BOUT++ code structure

Ben Dudson
benjamin.dudson@york.ac.uk

Department of Physics, University of York, Heslington, York YO10 5DD, UK
Lawrence Livermore National Laboratory

14^{th} September 2011

BOUT++ conventions

Before getting into the code, there are some conventions used throughout:

- The X direction is usually ψ, and has boundaries called `core`, `pf` and `sol` (also `xinner` and `xouter`)
- The Y direction is along the field-line (for Clebsch coordinate operators). Boundaries called `target`, or `yupper` and `ylower`
- The Z direction is axisymmetric, so all metric tensors are constant in Z and FFTs can be used easily

BOUT++ conventions

Before getting into the code, there are some conventions used throughout:

- The X direction is usually ψ, and has boundaries called `core`, `pf` and `sol` (also `xinner` and `xouter`)
- The Y direction is along the field-line (for Clebsch coordinate operators). Boundaries called `target`, or `yupper` and `ylower`
- The Z direction is axisymmetric, so all metric tensors are constant in Z and FFTs can be used easily

The BOUT++ code is divided into two parts:

- The BOUT++ library, which provides generic routines for manipulating data, calculating differential operators, integrating ODEs etc.
- The physics module which describes a particular set of equations, coordinate system, and normalisation.

Aim is to separate out all the generic code, so this can be tested and not re-written every time. Physics code becomes smaller and more understandable.

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 several physics modules, including
  - `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 several physics modules, including
- `tools/` contains pre- and post-processing codes
  - `idllib/` lots of useful routines for reading and writing data, collecting and plotting the output from BOUT++
  - `pylib/` Beginnings of a library of Python routines
  - `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 several physics modules, including
- `tools/` contains pre- and post-processing codes
- `include/` and `lib/` contain header files and BOUT++ library

BOUT++ components

BOUT++ consists of some low-level data handling classes, and a collection of independent routines for manipulating them built on top

- Base classes and interfaces: `Field`, `FieldData`
- Classes representing scalar fields:
  - `Field2D`, representing quantities varying in X and Y. This includes metric tensor components, and usually equilibrium plasma quantities
  - `Field3D` represents a 3D array in X, Y and Z.
  - `FieldPerp`
- Classes representing vector fields: `Vector2D`, `Vector3D`
- Log file output: `Output` class
- Debugging message stack: `MsgStack` class
- Binary data input and output

Field classes

- 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. Catches use of uninitialised data.
- When fields are destroyed, memory will automatically be free'd or re-used. Field3D's internally use pointers to avoid data copying, allocation and freeing

Field classes

- 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. Catches use of uninitialised data.
- When fields are destroyed, memory will automatically be free'd or re-used. Field3D's internally use pointers to avoid data copying, allocation and freeing
- Fields have lots of overloaded operators, to allow expressions like

  ```
  Field3D a, b, c;
  ...
  a = b + (c^2) / b
  ```

  Each operation is calculated separately, looping over the mesh
- Isolates loops, making the rest of the code clearer

Log file output

To write messages to a log file, there is the `Output` class and global instance `output`. This can be used either like C's `printf`:

```
output.write("Message text", ...);
```

or using C++ streams:

```
output << "Message text" << ...;
```

Whatever is sent to output is sent to a file `BOUT.log.#` where '#' is the processor number. The output from processor 0 is also sent to `stdout`.

Source code in: `src/sys/output.cxx`
Global object in `include/globals.hxx`, line 124

Debugging messages

To help find bugs, BOUT++ uses a class called `MsgStack` with a single global instance `msg_stack`.

- At the beginning of a function or section of code, a message can be put onto the stack:

  ```
  msg_stack.push("Message text", ...);
  ```

  which has the same syntax as C's `printf` function.
- To remove the last message from the stack

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

- In the event of a segmentation fault, this is caught by `bout_signal_handler` (`src/bout++.cxx`, line 632) and the message stack is printed to the log file by calling

  ```
  msg_stack.dump();
  ```

Source code in: `src/sys/msg_stack.cxx`
Global object in `include/globals.hxx`, line 186

Binary data input and output

To read and write binary data, BOUT++ has the `Datafile` class in `include/datafile.hxx` and `src/fileio/datafile.cxx`.

- Variables are first added. The Datafile object stores a pointer to the variable, so it must not be destroyed before the datafile is used

```
Datafile file;
Field3D var;
file.add(var, "name");
```

- The variable can then be read or written to file

```
file.read("input_data.nc");

file.write("file_%d.nc", 10);
```

Datafile also handles time-dependent data, allowing files to be appended.

Binary data input and output

Reading: The Mesh class handles splitting the mesh between processors, reading data from the input file, and communications. To read a variable from the mesh file:

```
Field2D Ni0;
mesh->get(Ni0, "Ni0");
```

A shorthand if the name of the variable and the name in the input file are the same is

```
GRID_LOAD(Ni0);
```

Writing: There is a global Datafile object `dump` defined in `include/globals.hxx`, line 127. The macros

```
SAVE_ONCE(var);    // Output once
SAVE_REPEAT(var2); // Every time-step
```

save variables into the output file. Also `SAVE_ONCE2`...`SAVE_ONCE6` and `SAVE_REPEAT2`...`SAVE_REPEAT6`

Binary data input and output

Currently BOUT++ supports PDB and NetCDF file formats. This is done by having a common interface to file formats:

- `include/dataformat.hxx` defines which members must be defined
- The PDB file format is implemented in `src/fileio/pdb_format.hxx` and `src/fileio/pdb_format.cxx`
- The NetCDF format is implemented in `src/fileio/nc_format.hxx` and `src/fileio/nc_format.cxx`
- To add a new file format, create a new class which implements all the interface functions in `include/dataformat.hxx`. Add some code to the `data_format` function in `src/fileio/datafile.cxx` to detect the new format from the file name.

