BOUT++ Reference Manual

B.Dudson, University of York

December 17, 2013

Contents

1 Introduction

This is the reference manual for the BOUT++ code[**?**, **?**], a C++ development of the BOUT code developed by X.Xu and M.Umansky at LLNL. This document describes the inner workings of the code. Users of the code who want to modify the equations solved should read the users manual first.

2 Overview

Classes implemented in BOUT++ are:

- `Field2D` - An object for 2D X-Y profiles. Handles memory allocation transparently, and uses overloaded operators to simplify indexing.
- `Field3D` - Wrapper for 3D fields. Again uses overloaded operators.
- `FieldPerp` - Perpendicular (X-Z) fields, primarily used for fields solvers.
- `Vector2D` - An object storing a vector (3 Field2D components) constant in Z.
- `Vector3D` - Stores a 3D vector (3 Field3D components)
- `Solver` - Interface between BOUT++ and the solver (PVODE). All functions specific to PVODE are contained in this class.
- `Datafile` - Provides a simple means of reading and writing dump and restart files, currently to PDB. Changing the file format means changing this class.
- `Communicator` - Parallel communication object. Wraps up the details of transferring data between processors, providing interface independent of MPI or details like number of processors and topology.
- `bstencil`
- `bvalue`
- `OptionFile` - reads a settings file, providing an interface to obtain values of named variables
- `Output` - A class for writing to stdout and/or a log file.

In addition there are several files which provide operations on these objects.

- **grid.cpp** - Reads a uedge.grd file
- **difops.cpp** - provides differential operators

- **boundary.cpp** - Boundary conditions NOTE: NEEDS IMPROVING
- **initialprofiles.cpp** - sets initial profiles for variables
- **invert_laplace.cpp** - Inverts a laplacian equation using fast linear method (FFT + tridiagonal solver)
- **invert_gmres.cpp** - provides a simple interface for inversion problems, intended for inverting for ϕ with nonlinear terms. NOTE: NOT FINISHED YET
- **topology.cpp** - calculates the mesh topology
- **utils.cpp** - provides some useful utilities, primarily memory allocation
- **bout++.cpp** - main function

The physics model specific code is in **2fluid.cpp**, solving simplified 2-fluid equations. This is intended for benchmarking.

3 Classes

3.1 Field2D, Field3D and FieldPerp

These field classes describe variables in 2 or 3 dimensions. All meaningful addition, subtraction, multiplication, division and exponentiation operators are overloaded, handling indexing transparently.

This is intended to separate details such as the size of the grid from the policy of what operations should be performed. This both makes the physics code easier to read and write, and reduces the risk of bugs.

3.1.1 Operators

Table 1: Field Assignments

Assign from	Function
`Field3D = Field3D`	copy all values
`Field3D = Field2D`	set constant in z
`Field3D = FieldPerp`	
`Field3D = bvalue`	changes a single value
`Field3D = real`	Sets all point to that value (e.g. set to zero)
`Field2D = Field2D`	
`Field2D = real`	
`FieldPerp = FieldPerp`	

3.1.2 Memory management

Many operations require the creation and destruction of intermediate fields, for example

```
Field2D a0
Field3D a, b;
b = (a + a0)^2.0
```

would require an intermediate Field3D object for `a + a0`. It would be inefficient to keep allocating and freeing memory each time, since the same operations will be performed for each time-step.

Memory is handled in all Field classes by maintaining a global stack of memory blocks (since all Fields of a given type are of the same size). When a new Field object is created, if a memory block is available then it is used, otherwise a new memory block is allocated. When an object is deleted, it's memory block is put back onto the stack rather than being freed.

The result of this is that the first time a given code is run, enough memory is allocated to hold the maximum number of fields used at any given time (since these blocks will be re-used by several fields). After this first time, no memory needs to be allocated or freed which should result in a significant increase in speed.

3.2 Vector2D and Vector3D

These classes implement 3D (x,y,z) vectors. The `Vector2D` class uses 3 `Field2D` objects to store the components and so has no toroidal (z) variation. The `Vector3D` class uses three `Field3D` objects.

These vectors can be added/subtracted from each other, and multiplied/divided by reals, `Field2D` or `Field3D` objects. The dot-product is done by overloading the `*` operator, whilst the cross-product is done using the $\wedge$ operator.

Since all the actual data is stored in `Field` objects, all memory management and operations on the data are done by the `Field` methods.

The vector classes are written to be applicable to a general curvilinear coordinate system; the metric tensor and Jacobian (g** and J in **globals.h**) are used to perform dot and cross-products.

3.3 Solver

This is an interface to the time-integration code, currently PVODE.

