,y,z] 3D [x,y,z] or [t,x,z] 2D [x,z]
- zangle (array_like) – The shift angle 2D [x,y] (if var is 4D or 3D [x,y,z]) 1D [x] (if var is 3D [t,x,z] or 2D)
- zperiod (float, optional) – The fraction of 2pi covered by the variable in Z. This corre-
sponds to the ZPERIOD variable in BOUT.inp and multiplies the kz wavenumbers by
this factor.

Returns A numpy array of the same size and shape as var

Return type ndarray

Examples
nxz is now in orthogonal X-Z coordinates (X is psi).
Note that in older grid files “qinty” is used rather than “zShift”.

L.4.16 Module contents

Routines for exchanging data to/from BOUT++

**boutdata.**

Attributes (varname, path='.', prefix='BOUT.dmp')

- Collect a variable from a set of BOUT++ outputs.

- **Parameters**
  - varname (str) – Name of the variable
  - path (str, optional) – Path to data files (default: “.”)
  - prefix (str, optional) – File prefix (default: “BOUT.dmp”)

- **Returns**
  - A dictionary of attributes of varname

- **Return type**
  - dict

Collect a variable from a set of BOUT++ outputs.

- **Parameters**
  - varname (str) – Name of the variable
  - xind, yind, zind, tind (int, slice or list of int, optional) – Range of X, Y, Z or time indices to collect. Either a single index to collect, a list containing [start, end] (inclusive end), or a slice object (usual python indexing). Default is to fetch all indices
  - path (str, optional) – Path to data files (default: “.”)
  - prefix (str, optional) – File prefix (default: “BOUT.dmp”)
  - yguards (bool or “include_upper”, optional) – Collect Y boundary guard cells? (default: False) If yguards="include_upper" the y-boundary cells from the upper (second) target are also included.
  - xguards (bool, optional) – Collect X boundary guard cells? (default: True) (Set to True to be consistent with the definition of nx)
  - info (bool, optional) – Print information about collect? (default: True)
  - strict (bool, optional) – Fail if the exact variable name is not found? (default: False)
  - tind_auto (bool, optional) – Read all files, to get the shortest length of time_indices. Useful if writing got interrupted (default: False)
• **datafile_cache** *(datafile_cache_tuple, optional)* – Optional cache of open DataFile instances: namedtuple as returned by create_cache. Used by BoutOutputs to pass in a cache so that we do not have to re-open the dump files to read another variable (default: None)

**Examples**

```python
>>> collect(name)
BoutArray([[[[...]]]])
```

## L.5 boututils package

### L.5.1 Module contents

Generic routines, useful for all data

### L.5.2 Submodules

### L.5.3 boututils.analyse_equil_2 module

### L.5.4 boututils.anim module

### L.5.5 boututils.ask module

Ask a yes/no question and return the answer.

```python
boututils.ask.query_yes_no(question, default='yes')
```

Ask a yes/no question via input() and return their answer.

Answers are case-insensitive.


**Parameters**

- **question** *(str)* – Question to be presented to the user
- **default** *(f"yes", "no", None)* – The presumed answer if the user just hits <Enter>. It must be “yes” (the default), “no” or None (meaning an answer is required of the user).

**Returns**  True if the answer was “yes” or “y”. False if “no” or “n”

**Return type**  bool

### L.5.6 boututils.boutarray module

Wrapper for ndarray with extra attributes for BOUT++ fields.

```python
class boututils.boutarray.BoutArray
Bases: numpy.ndarray
```

Wrapper for ndarray with extra attributes for BOUT++ fields.
Parameters

- **input_array** (*array_like*) – Data to convert to BoutArray
- **attributes** (*dict*) – Dictionary of extra attributes for BOUT++ fields

Notably, these attributes should contain `bout_type`. Possible values are:

- scalar
- Field2D
- Field3D

If the variable is an evolving variable (i.e. has a time dimension), then it is appended with a "-_t"

### L.5.7 boututils.boutgrid module

### L.5.8 boututils.boutwarnings module

Wrappers for warning functions.

Allows raising warnings that are always printed by default.

**exception** `boututils.boutwarnings.AlwaysWarning(*args, **kwargs)`

- `boututils.boutwarnings.alwayswarn(message)`
- `boututils.boutwarnings.defaultwarn(message)`

### L.5.9 boututils.calculus module

Derivatives and integrals of periodic and non-periodic functions

B. Dudson, University of York, Nov 2009

`boututils.calculus.deriv(*args, **kwargs)`

Take derivative of 1D array

- `result = deriv(y) result = deriv(x, y)`
- **keywords**
- `periodic = False` Domain is periodic

`boututils.calculus.deriv2D(data, axis=-1, dx=1.0, noise_suppression=True)`

Takes 1D or 2D Derivative of 2D array using convolution

- `result = deriv2D(data) result = deriv2D(data, dx)`
- **output is 2D** (if only one axis specified) output is 3D if no axis specified [nx, ny, 2] with the third dimension being `dfdx, dfdy`
- **keywords**: `axis = 0/1` If no axis specified 2D derivative will be returned `dx = 1.0` axis spacing, must be 2D if 2D deriv is taken - default is [1.0, 1.0] noise_suppression = True noise suppressing coefficients used to take derivative - default = True

`boututils.calculus.integrate(var, periodic=False)`

Integrate a 1D array

- Return array is the same size as the input
boututils.calculus.simpson_integrate (data, dx, dy, kernel=0.0, weight=1.0)
Integrates 2D data to one value using the simpson method and matrix convolution
result = simpson_integrate(data, dx, dy)

keywords:
kernel - can be supplied if the simpson matrix is calculated ahead of time
   • if not supplied, is calculated within this function
   • if you need to integrate the same shape data over and over, calculated
      it ahead of time using: kernel = simpson_matrix(Nx, Ny, dx, dy)
weight - can be used to scale data if single number
   • can be used to mask data if weight is array (same size as data)

boututils.calculus.simpson_matrix (Nx, Ny, dx, dy)
Creates a 2D matrix of coefficients for the simpson_integrate function
Call ahead of time if you need to perform integration of the same size data with the same dx and dy
Otherwise, simpson_integrate will automatically call this

L.5.10 boututils.closest_line module

boututils.closest_line.closest_line (n, x, y, ri, zi, mind=None)

L.5.11 boututils.contour module

Contour calculation routines
boututils.contour.contour (f, level)
Return a list of contours matching the given level
boututils.contour.find_opoints (var2d)
Find O-points in psi i.e. local minima/maxima
boututils.contour.find_xpoints (var2d)
Find X-points in psi i.e. inflection points

L.5.12 boututils.crosslines module

boututils.crosslines.find_inter (contour1, contour2)
boututils.crosslines.linelineintersect (a, b, atol=1e-08)
Find all intersection points of two lines defined by series of x,y pairs
Intersection points are unordered Colinear lines that overlap intersect at any end points that fall within the overlap

Parameters a, b (ndarray) – 2 column ndarray of x,y values defining a two dimensional line. 1st column is x values, 2nd column is x values.
Notes

An example of 3 segment line is: \( a = \text{numpy.array}([[0.0],[5.0,3.0],[4.0,1]]) \) Function faster when there are no overlapping line segments

add some lines for preventing zero-division

\texttt{boututils.crosslines.unique(a, atol=1e-08)}

Find unique rows in 2d array

**Parameters**

- \( a \) (2d \text{ndarray}, \text{float}) – array to find unique rows in
- \( \text{atol} \) (float, optional) – tolerance to check uniqueness

**Returns**

- \( \text{out} \) – array of unique values

**Return type**

2d \text{ndarray}, \text{float}

Notes


L.5.13 \texttt{boututils.datafile} module

File I/O class

A wrapper around various NetCDF libraries and h5py, used by BOUT++ routines. Creates a consistent interface across machines

Supported libraries:

- \texttt{h5py} (for HDF5 files)
- \texttt{netCDF4} (preferred NetCDF library)

Note: NetCDF and HDF5 include unlimited dimensions, but this library is just for very simple I/O operations. Educated guesses are made for the dimensions.

class \texttt{boututils.datafile.DataFile}(filename=None, write=False, create=False, format='NETCDF3_64BIT', **kwargs)

File I/O class

A wrapper around various NetCDF libraries and h5py, used by BOUT++ routines. Creates a consistent interface across machines

**Parameters**

- \texttt{filename} (str, optional) – Name of file to open. If no filename supplied, you will need to call \texttt{open} and supply \texttt{filename} there
- \texttt{write} (bool, optional) – If True, open the file in read-write mode (existing files will be appended to). Default is read-only mode
- \texttt{create} (bool, optional) – If True, open the file in write mode (existing files will be truncated). Default is read-only mode
• `format (str, optional) – Name of a filetype to use (e.g. NETCDF3_CLASSIC, NETCDF3_64BIT, NETCDF4, HDF5)`

`attributes (varname)`  
Return a dictionary of attributes  

**Parameters**  
`varname (str) – The name of the variable`  

**Returns**  
The attribute names and their values  

**Return type**  
dict

`bout_type (varname)`  
Return the name of the BOUT++ type of a variable  

Possible values are:  
• scalar  
• Field2D  
• Field3D  

If the variable is an evolving variable (i.e. has a time dimension), then it is appended with a “.t”  

**Parameters**  
`varname (str) – The name of the variable`  

**Returns**  
The name of the BOUT++ type  

**Return type**  
str

`close ()`  
Close a file and flush data to disk

`dimensions (varname)`  
Return the names of all the dimensions of a variable  

**Parameters**  
`varname (str) – The name of the variable`  

**Returns**  
The names of the variable’s dimensions  

**Return type**  
tuple of str

`impl = None`  

`keys ()`  
A synonym for `list`

`list ()`  
List all variables in the file  

**Returns**  
A list containing all the names of the variables  

**Return type**  
list of str

`ndims (varname)`  
Return the number of dimensions for a variable  

**Parameters**  
`varname (str) – The name of the variable`  

**Returns**  
The number of dimensions  

**Return type**  
int

`open (filename, write=False, create=False, format='NETCDF3_CLASSIC')`  
Open the file  

**Parameters**
• **filename**(str, optional) – Name of file to open

• **write**(bool, optional) – If True, open the file in read-write mode (existing files will be appended to). Default is read-only mode

• **create**(bool, optional) – If True, open the file in write mode (existing files will be truncated). Default is read-only mode

• **format**(str, optional) – Name of a filetype to use (e.g. NETCDF3_CLASSIC, NETCDF4, HDF5)

### read(name, ranges=None, asBoutArray=True)

Read a variable from the file

**Parameters**

• **name**(str) – Name of the variable to read

• **ranges**(list of slice objects, optional) – Slices of variable to read, can also be converted from lists or tuples of (start, stop, stride). The number of elements in ranges should be equal to the number of dimensions of the variable you wish to read. See **size** for how to get the dimensions

• **asBoutArray**(bool, optional) – If True, return the variable as a **BoutArray** (the default)

**Returns**
The variable from the file (**BoutArray** if **asBoutArray** is True)

**Return type**
ndarray or **BoutArray**

### size(varname)

Return the size of each dimension of a variable

**Parameters**

• **varname**(str) – The name of the variable

**Returns**
The size of each dimension

**Return type**
tuple of int

### sync()

Write pending changes to disk.

### write(name, data, info=False)

Write a variable to file

If the variable is not a **BoutArray** with the **bout_type** attribute, a guess will be made for the dimensions

**Parameters**

• **name**(str) – Name of the variable to use in the file

• **data**(**BoutArray** or ndarray) – An array containing the variable data

• **info**(bool, optional) – If True, print information about what is being written to file

**Returns**

**Return type**
None

```python
class boututils.datafile.DataFile_HDF5(filename=None, write=False, create=False, format=None)
```

### attributes(varname)

Return a dictionary of attributes

**Parameters**

• **varname**(str) – The name of the variable
Returns: The attribute names and their values

Return type: dict

close()
Close a file and flush data to disk

dimensions(varname)
Return the names of all the dimensions of a variable

Parameters:
varname (str) – The name of the variable

Returns: The names of the variable’s dimensions

Return type: tuple of str

handle = None

keys()
A synonym for list

list()
List all variables in the file

Returns: A list containing all the names of the variables

Return type: list of str

ndims(varname)
Return the number of dimensions for a variable

Parameters:
varname (str) – The name of the variable

Returns: The number of dimensions

Return type: int

open(filename, write=False, create=False, format=None)
Open the file

Parameters:

- filename (str, optional) – Name of file to open
- write (bool, optional) – If True, open the file in read-write mode (existing files will be appended to). Default is read-only mode
- create (bool, optional) – If True, open the file in write mode (existing files will be truncated). Default is read-only mode
- format (str, optional) – Name of a filetype to use (e.g. NETCDF3_CLASSIC, NETCDF4, HDF5)

read(name, ranges=None, asBoutArray=True)
Read a variable from the file

Parameters:

- name (str) – Name of the variable to read
- ranges (list of slice objects, optional) – Slices of variable to read, can also be converted from lists or tuples of (start, stop, stride). The number of elements in ranges should be equal to the number of dimensions of the variable you wish to read. See size for how to get the dimensions
- asBoutArray (bool, optional) – If True, return the variable as a BoutArray (the default)
Returns  The variable from the file \((\text{BoutArray} \text{ if asBoutArray is True})\)

Return type  ndarray or \textit{BoutArray}

\section*{size (\texttt{varname})}
Return the size of each dimension of a variable

Parameters varname (\texttt{str}) – The name of the variable

Returns  The size of each dimension

Return type  tuple of int

\section*{sync ()}
Write pending changes to disk.

\section*{write (\texttt{name, data, info=False})}
Write a variable to file

If the variable is not a \textit{BoutArray} with the \textit{bout_type} attribute, a guess will be made for the dimensions

Parameters

- name (\texttt{str}) – Name of the variable to use in the file
- data (\textit{BoutArray} or \texttt{ndarray}) – An array containing the variable data
- info (\texttt{bool, optional}) – If True, print information about what is being written to file

Returns

Return type  None

\section*{class \texttt{boututils.datafile.DataFile_netCDF (filename=None, write=False, create=False, format='NETCDF3_CLASSIC', **kwargs)}}

\section*{attributes (\texttt{varname})}
Return a dictionary of attributes

Parameters varname (\texttt{str}) – The name of the variable

Returns  The attribute names and their values

Return type  dict

\section*{close ()}
Close a file and flush data to disk

\section*{dimensions (\texttt{varname})}
Return the names of all the dimensions of a variable

Parameters varname (\texttt{str}) – The name of the variable

Returns  The names of the variable’s dimensions

Return type  tuple of str

\section*{handle = None}

\section*{keys ()}
A synonym for \texttt{list}

\section*{list ()}
List all variables in the file

Returns  A list containing all the names of the variables
Return type list of str

ndims(varname)

Return the number of dimensions for a variable

Parameters varname (str) – The name of the variable

Returns The number of dimensions

Return type int

open(filename, write=False, create=False, format='NETCDF3_CLASSIC')

Open the file

Parameters

• filename (str, optional) – Name of file to open
• write (bool, optional) – If True, open the file in read-write mode (existing files will be appended to). Default is read-only mode
• create (bool, optional) – If True, open the file in write mode (existing files will be truncated). Default is read-only mode
• format (str, optional) – Name of a filetype to use (e.g. NETCDF3_CLASSIC, NETCDF4, HDF5)

read(name, ranges=None, asBoutArray=True)

Read a variable from the file.

size(varname)

Return the size of each dimension of a variable

Parameters varname (str) – The name of the variable

Returns The size of each dimension

Return type tuple of int

sync()

Write pending changes to disk.

write(name, data, info=False)

Write a variable to file

If the variable is not a BoutArray with the bout_type attribute, a guess will be made for the dimensions

Parameters

• name (str) – Name of the variable to use in the file
• data (BoutArray or ndarray) – An array containing the variable data
• info (bool, optional) – If True, print information about what is being written to file

Returns

Return type None

L.5.14 boututils.efit_analyzer module

L.5.15 boututils.fft_deriv module

boututils.fft_deriv.fft_deriv(var)
L.5.16 boututils.fft_integrate module

boututils.fft_integrate.fft_integrate(y, loop=None)
boututils.fft_integrate.test_integrate()

L.5.17 boututils.file_import module

Import an entire BOUT++ DataFile into memory

boututils.file_import.file_import(name)
Read all variables from file into a dictionary

Parameters name (str) – Name of file to read

Returns Dictionary containing all the variables in the file

Return type dict

L.5.18 boututils.geqdsk module

class boututils.geqdsk.Geqdsk

get(varname)
getAll()
getAllVars()
getDescriptor(varname)
openFile(filename)
open geqdsk file and parse its content

boututils.geqdsk.main()

L.5.19 boututils.idl_tabulate module

boututils.idl_tabulate.idl_tabulate(x, f, p=5)

L.5.20 boututils.int_func module

boututils.int_func.int_func(xin, fin=None, simple=None)

L.5.21 boututils.linear_regression module

boututils.linear_regression.linear_regression(x, y)
Simple linear regression of two variables

\[ y = a + bx \]

\( a, b = \text{linear_regression}(x, y) \)
L.5.22 boututils.local_min_max module

boututils.local_min_max.detect_local_maxima(arr)
Takes an array and detects the peaks using the local maximum filter. Returns a boolean mask of the troughs (i.e. 1 when the pixel’s value is the neighborhood maximum, 0 otherwise)

boututils.local_min_max.detect_local_minima(arr)
Takes an array and detects the troughs using the local maximum filter. Returns a boolean mask of the troughs (i.e. 1 when the pixel’s value is the neighborhood maximum, 0 otherwise)

L.5.23 boututils.mode_structure module

boututils.mode_structure.mode_structure(var_in, grid_in, period=1, zangle=0.0, n=None, addq=None, output=None, xq=None, xpsi=None, slow=None, subset=None, filter=None, famp=None, quiet=None, ergos=None, title=None, xrange=None, yrange=None, rational=None, pmodes=None, _extra=None)
boututils.mode_structure.zinterp(v, zind)

L.5.24 boututils.moment_xyzt module

boututils.moment_xyzt.RMSvalue(vec1d)
boututils.moment_xyzt.moment_xyzt(sig_xyzt, *args)

L.5.25 boututils.options module

Module to allow BOUT.inp files to be read into python and manipulated with ease.
Nick Walkden, June 2015 nick.walkden@ccfe.ac.uk
class boututils.options.BOUTOptions(inp_path=None)
Class to store and interact with options from BOUT++

Parameters
inp_path (str, optional) – Path to BOUT++ options file

Examples

Instantiate with

```python
>>> myOpts = BOUTOptions()
>>> myOpts.read_inp('path/to/input/file')
```

or

```python
>>> myOpts = BOUTOptions('path/to/input/file')
```

To get a list of sections use

```python
>>> section_list = myOpts.list_sections
>>> # Also print to screen:
>>> section_list = myOpts.list_sections(verbos=True)
```
Each section of the input is stored as a dictionary attribute so that, if you want all the settings in the section [ddx]:

```python
>>> ddx_opt_dict = myOpts.ddx
```

and access individual settings by

```python
>>> ddx_setting = myOpts.ddx['first']
```

Any settings in BOUT.inp without a section are stored in

```python
>>> root_dict = myOpts.root
```

### add_section

**add_section** *(section)*

Add a section to the options

**Parameters**

- **section** *(str)* – The name of a new section

### list_sections

**list_sections** *(verbose=False)*

Return all the sections in the options

**Parameters**

- **verbose** *(bool, optional)* – If True, print sections to screen

### read_inp

**read_inp** *(inp_path=“”)*

Read a BOUT++ input file

**Parameters**

- **inp_path** *(str, optional)* – Path to the input file (default: current directory)

### remove_section

**remove_section** *(section)*

Remove a section from the options

**Parameters**

- **section** *(str)* – The name of a section to remove

---

### L.5.26 boututils.plotdata module

**boututils.plotdata** *(data, x=None, y=None, title=None, xtitle=None, ytitle=None, output=None, range=None, fill=True, mono=False, colorbar=True, xerr=None, yerr=None)*

Plot 1D or 2D data, with a variety of options.

**boututils.plotdata.test()**

Test the plotdata routine.

---

### L.5.27 boututils.plotpolslice module

### L.5.28 boututils.radial_grid module

**boututils.radial_grid** *(n, pin, pout, include_in, include_out, seps, sep_factor, in_dp=None, out_dp=None)*

---

### L.5.29 boututils.read_geqdsk module

### L.5.30 boututils.run_wrapper module

Collection of functions which can be used to make a BOUT++ run
boututils.run_wrapper.build_and_log(test)
   Run make and redirect the output to a log file. Prints input
   
   On Windows, does nothing because executable should have already been built

boututils.run_wrapper.determineNumberOfCPUs()
   Number of virtual or physical CPUs on this system
   i.e. user/real as output by time(1) when called with an optimally scaling userspace-only program
   
   Taken from a post on stackoverflow: https://stackoverflow.com/questions/1006289/
   how-to-find-out-the-number-of-cpus-in-python

   Returns  The number of CPUs
   Return type  int

boututils.run_wrapper.getmpirun(default='mpirun -np')
   Return environment variable named MPIRUN, if it exists else return a default mpirun command

   Parameters  default (str, optional) – An mpirun command to return if MPIRUN is not set in the environment

boututils.run_wrapper.launch(command, runcmd=None, nproc=None, mthread=None, output=None, pipe=False, verbose=False)
   Launch parallel MPI jobs

   >>> status = launch(command, nproc, output=None)

   Parameters
   
   •  command (str) – The command to run
   •  runcmd (str, optional) – Command for running parallel job; defaults to what getmpirun() returns
   •  nproc (int, optional) – Number of processors (default: all available processors)
   •  mthread (int, optional) – Number of omp threads (default: the value of the OMP_NUM_THREADS environment variable)
   •  output (str, optional) – Name of file to save output to
   •  pipe (bool, optional) – If True, return the output of the command
   •  verbose (bool, optional) – Print the full command to be run before running it

   Returns  tuple – The return code, and either command output if pipe=True else None
   Return type  (int, str)

boututils.run_wrapper.launch_safe(command, *args, **kwargs)
   ‘Safe’ version of launch.

   Raises an RuntimeError exception if the command is not successful

   Parameters
   
   •  command (str) – The command to run
   •  *args, **kwargs – Optional arguments passed to shell

boututils.run_wrapper.shell(command, pipe=False)
   Run a shell command
**Parameters**

- **command** (*list of str*) – The command to run, split into (shell) words
- **pipe** (*bool, optional*) – Grab the output as text, else just run the command in the background

**Returns** tuple – The return code, and either command output if pipe=True else None

**Return type** (int, str)

`boututils.run_wrapper.shell_safe(command, *args, **kwargs)`  
‘Safe’ version of shell.

Raises a `RuntimeError` exception if the command is not successful

**Parameters**

- **command** (*str*) – The command to run
- ***args, **kwargs** – Optional arguments passed to `shell`

---

**L.5.31 boututils.showdata module**

Visualisation and animation routines

Written by Luke Easy le590@york.ac.uk Last Updated 19/3/2015 Additional functionality by George Breyiannis 26/12/2014

`boututils.showdata.showdata(var, titles=[], legendlabels=[], surf=[], polar=[], t_slice=0, t_array=None, movie=0, fps=28, dpi=200, intv=1, Ncolors=25, x=[], y=[], global_colors=False, symmetric_colors=False, hold_aspect=False, cmap=None, clear_between_frames=None, return_animation=False, window_title="")

A Function to animate time dependent data from BOUT++

To animate multiple variables on different axes:

```python
>>> showdata([var1, var2, var3])
```

To animate more than one line on a single axes:

```python
>>> showdata([[var1, var2, var3]])
```

The default graph types are: 2D (time + 1 spatial dimension) arrays = animated line plot 3D (time + 2 spatial dimensions) arrays = animated contour plot.

To use surface or polar plots:

```python
>>> showdata(var, surf=1)  
>>> showdata(var, polar=1)
```

Can plot different graph types on different axes. Default graph types will be used depending on the dimensions of the input arrays. To specify polar/surface plots on different axes:

```python
>>> showdata([var1, var2], surf=[1, 0], polar=[0, 1])
```

Movies require FFmpeg (for .mp4) and/or ImageMagick (for .gif) to be installed. The ‘movie’ option can be set to 1 (which will produce an mp4 called ‘animation.mp4’), to a name with no extension (which will produce an mp4 called ‘<name>.mp4’).
The `tslice` variable is used to control the time value that is printed on each frame of the animation. If the input data matches the time values found within BOUT++’s dmp data files, then these time values will be used. Otherwise, an integer counter is used.

The `cmap` variable (if specified) will set the colormap used in the plot. `cmap` must be a matplotlib colormap instance, or the name of a registered matplotlib colormap.

During animation, click once to stop in the current frame. Click again to continue.

