BOUT++ Users Manual

B.Dudson, University of York

July 14, 2013

Contents

1 Introduction

BOUT++ is a C++ framework for writing plasma fluid simulations with an arbitrary number of equations in 3D curvilinear coordinates [1, 2]. It has been developed from the original **BOU**ndary **T**urbulence 3D 2-fluid edge simulation code [3, 4, 5] written by X.Xu and M.Umansky at LLNL.

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 simulate (for example) plasmas in 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 $\mathbf{B}$) directions.

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 the 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.

This manual is written for the user who wants to run (or modify) existing plasma models, or specify a new problem (grid and equations) to be solved. In either case, it's assumed that the user isn't all that interested in the details of the code. For a more detailed descriptions of the code internals, see the developer and reference quides. After describing how to install BOUT++ (section 2), run the test suite (section 2.6) and a few examples (section 4, more detail in section 14), 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 6. Checking that the options are doing what you think they should be by looking at the output logs is described in section 4, and an overview of the IDL analysis routines for data post-processing and visualisation is given in section 5. Generating new grid files, particularly for tokamak equilibria, is described in section 7.

Up to this point, little programming experience has been assumed, but performing more drastic alterations to the physics model requires modifying C++ code. Section 8 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 11 describes the differential operators and methods available; section 12 covers the experimental staggered grid system.

Various sources of documentation are:

- 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.
- This user's manual, which goes through BOUT++ from a user's point of view
- The developer's manual, which gives details of the internal working of the code.
- The reference guide, which summarises functions, settings etc. Intended more for quick reference rather than a guide.
- Most of the code contains Doxygen comment tags (which are slowly getting better). Running doxygen (`www.doxygen.org`) 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

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 [1] and/or other BOUT++ papers. See the file CITATION for details.

2 Getting started

This section goes through the process of getting, installing, and starting to run BOUT++. Only the basic functionality needed to use BOUT++ is described here; the next section (3) goes through more advanced options, and how to fix some common problems. This section will go through the following steps:

1. Obtaining a copy of BOUT++
2. Installing an MPI compiler (2.2)

3. Installing libraries (2.3)
4. Configuring BOUT++ analysis codes (2.4)
5. Compiling BOUT++ (2.5)
6. Running the test suite (2.6)

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

2.1 Obtaining BOUT++

BOUT++ is now hosted publicly on github (`http://github.com/bendudson/BOUT`), and includes instructions on downloading and installing BOUT++ in wiki pages. This website also has a list of current issues and history of changes. To obtain a copy of the latest version, run

```
$ git clone git://github.com/bendudson/BOUT.git
```

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

```
$ git pull
```

For more details on using git to work with BOUT++, see the developer's manual.

2.2 Installing an MPI compiler

To compile and run the examples BOUT++ needs an MPI compiler. 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. If you're installing on a smaller server or your own machine then you need to check that you have an MPI compiler by running

```
$ mpicc
```

This should produce an error message along the lines of "no input files", but if you see something like "command not found" then you need to install MPI first. There are several free MPI distributions available, the main ones currently being MPICH2 (`www.mcs.anl.gov/mpi/mpich2`), OpenMPI (`www.open-mpi.org/`), and LAM (`www.lam-mpi.org/`). On Ubuntu or Debian distributions if you have administrator rights then you can install MPICH2 by running

```
$ sudo apt-get install mpich2 libmpich2-dev
```

If this works, and you now have a working `mpicc` command, skip to the next section on installing libraries. If not, and particularly if you don't have administrator rights, you should install MPI in your home directory by compiling it 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 MPICH2 from `http://www.mcs.anl.gov/research/projects/mpich2/downloads/` and put the file in the "install" subdirectory created above. At the time of writing (June 2012), the file was called `mpich2-1.4.1p1.tar.gz`. Untar the file:

```
$ tar -xzvf mpich2-1.4.1p1.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

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

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

```
% 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.

2.3 Installing libraries

After getting an MPI compiler, the next step is to make sure the libraries BOUT++ needs are installed. At minimum BOUT++ needs the FFTW-3 library, and to run any of the examples you'll also need NetCDF-4 (prior to 4.2) installed.

> **NOTE**: There are currently issues with support for NetCDF's new C++ API, which appeared in 4.2. For now use NetCDF 4.1.x

Most large machines (e.g. NERSC Hopper, HECToR, HPC-FF etc.) will have these libraries and many more already installed, but you may need to load a module to use them. To see a list of the available modules, try running

```
modules avail
```

which works on many systems, but not all. 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.

If you're installing on your own machine, then install the packages for your distribution. On Ubuntu or Debian, the necessary packages can be installed by running

```
$ sudo apt-get install libfftw3-dev libnetcdf-dev
```

The easiest way to test if the libraries are installed correctly is try configuring BOUT++. In the `BOUT` directory obtained previously, run

```
$ ./configure
```

If this finishes by printing a summary, and paths for IDL, Python, and Octave, then the libraries are set up and you can skip to the next section. If you see a message "`ERROR: FFTW not found. Required by BOUT++`" then you need to install FFTW-3. 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` 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
```

Go back to the `BOUT` directory and re-run the configure script. If you used `$HOME/local` as the prefix, BOUT++ configure should find the FFTW library now. If you installed somewhere else, you can specify the directory with the `--with-fftw=` option:

```
$ ./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 PACT or pnetcdf installed, this will be followed by a message `ERROR: At least one file format must be supported`.

Download the 4.1.3 **bundled**[1] release of NetCDF from `http://www.unidata.ucar.edu/downloads/netcdf/netcdf-4_1_3/` and put the `netcdf-4.1.3.tar.gz` file into your "install" directory. Untar the file and 'cd' into the resulting directory:

```
$ tar -xzvf netcdf-4.1.3.tar.gz
$ cd netcdf-4.1.3
```

As with MPI compilers and FFTW, configure, then make and make install:

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

Sometimes configure can fail, in which case try disabling fortran and the HDF5 interface:

```
$ ./configure --prefix=$HOME/local --disable-fortran --disable-netcdf-4
$ make
$ make install
```

Go back to the BOUT directory and run the configure script again, this time specifying both the location of FFTW (if you installed it from source above), and the NetCDF library:

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

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

```
Configuration summary
  FACETS support: no
  PETSc support: no
  IDA support: no
  CVODE support: no
```

[1] Support for the more recent unbundled versions is coming soon

```
NetCDF support: yes
Parallel-NetCDF support: no
PDB support: no
Hypre support: no
MUMPS support: no
```

If not, see section 3 for some things you can try to resolve common problems.

2.4 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 in IDL, but an increasing amount is in Python. In particular all the test suite scripts use Python, so to run these you'll need this configured. If you just want to compile BOUT++ then you can skip to the next section, but make a note of what configure printed out.

When the configure script finishes, it prints out the paths you need to get IDL, Python, and Octave analysis routines working. After running the command which looks like

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

check that `idl` can find the analysis routines by running:

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

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

```
IDL> !path = !path + ":/path/to/BOUT/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.

To use Python, you will need the NumPy and SciPy libraries. On Debian or Ubuntu these can be installed with

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

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

```
$ python
>>> import scipy
```

If not, see the SciPy website `http://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:

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

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

To test if this command has worked, try running

```
$ python
>>> import boutdata
```

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

2.5 Compiling BOUT++

Once BOUT++ has been configured, you can compile the bulk of the code by going to the `BOUT` 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=13571f760cec446d907e1bbeb1d7a3b1c6e0212a \
 -DNCDF -DBOUT_HAS_PVODE
CHECKSUM =  ff3fb702b13acc092613cfce3869b875
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 send an error report to a BOUT++ developer such as `mailto:benjamin.dudson@york.ac.uk` containing

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

2.6 Running the test suite

In the `examples/` subdirectory there are a set of short test cases which are intended to test portions of the BOUT++ code and catch any bugs which could be introduced. To run the test cases, the Python libraries must first be set up by following the instructions in section 2.4. Go into the `examples` subdirectory and run

```
$ ./test_suite
```

This will go through a set of tests, each on a variety of different processors. **Note:** currently this 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 send an error report to `mailto:benjamin.dudson@york.ac.uk` 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 4 to start running the code.

3 Advanced installation options

This section describes some common issues encountered when configuring and compiling BOUT++, and how to configure optional libraries like SUNDIALS and PETSc.

3.1 File formats

BOUT++ can currently use three different file formats: Portable Data Binary (PDB) which is part of PACT[2], NetCDF-4[3], and experimental support for Parallel NetCDF. PDB was developed at LLNL, was used in the UEDGE and BOUT codes, and was the original format used by BOUT++. NetCDF is a more 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

[2] `http://pact.llnl.gov`

[3] `http://www.unidata.ucar.edu/software/netcdf/`

compatible with) HDF5. If you have both libraries installed then BOUT++ can use both simultaneously, for example reading in grid files in PDB format, but writing output data in NetCDF 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.

PACT `http://pact.llnl.gov/` is needed for reading and writing Portable Data Binary (PDB) format files. This is mainly for backwards compatibility with BOUT and UEDGE, and NetCDF-4 is recommended. If you need to be able to read or write PDB files, details on installing PACT are given in Appendix A.

3.2 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 I am aware), several features such as user-supplied preconditioners and constraints cannot be used with this solver. Currently, BOUT++ also supports the SUNDIALS solvers CVODE and IDA, which are available from `https://computation.llnl.gov/casc/sundials/main.html`.

The SUNDIALS library needs to have MPI enabled, so after configuring, check the output and look for something like this:

```
MPI-C Settings
--------------

checking if using MPI-C script... yes
checking if absolute path to mpicc was given... no
checking for mpicc... none
configure: WARNING: cannot find MPI-C compiler
```

If you see this warning, you need to tell CVODE which MPI compiler to use. This happens on machines where the MPI compilers are given non-standard names. For IBM AIX machines, use

```
$ ./configure --prefix=$HOME/local/ --with-mpicc=xlC
```

On NERSC's Franklin, SUNDIALS needs to be compiled with "–with-mpicc=cc" to force it to build the parallel libraries

The modern SUNDIALS CVODE solver is essentially the same as the 1998 CVODE (with some tweaking, re-arranging etc.), but this solver allows users to supply their own preconditioner and Jacobian functions. To use this solver, use

```
./configure --with-cvode
```

or

```
./configure --with-cvode=/path/to/cvode/
```

if your CVODE library is in a non-standard place.

The SUNDIALS IDA solver is a Differential-Algebraic Equation (DAE) solver, which evolves a system of the form $\mathbf{f}(\mathbf{u}, \dot{\mathbf{u}}, t) = 0$. This allows algebraic constraints on variables to be specified. If you want this functionality, compile in the IDA library using

```
./configure --with-ida
```

or

```
./configure --with-ida=/path/to/ida/
```

You can compile in several of these libraries, for example

```
./configure --with-cvode --with-ida
```

is valid. This will allow you to select at run-time which solver to use. See section 6.3 for more details on how to do this.

3.3 PETSc

BOUT++ can use PETSc for time-integration and for solving elliptic problems, such as inverting Poisson and Helmholtz equations.

```
./configure --with-petsc
```

The development version of PETSc has the latest and greatest, and is needed for some shiny features like timestepping. To install it, follow the instructions here: `http://www.mcs.anl.gov/petsc/petsc-as/developers/index.html`. At the time of writing, this consisted of fetching the latest version (**NB** needs Mercurial 'hg' to be installed):

```
hg clone http://petsc.cs.iit.edu/petsc/petsc-dev
cd petsc-dev
hg clone http://petsc.cs.iit.edu/petsc/BuildSystem config/BuildSystem
```

then configure PETSc, making sure to enable C++.

```
./configure --with-fortran=0 --with-c++-support=1 --with-mpi=1 \
            --with-sundials=1 --with-sundials-dir=$HOME/local/
```

You can add SUNDIALS support, but it is not required. To do this, add the following to the end of the configure command:

```
--with-sundials=1 --with-sundials-dir=$HOME/local/
```

replacing `$HOME/local/` with the location of your SUNDIALS installation. It is also useful to get PETSc to download and install MUMPS (see below), by adding

```
--download-mumps
```

Finally compile PETSc:

```
make
```

To use PETSc, you have to define the variable `PETSC_DIR` to point to the petsc-dev directory so add something like this to your startup file `$HOME/.bashrc`

```
export PETSC_DIR=$HOME/petsc-dev
```

3.4 LAPACK

BOUT++ comes with linear solvers for tridiagonal and band-diagonal systems, but these are not particularly optimised and are in any case descended from Numerical Recipes code (hence NOT covered by LGPL license).

To replace these routines, BOUT++ can use the LAPACK library. This 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
```

3.5 MUMPS

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

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

MUMPS has many dependencies, including ScaLapack and ParMetis, which the configuration script assumes are in the same place as MUMPS. The easiest way to get MUMPS installed is to install PETSc with MUMPS, as the configuration script will check the PETSc directory.

3.6 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.

3.7 Issues

3.7.1 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/instal` 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.

