BOUT++ Developers' Manual

B.Dudson, University of York

December 17, 2013

Contents

1 Introduction

This is a manual describing the core BOUT++ code[**?**, **?**], and is intended for anyone who wants to work on improving BOUT++. It does its best to describe the details of how BOUT++ works, and assumes that the user is very comfortable with C++. For a general introduction, and instructions for using BOUT++ see the users' guide. The user's guide assumes only minimal knowledge of C++, and provides only those details needed to use BOUT++.

Since BOUT++ is a scientific code, it is constantly changing and (hopefully) being improved. This provides a moving target for documentation, and means that describing all the details of how BOUT++ works in one document is probably impossible. This is particularly true since often our first priority is to write papers and code - not documentation - and so whatever is documented is likely to be slightly out of date. A source of up-to-date documentation of the BOUT++ code is the comments and Doxygen tags: running `doxygen` on the source should produce a set of HTML documentation. See **www.doxygen.org** for more details.

1.1 Using the BOUT++ repository

As of June 2009, the BOUT++ distribution is hosted on Github:

http://github.com/bendudson/BOUT/

For a full guide to using Git, see the git website[1] or online tutorials. This manual just explains some basic ways to use Git, and the recommended work flow when working with BOUT++.

If you're just starting with BOUT++, current developers will want to check your changes before submitting them to the repository. In this case you should clone (checkout) the git repository, make any changes and then submit patches to one of the developers (e.g. to `bd512@york.ac.uk`). Fortunately Git makes this process quite easy: First get a copy of BOUT++

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

[1] `http://git-scm.com/`

The BOUT++ repository will now be in a directory called "BOUT" (sorry - github doesn't like '+' in project names). To get the latest changes (`svn update` equivalent), use

```
$ git pull
```

To see the status of the repository, commits etc. use

```
$ gitk
```

This is also useful for showing what changes you've made which need to be committed, or which haven't yet been sent to the main repository.

You can make edits as normal, and commit them using

```
$ git commit -a
```

which is pretty much the equivalent of `svn commit` in that it commits all changes, though importantly it doesn't send them to a central server. To see which changes will be committed, use

```
$ git status
```

To choose which files you want to commit, use

```
$ git add file1, file2, ...
$ git commit
```

(Git can actually only commit selected parts of files if you want). To make using Git easier, you can create a config file **$HOME/.gitconfig** containing:

```
[user]
  name = Ben Dudson
  email = bd512@york.ac.uk

[alias]
  st = status
  ci = commit
  br = branch
  co = checkout
  df = diff
  lg = log -p
  who = shortlog -s --
```

(though obviously you should change the name and email).

Once you're done making changes, you should first pull the latest changes from the server:

```
$ git pull
```

Read carefully what git prints out. If there are conflicts then git will try to resolve them, but in some cases you will have to resolve them yourself. To see a list of conflicting changes run `git status` (or `git st` if you're using the above **.gitconfig** file). Once you've finished resolving conflicts, run `git commit -a` to commit the merge.

After you've got the latest changes, and resolved conflicts, create a patch:

```
$ git format-patch origin/master --stdout > your-patch-file.diff
```

Now you can email this patch to someone who can commit to the Git archive (e.g. Ben Dudson, bd512@york.ac.uk).

1.1.1 Developing BOUT++

If you are doing a lot of development of BOUT++, it will probably make sense for you to push changes directly to the online repository. In this case you'll need to sign up for an account on **github.com**, then upload an ssh key and ask to be added. The process is then almost identical except that you clone using SSH:

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

and rather than creating a patch, you push changes to the repository:

```
$ git push
```

1.1.2 Accessing github from behind a firewall

If you're working on a machine which can't access github directly (such as grendel, smaug etc. at LLNL), you can still seamlessly access github by using another machine as a proxy over SSH. To do this, edit your SSH config file `/.ssh/config` and add the following lines:

```
Host        gh
HostName    github.com
User        git
ProxyCommand    ssh -q -x user@euclid.nersc.gov nc %h %p
```

where `euclid.nersc.gov` can be replaced by any machine you can access which has netcat (`nc`) installed, and which can access github.com. If you have set up a github account with SSH keys, you should now be able to get a copy of BOUT++ by running

```
$ git clone gh:bendudson/BOUT
```

1.1.3 Creating a private repository

Whilst we would prefer it if improvements to BOUT++ were shared, sometimes you might want to keep changes private for a while before publishing them. Creating a private repository with Git is very simple, because every clone of a repository is itself a repository. Git doesn't have the concept of a central repository, which can seem strange coming from the world of SVN and CVS. What it means is that you can create your own private repository anywhere you have access to. Sharing it with only some people means as giving them read or write access to the repository directory.

The following assumes you have a NERSC account and want to create a private repository on Franklin. To apply this to a different machine just replace `franklin.nersc.gov` with the machine you want to put the repository on.

1. SSH to `franklin.nersc.gov`, or wherever you want your repository

   ```
   $ ssh username@franklin.nersc.gov
   ```

2. Create a "bare" Git repository by cloning a repository with the `--bare` option:

   ```
   $ cd ~
   $ git clone --bare git@github.com:bendudson/BOUT.git  bout_private
   ```

   where you can replace `git@github.com:bendudson/BOUT.git` with any other repository you can access. `bout_private` will be the name of the directory which will be created. This will make a repository without a working version. This means you can't modify the code in it directly, but can pull and push changes to it. If you want to work on the code on Franklin, make a clone of your private repository:

   ```
   $ git clone bout_private bout
   ```

   which creates a repository `bout` from your private repository. Running `git pull` and `git push` from within this new repository will exchange patches with your `bout_private` repository.

3. You can now clone, pull and push changes to your private repository over SSH e.g.

   ```
   $ git clone username@franklin.nersc.gov:bout_private
   ```

4. To keep your private repository up to date you may want to pull changes from github into your private repository. To do this, you need to use a third repository. Log into Franklin again:

   ```
   $ cd ~
   $ git clone bout_private bout_tmp
   ```

   This creates a repository `bout_tmp` from your private repository. Now cd to the new directory and pull the latest changes from github:

   ```
   $ cd bout_tmp
   $ git pull git://github.com/bendudson/BOUT.git
   ```

   Note: You should be able to access this repository from Franklin, but if not then see the previous subsection for how to access github from behind a firewall.

5. This pull might result in some conflicts which need to be resolved. If so, git will tell you, and running

   ```
   $ git status
   ```

   will give a list of files which need to be resolved. Edit each of the files listed, and when you're happy commit the changes

   ```
   $ git commit -a
   ```

6. Your `bout_tmp` directory now contains a merge of your private repository and the repository on github. To update your private repository, just push the changes back:

   ```
   $ git push
   ```

   You can now delete the `bout_tmp` repository if you want.

1.2 House rules

BOUT++ consists of about 39,400 lines of C/C++[2], along with 18,500 lines of IDL and 2,900 of Python. Of this, about 24,700 lines is the core BOUT++ code, and the remainder a mix of pre- and post-processors, and physics modules. As production codes go, this is not particularly huge, but it is definitely large enough that keeping the code 'clean' and understandable is necessary. This is vital if many people are going to work on the code, and also greatly helps code debugging and verification. There are therefore a few house rules to keep in mind when modifying the BOUT++ code.

When modifying the core BOUT++ code, please keep in mind that this portion of the code is intended to be general (i.e. independent of any particular physical system of equations), and to be used by a wide range of users. Making code clear is also more important in this section than the physics model since the number of developers is potentially much greater.

Here are some rules for editing the core BOUT++ code:

- **NO FORTRAN**. EVER. Though it may be tempting for scientific programmers to use a little FORTRAN now and then, please please don't put any into BOUT++. Use of FORTRAN, particularly when mixed with C/C++, is the cause of many problems in porting and modifying codes.
- If a feature is needed to study a particular system, only include it in the core code if it is more generally applicable, or cannot be put into the physics module.

1.3 Coding conventions

The naming of BOUT++ classes, functions etc. has not been very consistent historically, but please try to follow the following rules:

- Directories and files are all lower case (with underscores)
- Class names start with a capital letter, and each word is capitalised e.g. `BoutMesh`. No underscores.
- Class member functions start with a lower case letter with each subsequent word capitalised (e.g. `write()`, `setSection()`), also without underscores.
- Variable names are all lower case, with underscores if you want.

Curly braces should **not** start on a new line: please use

[2]generated using David A. Wheeler's 'SLOCCount'

```
if( ... ) {
  ...
}
```

and

```
type function( ... ) {
  ...
}
```

1.4 Testing

Two types of tests are currently used in BOUT++ to catch bugs as early as possible: Unit tests, which check a small piece of the code separately, and a test suite which runs the entire code on a short problem.

Unit tests can be run using the **src/unit_tests** Python script. This searches through the directories looking for an executable script called **unit_test**, runs them, and collates the results. Not many tests are currently available as much of the code is too tightly coupled. If done correctly, the unit tests should describe and check the behavior of each part of the code, and hopefully the number of these will increase over time.

The test suite is in the **examples** directory, and is run using the **test_suite** python script. At the top of this file is a list of the subdirectories to run (e.g. **test-io**, **test-laplace**, and **interchange-instability**). In each of those subdirectories the script **runtest** is executed, and the return value used to determine if the test passed or failed.

All tests should be short, otherwise it discourages people from running the tests before committing changes. A few minutes or less on a typical desktop, and ideally only a few seconds. If you have a large simulation which you want to stop anyone breaking, find starting parameters which are as sensitive as possible so that the simulation can be run quickly.

2 Code layout

BOUT++ is organised into classes and groups of functions which operate on them: It's not purely object-oriented, but takes advantage of many of C++'s object-oriented features.

Figure 1 shows the most important parts of BOUT++ and how they fit together. The initialisation process is shown in red: basic information is first read from the grid file (e.g. size of the grid, topology etc.), then the user-supplied initialisation code is called. This code can read other variables from the grid, and makes at least one call to `bout_solve` to specify a variable to be evolved. The main thing `bout_solve` does is to add these variables to the solver.