- `void add(`**Field2D&** `v, `**Field2D&** `F_v);`
- `void add(`**Field3D&** `v, `**Field3D&** `F_v);`
- `void setPrecon(`**rhsfunc** `g);`

- **int** `init(`**real** `tstart, `**rhsfunc** `f, `**int** `argc, `**char**** `argv);`
- **real** `run(`**real** `tout);`

3.4 Datafile

- `void add(`**int&** `i, `**char*** `name, `**int** `grow);`
- `void add(`**real&** `i, `**char*** `name, `**int** `grow);`
- `void add(`**Field2D&** `f, `**char*** `name, `**int** `grow);`
- `void add(`**Field3D&** `f, `**char*** `name, `**int** `grow);`
- **int** `read(`**char*** `filename);`
- **int** `write(`**char*** `filename);`
- **int** `append(`**char*** `filename);`

To use this, a set of variables are first added to the object. If the `grow` flag is set, then the number of dimensions in the output is increased by one and extended each time `append` is called. For example, the code:

```
Datafile output;
int it;
real time;

output.add(it, ``iteration'', 0);
output.add(time, ``time'', 0);
output.add(time, ``t_array'', 1);

it = 0;
time = 0;

output.write(``file.pdb'');
do {
  it++;
  time = 0.1*( (real) it);
  output.append(``file.pdb'');
}while(it < 9);
```

would result in an output file containing

```
integer  iteration   = 9
real     time        = 0.9
real     t_array[10] = {0.0, 0.1, ... , 0.9}
```

3.5 Communicator

This is a class which contains all the parallel communication code. The user first specifies which variables are to be communicated, then whenever necessary tells the object to perform the communication. This means that several different sets of variables can be communicated at different times by using several Communicator objects.

- `void add(`**Field2D&** `f);` - Specifies that the supplied 2D field is to be communicated. The memory location of this object should not change over the lifetime of a Communicator object.
- `void add(`**Field3D&** `f);` - Specifies a 3D field to be communicated.
- `void send();` - Begins the communication process, by posting receives and sending the data
- `void receive();` - Finishes a communication by waiting for all variables to be received. send() and receive() calls should always be used in that order in pairs.
- `void run();` - This performs communications by calling send() then receive(). The purpose of having separate calls is so that a code could perform additional calculations whilst waiting for data transfer.

An example of using the communication object:

```
Communicator comms;
Field2D a0;
Fiedl3D a, b;

// Add variables to the communication object
comms.add(a0);
comms.add(a);
comms.add(b);

// Set values
a0 = ...
a = ...
b = ...

comms.run() // perform communications

// more calculations here
```

3.6 bstencil and bvalue

3.7 OptionFile

This is an object to read a text file containing settings, formatted like an ini file. Sections are given in square brackets, and a semicolon comments out the rest of the line. Whitespace (tabs, spaces, empty lines) is ignored:

```
[section name]
name = value  ; comments
```

Functions provided are:

- **int** `read(`**char*** `filename);`
- **int** `getInt(`**char*** `name,` **int&** `val);`
- **int** `getInt(`**char*** `section,` **char*** `name,` **int&** `val);`
- **int** `getReal(`**char*** `name,` **real&** `val);`
- **int** `getReal(`**char*** `section,` **char*** `name,` **real&** `val);`
- **char*** `getString(`**char*** `name);`
- **char*** `getString(`**char*** `section,` **char*** `name);`
- **int** `getBool(`**char*** `name,` **int&** `val);`
- **int** `getBool(`**char*** `section,` **char*** `name,` **int&** `val);`

To use this class, first read in an option file (`read` routine), then request members by name, passing the value by reference. These functions return zero if the option exists, except the `getString` routines which return `NULL` if the option isn't found. The `getBool` functions treat any string beginning with a '1', 'y' or 't' as true (1), and any string beginning with '0', 'n' or 'f' as false (0).

3.8 Output

This class implements a similar variable-argument function to `printf`, redirecting the output to stdout and/or a specified log file.

4 Other functions

4.1 Grid file reading (grid.cpp)

The functions to read uedge.grd files are in **grid.cpp**, which supplies the following functions:

- **int** `grid_read(char *name);` - Read the geometry values needed by BOUT++ into global variables.
- **int** `grid_load2d(Field2D &var, char *name);` - Read a single 2D field from the grid file. This is used by the user to read in initial profiles.