3.7.2 Compiling cvode.cxx fails

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

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

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

```
typedef int CVODEINT;
```

to

```
typedef long CVODEINT;
```

4 Running BOUT++

The `examples/` directory contains some test cases for a variety of fluid models. The ones starting **test-** are short tests, 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:

$$\frac{\partial T}{\partial t} = \nabla_{||}\left(\chi \partial_{||} T\right)$$

There are several files involved:

- **conduction.cxx** contains the source code which specifies the equation to solve
- **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 section 7.
- **generate.py** is a Python script to create the grid file. In this case it just writes nx and ny
- **data/BOUT.inp** is the settings file, specifying how many output timesteps to take, differencing schemes to use, and many other things. In this case it's mostly empty so the defaults are used.

First you need to compile the example:

```
$ gmake
```

which should print out something along the lines of

```
  Compiling  conduction.cxx
  Linking conduction
```

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

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

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

If you're running on a cluster or supercomputer, you should find out how to submit jobs. This varies, but usually on these bigger machines there will be a queueing system and you'll need to use `qsub`, `msub`, `llsubmit` or similar to submit jobs.

When the example runs, it should print a lot of output. This is recording all the settings being used by the code, and is also written to log files for future reference. The test should take a few seconds to run, and produce a bunch of files in the **data/** subdirectory.

- **BOUT.log.*** contains a log from each process, so because we ran with "-np 2" there should be 2 logs. The one from processor 0 will be the same as what was printed to the screen. This is mainly useful because if one process crashes it may only put an error message into its own log.
- **BOUT.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 section 6.4.
- **BOUT.dmp.*.nc** contain the output data, including time history. As with the restart files, each processor currently outputs a separate file.

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

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

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

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

which will then append any new outputs to the end of the old data files. For more information on restarting, see section 4.4.

To analyse the output of the simulation, cd into the **data** subdirectory and start IDL. If you don't have IDL, don't panic as all this is also possible in Python and discussed in section 5.5. First, list the variables in one of the data files:

```
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> time = collect(var="t_array")
IDL> print, time
      1.10000      1.20000      1.30000      1.40000      1.50000 ...
```

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> 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> 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

```
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. To print a list of variables in a file:

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

To collect a variable,

```
>>> from boutdata import collect
>>> T = collect("T")
>>> T.shape
```

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

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

The next example to look at is **test-wave**, which is solving a wave equation using

$$\frac{\partial f}{\partial t} = \partial_{||} g \qquad \frac{\partial g}{\partial t} = \partial_{||} f$$

using two different methods. Other examples contain two scripts: One for running the example and then an IDL script to plot the results:

```
./runcase.sh
idl runidl.pro
```

Assuming these examples work (which they should), looking through the scripts and code may give you an idea of how BOUT++ works. More information on setting up and running BOUT++ is given in section 4, and details of analysing the results using IDL are given in section 5.

4.1 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 section 8) 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 occured. If this stack isn't printed, make sure checking is set to level 2 or higher (./configure –with-checks=2)
- If the error is a segmentation fault, you can try a debugger such as totalview
- If the error is due to non-finite numbers, increase the checking level (./configure –with-checks=3) to perform more checking of values and (hopefully) find an error as soon as possible after it occurs.

4.2 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: c8794400adc256480f72c651dcf186fb6ea1da49
MD5 checksum: 8419adb752f9c23b90eb50ea2261963c
Code compiled on May 11 2011 at 18:22:37

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 section 2.5)

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

Thisn 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 PDB files aren't supported, but that NetCDF files are.

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:

```
Option /nout = 50 (data/BOUT.inp)
Option /timestep = 100 (data/BOUT.inp)
Option /grid = slab.6b5.r1.cdl (data/BOUT.inp)
Option /dump_float = true   (default)
Option /non_uniform = false (data/BOUT.inp)
Option /restart = false  (default)
Option /append = false  (default)
Option /dump_format = nc (data/BOUT.inp)
Option /StaggerGrids = false  (default)
```

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 $\frac{\partial\partial f}{\partial\partial t} = \underline{v}\cdot\nabla f$, and flux terms of the form $\frac{\partial\partial f}{\partial\partial t} = \nabla\cdot(\underline{v}f)$. 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 section 11.1 on page 71.

```
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)

WARNING: Number of inner y points 'ny_inner' not found. Setting to 32
```

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

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

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

```
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
```

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.

```
WARNING: Could not read 'zShift' from grid. Setting to 0.000000e+00
WARNING: Z shift for radial derivatives not found
```

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

```
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
```

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

```
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'
```

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

```
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.

```
Option fft/fft_measure = false  (default)
Running simulation

Run started at  : Wed May 11 18:23:20 2011

Option /wall_limit = -1 (default)
```

4.3 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

```
Sim Time  |  RHS evals  | Wall Time |  Calc    Inv   Comm    I/O   SOLVER
```

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

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

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

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

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

4.4 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.

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 "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":

```
$ 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 section 2.4) then you can use the `boutdata.restart.create` function in **tools/pylib/boutdata/restart.py**:

```
>>> 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 ".".

5 Output and post-processing

The majority of the existing analysis and post-processing code is written in IDL. The directory `idllib` contains many useful routines for reading PDB files and analysing data. A summary of available IDL routines is given in Appendix D.

Post-processing using Python is also possible, and there are some modules in the `pylib` directory, and a list of routines in Appendix E. This is a more recent addition, and so is not yet as developed as the IDL support.

5.1 Note on reading PDB files

You should never need to use PDB files with BOUT++, as all input and output routines have now been changed to use NetCDF. For backwards compatibility with BOUT, PDB files can still be used if needed. IDL comes with routines to manipulate NetCDF files, but to read PDB files you will need the PDB2IDL library supplied with BOUT++:

```
cd PDB2IDL
make
```

To use the `PDB2IDL` library and IDL analysis codes, set the following environment variables

```
IDL_PATH=$IDL_PATH:<bout>/idllib/
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<bout>/lib/
```

Before any of the PDB2IDL functions can be used, you first need to run

```
IDL> .r pdb2idl
```

This can be added to your IDL startup file (which is specified by the `IDL_STARTUP` environment variable).

5.2 Reading BOUT++ output into IDL

There are several routines provided for reading data from BOUT++ output into IDL. In the directory containing the BOUT++ output files (usually `data/`), you can list the variables available using

```
IDL> print, file_list("BOUT.dmp.0.nc")
Ajpar Apar BOUT_VERSION MXG MXSUB MYG MYSUB MZ NXPE NYPE Ni Ni0 Ni_x Te0 Te_x
Ti0 Ti_x ZMAX ZMIN iteration jpar phi rho rho_s t_array wci
```

The `file_list` procedure just returns an array, listing all the variables in a given file. This method (and all the file_ methods) works for both NetCDF and PDB files.

One thing new users can find confusing is that different simulations may have very different outputs. This is because **BOUT++ is not a single physics model**: the variables evolved and written to file are determined by the model, and will be very different between (for example) full MHD and reduced Braginskii models. There are however some variables which all BOUT++ output files contain:

- `BOUT_VERSION`, which gives the version number of BOUT++ which produced the file. This is mainly to help output processing codes handle changes to the output file format. For example, BOUT++ version 0.30 introduced 2D domain decomposition which needs to be handled when collecting data.
- `MXG`,`MYG`. These are the sizes of the X and Y guard cells
- `MXSUB`, the number of X grid points in each processor. This does not include the guard cells, so the total X size of each field will be `MXSUB + 2*MXG`.
- `MYSUB`, the number of Y grid points per processor (like MXSUB)
- `MZ`, the number of Z points
- `NXPE, NYPE`, the number of processors in the X and Y directions. `NXPE * MXSUB + 2*MXG= NX`, `NYPE * MYSUB = NY`
- `ZMIN`, `ZMAX`, the range of Z in fractions of 2π.
- `iteration`, the last timestep in the file
- `t_array`, an array of times

Most of these - particularly those concerned with grid size and processor layout - are used by post-processing routines such as `collect`, and are seldom needed directly. To read a single variable from a file, there is the `file_read` function:

```
IDL> wci = file_read("BOUT.dmp.0.nc", "wci")
IDL> print, wci
  9.58000e+06
```

NOTE: The `file_read` command (and NetCDF/PDB access generally) is case-sensitive: variable `Wci` is different to `wci`

To read in all the variables in a file into a structure, use the `file_import` function:

```
IDL> d = file_import("BOUT.dmp.0.nc")
IDL> print, d.wci
  9.58000e+06
```

This is often used to read in the entire grid file at once. Doing this for output data files can take a long time and use a lot of memory.

Reading from individual files is fine for scalar quantities and time arrays, but reading arrays which are spread across processors (i.e. evolving variables) is tedious to do manually. Instead, there is the `collect` function to automate this:

```
IDL> ni = collect(var="ni")
Variable 'ni' not found
-> Variables are case-sensitive: Using 'Ni'
Reading from .//BOUT.dmp.0.nc: [0-35][2-6] -> [0-35][0-4]
```

This function takes care of the case, so that reading "ni" is automatically corrected to "Ni". The result is a 4D variable:

```
IDL> help, ni
NI              FLOAT     = Array[36, 5, 64, 400]
```

with the indices `[X, Y, Z, T]`. Note that in the output files, these variables are stored in `[T, X, Y, Z]` format instead but this is changed by `collect`. Sometimes you don't want to read in the entire array (which may be very large). To read in only a subset, there are several optional keywords with `[min,max]` ranges:

```
IDL> ni = collect(var="Ni", xind=[10,20], yind=[2,2], zind=[0,31], tind=[300,399])
Reading from .//BOUT.dmp.0.nc: [10-20][4-4] -> [10-20][2-2]
IDL> help, ni
NI              FLOAT     = Array[11, 1, 32, 100]
```

5.3 Summary of IDL file routines

Functions file_* can read/write either PDB or NetCDF files, depending on the file extension. Hence same analysis / pre-processing codes can use PDB and/or NetCDF files. Any file ending ".nc", ".cdl", ".cdf" is assumed to be NetCDF, otherwise PDB.

Open a PDB or NetCDF file:

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

Array of variable names:

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

Number of dimensions:

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

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

```
data = file_read(handle, "variable", inds=inds)
data = file_read("filename", "variable", inds=inds)
```

Write a variable to file. For NetCDF it tries to match up dimensions, and defines new dimensions when needed

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

Close a file after use

```
file_close, handle
```

To read in all the data in a file into a structure:

```
data = file_import("filename")
```

and to write a structure to file:

```
status = file_export("filename", data)
```

Converting file types can now be done using

```
d = file_import("somefile.pdb")
s = file_export("somefile.nc", d)
```

Note that this will mess up the case of the variable names, and names may be changed to become valid IDL variable names. To convert PDB files to NetCDF there is also a code **pdb2cdf** in **BOUT/tools/archiving/pdb2cdf**.

5.4 IDL analysis routines

Now that the BOUT++ results have been read into IDL, all the usual analysis and plotting routines can be used. In addition, there are many useful routines included in the `idllib` subdirectory. There is a `README` file which describes what each of these routines, but some of the most useful ones are listed here. All these examples assume there is a variable `P` which has been read into IDL as a 4D [x,y,z,t] variable:

- `fft_deriv` and `fft_integrate` which differentiate and integrate periodic functions.
- `get_integer`, `get_float`, and `get_yesno` request integers, floats and a yes/no answer from the user respectively.
- `showdata` animates 1 or 2-dimensional variables. Useful for quickly displaying results in different ways. This is useful for taking a quick look at the data, but can also produce bitmap outputs for turning into a movie for presentation. To show an animated surface plot at a particular poloidal location (32 here):

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

  To turn this into a contour plot,

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

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

  ```
  IDL> showdata, p[*,32,0,*]
  ```

  There are a few other options, and ways to show data using this code; see the README file, or comments in `showdata.pro`. Instead of plotting to screen, showdata can produce a series of numbered bitmap images by using the `bmp` option

  ```
  IDL> showdata, p[*,32,*,*], /cont, bmp="result_"
  ```

  which will produce images called `result_0000.bmp`, `result_0001.bmp` and so on. Note that the plotting should not be obscured or minimised, since this works by plotting to screen, then grabbing an image of the resulting plot.
- `moment_xyzt` takes a 4D variable (such as those from `collect`), and calculates RMS, DC and AC components in the Z direction.

- `safe_colors` A general routine for IDL which arranges the color table so that colors are numbered 1 (black), 2 (red), 3 (green), 4 (blue). Useful for plotting, and used by many other routines in this library.

There are many other useful routines in the `idllib` directory. See the `idllib/README` file for a short description of each one.

5.5 Python routines

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`. To open a file using DataFile:

```
from boututils import DataFile

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

To list the variables in a file e.g.

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

and to list the names of the dimensions

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

or to get the sizes of the dimensions

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

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

```
1 from boututils import file_import
2
3 grid = file_import("grid.nc")
```

As for IDL, there is a `collect` routine which reads gathers together the data from multiple processors

```
1 from boutdata import collect
2
3 Ni = collect("Ni")  # Collect the variable "Ni"
```

5.6 Matlab routines

There are Matlab routines for collecting data, showing animations, and performing some basic analysis. See the **tools/matlablib/** directory and **README.txt** file.

5.7 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

```
AppendTo[$Path,"<full_path_to_BOUT>/tools/mathematicalib"]
```

in your Mathematica startup file (usually `\$HOME/.Mathematica/Kernel/init.m`). To use the package, call

```
Import["BoutCollect.m"]
```

from inside Mathematica. Then you can use e.g.