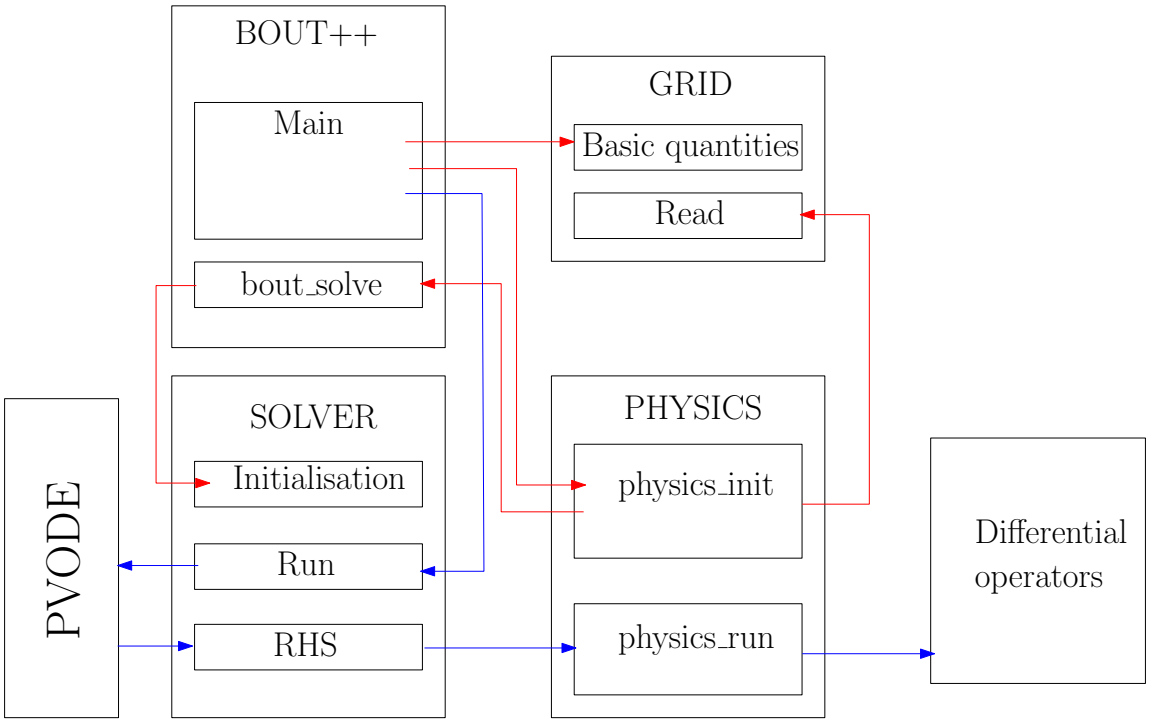

Figure 1: Overview of BOUT++ control flow during initialisation (red), and running (blue)

The process of running a timestep is shown in blue in figure 1: The main loop calls the solver, which in turn calls PVODE. To evolve the system PVODE makes calls to the RHS function inside solver. This moves data between PVODE and BOUT++, and calls the user-supplied `physics_run` code to calculate time-derivatives. Much of the work calculating time-derivatives involves differential operators.

Calculation of the RHS function `physics_run`, and handling of data in BOUT++ involves many different components. Figure 2 shows (most) of the classes and functions involved, and the relationships between them. Some thought was put into how this should be organised, but it has also changed over time, so some parts could be cleaner.

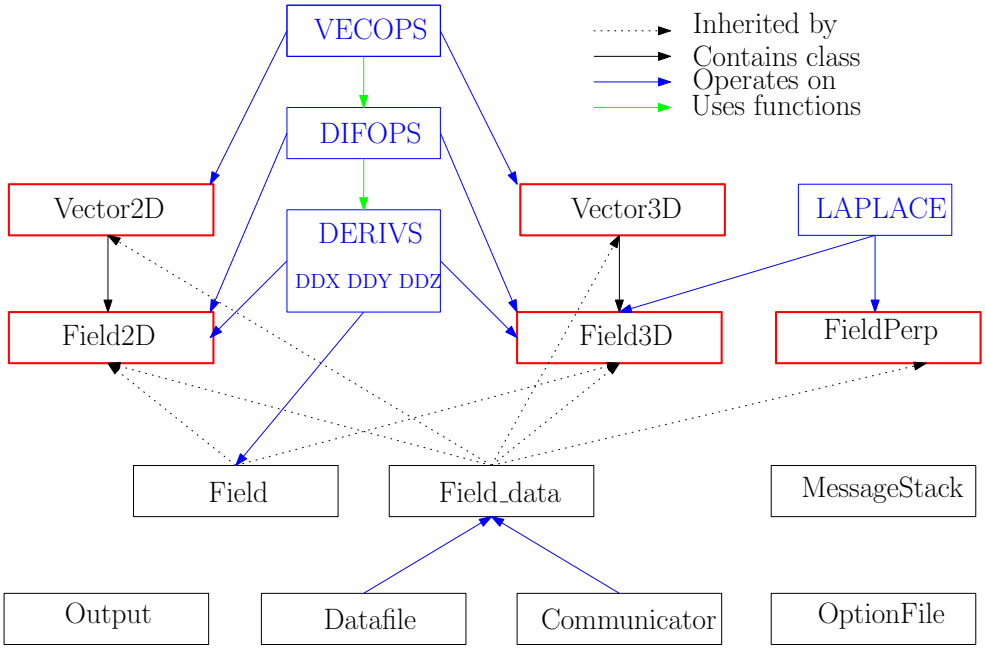

Figure 2: Relationship between important classes and functions used in calculating the RHS function

2.1 Directories

The source code for the core of BOUT++ is divided into include files (which can be used in physics models) in **bout++/include**, and source code and low-level includes in **bout++/src**. Many parts of the code are defined by their interface, and can have multiple different implementations. An example is the time-integration solvers: many different implementations are available, some of which use external libraries, but all have the same interface and can be used interchangeably. This is reflected in the directory structure inside

bout++/src. A common pattern is to store individual implementations of an interface in a subdirectory called `impls`.

```
include/foo.hxx
src/.../foo.cxx
src/.../foo_factory.hxx
src/.../foo_factory.cxx
src/.../impls/one/one.hxx
src/.../impls/one/one.cxx
```

where `foo.hxx` defines the interface, `foo.cxx` implements common functions used in several implementations. `foo_factory` creates new implementations, and is the only file which includes all the implementations. Individual implementations are stored in their own subdirectories of `impls`. Components which follow this pattern include `fileio` formats, `invert/laplace` and `invert/parderiv` inversion codes, `mesh`, and `solver`.

The current source code files are:

- **bout++.cxx** : Main file which initialises, runs and finalises BOUT++. Currently contains a `main()` function, though this is being removed shortly.
- field
  - **field2d.cxx** implements the `Field2D` class. This is a scalar field which varies only in x and y and is used for things like metric tensor components and initial profiles. It supplies lots of overloaded operators and functions on these objects.
  - **field3d.cxx** implements the `Field3D` class, which varies in x, y and z. Since these handle a lot more memory than Field2D objects, the memory management is more complicated and includes reference counting. See section 3.5 for more details.
  - **field_data.cxx** Implements some functions in the `FieldData` class. This is a mainly pure virtual interface class which is inherited by `Field2D` and `Field3D`.
  - **fieldperp.cxx** implements a `FieldPerp` class to store slices perpendicular to the magnetic field i.e. they are a function of x and z only. This is mainly used for Laplacian inversion routines, and needs to be integrated with the other fields better.
  - **initialprofiles.cxx** routines to set the initial values of fields when a simulation first starts. Reads settings from the option file based on the name of the variable.
  - **vecops.cxx** a collection of function to operate on vectors. Contains things like `Grad`, `Div` and `Curl`, and uses a combination of field differential operators (in **difops.cxx**) and metric tensor components (in `Mesh`).

  - **vector2d.cxx** implements the `Vector2D` class, which uses a `Field2D` object for each of its 3 components. Overloads operators to supply things like dot and cross products.
  - **vector3d.cxx** implements `Vector3D` by using a `Field3D` object for each component.
  - **where.cxx** supplies functions for choosing between values based on selection criteria.

- fileio
  - **datafile.cxx** supplies an abstract `DataFile` interface for data input and output. Handles the conversion of data in fields and vectors into blocks of data which are then sent to a specific file format.
  - **formatfactory.cxx**
  - **formatfactory.hxx**
  - impls
    - **emptyformat.hxx**
    - netcdf
      - **nc_format.cxx** implements an interface to the NetCDF-4 library
      - **nc_format.hxx**
    - pdb
      - **pdb_format.cxx** implements an interace to Portable Data Binary (PDB) formatted files.
      - **pdb_format.hxx**
    - pnetcdf
      - **pnetcdf.cxx** Parallel NetCDF interface
      - **pnetcdf.hxx**

- invert
  - **fft_fftw.cxx** implements the `fft.hxx` interface by calling the Fastest Fourier Transform in the West (FFTW) library.
  - **full_gmres.cxx**
  - **inverter.cxx** is a `FieldPerp` inversion class currently under development. It is intended to provide a way to solve nonlinear problems using a GMRES iterative method.
  - **invert_gmres.cxx**

  - **invert_laplace_gmres.cxx** inherits the `Inverter` class and will solve more general Laplacian problems, using the `invert_laplace` routines as preconditioners.

- invert / laplace
  - **invert_laplace.cxx** uses Fourier decomposition in z combined with tri- and band-diagonal solvers in x to solve Laplacian problems.
  - **laplacefactory.hxx**
  - **laplacefactory.cxx**
  - impls
    - serial_tri
      - **serial_tri.hxx**
      - **serial_tri.cxx**
    - serial_band
      - **serial_band.hxx**
      - **serial_band.cxx**
    - spt
      - **spt.hxx**
      - **spt.cxx**
    - pdd
      - **pdd.hxx**
      - **pdd.cxx**

- invert / parderiv
  - **invert_parderiv.cxx** inverts a problem involving only parallel y derivatives. Intended for use in some preconditioners.
  - **parderiv_factory.hxx**
  - **parderiv_factory.cxx**
  - impls
    - serial
      - **serial.cxx**
      - **serial.hxx**
    - cyclic
      - **cyclic.cxx**
      - **cyclic.hxx**

- **lapack_routines.cxx** supplies an interface to the LAPACK linear solvers, which are used by the `invert_laplace` routines.
- mesh
  - **boundary_factory.cxx** creates boundary condition operators which can then be applied to fields. Described in section 7.4.
  - **boundary_region.cxx** implements a way to describe and iterate over boundary regions. Created by the mesh, and then used by boundary conditions. See section 7.1 for more details.
  - **boundary_standard.cxx** implements some standard boundary operations and modifiers such as `Neumann` and `Dirichlet`.
  - **difops.cxx** is a collection of differential operators on scalar fields. It uses the differential methods in **derivs.cxx** and the metric tensor components in `Mesh` to compute operators.
  - **grid.cxx** contains some routines which are used by the `Mesh` to read data.
  - **interpolation.cxx** contains functions for interpolating fields
  - **mesh.cxx** is the base class for the `Mesh` object. Contains routines useful for all `Mesh` implementations.
  - impls
    - **domain.cxx**
    - **domain.hxx**
    - **partition.cxx**
    - **partition.hxx**
    - bout
      - **boutmesh.cxx** implements a mesh interface which is compatible with BOUT grid files.
      - **boutmesh.hxx**
    - quilt
      - **quiltmesh.cxx** is an implementation of `Mesh` which is currently under development. It is intended to handle more general mesh shapes and topology than the currently used `BoutMesh` can handle.
      - **quiltmesh.hxx**
- physics
  - **gyro_average.cxx** gyro-averaging operators

  - **smoothing.cxx** provides smoothing routines on scalar fields
  - **sourcex.cxx** contains some useful routines for creating sources and sinks in physics equations.

