BOUT++ code structure

Ben Dudson

York Plasma Institute, Department of Physics,
University of York, Heslington, York YO10 5DD, UK

BOUT++ Workshop

14th September 2015

Getting BOUT++

Use of BOUT++

Contributing:

- BOUT++ is under the LGPL license, so code which uses it can be proprietry. Modifications to the BOUT++ library do come under the LGPL
- You're free to take and modify BOUT++ for any purpose
- We would appreciate it if you contributed back improvements you make to the code

Use of BOUT++

Contributing:

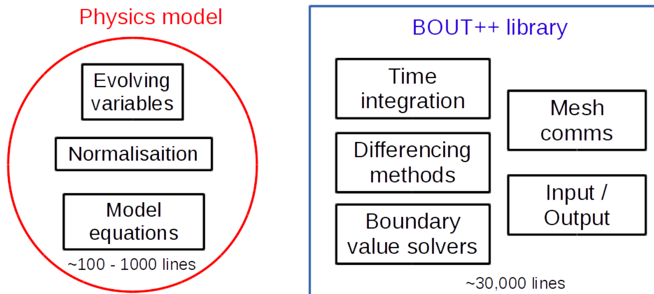
- BOUT++ is under the LGPL license, so code which uses it can be proprietry. Modifications to the BOUT++ library do come under the LGPL
- You're free to take and modify BOUT++ for any purpose
- We would appreciate it if you contributed back improvements you make to the code

Support:

- We're happy to help, but our time is limited
- One aim of this workshop is to get a group of people comfortable with using BOUT++ and (eventually) help support each other
- There is a BOUT++ development mailing list. Please let me know if you'd like to join it

Code structure

- Separates generic methods from model-specific code
- Most of the code doesn't know or care about what a variable represents, its normalisation etc. Only needs to know the geometry and which operation to perform



BOUT++ repository layout

The repository contains the following main directories:

- `manual/` contains documentation
  - **User manual**, introduction to BOUT++, installing and running
  - **Developer manual**, describes the internals of BOUT++
  - **Coordinates manual**, a collection of useful derivations in the field-aligned coordinate system used for tokamak simulations

BOUT++ repository layout

The repository contains the following main directories:

- `manual/` contains documentation
- `src/` contains BOUT++ library code
  - **`field/`** memory handling and arithmetic used throughout the codeoperations
  - **`fileio/`** Binary file input and output
  - **`invert/`** Inversion routines, particularly Laplacian inversion
  - **`mesh/`** Handling of mesh topology, metric tensor and MPI communication
  - **`physics/`** Miscellaneous routines useful for writing physics modules, such as gyro-averaging operators
  - **`solver/`** Time-integration solvers
  - **`sys/`** Miscellaneous low-level routines

BOUT++ repository layout

The repository contains the following main directories:

- `manual/` contains documentation
- `src/` contains BOUT++ library code
- `examples/` contains test suite and physics models
  - **`blob2d/`**, plasma blob in 2D
  - **`hasegawa-wakatani/`**, drift-wave turbulence in 2D
  - **`drift-instability/`**, resistive drift wave instability
  - **`interchange-instability/`**, resistive interchange mode
  - **`shear-alfven-wave/`**, Shear Alfvén wave
  - **`sod-shock/`**, standard 1D fluid shock problem
  - **`orszag-tang/`**, 2D MHD problem
  - **`uedge-benchmark/`**, 2D benchmark against UEDGE code
  - **`elm-pb/`**, ELM simulation code

BOUT++ repository layout

The repository contains the following main directories:

- `manual/` contains documentation
- `src/` contains BOUT++ library code
- `examples/` contains test suite and physics models
- `tools/` contains pre- and post-processing codes
  - **`idllib/`** Library of routines in IDL
  - **`pylib/`** Library of routines in Python
  - **`matlablib/`** Read BOUT++ output into Matlab
  - **`mathematicalib/`** Read data into Mathematica
  - **`slab/`** Sheared slab grid generator
  - **`tokamak_grids/`** codes for generating and converting tokamak equilibria and grid files

BOUT++ repository layout

The repository contains the following main directories:

- `manual/` contains documentation
- `src/` contains BOUT++ library code
- `examples/` contains test suite and physics models
- `tools/` contains pre- and post-processing codes
- `include/` and `lib/` contain header files and BOUT++ library

Starting with an example : `examples/conduction`

This solves heat conduction in 1D (in y):

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

Two variables are needed: T and χ (chi). In the code we define

```
Field3D T;
BoutReal chi;
```

- BoutReal is just an alias for double
- Field3D is a BOUT++ **class** or type, which handles 3D arrays. [Defined in `include/field3d.hxx`, code in `src/field/field3d.cxx`]

BOUT++ Fields

- The main function of the field classes is to provide automatic memory management, and looping over array indices.
- Before being used, must first be allocated or assigned a value

```
Field3D a;          // a has no data
a(1,3,2) = 1.0;     // Error!
a.allocate();       // a has data, undefined values
a(1,3,2) = 1.0;     // ok

Field3D b = 0.0;    // b has data, all zero
b(2,3,1) = 1.0;     // ok
```

This catches use of uninitialised data

BOUT++ Fields

- The main function of the field classes is to provide automatic memory management, and looping over array indices.
- Before being used, must first be allocated or assigned a value
  This catches use of uninitialised data
- Fields have overloaded operators and functions:

```
Field3D a = 1.0;   // Define a, set to 1.0
Field3D b = 2.0;   // Define b, set to 2.0

Field3D c = a + sqrt(a/b);
```

  Should be quite familiar to Fortran users, just remember **indices start from 0** in C/C++.