4.2 Differential operators (difops.cpp)

4.3 Vector differential operators (vecops.cpp)

This file implements the following operators

$$\begin{aligned} \mathbf{v} &= \nabla f \\ f &= \nabla \cdot \mathbf{a} \\ \mathbf{v} &= \nabla \times \mathbf{a} \\ \mathbf{v} &= \mathbf{a} \cdot \nabla \mathbf{b} \end{aligned}$$

4.4 Boundary conditions (boundary.cpp)

These functions currently just set Neumann or Dirichlet boundary conditions, either by copying values into boundary conditions (zero gradient), or setting boundary regions to zero (zero value).

4.4.1 Possible implementation mechanisms

Regardless of the method used internally, the user interface to supply boundary conditions should be simple, intuitive, and require no

5 Wish List

List of things which could (and some which should) be added or improved

- Optimize Field classes by using copy-on-change. Setting `a = b` would just cause a's data to point to the same location as b. Only once one of these objects was changed would another copy of the data be created. This is also necessary for changing the number of grid-points dynamically (see next item)

- Allow variable number of grid-points on a processor. Putting branch-cuts into stencils, it should be possible to run on a single processor. This would allow
  - More flexibility in the number of grid-points in the legs
  - Load balancing by shifting grid-points between neighbouring processors during a run
  - Adaptive gridding (eventually)

  Since all fields are stored in global arrays (within classes), it should be fairly easy to re-size all arrays simultaneously, regardless of whether they are global or user arrays.
- Create Vector3D and Vector2D classes, and differential operators on them. This would simplify implementation of ideal MHD or similar equations.
- Upgrade the Solver class to use a new version of CVODE. Amongst other things, the new version allows a return code to signal unphysical values.
- Boundary conditions: Need to think of how to properly implement boundary conditions, particularly more complicated conditions such as sheath BCs.

6 Notes for users of BOUT

Significant changes

- The definition of ZMIN, ZMAX, zlength and dz has been changed. These no longer need a factor of hthe0. zlength and dz are in radians, and ZMIN/ZMAX are fractions of 2π, so to simulate $1/5^{th}$ of a torus, $ZMAX - ZMIN = 0.2$.

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.

Table 2: Field compound assignment operators

Operator	Function
`Field3D += Field3D`	Add all points
`Field3D += Field2D`	Treat 2D field as constant value in z
`Field2D += Field2D`	
`FieldPerp += FieldPerp`	
`FieldPerp += Field3D`	
`FieldPerp += Field2D`	
`FieldPerp += real`	
`Field3D -= Field3D`	subtract all points
`Field3D -= Field2D`	subtract constant value in z
`Field2D -= Field2D`	
`FieldPerp -= FieldPerp`	
`FieldPerp -= Field3D`	
`FieldPerp -= Field2D`	
`FieldPerp -= real`	
`Field3D *= Field3D`	
`Field3D *= Field2D`	
`Field3D *= real`	
`Field2D *= Field2D`	
`Field2D *= real`	
`FieldPerp *= FieldPerp`	
`FieldPerp *= Field3D`	
`FieldPerp *= Field2D`	
`FieldPerp *= real`	
`Field3D /= Field3D`	
`Field3D /= Field2D`	
`Field3D /= real`	
`Field2D /= Field2D`	
`Field2D /= real`	
`FieldPerp /= FieldPerp`	
`FieldPerp /= Field3D`	
`FieldPerp /= Field2D`	
`FieldPerp /= real`	
`Field3D` $\wedge =$ `Field3D`	
`Field3D` $\wedge =$ `Field2D`	
`Field3D` $\wedge =$ `real`	
`Field2D` $\wedge =$ `Field2D`	
`Field2D` $\wedge =$ `real`	
`FieldPerp` $\wedge =$ `FieldPerp`	
`FieldPerp` $\wedge =$ `Field3D`	
`FieldPerp` $\wedge =$ `Field2D`	
`FieldPerp` $\wedge =$ `real`	

Table 3: Binary field operations

Binary operator	Result	source file
Field2D + Field2D	Field2D	field2d.c
Field2D + Field3D	Field3D	field2d.c
Field2D - Field2D	Field2D	field2d.c
Field2D - Field3D	Field3D	field2d.c
Field2D * Field2D	Field2D	field2d.c
Field2D * real	Field2D	field2d.c
real * Field2D	Field2D	field2d.c (non-member)
Field2D * Field3D	Field3D	field2d.c
Field2D / Field2D	Field2D	field2d.c
Field2D / real	Field2D	field2d.c
real / Field2D	Field2D	field2d.c (non-member)
Field2D / Field3D	Field3D	field2d.c
Field2D $\wedge$ Field2D	Field2D	field2d.c
Field2D $\wedge$ real	Field2D	field2d.c
real $\wedge$ Field2D	Field2D	field2d.c (non-member)
Field2D $\wedge$ Field3D	Field3D	field2d.c