- precon
  - **jstruc.cxx** is an experimental code for preconditioning using PETSc

- solver
  - **solver.cxx** is the interface for all solvers
  - **solverfactory.cxx** creates solver objects
  - **solverfactory.hxx**
  - impls
    - cvode
      - **cvode.cxx** is the implementation of `Solver` which interfaces with the SUNDIALS CVODE library.
      - **cvode.hxx**
    - ida
      - **ida.cxx** is the implementation which interfaces with the SUNDIALS IDA library
      - **ida.hxx**
    - petsc
      - **petsc.cxx** is the interface to the PETSc time integration routines
      - **petsc.hxx**
    - pvode
      - **pvode.cxx** interfaces with the 1998 (pre-SUNDIALS) version of PVODE (which became CVODE).
      - **pvode.hxx**

- sys
  - **boutcomm.cxx**
  - **boutexception.cxx** is an exception class which are used for error handling
  - **comm_group.cxx** provides routines for non-blocking collective MPI operations. These are not available in MPI-2, though are planned for MPI-3.

- **dcomplex.cxx** provides a `dcomplex` complex type. It is here because the original type `real` conflicted with the STL definition. This could probably be replaced with the STL implementation now.
- **derivs.cxx** contains basic derivative methods such as upwinding, central difference and WENO methods. These are then used by **difops.cxx**. Details are given in section 4.
- **diagnos.cxx** is the start of a diagnostics code which will extract values from fields each timestep and print to the output log file. Not yet finished.
- **msg_stack.cxx** is part of the error handling system. It maintains a stack of messages which can be pushed onto the stack at the start of a function, then removed (popped) at the end. If an error occurs or a segmentation fault is caught then this stack is printed out and can help to find errors.
- **options.cxx** provides an interface to the BOUT.inp option file and the command-line options.
- **optionsreader.cxx**
- **output.cxx**
- **range.cxx** Provides the RangeIterator class, used to iterate over a set of ranges. Described in section 12.3
- **stencils.cxx** contains methods to operate on stencils which are used by differential methods.
- **timer.cxx** a class for timing parts of the code like communications and file I/O. Described in section 12.2
- **utils.cxx** contains miscellaneous small useful routines such as allocating and freeing arrays.
- options
  - **optionparser.hxx**
  - **options_ini.cxx**
  - **options_ini.hxx**

3 Data types

The classes outlines in red in figure 2 are data types currently implemented in BOUT++.

3.1 FieldData

All BOUT++ data types implement a standard interface for accessing their data, which is then used in communication and file I/O code. This interface is in **src/field/field_data.hxx**. The mandatory (pure virtual) functions are:

```
bool isReal(); // Returns true if field consists of real values
bool is3D() const;   // True if variable is 3D

int byteSize() const; // Number of bytes for a single point
int realSize() const; // Number of reals (not implemented if not real)

int getData(int x, int y, int z, void *vptr) const; // Return number of bytes
int getData(int x, int y, int z, BoutReal *rptr) const; // Return number of reals

int setData(int x, int y, int z, void *vptr);
int setData(int x, int y, int z, BoutReal *rptr);
```

To support file I/O there are also some additional functions which may be implemented. A code can check if they are implemented by calling `ioSupport`. If one of them is implemented then they all should be.

```
bool  ioSupport();  // Return true if these functions are implemented
const string getSuffix(int component) const; // For vectors e.g. "_x"
void* getMark() const; // Store current settings (e.g. co/contra-variant)
void  setMark(void *setting); // Return to the stored settings
BoutReal* getData(int component);
void  zeroComponent(int component); // Set a component to zero
```

For twist-shift conditions, the optional function `shiftZ` is called in the communication routines.

```
void shiftZ(int jx, int jy, double zangle);
```

3.2 Field

The two main types are `Field2D`, and `Field3D`. Their main functions are to provide an easy way to manipulate data; they take care of all memory management, and most looping over grid-points in algebraic expressions. The 2D field implementation is relatively simple, but

more optimisations are used in the 3D field implementation because they are much larger (factor of ~ 100).

To handle time-derivatives, and enable expressions to be written in the following form:

```
ddt(Ni) = -b0xGrad_dot_Grad(phi, Ni);
```

fields (and vectors, see below) have a function:

```
Field3D* timeDeriv();
```

which returns a pointer to the field holding the time-derivative of this variable. This function ensures that this field is unique using a singleton pattern.

3.3 Vector

Vector classes build on the field classes, just using a field to represent each component.

To handle time-derivatives of vectors, some care is needed to ensure that the time-derivative of each vector component points to the same field as the corresponding component of the time-derivative of the vector:

```
ddt(v.x) = ddt(v).x
```

3.4 dcomplex

Several parts of the BOUT++ code involve FFTs and are therefore much easier to write using complex numbers. Unfortunately, the C++ complex library also tries to define a `real` type, which is already defined by PVODE. Several work-arounds were tried, some of which worked on some systems, but it was easier in the end to just implement a new class `dcomplex` to handle complex numbers.

3.5 Memory management

This code has been thoroughly tested/debugged, and should only be altered with great care, since just about every other part of BOUT++ depends on this code working correctly. Two optimisations used in the data objects to speed up code execution are memory recycling, which eliminates allocation and freeing of memory; and copy-on-change, which minimises unnecessary copying of data.

Both of these optimisations are done "behind the scenes", hidden from the remainder of the code, and are illustrated in figure 3: The objects (A,B,C) accessed by the user in operations discussed in the previous section act as an interface to underlying data (a,b). Memory recycling can be used because all the scalar fields are the same size (and vector fields are implemented as a set of 3 scalar fields). Each class implements a global stack of

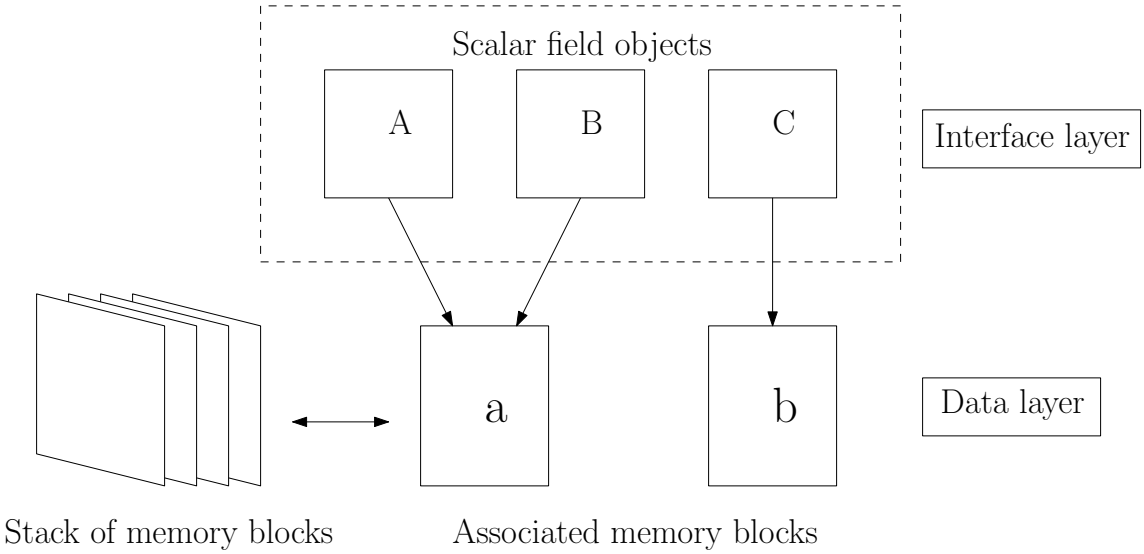

Figure 3: Memory handling in BOUT++. Memory allocation and freeing is eliminated by recycling memory blocks, and assignments without changes (`A = B`) do not result in copying data, only pointers to the data. Both these optimisations are handled internally, and are invisible to the programmer.

available memory blocks. When an object is assigned a value, it attempts to grab one of these memory blocks, and if none are available then a new block is allocated. When an object is destroyed, its memory block is not freed, but is put onto the stack. Since the evaluation of the time-derivatives involves the same set of operations each time, this system means that memory is only allocated the first time the time-derivatives are calculated, after which the same memory blocks are re-used. This eliminates the often slow system calls needed to allocate and free memory, replacing them with fast pointer manipulation.

Copy-on-change (reference counting) further reduces memory useage and unnecessary copying of data. When one field is set equal to another (e.g. `Field3D A = B` in figure 3), no data is copied, only the reference to the underlying data (in this case both A and B point to data block a). Only when one of these objects is modified is a second memory block used to store the different value. This is particularly useful when returning objects from a routine. Usually this would involve copying data from one object to another, and then destroying the original copy. Using reference counting this copying is eliminated.

NOTE: For debugging and Valgrind output, it can be useful to disable this memory block handling. To do this, add `-DDISABLE_FREELIST` to the compile flags

4 Derivatives

This is probably the part of the code most people will want to alter, and is in `bout++/src/sys/derivs.cxx`. The main task of this module is to map functions on fields like `DDX` to direction-independent differential methods on stencils such as 4^{th}-order central differencing. This mapping depends on global settings in **BOUT.inp** and is illustrated in figure 4.