Physics model parts

Every physics model has two parts:

1. An initialisation function which is called (run) once at the start of a simulation
2. A run function which is usually called every time step

In the **examples/conduction** code, these appear as two C-style functions

```
int physics_init(bool restarting) {

  return 0;
}

int physics_run(BoutReal t) {

  return 0;
}
```

Physics model parts

Every physics model has two parts:

1. An initialisation function which is called (run) once at the start of a simulation
2. A run function which is usually called every time step

For those who prefer a more C++ style interface, **examples/conduction-newapi**:

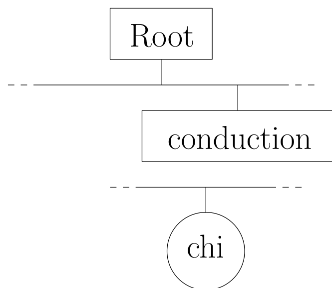
```
class Conduction : public PhysicsModel {
  protected:

  int init(bool restarting) {
    return 0;
  }

  int rhs(BoutReal t) {
    return 0;
  }
};
```

Reading options

We need a way to set the parameter χ. The **Options** class is a way to get input options. For example:

```
Options *options = Options::getRoot();
options = options->getSection("conduction");
options->get("chi", chi, 1.0);   // Read the option
```

If no value is set then the default (here 1.0) is used)



Code in `include/options.hxx` and `src/sys/options.cxx`

Setting options

To set this option we could either:

1. Put it in the input settings file `BOUT.inp`

```
[conduction]
chi = 2.5   # Heat conduction coefficient
```

2. Or override this on the command line

```
$ ./conduction conduction:chi=3.2
```

(note no space around ':' or '=')

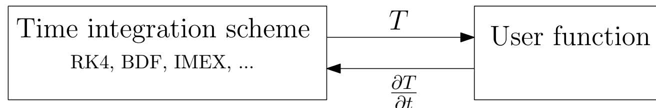
The value of `chi` used is printed to the log files `BOUT.log.*`

Evolving variables

To tell BOUT++ to evolve `T` in time, in the `init` function we call

```
SOLVE_FOR(T);
```

1. If starting a new simulation, T is set to initial value from options
2. At every time step solver will set T, then run the user code (`physics_run`)
3. The user code must calculate the time-derivatives, and return to the solver



Time and space are discretised separately: Method of Lines (MOL)

Calculating time derivatives

In the physics_run function, the evolving fields are given values, and the simulation time is an input parameter (t).

- **Communications are not done automatically**, so before taking derivatives of a field that field should be communicated

```
mesh->communicate(T); // Communicate guard cells
```

- The time derivative of a Field3D is another Field3D which can be accessed using ddt() (meaning $\partial/\partial t$, not d/dt)
- In this case we want an operator $\nabla\cdot\left(K\partial_{||}f\right)$. Fortunately there is a function Div_par_K_Grad_par which does this:

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

Derivative operators

The operators are usually found in `include/difops.hxx` and `src/mesh/difops.cxx`
Line 502:

```
const Field3D Div_par_K_Grad_par(BoutReal kY, Field3D
  return kY*Grad2_par2(f);
}
```

This is a function which takes two inputs: a heat conduction coefficient which doesn't depend on space, and a `Field3D` which depends on mesh location. It returns a `Field3D` containing the second derivative of the input.

$\Rightarrow$ Spatial operators are just functions which evaluate a finite difference formula and return a field.

Selecting the time integrator

- Many time integration schemes can be used, set by option

```
[solver]
type = rk4
```

  (command-line `solver:type=rk4`)
- A number of components can be changed like this at run-time
- Code does not depend on what type of solver is used

- Done by defining an interface and using a **factory** pattern

Factory pattern: consistent interfaces

- First define an interface which all solvers should have: include/bout/solver.hxx).

```
class Solver {
 public:
  virtual int init(bool restarting, int nout, Boutl
  virtual int run() = 0;  // Must be implemented
};
```

- Each implementation has this same interface: src/solver/impls/rk4/rk4.hxx

```
class RK4Solver : public Solver { // Is a type of
 public:
  int init(bool restarting, int nout, BoutReal tste
  int run();
};
```

Factory pattern: creating

When you need to create a solver

```
solver = Solver::create();
```

(`include/boutmain.hxx` line 111). This calls `src/solver/solver.cxx` line 861, which calls `src/solver/solverfactory.cxx`.

- The solver factory:
- Includes all the header files for each implementation
  1. Reads the “type” option (line 59)
  2. Then chooses which solver to create (line 70...) and returns it
- Returns the same thing (a `Solver*`) regardless of implementation
- This means that the code which called `Solver::create()` has no way of knowing which solver it got (*)

Factory pattern: benefits and issues

- This may seem to be a problem, but is actually a Good Thing
- The rest of the code must be independent of solver type, and solvers can be added easily
- There can be a temptation to “reach inside” a solver and access implementation-specific data or functions.
  - This must be resisted! It leads to messy, fragile code, more work, and lasting regret...
  - Instead the interface should be carefully considered: What is it a solver should do?
- Designing good interfaces is very hard; the BOUT++ ones have changed over time

Summary

- BOUT++ is a collection of useful classes and functions which work together
- Data on grid points is manipulated using arrays wrapped up in `Field3D` and objects (and siblings).
- Many components have a fixed interface, and implementation can be changed at run-time using the factory pattern
- Having good interfaces is important
- Major changes to some parts of the code coming this year (reorganisation of Mesh)