**Parameters**

- `vars (array_like or list of array_like)` – Variable or list of variables to plot
- `titles (str or list of str, optional)` – Title or list of titles for each axis
- `legendlabels (str or list of str, optional)` – Legend or list of legends for each variable
- `surf (list of int)` – Which axes to plot as a surface plot
- `polar (list of int)` – Which axes to plot as a polar plot
- `tslice (list of int)` – Use these time values from a dump file (see above)
- `t_array (array)` – Pass in `t_array` using this argument to use the simulation time in plot titles. Otherwise, just use the t-index.
- `movie (int)` – If 1, save the animation to file
- `fps (int)` – Number of frames per second to use when saving animation
- `dpi (int)` – Dots per inch to use when saving animation
- `intv (int)` – ???
- `Ncolors (int)` – Number of levels in contour plots
- `x, y (array_like, list of array_like)` – X, Y coordinates
- `global_colors (bool)` – If “vars” is a list the colorlevels are determined from the maximum of the maxima and the minimum of the minima in all fields in vars
- `symmetric_colors (bool)` – Colour levels are symmetric
- `hold_aspect (bool)` – Use equal aspect ratio in plots
- `cmap (colormap)` – A matplotlib colormap instance to use
- `clear_between_frames (bool, optional)` –
  - Default (None) - all plots except line plots will clear between frames
  - True - all plots will clear between frames
  - False - no plots will clear between frames
- `return_animation (bool)` – Return the matplotlib animation instance
- `window_title (str)` – Give a title for the animation window

**L.5.32 boututils.spectrogram module**

Creates spectrograms using the Gabor transform to maintain time and frequency resolution

written by: Jarrod Leddy updated: 23/06/2016
boututils.spectrogram.spectrogram\( (\text{data}, \ dx, \ \sigma, \ \text{clip}=1.0, \ \text{optimise_clipping}=\text{True}, \ nskip=1.0) \)

Creates spectrograms using the Gabor transform to maintain time and frequency resolution

**Note:** Very early and very late times will have some issues due to the method - truncate them after taking the spectrogram if they are below your required standards

**Note:** If you are seeing issues at the top or bottom of the frequency range, you need a longer time series

written by: Jarrod Leddy updated: 23/06/2016

**Parameters**

- **data** \((\text{array_like})\) – The time series you want spectrogrammed
- **dt** \((\text{float})\) – Time resolution
- **sigma** \((\text{float})\) – Used in the Gabor transform, will balance time and frequency resolution suggested value is 1.0, but may need to be adjusted manually until result is as desired:
  - If bands are too tall raise sigma
  - If bands are too wide, lower sigma
- **clip** \((\text{float, optional})\) – Makes the spectrogram run faster, but decreases frequency resolution. clip is by what factor the time spectrum should be clipped by \(\rightarrow N_{\text{new}} = N / \text{clip} \)
- **optimise_clip** \((\text{bool})\) – If true (default) will change the data length to be \(2^N\) (rounded down from your inputted clip value) to make FFT’s fast
- **nskip** \((\text{float})\) – Scales final time axis, skipping points over which to centre the gaussian window for the FFTs

**Returns** \(\text{tuple}\) – A tuple containing the spectrogram, frequency and time

**Return type** \((\text{array_like}, \ \text{array_like}, \ \text{array_like})\)

boututils.spectrogram.test_spectrogram\( (n, \ d, \ s) \)

Function used to test the performance of spectrogram with various values of sigma

**Parameters**

- **n** \((\text{int})\) – Number of points
- **d** \((\text{float})\) – Grid spacing
- **s** \((\text{float})\) – Initial sigma

L.5.33 boututils.surface_average module

Average over a surface

boututils.surface_average.surface_average\( (\text{var}, \ \text{grid}, \ \text{area}=\text{None}) \)

Average a variable over a surface

**Parameters**

- **var** \((\text{array_like})\) – 3-D or 4D variable to integrate (either \([x,y,z]\) or \([t,x,y,z]\))
- **grid** \((\text{dict})\) – A dictionary of various grid quantities
• **area (bool)** – Average by flux-surface area = \((B/Bp) \times dl \times R \times dz\)

**Returns** Surface average of variable

**Return type** float

### L.5.34 boututils.volume_integral module

Integrate over a volume

`boututils.volume_integral.volume_integral(var, grid, xr=False)`

Integrate a variable over a volume

**Parameters**

- **var** (`array_like`) – Variable to integrate
- **grid** (`dict`) – A dictionary of various grid quantities
- **xr** (`(int, int)`, optional) – Range of x indices (default: all of x)

**Returns** Volume integral of variable

**Return type** float

### L.5.35 boututils.watch module

Routines for watching files for changes

`boututils.watch.watch(files, timeout=None, poll=2)`

Watch a given file or collection of files until one changes. Uses polling.

**Parameters**

- **files** (`str or list of str`) – Name of one or more files to watch
- **timeout** (`int`, optional) – Timeout in seconds (default is no timeout)
- **poll** (`int`, optional) – Polling interval in seconds (default: 2)

**Returns** The name of the first changed file, or None if timed out before any changes

**Return type** str

#### Examples

To watch one file, timing out after 60 seconds:

```
>>> watch('file1', timeout=60)
```

To watch 2 files, never timing out:

```
>>> watch(['file1', 'file2'])
```

**Author:** Ben Dudson <benjamin.dudson@york.ac.uk>

### L.5.36 Module contents

Generic routines, useful for all data
L.6 zoidberg package

L.6.1 Module contents

```python
zoidberg.make_maps(grid, magnetic_field, nslice=1, quiet=False, **kwargs)
```

Make the forward and backward FCI maps

**Parameters**

- `grid` *(zoidberg.grid.Grid)* – Grid generated by Zoidberg
- `magnetic_field` *(zoidberg.field.MagneticField)* – Zoidberg magnetic field object
- `nslice` *(int)* – Number of parallel slices in each direction
- `quiet` *(bool)* – Don’t display progress bar
- `kwargs` – Optional arguments for field line tracing, etc.

**Returns**

Dictionary containing the forward/backward field line maps

**Return type** `dict`

```python
zoidberg.write_maps(grid, magnetic_field, maps, gridfile='fci.grid.nc', new_names=False, metric2d=True, format='NETCDF3_64BIT', quiet=False)
```

Write FCI maps to BOUT++ grid file

**Parameters**

- `grid` *(zoidberg.grid.Grid)* – Grid generated by Zoidberg
- `magnetic_field` *(zoidberg.field.MagneticField)* – Zoidberg magnetic field object
- `maps` *(dict)* – Dictionary of FCI maps
- `gridfile` *(str, optional)* – Output filename
- `new_names` *(bool, optional)* – Write “g_yy” rather than “g_22”
- `metric2d` *(bool, optional)* – Output only 2D metrics
- `format` *(str, optional)* – Specifies file format to use, passed to boutdata.DataFile
- `quiet` *(bool, optional)* – Don’t warn about 2D metrics

**Returns**

**Return type** Writes the following variables to the grid file

L.6.2 Submodules

L.6.3 zoidberg.boundary module

Boundary objects that define an ‘outside’

```python
class zoidberg.boundary.NoBoundary
```

No boundary, so no points outside

```python
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

L.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) * 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 (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 \([\text{meters}]\)

Returns None if in Cartesian coordinates

**Parameters** \(x, z, \phi\) \((array\_like)\) – \(X, Z, \) and toroidal coordinates

**Returns** The major radius

**Return type** ndarray

class zoidberg.field.GEQDSK \((gfile)\)
Read a EFIT G-Eqdsk file for a toroidal equilibrium

This generates a grid in cylindrical geometry

**Parameters** \(gfile\) \((str)\) – Name of the file to open

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 \([\text{meters}]\)

Returns None if in Cartesian coordinates

**Parameters** \(x, z, \phi\) \((array\_like)\) – \(X, Z, \) and toroidal coordinates

**Returns** The major radius

**Return type** ndarray

pressure \((x, z, \phi)\)
Pressure \([\text{Pascals}]\)

**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
Return type  ndarray

\textbf{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

\textbf{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

\textbf{attributes} = {}

\textbf{boundary} = \texttt{<zoidberg.boundary.NoBoundary object>}

\textbf{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**

- \textbf{pos} (ndarray) – 2-D NumPy array, with the second dimension being [x,z], with x and z in meters
- \textbf{ycoord} (float) – Toroidal angle in radians if cylindrical coordinates, metres if Cartesian
- \textbf{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*\text{Bx}/Bphi, \text{R*Bz}/Bphi) if cylindrical
- (\text{Bx}/\text{By}, \text{Bz}/\text{By}) if Cartesian

**Return type**  list of floats or ndarray

\textbf{pressure}(x, z, \phi)

Pressure [Pascals]

**Parameters**  x, z, \phi (array_like) – X, Z, and toroidal coordinates

**Returns**  The plasma pressure

**Return type**  ndarray

\textbf{class}  \texttt{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**

- \textbf{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 (xa, za)
Linearly damp the field to be parallel to the edges of the box
Should take some parameters to adjust rate of smoothing, etc.

class zoidberg.field.StraightStellarator(*args, **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 Straight-Stellarator is ever used.

class zoidberg.field.VMEC (vmec_file, ntheta=None, nzeta=None, nr=32, nz=32)
A numerical magnetic field from a VMEC equilibrium file

Parameters

• vmec_file (str) – Name of the VMEC file to read
• ntheta (int, optional) – Number of theta points to use (default: use ‘mpol’ from VMEC file)
• nzeta (int, optional) – Number of zeta points to use (default: use ‘ntor’ from VMEC file)
• nr (int) – Number of R points to use
• 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

cfunc (field)
VMEC DCT

read_vmec_file (vmec_file, ntheta=None, nzeta=None)
Read a VMEC equilibrium file

sfunc (field)
VMEC DST
L.6.5 zoidberg.fieldtracer module

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 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

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:

\[ pos_{n+1} = pos_n + 0.5 \cdot (f(pos_n) + f(pos_{n+1})) \cdot dy \]

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`

```python
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`

### L.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**

Tuple of grid sizes (nx, ny, nz)

**Type** (int, int, int)

**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**

```python
>>> 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:

```python
>>> 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

**metric()**

Return the metric tensor, dx and dz

**Returns**

- **dict**

**numberOfPoloidalGrids()**

Returns the number of poloidal grids i.e. number of points in Y

**Returns**

- **int**

**zoidberg.grid.rectangular_grid(nx, ny, nz, Lx=1.0, Ly=10.0, Lz=1.0, xcentre=0.0, zcentre=0.0, yperiodic=False)**

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

### L.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**

- **grid** *(‘zoidberg.grid.Grid‘)* – Grid generated by Zoidberg
- **maps** *(dict)* – Dictionary containing the backward FCI maps
- **yslice** *(int, optional)* – Originating y-index to plot map from
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, yslice=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

L.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.polaroidal_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

ReturnType (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

ReturnType (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

ReturnType  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*

```python
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 in the domain \((0, 1)\) and \( z \) in \((0, 2\pi)\)

The above equations are inverted, giving:

\[
\begin{align*}
   a \cdot R_x x - 2 \cdot b \cdot R_x z + c \cdot R_z z &= 0 \\
   a \cdot Z_x x - 2 \cdot b \cdot Z_x z + c \cdot Z_z z &= 0
\end{align*}
\]

where

\[
\begin{align*}
   a &= R_z^2 + Z_z^2 \\
   b &= R_z \cdot R_x + Z_z \cdot Z_z \\
   c &= R_x^2 + Z_x^2
\end{align*}
\]

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

**References**

- [https://www.nada.kth.se/kurser/kth/2D1263/l2.pdf](https://www.nada.kth.se/kurser/kth/2D1263/l2.pdf)
L.6.9 zoidberg.progress module

zoidberg.progress.update_progress(progress, barLength=10, ascii=False, **kwargs)
Displays or updates a console progress bar

Accepts a float between 0 and 1. Any int will be converted to a float. A value under 0 represents a ‘halt’. A value at 1 or bigger represents 100%

Parameters
- progress (float) – Number between 0 and 1
- barLength (int, optional) – Length of the progress bar
- ascii (bool, optional) – If True, use ‘#’ as the progress indicator, otherwise use a Unicode character (the default)

L.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=theta, deriv=deriv)`

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, subdivde)`

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=sample)`

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 \((\text{theta}=\text{None})\)
Calculate the value of both \(R, Z\) at given theta locations

**Parameters**
- **\text{theta}** \((\text{array\_like}, \text{optional})\) – 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** \((\text{ndarray}, \text{ndarray})\)

position\text{Polygon} \((\text{theta}=\text{None})\)
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**
- **\text{theta}** \((\text{array\_like}, \text{optional})\) – 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** \((\text{ndarray}, \text{ndarray})\)

\text{zoidberg.rzline.circle} \((R0=1.0, r=0.5, n=20)\)
Creates a pair of RZline objects, for inner and outer boundaries

**Parameters**
- **\text{R0}** \((\text{float}, \text{optional})\) – Centre point of the circle
- **\text{r}** \((\text{float}, \text{optional})\) – Radius of the circle
- **\text{n}** \((\text{int}, \text{optional})\) – Number of points to use in the boundary

**Returns**
- A circular RZline

**Return type** \(\text{RZline}\)

\text{zoidberg.rzline.line\_from\_points} \((\text{rarray}, \text{zarray}, \text{show}=\text{False})\)
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**
- **\text{rarray}, \text{zarray}** \((\text{array\_like})\) – R, Z coordinates. These arrays should be the same length

**Returns**
- An RZline object representing a periodic line

**Return type** \(\text{RZline}\)

\text{zoidberg.rzline.line\_from\_points\_poly} \((\text{rarray}, \text{zarray}, \text{show}=\text{False})\)
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**
- **\text{rarray}, \text{zarray}** \((\text{array\_like})\) – R, Z coordinates. These arrays should be the same length

**Returns**
- An RZline object representing a periodic line

**Return type** \(\text{RZline}\)

\text{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**
- **\text{R0}** \((\text{float}, \text{optional})\) – Major radius
• \texttt{a (float, \textit{optional})} – Minor radius
• \texttt{elong (float, \textit{optional})} – Elongation, 0 for a circle
• \texttt{triang (float, \textit{optional})} – Triangularity, 0 for a circle
• \texttt{indent (float, \textit{optional})} – Indentation, 0 for a circle

\textbf{Returns} An \texttt{RZline} matching the given parameterisation

\textbf{Return type} \texttt{RZline}

**L.6.11 zoidberg.test_field module**

\texttt{zoidberg.test_field.test_curved_slab()}
\texttt{zoidberg.test_field.test_slab()}

**L.6.12 zoidberg.test_fieldtracer module**

\texttt{zoidberg.test_fieldtracer.test_FieldTracerReversible_slab()}
\texttt{zoidberg.test_fieldtracer.test_poincare()}
\texttt{zoidberg.test_fieldtracer.test_slab()}

**L.6.13 zoidberg.test_grid module**

\texttt{zoidberg.test_grid.test_getPoloidalGrid()}
\texttt{zoidberg.test_grid.test_getPoloidalGrid_periodic()}

**L.6.14 zoidberg.test_poloidal_grid module**

\texttt{zoidberg.test_poloidal_grid.test_out_of_domain()}

**L.6.15 zoidberg.test_rzline module**

\texttt{zoidberg.test_rzline.test_circular_boundaries()}
\texttt{zoidberg.test_rzline.test_distance()}
\texttt{zoidberg.test_rzline.test_line_from_points()}
\texttt{zoidberg.test_rzline.test_order_by_distance()}

**L.6.16 zoidberg.test_zoidberg module**

\texttt{zoidberg.test_zoidberg.test_make_maps_slab()}
\texttt{zoidberg.test_zoidberg.test_make_maps_straight_stellarator()}
**L.6.17 zoidberg.zoidberg module**

zoidberg.zoidberg.fci_to_vtk (infile, outfile, scale=5)

zoidberg.zoidberg.make_maps (grid, magnetic_field, nslice=1, quiet=False, **kwargs)

Make the forward and backward FCI maps

**Parameters**

- grid (zoidberg.grid.Grid) – Grid generated by Zoidberg
- magnetic_field (zoidberg.field.MagneticField) – Zoidberg magnetic field object
- nslice (int) – Number of parallel slices in each direction
- quiet (bool) – Don’t display progress bar
- kwargs – Optional arguments for field line tracing, etc.

**Returns** Dictionary containing the forward/backward field line maps

**Return type** dict

zoidberg.zoidberg.make_surfaces (grid, magnetic_field, nsurfaces=10, revs=100)

Essentially interpolate a poincare plot onto the grid mesh

**Parameters**

- grid (zoidberg.grid.Grid) – Grid generated by Zoidberg
- magnetic_field (zoidberg.field.MagneticField) – Zoidberg magnetic field object
- nsurfaces (int, optional) – Number of surfaces to interpolate to [10]
- revs (int, optional) – Number of points on each surface [100]

**Returns** Array of psuedo-psi on the grid mesh

**Return type** surfaces

zoidberg.zoidberg.parallel_slice_field_name (field, offset)

Form a unique, backwards-compatible name for field at a given offset

**Parameters**

- field (str) – Name of the field to convert
- offset (int) – Parallel slice offset

zoidberg.zoidberg.upscale (field, maps, upscale_factor=4, quiet=True)

Increase the resolution in y of field along the FCI maps.

First, interpolate onto the (forward) field line end points, as in normal FCI technique. Then interpolate between start and end points. We also need to interpolate the xt_primes and zt_primes. This gives a cloud of points along the field lines, which we can finally interpolate back onto a regular grid.

**Parameters**

- field (array_like) – 3D field to be upscaled
- maps (dict) – Zoidberg field line maps
- upscale_factor (int, optional) – Factor to increase resolution by [4]
- quiet (bool, optional) – Don’t show progress bar [True]
Returns

- Field with y-resolution increased \( \text{upscale\_factor} \) times. Shape is \((nx, \text{upscale\_factor} \times ny, nz)\).

\[
\text{zoidberg.zoidberg.write\_Bfield\_to\_vtk}(\text{grid}, \text{magnetic\_field}, \text{scale}=5, \text{vtkfile}=\text{fci\_zoidberg}, \text{psi}=\text{True})
\]

Write the magnetic field to a VTK file

Parameters

- \text{grid} (\text{zoidberg.grid.Grid}) – Grid generated by Zoidberg
- \text{magnetic\_field} (\text{zoidberg.field.MagneticField}) – Zoidberg magnetic field object
- \text{scale} (\text{int}, \text{optional}) – Factor to scale x, z dimensions by [5]
- \text{vtkfile} (\text{str}, \text{optional}) – Output filename without extension [“fci\_zoidberg”]
- \text{psi} (\text{bool}, \text{optional}) – Write psi?

Returns

Return type path - Full path to vtkfile

\[
\text{zoidberg.zoidberg.write\_maps}(\text{grid}, \text{magnetic\_field}, \text{maps}, \text{gridfile}=\text{fci.grid.nc}, \text{new\_names}=\text{False}, \text{metric2d}=\text{True}, \text{format}=\text{’NETCDF3_64BIT’}, \text{quiet}=\text{False})
\]

Write FCI maps to BOUT++ grid file

Parameters

- \text{grid} (\text{zoidberg.grid.Grid}) – Grid generated by Zoidberg
- \text{magnetic\_field} (\text{zoidberg.field.MagneticField}) – Zoidberg magnetic field object
- \text{maps} (\text{dict}) – Dictionary of FCI maps
- \text{gridfile} (\text{str}, \text{optional}) – Output filename
- \text{new\_names} (\text{bool}, \text{optional}) – Write “g\_yy” rather than “g\_22”
- \text{metric2d} (\text{bool}, \text{optional}) – Output only 2D metrics
- \text{format} (\text{str}, \text{optional}) – Specifies file format to use, passed to boutdata.DataFile
- \text{quiet} (\text{bool}, \text{optional}) – Don’t warn about 2D metrics

Returns

Return type Writes the following variables to the grid file

L.6.18 Module contents

\[
\text{zoidberg.make\_maps}(\text{grid}, \text{magnetic\_field}, \text{nslice}=1, \text{quiet}=\text{False}, **\text{kwargs})
\]

Make the forward and backward FCI maps

Parameters

- \text{grid} (\text{zoidberg.grid.Grid}) – Grid generated by Zoidberg
- \text{magnetic\_field} (\text{zoidberg.field.MagneticField}) – Zoidberg magnetic field object
• **nslice** *(int)* – Number of parallel slices in each direction

• **quiet** *(bool)* – Don’t display progress bar

• **kwargs** – Optional arguments for field line tracing, etc.

**Returns** Dictionary containing the forward/backward field line maps

**Return type** `dict`