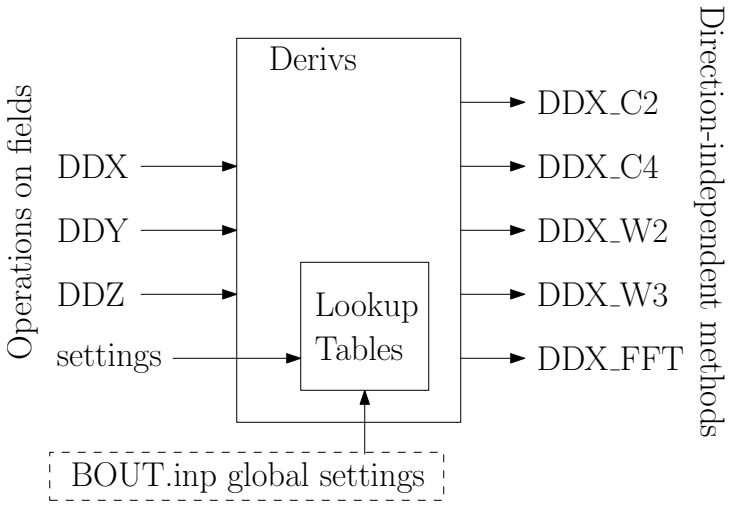

Figure 4: Overview of `derivs` module, mapping derivative functions on fields to direction-independent differential methods

Four kinds of differencing methods are supported

1. First derivative `DDX`, `DDY`, `DDZ`
   Central differencing type schemes for first-order derivatives

2. Second derivatives `D2DX2`, `D2DZ2`, `D2DZ2`
   Central differencing second derivatives e.g. for ∇^2

3. Upwinding `VDDX`, `VDDY`, `VDDZ`
   Terms like $\mathbf{v} \cdot \nabla$

4. Flux methods `FDDX`, `FDDY`, `FDDZ`
   Flux conserving, limiting methods for terms like $\nabla \cdot (\mathbf{v}f)$

The differencing methods themselves are independent on direction, and have types defined in `derivs.cxx`

```
1 typedef BoutReal (*deriv_func)(stencil &); // f
2 typedef BoutReal (*upwind_func)(stencil &, stencil &); // v, f
```

These operate on `stencil` objects. This class is in **stencils.hxx**

```
class stencil {
  public:
    int jx, jy, jz;  // Central location
    BoutReal c, p, m, pp, mm; // stencil 2 each side of the centre
    Overloaded operators
      =,+,-,*,/
    Functions
      min, max, abs
};
```

The main purpose of this class is to store a 5-element stencil. To simplify some code this class also has a bunch of overloaded operators on BoutReals and other stencil objects. There are also some functions to calculate things like absolute, minimum, and maximum values.

4.1 Lookup tables

To convert between short variable names ("C2"), long descriptions ("2nd order Central Differencing"), `DIFF_METHOD` enums used to specify methods at runtime (DIFF_C2, defined in `bout_types.hxx`), and function pointers (`DDX_C2`), taking into account whether variables are shifted or not, BOUT++ uses a set of lookup tables.

To find function pointers, tables of the following type are used:

```
/// Translate between DIFF_METHOD codes, and functions
struct DiffLookup {
  DIFF_METHOD method;
  deriv_func func;     // Single-argument differencing function
  upwind_func up_func; // Upwinding function
};
```

Because the `DiffLookup` type contains a `deriv_func` and `upwind_func` pointer, it is used for all function lookup tables. There is a separate table for each type of differencing method, so for example the table of non-staggered upwinding methods is

```
/// Upwinding functions lookup table
static DiffLookup UpwindTable[] = { {DIFF_U1, NULL, VDDX_U1},
            {DIFF_C2, NULL, VDDX_C2},
            {DIFF_U4, NULL, VDDX_U4},
            {DIFF_W3, NULL, VDDX_WENO3},
            {DIFF_C4, NULL, VDDX_C4},
            {DIFF_DEFAULT}};
```

The DIFF_DEFAULT at the end is used to terminate the array. These tables are used by functions

```
deriv_func lookupFunc(DiffLookup* table, DIFF_METHOD method);
upwind_func lookupUpwindFunc(DiffLookup* table, DIFF_METHOD method);
```

which return the function pointer corresponding to the given method. If the method isn't in the table, then the first entry in the table is used. These functions can be used at run-time to allow a user to specify the method to use for specific operators.

When reading settings from the input file, they are specified as short strings like "C2", and a longer description of the method chosen should be written to the output log. To do this, there is a name lookup table:

```
/// Translate between short names, long names and DIFF_METHOD codes
struct DiffNameLookup {
  DIFF_METHOD method;
  const char* label; // Short name
  const char* name;  // Long name
};

static DiffNameLookup DiffNameTable[] = {
  {DIFF_U1, "U1", "First order upwinding"},
  {DIFF_C2, "C2", "Second order central"},
  {DIFF_W2, "W2", "Second order WENO"},
  {DIFF_W3, "W3", "Third order WENO"},
  {DIFF_C4, "C4", "Fourth order central"},
  {DIFF_U4, "U4", "Fourth order upwinding"},
  {DIFF_FFT, "FFT", "FFT"},
  {DIFF_DEFAULT}}; // Use to terminate the list
```

To search this table, there is the function

```
DIFF_METHOD lookupFunc(DiffLookup *table, const string &label)
```

During initialisation, the lookup therefore works in two stages, shown in figure 5. First the short description is turned into a DIFF_METHOD enum code, then this code is turned into a function pointer.

4.2 Staggered grids

NOTE: This feature is currently very experimental, and doesn't appear to work as it should

By default, all quantities in BOUT++ are defined at cell centre, and all derivative methods map cell-centred quantities to cell centres. Switching on staggered grid support in

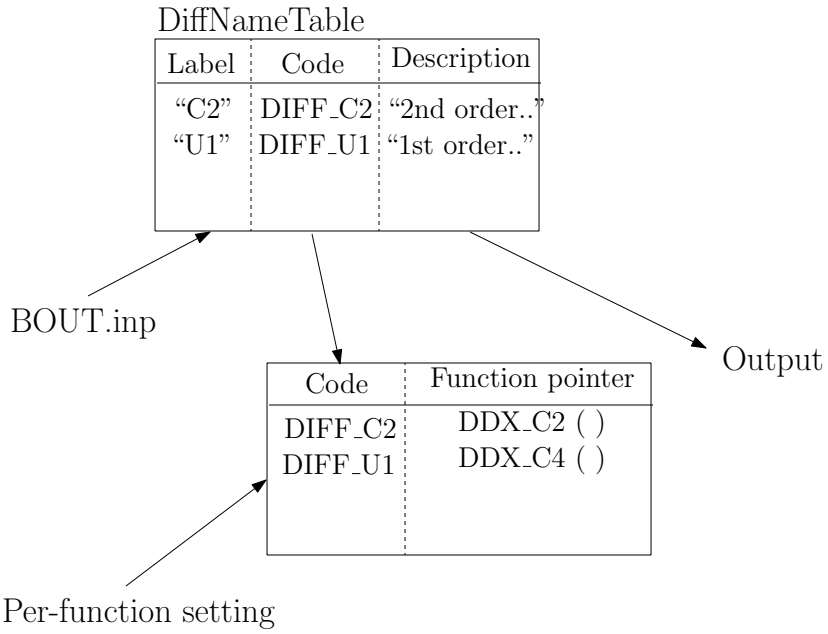

Figure 5: Lookup tables for mapping between differential method labels, codes, descriptions and function pointers

BOUT.inp:

```
StaggerGrids = true
```

allows quantities to be defined on cell boundaries. Functions such as `DDX` now have to handle all possible combinations of input and output locations, in addition to the possible derivative methods.

Several things are not currently implemented, which probably should be:

- Only 3D fields currently have a cell location attribute. The location (cell centre etc) of 2D fields is ignored at the moment. The rationale for this is that 2D fields are assumed to be slowly-varying equilibrium quantities for which it won't matter so much. Still, needs to be improved in future
- Twist-shift and X shifting still treat all quantities as cell-centred.
- No boundary condition functions yet account for cell location.

Currently, BOUT++ does not support values at cell corners; values can only be defined at cell centre, or at the lower X,Y, or Z boundaries. This is

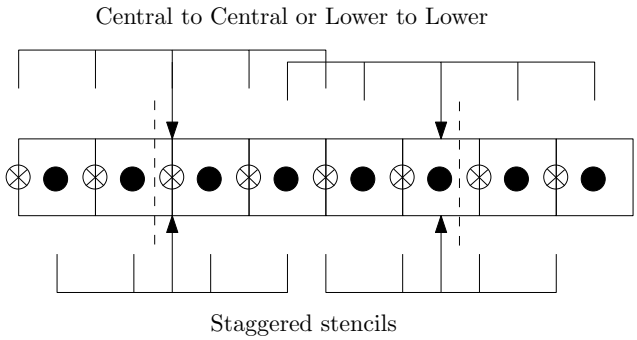

Figure 6: Stencils with cell-centred (solid) and lower shifted values (open). Processor boundaries marked by vertical dashed line

Once staggered grids are enabled, two types of stencil are needed: those which map between the same cell location (e.g. cell-centred values to cell-centred values), and those which map to different locations (e.g. cell-centred to lower X).

Central differencing using 4-point stencil:

$$\begin{aligned} y &= \left(9y_{-1/2} + 9y_{1/2} - y_{-3/2} - y_{3/2}\right)/16 \\ \frac{\partial y}{\partial x} &= \left(27y_{1/2} - 27y_{-1/2} - y_{3/2} + y_{-3/2}\right)/24\Delta x \\ \frac{\partial^2 y}{\partial x^2} &= \left(y_{3/2} + y_{-3/2} - y_{1/2} - y_{-1/2}\right)/2\Delta x^2 \end{aligned}$$

NOTE: What should the default cell location of a derivative be? Currently the default is to remain the same as without staggered grids. Setting `StaggerGrids = true` by itself has no effect - derivative output locations have to be explicitly set.

Table 1: DDX actions depending on input and output locations. Uses first match.

Input	Output	Actions
Same locations		Central stencil
CENTRE	XLOW	Lower staggered stencil
XLOW	CENTRE	Upper staggered stencil
XLOW	Any	Staggered stencil to CENTRE, then interpolate
CENTRE	Any	Central stencil, then interpolate
Any	Any	Interpolate to centre, use central stencil, then interpolate

5 Laplacian inversion

NOTE: This part of the code needs some algorithm improvement (better parallel tridiagonal solver).

The Laplacian inversion code solves the equation:

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

where x and b are 3D variables, whilst a, c and d are 2D variables. Several different algorithms are implemented for Laplacian inversion, and they differ between serial and parallel versions. Serial inversion can currently either be done using a tridiagonal solver (Thomas algorithm), or a band-solver (allowing 4^{th}-order differencing).