```
f=BoutCollect[variable,path->"data"]
```

or

```
f=BoutCollect[variable,path->"data"]
```

'`bc`' is a shorthand for '`BoutCollect`'. All options supported by the Python `collect()`↩ function are included, though Info does nothing yet.

5.8 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.

```
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
```

6 BOUT++ options

The inputs to BOUT++ are a binary grid file in NetCDF or PDB format, and a text file with options. Generating input grids for tokamaks is described in section 7. 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.

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. Comments are started with a hash (#) or semi-colon, which comments out the rest of the line. values can be:

- Integers
- Real values
- Booleans
- Strings

Options are also divided into sections, which start with the section name in square brackets.

```
[section1]

something = 132          # an integer
another = 5.131          # a real value
yetanother = true        # a boolean
finally = "some text"    # a string
```

NOTE: Options are NOT case-sensitive: `TwistShift` and `twistshift` are the same variable

Subsections can also be used, separated by colons ':', e.g.

```
[section:subsection]
```

Have a look through the examples to see how the options are used.

6.1 Command line options

All options 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 section 4. 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 (section 6.3) 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.

6.2 General options

At the top of the BOUT.inp file (before any section headers), options which affect the core code are listed. These are common to all physics models, and the most useful of them are:

```
NOUT = 100        # number of time-points output
TIMESTEP = 1.0    # time between outputs
```

which set the number of outputs, and the time step between them. Note that this has nothing to do with the internal timestep used to advance the equations, which is adjusted automatically. What time-step to use depends on many factors, but for high-β reduced MHD ELM simulations reasonable choices are `1.0` for the first part of a run (to handle initial transients), then around `10.0` for the linear phase. Once non-linear effects become important, you will have to reduce the timestep to around `0.1`.

Most large clusters or supercomputers have a limit on how long a job can run for called "wall time", because it's the time taken according to a clock on the wall, as opposed to the CPU time actually used. If this is the case, you can use the option

```
wall_limit = 10 # wall clock limit (in hours)
```

BOUT++ will then try to quit cleanly before this time runs out. Setting a negative value (default is -1) means no limit.

Often it's useful to be able to restart a simulation from a chosen point, either to reproduce a previous run, or to modify the settings and re-run. A restart file is output every timestep, but this is overwritten each time, and so the simulation can only be continued from the end of the last simulation. Whilst it is possible to create a restart file from the output data afterwards, it's much easier if you have the restart files. Using the option

```
archive = 20
```

saves a copy of the restart files every 20 timesteps, which can then be used as a starting point.

The X and Y size of the computational grid is set by the grid file, but the number of points in the Z (axisymmetric) direction is specified in the options file:

```
MZ = 33
```

This must be `MZ` $= 2^n + 1$, and can be $2, 3, 5, 9, \ldots$. The power of 2 is so that FFTs can be used in this direction; the +1 is for historical reasons (inherited from BOUT) and is going to be removed at some point.

Since the Z dimension is periodic, the domain size is specified as multiples or fractions of 2π. To specify a fraction of 2π, use

```
ZPERIOD = 10
```

This specifies a Z range from 0 to 2π/`ZPERIOD`, and is useful for simulation of tokamaks to make sure that the domain is an integer fraction of a torus. If instead you want to specify the Z range directly (for example if Z is not an angle), there are the options

```
ZMIN = 0.0
ZMAX = 0.1
```

which specify the range in multiples of 2π.

> **NOTE**: For users of BOUT, the definition of `ZMIN` and `ZMAX` has been changed. These are now fractions of 2π radians i.e. `dz` $= 2\pi$(`ZMAX - ZMIN`)/(`MZ-1`)

In BOUT++, grids can be split between processors in both X and Y directions. By default only Y decomposition is used, and to use X decomposition you must specify the number of processors in the X direction:

```
NXPE = 1  # Set number of X processors
```

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)

```
grid = "data/cbm18_8_y064_x260.pdb"
```

6.3 Time integration solver

BOUT++ can be compiled with several different time-integration solvers (see section **??**), 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 9 gives a list of time integration solvers, along with any compile-time options needed to make the solver available.

Table 1: Available time integration solvers

Name	Description	Compile options
euler	Euler explicit method	Always available
rk4	Runge-Kutta 4th-order explicit method	Always available
karniadakis	Karniadakis explicit method	Always available
pvode	1998 PVODE with BDF method	Always available
cvode	SUNDIALS CVODE. BDF and Adams methods	–with-cvode
ida	SUNDIALS IDA. DAE solver	–with-ida
petsc	PETSc TS methods	–with-petsc

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 2. The most commonly changed options are the absolute and relative solver tolerances, `ATOL` and `RTOL` which should be varied to check convergence.

6.4 Input and Output

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 3.

enabled is useful mainly for doing performance or scaling tests, where you want to exclude I/O from the timings. **floats** is used to reduce the size of the output files: restart files are stored as double by default (since these will be used to restart a simulation), but output dump files are set to floats by default.

To enable parallel I/O for either output or restart files, set

```
parallel = true
```

in the output or restart section. If you have compiled BOUT++ with a parallel I/O library such as pnetcdf (see section 3), 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

Table 2: Time integration solver options

Option	Description	Solvers used
atol	Absolute tolerance	rk4, pvode, cvode, ida
rtol	Relative tolerance	rk4, pvode, cvode, ida
mxstep	Maximum internal steps per output step	rk4
max_timestep	Maxmimum timestep	rk4, cvode
timestep	Starting timestep	rk4, karniadakis, euler
adaptive	Adapt timestep? (Y/N)	rk4
use_precon	Use a preconditioner? (Y/N)	pvode, cvode, ida
mudq, mldq mukeep, mlkeep maxl	BBD preconditioner settings	pvode, cvode, ida
use_jacobian	Use user-supplied Jacobian? (Y/N)	cvode
adams_moulton	Use Adams-Moulton method rather than BDF	cvode
diagnose	Collect and print additional diagnostics	cvode

Table 3: Output file options

Option	Description	Default value
enabled	Writing is enabled	true
floats	Write floats rather than doubles	true (dmp)
flush	Flush the file to disk after each write	true
guards	Output guard cells	true
openclose	Re-open the file for each write, and close after	true
parallel	Use parallel I/O	false

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.

6.5 Laplacian inversion

A common problem in plasma models is to solve an equation of the form

$$d\nabla_\perp^2 x + \frac{1}{c}\nabla_\perp c \cdot \nabla_\perp x + ax = b$$

where x and b are 3D variables, whilst a, c and d are 2D variables. BOUT++ includes several routines for solving this equation; see the developer's manual for details.

Table 4: Global Laplacian options

Option	Description	Default
`low_mem`	Reduces memory usage	`false`
`use_pdd`	Use the PDD algorithm	`false`
`all_terms`	Include all terms	`false`
`laplace_nonuniform`	Non-uniform mesh corrections	`false`
`filter`	Fraction of modes to filter	0.2
`max_mode`	Maximum Z mode	filter

6.6 Communications

The communication system has a section `[comms]`, with a true/false option `async`. This determines whether asyncronous MPI sends are used; which method is faster varies (though not by much) with machine and problem.

6.7 Differencing methods

Differencing methods are specified in three section (`[ddx]`, `[ddy]` and `[ddz]`), one for each dimension.

- `first`, the method used for first derivatives
- `second`, method for second derivatives
- `upwind`, method for upwinding terms
- `flux`, for conservation law terms

The methods which can be specified are 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.

6.8 Model-specific options

The options which affect a specific physics model vary, since they are defined in the physics module itself (see section 8.3). They should have a separate section, for example the high-β 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 section 5 for more details.

6.9 Variable initialisation

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

There are two ways to specify the initial conditions for a variable: the original method (similar to that used by BOUT-06) which covers the most commonly needed functions. If more flexibility is needed then a more general analytical expression can be given.

6.9.1 Original method

The shape of the initial value is specified for each dimension separately using the options `xs_opt`, `ys_opt`, and `zs_opt`. These are set to an integer:

0. Constant (this is the default)
1. Gaussian, with a peak location given by `xs_s0`, `ys_s0`, `zs_s0` as a fraction of the domain (i.e. $0 \rightarrow 1$. The width is given by `*s_wd`, also as a fraction of the domain size.
2. Sinusoidal, with the number of periods given by `*s_mode`.
3. Mix of mode numbers, with psuedo-random phases.

The magnitude of the initial value is given by the variable `scale`.

Defaults for all variables can be set in a section called `[All]`, so for example the options below:

```
[All]
scale = 0.0 # By default set variables to zero

xs_opt = 1  # Gaussian in X
ys_opt = 1  # Gaussian in Y
zs_opt = 2  # Sinusoidal in Z (axisymmetric direction)
```

```
xs_s0 = 0.5 # Peak in the middle of the X direction
xs_wd = 0.1 # Width is 10% of the domain

ys_s0 = 0.5 # Peak in the middle of the Y direction
ys_wd = 0.3 # Width is 30% of the Y domain

zs_mode = 3 # 3 periods in the Z direction

[U]
scale = 1.0e-5  # Amplitude for the U variable, overrides default
```

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, a truncated Ballooning transformation can be used to construct a smooth initial perturbation:

$$U_0^{balloon} = \sum_{i=-N}^{N} F(x)\, G(y + 2\pi i)\, H(z + q 2\pi i)$$

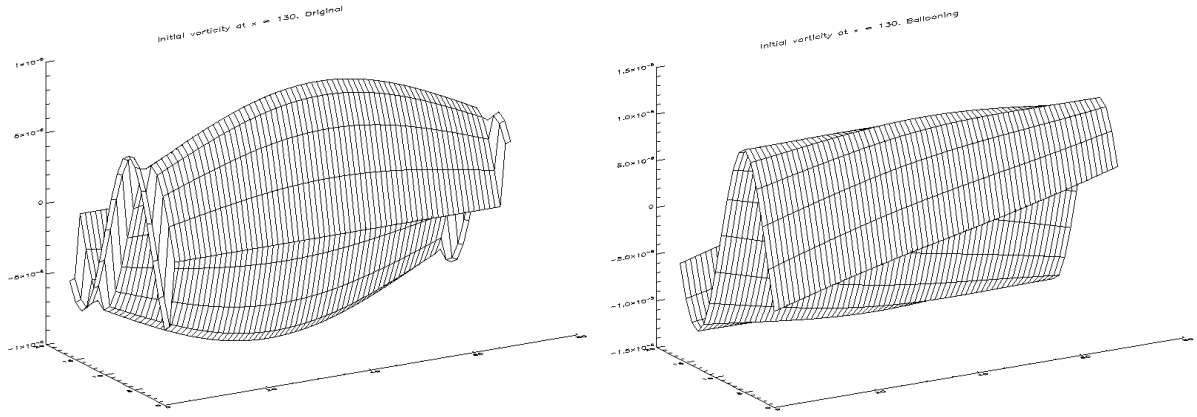


Figure 1: Initial profiles in twist-shifted grid. **Left**: Without ballooning transform, showing discontinuity at the matching location **Right**: with ballooning transform

> **NOTE**: The initial profiles code currently doesn't work very well for grids with branch-cuts (e.g. divertor tokamak), and will often have jumps which then make timesteps smaller

6.9.2 Expressions

If a more general function is needed, a variable can be initialised using the `function` option for each variable. This overrides the original method for that variable. e.g.

```
[all]

xs_opt = 1  # Gaussian in X
ys_opt = 1  # Gaussian in Y
zs_opt = 2  # Sinusoidal in Z (axisymmetric direction)

[U]
scale = 1.0e-5

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

will use the original method to set U, but use the given expression to set p.

Expressions can include the usual operators (`+`,`-`,`*`,`/`), including `^` for exponents. The following values are also already defined: By default, x is defined as `i / (nx - 2*MXG)`,

Table 5: Initialisation expression values

Name	Description
x	x position between 0 and 1
y	y position between 0 and 2π
z	z position between 0 and 2π
pi	3.1415...

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

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

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

The following functions are also available in expressions:

6.10 Boundary conditions

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

Table 6: Initialisation expression functions

Name	Description
abs(x)	Absolute value $\|x\|$
asin(x), acos(x), atan(x), atan(y,x)	Inverse trigonometric functions
cos(x)	Cosine
cosh(x)	Hyperbolic cosine
exp(x)	Exponential
tanh(x)	Hyperbolic tangent
gauss(x)	Gaussian $\exp\left(-x^2/2\right)/\sqrt{2\pi}$
gauss(x, w)	Gaussian $\exp\left[-x^2/\left(2w^2\right)\right]/\left(w\sqrt{2\pi}\right)$
log(x)	Natural logarithm
min(x,y,...)	Minimum (variable arguments)
max(x,y,...)	Maximum (variable arguments)
sin(x)	Sine
sinh(x)	Hyperbolic sine
sqrt(x)	$\sqrt{x}$
tan(x)	Tangent
H(x)	Heaviside function: 1 if $x > 0$ otherwise 0

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:

$$\begin{aligned}\nabla_\perp^2 f &= 0 \\ &\simeq g^{xx}\frac{\partial^2 f}{\partial x^2} + g^{zz}\frac{\partial^2 f}{\partial z^2}\end{aligned}$$

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

$$\hat{f} = Ae^{xk_z\sqrt{g^{zz}/g^{xx}}} + Be^{-xk_z\sqrt{g^{zz}/g^{xx}}}$$

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 behavior of boundary conditions, and more than one modifier can be used. Currently the following are available:

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

These are described in the following subsections.