```python
zoidberg.write_maps(grid, magnetic_field, maps, gridfile='fci.grid.nc', new_names=False, metric2d=True, format='NETCDF3_64BIT', quiet=False)
```

Write FCI maps to BOUT++ grid file

**Parameters**

• **grid** *(zoidberg.grid.Grid)* – Grid generated by Zoidberg

• **magnetic_field** *(zoidberg.field.MagneticField)* – Zoidberg magnetic field object

• **maps** *(dict)* – Dictionary of FCI maps

• **gridfile** *(str, optional)* – Output filename

• **new_names** *(bool, optional)* – Write “g_yy” rather than “g_22”

• **metric2d** *(bool, optional)* – Output only 2D metrics

• **format** *(str, optional)* – Specifies file format to use, passed to boutdata.DataFile

• **quiet** *(bool, optional)* – Don’t warn about 2D metrics

**Returns**

**Return type** Writes the following variables to the grid file
Indices and tables

- genindex
- search
[Dudson2009] https://doi.org/10.1016/j.cpc.2009.03.008


b
boutcore, 132
boutdata, 711
boutdata.cbdtoeqdsk, 714
boutdata.collect, 714
boutdata.data, 716
boutdata.gen_surface, 720
boutdata.griddata, 721
boutdata.input, 722
boutdata.pol_slice, 723
boutdata.processor_rearrange, 723
boutdata.restart, 724
boutdata.settings, 729
boutdata.shiftz, 729
boututils, 711
boututils.ask, 731
boututils.boutarray, 731
boututils.boutwarnings, 732
boututils.calculus, 732
boututils.closest_line, 733
boututils.contour, 733
boututils.crosslines, 733
boututils.datafile, 734
boututils.fft_deriv, 739
boututils.fft_integrate, 740
boututils.file_import, 740
boututils.geqdsk, 740
boututils.idl_tabulate, 740
boututils.int_func, 740
boututils.linear_regression, 740
boututils.local_min_max, 741
boututils.mode_structure, 741
boututils.moment_xyzt, 741
boututils.options, 741
boututils.plotdata, 742
boututils.radial_grid, 742
boututils.run_wrapper, 742
boututils.showdata, 744
boututils.spectrogram, 745
boututils.surface_average, 746
boututils.volume_integral, 747
boututils.watch, 747

Z
zoidberg, 769
zoidberg.boundary, 748
zoidberg.field, 749
zoidberg.fieldtracer, 755
zoidberg.grid, 756
zoidberg.plot, 758
zoidberg.poloidal_grid, 759
zoidberg.progress, 764
zoidberg.rzline, 764
zoidberg.test_field, 767
zoidberg.test_fieldtracer, 767
zoidberg.test_grid, 767
zoidberg.test_poloidal_grid, 767
zoidberg.test_rzline, 767
zoidberg.test_zoidberg, 767
zoidberg.zoidberg, 768
Symbols

_(C macro), 469
__GET_FOR_EACH_EXPANSION (C macro), 528
__BOUT_GENERIC_FACTORY_H__ (C macro), 466
__FIELD2D_H__ (C macro), 416
__FIELD3D_H__ (C macro), 422
__FIELD_DATA_H__ (C macro), 438
__FUNCT__ (C macro), 485, 486, 521, 601, 618
__OPTIONS_H__ (C macro), 572
__OPTIONS_NETCDF_H__ (C macro), 581
__OUTPUT_H__ (C macro), 583
__RVEC_H__ (C macro), 648
__VARIANT_HXX__ (C macro), 696
__VECTOR2D_H__ (C macro), 702
__VECTOR3D_H__ (C macro), 706
__thefunc__ (C macro), 549
_ecexpand_1 (C macro), 320
_ecexpand_10 (C macro), 320
_ecexpand_2 (C macro), 320
_ecexpand_3 (C macro), 320
_ecexpand_4 (C macro), 320
_ecexpand_5 (C macro), 320
_ecexpand_6 (C macro), 320
_ecexpand_7 (C macro), 320
_ecexpand_8 (C macro), 320
_ecexpand_9 (C macro), 320
_\text{fe\_1} (C macro), 527
_\text{fe\_10} (C macro), 528
_\text{fe\_2} (C macro), 527
_\text{fe\_3} (C macro), 527
_\text{fe\_4} (C macro), 527
_\text{fe\_5} (C macro), 527
_\text{fe\_6} (C macro), 528
_\text{fe\_7} (C macro), 528
_\text{fe\_8} (C macro), 528
_\text{fe\_9} (C macro), 528
_\text{me\_1} (C macro), 527
_\text{me\_10} (C macro), 527
_\text{me\_2} (C macro), 527
_\text{me\_3} (C macro), 527
_\text{me\_4} (C macro), 527
_\text{me\_5} (C macro), 527
_\text{me\_6} (C macro), 527
_\text{me\_7} (C macro), 527
_\text{me\_8} (C macro), 527
_\text{me\_9} (C macro), 527
[anonymous] (C++ enum), 676

A

abs (C++ function), 433, 702, 703, 706, 707
action (gen_fieldops attribute), 461
AdamsBashforthSolver (C++ class), 289
AdamsBashforthSolver::AdamsBashforthSolver (C++ function), 289
AdamsBashforthSolver::AdamsBashforthSolver (C++ function), 289
AdamsBashforthSolver::AdamsBashforthSolver (C++ function), 289
AdamsBashforthSolver::adaptive (C++ member), 290
AdamsBashforthSolver::adaptive_order (C++ member), 290
AdamsBashforthSolver::atol (C++ member), 290
AdamsBashforthSolver::current_order (C++ member), 290
AdamsBashforthSolver::dtFac (C++ member), 290
AdamsBashforthSolver::followHighOrder (C++ member), 290
AdamsBashforthSolver::getCurrentTimestep (C++ function), 289
AdamsBashforthSolver::history (C++ member), 290
AdamsBashforthSolver::init (C++ function), 289
AdamsBashforthSolver::max_timestep (C++ member), 290
AdamsBashforthSolver::maximum_order (C++ member), 290

777
AdamsBashforthSolver::mxstep (C++ member), 290
AdamsBashforthSolver::neq (C++ member), 290
AdamsBashforthSolver::nextState (C++ member), 290
AdamsBashforthSolver::nlocal (C++ member), 290
AdamsBashforthSolver::nsteps (C++ member), 290
AdamsBashforthSolver::out_timestep (C++ member), 290
AdamsBashforthSolver::resetInternalField (C++ function), 289
AdamsBashforthSolver::rtol (C++ member), 290
AdamsBashforthSolver::run (C++ function), 289
AdamsBashforthSolver::setMaxTimestep (C++ function), 289
AdamsBashforthSolver::state (C++ member), 290
AdamsBashforthSolver::take_step (C++ function), 290
AdamsBashforthSolver::times (C++ member), 290
AdamsBashforthSolver::timestep (C++ member), 290
Add (C++ class), 405
add (C++ function), 401
add_section (C++ function), 405
addItemToDeferredFunction (C++ function), 681
addnoise () (in module boutdata.restart), 724
addvar () (in module boutdata.restart), 724
advanceStepWrapper (C++ function), 653
Aligned (C++ enumerator), 323
all (C++ enumerator), 301, 322
alwayswarn () (in module boututils.boutwarnings), 724
AlwaysWarning, 732
animate () (zoidberg.plot.AnimateVectorField method), 758
AnimateVectorField (class in zoidberg.plot), 758
arakawa (C++ enumerator), 394
arakawa_old (C++ enumerator), 394
areDirectionsCompatible (C++ function), 321, 324
areFieldsCompatible (C++ function), 432
arkode_bbd_rhs (C++ function), 291
arkode_jac (C++ function), 291
arkode_pre (C++ function), 291
arkode_pre_shim (C++ function), 291
arkode_rhs (C++ function), 291
arkode_rhs_explicit (C++ function), 291
arkode_rhs_implicit (C++ function), 291
ARKODEINT (C++ type), 290
ArkodeSolver (C++ class), 292
ArkodeSolver::ArkodeSolver (C++ function), 292
ArkodeSolver::arkode_mem (C++ member), 293
ArkodeSolver::ArkodeSolver (C++ function), 292
ArkodeSolver::diagnose (C++ member), 293
ArkodeSolver::NOUT (C++ member), 293
ArkodeSolver::pre (C++ function), 292
ArkodeSolver::pre_ncalls (C++ member), 293
ArkodeSolver::pre_Wtime (C++ member), 293
ArkodeSolver::rhs (C++ function), 292
ArkodeSolver::rhs_e (C++ function), 292
ArkodeSolver::rhs_i (C++ function), 292
ArkodeSolver::run (C++ function), 292
ArkodeSolver::set_abstol_values (C++ function), 293
ArkodeSolver::setJacobian (C++ function), 292
ArkodeSolver::TIMESTEP (C++ member), 293
ArkodeSolver::uvec (C++ member), 293
ARKStepSetFixedStep (C++ function), 291
ARKStepSetFixedPoint (C++ member), 291
ARKStepSetExplicit (C++ member), 291
ARKStepSetFixedStep (C++ member), 291
ARKStepSetAdaptivityMethod (C++ member), 291
ARKStepGetDky (C++ function), 291
ARKStepGetCurrentTime (C++ member), 291
ARKStepGetNumRhsEvals (C++ function), 291
ARKStepGetNumSteps (C++ function), 291
ARKStepGetNumEvals (C++ member), 291
ARKStepGetNumRhsEvals (C++ member), 291
ARKStepGetLastStep (C++ member), 291
ARKStepGetNumLinIters (C++ member), 291
ARKStepGetNonlinSolvIters (C++ member), 291
ARKStepGetNumPrecEvals (C++ member), 291
ARKStepGetPrecEvals (C++ function), 291
ARKStepGetNumLinPrecEvals (C++ member), 291
ARKStepGetLinPrecEvals (C++ member), 291
ARKStepGetNonlinSolvIters (C++ member), 291
ARKStepSetCFLFraction (C++ member), 291
ARKStepGetNumLinVts (C++ member), 291
BoundaryDirichlet::apply
BoundaryDirichlet
BoundaryConstLaplace::clone
BoundaryConstLaplace::BoundaryConstLaplace
BoundaryDirichlet::gen
BoundaryDirichlet::clone
BoundaryDirichlet::BoundaryDirichlet
BoundaryDirichlet::apply_ddt
BoundaryConstLaplace::apply
BoundaryConstGradient::clone
BoundaryConstGradient::BoundaryConstGradient
BoundaryConstGradient::apply
BoundaryConstGradient::apply
BoundaryFactory::instance
BoundaryFactory::getInstance
BoundaryFactory::findBoundaryOpPar
BoundaryFactory::findBoundaryOp
BoundaryFactory::findBoundaryMod
BoundaryFactory::createFromOptions
BoundaryFactory::create
BoundaryFactory::~BoundaryFactory
BoundaryDivCurl::clone
BoundaryDivCurl::BoundaryDivCurl
BoundaryDivCurl::apply
BoundaryDivCurl::apply
BoundaryDivCurl::apply
BoundaryDivCurl::BoundaryDivCurl
BoundaryDirichlet_O4::gen
BoundaryDirichlet_O4::clone
BoundaryDirichlet_O4::BoundaryDirichlet_O4
BoundaryDirichlet_O4::apply_ddt
BoundaryDirichlet_O3::gen
BoundaryDirichlet_O3::clone
BoundaryDirichlet_O3::BoundaryDirichlet_O3
BoundaryDirichlet_O3::apply_ddt
BoundaryDirichlet_O3::apply
BoundaryDirichlet_O3
BoundaryDirichlet_4thOrder::val
BoundaryDirichlet_4thOrder::clone
BoundaryDirichlet_4thOrder::BoundaryDirichlet_4thOrder
BoundaryDirichlet_4thOrder::apply_ddt
BoundaryDirichlet_2ndOrder::apply
BoundaryDirichlet_2ndOrder
BoundaryDirichlet_2ndOrder::val
BoundaryDirichlet_2ndOrder::clone
BoundaryDivCurl
BoundaryDivCurl::apply
BoundaryDivCurl::BoundaryDivCurl
BoundaryFactory
BoundaryFactory::BoundaryFactory
BoundaryFactory::addMod
BoundaryFactory::add
BoundaryFactory::~BoundaryFactory
BoundaryFactory::findBoundaryMod
BoundaryFactory::findBoundaryOp
BoundaryFactory::findBoundaryOpPar
BoundaryFactory::getInstance
BoundaryFactory::instance
BoundaryFactory::modmap (C++ member), 299
BoundaryFactory::opmap (C++ member), 299
BoundaryFactory::par_opmap (C++ member), 299
BoundaryFree (C++ class), 313
BoundaryFree::apply (C++ function), 313
BoundaryFree::apply_dtt (C++ function), 313
BoundaryFree::BoundaryFree (C++ function), 313
BoundaryFree::clone (C++ function), 313
BoundaryFree::val (C++ member), 313
BoundaryFree_O2 (C++ class), 313
BoundaryFree_O2::BoundaryFree_O2 (C++ function), 313
BoundaryFree_O2::apply (C++ function), 313
BoundaryFree_O2::BoundaryFree_O2 (C++ function), 313
BoundaryFree_O2::apply (C++ function), 313
BoundaryFree_O2::apply_dtt (C++ function), 313
BoundaryFree_O2::BoundaryFree_O2 (C++ function), 313
BoundaryFree_O2::apply (C++ function), 313
BoundaryFree_O3 (C++ class), 314
BoundaryFree_O3::BoundaryFree_O3 (C++ function), 314
BoundaryFree_O3::apply (C++ function), 314
BoundaryFree_O3::apply_dtt (C++ function), 314
BoundaryFromFieldAligned (C++ class), 316
BoundaryFromFieldAligned::apply (C++ function), 316
BoundaryFromFieldAligned::apply_dtt (C++ function), 316
BoundaryNeumann (C++ class), 309
BoundaryNeumann (C++ class), 308
BoundaryNeumann (C++ class), 309
BoundaryNeumann_2ndOrder (C++ class), 308
BoundaryNeumann_2ndOrder::apply (C++ function), 309
BoundaryNeumann_2ndOrder::apply_dtt (C++ function), 309
BoundaryNeumann_2ndOrder::BoundaryNeumann_2ndOrder (C++ function), 309
BoundaryNeumann_2ndOrder::clone (C++ function), 309
BoundaryNeumann_2ndOrder::val (C++ member), 309
BoundaryNeumann_4thOrder (C++ class), 310
BoundaryNeumann_4thOrder::apply (C++ function), 310
BoundaryNeumann_4thOrder::apply_dtt (C++ function), 310
BoundaryNeumann_4thOrder::BoundaryNeumann_4thOrder (C++ function), 310
BoundaryNeumann_4thOrder::clone (C++ function), 310
BoundaryNeumann_4thOrder::val (C++ member), 310
BoundaryNeumann_NonOrthogonal (C++ class), 308
BoundaryNeumann_NonOrthogonal::apply (C++ function), 308
BoundaryNeumann_NonOrthogonal::BoundaryNeumann_NonOrthogonal (C++ function), 308
BoundaryNeumann_NonOrthogonal::clone (C++ function), 308
BoundaryNeumann_NonOrthogonal::val (C++ member), 308
BoundaryNeumann_04 (C++ class), 310
BoundaryNeumann_04::apply (C++ function), 310
BoundaryNeumann_04::apply_dtt (C++ function), 310
BoundaryNeumann_04::BoundaryNeumann_04 (C++ function), 310
BoundaryNeumann_04::clone (C++ function), 310
BoundaryOp (C++ class), 299
BoundaryOp::BoundaryOp (C++ function), 300
BoundaryOp::apply_dtt (C++ function), 300
BoundaryOp::apply_to_dtt (C++ member), 300
BoundaryOp::bndry (C++ member), 300
BoundaryOp::BoundaryOp (C++ function), 300
BoundaryOp::clone (C++ function), 300
BoundaryOpBase (C++ class), 299
BoundaryOpBase::~BoundaryOpBase (C++ function), 299
BoundaryOpBase::apply (C++ function), 299
BoundaryOpBase::BoundaryOpBase (C++ function), 299
BoundaryOpPar (C++ class), 587
BoundaryOpPar::~BoundaryOpPar (C++ function), 587
BoundaryOpPar::apply (C++ function), 587
BoundaryOpPar::bndry (C++ member), 588
BoundaryOpPar::BoundaryOpPar (C++ function), 587
BoundaryOpPar::clone (C++ function), 587
BoundaryOpPar::field_values (C++ member), 588
BoundaryOpPar::gen_values (C++ member), 588
BoundaryOpPar::getValue (C++ function), 588
BoundaryOpPar::real_value (C++ member), 588
BoundaryOpPar::value_type (C++ member), 588
BoundaryOpPar_dirichlet (C++ class), 588
BoundaryOpPar_dirichlet::apply (C++ function), 588
BoundaryOpPar_dirichlet::BoundaryOpPar_dirichlet (C++ function), 588
BoundaryOpPar_dirichlet::clone (C++ function), 588
BoundaryOpPar_dirichlet_inter::apply (C++ function), 589
BoundaryOpPar_dirichlet_inter::BoundaryOpPar_dirichlet_inter (C++ function), 589
BoundaryOpPar_dirichlet_inter::clone (C++ function), 589
BoundaryOpPar_dirichlet_inter_03 (C++ class), 588
BoundaryOpPar_dirichlet_inter_03::apply (C++ function), 589
BoundaryOpPar_dirichlet_inter_03::BoundaryOpPar_dirichlet_inter_03 (C++ function), 589
BoundaryOpPar_dirichlet_inter_03::clone (C++ function), 589
BoundaryOpPar_neumann (C++ class), 589
BoundaryOpPar_neumann::apply (C++ function), 590
BoundaryOpPar_neumann::BoundaryOpPar_neumann (C++ function), 589
BoundaryOpPar_neumann::clone (C++ function), 589
BoundaryRegion (C++ class), 302
BoundaryRegion::~BoundaryRegion (C++ function), 302
BoundaryRegion::BoundaryRegion (C++ function), 302
BoundaryRegion::bx (C++ member), 302
BoundaryRegion::by (C++ member), 302
BoundaryRegion::next1d (C++ function), 302
BoundaryRegion::nextX (C++ function), 302
BoundaryRegion::nextY (C++ function), 302
BoundaryRegion::width (C++ member), 302
BoundaryRegion::x (C++ member), 302
BoundaryRegion::y (C++ member), 302
BoundaryRegionBase (C++ class), 301
BoundaryRegionBase::~BoundaryRegionBase (C++ function), 301
BoundaryRegionBase::BoundaryRegionBase (C++ function), 301
BoundaryRegionBase::first (C++ function), 301
BoundaryRegionBase::isDone (C++ function), 301
BoundaryRegionBase::isParallel (C++ member), 302
BoundaryRegionBase::label (C++ member), 302
BoundaryRegionBase::localmesh (C++ member), 302
BoundaryRegionBase::location (C++ member), 302
BoundaryRegionBase::next (C++ function), 301
BoundaryRegionPar (C++ class), 590
BoundaryRegionPar::add_point (C++ function), 590
BoundaryRegionPar::angle (C++ member), 590
BoundaryRegionPar::bdry_points (C++ function), 591
BoundaryRegionPar::bdry_position (C++ member), 591
BoundaryRegionPar::BoundaryRegionPar (C++ function), 590
BoundaryRegionPar::dir (C++ member), 590
BoundaryRegionPar::first (C++ function), 590
BoundaryRegionPar::IndexPoint (C++ class), 591
BoundaryRegionPar::IndexPoint::jx (C++ member), 591
BoundaryRegionPar::IndexPoint::jy (C++ member), 591
BoundaryRegionPar::IndexPoint::jz (C++ member), 591
BoundaryRegionPar::Indices (C++ class), 591
BoundaryRegionPar::Indices::angle (C++ function), 591
BoundaryRegionPar::Indices::index (C++ member), 591
BoundaryRegionPar::Indices::intersection (C++ member), 591
BoundaryRegionPar::Indices::length (C++ member), 591
BoundaryRegionPar::IndicesIter (C++ type), 591
BoundaryRegionPar::IndicesVec (C++ type), 591
BoundaryRegionPar::isDone (C++ function), 591
BoundaryRegionPar::length (C++ member), 590
BoundaryRegionPar::next (C++ function), 590
BoundaryRegionPar::RealPoint (C++ class), 591
BoundaryRegionPar::RealPoint::s_x (C++ member), 591
BoundaryRegionPar::RealPoint::s_y (C++ member), 591
BoundaryRegionPar::RealPoint::s_z (C++ member), 590
BoundaryRegionPar::s_x (C++ member), 590
BoundaryRegionPar::s_y (C++ member), 590
BoundaryRegionPar::s_z (C++ member), 590
BoundaryRegionPar::x (C++ member), 590
BoundaryRegionPar::y (C++ member), 590
BoundaryRegionPar::z (C++ member), 590
BoundaryRegionXIn::BoundaryRegionXIn (C++ function), 302
BoundaryRegionXIn::first (C++ function), 303
BoundaryRegionXIn::isDone (C++ function), 303
BoundaryRegionXIn::next (C++ function), 303
BoundaryRegionXIn::nextX (C++ function), 303
BoundaryRegionXIn::nextY (C++ function), 303
BoundaryRegionXIn::ye (C++ member), 304
BoundaryRegionXIn::ys (C++ member), 304
BoundaryRegionYDown (C++ class), 304
BoundaryRegionYDown::BoundaryRegionYDown (C++ function), 304
BoundaryRegionYDown::first (C++ function), 304
BoundaryRegionYDown::isDone (C++ function), 304
BoundaryRegionYDown::next (C++ function), 304
BoundaryRegionYDown::next1d (C++ function), 304
BoundaryRegionYDown::nextX (C++ function), 304
BoundaryRegionYDown::nextY (C++ function), 304
BoundaryRegionYDown::xe (C++ member), 304
BoundaryRegionYDown::xs (C++ member), 304
BoundaryRegionYUp (C++ class), 304
BoundaryRegionYUp::BoundaryRegionYUp (C++ function), 304
BoundaryRegionYUp::first (C++ function), 304
BoundaryRegionYUp::isDone (C++ function), 304
BoundaryRegionYUp::next (C++ function), 304
BoundaryRegionYUp::next1d (C++ function), 304
BoundaryRegionYUp::nextX (C++ function), 304
BoundaryRegionYUp::nextY (C++ function), 304
BoundaryRegionYUp::xs (C++ member), 304
BoundaryRegionYUp::ye (C++ member), 304
BoundaryRegionXOut::nextX (C++ function), 303
BoundaryRegionXOut::nextY (C++ function), 303
BoundaryRegionXOut::ye (C++ member), 304
BoundaryRegionXOut::ys (C++ member), 304
BoundaryRegionYDown::BoundaryRegionYDown (C++ function), 304
BoundaryRegionYDown::first (C++ function), 304
BoundaryRegionYDown::isDone (C++ function), 304
BoundaryRegionYDown::next (C++ function), 304
BoundaryRegionYDown::next1d (C++ function), 304
BoundaryRegionYDown::nextX (C++ function), 304
BoundaryRegionYDown::nextY (C++ function), 304
BoundaryRegionYDown::xe (C++ member), 304
BoundaryRegionYDown::xs (C++ member), 304
BoundaryRegionYUp (C++ class), 304
BoundaryRegionYUp::BoundaryRegionYUp (C++ function), 304
BoundaryRegionYUp::first (C++ function), 304
BoundaryRegionYUp::isDone (C++ function), 304
BoundaryRegionYUp::next (C++ function), 304
BoundaryRegionYUp::next1d (C++ function), 304
BoundaryRegionYUp::nextX (C++ function), 304
BoundaryRegionYUp::nextY (C++ function), 304
BoundaryRegionYUp::xs (C++ member), 304
BoundaryRegionYUp::ye (C++ member), 304
BoundaryRegionXOut::BoundaryRegionXOut (C++ function), 303
BoundaryRegionXOut::first (C++ function), 303
BoundaryRegionXOut::isDone (C++ function), 303
BoundaryRegionXOut::next (C++ function), 303
BoundaryRegionXOut::next1d (C++ function), 303
BoundaryRegionXOut::nextX (C++ function), 303
BoundaryToFieldAligned::apply_ddt (C++ function), 315
BoundaryToFieldAligned::BoundaryToFieldAligned (C++ function), 315
BoundaryToFieldAligned::cloneMod (C++ function), 315
BoundaryWidth (C++ class), 315
BoundaryWidth::width (C++ member), 315
BoundaryWidth::apply (C++ function), 315
BoundaryWidth::apply_ddt (C++ function), 315
BoundaryWidth::BoundaryWidth (C++ function), 315
BoundaryWidth::cloneMod (C++ function), 315
BoundaryZeroLaplace (C++ class), 312
BoundaryZeroLaplace2 (C++ class), 312
BoundaryZeroLaplace2::apply (C++ function), 312
BoundaryZeroLaplace2::BoundaryZeroLaplace2 (C++ function), 312
BoundaryZeroLaplace2::clone (C++ function), 312
BoundaryZeroLaplace::apply (C++ function), 312
BoundaryZeroLaplace::BoundaryZeroLaplace (C++ function), 312
BoundaryZeroLaplace::clone (C++ function), 312
bout (C++ type), 305, 317, 318, 339, 413, 414, 437, 464, 474, 496, 516, 581, 660, 685, 686, 695, 696
bout2sonnet () (in module boutdata.griddata), 721
bout::build (C++ type), 339
bout::build::check_level (C++ member), 339
bout::build::has_fft (C++ member), 339
bout::build::has_fftw (C++ member), 339
bout::build::hasftware (C++ member), 339
bout::build::has_lapack (C++ member), 340
bout::build::has_netcdf (C++ member), 340
bout::build::has_petsc (C++ member), 340
bout::build::has_pretty_function (C++ member), 340
bout::build::has_pvode (C++ member), 340
bout::build::has_scorer (C++ member), 340
bout::build::has_slepc (C++ member), 340
bout::build::has_sundials (C++ member), 340
bout::build::openmp_schedule (C++ member), 339
bout::build::use_backtrace (C++ member), 340
bout::build::use_color (C++ member), 340
bout::build::use_openmp (C++ member), 340
bout::build::use_output_debug (C++ member), 340
bout::build::use_sgfpe (C++ member), 340
bout::build::use_signal (C++ member), 340
bout::build::checkFinite (C++ function), 438
bout::checkPositive (C++ function), 438
bout::derivatives (C++ type), 496
bout::derivatives::index (C++ type), 496
bout::derivatives::index::D2DX2 (C++ function), 496
bout::derivatives::index::D2DY2 (C++ function), 496
bout::derivatives::index::D2DZ2 (C++ function), 496
bout::derivatives::index::D4DX4 (C++ function), 496
bout::derivatives::index::D4DY4 (C++ function), 496
bout::derivatives::index::D4DZ4 (C++ function), 497
bout::derivatives::index::DDX (C++ function), 496
bout::derivatives::index::DDY (C++ function), 496
bout::derivatives::index::DDZ (C++ function), 496
bout::derivatives::index::F2DX (C++ function), 497
bout::derivatives::index::F2DY (C++ function), 497
bout::derivatives::index::F2DZ (C++ function), 497
bout::derivatives::index::flowDerivative (C++ function), 496
bout::derivatives::index::standardDerivative (C++ function), 496
bout::derivatives::index::VDDX (C++ function), 497
bout::derivatives::index::VDDY (C++ function), 497
bout::derivatives::index::VDDZ (C++ function), 497
bout::derivatives::index::V3DX (C++ function), 497
bout::derivatives::index::V3DY (C++ function), 497
bout::derivatives::index::V3DZ (C++ function), 497
bout::experimental (C++ type), 318, 581
bout::experimental::append (C++ enumerator), 581
bout::experimental::checkDataDirectoryIsAccessible (C++ function), 319
bout::experimental::CommandLineArgs (C++ class), 320
bout::experimental::CommandLineArgs::color_output (C++ member), 320
bout::experimental::CommandLineArgs::data_dir (C++ member), 320