To support multiple implementations, a base class `Laplacian` is defined in **include/invert_laplace.hxx**. This defines a set of functions which all implementations must provide:

```
class Laplacian {
 public:
  virtual void setCoefA(const Field2D &val) = 0;
  virtual void setCoefC(const Field2D &val) = 0;
  virtual void setCoefD(const Field2D &val) = 0;

  virtual const FieldPerp solve(const FieldPerp &b) = 0;
}
```

At minimum, all implementations must provide a way to set coefficients, and a solve function which operates on a single FieldPerp (X-Y) object at once. Several other functions are also virtual, so default code exists but can be overridden by an implementation.

For convenience, the `Laplacian` base class also defines a function to calculate coefficients in a Tridiagonal matrix

```
  void tridagCoefs(int jx, int jy, int jz, dcomplex &a, dcomplex &b, ↩
     dcomplex &c, const Field2D *ccoef = NULL, const Field2D *d=NULL)↩
     ;
```

For the user of the class, some static functions are defined:

```
  static Laplacian* create(Options *opt = NULL);
  static Laplacian* defaultInstance();
```

The create function allows new Laplacian implementations to be created, based on options. To use the options in the `[laplace]` section, just use the default:

```
  Laplacian* lap = Laplacian::create();
```

The code for the `Laplacian` base class is in **src/invert/laplace/invert_laplace.cxx**. The actual creation of new Laplacian implementations is done in the `LaplaceFactory` class, defined in **src/invert/laplace/laplacefactory.cxx**. This file includes all the headers for the implementations, and chooses which one to create based on the "type" setting in the input options. This factory therefore provides a single point of access to the underlying Laplacian inversion implementations.

Each of the implementations is in a subdirectory of **src/invert/laplace/impls** and is discussed below.

5.1 Serial tridiagonal solver

This is the simplest implementation, and is in **src/invert/laplace/impls/serial_tri/**

5.2 Serial band solver

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

5.3 SPT parallel tridiagonal

This is a reference code which performs the same operations as the serial code. To invert a single XZ slice (`FieldPerp` object), data must pass from the innermost processor (`mesh->PE_XIND` = 0) to the outermost `mesh->PE_XIND` = `mesh->NXPE`−1 and back again.

Some parallelism is achieved by running several inversions simultaneously, so while processor 1 is inverting Y=0, processor 0 is starting on Y=1. This works ok as long as the number of slices to be inverted is greater than the number of X processors (`MYSUB` > `mesh->NXPE`). If `MYSUB` < `mesh->NXPE` then not all processors can be busy at once, and so efficiency will fall sharply. Figure 7 shows the useage of 4 processors inverting a set of 3 poloidal slices (i.e. MYSUB=3)

5.4 PDD algorithm

This is the Parallel Diagonally Dominant (PDD) algorithm. It's very fast, but achieves this by neglecting some cross-processor terms. For ELM simulations, it has been found that these terms are important, so this method is not usually used.

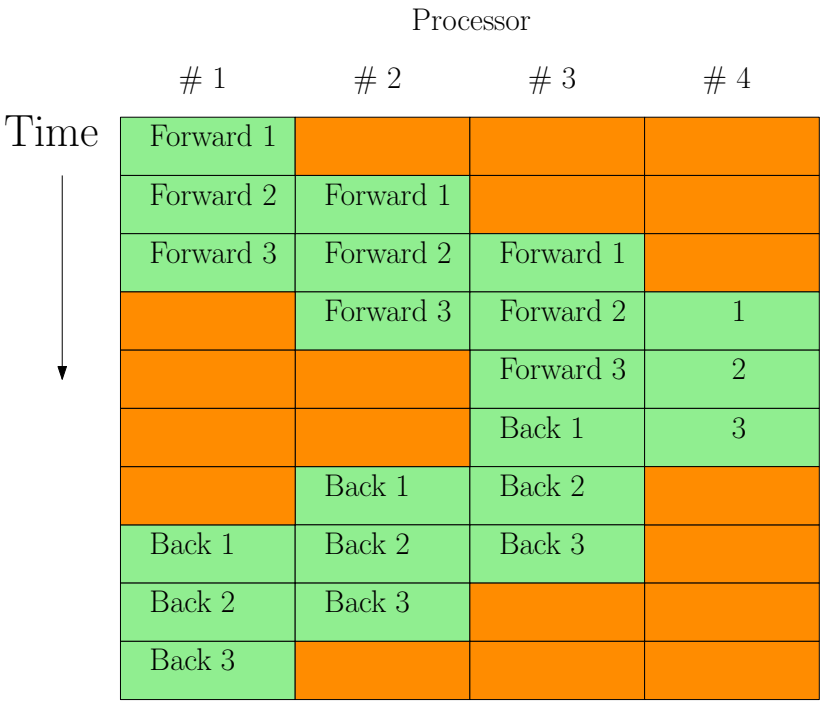

Figure 7: Parallel Laplacian inversion with MYSUB=3 on 4 processors. Red periods are where a processor is idle - in this case about 40% of the time

6 Mesh

The mesh is used in pretty much all parts of the code, and deals with things like the geometry of the mesh (metric tensors etc.), and how the mesh is divided between processors (communications). The Mesh class (**include/mesh.hxx**) defines an interface, and there are currently two implementations:

- `BoutMesh` (**src/mesh/boutmesh.cxx**) which is backwards compatible with the BOUT and BOUT-06 codes. This is a logically rectangular mesh so the number of radial points (x) can't change in the poloidal direction (y).
- `QuiltMesh` (**src/mesh/quiltmesh.cxx**) is a more general mesh under development (i.e. **not** recommended except for testing). The primary advantage is that it allows the number of points in x to vary between regions so the number of radial grid points in the core no longer needs to be the same as the number in the private flux regions.

6.1 Grid data sources

All data sources inherit from `GridDataSource`, defined in **grid.hxx** at line 43. They must supply a method to test if a variable exists:

```
47 bool GridDataSource::hasVar(const char *name);
```

a method to get the size of the variable

```
49 vector<int> GridDataSource::getSize(const char *name);
```

To fetch data, first the (x,y,z) origin must be set:

```
52 bool GridDataSource::setOrigin(int x = 0, int y = 0, int z = 0);
```

and then use methods to fetch integers or reals:

```
52 bool GridDataSource::fetch(int *var, const string &name, int lx = 1, ↩
      int ly = 0, int lz = 0);
53 bool GridDataSource::fetch(BoutReal *var, const string &name, int lx ↩
      = 1, int ly = 0, int lz = 0);
```

In addition, GridDataSource implementations can have methods which should be called before and after variables are accessed:

```
52 void GridDataSource::open(const char *name = NULL);
53 void GridDataSource::close();
```

6.2 Loading a mesh

To load in a mesh from a file or other source, there are the commands:

```
int addSource(GridDataSource);    // Add a data source
int load();                       // Load from added data sources
int load(GridDataSource);         // Load from specified data source
```

all of which return an error code (0 if successful). `addSource` is used to add a set of input data sources which inherit from `GridDataSource`. `load`() loads the mesh from these sources, querying each data source in turn for the required variables (in the order in which they were added). `load`(`GridDataSource`) loads the mesh from only the supplied data source.

In **bout++.cxx**, this is used to initialise the mesh:

```
mesh->addSource(new GridFile(data_format(grid_name), grid_name));
if(mesh->load()) {
  output << "Failed to read grid. Aborting\n";
  return 1;
}
```

which creates a `GridFile` object based on the data format of the grid file name, then adds that as a source of data for Mesh.

For post-processing of the results, it's useful to have mesh quantities in the dump files along with the results. To do this, there's the function

```
void outputVars(Datafile &file); // Add mesh vars to file
```

which is called during BOUT++ initialisation.

6.2.1 Implementation: BoutMesh

BoutMesh class uses the BOUT indices (which trace back to UEDGE):

```
int ixseps1, ixseps2, jyseps1_1, jyseps2_1, jyseps1_2, jyseps2_2;
```

`ixseps1` and `ixseps2` give the X location of the separatrices, and are equal in the case of single-null configurations. The indexing is such that all points `0 <= x < ixseps1` are inside the separatrix, whilst `ixseps1 <= x < ngx` are outside.

6.2.2 Implementation: QuiltMesh

6.3 Index ranges

The Mesh class includes several public members which describe the size of the mesh, and are used all over BOUT++ to loop over variables:

```
/// Size of the mesh on this processor including guard/boundary cells
int ngx, ngy, ngz;
/// Local ranges of data (inclusive), excluding guard cells
int xstart, xend, ystart, yend;
```

6.4 Getting data

The `load()` code above needs to read data for the mesh, and physics codes usually need to read their initial profiles during initialisation. To do this, Mesh provides an overloaded function `get`:

```
int get(var, const char *name); // Request data from mesh file
```

where `var` can be just about any BOUT++ datatype (`Field2D`, `Vector3D` etc.).

6.4.1 Implementation: BoutMesh

For integers and BoutReals, the implementation is fairly trivial. Uses the Mesh protected functions to find a data source and read data from it.

```
GridDataSource* s = findSource(name);  // Find a source of data
s->open(name);                          // Open the source
bool success = s->fetch(&ival, name);   // Get the data
s->close();                             // Close the source
```

To read 2D and 3D fields, the branch-cuts need to be taken into account.

6.5 Communications

The most common type of communication is to just exchange all guard cells with neighboring processors. Mesh provides the following commands for doing this:

```
int communicate(FieldData, ...); // Communicate one or more fields
int communicate(FieldGroup);     // Communicate a group of fields
int communicate(FieldData);      // Returns error code
comm_handle send(FieldGroup);    // Send data
int wait(comm_handle);           // Receive data
```

`communicate(FieldData)` can (currently) be used to communicate up to 4 variables together, and makes the code quite clear. For example in **examples/DriftInstability/2fluid.cxx** around line 360:

```
// Need to communicate jpar
mesh->communicate(jpar);
```

Since this uses the `FieldData` interface like Datafile, this can be used to communicate all BOUT++ field data types. The limit of 4 is because the C-style `varargs` system doesn't work with "non POD" variables, i.e. classes. To communicate a larger number of variables, create a `FieldGroup` object to group fields together, then communicate them all together:

```
FieldGroup comgrp;  // Group of variables for communication
Field3D P;
Vector3D V;

comgrp.add(P); // Add the variables
comgrp.add(V); // Usually done in physics_init

mesh->communicate(comgrp); // Communicate in physics_run
```

If you want to overlap communications with calculations then use the `send` and `wait` functions instead of `communicate`.

```
comm_handle ch = mesh->send(comgrp); // Start the communications
// Calculations which don't need variables in comgrp
wait(ch); // Wait for all communications to finish
```

6.5.1 Implementation: BoutMesh

In BoutMesh, the communication is controlled by the variables

```
int UDATA_INDEST, UDATA_OUTDEST, UDATA_XSPLIT;
int DDATA_INDEST, DDATA_OUTDEST, DDATA_XSPLIT;
int IDATA_DEST, ODATA_DEST;
```

In the Y direction, each boundary region (**U**p and **D**own in Y) can be split into two, with `0 <= x < UDATA_XSPLIT` going to the processor index `UDATA_INDEST`, and `UDATA_INDEST <= x < ngx` going to `UDATA_OUTDEST`. Similarly for the Down boundary. Since there are no branch-cuts in the X direction, there is just one destination for the **I**nner and **O**uter boundaries. In all cases a negative processor number means that there's a domain boundary.