6.10.1 Relaxing boundaries

All boundaries can be modified to be "relaxing" which are a combination of zero-gradient time-derivative, and whatever boundary condition they are applied to. The idea is that this prevents sharp discontinuities at boundaries during transients, whilst maintaining the desired boundary condition on longer timescales. 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 τ is a timescale 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 timescale 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.

6.10.2 Shifted boundaries

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

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

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

6.10.3 Changing the width of boundaries

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

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

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

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

would relax the last 3 cells towards zero, whereas

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

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

Limitations:

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

6.10.4 Examples

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

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

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

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

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

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

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

then the "target" boundary condition for `Ni` would first search in the `[Ni]` section for `bndry_target`, then for `bndry_all` in the `[Ni]` section. This is set to `relax(dirichlet(0.1))`, not the Neumann condition desired.

7 Generating input grids

The simulation mesh describes the number and topology of grid points, the spacing between them, and the coordinate system. For many problems, a simple mesh can be created using options.

```
[mesh]
nx = 260  # X grid size
ny = 256  # Y grid size

dx = 0.1  # X mesh spacing
dy = 0.1  # Y mesh spacing
```

The above options will create a 260×256 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.

NOTE: Currently only scalars are allowed, not expressions

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 or PDB 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**
- Differencing quantities in 2D arrays `dx[nx][ny]` and `dy[nx][ny]`. If these are not found they will be set to 1.
- Diagonal terms of the metric tensor g^{ij} `g11[nx][ny]`, `g22[nx][ny]`, and `g33[nx][ny]`. If not found, these will be set to 1.
- Off-diagonal metric tensor g^{ij} elements `g12[nx][ny]`, `g13[nx][ny]`, and `g23[nx][ny]`. If not found, these will be set to 0.
- Z shift for sheared grids `zshift[nx][ny]`. This is intended for dpsi derivatives in sheared coordinates. If not found, 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 `twistshift[nx]`. This is applied in the "core" region between indices `jyseps2_2`, and `jyseps1_1 + 1`, if enabled in the options file. If not given, this is set to zero.

NOTE: All input quantities should be normalised - no normalisation is performed by the BOUT++ code. Normalisation can be performed in the initialisation code, provided a call to `geometry()` is made after any changes to the metrics.
For users of BOUT, the radial derivative is `dx = dpsi / (bmag/1e4)`

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.

Figure 2 shows the routines and file formats used in taking output from different codes and converting into input to BOUT++.

7.1 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.

7.2 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 or PDB format which describes the Grad-Shafranov equilibrium. These intermediate files are then converted to BOUT++ grids using an interactive IDL script.

7.3 Generating equilibria

The directory `tokamak_grids/shifted_circle` contains IDL code to generate shifted circle (large aspect ratio) Grad-Shafranov equilibria.

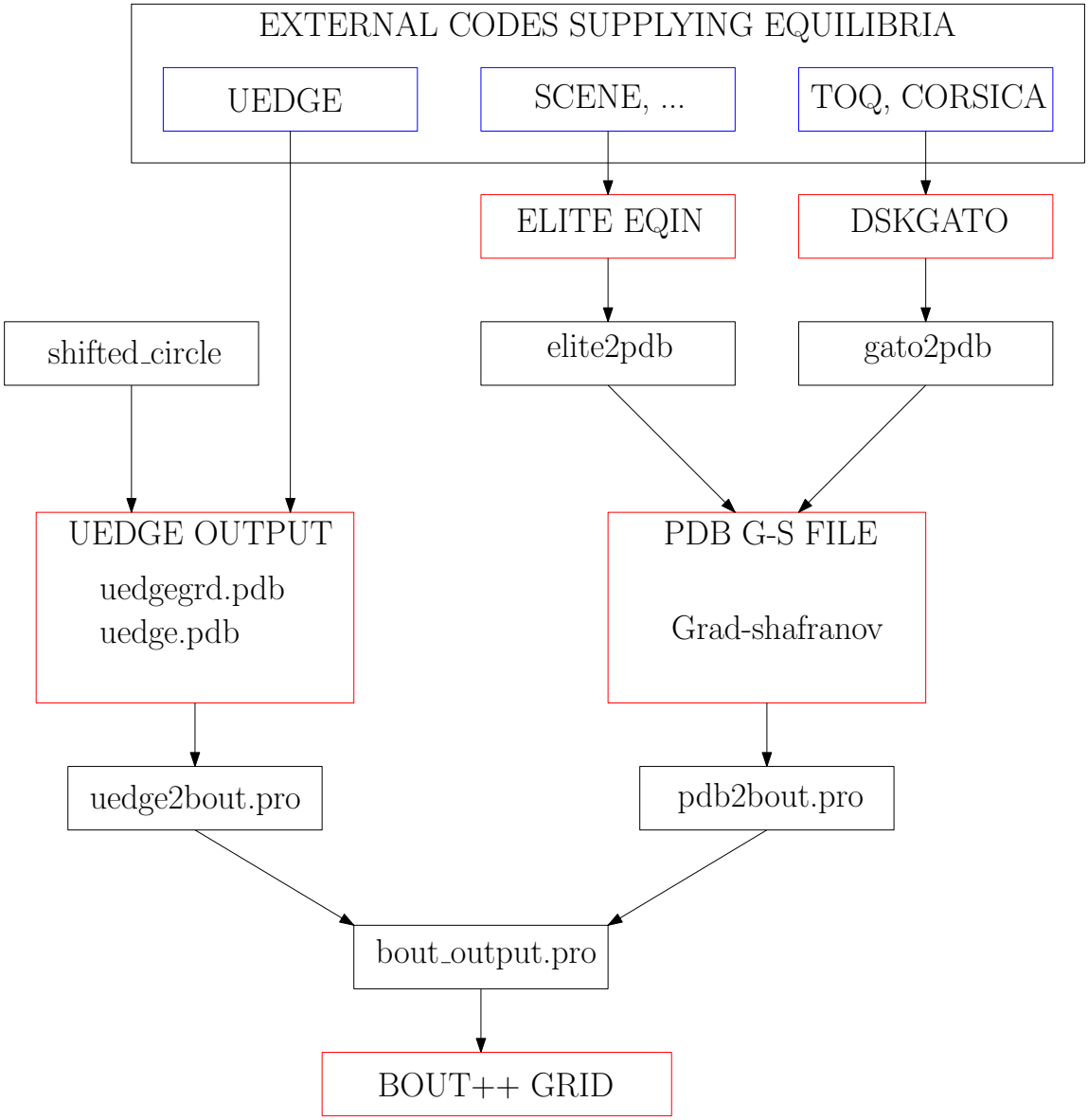


Figure 2: Generation of BOUT++ grid files. In red are the file formats, and in black the conversion routines. Blue are external codes.

7.4 Running pdb2bout

There are many options which are set interactively, so here's a run-through of the code (only showing most important outputs):

```
IDL> pdb2bout, "cbm18_dens6.dskgato.pdb", output="test.pdb"
***Maximum mu0p is       23071.7
Is this pressure (not mu0*pressure)?
```

This is needed because although many grid formats claim to store $\mu_0 P$, they actually store P. Since the given maximum value is very large, it must be in Pascals, so answer yes (y).

The grid will then be displayed along with the safety factor and pressure profiles against normalised ψ. In all three plots, a red line marks the location of the plasma edge. You must then choose the radial domain in normalised ψ:

```
Inner Psi boundary:0.6
Outer Psi boundary:1.2
Number of radial grid points:           69
Of which inside the plasma:           49
Is this range ok?
```

The plots will now also show two green lines for the inner and outer boundaries. Enter no (n) to specify a different range.

The code then checks to see if any plasma density of temperatures have been set in the input file:

```
====== SETTING PLASMA PROFILES =======
Some plasma parameters given in input file
Use given parameters?
```

Saying yes will just use the given values, but saying no will give you some more options:

```
Generating plasma profiles:
  1. Flat temperature profile
  2. Flat density profile
  3. Te proportional to density
Profile option:
```

The procedure is the same for each of these, so taking option 2 (flat density):

```
Setting flat density profile
Density [10^20 m^-3]:1.0
```

Anything can be entered here, depending on what you want to simulate. The code will ensure that whatever you enter, the equilibrium pressure is maintained. In this case the temperature is calculated from pressure and the specified density.

```
Maximum temperature (eV):      720.090
Is this ok?
```

NOTE: This is the maximum temperature anywhere on the input grid (i.e. in this case at $\psi = 0$), not just inside the chosen domain. Entering no will go back and you can specify a different density.

Earlier the radial resolution was printed, in this case 69 grid points. You can now change this if you need to:

```
Increase radial resolution?
```

Although it says increase, you can also decrease the resolution. Entering yes will allow you to enter a different number of radial grid points. It's recommended to use $2^n + 4$ grid points because this makes it easier to decompose the grid (4 cells for boundary, remainder equally divided between processors).

At this point, an orthogonal grid is generated:

```
======== GENERATING ORTHOGONAL COORDINATES FOR BOUT =========
Number of poloidal grid points: 64
Enter x index of equal hthe [0, 68] :35
```

Using 2^n points is highly recommended, but only because the points must be equally divided between processors. The x index of equal hthe shouldn't matter very much except for highly shaped plasmas. Recommend that you set it somewhere around the peak pressure gradient, or middle of the grid.

```
Interpolating Rxy
Interpolating Zxy
Is this ok?
```

Two plots are shown: On the left the original mesh, and on the right the new orthogonal mesh. If this doesn't look right you can enter 'no' and change the poloidal resolution and location of equal h_θ.

```
Add vacuum region?n
```

This is a little experimental, and extends the grid into vacuum. This is useful if the equilibrium supplied doesn't include a vacuum region. In this case we already have a vacuum region, so can answer no.

You're now presented with several options:

```
Equilibrium correction options:
  0  No correction
  1  RBt using force balance
  2  hthe and RBt using force balance and q (FAILS)
  3  hthe and RBt using force balance and jpar
Enter option:
```

Because the input Grad-shafranov solution is probably not perfect to begin with, and has now been interpolated onto a new grid, the force balance in ballooning coordinates is not quite satisfied. These options attempt to correct the equilibrium slightly to ensure force balance, the given q profile, and the given $j_{||}$ profile. Option 1 can sometimes work well, other times it can fail to converge (as in this case). It's safe to just use option 0.

```
Calculating poloidal arc length dthe
Maximum difference in hthe:       0.23551123
Maximum percentage difference:        16.745859
Use new hthe?
```

A key metric in the BOUT/BOUT++ coordinates is the poloidal arc length h_θ. A plot will shot this quantity calculated geometrically (solid line), and calculated by enforcing force balance (red symbols) at the outboard midplane. The difference between these two methods is an indication of the quality of the Grad-shafranov solution. Entering 'y' will use the "new" h_θ calculated from force balance, whilst 'n' will use the h_θ calculated geometrically. Personally. i prefer to make sure force balance is satisfied so enter 'y'.

```
Checking parallel current
****Equilibrium has -ve toroidal field
```

Because of the varied and confusing ways different codes define the poloidal and toroidal directions, this code currently just sets Bp and Bt positive, and then uses the expression for Jpar to work out what sign Bt should have. This is fine if you just want an equilibrium, but for detailed comparison to experiment where the sign of Bt may/will make a difference this needs to be changed.

Jpar calculated from quantities such as Bp, Bt and hthe is now shown as red symbols, with the jpar from the original Grad-shafranov solution as a black line. Like the hthe display, this is a good consistency check.

```
Use new Jpar?
```

Entering 'y' will use the calculated jpar i.e. consistent with the other grid quantities, but probably more noisy and slightly different to the original. Entering 'n' will use the original jpar profiles.

```
q is negative. Reversing values from equilibrium
```

This can be printed because the q profile given in the grid file is almost always positive, whereas qsafe calculated by integrating the pitch angle can be positive or negative. In this case the toroidal field has been set negative (see above), and so qinty is negative too.

```
Use new qsafe?
```

As with hthe and jpar, the qsafe specified in the original grid file is plotted as a black line, and the value calculated by integrating quantities on the new mesh is shown as red symbols. Entering 'y' uses the values consistent on the new grid, whilst 'n' uses the original safety factor profile. In most cases i'd prefer the grid to be consistent, rather than being identical to the input, so answer 'y'. You may have to do some experimentation though.

```
****Minimum pressure is very small:       0.0000000
****Setting minimum pressure to 1% of maximum
```

This is because having negative pressures is very bad for BOUT/BOUT++ runs, and can easily be caused by overshoots or even rounding error when the pressure is too low. Because the equilibrium doesn't depend on absolute pressure, this just adds a constant pressure across the entire profile.