bout::experimental::CommandLineArgs::log_file (C++ member), 320
bout::experimental::CommandLineArgs::opt_file (C++ member), 320
bout::experimental::CommandLineArgs::original_argv (C++ member), 320
bout::experimental::CommandLineArgs::set_file (C++ member), 320
bout::experimental::CommandLineArgs::verbosity (C++ member), 320
bout::experimental::defaultSignalHandler (C++ member), 320
bout::experimental:: FileMode (C++ enum), 581
bout::experimental::OptionsNetCDF (C++ class), 581
bout::experimental::OptionsNetCDF::OptionsNetCDF (C++ function), 581
bout::experimental::OptionsNetCDF::read (C++ function), 581
bout::experimental::OptionsNetCDF::write (C++ function), 581
bout::experimental::parseCommandLineArgs (C++ function), 319
bout::experimental::printCommandLineArguments (C++ function), 319
bout::experimental::printCompileTimeOptions (C++ function), 319
bout::experimental::printStartupHeader (C++ function), 319
bout::experimental::replace (C++ enumera- tor), 581
bout::experimental::savePIDtoFile (C++ function), 581
bout::experimental::setRunFinishInfo (C++ function), 581
bout::experimental::setRunStartInfo (C++ function), 581
bout::experimental::setupBoutLogColor (C++ function), 581
bout::experimental::setupDumpFile (C++ function), 581
bout::experimental::setupGetText (C++ function), 581
bout::experimental::setupOutput (C++ function), 581
bout::experimental::setupSignalHandler (C++ function), 581
bout::experimental::SignalHandler (C++ type), 581
bout::experimental::writeSettingsFile (C++ function), 581
bout::fft (C++ type), 413, 414
bout::fft::DST (C++ function), 413, 415
bout::fft::DST_rev (C++ function), 414, 415
bout::fft::fft_init (C++ function), 414
bout::fft::fft_initialised (C++ member), 415
bout::fft::fft_measurement_flag (C++ member), 415
bout::fft::irfft (C++ function), 413–415
bout::fft::rfft (C++ function), 413–415
bout::generator (C++ type), 464
bout::generator::Context (C++ class), 464
bout::generator::Context::Context (C++ function), 465
bout::generator::Context::get (C++ function), 465
bout::generator::Context::getMesh (C++ function), 465
bout::generator::Context::localmesh (C++ member), 465
bout::generator::Context::set (C++ function), 465
bout::generator::Context::set (C++ function), 465
bout::generator::Context::t (C++ function), 465
bout::generator::Context::x (C++ function), 465
bout::generator::Context::y (C++ function), 465
bout::generator::Context::z (C++ function), 465
bout::globals (C++ type), 305, 474
bout::globals::dump (C++ member), 474
bout::globals::mesh (C++ member), 305
bout::globals::mpi (C++ member), 474
bout::HeatFluxSNB (C++ class), 660
bout::HeatFluxSNB::~HeatFluxSNB (C++ function), 660
bout::HeatFluxSNB::beta_max (C++ member), 661
bout::HeatFluxSNB::divHeatFlux (C++ function), 660
bout::HeatFluxSNB::groupWeight (C++ function), 660
bout::HeatFluxSNB::HeatFluxSNB (C++ function), 661
bout::HeatFluxSNB::int_beta4_exp (C++ function), 661
bout::HeatFluxSNB::invertpar (C++ member), 661
bout::HeatFluxSNB::ngroups (C++ member), 661
bout::HeatFluxSNB::operator= (C++ function), 660
bout::HeatFluxSNB::r (C++ member), 661
bout::HeatFluxSNB::z (C++ member), 661
bout::inversion (C++ type), 516
bout::inversion::fieldToPetscVec (C++ function), 516
bout::inversion::identity (C++ function), 516
bout::inversion::InvertableOperator (C++ class), 517
bout::inversion::InvertableOperator::apply (C++ function), 517
bout::inversion::InvertableOperator::doneSetup (C++ member), 518
bout::inversion::InvertableOperator::functionWrapper (C++ function), 518
bout::inversion::InvertableOperator::invert (C++ function), 517
bout::inversion::InvertableOperator::InvertableOperator (C++ function), 517
bout::inversion::InvertableOperator::ksp (C++ member), 518
bout::inversion::InvertableOperator::lhs (C++ member), 518
bout::inversion::InvertableOperator::opt (C++ member), 518
bout::inversion::InvertableOperator::reportTime (C++ function), 697
bout::inversion::InvertableOperator::setup (C++ function), 517
bout::inversion::InvertableOperator::verify (C++ function), 518
bout::inversion::InvertableOperator<T>::data_type (C++ type), 517
bout::inversion::InvertableOperator<T>::function_signature (C++ type), 517
bout::inversion::petscVecToField (C++ function), 517
bout::utils (C++ type), 685, 686, 695, 696
bout::utils::_Unique_if (C++ class), 695
bout::utils::_Unique_if<T>::_Single_object (C++ type), 695
bout::utils::_Unique_if<T[N]>::_Known_bound (C++ type), 695
bout::utils::_Unique_if<T[N]>::_Unknown_bound (C++ type), 695
bout::utils::_Unique_if<T>::_Unknown_bound (C++ type), 695
bout::utils::details (C++ type), 686, 697
bout::utils::details::and_all (C++ function), 686
bout::utils::details::CompareTypes (C++ class), 697
bout::utils::details::CompareTypes::operator() (C++ function), 697
bout::utils::details::IsEqual (C++ class), 697
bout::utils::details::IsEqual::IsEqual (C++ function), 697
bout::utils::details::IsEqual::t (C++ member), 697
bout::utils::details::StaticCastOrThrow (C++ class), 698
bout::utils::details::StaticCastOrThrow::operator() (C++ function), 698
bout::utils::EnableIfField (C++ type), 685
bout::utils::EnableIfField2D (C++ type), 686
bout::utils::EnableIfField3D (C++ type), 686
bout::utils::EnableIfFieldPerp (C++ type), 686
bout::utils::function_traits::nargs (C++ member), 696
bout::utils::function_traits<R(*)(Args...)>::arg (C++ class), 695
bout::utils::function_traits<R(*)(Args...)>::arg (C++ class), 695
BoutIterationFail::BoutIterationFail
(C++ function), 327
BOUTMAIN (C macro), 623
BoutMask (C++ class), 528
BoutMask::BoutMask (C++ function), 529
BoutMask::mask (C++ member), 529
BoutMask::operator () (C++ function), 529
BoutMask::operator= (C++ function), 529
BOUTMAX (C++ function), 690
BoutMesh (C++ class), 329
BoutMesh::~BoutMesh (C++ function), 329
BoutMesh::add_target (C++ function), 336
BoutMesh::addBoundaryPar (C++ function), 334
BoutMesh::addBoundaryRegions (C++ function), 336
BoutMesh::async_send (C++ function), 336
BoutMesh::boundary (C++ member), 338
BoutMesh::clear_handles (C++ function), 336
BoutMesh::comm_inner (C++ member), 338
BoutMesh::comm_list (C++ member), 338
BoutMesh::comm_middle (C++ member), 338
BoutMesh::comm_outer (C++ member), 338
BoutMesh::comm_x (C++ member), 338
BoutMesh::CommHandle (C++ class), 338
BoutMesh::CommHandle::ddata_xsplit (C++ member), 339
BoutMesh::CommHandle::ddata_ysplit (C++ member), 339
BoutMesh::CommHandle::has_y_communication (C++ member), 339
BoutMesh::CommHandle::has_x_corners (C++ member), 339
BoutMesh::CommHandle::imsg_recvbuff (C++ member), 339
BoutMesh::CommHandle::msg_recvbuff (C++ member), 339
BoutMesh::CommHandle::msend_sendbuff (C++ member), 339
BoutMesh:: CommHandle::request (C++ member), 338
BoutMesh::CommHandle::sendreq (C++ member), 338
BoutMesh::CommHandle::umsg_recvbuff (C++ member), 339
BoutMesh::CommHandle::umsg_sendbuff (C++ member), 339
BoutMesh::CommHandle::var_list (C++ member), 339
BoutMesh::CommHandle::xbufflen (C++ member), 339
BoutMesh::CommHandle::ybufflen (C++ member), 339
BoutMesh::DDATA_INDEST (C++ member), 338
BoutMesh::DDATA_OUTDEST (C++ member), 338
BoutMesh::DDATA_XSPLIT (C++ member), 338
BoutMesh::default_connections (C++ function), 336
BoutMesh::DownXSplitIndex (C++ function), 333
BoutMesh::firstX (C++ function), 331
BoutMesh::firstY (C++ function), 332
BoutMesh::free_handle (C++ function), 336
BoutMesh::get_handle (C++ function), 336
BoutMesh::getBoundaries (C++ function), 334
BoutMesh::getBoundariesPar (C++ function), 334
BoutMesh::getGlobalXIndex (C++ function), 334
BoutMesh::getGlobalYIndexNoBoundaries (C++ function), 334
BoutMesh::getGlobalYIndexNoBoundaries (C++ function), 335
BoutMesh::getGlobalYIndexNoBoundaries (C++ function), 335
BoutMesh::getGlobalZIndex (C++ function), 335
BoutMesh::getGlobalZIndexNoBoundaries (C++ function), 335
BoutMesh::getGlobalZIndexNoBoundaries (C++ function), 335
BoutMesh::getIxseps1 (C++ function), 334
BoutMesh::getIxseps2 (C++ function), 334
BoutMesh::getLocalXIndex (C++ function), 334
BoutMesh::getLocalXIndexNoBoundaries (C++ function), 335
BoutMesh::getLocalYIndex (C++ function), 335
BoutMesh::getLocalYIndexNoBoundaries (C++ function), 335
BoutMesh::getLocalZIndex (C++ function), 335
BoutMesh::getLocalZIndexNoBoundaries (C++ function), 335
BoutMesh::getNx (C++ function), 334
BoutMesh::getNXPE (C++ function), 330
BoutMesh::getNy (C++ function), 334
BoutMesh::getNYPE (C++ function), 330
BoutMesh::getXcomm (C++ function), 331
BoutMesh::getXProcIndex (C++ function), 330
BoutMesh::getYcomm (C++ function), 332
BoutMesh::getYProcIndex (C++ function), 331
BoutMesh::GlobalX (C++ function), 334
BoutMesh::GlobalY (C++ function), 334
BoutMesh::gridname (C++ member), 337
BoutMesh::hasBranchCutLower (C++ function), 332
Coordinates::ShiftTorsion (C++ member), 345
Coordinates::transform (C++ member), 345
Coordinates::Vpar_Grad_par (C++ function), 343
Coordinates::zlength (C++ function), 342
copy (C++ function), 293, 435
copy_string (C++ function), 688, 691
cos (C++ function), 434
cosh (C++ function), 435
create() (in module boutdata.restart), 724
create_cache() (in module boutdata.collect), 715
create_processor_layout() (in module boutdata.processor_rearrange), 723
CROSS (C macro), 705
cross (C++ function), 702, 706, 707
ctu (C++ enumerator), 394
Curl (C++ function), 699, 701
CurvedSlab (class in zoidberg.field), 749
cvode_bbd_rhs (C++ function), 346
cvode_jac (C++ function), 346
cvode_pre (C++ function), 346
cvode_pre_shim (C++ function), 346
cvode_rhs (C++ function), 346
CVODEINT (C++ type), 345
CvodeSolver (C++ class), 346
CvodeSolver::~CvodeSolver (C++ function), 346
CvodeSolver::cvode_initialised (C++ member), 347
CvodeSolver::cvode_mem (C++ member), 347
CvodeSolver::CvodeSolver (C++ function), 346
CvodeSolver::diagnose (C++ member), 347
CvodeSolver::getCurrentTimestep (C++ function), 346
CvodeSolver::hcur (C++ member), 347
CvodeSolver::init (C++ function), 346
CvodeSolver::jac (C++ function), 346
CvodeSolver::jacfunc (C++ member), 347
CvodeSolver::loop_abstol_values_op (C++ function), 347
CvodeSolver::NOUT (C++ member), 347
CvodeSolver::pre (C++ function), 346
CvodeSolver::pre_ncalls (C++ member), 347
CvodeSolver::pre_Wtime (C++ member), 347
CvodeSolver::resetInternalFields (C++ function), 346
CvodeSolver::rhs (C++ function), 346
CvodeSolver::run (C++ function), 346
CvodeSolver::set_abstol_values (C++ function), 347
CvodeSolver::setJacobian (C++ function), 346
CvodeSolver::TIMESTEP (C++ member), 347
CvodeSolver::uvec (C++ member), 347
CVSpilsSetJacTimes (C++ function), 346
cyclic_tridag (C++ function), 520, 521
CyclicReduce (C++ class), 350
CyclicReduce::~CyclicReduce (C++ function), 350
CyclicReduce::allocMemory (C++ function), 351
CyclicReduce::back_solve (C++ function), 351
CyclicReduce::coeffs (C++ member), 351
CyclicReduce::comm (C++ member), 351
CyclicReduce::CyclicReduce (C++ function), 350
CyclicReduce::if2x2 (C++ member), 352
CyclicReduce::ifs (C++ member), 351
CyclicReduce::ifp (C++ member), 352
CyclicReduce::ifx (C++ member), 352
CyclicReduce::myn (C++ member), 351
CyclicReduce::myproc (C++ member), 351
CyclicReduce::N (C++ member), 351
CyclicReduce::nprocs (C++ member), 351
CyclicReduce::periodic (C++ member), 351
CyclicReduce::recvbuffer (C++ member), 351
CyclicReduce::reduce (C++ function), 351
CyclicReduce::setCoefs (C++ function), 350
CyclicReduce::setPeriodic (C++ function), 350
CyclicReduce::setup (C++ function), 350
CyclicReduce::solve (C++ function), 350
CyclicReduce::sys0 (C++ member), 351
CyclicReduce::sys1 (C++ member), 352
CyclicReduce::xn (C++ member), 352

D
D2DX2 (C++ function), 366, 378
D2DXY (C++ function), 369, 387, 388
D2DXZ (C++ function), 369, 370, 388, 389
D2DY2 (C++ function), 366, 378, 379
D2DYDZ (C++ function), 370, 389, 390
D2DZ2 (C++ function), 367, 379, 380
D4DX (C++ function), 367, 368, 380, 381
D4DY4 (C++ function), 368, 381
D4DZ4 (C++ function), 368, 369, 381, 382
data_format (C++ function), 356
datafile (C++ class), 352
datafile (class in boututils.datafile), 734
datafile::~Datafile (C++ function), 353
datafile::add (C++ function), 353, 354
datafile::addOnce (C++ function), 353
datafile::addRepeat (C++ function), 353
datafile::appending (C++ member), 355
datafile::bool_arr (C++ member), 355
datafile::BoutReal_arr (C++ member), 355
Datafile::close (C++ function), 353
Datafile::Datafile (C++ function), 353, 354
Datafile::enabled (C++ member), 355
Datafile::f2d_arr (C++ member), 355
Datafile::f3d_arr (C++ member), 355
Datafile::file (C++ member), 355
Datafile::filename (C++ member), 355
Datafile::first_time (C++ member), 355
Datafile::floats (C++ member), 354
Datafile::flush (C++ member), 354
Datafile::flushFrequency (C++ member), 355
Datafile::flushFrequencyCounter (C++ member), 355
Datafile::fperp_arr (C++ member), 355
Datafile::guards (C++ member), 354
Datafile::init_missing (C++ member), 355
Datafile::isValid (C++ function), 353
Datafile::Lx (C++ member), 355
Datafile::Ly (C++ member), 355
Datafile::Lz (C++ member), 355
Datafile::mesh (C++ member), 354
Datafile::open (C++ function), 353
Datafile::openclose (C++ member), 355
Datafile::openr (C++ function), 353
Datafile::openw (C++ function), 353
Datafile::operator= (C++ function), 353
Datafile::parallel (C++ member), 354
Datafile::read (C++ function), 354
Datafile::read_f2d (C++ function), 354
Datafile::read_f3d (C++ function), 354
Datafile::read_fperp (C++ function), 354
Datafile::setAttribute (C++ function), 354
Datafile::setLowPrecision (C++ function), 353
Datafile::shiftOutput (C++ member), 355
Datafile::v2d_arr (C++ member), 355
Datafile::v3d_arr (C++ member), 355
Datafile::varAdded (C++ function), 354
Datafile::varPtr (C++ function), 354
Datafile::VarStr (C++ class), 355
Datafile::VarStr::covar (C++ member), 355
Datafile::VarStr::name (C++ member), 355
Datafile::VarStr::ptr (C++ member), 355
Datafile::VarStr::save_repeat (C++ member), 355
Datafile::writable (C++ member), 355
Datafile::write (C++ function), 354
Datafile::write_f2d (C++ function), 354
Datafile::write_f3d (C++ function), 354
Datafile::write_fperp (C++ function), 354
Datafile::write_int (C++ function), 354
Datafile::write_real (C++ function), 354
DataFile_HDF5 (class in boututils.datafile), 736
DataFile_netCDF (class in boututils.datafile), 738
DataFormat (C++ class), 356
DataFormat::~DataFormat (C++ function), 356
DataFormat::addVarBoutReal (C++ function), 357
DataFormat::addVarField2D (C++ function), 357
DataFormat::addVarField3D (C++ function), 357
DataFormat::addVarFieldPerp (C++ function), 357
DataFormat::addVarInt (C++ function), 357
DataFormat::close (C++ function), 356
DataFormat::DataFormat (C++ function), 356
DataFormat::flush (C++ function), 356
DataFormat::getAttribute (C++ function), 358, 359
DataFormat::getSize (C++ function), 356
DataFormat::isValid (C++ function), 356
DataFormat::is_valid (C++ function), 356
DataFormat::mesh (C++ member), 359
DataFormat::openr (C++ function), 356
DataFormat::openw (C++ function), 356
DataFormat::read (C++ function), 357
DataFormat::read_perp (C++ function), 357
DataFormat::read_rec (C++ function), 357
DataFormat::read_rec_perp (C++ function), 357
DataFormat::readFieldAttributes (C++ function), 359
DataFormat::setAttribute (C++ function), 357, 358
DataFormat::setGlobalOrigin (C++ function), 357
DataFormat::setLocalOrigin (C++ function), 357
DataFormat::setLowPrecision (C++ function), 357
DataFormat::setRecord (C++ function), 357
DataFormat::write (C++ function), 357
DataFormat::write_perp (C++ function), 357
DataFormat::write_rec (C++ function), 357
DataFormat::write_rec_perp (C++ function), 357
DataFormat::writeFieldAttributes (C++ function), 359
DC (C++ function), 422, 424
dcomplex (C++ type), 360
ddt (C++ function), 417, 424, 703, 707
DDX (C++ function), 363, 364, 375, 376
DDY (C++ function), 364, 376
DDZ (C++ function), 364, 365, 376–378
default (gen_fieldops attribute), 461
DEFAULT_DIR (C++ member), 317
DerivativeStore::getFlowDerivative (C++ function), 362
DerivativeStore::getAvailableMethods (C++ function), 362
DerivativeStore::forceDefaultMethod (C++ function), 362
DerivativeStore::defaultMethods (C++ function), 362
DerivativeStore::clear (C++ function), 362
DeferredFunction::DeferredFunction (C++ function), 362
DeferredFunction::operator() (C++ function), 362
DeferredFunction::storedFunc (C++ member), 362
DEFINe_Binary_op (C macro), 401
DEFINe_Flux_deriv (C macro), 494
DEFINe_Flux deriv core (C macro), 494
DEFINe_Flux deriv staggered (C macro), 494
DEFINe_oVerload_func (C macro), 401
DEFINe_StANDard deriv (C macro), 494
DEFINe_StANDard deriv core (C macro), 494
DEFINe_StANDard deriv staggered (C macro), 494
DEFINe_upwind deriv (C macro), 494
DEFINe_upwind deriv core (C macro), 494
DEFINe_upwind deriv staggered (C macro), 494
deflt (C++ enumerator), 321, 322
deflt2 (C++ function), 392, 397
DEPRECATED (C macro), 360
DERIV (C++ enum), 323
deriv() (in module boututils.calculus), 732
deriv2d() (in module boututils.calculus), 732
DERIV_FUNC REGION ENUM to STRING macro), 341, 375
DERIV_STRING (C++ function), 324
DerivativeStore (C++ class), 361
DerivativeStore::clear (C++ function), 362
DerivativeStore::defaultMethods (C++ member), 363
DerivativeStore::DerivativeStore (C++ function), 361, 363
DerivativeStore::flux (C++ member), 363
DerivativeStore::forceDefaultMethod (C++ function), 362
DerivativeStore::getAvailableMethods (C++ function), 361
DerivativeStore::getFlowDerivative (C++ function), 362
DerivativeStore::getfluxDerivative (C++ function), 362
DerivativeStore::getInstance (C++ function), 362
DerivativeStore::getKey (C++ function), 363
DerivativeStore::getMethodName (C++ function), 363
DerivativeStore::getStandard2ndDerivative (C++ function), 362
DerivativeStore::getStandard4thDerivative (C++ function), 362
DerivativeStore::getStandardDerivative (C++ function), 362
DerivativeStore::getUpwindDerivative (C++ function), 362
DerivativeStore::initialise (C++ function), 362
DerivativeStore::isEmpty (C++ function), 361
DerivativeStore::listAvailableMethods (C++ function), 361
DerivativeStore::nameLookup (C++ function), 363
DerivativeStore::registerDerivative (C++ function), 362
DerivativeStore::registeredMethods (C++ member), 363
DerivativeStore::reset (C++ function), 362
DerivativeStore::setDefaults (C++ function), 363
DerivativeStore::standard (C++ member), 363
DerivativeStore::standardFourth (C++ member), 363
DerivativeStore::standardSecond (C++ member), 363
DerivativeStore<FieldType>::flowFunc (C++ type), 361
DerivativeStore<FieldType>::fluxFunc (C++ type), 361
DerivativeStore<FieldType>::standardFunc (C++ type), 361
DerivativeStore<FieldType>::storageType (C++ type), 361
DerivativeStore<FieldType>::upwindFunc (C++ type), 361
DerivativeType (C++ class), 495
DerivativeType::apply (C++ function), 495
DerivativeType::func (C++ member), 495
DerivativeType::meta (C++ member), 495
DerivativeType::standard (C++ function), 495
DerivativeType::upwindOrFlux (C++ function), 495
dest (gen_fieldops attribute), 461
detect_local_maxima () (in module boututils.local_min_max), 741
detect_local_minima () (in module boututils.local_min_max), 741
determineNumberOfCPUs () (in module boututils.run_wrapper), 743
DIFF_C2 (C++ member), 324
DIFF_C4 (C++ member), 324
DIFF_DEFAULT (C++ member), 324
DIFF_FFT (C++ member), 324
DIFF_METHOD (C++ enum), 322
DIFF_METHOD_STRING (C++ function), 323
DIFF_S2 (C++ member), 324
DIFF_SPLIT (C++ member), 324
DIFF_U1 (C++ member), 324
DIFF_U2 (C++ member), 324
DIFF_U3 (C++ member), 324
DIFF_W2 (C++ member), 324
dimensions (gen_fieldops.Field attribute), 462
dimensions() (boututils.datafile.DataFile method), 735
dimensions() (boututils.datafile.DataFile_HDF5 method), 737
dimensions() (boututils.datafile.DataFile_netCDF method), 738
directions (in module boutdata.collect), 715
dIRECTION (C++ enum), 322
Direction (C++ enum), 474
dIRECTION_STRING (C++ function), 324
directionTypes (C++ class), 325
directionTypes::y (C++ member), 325
directionTypes::z (C++ member), 325
distance() (zoidberg.rzline.RZline method), 765
Div (C++ function), 699, 701
Div_par (C++ function), 391, 395, 396
Div_par_CtoL (C++ function), 397
Div_par_flux (C++ function), 391, 396
Div_par_K_Grad_par (C++ function), 392, 397
Div_par_LtoC (C++ function), 397
Divide (C++ class), 405
Divide::apply (C++ function), 405
double_arg_op (C++ type), 443
dST (C++ function), 413
dST_rev (C++ function), 413
dummyOutput (C++ class), 585
dummyOutput::disable (C++ function), 586
dummyOutput::setEnabled (C++ function), 586
dummyOutput::is_enabled (C++ function), 586
dummyOutput::print (C++ function), 586
dummyOutput::write (C++ function), 586
EmptyFormat (C++ class), 399
EmptyFormat::close (C++ function), 399
EmptyFormat::EmptyFormat (C++ function), 399
EmptyFormat::getSize (C++ function), 399
EmptyFormat::is_valid (C++ function), 399
EmptyFormat::openr (C++ function), 399
EmptyFormat::openw (C++ function), 399
EmptyFormat::read (C++ function), 399
EmptyFormat::read_rec (C++ function), 399, 400
EmptyFormat::setOrigin (C++ function), 399
EmptyFormat::setRecord (C++ function), 399
EmptyFormat::write (C++ function), 399
EmptyFormat::write_rec (C++ function), 400
emptyFrom (C++ function), 432, 703, 707
emptyFrom<FieldPerp> (C++ function), 454
ENUMSTR (C macro), 321
e numWrapper (C++ class), 325
e numWrapper::lookup (C++ function), 325
e numWrapper::value (C++ member), 325
e numWrapper<T, val>::type (C++ type), 325
equallySpaced() (zoidberg.rzline.RZline method), 765
estimate (C++ enumerator), 412
EulerSolver (C++ class), 400
EulerSolver::EulerSolver (C++ function), 400
EulerSolver::cfl_factor (C++ member), 400
EulerSolver::EulerSolver (C++ function), 400
EulerSolver::f0 (C++ member), 400
EulerSolver::f1 (C++ member), 400
EulerSolver::getCurrentTimestep (C++ function), 400
EulerSolver::init (C++ function), 400
EulerSolver::maxstep (C++ member), 400
EulerSolver::nlocal (C++ member), 401
EulerSolver::nsteps (C++ member), 401
EulerSolver::out_timestep (C++ member), 400
EulerSolver::run (C++ function), 400