Input options

- Options are handled using a tree structure of `Options` objects, defined in `include/options.hxx` and `src/sys/options.cxx`
- There is a root object defined as a singleton in `include/options.hxx`, line 88. Obtain using

  ```
  Options *options = Options->getRoot()
  ```
- The `getSection()` and `get()` methods extract values:

  ```
  int setting;
  options->getSection("mysection")->get("mysetting",
  setting, 1);
  ```

  This will fetch a value called "mysetting" in a section "mysection", and attempt to convert it to an integer. If the setting isn't found, then the default value (1 here) will be used.

Input options shorthand

Usually the name of the variable, and the name of the setting are the same, so to save typing there are some shortcut macros defined in `globals.hxx`, line 62

First get the section you want

```
Options *options = Options->getRoot(); // Get root
options = options->getSection("mysection");
```

then use macros to get the options:

```
int a;
OPTION(options, a, 4);
BoutReal b;
OPTION(options, b, 3.14);
```

To read several options, there are additional macros

```
int a, b;
OPTION2(options, a, b, 4);
```

for up to 6 variables: `OPTION2` ... `OPTION6`.

Solving physics problems

The `Field` classes, text and binary data input and output, and error handling provide the basic functionality on which the rest of the BOUT++ code is built

- Time-integration solvers, such as RK4 and interfaces to external timestepping routines in SUNDIALS and PVODE
- Mesh handling, communications
- Boundary conditions
- Differential operators, combining differencing methods with metric tensor components

Time-integration

To advance the time, the time-derivative of all quantities needs to be calculated. To store this data, every field variable contains a pointer to another field which contains it's time-derivative. This can be accessed using the `timeDeriv()` method:

```
Field3D var;
Field3D *deriv = var.timeDeriv();
```

The time integration solvers supply the system state in `var`, and expect the time-derivative values to be in `deriv`. As a shorthand,, a macro is defined in `include/globals.hxx`, line 231:

```
#define ddt(f) (*((f).timeDeriv()))
```

which allows us to use `ddt(var)` as a variable, e.g:

```
ddt(var) = ...
```

Time-integration

To tell BOUT++ that a variable should be evolved, there are the functions:

```
bout_solve(Ni, "density");
```

defined in `src/bout++.cxx` at line 567. This just calls `solver->add`, associating the variable with its time-derivative. As with the file input/output and options, there is a shorthand macro if the name of the variable and the name of the output are the same:

```
SOLVE_FOR(Ni);
```

and also `SOLVE_FOR2` ... `SOLVE_FOR6`
There is no limit on the number of variables which can be evolved, apart from memory and run-time.

Time-integration

Like the binary data files, BOUT++ defines an interface which solvers must implement. Multiple solvers are compiled into the library, and can be switched at run-time.

- `Solver` base class provides generic routines for solvers, such as loading data to and from variables and time-derivatives. Defined in `include/solver.hxx` and `src/solver/solver.cxx`
- Time-integration solvers implemented in `src/solver/impls/`
- See Euler and RK4 solvers to see how they work. More tomorrow on solvers

Differential operators

Differential operators such as $\nabla_{||}$ or $\mathbf{b} \times \nabla f \cdot \nabla g$ are handled in two levels:

- Low-level differentials, which just calculate $\partial / \partial x$
  These are in `src/sys/derivs.cxx`
- High-level operations, which combine differentials and metric tensor components into physical operators like $\nabla_{||}$ and $\mathbf{b} \times \nabla f \cdot \nabla g$
  These are in `src/mesh/difops.cxx` and `src/field/vecops.cxx`

Adding a new differencing scheme

If the differencing operator can be implemented as a 1D operator (MOL), then it is in `src/sys/derivs.cxx`

- The function needs to be defined on a stencil, see existing implementations at top of `src/sys/derivs.cxx`
- Define a new DIFF_METHOD code for your method. `bout_types.hxx` line 39
- To translate between input strings and DIFF_METHOD codes, put your method into `DiffNameTable` in `src/sys/derivs.cxx`, line 374
- Add your function to a lookup table corresponding to the type of derivative, starting in `src/sys/derivs.cxx` at line 386

If your method needs more information than a single stencil, define it in `src/mesh/difops.cxx`. See the `bracket` function there for some examples e.g. Arakawa scheme.

Summary

BOUT++ is a collection of classes and routines which allow plasma fluid simulations to be quickly developed and different numerical methods tried

- Generic data handling, and input/output facilities
- On top of these are built differential operators and interfaces to time-integration solvers
- Separation into interfaces and implementations, allowing methods to be chosen at run-time using options
- Allows a lot of common code to be written and debugged once then re-used many times
- Improvements to the core BOUT++ library can be used by all physics codes without any changes (if backwards-compatible)