6.6 X communications

For parallel Laplacian inversions, communication is needed in the X direction only, and involves quantities which are not in Fields.

```
bool firstX();  // True if at the inner X boundary
bool lastX();   // True if at the outer X boundary
int NXPE, PE_XIND; // Number of processors in X, and X processor ↩
   index
```

```
4 int sendXOut(BoutReal *buffer, int size, int tag);
5 sendXIn(BoutReal *buffer, int size, int tag);
6 comm_handle irecvXOut(BoutReal *buffer, int size, int tag);
7 comm_handle irecvXIn(BoutReal *buffer, int size, int tag);
```

The variables `NXPE` and `PE_XIND` shouldn't really be there, but are currently needed because the SPT algorithm in **invert_laplace.cxx** needs to know when it's going to be next and so keep track of which processor number is currently working. This logic to pass a problem along a chain in X should really be moved into Mesh.

6.7 Y-Z surface communications

Some operations (like parallel inversions in `bout++/src/invert/invert_parderiv.cxx`) need to be performed on Y-Z surfaces, i.e. slices at constant X. This needs to be able to handle open and closed surfaces, and that closed surfaces may need a shift in the Z direction to match one end onto the other (a twist-shift condition).

The simplest operation is to average a quantity over Y:

```
1 const Field2D averageY(const Field2D &f); // Average in Y
```

Currently this is only implemented for 2D fields. More generally a set of FieldData objects could be used.

To test if a particular surface is closed, there is the function

```
1 bool surfaceClosed(int jx, BoutReal &ts); // Test if a surface is ↩
    closed, and if so get the twist-shift angle
```

The most general way to access data on surfaces is to use an iterator, which can be created using:

```
1 SurfaceIter* iterateSurfaces();
```

This then allows looping over the surfaces in the usual way

```
1 for(surf->first(); !surf->isDone(); surf->next()) {
2   ...
3 }
```

NB: This iterator splits the surfaces between processors, so each individual processor will iterate over a different set of surfaces. This is to allow automatic load balancing when gathering and scattering data from an entire surface onto one processor using:

```
1 surf->gather(FieldData, BoutReal *recvbuffer);
2 surf->scatter(BoutReal *sendbuffer, Field result);
```

The buffer is assumed to be large enough to hold all the data. To get the number of points in Y for this surface, use

```
int ysize = surf->ysize();
```

To test if the surface is closed, there's the test

```
bool surf->closed(BoutReal &ts)
```

which returns true if the surface is closed, along with the twist-shift angle.

6.8 Boundary regions

The boundary condition code (see section 7) needs ways to loop over the boundary regions, without needing to know the details of the mesh.

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

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

bool hasBndryLowerY();
bool hasBndryUpperY();

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

The `RangeIterator` class is an iterator which allows looping over a set of indices. Details are given in section 12.3. For example, in **src/solver/solver.cxx** to loop over the upper Y boundary of a 2D variable `var`:

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

The `BoundaryRegion` class is defined in **include/boundary_region.hxx**

6.9 Initial profiles

The initial profiles code needs to construct a solution which is smooth everywhere, with a form of perturbation specified in the input file for each direction. In order to do this, it needs a continuous function to use as an index. This is supplied by the functions:

```
BoutReal GlobalX(int jx); // Continuous X index between 0 and 1
BoutReal GlobalY(int jy); // Continuous Y index (0 -> 1)
```

which take a local x or y index and return a globally continuous x or y index.

6.10 Differencing

The mesh spacing is given by the public members

```
// These used for differential operators
Field2D dx, dy;
Field2D d2x, d2y;    // 2nd-order correction for non-uniform meshes
BoutReal zlength, dz;    // Derived from options (in radians)
```

6.11 Metrics

The contravariant and covariant metric tensor components are public members of `Mesh`:

```
// Contravariant metric tensor (g^{ij})
Field2D g11, g22, g33, g12, g13, g23; // These are read in grid.cxx

// Covariant metric tensor
Field2D g_11, g_22, g_33, g_12, g_13, g_23;

int calcCovariant();     // Invert contravatiant metric to get ↩
   covariant
int calcContravariant(); // Invert covariant metric to get ↩
   contravariant
```

If only one of these sets is modified by an external code, then `calc_covariant` and `calc_contravariant` can be used to calculate the other (uses Gauss-Jordan currently).

From the metric tensor components, Mesh calculates several other useful quantities:

```
int jacobian(); // Calculate J and Bxy
Field2D J; // Jacobian
Field2D Bxy; // Magnitude of B = nabla z times nabla x

/// Calculate differential geometry quantities from the metric tensor
int geometry();

// Christoffel symbol of the second kind (connection coefficients)
Field2D G1_11, G1_22, G1_33, G1_12, G1_13;
Field2D G2_11, G2_22, G2_33, G2_12, G2_23;
Field2D G3_11, G3_22, G3_33, G3_13, G3_23;

Field2D G1, G2, G3;
```

These quantities are public and accessible everywhere, but this is because they are needed in a lot of the code. They shouldn't change after initialisation, unless the physics model

starts doing fancy things with deforming meshes.

6.12 Miscellaneous

There are some public members of Mesh which are there for some specific task and don't really go anywhere else (yet).

To perform radial derivatives in tokamak geometry, interpolation is needed in the Z direction. This is done by shifting in Z by a phase factor, performing the derivatives, then shifting back. The following public variables are currently used for this:

```
bool ShiftXderivs; // Use shifted X derivatives
int  ShiftOrder;   // Order of shifted X derivative interpolation
Field2D zShift;    // Z shift for each point (radians)

Field2D ShiftTorsion; // d <pitch angle> / dx. Needed for vector ↩
   differentials (Curl)
Field2D IntShiftTorsion; // Integrated shear (I in BOUT notation)
bool IncIntShear; // Include integrated shear (if shifting X)
```

```
int  TwistOrder;   // Order of twist-shift interpolation
```

This determines what order method to use for the interpolation at the twist-shift location, with `0` meaning FFT during communication. Since this must be 0 at the moment it's fairly redundant and should be removed.

A (currently experimental) feature is

```
bool StaggerGrids;    ///< Enable staggered grids (Centre, Lower). ↩
   Otherwise all vars are cell centred (default).
```

7 Boundary conditions

The boundary condition system needs to be very flexible in order to handle:

- Meshes which can divide up the boundary into an arbitrary number of regions, giving each one a label. For example in BoutMesh (specific to tokamaks), the boundary regions are labelled "core", "sol", "pf" and "target".
- Each variable can have a different boundary condition in each region. It should be possible to have a global setting "all variables have dirichlet conditions on all boundaries", which is over-ridden by more specific settings such as "All variables have neumann conditions on the inner x boundaries", and finally to "variable 'Ni' has laplacian boundary conditions in the 'sol' regions"

- Boundary conditions can be modified to be "relaxing". This means that rather than enforcing a strict boundary condition, it's a mixture of zero-gradient in the time-derivative combined with a damping (relaxation) towards the desired boundary condition. This can help improve the numerics of turbulence simulations.
- Users should be able to implement their own boundary conditions, and add them to the system at run-time without modifying the core code.
- After `physics_init`, a boundary condition must be applied to the variables. During a simulation (at the end of `physics_run`), boundary conditions need to be applied to the time-derivatives. The boundary system should ensure that these conditions are consistent.

7.1 Boundary regions

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

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

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

```
12 /// Describes a region of the boundary, and a means of iterating over↩
       it
13 class BoundaryRegion {
14   public:
15   BoundaryRegion();
16   BoundaryRegion(const string &name, int xd, int yd);
17   virtual ~BoundaryRegion();
18
19   string label; // Label for this boundary region
20
21   BndryLoc location; // Which side of the domain is it on?
22
23   int x,y; // Indices of the point in the boundary
24   int bx, by; // Direction of the boundary [x+dx][y+dy] is going ↩
       outwards
25
26   virtual void first() = 0;
```

```
27   virtual void next() = 0; // Loop over every element from inside out↩
         (in X or Y first)
28   virtual void nextX() = 0; // Just loop over X
29   virtual void nextY() = 0; // Just loop over Y
30   virtual bool isDone() = 0; // Returns true if outside domain. Can ↩
       use this with nested nextX, nextY
31 };
```

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

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

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

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

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

7.2 Boundary operations

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

```
21 /// An operation on a boundary
22 class BoundaryOp {
23  public:
24   BoundaryOp() {bndry = NULL;}
25   BoundaryOp(BoundaryRegion *region)
26
27   // Note: All methods must implement clone, except for modifiers (↩
       see below)
28   virtual BoundaryOp* clone(BoundaryRegion *region, const list<string↩
       > &args);
```

```
29
30   /// Apply a boundary condition on field f
31   virtual void apply(Field2D &f) = 0;
32   virtual void apply(Field3D &f) = 0;
33
34   virtual void apply(Vector2D &f);
35
36   virtual void apply(Vector3D &f);
37
38   /// Apply a boundary condition on ddt(f)
39   virtual void apply_ddt(Field2D &f);
40   virtual void apply_ddt(Field3D &f);
41   virtual void apply_ddt(Vector2D &f);
42   virtual void apply_ddt(Vector3D &f);
43
44   BoundaryRegion *bndry;
45 };
```

(where the implementations have been removed for clarity). Which has a pointer to a `BoundaryRegion` object specifying which region this boundary is operating on.

Boundary conditions need to be imposed on the initial conditions (after `physics_init()`), and on the time-derivatives (after `physics_run()`). The `apply()` functions are therefore called during initialisation and given the evolving variables, whilst the `apply_ddt` functions are passed the time-derivatives.

To implement a boundary operation, as a minimum the `apply(Field2D)`, `apply(Field2D)` and `clone()` need to be implemented: By default the `apply(Vector)` will call the `apply(Field)` functions on each component individually, and the `apply_ddt()` functions just call the `apply()` functions.