EulerSolver::setMaxTimestep (C++ function), 400
EulerSolver::take_step (C++ function), 400
EulerSolver::timestep (C++ member), 401
EulerSolver::timestep_reduced (C++ member), 401
eval3D (C++ function), 401
evaluate() (boutdata.data.BoutOptionsFile method), 719
evaluate_scalar() (boutdata.data.BoutOptionsFile method), 718
evolveVariables() (boutdata.data.BoutOutputs method), 720
exhaustive (C++ enumerator), 412
exp (C++ function), 433, 434
ExpressionParser (C++ class), 407
ExpressionParser::ExpressionParser (C++ function), 407
ExpressionParser::addBinaryOp (C++ function), 407
ExpressionParser::addGenerator (C++ function), 407
ExpressionParser::bin_op (C++ member), 408
ExpressionParser::ExpressionParser (C++ function), 407
ExpressionParser::gen (C++ member), 408
ExpressionParser::LexInfo (C++ class), 408
ExpressionParser::LexInfo::curident (C++ member), 408
ExpressionParser::LexInfo::curtok (C++ member), 408
ExpressionParser::LexInfo::curval (C++ member), 408
ExpressionParser::LexInfo::LastChar (C++ function), 408
ExpressionParser::LexInfo::nextToken (C++ function), 408
ExpressionParser::LexInfo::reserved_chars (C++ member), 409
ExpressionParser::LexInfo::ss (C++ member), 409
ExpressionParser::parseBinOpRHS (C++ function), 408
ExpressionParser::parseContextExpr (C++ function), 408
ExpressionParser::parseExpression (C++ function), 408
ExpressionParser::parseIdentifierExpr (C++ function), 408
ExpressionParser::parseParenExpr (C++ function), 408
ExpressionParser::parsePrimary (C++ function), 408
ExpressionParser::parseString (C++ function), 407
ExpressionParser::resolve (C++ function), 407
exprTraits (C++ class), 402
eexprTraits<double> (C++ class), 402
eexprTraits::expr_type (C++ type), 402
eexprTraits<ExprT>::expr_type (C++ type), 402
eexprTraits<float> (C++ class), 402
eexprTraits<float>::expr_type (C++ type), 403
eexprTraits<int> (C++ class), 403
eexprTraits<int>::expr_type (C++ type), 403
Factory (C++ class), 466
Factory:::-Factory (C++ function), 466
Factory::add (C++ function), 466
Factory::create (C++ function), 467
Factory::ensureRegistered (C++ function), 468
Factory::Factory (C++ function), 468
Factory::getDefaultType (C++ function), 468
Factory::getInstance (C++ function), 468
Factory::getOptionName (C++ function), 468
Factory::getOptionName (C++ function), 468
Factory::getType (C++ function), 467
Factory::listAvailable (C++ function), 467
Factory::optionsOrDefaultSection (C++ function), 468
Factory::remove (C++ function), 467
Factory::type_map (C++ member), 468
Factory<DerivedFactory, BaseType>:: TypeCreator>::ReturnType (C++ type), 468
false (gen_fieldops attribute), 461
fci_to_vtk () (in module zoidberg.zoidberg), 768
FCIMap (C++ class), 410
FCIMap::boundary_mask (C++ member), 411
FCIMap::corner_boundary_mask (C++ member), 411
FCIMap::corner_boundary_mask (C++ member), 411
FCIMap::corner_boundary_mask (C++ member), 411
FCITransform (C++ class), 411
FCITransform::calcParallelSlices (C++ function), 411
FCITransform::canToFromFieldAligned (C++ function), 411
FCITransform::checkInputGrid (C++ function), 411
FCITransform::checkInputGrid (C++ function), 411
FCITransform::field_line_maps (C++ member), 412
FCITransform::fromFieldAligned (C++ function), 411
FCITransform::integrateParallelSlices (C++ function), 411
FCITransform::requiresTwistShift (C++ function), 411
FCITransform::toFieldAligned (C++ function), 411
fcmlx (C++ class), 360
fcmlx::i (C++ member), 360
fcmlx::r (C++ member), 360
FDDX (C++ function), 372, 373, 385, 386
FDDY (C++ function), 373, 386
FDDZ (C++ function), 374, 387
fft (C++ enumerator), 322
fft_deriv() (in module boututils.fft_deriv), 739
fft_integrate() (in module boututils.fft_integrate), 740

FIELD (C++ enumerator), 588
Field2D (C++ class), 417
Field2D::~Field2D (C++ function), 418
Field2D::accept (C++ function), 419
Field2D::allocate (C++ function), 418
Field2D::applyBoundary (C++ function), 420
Field2D::applyTDerivBoundary (C++ function), 420
Field2D::begin (C++ function), 419
Field2D::BoutRealSize (C++ function), 420
Field2D::byteSize (C++ function), 420
Field2D::dData (C++ member), 420
Field2D::deriv (C++ member), 420
Field2D::doneComms (C++ function), 420
Field2D::end (C++ function), 419
Field2D::Field2D (C++ function), 417, 418
Field2D::getNx (C++ function), 418
Field2D::getNy (C++ function), 418
Field2D::getNz (C++ function), 418
Field2D::getRegion (C++ function), 418, 419
Field2D::hasParallelSlices (C++ function), 418
Field2D::hasYupYdown (C++ function), 418
Field2D::is3D (C++ function), 417
Field2D::isAllocated (C++ function), 418
Field2D::isReal (C++ function), 419
Field2D::nx (C++ member), 420
Field2D::ny (C++ member), 420
Field2D::operator () (C++ function), 419
Field2D::operator= (C++ function), 419
Field2D::operator+= (C++ function), 419
Field2D::operator-= (C++ function), 419
Field2D::operator/= (C++ function), 419
Field2D::operator= (C++ function), 419
Field2D::operator[] (C++ function), 419
Field2D::setBoundaryTo (C++ function), 420
Field2D::setDirectionY (C++ function), 418
Field2D::setLocation (C++ function), 418
Field2D::timeDeriv (C++ function), 418
Field2D::valueDeriv (C++ function), 417
Field2D::ydown (C++ function), 418
Field2D::ynext (C++ function), 418
Field2D::yup (C++ function), 418
Field2DExpr (C++ class), 402
Field2DExpr::Field2DExpr (C++ function), 402
Field2DExpr::operator () (C++ function), 402
Field2DExpr::type (C++ type), 402
Field3D (C++ class), 425
Field3D::~Field3D (C++ function), 428
Field3D::accept (C++ function), 430
Field3D::allocate (C++ function), 428
Field3D::applyBoundary (C++ function), 430
Field3D::applyParallelBoundary (C++ function), 430, 431
Field3D::applyTDerivBoundary (C++ function), 430
Field3D::background (C++ member), 431
Field3D::begin (C++ function), 429
Field3D::BoutRealSize (C++ function), 430
Field3D::byteSize (C++ function), 430
Field3D::calcParallelSlices (C++ function), 430
Field3D::clearParallelSlices (C++ function), 428
Field3D::data (C++ member), 431
Field3D::deriv (C++ member), 431
Field3D::doneComms (C++ function), 430
Field3D::end (C++ function), 429
Field3D::Field3D (C++ function), 428
Field3D::getNx (C++ function), 428
Field3D::getNy (C++ function), 428
Field3D::getNz (C++ function), 428
Field3D::getRegion (C++ function), 429
Field3D::getRegion2D (C++ function), 429
Field3D::hasParallelSlices (C++ function), 429
Field3D::hasYupYdown (C++ function), 429
Field3D::ind_type (C++ type), 427
Field3D::is3D (C++ function), 430
Field3D::isAllocated (C++ function), 428
Field3D::isReal (C++ function), 430
Field3D::mergeYupYdown (C++ function), 428
Field3D::nx (C++ member), 431
Field3D::ny (C++ member), 431
Field3D::nz (C++ member), 431
Field3D::operator () (C++ function), 429, 430
Field3D::operator= (C++ function), 427
Field3D::operator/= (C++ function), 427
Field3D::operator/= (C++ function), 427
Field3D::operator/= (C++ function), 427
Field3D::operator/= (C++ function), 427
Field3D::operator/= (C++ function), 427
Field3D::operator/= (C++ function), 427
Field3D::requiresTwistShift (C++ function), 429
Field3D::setBoundaryTo (C++ function), 430
Field3D::setDirectionY (C++ function), 428
Field3D::setLocation (C++ function), 428
Field3D::splitParallelSlices (C++ function), 428
Field3D::splitYupYdown (C++ function), 428
Field3D::timeDeriv (C++ function), 428
Field3D::value_type (C++ type), 427
Field3D::ydown (C++ function), 429
Field3D::ydown_fields (C++ member), 431
Field3D::yup (C++ function), 429
Field3D::yup_fields (C++ member), 431
Field3DExpr::clone (C++ class), 401
Field3DExpr::data (C++ member), 402
Field3DExpr::Field3DExpr (C++ function), 402
Field3DExpr::type (C++ type), 402
Field::~Field (C++ function), 436
Field::bndry_xin (C++ member), 437
Field::bndry_xout (C++ member), 437
Field::bndry_ydown (C++ member), 437
Field::bndry_yup (C++ member), 437
Field::bndryValid (C++ function), 436
Field::copyFieldMembers (C++ function), 437
Field::directions (C++ member), 437
Field::Field (C++ function), 436
Field::fieldCoordinates (C++ member), 437
Field::fieldmesh (C++ member), 437
Field::getCoordinates (C++ function), 437
Field::getDirections (C++ function), 436
Field::getDirectionY (C++ function), 436
Field::getDirectionZ (C++ function), 436
Field::getLocation (C++ function), 436
Field::getMesh (C++ function), 437
Field::getNx (C++ function), 437
Field::getNy (C++ function), 437
Field::getNz (C++ function), 437
Field::getLocation (C++ member), 437
Field::name (C++ member), 437
Field::operator= (C++ function), 436
Field::setDirectionY (C++ function), 436
Field::setDirectionZ (C++ function), 436
Field::setLocation (C++ function), 436
field_direction() (zoldberg.field.MagneticField method), 752
FIELD_FUNC (C macro), 432
field_type (gen_fieldops.Field attribute), 462
FieldATan (C++ class), 445
FieldATan::A (C++ member), 445
FieldATan::B (C++ member), 445
FieldATan::clone (C++ function), 445
FieldATan::FieldATan (C++ function), 445
FieldATan::generate (C++ function), 445
FieldBallooning (C++ class), 448
FieldBallooning::arg (C++ member), 449
FieldBallooning::ball_n (C++ member), 449
FieldFactory::create3D (C++ function), 440
FieldFactory::createPerp (C++ function), 440
FieldFactory::FieldFactory (C++ function),
FieldFactory::fieldmesh (C++ member), 441
FieldFactory::findOption (C++ function), 441
FieldFactory::get (C++ function), 441
FieldFactory::lookup (C++ member), 441
FieldFactory::max_recursion_depth (C++ member), 441
FieldFactory::options (C++ member), 441
FieldFactory::parse (C++ function), 440
FieldFactory::resolve (C++ function), 441
FieldFactory::transform_from_field_alignment (C++ member), 441
FieldFunction::FieldFunction (C++ function), 441
FieldFunction::::func (C++ member), 442
FieldFunction::generate (C++ function), 441
FieldGaussian::FieldGaussian (C++ class), 445
FieldGaussian::clone (C++ function), 445
FieldGaussian::FieldGaussian (C++ function), 445
FieldGaussian::generate (C++ function), 445
FieldGaussian:::: (C++ member), 446
FieldGaussian::::X (C++ member), 446
FieldGenerator (C++ class), 406
FieldGenerator::~FieldGenerator (C++ function), 406
FieldGenerator::clone (C++ function), 406
FieldGenerator::generate (C++ function), 406
FieldGeneratorPtr (C++ type), 406
FieldGenOneArg (C++ class), 443
FieldGenOneArg::clone (C++ function), 443
FieldGenOneArg::FieldGenOneArg (C++ function), 443
FieldGenOneArg::::gen (C++ member), 444
FieldGenOneArg::generate (C++ function), 444
FieldGenOneArg::name (C++ member), 444
FieldGenOneArg::str (C++ function), 444
FieldGenTwoArg (C++ class), 444
FieldGenTwoArg::A (C++ member), 444
FieldGenTwoArg::B (C++ member), 444
FieldGenTwoArg::clone (C++ function), 444
FieldGenTwoArg::FieldGenTwoArg (C++ function), 444
FieldGenTwoArg::generate (C++ function), 444
FieldGenTwoArg::name (C++ member), 444
FieldGenTwoArg::str (C++ function), 444
FieldGroup (C++ class), 451
FieldGroup::add (C++ function), 452, 453
FieldGroup::begin (C++ function), 453
GlobalField::mesh (C++ member), 470
GlobalField::mype (C++ member), 470
GlobalField::npes (C++ member), 470
GlobalField::nx (C++ member), 470
GlobalField::ny (C++ member), 470
GlobalField::nz (C++ member), 470
GlobalField::proc_local_origin (C++ function), 469
GlobalField::proc_origin (C++ function), 470
GlobalField::proc_size (C++ function), 470
GlobalField::proc_valid (C++ function), 469
GlobalField::proc_world_size (C++ function), 469
GlobalField::operator() (C++ function), 469
GlobalField::nz (C++ function), 469
GlobalIndexer::globalStart (C++ function), 606, 608
GlobalIndexer::indices (C++ member), 606, 608
GlobalIndexer::initialise (C++ function), 605, 607
GlobalIndexer::initialiseTest (C++ function), 605, 607
GlobalIndexer::insertIndex (C++ function), 607, 609
GlobalIndexer::isLocal (C++ function), 606, 607
GlobalIndexer::numDiagonal (C++ member), 607, 609
GlobalIndexer::numOffDiagonal (C++ member), 607, 609
GlobalIndexer::regionAll (C++ member), 606, 608
GlobalIndexer::regionBndry (C++ member), 607, 609
GlobalIndexer::regionInnerX (C++ member), 607, 609
GlobalIndexer::regionLowerY (C++ member), 606, 608
GlobalIndexer::regionOuterX (C++ member), 607, 609
GlobalIndexer::regionUpperY (C++ member), 606, 608
GlobalIndexer::registerFieldForTest (C++ function), 606, 608
GlobalIndexer::size (C++ function), 606, 608
GlobalIndexer::sparsityCalculated (C++ member), 607, 609
GlobalIndexer::sparsityPatternAvailable (C++ function), 606, 608
GlobalIndexer::stencils (C++ member), 606, 608
GlobalIndexer<T>::ind_type (C++ type), 605, 607
GLOBALORIGIN (C macro), 316
Grad (C++ function), 698, 700
Grad2_par (C++ function), 391, 392, 396
GRAD_FUNC_REGION_ENUM_TO_STRING (C macro), 341
Grad_par (C++ function), 390, 394, 395
Grad_par_CtoL (C++ function), 396
Grad_par_LtoC (C++ function), 396, 397
Grad_parP (C++ function), 390, 395
Grad_perp (C++ function), 698–700
Grid (class in zoidberg.grid), 756
grid_annulus () (in module zoidberg.poloidal_grid), 762
grid_elliptic () (in module zoidberg.poloidal_grid), 763
GRID_LOAD (C macro), 474
GRID_LOAD1 (C macro), 473
IMEXBDF2::getCurrentTimestep (C++ function), 488
IMEXBDF2::gFac (C++ member), 490
IMEXBDF2::have_constraints (C++ member), 491
IMEXBDF2::IMEXBDF2 (C++ function), 488
IMEXBDF2::implicit_curtime (C++ member), 490
IMEXBDF2::implicit_gamma (C++ member), 490
IMEXBDF2::init (C++ function), 488
IMEXBDF2::is_dae (C++ member), 491
IMEXBDF2::Jmf (C++ member), 491
IMEXBDF2::lib (C++ member), 491
IMEXBDF2::linear_fails (C++ member), 491
IMEXBDF2::loadVars (C++ function), 489
IMEXBDF2::loopVars (C++ function), 489
IMEXBDF2::MAX_SUPPORTED_ORDER (C++ member), 491
IMEXBDF2::maxOrder (C++ member), 489
IMEXBDF2::mxstep (C++ member), 490
IMEXBDF2::mxstepAdapt (C++ member), 490
IMEXBDF2::nadapt (C++ member), 490
IMEXBDF2::neq (C++ member), 490
IMEXBDF2::ninternal (C++ member), 490
IMEXBDF2::nlocal (C++ member), 491
IMEXBDF2::nonlinear_fails (C++ member), 491
IMEXBDF2::nsteps (C++ member), 489
IMEXBDF2::out_timestep (C++ member), 489
IMEXBDF2::precon (C++ function), 488
IMEXBDF2::predictor (C++ member), 491
IMEXBDF2::rhs (C++ member), 490
IMEXBDF2::run (C++ function), 488
IMEXBDF2::saveDerivs (C++ function), 489
IMEXBDF2::saveVars (C++ function), 489
IMEXBDF2::scaleCushDown (C++ member), 490
IMEXBDF2::scaleCushUp (C++ member), 490
IMEXBDF2::shuffleState (C++ function), 489
IMEXBDF2::snes (C++ member), 491
IMEXBDF2::snes_f (C++ member), 491
IMEXBDF2::snes_function (C++ function), 488
IMEXBDF2::snes_x (C++ member), 491
IMEXBDF2::snesAlt (C++ member), 491
IMEXBDF2::snesUse (C++ member), 491
IMEXBDF2::solve_implicit (C++ function), 489
IMEXBDF2::take_step (C++ function), 489
IMEXBDF2::timestep (C++ member), 489
IMEXBDF2::timesteps (C++ member), 490
IMEXBDF2::u (C++ member), 490
IMEXBDF2::uFac (C++ member), 490
IMEXBDF2::uV (C++ member), 490
IMEXBDF2::verbose (C++ member), 491
IMEXBDF2::PCapply (C++ function), 486
impl (boututils.datafile.DataFile attribute), 735
IN_SEN'T DOWN (C++ member), 329
IN_SEN'T OUT (C++ member), 329
IN_SEN'T UP (C++ member), 329
Ind2D (C++ type), 635
Ind3D (C++ type), 635
IND_2D (C++ enumerator), 635
IND_3D (C++ enumerator), 635
IND_ERP (C++ enumerator), 635
IND_TYPE (C++ enum), 635
index_var (gen_fieldops.Field attribute), 462
IndexerPtr (C++ type), 605
IndexOffset (C++ class), 569
IndexOffset::dx (C++ member), 569
IndexOffset::dy (C++ member), 569
IndexOffset::operator= (C++ function), 569
IndexOffset::operator-= (C++ function), 569
IndexOffset::xm (C++ function), 569
IndexOffset::xp (C++ function), 569
IndexOffset::yp (C++ function), 569
IndexOffset::zm (C++ function), 569
IndexOffset::zp (C++ function), 569
INDIRECT0 (C macro), 567
INDIRECT0_BOUTMAIN (C macro), 316
INDIRECT1 (C macro), 568
INDIRECT1_BOUTMAIN (C macro), 316
INDIRECT2 (C macro), 568
IndPerp (C++ type), 635
initial_profile (C++ function), 497–499
int_func() (in module boututils.int_func), 740
intergrate() (in module boututils.calculus), 732
interp (C++ function), 499
interp_to (C++ function), 499, 500
interpolate (C++ function), 500
InterpolationWeights (C++ type), 605
invalidateGuards (C++ function), 415, 417, 422, 424, 445
invert3x3 (C++ function), 690
INVERT_4TH_ORDER (C++ member), 509
INVERT_AC_GRAD (C++ member), 509
INVERT_AC_LAP (C++ member), 509
INVERT_BNDRY_ONE (C++ member), 509
INVERT_BOTH_BNDRY_ONE (C++ member), 509
INVERT_DC_GRAD (C++ member), 509
INVERT_DC_GRADPAR (C++ member), 509
INVERT_DC_GRADPARINV (C++ member), 509
INVERT_DC_LAP (C++ member), 509
INVERT_IN_CYLINDER (C++ member), 509
INVERT_KX_ZERO (C++ member), 509
INVERT_RHS (C++ member), 509
INVERT_SET (C++ member), 509
INVERT_START_NEW (C++ member), 509
INVERT_SYM (C++ member), 509
INVERT_ZERO_DC (C++ member), 509
InvertPar (C++ class), 515
InvertPar:::InvertPar (C++ function), 515
InvertPar::create (C++ function), 516
InvertPar:::InvertPar (C++ function), 515
InvertPar:::localmesh (C++ member), 516
InvertPar:::location (C++ member), 516
InvertPar:::setCoefA (C++ function), 515
InvertPar:::setCoefB (C++ function), 515, 516
InvertPar:::setCoefC (C++ function), 516
InvertPar:::setCoefD (C++ function), 516
InvertPar:::setCoefE (C++ function), 516
InvertPar:::solve (C++ function), 515
InvertParCR (C++ class), 348
InvertParCR::A (C++ member), 348
InvertParCR::B (C++ member), 348
InvertParCR::C (C++ member), 348
InvertParCR::D (C++ member), 348
InvertParCR::E (C++ member), 348
InvertParCR:::InvertParCR (C++ function), 348
InvertParCR::nsys (C++ member), 348
InvertParCR:::setCoefA (C++ function), 348
InvertParCR:::setCoefB (C++ function), 348
InvertParCR:::setCoefC (C++ function), 348
InvertParCR:::setCoefD (C++ function), 348
InvertParCR:::setCoefE (C++ function), 348
InvertParCR:::sg (C++ member), 348
InvertParCR:::solve (C++ function), 348
InvertParFactory (C++ class), 514
InvertParFactory:::create (C++ function), 514
InvertParFactory:::default_type (C++ member), 515
InvertParFactory:::ensureRegistered (C++ function), 514
InvertParFactory:::option_name (C++ member), 515
InvertParFactory:::section_name (C++ member), 515
InvertParFactory:::type_name (C++ member), 515
iopt (C++ member), 631
irfft (C++ function), 413
is_pow2 (C++ function), 690
isMultiple (C++ function), 545
iteration (C++ member), 317
J
Jacobian (C++ type), 663
jz_var (gen_fieldops.Field attribute), 462
K
KarniadakisSolver (C++ class), 518
KarniadakisSolver:::KarniadakisSolver (C++ function), 519
KarniadakisSolver::D0 (C++ member), 519
KarniadakisSolver::f0 (C++ member), 519
KarniadakisSolver::f1 (C++ member), 519
KarniadakisSolver::first_time (C++ member), 519
KarniadakisSolver::fm1 (C++ member), 519
KarniadakisSolver::fm2 (C++ member), 519
KarniadakisSolver::getCurrentTimestep (C++ function), 519
KarniadakisSolver::init (C++ function), 519
KarniadakisSolver::KarniadakisSolver (C++ function), 519
KarniadakisSolver::nlocal (C++ member), 519
KarniadakisSolver::nsteps (C++ member), 519
KarniadakisSolver::nsubsteps (C++ member), 519
KarniadakisSolver::out_timestep (C++ member), 519
KarniadakisSolver::resetInternalFields (C++ function), 519
KarniadakisSolver::run (C++ function), 519
KarniadakisSolver::s0 (C++ member), 519
KarniadakisSolver::sm1 (C++ member), 519
KarniadakisSolver::sm2 (C++ member), 519
KarniadakisSolver::take_step (C++ function), 519
KarniadakisSolver::timestep (C++ member), 519
keys() (boutdata.data.BoutOptions method), 718
keys() (boutdata.data.BoutOutputs method), 720
keys() (boututils.datafile.DataFileMethod method), 735
keys() (boututils.datafile.DataFile_HDF5 method), 737
keys() (boututils.datafile.DataFile_netCDF method), 738
KSP_BCGS (C macro), 618
KSP_BICG (C macro), 618
KSP(CG) (C macro), 618
KSP_CG (C macro), 618
KSP_CGMS (C macro), 618
KSP_CHEBYSHEV (C macro), 618
KSP_CR (C macro), 618
KSP_GMRES (C macro), 618
KSP_LSQR (C macro), 618
KSP_PREONLY (C macro), 618
KSP_RICHARDSON (C macro), 618
KSP_TCQMR (C macro), 618
KSP_TFQMR (C macro), 618
L
L2C (C++ enumerator), 323
Laplace (C++ function), 392, 397, 398
LAPLACE_BAND (C++ member), 509
LAPLACE_CYCLIC (C++ member), 509
LAPLACE_MULTIGRID (C++ member), 509
LAPLACE_NAULIN (C++ member), 509
Laplace_par (C++ function), 392, 397
LAPLACE_PDD (C++ member), 509
Laplace_perp (C++ function), 392, 397
Laplace_perpXY (C++ function), 392, 398
LAPLACE_PETSC (C++ member), 509
LAPLACE_PETSC3DAMG (C++ member), 509
LAPLACE_PETSCAMG (C++ member), 509
LAPLACE_SPT (C++ member), 509
LAPLACE_SHOOT (C++ member), 509
LAPLACE_TRI (C++ member), 509
laplace_tridag_coefs (C++ function), 508
LaplaceCyclic (C++ class), 348
LaplaceCyclic::LaplaceCyclic (C++ function), 349
LaplaceCyclic::a (C++ member), 349
LaplaceCyclic::Acoef (C++ member), 349
LaplaceCyclic::b (C++ member), 349
LaplaceCyclic::bcmplx (C++ member), 349
LaplaceCyclic::c (C++ member), 349
LaplaceCyclic::C1coef (C++ member), 349
LaplaceCyclic::C2coef (C++ member), 349
LaplaceCyclic::cr (C++ member), 350
LaplaceCyclic::Cpcoef (C++ member), 349
LaplaceCyclic::d (C++ member), 349
LaplaceCyclic::dst (C++ member), 350
LaplaceCyclic::LaplaceCyclic (C++ function), 349
LaplaceCyclic::m (C++ member), 349
LaplaceCyclic::setCoefA (C++ function), 349
LaplaceCyclic::setCoefC (C++ function), 349
LaplaceCyclic::setCoefC1 (C++ function), 349
LaplaceCyclic::setCoefC2 (C++ function), 349
LaplaceCyclic::setCoefD (C++ function), 349
LaplaceCyclic::setCoefEx (C++ function), 349
LaplaceCyclic::setCoefEz (C++ function), 349
LaplaceCyclic::solve (C++ function), 349
LaplaceCyclic::xcmplx (C++ member), 349
LaplaceCyclic::xe (C++ member), 349
LaplaceCyclic::xs (C++ member), 349
LaplaceFactory (C++ class), 510
LaplaceFactory::default_type (C++ member), 510
LaplaceFactory::option_name (C++ member), 510
LaplaceFactory::section_name (C++ member), 510
LaplaceFactory::type_name (C++ member), 510
LaplaceMultigrid (C++ class), 554
LaplaceMultigrid::LaplaceMultigrid (C++ function), 554
LaplaceMultigrid::A (C++ member), 555
LaplaceMultigrid::ratol (C++ member), 556
LaplaceMultigrid::rtol (C++ member), 556
LaplaceMultigrid::cl (C++ member), 555
LaplaceMultigrid::ct (C++ member), 555
LaplaceMultigrid::cftype (C++ member), 555
LaplaceMultigrid::comms_tagbase (C++ member), 556