Finally, the grid file is written to PDB format

```
Cannot write 2 dimensional double dx. Writing as float
.
.
.
```

These warnings are because the PDB2IDL library currently doesn't have any functions for writing doubles, and pdb2bout does calculations in double precision. The output is therefore converted to single-precision floats.

8 Fluid equations

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 uses the ideal MHD equations as an example, demonstrating how a BOUT++ physics module 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 module. 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.

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. The equations to be solved are:

$$\begin{aligned}
\frac{\partial \rho}{\partial t} &= -\mathbf{v}\cdot\nabla\rho - \rho\nabla\cdot\mathbf{v} \\
\frac{\partial p}{\partial t} &= -\mathbf{v}\cdot\nabla p - \gamma p\nabla\cdot\mathbf{v} \\
\frac{\partial \mathbf{v}}{\partial t} &= -\mathbf{v}\cdot\nabla\mathbf{v} + \frac{1}{\rho}\left(-\nabla p + (\nabla\times\mathbf{B})\times\mathbf{B}\right) \\
\frac{\partial \mathbf{B}}{\partial t} &= \nabla\times(\mathbf{v}\times\mathbf{B})
\end{aligned}$$

There are two ways to specify a set of equations to solve in BOUT++. For advanced users, an object-oriented interface is available and described in section 10. The simplest way to start is to use a C-like interface and define two functions:

```
1 int physics_init(bool restarting) {
2   return 0;
3 }
4
5 int physics_run(BoutReal t) {
6   return 0;
7 }
```

The first of these 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 second function `physics_run` 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.

8.1 Variables

We need to define the variables to evolve as global variables (so they can be used in `physics_init` and `physics_run`.

> **NOTE**: Version 0.85 and earlier needed two variables to be defined, so if you're upgrading then you can remove the time-derivative variables

For ideal MHD, we need two 3D scalar fields density ρ and pressure p, and two 3D vector fields velocity v, and magnetic field B:

```
Field3D rho, p; // 3D scalar fields
Vector3D v, B;  // 3D vector fields

int physics_init(bool restarting) {
}
```

Scalar and vector fields behave much as you would expect: `Field3D` objects can be added, subtracted, multiplied, divided and exponentiated, so the following examples are all valid operations:

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

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

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

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

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

In addition the dot and cross products are represented by $*$ and $\wedge$ symbols:

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

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

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

```
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: In C++ the ∧ operator has lower precedence than the `*` or `+` operators. To be safe, always put exponentiation and cross-product operations in brackets

8.2 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 `physics_init`:

```
1 int physics_init(bool restarting) {
2   bout_solve(rho, "density");
3   bout_solve(p,   "pressure");
4   bout_solve(v,   "v");
5   bout_solve(B,   "B");
6
7   return 0;
8 }
```

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 section **??**). 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`:

```
1 SOLVE_FOR(v);
2 SOLVE_FOR(B);
```

To make this even shorter, we can use macros `SOLVE_FOR2`, `SOLVE_FOR3`, ..., `SOLVE_FOR6` to shorten our initialisation code to

```
1 int physics_init(bool restarting) {
2   bout_solve(rho, "density");
3   bout_solve(p,   "pressure");
4   SOLVE_FOR2(v, B);
5
6   return 0;
7 }
```

The equations to be solved can now be written in the `physics_run` function. The value passed to the function (`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 `var`, use `ddt(var)`. The ideal MHD equations can be written as:

```
1 int physics_run(BoutReal t) {
2   ddt(rho) = -V_dot_Grad(v, rho) - rho*Div(v);
```

```
3   ddt(p) = -V_dot_Grad(v, p) - gamma*p*Div(v);
4   ddt(v) = -V_dot_Grad(v, v) + ( (Curl(B)^B) - Grad(p) ) / rho;
5   ddt(B) = Curl(v^B);
6 }
```

Where the differential operators `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 `v*Grad(rho)`, but this would then use central differencing in the Grad operator. Instead, the function `V_dot_Grad` uses upwinding methods for these advection terms. In addition, the `Grad` function will not operate on vector objects (since result is neither scalar nor vector), so the $\mathbf{v} \cdot \nabla \mathbf{v}$ term CANNOT be written as `v*Grad(v)`.

8.3 Input options

Note that in the above equations the extra parameter `gamma` has been used. To enable this to be set in the input options file (see section 6), we use the `options` object in the initialisation function:

```
1 BoutReal gamma;
2
3 int physics_init(bool restarting) {
4   Options *globalOptions = Options::getRoot();
5   Options *options = globalOptions->getSection("mhd");
6
7   options->get("gamma", gamma, 5.0/3.0);
```

This specifies that an option called "gamma" in a section called "mhd" should be put into the variable `gamma`. 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 gamma is not set in the input file the following line will appear:

```
Option mhd / gamma = 1.66667 (default)
```

This function can be used to get integers and booleans. To get strings, there is the function (`char* options.getString(section, name)`. To separate options specific to the physics model, these options should be put in a separate section, for example here the "mhd" section has been specified. To save having to write the section name for every option, there is the `setSection` function:

```
1 BoutReal gamma;
2 int someint;
3
4 int physics_init(bool restarting) {
```

```
5   Options *globalOptions = Options::getRoot();
6   Options *options = globalOptions->getSection("mhd");
7
8   options->get("gamma", gamma, 5.0/3.0);
9   options->get("someint", someint, 0);
```

Most of the time, the name of the variable (e.g. `gamma`) will be the same as the identifier in the options file ("gamma"). In this case, there is the macro

```
OPTION(options, gamma, 5.0/3.0);
```

which is equivalent to

```
options->get("gamma", gamma, 5.0/3.0);
```

See section 6 for more details of how to use the input options.

8.4 Communication

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

If you only need to communicate a small number (up to 5 currently) of variables then just call the `mesh->communicate` function directly. For the MHD code, we need to communicate the variables `rho,p,v,B` at the beginning of the `physics_run` function before any derivatives are calculated:

```
1 int physics_run(BoutReal t) {
2   mesh->communicate(rho, p, v, B);
```

If you need to communicate lots of variables, or want to change at run-time which variables are evolved (e.g. depending on input options), then you can create a group of variables and communicate them later. To do this, first create a `FieldGroup` object, in this case called `comms`, then use the add method. This method does no communication, but records which variables to transfer when the communication is done later.

```
1 FieldGroup comms;
2
3 int physics_init() {
4   .
5   .
6   .
7   comms.add(rho);
8   comms.add(p);
```

```
9     comms.add(v);
10    comms.add(B);
11
12    return 0;
13  }
```

The comms.add() routine can be given up to 6 variables at once (there's no practical limit on the total number of variables which are added to a FieldGroup), so this can be shortened to

```
1   FieldGroup comms;
2
3   int physics_init() {
4     .
5     .
6     .
7     comms.add(rho, p, v, B);
8
9     return 0;
10  }
```

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 physics_run routine:

```
1   int physics_run(BoutReal t) {
2     mesh->communicate(comms);
3     .
4     .
5     .
```

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

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

```
1   int physics_run(BoutReal t) {
2     mesh->communicate(rho, p, v); // sends and receives rho, p and v
3     comm_handle ch = mesh->send(B);// only send B
4
```

```
5     ddt(rho) = ...
6     ddt(p) = ...
7
8     mesh->wait(ch); // now wait for B to arrive
9
10    ddt(v) = ...
11    ddt(B) = ...
12
13    return 0;
14  }
```

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!).

> **NOTE**: Before using the result of a differential operator as input to another differential operator, communications must be performed for the intermediate result

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 section 11.

8.5 Boundary conditions

All evolving variables have boundary conditions applied automatically after the `physics_run` has finished. Which condition is applied depends on the options file settings (see section 6.10). 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:

```
1   Field3D var;
2   ...
3   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 `physics_init`, then just apply them every time:

```
Field3D var;

int physics_init() {
  ...
  var.setBoundary("myVar");
  ...
}

int physics_run(BoutReal t) {
  ...
  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 `bout_solve` (or `SOLVE_FOR`) the `setBoundary` method is called, and then after `physics_run` the 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:

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

8.5.1 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`.

```
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->ngy; y++)
      for(int z=0; z < mesh->ngz; 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->ngx-2; x < mesh->ngx; x++)
    for(int y=0; y < mesh->ngy; y++)
      for(int z=0; z < mesh->ngz; z++) {
        f[x][y][z] = ...
      }
}
```

note the size of the local mesh including guard cells is given by `mesh->ngx`, `mesh->ngy`, and `mesh->ngz`. 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--)  // Boundary width 3 points
    for(int z=0;z<mesh->ngz;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->ngy-3;y<mesh->ngy;y--)  // Boundary width 3 points
      for(int z=0;z<mesh->ngz;z++) {
        ddt(f)[it.ind][y][z] = 0.;  // Set time-derivative to zero in boundary
      }
  }
```

8.6 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 $\rho(t) \to \rho_0 + \tilde{\rho}(t)$. For this reason, **the initialisation of all variables passed to the `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 `grid.get` function:

```
Field2D Ni0; // Background density

int physics_init(bool restarting) {
  ...
  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");
```

8.7 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);
```

9 Fluid equations 2: 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

$$\begin{aligned}\frac{\partial U}{\partial t} &= -\frac{1}{B}\mathbf{b}_0 \times \nabla\phi \cdot \nabla U + B^2\nabla_{||}\left(j_{||}/B\right) \\ \frac{\partial A_{||}}{\partial t} &= -\frac{1}{\hat{\beta}}\nabla_{||}\phi - \eta\frac{1}{\hat{\beta}}j_{||}\end{aligned}$$

with ϕ and $j_{||}$ given by

$$\begin{aligned}U &= \frac{1}{B}\nabla_\perp^2\phi \\ j_{||} &= -\nabla_\perp^2 A_{||}\end{aligned}$$

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

```
Field3D U, Apar; // Evolving variables

int physics_init(bool restarting) {

  SOLVE_FOR2(U, Apar);
}

int physics_run(BoutReal t) {
  mesh->communicate(U, Apar);

}
```

In order to calculate the time derivatives, we need the auxilliary variables ϕ and $j_{||}$. Calculating $j_{||}$ from $A_{||}$ is a straightforward differential operation, but getting ϕ from U means inverting a Laplacian.

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

int physics_init(bool restarting) {
  SOLVE_FOR2(U, Apar);
  SAVE_REPEAT2(phi, jpar); // Save variables in output file
  return 0;
}

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

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

```
int physics_run(BoutReal t) {
  ...
  mesh->communicate(jpar);

  ddt(U) = -b0xGrad_dot_Grad(phi, U) + SQ(mesh->Bxy)*Grad_par(Jpar / ↩
     mesh->Bxy)
```

```
6   ddt(Apar) = -Grad_par(phi) / beta_hat - eta*jpar / beta_hat;
7 }
```

9.1 Printing messages/warnings

In order to print to screen and/or a log file, the object `output` is provided. This provides two different ways to write output: the C (`printf`) way, and the C++ stream way. This is because each method can be clearer in different circumstances, and people have different tastes in these matters.

The C-like way (which is the dominant way in BOUT++) is to use the `write` function, which works just like `printf`, and takes all the same codes (it uses `sprintf` internally).

```
output.write(const char *format, ...)
```

For example:

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

For those who prefer the C++ way of doing things, a completely equivalent way is to treat `output` as you would `cout`:

```
output << "This is an integer: " << 5 << ", and this a real: " << 2.0↩
    << endl;
```

which will produce exactly the same result as the `output.write` call above.

On all processors, anything sent to `output` will be written to a log file called `BOUT.log.#` with # replaced by the processor number. On processor 0, anything written to the output will be written to screen (stdout), in addition to the log file. Unless there is a really good reason not to, please use this `output` object when writing text output.

9.2 Laplacian inversion

Quite a common problem in plasma simulation codes is to invert an equation of the form

$$\nabla_\perp^2 x + ax = b$$

where a is symmetric in z, and the operator $\nabla_\perp = \nabla - \mathbf{b}\left(\mathbf{b}\cdot\nabla\right) = -\mathbf{b}\times\left(\mathbf{b}\times\nabla\right)$. For example, this operator appears in reduced MHD for the vorticity inversion and $j_{||}$.

Efficiently inverting this operator is done by taking FFTs in the z direction to transform this problem into a set of 1D inversion problems (in x) for each Fourier mode. These inversion problems are band-diagonal (tri-diagonal in the case of 2nd-order differencing) and so inversions can be very efficient: $O\left(n_z \log n_z\right)$ for the FFTs, $O\left(n_x\right)$ for tridiagonal inversion using the Thomas algorithm [6], where n_x and n_z are the number of grid-points in the x and z directions respectively.

The `Laplacian` class is defined in `invert_laplace.hxx` and solves problems of the form

$$d\nabla_\perp^2 x + \frac{1}{c}\nabla_\perp c \cdot \nabla_\perp x + ax = b$$

To use this class, first create an instance of it:

```
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 $d = a = 1$ and the c term is switched off. To set the values of these coefficients, there are the `setCoefA`(), `setCoefC`(), and `setCoefD`() methods:

```
Field2D a = ...;
lap->setCoefA(a);
lap->setCoefC(0.5);
```

arguments can be `Field2D`, `Field3D`, or real values. Settings are controlled using `setFlags`

```
lap->setFlags(flags);
```

`flags` is an `int` which determines boundary conditions and other options. Its value is calculated by adding the settings given in `invert_laplace.hxx`, and reproduced in table 7.