Example: Neumann boundary conditions are defined in **boundary_standard.hxx**:

```
22 /// Neumann (zero-gradient) boundary condition
23 class BoundaryNeumann : public BoundaryOp {
24  public:
25   BoundaryNeumann() {}
26  BoundaryNeumann(BoundaryRegion *region):BoundaryOp(region) { }
27   BoundaryOp* clone(BoundaryRegion *region, const list<string> &args)↩
       ;
28   void apply(Field2D &f);
29   void apply(Field3D &f);
30 };
```

and implemented in **boundary_standard.cxx**

```
52 void BoundaryNeumann::apply(Field2D &f) {
53   // Loop over all elements and set equal to the next point in
54   for(bndry->first(); !bndry->isDone(); bndry->next())
55     f[bndry->x][bndry->y] = f[bndry->x - bndry->bx][bndry->y - bndry->by];
56 }
57
58 void BoundaryNeumann::apply(Field3D &f) {
59   for(bndry->first(); !bndry->isDone(); bndry->next())
60     for(int z=0;z<mesh->ngz;z++)
61       f[bndry->x][bndry->y][z] = f[bndry->x - bndry->bx][bndry->y - bndry->by][z];
62 }
```

This is all that's needed in this case since there's no difference between applying Neumann conditions to a variable and to its time-derivative, and Neumann conditions for vectors are just Neumann conditions on each vector component.

To create a boundary condition, we need to give it a boundary region to operate over:

```
BoundaryRegion *bndry = ...
BoundaryOp op = new BoundaryOp(bndry);
```

The `clone` function is used to create boundary operations given a single object as a template in `BoundaryFactory`. This can take additional arguments as a vector of strings - see explanation in section 7.4.

7.3 Boundary modifiers

To create more complicated boundary conditions from simple ones (such as Neumann conditions above), boundary operations can be modified by wrapping them up in a `BoundaryModifier` object, defined in **boundary_op.hxx**:

```
63 class BoundaryModifier : public BoundaryOp {
64  public:
65   virtual BoundaryOp* clone(BoundaryOp *op, const list<string> &args) = 0;
66  protected:
67   BoundaryOp *op;
68 };
```

Since `BoundaryModifier` inherits from `BoundaryOp`, modified boundary operations are just a different boundary operation and can be treated the same (Decorator pattern). Boundary modifiers could also be nested inside each other to create even more complicated boundary

operations. Note that the `clone` function is different to the `BoundaryOp` one: instead of a `BoundaryRegion` to operate on, modifiers are passed a `BoundaryOp` to modify.

Currently the only modifier is `BoundaryRelax`, defined in **boundary_standard.hxx**:

```
64 /// Convert a boundary condition to a relaxing one
65 class BoundaryRelax : public BoundaryModifier {
66  public:
67   BoundaryRelax(BoutReal rate) {r = fabs(rate);}
68   BoundaryOp* clone(BoundaryOp *op, const list<string> &args);
69
70   void apply(Field2D &f);
71   void apply(Field3D &f);
72
73   void apply_ddt(Field2D &f);
74   void apply_ddt(Field3D &f);
75  private:
76   BoundaryRelax() {} // Must be initialised with a rate
77   BoutReal r;
78 };
```

7.4 Boundary factory

The boundary factory creates new boundary operations from input strings, for example turning "relax(dirichlet)" into a relaxing Dirichlet boundary operation on a given region. It is defined in **boundary_factory.hxx** as a Singleton, so to get a pointer to the boundary factory use

```
  BoundaryFactory *bfact = BoundaryFactory::getInstance();
```

and to delete this singleton, free memory and cleanup at the end use:

```
  BoundaryFactory::cleanup();
```

Because users should be able to add new boundary conditions during `physics_init()`, boundary conditions are not hard-wired into `BoundaryFactory`. Instead, boundary conditions must be registered with the factory, passing an instance which can later be cloned. This is done in **bout++.cxx** for the standard boundary conditions:

```
258   BoundaryFactory* bndry = BoundaryFactory::getInstance();
259   bndry->add(new BoundaryDirichlet(), "dirichlet");
260   ...
261   bndry->addMod(new BoundaryRelax(10.), "relax");
```

where the `add` function adds BoundaryOp objects, whereas `addMod` adds `BoundaryModifier` objects. **Note**: The objects passed to `BoundaryFactory` will be deleted when `cleanup()` is called.

When a boundary operation is added, it is given a name such as "dirichlet", and similarly for the modifiers ("relax" above). These labels and object pointers are stored internally in `BoundaryFactory` in maps defined in **boundary_factory.hxx**:

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

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

```
24  /// Create a boundary operation object
25  BoundaryOp* create(const string &name, BoundaryRegion *region);
26  BoundaryOp* create(const char* name, BoundaryRegion *region);
```

to turn a string such as "relax(dirichlet)" and a `BoundaryRegion` pointer into a `BoundaryOp` object. These functions are implemented in **boundary_factory.cxx**, starting around line 42. The parsing is done recursively by matching the input string to one of:

- `modifier(<expression>, arg1, ...)`
- `modifier(<expression>)`
- `operation(arg1, ...)`
- `operation`

the `<expression>` variable is then resolved into a BoundaryOp object by calling `create(<expression, region)`.

When an operator or modifier is found, it is created from the pointer stored in the `opmap` or `modmap` maps using the `clone` method, passing a `list<string>` reference containing any arguments. It's up to the operation implementation to ensure that the correct number of arguments are passed, and to parse them into floats or other types.

Example: The Dirichlet boundary condition can take an optional argument to change the value the boundary's set to. In **boundary_standard.cxx**:

```
13 BoundaryOp* BoundaryDirichlet::clone(BoundaryRegion *region, const ↩
     list<string> &args) {
14   if(!args.empty()) {
15     // First argument should be a value
16     stringstream ss;
17     ss << args.front();
18
```

```
19      BoutReal val;
20      ss >> val;
21      return new BoundaryDirichlet(region, val);
22    }
23    return new BoundaryDirichlet(region);
24  }
```

If no arguments are passed i.e. the string was "dirichlet" or "dirichlet()" then the `args` list is empty, and the default value (0.0) is used. If one or more arguments is used then the first argument is parsed into a `BoutReal` type and used to create a new `BoundaryDirichlet` object. If more arguments are passed then these are just ignored; probably a warning should be printed.

To set boundary conditions on a field, `FieldData` methods are defined in **field_data.hxx**:

```
1  // Boundary conditions
2    void setBoundary(const string &name); ///< Set the boundary ↩
       conditions
3    void setBoundary(const string &region, BoundaryOp *op); ///< ↩
       Manually set
4    virtual void applyBoundary() {}
5    virtual void applyTDerivBoundary() {};
6   protected:
7    vector<BoundaryOp*> bndry_op; // Boundary conditions
```

The `setBoundary(const string &name)` method is implemented in **field_data.cxx**. It first gets a vector of pointers to `BoundaryRegion`s from the mesh, then loops over these calling `BoundaryFactory::createFromOptions` for each one and adding the resulting boundary operator to the `bndry_op` vector.

8 Variable initialisation

8.1 `FieldFactory` class

This class provides a way to generate a field with a specified form. It implements a recursive descent parser to turn a string containing something like `"gauss(x-0.5,0.2)*↩ gauss(y)*sin(3*z)"` into values in a `Field3D` or `Field2D` object. Examples are given in the `test-fieldfactory` example:

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

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

```
49 class FieldGenerator {
50  public:
51   virtual ~FieldGenerator() { }
52   virtual FieldGenerator* clone(const list<FieldGenerator*> args) {↩
        return NULL;}
53   virtual BoutReal generate(int x, int y, int z) = 0;
54 };
```

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

The simplest generator is a fixed numerical value, which is represented by a `FieldValue` object:

```
59 class FieldValue : public FieldGenerator {
60  public:
61   FieldValue(BoutReal val) : value(val) {}
62   BoutReal generate(int x, int y, int z) { return value; }
63  private:
64   BoutReal value;
65 };
```

8.2 Adding a new function

To add a new function to the FieldFactory, a new `FieldGenerator` class must be defined. Here we will use the example of the `sinh` function, implemented using a class `FieldSinh`. This takes a single argument as input, but `FieldPI` takes no arguments, and `FieldGaussian` takes either one or two. Study these after reading this to see how these are handled.

First, edit **include/field_factory.hxx** and add a class definition:

```
122 class FieldSinh : public FieldGenerator {
123  public:
124   FieldSinh(FieldGenerator* g) : gen(g) {}
125   ~FieldSinh() {if(gen) delete gen;}
126
127   FieldGenerator* clone(const list<FieldGenerator*> args);
128   BoutReal generate(int x, int y, int z);
129  private:
130   FieldGenerator *gen;
```

```
131 };
```

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

Next edit **src/field/field_factory.cxx** and add the implementation of the `clone` and `generate` functions:

```
100 FieldGenerator* FieldSinh::clone(const list<FieldGenerator*> args) {
101   if(args.size() != 1) {
102     output << "FieldFactory error: Incorrect number of arguments to ↩
          sinh function. Expecting 1, got " << args.size() << endl;
103     return NULL;
104   }
105
106   return new FieldSinh(args.front());
107 }
108
109 BoutReal FieldSinh::generate(int x, int y, int z) {
110   return sinh(gen->generate(x,y,z));
111 }
```

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

The `generate` function for `sinh` just gets the value of the input by calling `gen->generate(x,y,z)`, calculates `sinh` of it and returns the result.

The `clone` function means that the parsing code can make copies of any `FieldGenerator` class if it's given a single instance to start with. The final step is therefore to give the `FieldFactory` class an instance of this new generator. Edit the `FieldFactory` constructor `FieldFactory::FieldFactory()` in **src/field/field_factory.cxx** and add the line:

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

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

8.3 Parser internals

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

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

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

```
259 char FieldFactory::nextToken() {
260  ...
```

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

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

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

```
484 FieldGenerator* FieldFactory::parse(const string &input) {
485    ...
```

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

```
477 FieldGenerator* FieldFactory::parseExpression() {
478    ...
```

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

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

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

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

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

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

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

9 Solver

The solver is the interface between BOUT++ and the time-integration code such as SUNDIALS. All solvers implement the `Solver` class interface (see **src/solver/generic_solver.hxx**).