LaplaceMultigrid::commX (C++ member), 556
LaplaceMultigrid::D (C++ member), 555
LaplaceMultigrid::dtol (C++ member), 556
LaplaceMultigrid::generateMatrixF (C++ function), 555
LaplaceMultigrid::kMG (C++ member), 555
LaplaceMultigrid::LaplaceMultigrid (C++ function), 554
LaplaceMultigrid::mgcount (C++ member), 555
LaplaceMultigrid::mgl (C++ member), 555
LaplaceMultigrid::mgmpi (C++ member), 556
LaplaceMultigrid::mgplag (C++ member), 555
LaplaceMultigrid::mgsm (C++ member), 555
LaplaceMultigrid::Nx_local (C++ member), 555
LaplaceMultigrid::Nx_global (C++ member), 555
LaplaceMultigrid::Nx_local (C++ member), 555
LaplaceMultigrid::Nx_global (C++ member), 555
LaplaceMultigrid::Ng_local (C++ member), 555
LaplaceMultigrid::Ng_global (C++ member), 555
LaplaceMultigrid::omega (C++ member), 556
LaplaceMultigrid::opts (C++ member), 556
LaplaceMultigrid::pcheck (C++ member), 555
LaplaceMultigrid::rtol (C++ member), 556
LaplaceMultigrid::setCoefA (C++ function), 554, 555
LaplaceMultigrid::setCoefC (C++ function), 554, 555
LaplaceMultigrid::setCoefC1 (C++ function), 554, 555
LaplaceMultigrid::setCoefC2 (C++ function), 554, 555
LaplaceMultigrid::setCoefD (C++ function), 555
LaplaceMultigrid::setCoefEx (C++ function), 555
LaplaceMultigrid::setCoefEz (C++ function), 555
LaplaceNaulin (C++ class), 561
LaplaceNaulin::-LaplaceNaulin (C++ function), 562
LaplaceNaulin::Acoef (C++ member), 562
LaplaceNaulin::atol (C++ member), 562
LaplaceNaulin::C1coef (C++ member), 562
LaplaceNaulin::C2coef (C++ member), 562
LaplaceNaulin::copy_x_boundaries (C++ function), 562
LaplaceNaulin::Dcoef (C++ member), 562
LaplaceNaulin::delp2solver (C++ member), 562
LaplaceNaulin::getMeanIterations (C++ function), 562
LaplaceNaulin::initial_underrelax_factor (C++ member), 562
LaplaceNaulin::LaplaceNaulin (C++ function), 561, 562
LaplaceNaulin::maxits (C++ member), 562
LaplaceNaulin::naulinSolver_mean_iter (C++ member), 563
LaplaceNaulin::naulinSolver_mean_underrrelax_factor (C++ member), 563
LaplaceNaulin::ncalls (C++ member), 563
LaplaceNaulin::operator= (C++ function), 562
LaplaceNaulin::resetMeanIterations (C++ function), 562
LaplaceNaulin::rtol (C++ member), 562
LaplaceNaulin::setCoefA (C++ function), 561
LaplaceNaulin::setCoefC (C++ function), 561
LaplaceNaulin::setCoefC1 (C++ function), 561
LaplaceNaulin::setCoefC2 (C++ function), 561
LaplaceNaulin::setCoefD (C++ function), 561
LaplaceNaulin::setCoefEx (C++ function), 561
LaplaceNaulin::setCoefEz (C++ function), 561
LaplaceNaulin::setGlobalFlags (C++ function), 562
LaplaceNaulin::setInnerBoundaryFlags (C++ function), 562
LaplaceNaulin::setOuterBoundaryFlags (C++ function), 562
LaplaceNaulin::solve (C++ function), 562
LaplaceNaulin::uses3DCoefs (C++ function), 561
LaplacePCapply (C++ function), 521, 618
LaplacePDD (C++ class), 596
LaplacePDD::-LaplacePDD (C++ function), 597
LaplacePDD::Acoef (C++ member), 597
LaplacePDD::Ccoef (C++ member), 597
LaplacePDD::Dcoef (C++ member), 597
LaplacePDD::finish (C++ function), 597
LaplacePDD::LaplacePDD (C++ function), 597
LaplacePDD::next (C++ function), 597
LaplacePDD::PDD_COMM_XV (C++ member), 597
LaplacePDD::PDD_COMM_Y (C++ member), 597
LaplacePDD::PDD_data (C++ class), 597
LaplacePDD::PDD_data::avec (C++ member), 597
LaplacePDD::PDD_data::ibk (C++ member), 598
LaplacePDD::PDD_data::bvec (C++ member), 598
LaplacePDD::PDD_data::cvec (C++ member), 598
LaplacePDD::PDD_data::jy (C++ member), 598
LaplacePDD::PDD_data::rcv (C++ member), 598
LaplacePDD::PDD_data::recv_handle (C++ member), 598
LaplacePDD::PDD_data::snd (C++ member), 598
LaplacePDD::setCoefA (C++ function), 597
LaplacePDD::setCoefC (C++ function), 597
LaplacePDD::setCoefD (C++ function), 597
LaplacePDD::setCoefEx (C++ function), 597
LaplacePDD::setCoefEz (C++ function), 597
LaplacePDD::solve (C++ function), 597
LaplacePDD::start (C++ function), 597
LaplacePetsc (C++ class), 618
LaplacePetsc3dAmg (C++ class), 598
LaplacePetsc3dAmg::-LaplacePetsc3dAmg (C++ function), 598
LaplacePetsc3dAmg::A (C++ member), 599
LaplacePetsc3dAmg::atol (C++ member), 600
LaplacePetsc3dAmg::c1 (C++ member), 599
LaplacePetsc3dAmg::c2 (C++ member), 599
LaplacePetsc3dAmg::chebyshev_max (C++ member), 600
LaplacePetsc3dAmg::chebyshev_min (C++ member), 600
LaplacePetsc3dAmg::D (C++ member), 599
LaplacePetsc3dAmg::direct (C++ member), 600
LaplacePetsc3dAmg::dtol (C++ member), 600
LaplacePetsc3dAmg::Ex (C++ member), 599
LaplacePetsc3dAmg::Ez (C++ member), 599
LaplacePetsc3dAmg::getIndexer (C++ function), 599
LaplacePetsc3dAmg::getMatrix3D (C++ function), 599
LaplacePetsc3dAmg::getStencil (C++ function), 600
LaplacePetsc3dAmg::gmres_max_steps (C++ member), 600

Index
LaplacePetsc3dAmg::implemented_boundary_flags (C++ member), 600
LaplacePetsc3dAmg::implemented_flags (C++ member), 600
LaplacePetsc3dAmg::indexer (C++ member), 600
LaplacePetsc3dAmg::issetC (C++ member), 599
LaplacePetsc3dAmg::issetD (C++ member), 599
LaplacePetsc3dAmg::issetE (C++ member), 599
LaplacePetsc3dAmg::ksp (C++ member), 600
LaplacePetsc3dAmg::ksptype (C++ member), 600
LaplacePetsc3dAmg::kspInitialised (C++ member), 600
LaplacePetsc3dAmg::lowerY (C++ member), 600
LaplacePetsc3dAmg::maxits (C++ member), 600
LaplacePetsc3dAmg::meshx (C++ member), 599
LaplacePetsc3dAmg::meshy (C++ member), 600
LaplacePetsc3dAmg::operator3D (C++ member), 600
LaplacePetsc3dAmg::opts (C++ member), 600
LaplacePetsc3dAmg::pctype (C++ member), 600
LaplacePetsc3dAmg::richardson_damping_factor (C++ member), 600
LaplacePetsc3dAmg::rightprec (C++ member), 600
LaplacePetsc3dAmg::rtol (C++ member), 600
LaplacePetsc3dAmg::setCoefA (C++ function), 598
LaplacePetsc3dAmg::setCoefC (C++ function), 598
LaplacePetsc3dAmg::setCoefC1 (C++ function), 598
LaplacePetsc3dAmg::setCoefC2 (C++ function), 598, 599
LaplacePetsc3dAmg::setCoefD (C++ function), 598, 599
LaplacePetsc3dAmg::setCoefEx (C++ function), 598, 599
LaplacePetsc3dAmg::solve (C++ member), 600
LaplacePetsc3dAmg::solveMatrix3D (C++ function), 599
LaplacePetsc3dAmg::upper_boundary_flags (C++ member), 599
LaplacePetsc3dAmg::upperY (C++ member), 600
LaplacePetsc3dAmg::use_precon (C++ member), 600
LaplacePetsc::A (C++ member), 621
LaplacePetsc::atol (C++ member), 621
LaplacePetsc::bs (C++ member), 621
LaplacePetsc::C1 (C++ member), 621
LaplacePetsc::C2 (C++ member), 621
LaplacePetsc::chebyshev_max (C++ member), 621
LaplacePetsc::chebyshev_min (C++ member), 621
LaplacePetsc::Coef (C++ function), 620
LaplacePetsc::comm (C++ member), 621
LaplacePetsc::D (C++ member), 621
LaplacePetsc::direct (C++ member), 621
LaplacePetsc::dtol (C++ member), 621
LaplacePetsc::Element (C++ function), 620
LaplacePetsc::Ex (C++ member), 621
LaplacePetsc::Ez (C++ member), 621
LaplacePetsc::fieldToVec (C++ function), 620
LaplacePetsc::fourth_order (C++ member), 622
LaplacePetsc::gmres_max_steps (C++ member), 621
LaplacePetsc::Iend (C++ member), 621
LaplacePetsc::implemented_boundary_flags (C++ member), 622
LaplacePetsc::implemented_flags (C++ member), 622
LaplacePetsc::issetC (C++ member), 621
LaplacePetsc::issetD (C++ member), 621
LaplacePetsc::issetE (C++ member), 621
LaplacePetsc::ksp (C++ member), 621
LaplacePetsc::ksptype (C++ member), 621
LaplacePetsc::LaplacePetsc (C++ function), 619
LaplacePetsc::localN (C++ member), 621
LaplacePetsc::maxits (C++ member), 600
LaplacePetsc::meshx (C++ member), 600
LaplacePetsc::meshy (C++ member), 600
LaplaceXZ::LaplaceXZ (C++ function), 526
LaplaceXZ::localmesh (C++ member), 527
LaplaceXZ::location (C++ member), 527
LaplaceXZ::setCoefs (C++ function), 526
LaplaceXZ::solve (C++ function), 526
LaplaceXZcyclic (C++ class), 524
LaplaceXZcyclic::~LaplaceXZcyclic (C++ function), 524
LaplaceXZcyclic::acoef (C++ member), 524
LaplaceXZcyclic::bcoef (C++ member), 524
LaplaceXZcyclic::ccoef (C++ member), 524
LaplaceXZcyclic::cr (C++ member), 524
LaplaceXZcyclic::inner_boundary_flags (C++ member), 524
LaplaceXZcyclic::k1d_2 (C++ member), 524
LaplaceXZcyclic::nloc (C++ member), 524
LaplaceXZcyclic::nsys (C++ member), 524
LaplaceXZcyclic::outer_boundary_flags (C++ member), 524
LaplaceXZcyclic::rhcsmplx (C++ member), 524
LaplaceXZcyclic::setCoefs (C++ function), 524
LaplaceXZcyclic::solve (C++ function), 524
LaplaceXZcyclic::xcmplx (C++ member), 524
LaplaceXZcyclic::xloc (C++ member), 524
LaplaceXZFactory (C++ class), 526
LaplaceXZFactory::create (C++ function), 526
LaplaceXZFactory::default_type (C++ member), 526
LaplaceXZFactory::ensureRegistered (C++ function), 526
LaplaceXZFactory::option_name (C++ member), 526
LaplaceXZFactory::section_name (C++ member), 526
LaplaceXZFactory::type_name (C++ member), 526
LaplaceXZpetsc (C++ class), 524
LaplaceXZpetsc::~LaplaceXZpetsc (C++ function), 525
LaplaceXZpetsc::bs (C++ member), 525
LaplaceXZpetsc::coefs_set (C++ member), 525
LaplaceXZpetsc::implemented_boundary_flags (C++ member), 525
LaplaceXZpetsc::inner_boundary_flags (C++ member), 525
LaplaceXZpetsc::Lib (C++ member), 525
LaplaceXZpetsc::outer_boundary_flags (C++ member), 525
LaplaceXZpetsc::reuse_count (C++ member), 525
LaplaceXZpetsc::reuse_limit (C++ member), 525
LaplaceXZpetsc::setCoefs (C++ function), 525
LaplaceXZpetsc::slice (C++ member), 525
LaplaceXZpetsc::solve (C++ function), 525
LaplaceXZpetsc::xs (C++ member), 525
LaplaceXZpetsc::YSlice (C++ class), 525
LaplaceXZpetsc::YSlice::ksp (C++ member), 525
LaplaceXZpetsc::YSlice::MatA (C++ member), 525
LaplaceXZpetsc::YSlice::MatP (C++ member), 525
LaplaceXZpetsc::YSlice::yindex (C++ member), 525
Laplacian (C++ class), 510
Laplacian::~Laplacian (C++ function), 510
Laplacian::all_terms (C++ member), 513
Laplacian::async_send (C++ member), 513
Laplacian::cleanup (C++ function), 512
Laplacian::coords (C++ member), 514
Laplacian::create (C++ function), 512
Laplacian::defaultInstance (C++ function), 512
Laplacian::extra_yguards_lower (C++ member), 513
Laplacian::extra_yguards_upper (C++ member), 513
Laplacian::global_flags (C++ member), 514
Laplacian::include_yguards (C++ member), 513
Laplacian::inner_boundary_flags (C++ member), 514
Laplacian::instance (C++ member), 514
Laplacian::location (C++ member), 510
Laplacian::localmesh (C++ member), 514
Laplacian::low_mem (C++ member), 513
Laplacian::maxmode (C++ member), 513
Laplacian::nonuniform (C++ member), 513
Laplacian::outer_boundary_flags (C++ member), 514
Laplacian::setCoefA (C++ function), 510, 511
Laplacian::setCoefC (C++ function), 511
Laplacian::setCoefC1 (C++ function), 511
Laplacian::setCoefC2 (C++ function), 511
Laplacian::setCoefD (C++ function), 511
Mesh::DownXSplitIndex (C++ function), 537
Mesh::fft_deriv_filter (C++ member), 544
Mesh::firstX (C++ function), 535
Mesh::firstY (C++ function), 537
Mesh::fromFieldAligned (C++ function), 541
Mesh::get (C++ function), 531, 532
Mesh::getAllowedStaggerLoc (C++ function),
540
Mesh::getBoundaries (C++ function), 538
Mesh::getBoundariesPar (C++ function), 538
Mesh::getCoordinates (C++ function), 540
Mesh::getCoordinatesSmart (C++ function),
540
Mesh::getGlobalXIndex (C++ function), 539
Mesh::getGlobalXIndexNoBoundaries (C++ function), 539
Mesh::getGlobalYIndex (C++ function), 539
Mesh::getGlobalYIndexNoBoundaries (C++ function), 539
Mesh::getGlobalZIndex (C++ function), 539
Mesh::getGlobalZIndexNoBoundaries (C++ function), 539
Mesh::getLocalXIndex (C++ function), 540
Mesh::getLocalXIndexNoBoundaries (C++ function), 540
Mesh::getLocalYIndex (C++ function), 539
Mesh::getLocalYIndexNoBoundaries (C++ function), 539
Mesh::getLocalZIndex (C++ function), 540
Mesh::getLocalZIndexNoBoundaries (C++ function), 540
Mesh::getMpi (C++ function), 536
Mesh::getNguard (C++ function), 540
Mesh::getNpoints (C++ function), 540
Mesh::getNXPE (C++ function), 535
Mesh::getNPYE (C++ function), 535
Mesh::getParallelTransform (C++ function),
542
Mesh::getRegion (C++ function), 542, 543
Mesh::getRegion2D (C++ function), 542
Mesh::getRegion3D (C++ function), 542
Mesh::getRegionPerp (C++ function), 542
Mesh::getStagger (C++ function), 540
Mesh::getXcomm (C++ function), 536
Mesh::getXProcIndex (C++ function), 535
Mesh::getYcomm (C++ function), 536
Mesh::getYProcIndex (C++ function), 535
Mesh::GlobalNx (C++ member), 543
Mesh::GlobalNxNoBoundaries (C++ member),
543
Mesh::GlobalNy (C++ member), 543
Mesh::GlobalNyNoBoundaries (C++ member),
543
Mesh::GlobalNz (C++ member), 543
Mesh::GlobalNzNoBoundaries (C++ member),
543
Mesh::GlobalX (C++ function), 538
Mesh::GlobalY (C++ function), 538
Mesh::hasBndryLowerY (C++ function), 538
Mesh::hasBndryUpperY (C++ function), 538
Mesh::hasBranchCutLower (C++ function), 536
Mesh::hasBranchCutUpper (C++ function), 536
Mesh::hasRegion2D (C++ function), 542
Mesh::hasRegion3D (C++ function), 542
Mesh::hasRegionPerp (C++ function), 542
Mesh::IncIntShear (C++ member), 544
Mesh::include_corner_cells (C++ member),
544
Mesh::ind2Dto3D (C++ function), 542
Mesh::ind3Dto2D (C++ function), 542
Mesh::ind3DtoPerp (C++ function), 542
Mesh::indexD2DX2 (C++ function), 540
Mesh::indexD2DY2 (C++ function), 540
Mesh::indexD2DZ2 (C++ function), 541
Mesh::indexD4DX4 (C++ function), 540
Mesh::indexD4DY4 (C++ function), 540
Mesh::indexD4DZ4 (C++ function), 541
Mesh::indexDDX (C++ function), 540
Mesh::indexDDY (C++ function), 540
Mesh::indexDDZ (C++ function), 541
Mesh::indexFDDX (C++ function), 541
Mesh::indexFDDY (C++ function), 541
Mesh::indexFDDZ (C++ function), 541
Mesh::indexLookup3Dto2D (C++ member), 545
Mesh::indexVDDX (C++ function), 541
Mesh::indexVDDY (C++ function), 541
Mesh::indexVDDZ (C++ function), 541
Mesh::irecvXIn (C++ function), 536
Mesh::irecvXOut (C++ function), 535
Mesh::irecvYInIndest (C++ function), 537
Mesh::irecvYInOutdest (C++ function), 538
Mesh::irecvYOutIndest (C++ function), 537
Mesh::irecvYOutOutdest (C++ function), 537
Mesh::isDataSourceGridFile (C++ function),
533
Mesh::iterateBndryLowerInnerY (C++ function), 538
Mesh::iterateBndryLowerOuterY (C++ function), 538
Mesh::iterateBndryLowerY (C++ function), 538
Mesh::iterateBndryUpperInnerY (C++ function), 538
Mesh::iterateBndryUpperOuterY (C++ function), 538
Mesh::iterateBndryUpperY (C++ function), 538
Mesh::lastX (C++ function), 535
Mesh::lastY (C++ function), 537
Mesh::lastX (C++ function), 537
Mesh::lastY (C++ function), 537
Mesh::lastX (C++ function), 537
Mesh::lastY (C++ function), 537

Index
Mesh::load (C++ function), 531
Mesh::localNumCells2D (C++ member), 545
Mesh::load (C++ member), 545
Mesh::globalNumCells3D (C++ member), 545
Mesh::source (C++ function), 545
Mesh::LocalNy (C++ member), 543
Mesh::LocalNz (C++ member), 543
Mesh::mpi (C++ function), 530
Mesh::map3Dto2D (C++ function), 542
Mesh::maxregionblocksize (C++ member), 544
Mesh::mesh (C++ function), 530
Mesh::setParallelTransform (C++ function), 545
Mesh::sendYOutIndest (C++ function), 544
Mesh::sendYInIndest (C++ function), 544
Mesh::sendY (C++ function), 544
Mesh::sendXOut (C++ function), 544
Mesh::sendXIn (C++ function), 544
Mesh::sendX (C++ function), 544
Mesh::sendToProc (C++ function), 534
Mesh::receiveFromProc (C++ function), 534
Mesh::recalculateStaggeredCoordinates (C++ function), 540
Mesh::regionMap2D (C++ member), 545
Mesh::regionMap3D (C++ member), 545
Mesh::regionMapPerp (C++ member), 545
Mesh::send (C++ function), 534
Mesh::sendToProc (C++ function), 534
Mesh::sendX (C++ function), 534
Mesh::sendXIn (C++ function), 535
Mesh::sendXOut (C++ function), 535
Mesh::sendY (C++ function), 534
Mesh::sendYOutIndest (C++ function), 537
Mesh::sendYOutIndest (C++ function), 537
Mesh::sendYOutIndest (C++ function), 537
Mesh::setParallelTransform (C++ function), 542
Mesh::smoothSeparatrix (C++ function), 538
Mesh::source (C++ member), 544
Mesh::sourceHasVar (C++ function), 533
Mesh::sourceHasXBoundaryGuards (C++ function), 533
Mesh::sourceHasYBoundaryGuards (C++ function), 533
Mesh::sourceHasXBoundaryGuards (C++ function), 533
Mesh::sourceHasYBoundaryGuards (C++ function), 533
Mesh::StaggerGrids (C++ member), 543
Mesh::toFieldAligned (C++ function), 541
Mesh::UpXSplitIndex (C++ function), 537
Mesh::wait (C++ function), 534
Mesh::xend (C++ member), 543
Mesh::xGLOBAL (C++ function), 539
Mesh::xLOCAL (C++ function), 539
Mesh::xstart (C++ member), 543
Mesh::yend (C++ member), 543
Mesh::YGLOBAL (C++ function), 539
Mesh::YLOCAL (C++ function), 539
Mesh::ySize (C++ function), 537
Mesh::ystart (C++ member), 543
Mesh::xend (C++ member), 543
Mesh::zstart (C++ member), 543
MeshFactory (C++ class), 530
MeshFactory::create (C++ function), 530
MeshFactory::default_type (C++ member), 530
MeshFactory::option_name (C++ member), 530
MeshFactory::section_name (C++ member), 530
MeshFactory::type_name (C++ member), 494
metaData (C++ class), 494
metaData::derivType (C++ member), 495
metaData::key (C++ member), 495
metaData::nGuards (C++ member), 495
metric () (zoidberg.grid.Grid method), 757
metric () (zoidberg.poloidal_grid.RectangularPoloidalGrid method), 761
metric () (zoidberg.poloidal_grid.StructuredPoloidalGrid method), 762
min (C++ function), 433
MINMOD (C++ function), 690
mixed_base_ind_var (gen_fields.Field attribute), 690
mode_structure () (in module boututils.mode_structure), 741
moment_xyzt () (in module boututils.moment_xxyz), 741
Monitor (C++ class), 545
Monitor::~Monitor (C++ function), 545
Monitor::call (C++ function), 546
Monitor::cleanup (C++ function), 546
Monitor::getTimeStep (C++ function), 546
Monitor::is_added (C++ function), 546
Monitor::Monitor (C++ function), 545
Monitor::period (C++ member), 546
Monitor::setTimeStep (C++ function), 546
Monitor::timestep (C++ member), 546
MonitorPosition (C++ enum), 663
monitorWrapper (C++ function), 653
mpiWrapper (C++ class), 547
mpiWrapper:~mpiWrapper (C++ function), 547
mpiWrapper::MPI_Abort (C++ function), 547
mpiWrapper::MPI_Allreduce (C++ function), 547
mpiWrapper::MPI_Barrier (C++ function), 547
BOUT++ Documentation, Release 5.0.0-alpha

MpiWrapper::MPI_Comm_create (C++ function), 547
MpiWrapper::MPI_Comm_dup (C++ function), 547
MpiWrapper::MPI_Comm_free (C++ function), 548
MpiWrapper::MPI_Group_union (C++ function), 548
MpiWrapper::MPI_Group_range_incl (C++ function), 548
MpiWrapper::MPI_Comm_split (C++ function), 548
MpiWrapper::MPI_Comm_size (C++ function), 548
MpiWrapper::MPI_Comm_rank (C++ function), 548
MpiWrapper::MPI_Comm_group (C++ function), 548
MpiWrapper::MPI_Comm_dup (C++ function), 548
MpiWrapper::MPI_Comm_create (C++ function), 548
msg_stack (C++ member), 549
MsgStack (C++ class), 549
MsgStack::dump (C++ function), 550
MsgStack::clear (C++ function), 550
MsgStack::stack (C++ member), 550
MsgStack::position (C++ function), 550
MsgStack::push (C++ function), 550
MsgStack::setPoint (C++ function), 550
MsgStack::pop (C++ function), 550
MsgStack::point (C++ member), 551
MsgStackItem (C++ class), 550
MsgStackItem::~MsgStackItem (C++ function), 551
MsgStackItem::point (C++ member), 551
mul (C++ function), 401
Multigrid1DP (C++ class), 554
Multigrid1DP::~Multigrid1DP (C++ function), 554
Multigrid1DP::comm2D (C++ member), 554
Multigrid1DP::convertMatrixF2D (C++ function), 554
Multigrid1DP::convertMatrixFS (C++ function), 554
Multigrid1DP::kflag (C++ member), 554
Multigrid1DP::lowestSolver (C++ function), 554
Multigrid1DP::Multigrid1DP (C++ function), 554
Multigrid1DP::rMG (C++ member), 554
Multigrid1DP::setMultigridC (C++ function), 554
Multigrid1DP::setPcheck (C++ function), 554
Multigrid1DP::setValueS (C++ function), 554
Multigrid1DP::sMG (C++ member), 554
Multigrid2DPf1D (C++ class), 553
Multigrid2DPf1D::~Multigrid2DPf1D (C++ function), 553
Multigrid2DPf1D::convertMatrixFS (C++ function), 553
Multigrid2DPf1D::kflag (C++ member), 553
Multigrid2DPf1D::lowestSolver (C++ function), 553
Multigrid2DPf1D::Multigrid2DPf1D (C++ function), 553
Multigrid2DPf1D::setMultigridC (C++ function), 553
Multigrid2DPf1D::setPcheck (C++ function), 553
Multigrid2DPf1D::setValueS (C++ function), 553
Multigrid2DPf1D::sMG (C++ member), 554
MultigridAlg (C++ class), 551
MultigridAlg::~MultigridAlg (C++ function), 551
MultigridAlg::atol (C++ member), 552
MultigridAlg::cftype (C++ member), 552
MultigridAlg::commMG (C++ member), 553
MultigridAlg::communications (C++ function), 552
MultigridAlg::cycleMG (C++ function), 552
MultigridAlg::dtol (C++ member), 552
MultigridAlg::getSolution (C++ function), 551
MultigridAlg::gnx (C++ member), 552
MultigridAlg::gnz (C++ member), 552
MultigridAlg::lnx (C++ member), 552
MultigridAlg::lnz (C++ member), 552
MultigridAlg::lowestSolver (C++ function), 552
MultigridAlg::lowestSolver (C++ function), 551
MultigridAlg::Mul (C++ function), 551
MultigridAlg::matmg (C++ member), 552
MultigridAlg::mglevel (C++ member), 552
MultigridAlg::mgplag (C++ member), 552
MultigridAlg::mgsm (C++ member), 552
MultigridAlg::multiAVec (C++ function), 552
MultigridAlg::MultigridAlg (C++ function), 552
MultigridAlg::numP (C++ member), 552
MultigridAlg::omega (C++ member), 552
MultigridAlg::pcheck (C++ member), 552
MultigridAlg::pGMRES (C++ function), 552
MultigridAlg::projection (C++ function), 552
MultigridAlg::prolongation (C++ function), 552
MultigridAlg::residualVec (C++ function), 552
MultigridAlg::rProcI (C++ member), 552
MultigridAlg::rtol (C++ member), 552
MultigridAlg::setMatrixC (C++ function), 552
MultigridAlg::setMultigridC (C++ function), 551
MultigridAlg::smoothings (C++ function), 552
MultigridAlg::solveMG (C++ function), 552
MultigridAlg::vectorProd (C++ function), 552
MultigridAlg::xNP (C++ member), 552
MultigridAlg::xProcI (C++ member), 553
MultigridAlg::xProcM (C++ member), 553
MultigridAlg::xProcP (C++ member), 553
MultigridAlg::zNP (C++ member), 552
MultigridAlg::zProcI (C++ member), 553
MultigridAlg::zProcM (C++ member), 553
MultigridAlg::zProcP (C++ member), 553
MultigridSerial::MultigridSerial (C++ class), 553
MultigridSerial::convertMatrixF (C++ function), 553
MultigridSerial::convertMatrixF (C++ function), 553
multiostream (C++ class), 557
multiostream::add (C++ function), 557
multiostream::multiostream (C++ function), 557
multiostream::remove (C++ function), 557
multiostream::remove (C++ function), 557
multiostream::streams_ (C++ member), 557
multiostream::xspu2n (C++ function), 556
multioutbuf_init (C++ class), 557