Table 7: Laplacian inversion flags: add the required quantities together.

Flag	Meaning
1	Zero-gradient DC on inner (X) boundary. Default is zero-value
2	Zero-gradient AC on inner boundary
4	Zero-gradient DC on outer boundary
8	Zero-gradietn AC on outer boundary
16	Zero DC component everywhere
32	Not used currently
64	Set width of boundary to 1 (default is `MXG`)
128	Use 4^{th}-order band solver (default is 2^{nd} order tridiagonal)
256	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.
512	Zero laplacian AC on outer boundary
1024	Symmetric boundary condition on inner boundary
2048	Symmetric outer boundary condition

To perform the inversion, there's the `solve` method

```
x = lap->solve(b);
```

If you prefer, there are functions compatible with older versions of the BOUT++ code:

```
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.

9.3 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 section 9.1). 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. At the start of a section of code, put a message onto the stack:

```
   msg_stack.push("Some message here");
```

which can also take arguments in `printf` format, as with `output.write`. At the end of the section of code, take the message off the stack again:

```
   msg_stack.pop();
```

If an error occurs, the message stack is printed out, and this can then help track down where the error originated.

10 Object-orientated interface

If you prefer to create classes rather than global variables and C functions for your physics model, this can be done using a (somewhat experimental) interface. To see the difference, compare **examples/advect1d/gas_compress.cxx** with **examples/advect1d-newapi/gas_compress.cxx**. The disadvantage of this interface is that it's marginally more complicated to set up, but it has several advantages: It makes splitting the model into multiple files easier (sharing global variables is a pain), models

can be combined together to enable coupling of models, and BOUT++ can be more easily used alongside other libraries. For large models, it's recommended to use this method. Converting C-style interface to a class is also quite straightforward, and discussed below.

In a header file (e.g. **examples/advect1d-newapi/gas_compress.hxx**), first put

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

(do NOT include **boutmain.hxx**, as that defines the C-like interface and a `main()` function).

Next define a class which inherits from `PhysicsModel`

```
class GasCompress : public PhysicsModel {
protected:
  int init(bool restarting);
  int rhs(BoutReal t);
private:
  // Evolving variables, parameters etc. here
};
```

As a minimum, you need to define the initialisation function `init` (it's a pure virtual member of PhysicsModel, so if you don't you'll get a compile-time error). Any variables being evolved should now be members of this class. If you are converting a C-style model, just move all the global variables into the `private` section.

Next create a source file (e.g. **examples/advect1d-newapi/gas_compress.cxx**, which includes your header file

```
#include "gas_compress.hxx"
```

Then implement the init and rhs functions:

```
int GasCompress::init(bool restarting) {
  ...
}

int GasCompress::rhs(BoutReal t) {
  ...
}
```

To convert simple physics models, just rename `physics_init` to `YourModel::init`, and `physics_run` to `YourModel::run`.

Finally, you need to create a `main()` function for your code. The easiest way to do this is to use the macro `BOUTMAIN`:

```
BOUTMAIN(GasCompress);
```

This is defined in **include/bout/physicsmodel.hxx**, and expands to

```
  int main(int argc, char **argv) {
```

```
2     BoutInitialise(argc, argv); // Initialise BOUT++
3
4     GasCompress *model = new GasCompress(); // Create a model
5
6     Solver *solver = Solver::create(); // Create a solver
7     solver->setModel(model); // Specify the model to solve
8     solver->addMonitor(bout_monitor); // Monitor the solver
9
10    solver->solve(); // Run the solver
11
12    delete model;
13    delete solver;
14    BoutFinalise(); // Finished with BOUT++
15    return 0;
16  }
```

If you like, you can define your own `main()` function, making it easier to combine BOUT++ with other libraries.

11 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 $\nabla_{||}$).

11.1 Differencing methods

Methods are implemented on 5-point stencils, and are divided into three categories:

- Central-differencing methods, for diffusion operators $\frac{df}{dx}$, $\frac{d^2f}{dx^2}$. Each method has a short code, and currently include
  - `C2`: 2^{nd} order $f_{-1} - 2f_0 + f_1$

  - `C4`: 4^{th} order $\left(-f_{-2} + 16f_{-1} - 30f_0 + 16f_1 - f_2\right)/12$
  - `W2`: 2^{nd} order CWENO
  - `W3`: 3^{rd} order CWENO
  - `FFT`: Fourier Transform method in Z (axisymmetric) direction only

- Upwinding methods for advection operators $v_x \frac{df}{dx}$
  - `U1`: 1^{st} order upwinding
  - `U4`: 4^{th} order upwinding
  - `W3`: 3^{rd} order Weighted Essentially Non-Oscillatory (WENO)[7]

- Flux conserving and limiting methods for terms of the form $\frac{d}{dx}\left(v_x f\right)$
  - `SPLIT`: split into upwind and central terms $\frac{d}{dx}\left(v_x f\right) = v_x \frac{df}{dx} + f\frac{dv_x}{dx}$
  - `NND`: Non-oscillatory, containing No free parameters and Dissipative (NND) scheme[8]

Both of these 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.

To use these differencing operators directly, add the following to the top of your physics module

```
1 #include <derivs.hxx>
```

By default the method used will be the one specified in the options input file (see section 6.7), but most of these methods can take an optional `DIFF\_METHOD` argument, specifying exactly which method to use.

11.2 Non-uniform meshes

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:

$$\frac{\partial f}{\partial x} \simeq \frac{1}{\Delta x}\frac{\partial f}{\partial i}$$

where i is the cell index number. The second derivative is therefore given by

$$\frac{\partial^2 f}{\partial x^2} \simeq \frac{1}{\Delta x^2}\frac{\partial^2 f}{\partial i^2} + \frac{1}{\Delta x}\frac{\partial f}{\partial x}\cdot\frac{\partial}{\partial i}\left(\frac{1}{\Delta x}\right)$$

Table 8: Coordinate derivatives

Function	Formula
DDX(f)	$\partial f/\partial x$
DDY(f)	$\partial f/\partial y$
DDZ(f)	$\partial f/\partial z$
D2DX2(f)	$\partial^2 f/\partial x^2$
D2DY2(f)	$\partial^2 f/\partial y^2$
D2DZ2(f)	$\partial^2 f/\partial z^2$
D2DX4(f)	$\partial^4 f/\partial x^4$
D2DY4(f)	$\partial^4 f/\partial y^4$
D2DZ4(f)	$\partial^4 f/\partial z^4$
D2DXDZ(f)	$\partial^2 f/\partial x \partial z$
D2DYDZ(f)	$\partial^2 f/\partial y \partial z$
VDDX(f, g)	$f\partial g/\partial x$
VDDY(f, g)	$f\partial g/\partial y$
VDDZ(f, g)	$f\partial g/\partial z$
FDDX(f, g)	$\partial/\partial x\,(f * g)$
FDDY(f, g)	$\partial/\partial x\,(f * g)$
FDDZ(f, g)	$\partial/\partial x\,(f * g)$

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

$$\texttt{d2x} = \frac{\partial \Delta x}{\partial i} = \frac{\partial^2 x}{\partial i^2}$$

The correction factor is then calculated from `d2x` using

$$\frac{\partial}{\partial i}\left(\frac{1}{\Delta x}\right) = -\frac{1}{\Delta x^2}\frac{\partial \Delta x}{\partial i}$$

11.3 Operators

These are differential operators which are independent of the coordinate system used.

$\mathbf{v}$	$=$	∇f	`Vector`	$=$	`Grad(Field)`
f	$=$	$\nabla \cdot \mathbf{a}$	`Field`	$=$	`Div(Vector)`
$\mathbf{v}$	$=$	$\nabla \times \mathbf{a}$	`Vector`	$=$	`Curl(Vector)`
f	$=$	$\mathbf{v} \cdot \nabla g$	`Field`	$=$	`V_dot_Grad(Vector, Field)`
$\mathbf{v}$	$=$	$\mathbf{a} \cdot \nabla \mathbf{b}$	`Vector`	$=$	`V_dot_Grad(Vector, Vector)`
f	$=$	$\nabla^2 f$	`Field`	$=$	`Laplacian(Field)`

$$\begin{aligned}
\nabla\phi &= \frac{\partial\phi}{\partial u^i}\nabla u^i \Rightarrow (\nabla\phi)_i = \frac{\partial\phi}{\partial u^i} \\
\nabla\cdot A &= = \frac{1}{J}\frac{\partial}{\partial u^i}\left(Jg^{ij}A_j\right) \\
\nabla^2\phi &= \Gamma^i\frac{\partial\phi}{\partial u^i} + g^{ij}\frac{\partial^2\phi}{\partial u^i\partial u^j} \\
&\quad \Gamma^i = \frac{1}{J}\frac{\partial}{\partial u^i}\left(Jg^{ij}\right)
\end{aligned}$$

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

$$\mathbf{B}_0 = \nabla z\times\nabla x \qquad |B_0| = \frac{\sqrt{g_{yy}}}{J}$$

Table 9: Clebsch operators

Function	Formula
`Grad_par`	$\partial_{\|\|}^0 = \mathbf{b}_0\cdot\nabla = \frac{1}{\sqrt{g_{yy}}}\frac{\partial}{\partial y}$
`Div_par`	$\nabla_{\|\|}^0 F = B_0\partial_{\|\|}^0\left(\frac{F}{B_0}\right)$
`Grad2_par2`	$\partial_{\|\|}^2\phi = \partial_{\|\|}^0\left(\partial_{\|\|}^0\phi\right) = \frac{1}{\sqrt{g_{yy}}}\frac{\partial}{\partial y}\left(\frac{1}{\sqrt{g_{yy}}}\right)\frac{\partial\phi}{\partial y} + \frac{1}{g_{yy}}\frac{\partial^2\phi}{\partial y^2}$
`Laplace_par`	$\nabla_{\|\|}^2\phi = \nabla\cdot\mathbf{bb}\cdot\nabla\phi = \frac{1}{J}\frac{\partial}{\partial y}\left(\frac{J}{g_{yy}}\frac{\partial\phi}{\partial y}\right)$
`Laplace_perp`	$\nabla_\perp^2 = \nabla^2 - \nabla_{\|\|}^2$
`Delp2`	Perpendicular Laplacian, neglecting all y derivatives
	The `Laplacian` solver performs the inverse operation

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

$$\nabla_\perp \equiv \nabla - \underline{b}\,(\underline{b}\cdot\nabla) \qquad \underline{b}\cdot\nabla = \frac{1}{JB}\frac{\partial}{\partial y}$$

$$\underline{b} = \frac{1}{JB}\underline{e}_y = \frac{1}{JB}\left[g_{xy}\nabla x + g_{yy}\nabla y + g_{yz}\nabla z\right]$$

In a Clebsch coordinate system $\underline{B} = \nabla z \times \nabla x = \frac{1}{J}\underline{e}_y$, $g_{yy} = \underline{e}_y \cdot \underline{e}_y = J^2 B^2$, and so the ∇y term cancels out:

$$\begin{aligned} \nabla_\perp &= \nabla x \left(\frac{\partial}{\partial x} - \frac{g_{xy}}{(JB)^2} \frac{\partial}{\partial y} \right) \\ &+ \nabla z \left(\frac{\partial}{\partial z} - \frac{g_{yz}}{(JB)^2} \frac{\partial}{\partial y} \right) \end{aligned}$$

11.4 Setting differencing method

12 Staggered grids

Until now all quantities have been cell-centred i.e. both velocities and conserved quantities were defined at the same locations. This is because these methods are simple and this was the scheme used in the original BOUT. This class of methods can however be susceptable 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 quantitity can be defined in BOUT++: centre, lower X, lower Y, and lower Z. These are illustrated in figure 3. To specify the location of a variable, use the method `setLocation()` with one of the locations `CELL\_CENTRE`, `CELL\_XLOW`, `CELL\_YLOW`, or `CELL\_ZLOW`.

> **NOTE**: If setting the location of an evolving variable, this should be done **before** the call to `bout_solve` or `SOLVE_FOR`

The key lines in the **test-staggered** example which specify the locations of the evolving variables are