First all the fields which are to be evolved need to be added to the solver. These are always done in pairs, the first specifying the field, and the second the time-derivative:

```
1 void add(Field2D &v, Field2D &F_v, const char* name);
```

This is normally called in the `physics_init` initialisation routine. Some solvers (e.g. IDA) can support constraints, which need to be added in the same way as evolving fields:

```
1 bool constraints();
2 void constraint(Field2D &v, Field2D &C_v, const char* name);
```

The `constraints()` function tests whether or not the current solver supports constraints. The format of `constraint(...)` is the same as `add`, except that now the solver will attempt

to make `C_v` zero. If `constraint` is called when the solver doesn't support them then an error should occur.

If the physics model implements a preconditioner or Jacobian-vector multiplication routine, these can be passed to the solver during initialisation:

```
typedef int (*PhysicsPrecon)(BoutReal t, BoutReal gamma, BoutReal ↩
   delta);
void setPrecon(PhysicsPrecon f); // Specify a preconditioner
typedef int (*Jacobian)(BoutReal t);
void setJacobian(Jacobian j); // Specify a Jacobian
```

If the solver doesn't support these functions then the calls will just be ignored.

Once the problem to be solved has been specified, the solver can be initialised using:

```
int init(rhsfunc f, int argc, char **argv, bool restarting, int nout,↩
    BoutReal tstep);
```

which returns an error code (0 on success). This is currently called in **bout++.cxx**:

```
if(solver.init(physics_run, argc, argv, restart, NOUT, TIMESTEP)) {
  output.write("Failed to initialise solver. Aborting\n");
  return(1);
}
```

which passes the (physics module) RHS function `physics_run` to the solver along with the number and size of the output steps.

To run the solver using the (already supplied) settings, there is the function:

```
typedef int (*MonitorFunc)(BoutReal simtime, int iter, int NOUT);
int run(MonitorFunc f);
```

9.1 Implementation: PVODE

9.2 Implementation: IDA

9.3 Implementation: PETSc

10 File I/O

BOUT++ needs to deal with binary format files to read the grid; read and write restart restart files; and write dump files. The two parts of the code which need to read and write data are therefore the grid routines (`grid.hxx` and `grid.cxx`), and the `Datafile` class (`datafile.hxx` and `datafile.cxx`). All other parts which need to read or write data go through these methods.

Several different file formats are commonly used, such as HDF, HDF5, and netCDF. For historical reasons (inherited from BOUT), BOUT++ originally used the Portable Data Binary (PDB) format developed at LLNL. To separate the basic file format functions from the higher level grid and Datafile classes, these use an abstract class `DataFormat`. Any class which implements the functions listed in `dataformat.hxx` can therefore be passed to grid or datafile. This makes implementing a new file format, and switching between formats at run-time, relatively straightforward.

Access to data in files is provided using a Bridge pattern: The `Datafile` class provides an interface to the rest of the code to read and write variables, whilst file formats implement the `Dataformat` interface.

```
class Datafile {
 public:
  Datafile();
  Datafile(DataFormat *format);
  ~Datafile();

  /// Set the file format by passing an interface class
  void setFormat(DataFormat *format);

  void setLowPrecision(); ///< Only output floats

  void add(var, const char *name, int grow = 0);

  int read(const char *filename, ...);
  int write(const char *filename, ...);
  int append(const char *filename, ...);
  bool write(const string &filename, bool append=false);

  /// Set this to false to switch off all data writing
  static bool enabled;
};
```

The important bits of the DataFormat interface are:

```
class DataFormat {
 public:
  bool openr(const char *name);
  bool openw(const char *name, bool append=false);

  bool is_valid();

  void close();
```

```
9
10    const char* filename();
11
12    const vector<int> getSize(const char *var);
13    const vector<int> getSize(const string &var);
14
15    // Set the origin for all subsequent calls
16    bool setOrigin(int x = 0, int y = 0, int z = 0);
17    bool setRecord(int t); // negative -> latest
18
19    // Read / Write simple variables up to 3D
20
21    bool read(int *var, const char *name, int lx = 1, int ly = 0, int ↩
          lz = 0);
22    bool read(BoutReal *var, const char *name, int lx = 1, int ly = 0, ↩
          int lz = 0);
23
24    bool write(int *var, const char *name, int lx = 0, int ly = 0, int ↩
          lz = 0);
25    bool write(BoutReal *var, const char *name, int lx = 0, int ly = 0,↩
           int lz = 0);
26
27    // Read / Write record-based variables
28
29    bool read_rec(int *var, const char *name, int lx = 1, int ly = 0, ↩
          int lz = 0);
30    bool read_rec(BoutReal *var, const char *name, int lx = 1, int ly =↩
           0, int lz = 0);
31
32    bool write_rec(int *var, const char *name, int lx = 0, int ly = 0, ↩
          int lz = 0);
33    bool write_rec(BoutReal *var, const char *name, int lx = 0, int ly ↩
          = 0, int lz = 0);
34
35    // Optional functions
36
37    void setLowPrecision();
38 };
```

11 Options

To control the behaviour of BOUT++ a set of options is used, with options organised into sections which can be nested. To represent this tree structure there is the `Options` class defined in **bout++/include/options.hxx**

```
class Options {
 public:
  // Setting options
  void set(const string &key,const int &val,const string &source="");
  ...
  // Testing if set
  bool isSet(const string &key);
  // Getting options
  void get(const string &key,int &val,const int &def,bool log=true);
  ...
  // Get a subsection. Creates if doesn't exist
  Options* getSection(const string &name);
};
```

To access the options, there is a static function (singleton)

```
  Options *options = Options::getRoot();
```

which gives the top-level (root) options class. Setting options is done using the `set()` methods which are currently defined for `int`, `BoutReal`, `bool` and `string`. For example:

```
  options->set("nout", 10);       // Set an integer
  options->set("restart", true); // A bool
```

Often it's useful to see where an option setting has come from e.g. the name of the options file or "command line". To specify a source, pass it as a third argument:

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

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

```
  Options *section = options->getSection("mysection");
  section->set("myswitch", true);
```

To get options, use the `get()` method which take the name of the option, the variable to set, and the default value.

```
  int nout;
  options->get("nout", nout, 1);
```

Internally, `Options` converts all types to strings and does type conversion when needed, so the following code would work:

```
Options *options = Options::getRoot();
options->set("test", "123");
int val;
options->get("test", val, 1);
```

This is because often the type of the option is not known at the time when it's set, but only when it's requested.

By default, the `get` methods output a message to the log files giving the value used and the source of that value. To suppress this, set the `log` argument to `false`:

```
options->get("test", val, 1, false);
```

11.1 Reading options

To allow different input file formats, each file parser implements the `OptionParser` interface defined in **bout++/src/sys/options/optionparser.hxx**

```
class OptionParser {
 public:
  virtual void read(Options *options, const string &filename) = 0;
 private:
};
```

and so just needs to implement a single function which reads a given file name and inserts the options into the given `Options` object.

To use these parsers and read in a file, there is the `OptionsReader` class defined in **bout++/include/optionsreader.hxx**

```
class OptionsReader {
 public:
 void read(Options *options, const char *file, ...);
 void parseCommandLine(Options *options, int argc, char **argv);
};
```

This is a singleton object which is accessed using

```
OptionsReader *reader = OptionsReader::getInstance();
```

so to read a file **BOUT.inp** in a directory given in a variable `data_dir` the following code is used in **bout++.cxx**:

```
Options *options = Options::getRoot();
OptionsReader *reader = OptionsReader::getInstance();
reader->read(options, "%s/BOUT.inp", data_dir);
```

To parse command line arguments as options, the `OptionsReader` class has a method:

```
reader->parseCommandLine(options, argc, argv);
```

This is currently quite rudimentary and needs improving.

12 Miscellaneous

Other small modules which don't really fit into any system, but are needed.

12.1 Printing messages

12.2 Timing

To time parts of the code, and calculate the percentage of time spent in communications, file I/O, etc. there is the `Timer` class defined in **include/bout/sys/timer.hxx**. To use it, just create a `Timer` object at the beginning of the function you want to time:

```
#include <bout/sys/timer.hxx>

void someFunction() {
  Timer timer("test")
  ...
}
```

Creating the object starts the timer, and since the object is destroyed when the function returns (since it goes out of scope) the destructor stops the timer.

```
class Timer {
public:
  Timer();
  Timer(const std::string &label);
  ~Timer();

  double getTime();
  double resetTime();
};
```

The empty constructor is equivalent to setting `label = ""`. Constructors call a private function `getInfo()`, which looks up the `timer_info` structure corresponding to the label in a `map<string, timer_info*>`. If no such structure exists, then one is created. This structure is defined as:

```
struct timer_info {
  double time;    ///< Total time
  bool running;   ///< Is the timer currently running?
```

```
4   double started; ///< Start time
5 };
```

Since each timer can only have one entry in the map, creating two timers with the same label at the same time will lead to trouble. Hence this code is **not** thread-safe.

The member functions `getTime()` and `resetTime()` both return the current time. Whereas `getTime()` only returns the time without modifying the timer, `resetTime()` also resets the timer to zero.

If you don't have the object, you can still get and reset the time using static methods:

```
1 double Timer::getTime(const std::string &label);
2 double Timer::resetTime(const std::string &label);
```

These look up the `timer_info` structure, and perform the same task as their non-static namesakes. These functions are used by the monitor function in **bout++.cxx** to print the percentage timing information.

12.3 Iterating over ranges

The boundary of a processor's domain may consist of a set of disjoint ranges, so the mesh needs a clean way to tell any code which depends on the boundary how to iterate over it. The `RangeIterator` class in **include/bout/sys/range.hxx** and **src/sys/range.cxx** provides this.

RangeIterator can represent a single continuous range, constructed by passing the minimum and maximum values.

```
1 RangeIterator it(1,4);  // Range includes both end points
2 for(it.first(); !it.isDone(); it.next())
3   cout << it.ind; // Prints 1234
```

A more canonical C++ style is also supported, using overloaded ++, $*$, and =! operators:

```
1 for(it.first(); it != RangeIterator::end(); it++)
2   cout << *it; // Prints 1234
```

where `it++` is the same as `it.next()`, and `*it` the same as `it.ind`.

To iterate over several ranges, `RangeIterator` can be constructed with the next range as an argument:

```
1 RangeIterator it(1,4, RangeIterator(6,9));
2 for(it.first(); it != RangeIterator::end(); it++)
3   cout << *it; // Prints 12346789
```

and these can be chained together to an arbitrary depth.

To support statements like

```
1 for(RangeIterator it = mesh->iterateBndryLowerY(); !it.isDone(); it↩
    ++)
2   ...
```

the initial call to `first()` is optional, and everything is initialised in the constructor.

12.4 Error handling

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.

Index