multioutbuf_init::buf (C++ function), 557
multioutbuf_init::buf_ (C++ member), 557
multioutbuf<CHAR_TYPE>
  traits>::stream_container (C++ type), 556
Multiout (C++ class), 405
Multiout::apply (C++ function), 405

N
name (gen_fieldops attribute), 461
name (gen_fieldops.Field attribute), 462
NcFormat (C++ class), 563
NcFormat::NcFormat (C++ function), 563
NcFormat::addVarBoutReal (C++ function), 564
NcFormat::addVarField2D (C++ function), 564
NcFormat::addVarField3D (C++ function), 564
NcFormat::addVarFieldPerp (C++ function), 564
NcFormat::addVarInt (C++ function), 564
NcFormat::checkName (C++ function), 566
NcFormat::close (C++ function), 564
NcFormat::dataFile (C++ member), 566
NcFormat::default_rec (C++ member), 567
NcFormat::dimList (C++ member), 566
NcFormat::filename (C++ function), 564
NcFormat::flush (C++ function), 564
NcFormat::fname (C++ member), 566
NcFormat::getAttribute (C++ function), 565,
      566
NcFormat::getSize (C++ function), 564
NcFormat::is_valid (C++ function), 563
NcFormat::lowPrecision (C++ member), 566
NcFormat::NcFormat (C++ function), 563
NcFormat::open (C++ function), 563
NcFormat::openw (C++ function), 563
NcFormat::read (C++ function), 564
NcFormat::read_perp (C++ function), 564
NcFormat::read_rec (C++ function), 564
NcFormat::read_rec_perp (C++ function), 564
NcFormat::rec_nr (C++ member), 567
NcFormat::recDimList (C++ member), 566
NcFormat::setAttribute (C++ function), 565
NcFormat::setGlobalOrigin (C++ function),
      564
NcFormat::setLocalOrigin (C++ function), 564
NcFormat::setLowPrecision (C++ function),
      565
NcFormat::setRecord (C++ function), 564
NcFormat::t0 (C++ member), 567
NcFormat::tDim (C++ member), 566
NcFormat::write (C++ function), 564
NcFormat::write_perp (C++ function), 564
NcFormat::write_rec (C++ function), 564, 565
Operator::operator+ (C++ function), 567
Operator::operator* (C++ function), 567
Operator::operator!= (C++ function), 567
openFile () (boututils.datafile.DataFile method), 735
ndims () (boututils.datafile.DataFile_HDF5 method), 737

doFilter (C++ function), 658, 660
doFilter_x (C++ function), 658, 659
doFilter_y (C++ function), 658, 659
doFilter_z (C++ function), 658, 660
nobdry (C++ enumerator), 322
NoBoundary (class in zoidberg.boundary), 748
Normal (class in boutdata.restart), 725
Normal_c (C++ enumerator), 322
Normal_c (C++ enumerator), 322
Normal_c (C++ enumerator), 322
NumberOfPoloidalGrids () (zoidberg.grid.Grid method), 757

offset (C++ function), 636
Offset2D (C++ type), 568
Offset3D (C++ type), 568
OffsetPerp (C++ type), 568
ONE (C macro), 290, 345, 484
open () (boututils.datafile.DataFile method), 735
open () (boututils.datafile.DataFile_HDF5 method), 737
open () (boututils.datafile.DataFile_netCDF method), 739
openFile () (boututils.geqdsk.Geqdsk method), 740
operator= (C++ function), 568, 635
operator* (C++ function), 416, 422, 423, 454, 462–
464, 605, 702, 706, 707
operator+ (C++ function), 416, 422, 423, 432, 451,
454, 463, 464, 569, 635, 636
operator- (C++ function), 415, 416, 421–423, 453,
454, 463, 464, 569, 635
operator/ (C++ function), 416, 422, 423, 454, 462–
464
operator== (C++ function), 416, 417, 422, 425, 568,
635, 671
operator> (C++ function), 635
operator>= (C++ function), 635
operator<= (C++ function), 635
operator<< (C++ function), 412, 416, 417, 422, 425,
494, 583, 636
OperatorStencil (C++ class), 569
OperatorStencil::add (C++ function), 570
OperatorStencil::begin (C++ function), 570
OperatorStencil::cbegin (C++ function), 570
OperatorStencil::end (C++ function), 570
OperatorStencil::crend (C++ function), 571
OperatorStencil::cend (C++ function), 571
OperatorStencil::end (C++ function), 570
OperatorStencil::getIndicesWithStencilIncluding
(C++ function), 570
OperatorStencil::getNumParts (C++ function), 570
OperatorStencil::getStencilNumber (C++ function), 571
OperatorStencil::getStencilPart (C++ function), 570
OperatorStencil::getStencilSize (C++ function), 570
OperatorStencil::rbegin (C++ function), 571
OperatorStencil::rend (C++ function), 571
OperatorStencil::Stencil (C++ class), 571
OperatorStencil::stencils (C++ member), 571
OperatorStencil<T>::const_iterator (C++ type), 570
OperatorStencil<T>::const_reverse_iterator
(C++ type), 570
OperatorStencil<T>::iterator (C++ type), 570
OperatorStencilt<T>::offset (C++ type), 570
OperatorStencilt<T>::reverse_iterator
(C++ type), 570
OperatorStencilt<T>::Stencil::part (C++ member), 571
OperatorStencilt<T>::Stencil::test (C++ member), 571
OperatorStencilt<T>::stencil_part (C++ type), 570
OperatorStencilt<T>::stencil_test (C++ type), 570

OPT_SIZE (C macro), 602, 654
OPTION (C macro), 572
OPTION2 (C macro), 572
OPTION3 (C macro), 572
OPTION4 (C macro), 572
OPTION5 (C macro), 572
OPTION6 (C macro), 572
OptionINI (C++ class), 580
OptionINI::getNextLine (C++ function), 580
OptionINI::parse (C++ function), 580
OptionINI::read (C++ function), 580
OptionINI::write (C++ function), 580
OptionINI::writeSection (C++ function), 580
OptionParser (C++ class), 571
OptionParser::~OptionParser (C++ function), 571
OptionParser::OptionParser (C++ function), 571
OptionParser::read (C++ function), 571
OptionParser::write (C++ function), 571
Options (C++ class), 572
Options::_set (C++ function), 578
Options::~Options (C++ function), 574
Options::as (C++ function), 575, 577, 578
Options::assign (C++ function), 572
Options::attributes (C++ member), 578
Options::AttributeType (C++ class), 579
Options::AttributeType::~AttributeType (C++ function), 579
Options::AttributeType::as (C++ function), 580
Options::AttributeType::AttributeType (C++ function), 579
Options::AttributeType::Base (C++ type), 579
Options::AttributeType::operator T (C++ function), 579
Options::AttributeType::operator= (C++ function), 579
Options::attributes (C++ member), 579
Options::cleanup (C++ function), 578
Options::dir (C++ function), 578
Options::force (C++ function), 575
Options::forceSet (C++ function), 576
Options::full_name (C++ member), 579
Options::get (C++ function), 576
Options::getChildren (C++ function), 577
Options::getParent (C++ function), 576
Options::getRoot (C++ function), 576
Options::isSection (C++ function), 576
Options::isSection (C++ member), 579
Options::is_value (C++ member), 579
Options::isSection (C++ function), 577
Options::isSet (C++ function), 575, 576
Options::isValue (C++ function), 577
Options::name (C++ function), 577
Options::operator T (C++ function), 575
Options::operator= (C++ function), 575
Options::operator= (C++ function), 576
Options::operator< (C++ function), 576
Options::operator[] (C++ function), 574, 575
Options::Options (C++ function), 574
Options::OptionValue (C++ class), 580
Options::OptionValue::OptionValue (C++ function), 580
Options::OptionValue::source (C++ member), 580
Options::OptionValue::used (C++ member), 580
Options::OptionValue::value (C++ member), 580
Options::overrideDefault (C++ function), 576
Options::parent (C++ function), 576
Options::parent_instance (C++ member), 579
Options::printUnused (C++ function), 577
Options::root (C++ function), 578
Options::root_instance (C++ member), 579
Options::set (C++ function), 576
Options::similar (C++ function), 578
Options::str (C++ function), 576
Options::subsections (C++ function), 577
Options::value (C++ member), 578
Options::value_used (C++ member), 579
Options::values (C++ function), 577
Options::ValuesMap (C++ type), 574
Options::ValueType (C++ type), 574
Options::valueUsed (C++ function), 577
Options::withDefault (C++ function), 575, 576
OptionsReader (C++ class), 581
OptionsReader::cleanup (C++ function), 582
OptionsReader::getInstance (C++ function), 582
OptionsReader::instance (C++ member), 582
OptionsReader::parseCommandLine (C++ function), 582
OptionsReader::read (C++ function), 582
OptionsReader::write (C++ function), 582
OUT_SENT_DOWN (C++ member), 329
OUT_SENT_IN (C++ member), 329
OUT_SENT_UP (C++ member), 329
Output (C++ class), 584
output (C++ member), 584
Output::__Tr (C++ type), 585
Output::~Output (C++ function), 584
Output::add (C++ function), 585
Output::close (C++ function), 585
Output::disable (C++ function), 584
Output::enable (C++ function), 584
Output::enabled (C++ member), 585
Output::file (C++ member), 585
Output::getBase (C++ function), 585
Output::getInstance (C++ function), 585
Output::is_enabled (C++ function), 585
Output::multiOutBuf_init (C++ type), 585
Output::open (C++ function), 584
Output::Output (C++ function), 584
Output::print (C++ function), 585
Output::remove (C++ function), 585
Output::write (C++ function), 585
output_debug (C++ member), 583, 584
output_error (C++ member), 583, 584
output_info (C++ member), 583, 584
output_progress (C++ member), 583, 584
output_verbose (C++ member), 584
output_warn (C++ member), 583, 584
outside() (zoidberg.boundary.NoBoundary method), 748
outside() (zoidberg.boundary.PolygonBoundaryXZ method), 749
outside() (zoidberg.boundary.RectangularBoundaryXZ method), 749

P
par_bwrd (C++ enumerator), 301
par_fd (C++ enumerator), 301
parallel_slice_field_name () (in module zoidberg.zoidberg), 768
ParallelTransform (C++ class), 591
ParallelTransform::~~ParallelTransform (C++ function), 592
ParallelTransform::calcParallelSlices (C++ function), 592
ParallelTransform::calcYupYdown (C++ function), 592
ParallelTransform::canToFromFieldAligned (C++ function), 592
ParallelTransform::checkInputGrid (C++ function), 593
ParallelTransform::fromFieldAligned (C++ function), 592
ParallelTransform::getWeightsForYApproximation (C++ function), 592
ParallelTransform::getWeightsForYUpApproximation (C++ function), 592
ParallelTransform::getWeightsForYDownApproximation (C++ function), 592
ParallelTransform::getWeightsForYUpApproximation (C++ function), 592
ParallelTransform::integrateParallelSlices (C++ function), 592
ParallelTransform::integrateYupYdown (C++ function), 592
ParallelTransform::mesh (C++ member), 592
ParallelTransform::options (C++ member), 593
ParallelTransform::outputVars (C++ function), 592
ParallelTransform::ParallelTransform (C++ function), 592
ParallelTransform::PositionsAndWeights (C++ class), 593
ParallelTransform::PositionsAndWeights::j (C++ member), 593
ParallelTransform::PositionsAndWeights::k (C++ member), 593
ParallelTransform::PositionsAndWeights::weight (C++ member), 593
ParallelTransform::requiresTwistShift (C++ function), 592
ParallelTransform::toFieldAligned (C++ function), 592
ParallelTransformIdentity (C++ class), 593
ParallelTransformIdentity::calcParallelSlices (C++ function), 593
ParallelTransformIdentity::canToFromFieldAligned (C++ function), 593
ParallelTransformIdentity::checkInputGrid (C++ function), 594
ParallelTransformIdentity::fromFieldAligned (C++ function), 593
ParallelTransformIdentity::getWeightsForYApproximation (C++ function), 593
ParallelTransformIdentity::ParallelTransformIdentity (C++ function), 593
ParallelTransformIdentity::requiresTwistShift (C++ function), 593
PARDERIVCYCLIC (C++ member), 514
ParseException (C++ class), 410
ParseException::~ParseException (C++ function), 593
ParseException::message (C++ member), 410
ParseException::ParseException (C++ function), 410
ParseException::message (C++ function), 410
ParseException::ParseException (C++ function), 410
ParseException::what (C++ function), 410
PetscLib::cleanup (C++ function), 518
PetscLib::count (C++ member), 623
PetscLib::help (C++ member), 623
PetscLib::options_prefix (C++ member), 623
PetscLib::pargc (C++ member), 623
PetscLib::parcv (C++ member), 623
PetscLib::PetscLib (C++ function), 622
PetscLib::PetscLib::PetscLib (C++ function), 622
PetscLib::setArgs (C++ function), 622
PetscLib::setOptionsFromInputFile (C++ function), 622
PetscLib::setPetscOptions (C++ function), 622
PetscLib::USER_EVENT (C++ member), 623
PetscMatrix (C++ class), 613, 615
PetscMatrix::assemble (C++ function), 614, 616

Index 819
PetscMatrix::destroy (C++ function), 614, 616
PetscMatrix::Element (C++ class), 614, 617
PetscMatrix::get (C++ function), 614, 616
PetscMatrix::indexConverter (C++ member), 614, 616
PetscMatrix::initialised (C++ member), 614, 617
PetscMatrix::lib (C++ member), 614, 616
PetscMatrix::matrix (C++ member), 614, 616
PetscMatrix::MatrixDeleter (C++ class), 615, 618
PetscMatrix::operator() (C++ function), 614, 616
PetscMatrix::operator= (C++ function), 613, 616
PetscMatrix::partialAssemble (C++ function), 614, 616
PetscMatrix::PetscMatrix (C++ function), 613, 616
PetscMatrix::pt (C++ member), 614, 616
PetscMatrix::ydown (C++ function), 614, 616
PetscMatrix::ynext (C++ function), 614, 616
PetscMatrix::yoffset (C++ member), 614, 617
PetscMatrix::yup (C++ function), 614, 616
PetscMatrix<T>::Element::Element (C++ function), 615, 616
PetscMatrix<T>::Element::operator BoutReal (C++ function), 615, 617
PetscMatrix<T>::Element::operator= (C++ function), 615, 617
PetscMatrix<T>::Element::operator=(C++ function), 615, 617
PetscMatrix<T>::Element::petscCol (C++ member), 615, 618
PetscMatrix<T>::Element::petscMatrix (C++ member), 615, 617
PetscMatrix<T>::Element::petscRow (C++ member), 615, 617
PetscMatrix<T>::Element::positions (C++ member), 615, 618
PetscMatrix<T>::Element::setValues (C++ function), 615, 617
PetscMatrix<T>::Element::value (C++ member), 615, 618
PetscMatrix<T>::Element::weights (C++ member), 615, 618
PetscMatrix<T>::ind_type (C++ type), 613, 616
PetscMatrix<T>::MatrixDeleter::operator (C++ function), 615, 618
PetscMonitor (C++ function), 601, 602, 604
PetscSNESMonitor (C++ function), 601, 602, 604
PetscSolver (C++ class), 602
PetscSolver::~PetscSolver (C++ function), 602
PetscSolver::adaptive (C++ member), 604
PetscSolver::bout_snes_time (C++ member), 604
PetscSolver::diagnose (C++ member), 604
PetscSolver::init (C++ function), 602
PetscSolver::init_event (C++ member), 603
PetscSolver::interpolate (C++ member), 604
PetscSolver::J (C++ member), 603
PetscSolver::jac (C++ function), 603
PetscSolver::jacfunc (C++ member), 603
PetscSolver::Jmf (C++ member), 603
PetscSolver::lib (C++ member), 603
PetscSolver::loop_event (C++ member), 603
PetscSolver::matfddcoloring (C++ member), 603
PetscSolver::next_output (C++ member), 604
PetscSolver::nout (C++ member), 603
PetscSolver::output_flag (C++ member), 604
PetscSolver::setJacobian (C++ function), 602
PetscSolver::setPrecon (C++ function), 602
PetscSolver::shift (C++ member), 603
PetscSolver::snes_list (C++ member), 604
PetscSolver::solver_event (C++ member), 603
PetscSolver::state (C++ member), 603
PetscSolver::ts (C++ member), 603
PetscSolver::ts_time (C++ member), 603
PetscSolver::tstep (C++ member), 603
PetscSolver::u (C++ member), 603
PetscVector (C++ class), 609, 611
PetscVector::assemble (C++ function), 610, 612
PetscVector::destroy (C++ function), 610, 612
PetscVector::Element (C++ class), 610, 612
PetscVector::get (C++ function), 610, 612
PetscVector::indexConverter (C++ member), 610, 612
PetscVector::initialised (C++ member), 610, 612
PetscVector::location (C++ member), 610, 612
PetscVector::operator= (C++ function), 609, 612
PetscVector::operator() (C++ function), 609, 612

BOUT++ Documentation, Release 5.0.0-alpha

Index
PetscVector::PetscVector (C++ function), 609, 611
PetscVector::toField (C++ function), 610, 612
PetscVector::vector (C++ member), 610, 612
PetscVector::VectorDeleter (C++ class), 611, 613
PetscVector<T>::Element::Element (C++ function), 610, 612
PetscVector<T>::Element::operator= (C++ function), 609, 611
PetscVector<T>::Element::operator+= (C++ function), 611, 613
PetscVector<T>::Element::operator=(C++ member), 611, 613
PetscVector<T>::Element::petscIndex (C++ member), 611, 613
PetscVector<T>::Element::petscVector (C++ member), 611, 613
PetscVector<T>::Element::value (C++ member), 611, 613
PetscVector<T>::ind_type (C++ type), 609, 611
PetscVector<T>::VectorDeleter::operator= (C++ function), 610, 613
PhysicsModel::bout_jacobian (C++ member), 625
PhysicsModel::bout_solve (C++ function), 626
PhysicsModel::convective (C++ function), 625
PhysicsModel::diffusive (C++ function), 625
PhysicsModel::dump (C++ member), 625
PhysicsModel::has_jacobian (C++ function), 625
PhysicsModel::has_precon (C++ function), 625
PhysicsModel::init (C++ function), 625
PhysicsModel::initialise (C++ function), 624
PhysicsModel::initialised (C++ member), 627
PhysicsModel::jacobianfunc (C++ type), 624
PhysicsModel::mesh (C++ member), 625
PhysicsModel::modelMonitor (C++ member), 626
PhysicsModel::outputMonitor (C++ function), 626
PhysicsModel::PhysicsModel (C++ function), 624
PhysicsModel::PhysicsModelMonitor (C++ class), 627
PhysicsModel::PhysicsModelMonitor::call (C++ function), 627
PhysicsModel::PhysicsModelMonitor::model (C++ member), 627
PhysicsModel::PhysicsModelMonitor::PhysicsModelMonitor (C++ function), 627
PhysicsModel::postInit (C++ function), 625
PhysicsModel::preconfunc (C++ type), 624
PhysicsModel::restart (C++ member), 626
PhysicsModel::rhs (C++ function), 625
PhysicsModel::runConvective (C++ function), 624
PhysicsModel::runDiffusive (C++ function), 625
PhysicsModel::runJacobian (C++ function), 625
PhysicsModel::run_precon (C++ function), 625
PhysicsModel::runRHS (C++ function), 624
PhysicsModel::runTimeStepMonitor (C++ function), 625
PhysicsModel::setJacobian (C++ function), 626
PhysicsModel::setPrecon (C++ function), 626
PhysicsModel::solver (C++ member), 626
PhysicsModel::splitop (C++ member), 627
PhysicsModel::splitOperator (C++ function), 624
PhysicsModel::timestepMonitor (C++ function), 626
PhysicsModel::user_jacobian (C++ member), 627
PhysicsModel::user_precon (C++ member), 627
PhysicsModel::userPCApply (C++ function), 601
PhysicsPrecon (C++ type), 663
PhysicsSNESApply (C++ function), 601
PI (C++ member), 340
plot () (zoidberg.poloidal_grid.PoloidalGrid method), 760
plot () (zoidberg.rzline.RZLine method), 765
plot_3d_field_line () (in module zoidberg.plot), 758
plot_backward_map () (in module zoidberg.plot), 758
plot_forward_map () (in module zoidberg.plot), 758
plot_poincare () (in module zoidberg.plot), 759
plot_streamlines () (in module zoidberg.plot), 759
plotdata () (in module boututils.plotdata), 742
pnc_get_vara_all (C++ function), 627
pnc_get_vara_all (C++ function), 627
pnc_put_vara_all (C++ function), 628
PncFormat (C++ class), 628
PncFormat::pnc_format (C++ function), 628
INDEX  823

INDEX  823

INDEX  823
RegionStats::minBlockSize (C++ member), 638
RegionStats::numBlocks (C++ member), 638
RegionStats::numMaxBlocks (C++ member), 638
RegionStats::numMinBlocks (C++ member), 638
RegionStats::numSmallBlocks (C++ member), 638
REGISTER_DERIVATIVE (C macro), 494
REGISTER_FLUX_DERIVATIVE (C macro), 494
REGISTER_FLUX_DERIVATIVE_STAGGERED (C macro), 494
REGISTER_FLUX_STAGGERED_DERIVATIVE (C macro), 494
REGISTER_STAGGERED_DERIVATIVE (C macro), 494
REGISTER_STANDARD_DERIVATIVE (C macro), 494
REGISTER_STANDARD_STAGGERED_DERIVATIVE (C macro), 494
REGISTER_UPWIND_DERIVATIVE (C macro), 494
REGISTER_UPWIND_DERIVATIVE_STAGGERED (C macro), 494
REGISTER_UPWIND_STAGGERED_DERIVATIVE (C macro), 494
RegisterInFactory (C++ class), 468
RegisterInFactory::RegisterInFactory (C++ function), 469
RegisterInvertPar (C++ class), 515
RegisterInvertPar::RegisterInvertPar (C++ function), 515
RegisterLaplace (C++ class), 510
RegisterLaplace::RegisterLaplace (C++ function), 510
RegisterLaplaceXZ (C++ class), 526
RegisterLaplaceXZ::RegisterLaplaceXZ (C++ function), 526
RegisterMesh (C++ class), 530
RegisterMesh::RegisterMesh (C++ function), 530
registerMethod (C++ class), 495
registerMethod::operator () (C++ function), 495
RegisterRKScheme (C++ type), 646
RegisterSolver (C++ type), 663
RegisterXZInterpolation (C++ class), 505
RegisterXZInterpolation::RegisterXZInterpolation (C++ function), 505
RegisterInterpolation (C++ class), 506
RegisterInterpolation::RegisterInterpolation (C++ function), 507
remove_section () (boututils.options.BOUTOptions method), 742
rename () (boutdata.data.BoutOptions method), 718
resize (in module boutdata.restart), 727
resize3DField() (in module boutdata.restart), 727
resizeY() (in module boutdata restart), 727
resizeZ() (in module boutdata restart), 728
rfft (C++ function), 412
Rfunc () (zoidberg_field.CurvedSlab method), 750
Rfunc () (zoidberg_field.GEQDSK method), 750
Rfunc () (zoidberg_field.MagneticField method), 752
Rfunc () (zoidberg_field.SmoothedMagneticField method), 753
Rfunc () (zoidberg_field.VMEC method), 754
RGN_ALL (C++ member), 324
RGN_NOBNDRY (C++ member), 324
RGN_NOX (C++ member), 325
RGN_NOY (C++ member), 325
RGN_NOZ (C++ member), 325
rhsfunc (C++ type), 602, 663
RK3SSP (C++ class), 642
RK3SSP::-RK3SSP (C++ function), 642
RK3SSP::f (C++ member), 642
RK3SSP::getCurrentTimestep (C++ function), 642
RK3SSP::init (C++ function), 642
RK3SSP::L (C++ member), 643
RK3SSP::max_timestep (C++ member), 642
RK3SSP::mxstep (C++ member), 642
RK3SSP::numBlock (C++ member), 642
RK3SSP::nlocal (C++ member), 642
RK3SSP::nsteps (C++ member), 642
RK3SSP::out_timestep (C++ member), 642
RK3SSP::RK3SSP (C++ function), 642
RK3SSP::run (C++ function), 642
RK3SSP::setMaxTimestep (C++ function), 642
RK3SSP::take_step (C++ function), 642
RK3SSP::timestep (C++ member), 642
RK3SSP::u1 (C++ member), 643
RK3SSP::u2 (C++ member), 643
RK3SSP::u3 (C++ member), 643
RK3SSP::u4 (C++ member), 643
RK4SIMPLEScheme (C++ class), 644
RK4SIMPLEScheme::RK4SIMPLEScheme (C++ function), 644
RK4SIMPLEScheme::setOutputStates (C++ function), 644
RK4Solver (C++ class), 643
RK4Solver::adaptive (C++ member), 644
RK4Solver::atol (C++ member), 643
RK4Solver::f0 (C++ member), 643
RK4Solver::f1 (C++ member), 643
RK4Solver::f2 (C++ member), 644
RK4Solver::getCurrentTimestep (C++ function), 643
RKSchemeFactory::type_name (C++ member), 646
RMSvalue() (in module boututils.moment_xzyt), 741
ropt (C++ member), 631
rotate() (in module boutdata.griddata), 722
ROUND (C++ function), 690
RunMetrics (C++ class), 546
RunMetrics::calculateDerivedMetrics (C++ function), 546
RunMetrics::ncalls (C++ member), 547
RunMetrics::ncalls_e (C++ member), 547
RunMetrics::ncalls_i (C++ member), 547
RunMetrics::outputVars (C++ function), 546
RunMetrics::t_elapsed (C++ member), 547