```
1 Field3D n, v;
2
3 int physics_init(bool restart) {
4   v.setLocation(CELL_YLOW); // Staggered relative to n
5   SOLVE_FOR2(n, v);
6   ...
```

which makes the velocity `v` staggered to the lower side of the cell in Y, whilst the density n remains cell centred.

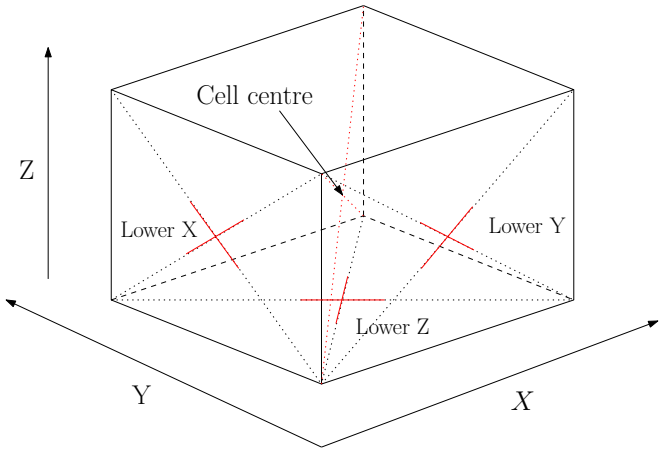


Figure 3: Locations in a grid cell where quantities may be defined.

Arithmetic operations between staggered quantities are handled by interpolating them to the same location according to the algorithm in figure 4. If performing an operation

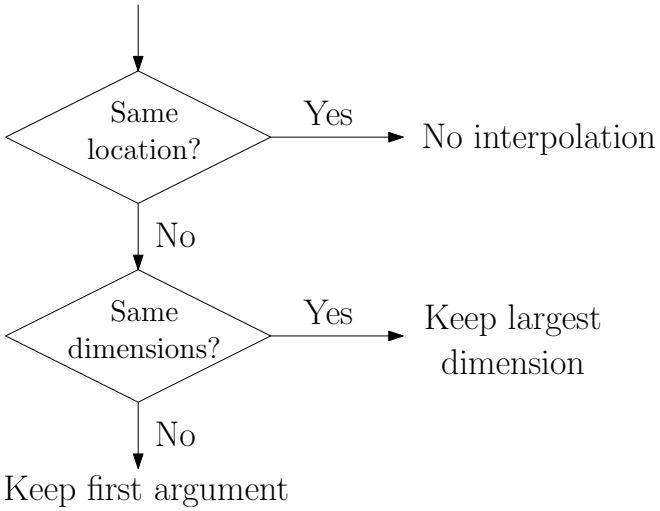


Figure 4: How the cell location of an arithmetic operation (`+,-,*,/,^`) is decided

between variables defined at two different locations, the order of the variables matter: the result will be defined at the locations of the **left** variable. For example, `n*v` would be `CELL_CENTRE` because this is the location of `n`, whilst `v*n` would be `CELL_YLOW`. Relying on this behavior could lead to trouble, to make your code clearer it's probably best to use the interpolation routines. Include the header file

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

then use the `interp_to(field, location)` function. Using this, `interp_to(n, CELL_YLOW)*v` would be `CELL_YLOW` as `n` would be interpolated.

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. As with the arithmetic operators, if you ask for the result to be staggered in a different direction from the input then the differencing will be to cell centre and then be interpolated. For example `Grad_par(v, CELL_XLOW)` would first perform staggered differencing from `CELL_YLOW` to get a result at `CELL_CENTRE`, and then interpolate the result to `CELL_XLOW`.

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, and if both are staggered to different locations then the behavior is less well defined (don't do it). As with other differential operators, the required location of the result can be given as an optional argument.

> **NOTE**: There are subtleties with boundary conditions when staggering variables. The test-staggered example manually applies a boundary condition to make the width of the boundary wider

13 Advanced methods

This section describes the more advanced methods which can be used to speed up simulations using implicit time stepping schemes. At the time of writing (Dec '12), they can be used with either the SUNDIALS CVODE or PETSc solvers.

13.1 Preconditioning

At every time step, an implicit scheme such as BDF has to solve a nonlinear 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 δt gets large compared with some timescales 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 timescales.

A simple example[4] is a coupled wave equation, solved in the `test-precon` example code:

$$\frac{\partial u}{\partial t} = \partial_{||} v \qquad \frac{\partial v}{\partial t} = \partial_{||} u$$

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

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

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

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

In general for nonlinear functions $\mathcal{J}$ gives the change in time-derivatives in response to changes in the state variables u and v.

In implicit timestepping, the preconditioner needs to solve an equation

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

where $\mathcal{I}$ is the idensity matrix, and γ depends on the time step and method (e.g. $\gamma = \delta t^2$ 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[5]

$$\begin{pmatrix} \mathbf{E} & \mathbf{U} \\ \mathbf{L} & \mathbf{D} \end{pmatrix}^{-1} = \begin{pmatrix} \mathbf{I} & -\mathbf{E}^{-1}\mathbf{U} \\ 0 & \mathbf{I} \end{pmatrix}\begin{pmatrix} \mathbf{E}^{-1} & 0 \\ 0 & \mathbf{P}_{Schur}^{-1} \end{pmatrix}\begin{pmatrix} \mathbf{I} & 0 \\ -\mathbf{L}\mathbf{E}^{-1} & \mathbf{I} \end{pmatrix}$$

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

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

The preconditioner is implemented by defining a function of the form

[4] Taken from a talk by L.Chacon available here `https://bout.llnl.gov/pdf/workshops/2011/talks/Chacon_bout2011.pdf`

[5] See paper `http://arxiv.org/abs/1209.2054` for an application to 2-fluid equations

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

which takes as input the current time, the γ factor appearing above, and δ 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 `ddt(u)` and `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:

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

NOTE: This changed in github/bendudson on 15th Aug 2014. In older versions the result must be returned in the state variables

To implement the preconditioner in equation 1, first apply the rightmost matrix to the given vector:

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

```
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 `ddt(u)`, it first needs to be communicated to exchange guard cell values.

The second matrix

$$\begin{pmatrix} \texttt{ddt(u)} \\ \texttt{ddt(v)} \end{pmatrix} \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & \left(1 - \gamma^2\partial_{||}^2\right)^{-1} \end{pmatrix} \begin{pmatrix} \texttt{ddt(u)} \\ \texttt{ddt(v)} \end{pmatrix}$$

doesn't alter u, but solves a parabolic equation in the parallel direction. There is a solver class to do this called `InvertPar` which solves the equation $\left(A + B\partial_{||}^2\right) x = b$ where A and B are `Field2D` or constants[6]. In `physics_init` we create one of these solvers:

```
InvertPar *inv; // Parallel inversion class
int physics_init(bool restarting) {
  ...
```

[6]This `InvertPar` class can handle cases with closed field-lines and twist-shift boundary conditions for tokamak simulations

```
  inv = InvertPar::Create();
  inv->setCoefA(1.0);
  ...
}
```

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

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

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

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

```
  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

```
  ddt(u).applyBoundary("dirichlet");
  ddt(v).applyBoundary("dirichlet");
```

To use the preconditioner, pass the function to the solver in `physics_init`

```
1 int physics_init(bool restarting) {
2   solver->setPrecon(precon);
3   ...
4 }
```

then in the `BOUT.inp` settings file switch on the preconditioner

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

13.2 Jacobian function

13.3 DAE constraint equations

Using the IDA solver, BOUT++ can solve Differential Algebraic Equations (DAEs), in which algebraic constraints are used for some variables.

13.4 Monitoring the simulation output

At every output timestep the solver calls a monitor function, 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 add monitor function(s).

You can call your monitor function whatever you like, but it must have 4 inputs and return an int:

```
int my_monitor(Solver *solver, BoutReal simtime, int iter, int NOUT) {
  ...
}
```

The first input is the solver object, the second is the current simulation time, the third is the output number, and the last is the total number of outputs requested. To get the solver to call this function every output time, put in your `physics\_init` code:

```
  solver->addMonitor(my_monitor);
```

If you want to later remove a monitor, you can do so with

```
  solver->removeMonitor(my_monitor);
```

A simple example using this monitor is:

```
int my_monitor(Solver *solver, BoutReal simtime, int iter, int NOUT) {
  output.write("My monitor, time = %e, dt = %e\n",
      simtime, solver->getCurrentTimestep());
}

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

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

14 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 [1, 2] were `drift-instability`, `interchange-instability` and `orszag-tang`.

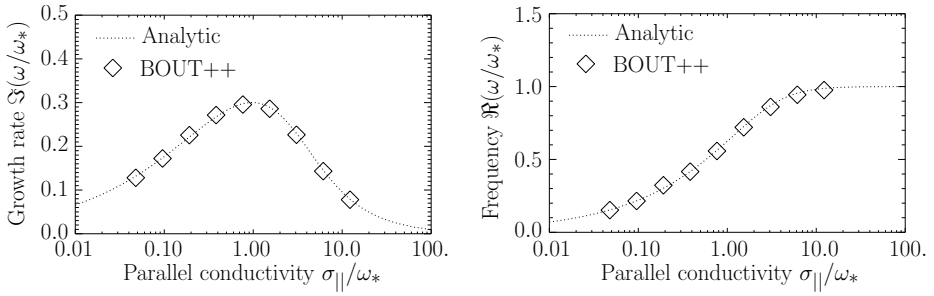
14.1 advect1d

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

14.2 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.



(a) Growth rate (b) Real Frequency

Figure 5: Resistive Drift wave instability test. Dashed lines are analytic results, diamonds from BOUT++ simulations

14.3 em-drift

14.4 gyro-gem

14.5 interchange-instability

14.6 jorek-compare

14.7 lapd-drift

14.8 orszag-tang

The file `mhd.cxx` solves the full MHD equations for the full values (perturbation + initial), whilst the file `mhd_perturb.cxx` solves for a perturbation about the equilibrium.

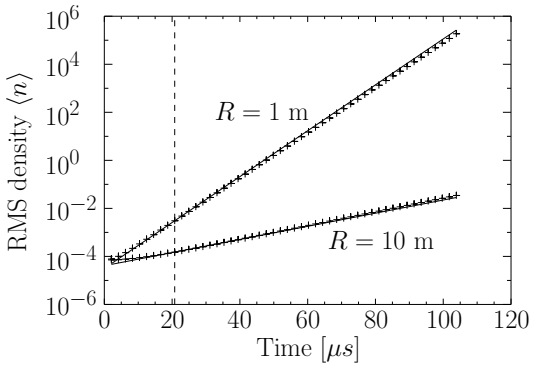


Figure 6: Interchange instability test. Solid lines are from analytic theory, symbols from BOUT++ simulations, and the RMS density is averaged over z. Vertical dashed line marks the reference point, where analytic and simulation results are set equal

14.9 shear-alfven-wave

14.10 sod-shock

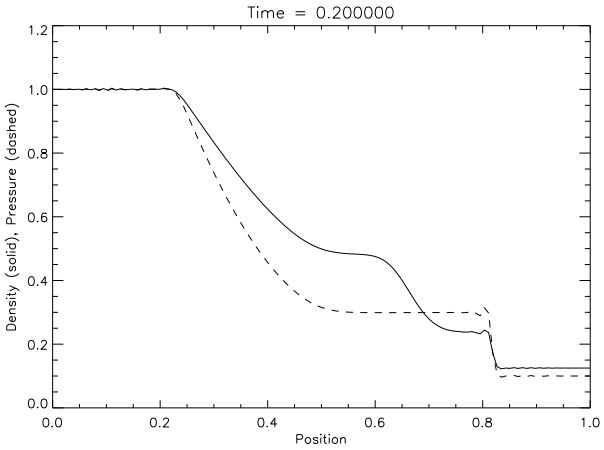


Figure 7: Sod shock-tube problem for testing shock-handling methods

14.11 uedge-benchmark

15 Notes

15.1 Compile options

Compiling with `-DCHECK` 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.

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

15.2 Adaptive grids

Two types of adaptive grids can be used in BOUT++: Moving meshes, and changing resolution.

15.2.1 Moving meshes

During either the initialisation, or the simulation itself, the metric tensors can be modified. This could be used to make the coordinate system time-dependent. Since currently the metric tensors are 2D fields, this would only allow axi-symmetric motion. Changing the tensors to be 3D objects is however possible with fairly small modification to the code.

Whenever one of the metrics g^{ij} are changed, a call to `geometry()` must be made.

15.2.2 Changing resolution

NOTE: Not implemented yet - this just for discussion

Since all 2D and 3D fields/vectors are located internally in global lists, the resolution of the grid can be changed when required by interpolation. **This requires a new, more efficient implementation of the Fields classes**.

References

[1] B.D. Dudson, M.V. Umansky, X.Q. Xu, P.B. Snyder, and H.R. Wilson. Bout++: A framework for parallel plasma fluid simulations. *Computer Physics Communications*, In Press, Corrected Proof:–, 2009.

[2] B D Dudson, M V Umansky, X Q Xu, P B Snyder, and H R Wilson. BOUT++: a framework for parallel plasma fluid simulations. *arXiv*, physics.plasm-ph:0810.5757, Nov 2008.

[3] M V Umansky, X Q Xu, B Dudson, and L L LoDestro. *BOUT Code Manual*. LLNL, June 2006. Available from www.mfescience.org/bout/.

[4] M V Umansky, X Q Xu, B Dudson, L L LoDestro, and J R Myra. Status and verification of edge plasma turbulence code bout. *Comp. Phys. Comm.*, page doi:10.1016/j.cpc.2008.12.012, 2008.

[5] X Q Xu, M V Umansky, B Dudson, and P B Snyder. Boundary plasma turbulence simulations for tokamaks. *Comm. in Comput. Phys.*, 4(5):pp. 949–979, November 2008.

[6] W H Press, S A Teukolsky, W T Vetterling, and B P Flannery. *Numerical recipes in C. The art of scientific computing*. Cambridge University Press, 1999.

[7] Guang-Shan Jiang and Danping Peng. Weighted ENO schemes for Hamilton-Jacobi equations. *SIAM J. Sci. Comp.*, 21(6):2126–2143, 2000.

[8] F Chen, A Xu, G Zhang, and Y Li. Three-dimensional lattice boltzmann model for high-speed compressible flows. *arXiv*, page 1010.4135v1, 2010.

A Installing PACT

There are two ways to install PACT, and usually one of them will work on a given system.

A.1 Self-extracting package

This is probably the easiest method (when it works). Download one of the "Executable UNIX distribution files" from the PACT website and run:

```
./pact07_07_18-src -sl -i $HOME/local/
```

The "-sl" flag tells it to generate shared libraries. If you don't plan on using IDL to read/write PDB files, then you can omit this. The "-i $HOME/local/" tells PACT to install in your home directory/local.

If this script fails, you will usually have to resort to either trying to understand DSYS, or going with the second method below.

A.2 PACT source distribution

The second method is to use a .tar.gz PACT source file. Here the version used is `pact-2.1.0.tar.gz`.

```
~/ $ cd install
~/install/ $ tar -xzvf pact-2.1.0.tar.gz
~/install/ $ cd pact-2.1.0/
~/install/pact-2.1.0/ $ ./configure --prefix=$HOME/local --enable-shared
```

> **NOTE**: On Franklin, PACT will compile without the –enable-shared option, but not with it. This is OK if you just want to run BOUT++, but the shared libraries are needed for reading the results into IDL (the PDB2IDL library)

At this point, the installation may fail with the following error:

```
configure: WARNING: yacc is a symbolic link to bison
configure: WARNING: bison is not a supported type of yacc
configure: error: No working yacc found
```

If this happens, you need to first install Berkeley Yacc into your home directory

```
~/install/ $ ls
byacc.tar.gz       netcdf-tar -xzvf byacc.tar.gz4.0.1.tar.gz  pact-2.1.0.tar.gz
fftw-3.2.1.tar.gz  pact-2.1.0           sundials-2.4.0.tar.gz

~/install/ $ tar -xzvf byacc.tar.gz
~/install/ $ cd byacc-20080826/
~/install/byacc-20080826/ $ ./configure --prefix=$HOME/local
~/install/byacc-20080826/ $ gmake
~/install/byacc-20080826/ $ mkdir ~/local/bin
~/install/byacc-20080826/ $ cp yacc ~/local/bin/
```

NB: We're copying the yacc executable manually because "gmake install" doesn't seem to work, and the fix which works for PACT (see later) doesn't work here.

Add this directory to your path:

```
~/install/byacc-20080826/ $ setenv PATH $HOME/local/bin:$PATH
```

You can check that this has worked by running "which yacc", which should then print your home directory /local/bin/yacc. You could also add this to your .profile startup scripts. Now go back to PACT:

```
~/install/byacc-20080826/ $ cd ../pact-2.1.0
~/install/pact-2.1.0/ $ ./configure --prefix=$HOME/local --enable-shared
~/install/pact-2.1.0/ $ gmake
~/install/pact-2.1.0/ $ gmake install
```

The last step may fail with a strange error message like:

```
The current directory must be set to the ITT directory.
Change the default to the ITT directory and re-run
this script.
```

This happens when the wrong "install" is being used. Check by running:

```
~/install/byacc-20080826/ $ which install
```

This should print "/usr/bin/install", but if not then run

```
~/install/byacc-20080826/ $ ln -s /usr/bin/install ~/local/bin/
~/install/pact-2.1.0/ $ ./configure --prefix=$HOME/local
~/install/pact-2.1.0/ $ gmake
~/install/pact-2.1.0/ $ gmake install
```

NOTE: configure needs to be run again after messing with install.
This should now install PACT into your local directory.

B 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.

B.1 SUNDIALS

To compile SUNDIALS, use

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

You may get an error message like:

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

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

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

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

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

If you see an error like this:

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

then you need to set the environment variable `OBJECT_MODE`

```
export OBJECT_MODE=64
```

Configuring BOUT++, you may get the error:

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

In that case, you can try using:

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

When compiling, you may see warnings

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

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

```
ld: 0711-317 ERROR: Undefined symbol: .NcError::NcError(NcError::Behavior)
ld: 0711-317 ERROR: Undefined symbol: .NcFile::is_valid() const
ld: 0711-317 ERROR: Undefined symbol: .NcError::~NcError()
ld: 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, raplacing `-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
```

C 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)`
  $\nabla_\perp^2$ operator
- `Field =` **Div**`(Vector)`
  Divergence of a vector
- `Field =` **Div_par**`(Field f)`
  Parallel divergence $B_0\mathbf{b}\cdot\nabla\left(f/B_0\right)$
- **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*x + B*Grad2_par2(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 $\mathbf{v} \cdot \nabla f$

- `Vector = V_dot_Grad(Vector v, Vector u)`
  Advection term $\mathbf{v} \cdot \nabla \mathbf{u}$

- `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`.

D IDL routines

List of IDL routines available in idllib. There are broadly three categories of routine:

- Completely general routines which could be useful outside BOUT++ work
  - Data plotting and animation: **contour2** and **showdata**
  - File reading and writing: **file_open**, **file_read** etc.
  - User input and output: **get_float**, **get_integer**, **get_yesno** and **str**
  - FFT routines for integrating, differentiating and filtering: **fft_integrate**, **fft_deriv**, **fft_filter**

- Routines for BOUT++, but not specific to any application
  - Modifying restart files: **expand_restarts**, **scale_restarts** and **split_restarts**
  - Processing 3D variables for input grid: **bout3dvar**

- Routines specifically for tokamak simulations
  - Reading A- and G-EQDSK format files into IDL: **read_aeqdsk** and **read_neqdsk**
  - Plotting results: **polslice**, **plotpolslice**

  Here the format is
  name, arguments, [optional arguments]

- var = **bout3dvar** (var)
  Converts 3D variables to and from BOUT++'s Fourier representation which is used for input grids. By default converts from [x,y,z] to [x,y,f]
  - **/reverse** Convert from [x,y,f] to [x,y,z]

  - **nf**=nf Set number of frequencies in the result
  - **nz**=nz When using /reverse, set number of Z points in the result

- var = **collect**()
  Read in data from a set of BOUT++ dump files
  - **var** = "name of variable"
  - **path** = "path/to/variable/"
  - **xind**, **yind**, **zind**, **tind** = [min, max] index pairs
  - **t_array** = Output 1D array of times

- **contour2**, data [, x, y]
  This is a replacement for the IDL contour which includes a scale color bar.
  - **data** can be either 2D (x,y) or 3D (x,y,t). If data is 3D then the color is scaled to the entire range.
  - **x** is an optional 2D (x,y) array of X coordinates
  - **y** is an optional 2D (x,y) array of Y coordinates
  - **t**=t is a time index for 3D data
  - **nlev**=nlev
  - **centre**=centre Make zero the middle of the color range (white if redblue)
  - **redblue**=redblue Use a blue-white-red color scheme
  - **revcolor**=revcolor Reverse color scheme

- **expand_restarts**, newz
  Increases the number of Z points in restart files. Together with scale_restarts and split_restarts, this makes it easier to modify a linear simulation as a start for nonlinear runs.
  - **newz** is the new value of NZ
  - **path**=path Input path
  - **output**=output Output path
  - **format**=format File extension of output

- result = **fft_deriv** (var1d)
  Calculates the derivative of a variable on a periodic domain.

- result = **fft_filter** (var, nf) Fourier filter a variable on a periodic domain. Arguments are a 1D variable and the number of Fourier components to keep
- result = **fft_integrate** (var1d) Integrates a variable on a periodic domain.
  - **loop**=loop The loop integral is returned in this variable
- **file_close**, handle
  Close a file opened using file_open()
- list = **file_list** (handle)
  Return a list of variable names in the file
- integer = **file_ndims** (handle , "variable")
  Get the number of dimensions of a variable
- handle = **file_open** ("file")
  Open a PDB or NetCDF file. File type is inferred from file name
  - **/write** Open file for writing (default is read only)
  - **/create** Create a new file, over-writing if already exists
- var = **file_read** (handle, "variable")
  - **inds** = [xmin, xmax, ymin, ymax, ...]
- float = **get_float** ("prompt")
  Ask the user for a float, using the given prompt
- integer = **get_integer** ("prompt")
  Ask the user for an integer
- integer = **get_yesno** ("prompt")
  Ask for a yes (1) or no (0) answer
- result = **gmres** (x0, operator, b)
  General Minimal Residual (GMRES)
  - **x0** is the starting guess at the solution
  - **operator**
  - **b**

  Optional arguments

  - **restart**=restart
  - **max_iter**=max_iter
  - **tol**=tol
  - **stats**=stats
  - **show**=show
  - **output**=output

- result = **int_func** ([x,] f)
  Integrate a function, always using the maximum number of grid-points possible for highest accuracy

- bool = **is_pow2** (value)
  Returns 1 (true) if the given number is a power of 2, 0 (false) otherwise

- **plotpolslice**, var3d, grid
  Takes a slice through a field-aligned tokamak domain, showing a poloidal cross-section.

  - **var3d** is a 3D (x,y,z) variable to plot. Needs all of the points to work properly.
  - **grid** is a structure from importing a grid file

  Optional arguments:

  - **period**=period
  - **zangle**=zangle
  - **nlev**=nlev
  - **yr**=yr
  - **profile**=profile
  - **output**=output
  - **lines**=lines
  - **linecol**=linecol
  - **filter**=filter

- **polslice**, data, gridfile
  Plots a 2D poloidal contour for single or double-null configurations, including color bar.

  - **xstart**=xstart X index where the data begins. Useful if only part of the domain has been collected

  - **ystart**=ystart Y index where data begins

- struct = **read_aeqdsk** ("filename")
  Reads an A-EQDSK file. Format is specified here: `https://fusion.gat.com/THEORY/efit/a_eqdsk.html`

- struct = **read_neqdsk** ("filename")
  Reads in an 'neqdsk' or G-EQDSK formatted tokamak equilibrium file. Format of G-EQDSK file is specified here: `https://fusion.gat.com/THEORY/efit/g_eqdsk.html`

- stringarray = **regex_extract** (line, pattern)
  Extract all matches to Regular Expression pattern contained in line. Useful for extracting numbers from FORTRAN-formatted text files.

  - **line** Input string
  - **pattern** Regular expression pattern to match
  - **nmatch**=nmatch

- var = **reverse_inds** (var)
  Reverse array indices e.g. `arr[t,z,y,x] -> arr[x,y,z,t]`. Works on up to 5 dimensional variables

- **safe_colors**
  Sets the color table to useful values for plotting.

  - **/first** Sets the first 10 colors to specific values, otherwise sets last 7

- **scale_restarts**, factor

  - **path**=path Path to the restart files (default is current directory '.')
  - **format**=format Specify what the file format is, otherwise goes on the file name

- **showdata**, data
  Display animations of 1D,2D and 3D data. Defaults:

  - 2D data Animate a line plot
  - 3D data Animate a surface plot
  - 4D data Animate a poloidal cross-section (tokamaks only)

  Optional arguments:

  - **/addsym** For 2D data (1D plots), add symbols to mark data points
  - **az**=angle Rotate surface plots
  - **/bw** Make contour plots greyscale
  - **chars**=size character size
  - **/contour** For 3D input, show color contour plot
  - **delay**=time Time delay between plots (default 0.2 seconds)
  - **/noscale** By default, all plots are on the same scale. This changes the scale for each plot's range
  - **profile**=array Background profile. Data is 3D: profile is 1D (X). Data is 4D -¿ profile is 2D (X,Y)
  - **yr**=[min,max] Y range

- result = **sign** (var)
  This returns +1 if the variable is > 0, -1 otherwise

- **spectrum**

- **split_restarts**, [nxpe], nype
  split restart files between a different number of processors
  - **nxpe** is an optional argument giving the number of processors in the X direction
  - **nype** is the number of processors in the Y direction
  - **path**=path Input path
  - **output**=output Output path
  - **format**=format File extension of output

- string = **str** (value)
  Convert a value to a string with whitespace trimmed. Arrays are converted to a comma-separated list in brackets.

- result = **zfamp** (var4d)
  Given a 4D variable [x,y,z,t], returns the Fourier amplitudes in [x,y,f,t]

- var = **zshift** (var, shift)
  Shifts a variable in the Z direction, useful for mapping between field-aligned and orthogonal coordinates.
  - **period**=period How many domains fit in 2π. Default is 1 (full torus)

E Python routines (alphabetical)

E.1 boututils

- `class Datafile` provides a convenient way to read and write NetCDF files. There are many different NetCDF libraries available for Python, so this class tries to provide a consistent interface to many of them.
- `deriv`()
- `determineNumberOfCPUs`()
- `file_import`() reads the contents of a NetCDF file into a dictionary
- `integrate`()
- `launch`()
- `linear_regression`()

E.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.

```
1    from boutdata import collect
2
3    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

Index