RunMetrics::writeProgress (C++ function), 546
RunMetrics::wtime (C++ member), 547
RunMetrics::wtime_comms (C++ member), 547
RunMetrics::wtime_invert (C++ member), 547
RunMetrics::wtime_io (C++ member), 547
RunMetrics::wtime_per_rhs (C++ member), 547
RunMetrics::wtime_per_rhs_e (C++ member), 547
RunMetrics::wtime_per_rhs_i (C++ member), 547
RunMetrics::wtime_rhs (C++ member), 547
Rvalue() (zoidberg.rzline.RZLine method), 764
rvec (C++ type), 648
RZline (class in zoidberg.rzline), 764

S
s2 (C++ enumerator), 322
SAVE_DERIVS (C++ enumerator), 663
SAVE_ONCE (C macro), 352
SAVE_ONCE1 (C macro), 352
SAVE_ONCE2 (C macro), 352
SAVE_ONCE3 (C macro), 352
SAVE_ONCE4 (C macro), 352
SAVE_ONCE5 (C macro), 352
SAVE_ONCE6 (C macro), 352
SAVE_REPEAT (C macro), 352
SAVE_REPEAT1 (C macro), 352
SAVE_REPEAT2 (C macro), 352
SAVE_REPEAT3 (C macro), 352
SAVE_REPEAT4 (C macro), 352
SAVE_REPEAT5 (C macro), 352
SAVE_REPEAT6 (C macro), 352
SAVE_VARS (C++ enumerator), 663
SaveDerivsOp (C++ class), 487
SaveDerivsOp::f_var2D (C++ member), 487
SaveDerivsOp::f_var3D (C++ member), 487
SaveDerivsOp::run (C++ function), 487
SaveDerivsOp::SaveDerivsOp (C++ function), 487
SaveVarOp (C++ class), 486
SaveVarOp::run (C++ function), 486, 487
SaveVarOp::SaveVarOp (C++ function), 486
SaveVarOp::var2D (C++ member), 487
SaveVarOp::var3D (C++ member), 487
scalevar() (in module boutdata.restart), 728
SCOREP0 (C macro), 648
SCOREP1 (C macro), 648
SCOREP2 (C macro), 648
SCOREP3 (C macro), 648
SCOREP_BASE_CALL (C macro), 648
SCOREPLVL (C macro), 648
sections() (boutdata.data.BoutOptions method), 718
Set (C++ class), 681
SET_ID (C++ enumerator), 663
settime (C++ member), 551
SETTING (C macro), 473
sfunct() (zoidberg.field.VMEC method), 754
shape (zoidberg.grid.Grid attribute), 756
shaped_line() (in module zoidberg.rzline), 766
shell() (in module boututils.run_wrapper), 743
shell_safe() (in module boututils.run_wrapper), 744
ShiftedMetric (C++ class), 594
ShiftedMetric::cachePhases (C++ function), 595
ShiftedMetric::calcParallelSlices (C++ function), 594
ShiftedMetric::canToFromFieldAligned (C++ function), 594
ShiftedMetric::checkInputGrid (C++ function), 594
ShiftedMetric::fromAlignedPhs (C++ member), 596
ShiftedMetric::fromFieldAligned (C++ function), 594
ShiftedMetric::location (C++ member), 596
ShiftedMetric::nmodes (C++ member), 596
ShiftedMetric::outputVars (C++ function), 594
ShiftedMetric::parallel_slice_phases (C++ member), 596
ShiftedMetric::ParallelSlicePhase (C++ class), 596
ShiftedMetric::ParallelSlicePhase::phase_shift (C++ member), 596
ShiftedMetric::ParallelSlicePhase::y_offset (C++ member), 596
ShiftedMetric::requiresTwistShift (C++ function), 594
ShiftedMetric::ShiftedMetric (C++ function), 594
ShiftedMetric::shiftZ(C++ function), 595, 596
ShiftedMetric::toAlignedPhs (C++ member), 596
ShiftedMetric::toFieldAligned (C++ function), 594
ShiftedMetric::zlength (C++ member), 596
ShiftedMetric::zShift (C++ member), 596
ShiftedMetricInterp (C++ class), 650
ShiftedMetricInterp::calcParallelSlices (C++ function), 651
ShiftedMetricInterp::canToFromFieldAligned (C++ function), 651
ShiftedMetricInterp::checkInputGrid (C++ function), 652
ShiftedMetricInterp::fromFieldAligned (C++ function), 651
ShiftedMetricInterp::getWeightsForYDownApproximation (C++ function), 651
ShiftedMetricInterp::getWeightsForYUpApproximation (C++ function), 651
ShiftedMetricInterp::interp_from_alignedSlab (class in zoidberg.field), 752
ShiftedMetricInterp::interp_to_aligned (C++ member), 652
ShiftedMetricInterp::location (C++ member), 652
ShiftedMetricInterp::parallel_slice_interpolators (C++ member), 652
ShiftedMetricInterp::requiresTwistShift (C++ function), 651
ShiftedMetricInterp::Shifter::toFieldAligned (C++ function), 651
ShiftedMetricInterp::ydown_index (C++ member), 652
ShiftedMetricInterp::yup_index (C++ member), 652
ShiftedMetricInterp::zShift (C++ member), 652
shiftz (C++ function), 421, 424
shiftz() (in module boutdata.shiftz), 729
SI::amu (C++ member), 341
SI::C (C++ member), 341
SI::e0 (C++ member), 341
SI::kb (C++ member), 341
SI::M_Deuterium (C++ member), 341
SI::M_Hydrogen (C++ member), 341
SI::M_Tritium (C++ member), 341
SI::Me (C++ member), 341
SI::Mp (C++ member), 341
SI::mu0 (C++ member), 341
SI::qe (C++ member), 341
SIGN (C++ function), 690
single (C++ enumerator), 394
simpson_integrate() (in module boututils.calculus), 732
simpson_matrix() (in module boututils.calculus), 733
simtime (C++ member), 317, 602
sin (C++ function), 434
single_arg_op (C++ type), 443
sink_tanhx (C++ function), 672
sink_tanhxl (C++ function), 672
sink_tanhxr (C++ function), 672
size (C++ function), 636
size() (boututils.datafile.DataFile method), 736
size() (boututils.datafile.DataFile_HDF5 method), 738
size() (boututils.datafile.DataFile_netCDF method), 739
SlepcLib (C++ class), 656
SlepcLib::-SlepcLib (C++ function), 656
SlepcLib::cleanup (C++ function), 656
SlepcLib::count (C++ member), 657
SlepcLib::help (C++ member), 657
SlepcLib::pargc (C++ member), 657
SlepcLib::pargv (C++ member), 657
SlepcLib::setArgs (C++ function), 656
SlepcLib::SlepcLib (C++ function), 656
SlepcLib::USER_EVENT (C++ member), 657
SlepcSolver (C++ class), 654
SlepcSolver::-SlepcSolver (C++ function), 654
SlepcSolver::add (C++ function), 654
SlepcSolver::advanceSolver (C++ member), 656
SlepcSolver::advanceStep (C++ function), 654
SlepcSolver::analyseResults (C++ function), 655
SlepcSolver::boutToSlepc (C++ function), 655
SlepcSolver::comm (C++ member), 655
SlepcSolver::compareEigs (C++ function), 654
SlepcSolver::compareState (C++ member), 655
SlepcSolver::constraint (C++ function), 655
SlepcSolver::constraints (C++ function), 655
SlepcSolver::createEPS (C++ function), 655
SlepcSolver::createShellMat (C++ function), 655
SlepcSolver::dadtMode (C++ member), 656
SlepcSolver::debugMonitor (C++ member), 656
SlepcSolver::fieldsToVec (C++ function), 655
SlepcSolver::f1 (C++ member), 656
SlepcSolver::f0 (C++ member), 656
SlepcSolver::getCurrentTimestep (C++ function), 655
SlepcSolver::init (C++ function), 654
SlepcSolver::localSize (C++ member), 656
SlepcSolver::monitor (C++ function), 654
SlepcSolver::nEig (C++ member), 656
SlepcSolver::n3Dvars (C++ function), 655
SlepcSolver::n2Dvars (C++ function), 655
SlepcSolver::nout (C++ member), 656
SlepcSolver::run (C++ function), 654
SlepcSolver::selfSolve (C++ member), 656
SlepcSolver::setMaxTimestep (C++ function), 655
SlepcSolver::setModel (C++ function), 654
SlepcSolver::setRHS (C++ function), 654
SlepcSolver::setSplitOperator (C++ function), 654
SlepcSolver::shellMat (C++ member), 655
SlepcSolver::slepcterToBout (C++ function), 655
SlepcSolver::slib (C++ member), 656
SlepcSolver::st (C++ member), 655
SlepcSolver::stIsShell (C++ member), 655
SlepcSolver::target (C++ member), 656
SlepcSolver::targIm (C++ member), 656
SlepcSolver::targRe (C++ member), 656
SlepcSolver::tol (C++ member), 656
SlepcSolver::tstep (C++ member), 656
SlepcSolver::useInitial (C++ member), 656
SlepcSolver::userWhich (C++ member), 656
SlepcSolver::vecToFields (C++ function), 655
slice() (in module boutdata.griddata), 722
sliceXz (C++ function), 453, 454
smooth_field_line() (zoidberg.field.SmoothedMagneticField method), 754
smooth_x (C++ function), 657, 658
smooth_y (C++ function), 657, 658
SmoothedMagneticField (class in zoidberg.field), 753
smoothXY (C++ function), 658
snes_info (C++ class), 602
snes_info::it (C++ member), 602
snes_info::linear_its (C++ member), 602
snes_info::norm (C++ member), 602
snes_info::time (C++ member), 602
SNESolver (C++ class), 661
SNESolver::SNESolver (C++ function), 661
SNESolver::init (C++ function), 661
SNESolver::Jmf (C++ member), 662
SNESolver::lib (C++ member), 662
SNESolver::mxstep (C++ member), 662
SNESolver::neg (C++ member), 662
SNESolver::nlocal (C++ member), 662
SNESolver::run (C++ function), 662
SNESolver::snes (C++ member), 662
SNESolver::snes_f (C++ member), 662
SNESolver::snes_function (C++ function), 662
SNESolver::snex_x (C++ member), 662
SNESolver::SNESolver (C++ function), 661
sotime (C++ member), 551
SOLVE_FOR (C macro), 624
SOLVE_FOR1 (C macro), 624
SOLVE_FOR2 (C macro), 624
SOLVE_FOR3 (C macro), 624
SOLVE_FOR4 (C macro), 624
SOLVE_FOR5 (C macro), 624
SOLVE_FOR6 (C macro), 624
Solver (C++ class), 644
Solver::FRONT (C++ member), 669
Solver::BACK (C++ member), 668
Solver:calculate_mms_error (C++ function), 669
Solver:call_monitors (C++ function), 668
Solver::call_timestep_monitors (C++ function), 668
Solver::canReset (C++ member), 667
Solver::constraint (C++ function), 666
Solver::contains (C++ function), 668
Solver::create (C++ function), 667
Solver::default_monitor_period (C++ member), 670
Solver::finishMonitorPeriods (C++ function), 670
Solver::f2d (C++ member), 669
Solver::f3d (C++ member), 669
Solver::finaliseMonitorPeriods (C++ function), 670
Solver::FRONT (C++ member), 668
Solver::getCurrentTimestep (C++ function), 666
SOLVERPETSC (C++ member), 663
SOLVERPOWER (C++ member), 664
SOLVERPODE (C++ member), 663
SOLVERRK3SSP (C++ member), 664
SOLVERRK4 (C++ member), 664
SOLVERRKGEMERIC (C++ member), 664
SOLVERSLPEC (C++ member), 663
SOLVERSLPECSELF (C macro), 654
SOLVERSNES (C++ member), 664
SolverType (C++ member), 663
SpecificInd (C++ class), 636
SpecificInd::ind (C++ member), 638
SpecificInd::minus (C++ function), 637
SpecificInd::ny (C++ member), 638
SpecificInd::nz (C++ member), 638
SpecificInd::offset (C++ function), 638
SpecificInd::operator++ (C++ function), 637
SpecificInd::operator= (C++ function), 637
SpecificInd::operator- (C++ function), 637
SpecificInd::plus (C++ function), 637
SpecificInd::SpecificInd (C++ function), 637
SpecificInd::x (C++ function), 637
SpecificInd::xm (C++ function), 637
SpecificInd::xmm (C++ function), 638
SpecificInd::xp (C++ function), 637
SpecificInd::xpp (C++ function), 638
SpecificInd::y (C++ function), 637
SpecificInd::ym (C++ function), 637
SpecificInd::ypp (C++ function), 637
SpecificInd::z (C++ function), 637
SpecificInd::zm (C++ function), 638
SpecificInd::zmm (C++ function), 638
SpecificInd::zp (C++ function), 638
SpecificInd::zpp (C++ function), 638
specific () (in module boututils.spectrogram), 745
split (C++ enumerator), 322
SplitFluxDerivativeType::class, 493
SplitFluxDerivativeType::meta, 493
SplitFluxDerivativeType::standard (C++ function), 493
SplitFluxDerivativeType::upwindOrFlux (C++ function), 493
SplitRK (C++ class), 672
SplitRK::adapt_period (C++ member), 674
SplitRK::adaptive (C++ member), 673
SplitRK::atol (C++ member), 673
SplitRK::diagnose (C++ member), 674
SplitRK::dydt (C++ member), 674
SplitRK::init (C++ function), 673
SplitRK::max_timestep (C++ member), 673
SplitRK::max_timestep_change (C++ member), 674
SplitRK::mxstep (C++ member), 674
SplitRK::neq (C++ member), 674
SplitRK::nlocal (C++ member), 674
SplitRK::nstages (C++ member), 673
SplitRK::nsteps (C++ member), 673
SplitRK::out_timestep (C++ member), 673
SplitRK::rtol (C++ member), 673
SplitRK::run (C++ function), 673
SplitRK::SplitRK (C++ function), 673
SplitRK::state (C++ member), 674
SplitRK::state1 (C++ member), 674
SplitRK::state2 (C++ member), 674
SplitRK::take_advection_step (C++ function), 673
SplitRK::take_diffusion_step (C++ function), 673
SplitRK::timestep (C++ function), 673
SplitRK::timeste (C++ member), 673
SplitRK::u1 (C++ member), 674
SplitRK::u2 (C++ member), 674
SplitRK::u3 (C++ member), 674
SPLITRK_HXX (C macro), 672
SPT_DATA (C++ enumerator), 676
SQ (C++ function), 690
sqrt (C++ function), 433
STAGGER (C++ enum), 323
STAGGER_STRING (C++ function), 324
Standard (C++ enumerator), 323
standard (C++ enumerator), 394
StandardFourth (C++ enumerator), 323
StandardSecond (C++ enumerator), 323
stApplyWrapper (C++ function), 653
stBackTransformWrapper (C++ function), 654
stencil (C++ class), 679
stencil::c (C++ member), 679
stencil::m (C++ member), 679
stencil::mm (C++ member), 679
stencil::p (C++ member), 679
stencil::pp (C++ member), 679
StraightStellarator (class in zoidberg.field), 754
stream_manipulator (C++ function), 583
STRENUM (C macro), 321
STRINGIFY (C macro), 316, 339
STRINGIFY1 (C macro), 339
stringToChar (C++ function), 688, 691
stringToReal (C++ function), 688, 691
strLocation (C++ function), 500
strsplit (C++ function), 688, 691
StructuredPoloidalGrid (class in zoidberg.poloidal_grid), 761
Subtract (C++ class), 405
Subtract::apply (C++ function), 405
surface_average () (in module boututils.surface_average), 746
SurfaceIter (C++ class), 679
SurfaceIter::closed (C++ function), 679
SurfaceIter::communicator (C++ function), 680
SurfaceIter::first (C++ function), 680
SurfaceIter::firstpos (C++ member), 680
SurfaceIter::firstY (C++ function), 680
SurfaceIter::isDone (C++ function), 680
SurfaceIter::lastpos (C++ member), 680
SurfaceIter::lastY (C++ function), 680
SurfaceIter::m (C++ member), 680
SurfaceIter::next (C++ function), 680
SurfaceIter::SurfaceIter (C++ class), 679
SurfaceIter::yGlobal (C++ function), 680
SurfaceIter::ySize (C++ function), 679
swap (C++ function), 296, 321, 324, 420, 431, 437, 605, 610, 612, 614, 617
sync () (boututils.datafile.DataFile method), 736
sync () (boututils.datafile.DataFile_HDF5 method), 738
sync () (boututils.datafile.DataFile_netCDF method), 739
test () (in module boututils.plotdata), 742
test_circular_boundaries () (in module zoidberg.test_rzline), 767
test_curved_slab () (in module zoidberg.test_field), 767
test_distance () (in module zoidberg.test_rzline), 767
test_FieldTracerReversible_slab () (in module zoidberg.test_fieldtracer), 767
test_getPoloidalGrid () (in module zoidberg.test_grid), 767
test_getPoloidalGrid_periodic () (in module zoidberg.test_grid), 767
test_integrate () (in module boututils.fft_integrate), 740
test_line_from_points () (in module zoidberg.test_rzline), 767
test_make_maps_slab () (in module zoidberg.test_zoidberg), 767
test_make_maps_straight_stellarator () (in module zoidberg.test_zoidberg), 767
test_order_by_distance () (in module zoidberg.test_rzline), 767
test_out_of_domain () (in module zoidberg.test_poloidal_grid), 767
test_poincare () (in module zoidberg.test_fieldtracer), 767
test_slab () (in module zoidberg.test_field), 767
test_slab () (in module zoidberg.test_fieldtracer), 767
test_spectrogram () (in module boututils.spectrogram), 746
theta (zoidberg.rzline.RZline attribute), 764
time_to_hms (C++ function), 316
Timer (C++ class), 682
Timer::~Timer (C++ function), 683
Timer::cleanup (C++ function), 683
Timer::clock_type (C++ type), 683
Timer::getAllInfo (C++ function), 683
Timer::getInfo (C++ function), 684
Timer::getTime (C++ function), 683, 684
Timer::getTotalTime (C++ function), 683, 684
Timer::info (C++ member), 684
Timer::printTimeReport (C++ function), 684
Timer::resetTime (C++ function), 683, 684
Timer::seconds (C++ type), 683
Timer::Timer (C++ function), 683
Timer::timer_info (C++ class), 684
Timer::timer_info::counter (C++ member), 684
Timer::timer_info::counter (C++ member), 684
Timer::timer_info::hits (C++ member), 684
Timer::timer_info::running (C++ member), 684
Timer::timer_info::started (C++ member), 684
Timer::timer_info::time (C++ member), 684
Timer::timer_info::total_time (C++ member), 684
Timer::timing (C++ member), 684
TimestepMonitorFunc (C++ type), 663
toFieldAligned (C++ function), 417, 432, 453, 703, 706, 707
toString (C++ function), 321, 323, 324, 412, 417, 424, 635, 689, 691
TRACE (C macro), 549
trace_poincare () (in module zoidberg.fieldtracer), 756
transform3D () (in module boutdata.input), 722
tridag (C++ function), 520
trim (C++ function), 689, 692
trimComments (C++ function), 689, 692
trimLeft (C++ function), 689, 692
trimRight (C++ function), 689, 692
TWOPI (C++ member), 340
TYPE_NAME_HXX (C macro), 686
TypeContainer (C++ class), 681
TypeContainer<T>::type (C++ type), 681
U
u1 (C++ enumerator), 322
u2 (C++ enumerator), 322
u3 (C++ enumerator), 322
Uncopyable (C++ class), 687
Uncopyable::~Uncopyable (C++ function), 687
Uncopyable::operator= (C++ function), 687
Uncopyable::operator= (C++ function), 687
unique (C++ function), 636
unique () (in module boututils.crosslines), 734
UNUSED (C macro), 687
update_progress () (in module zoidberg.progress), 764
uppercase (C++ function), 688, 691
upscale () (in module zoidberg.zoidberg), 768
Upwind (C++ enumerator), 323
user_requested_exit (C++ member), 317, 662
V
V_dot_Grad (C++ function), 699–702
VALID_COMMENTS (boutdata.data.BoutOptionsFile attribute), 719
values () (boutdata.data.BoutOptions method), 718
ValueType (C++ enum), 588
VAROPTION (C macro), 572
VDDX (C++ function), 370, 371, 382, 383
VDDY (C++ function), 371, 383, 384
VDDZ (C++ function), 372, 384, 385
VDERIV_FUNC_REGION_ENUM_TO_STRING (C macro), 375
Vector2D (C++ class), 703
Vector2D::~Vector2D (C++ function), 703
Vector2D::accept (C++ function), 705
Vector2D::applyBoundary (C++ function), 705
Vector2D::applyTDerivBoundary (C++ function), 705
Vector2D::BoutRealSize (C++ function), 705
Vector2D::byteSize (C++ function), 705
Vector2D::covariant (C++ member), 705
Vector2D::deriv (C++ member), 705
Vector2D::getLocation (C++ function), 704
Vector2D::is3D (C++ function), 705
Vector2D::isReal (C++ function), 705
Vector2D::location (C++ member), 705
Vector2D::operator* (C++ function), 704
Vector2D::operator*= (C++ function), 704
Vector2D::operator+= (C++ function), 704
Vector2D::operator+= (C++ function), 703
Vector2D::operator-= (C++ function), 704
Vector2D::operator-= (C++ function), 704
Vector2D::operator/= (C++ function), 704
Vector2D::operator/= (C++ function), 703
Vector2D::operator=(C++ function), 703
Vector2D::setLocation (C++ function), 704
Vector2D::timeDeriv (C++ function), 703
Vector2D::toContravariant (C++ function), 703
Vector2D::toCovariant (C++ function), 703
Vector2D::Vector2D (C++ function), 703
Vector3D::accept (C++ function), 709
Vector3D::applyBoundary (C++ function), 709
Vector3D::applyTDerivBoundary (C++ function), 709
Vector3D::BoutRealSize (C++ function), 709
Vector3D::byteSize (C++ function), 709
Vector3D::covariant (C++ member), 709
Vector3D::deriv (C++ member), 710
Vector3D::getLocation (C++ function), 709
Vector3D::is3D (C++ function), 709
Vector3D::isReal (C++ function), 709
Vector3D::location (C++ member), 710
Vector3D::operator* (C++ function), 708
Vector3D::operator*= (C++ function), 708
Vector3D::operator+= (C++ function), 708
Vector3D::operator+= (C++ function), 708
Vector3D::operator-= (C++ function), 708
Vector3D::operator-= (C++ function), 708
Vector3D::operator/= (C++ function), 709
Vector3D::operator/= (C++ function), 708
Vector3D::operator=(C++ function), 708
Vector3D::setLocation (C++ function), 709
Vector3D::timeDeriv (C++ function), 708
Vector3D::toContravariant (C++ function), 708
Vector3D::toCovariant (C++ function), 707
Vector3D::Vector3D (C++ function), 707
Vector3D::x (C++ member), 709
Vector3D::y (C++ member), 709
Vector3D::z (C++ member), 709
verifyNumPoints (C++ function), 305
VMEC (class in zoidberg.field), 754
Vol_Integral (C++ function), 658, 659
tils.volume_integral() (in module boututils.volume_integral), 747
Vpar_Grad_par (C++ function), 390, 391, 395
Vpar_Grad_par_LCtoC (C++ function), 396
vshift (C++ enumerator), 322
vUpDown (C++ function), 492
w2 (C++ enumerator), 322
w3 (C++ enumerator), 322
watch () (in module boututils.watch), 747
WENO_SMALL (C++ member), 494
where (C++ function), 710
wmultistream (C++ type), 556
WRAP_ENUM (C macro), 494
write() (boututils.datafile.DataFile method), 719
write() (boututils.datafile.DataFile method), 736
write() (boututils.datafile.DataFile_HDF5 method), 738
write() (boututils.datafile.DataFile_netCDF method), 739
write_Bfield_to_vtk() (in module zoidberg.zoidberg), 769
write_maps() (in module zoidberg), 748, 770
write_maps() (in module zoidberg::zoidberg), 769
X
X (C++ enumerator), 322, 474
xin (C++ enumerator), 301
xlow (C++ enumerator), 321
xout (C++ enumerator), 301
XZBilinear (C++ class), 503
XZBilinear::calcWeights (C++ function), 504
XZBilinear::i_corner (C++ member), 504
XZBilinear::interpolate (C++ function), 504
XZBilinear::k_corner (C++ member), 504
XZBilinear::w0 (C++ member), 504
XZBilinear::w1 (C++ member), 504
XZBilinear::w2 (C++ member), 504
XZBilinear::w3 (C++ member), 504
XZBilinear::XZBilinear (C++ function), 504
XZHermitSplline (C++ class), 501
XZHermitSplline::calcWeights (C++ function), 502
XZHermitSplline::getWeightsForYApproximation (C++ function), 502
XZHermitSplline::h00_x (C++ member), 502
XZHermitSplline::h00_z (C++ member), 502
XZHermitSplline::h01_x (C++ member), 502
XZHermitSplline::h01_z (C++ member), 502
XZHermitSplline::h10_x (C++ member), 502
XZHermitSplline::h10_z (C++ member), 502
XZHermitSplline::h11_x (C++ member), 502
XZHermitSplline::h11_z (C++ member), 502
XZHermitSplline::i_corner (C++ member), 502
XZHermitSplline::interpolate (C++ function), 502
XZInterpolation (C++ class), 500
XZInterpolation::-xZInterpolation (C++ function), 501
XZInterpolation::calcWeights (C++ function), 501
XZInterpolation::getWeightsForYApproximation (C++ function), 501
XZInterpolation::getWeightsForYDownApproximation (C++ function), 501
XZInterpolation::getWeightsForYUpApproximation (C++ function), 501
XZInterpolation::interpolate (C++ function), 501
XZInterpolation::localmesh (C++ member), 501
XZInterpolation::setMask (C++ function), 501
XZInterpolation::setYOffset (C++ function), 501
XZInterpolation::skip_mask (C++ member), 501
XZInterpolation::XZInterpolation (C++ function), 500
XZInterpolation::y_offset (C++ member), 501
XZInterpolationFactory (C++ class), 504
XZInterpolationFactory::create (C++ function), 504
XZInterpolationFactory::default_type (C++ member), 505
XZInterpolationFactory::ensureRegistered (C++ function), 504
XZInterpolationFactory::option_name (C++ member), 505
XZInterpolationFactory::section_name
zoidberg.test_field (module), 767
zoidberg.test_fieldtracer (module), 767
zoidberg.test_grid (module), 767
zoidberg.test_poloidal_grid (module), 767
zoidberg.test_rzline (module), 767
zoidberg.test_zoidberg (module), 767
zoidberg.zoidberg (module), 768
Zvalue() (zoidberg.rzline.RZline method), 765