Excalibur-Neptune report 2047356-TN-05-2

Task 4.2 BOUT++ multifluid implementation

Ben Dudson, Peter Hill, Ed Higgins, David Dickinson, and Steven Wright

University of York

David Moxey

University of Exeter

October 28, 2021

Contents

1 Executive summary

In this report BOUT++ [1] is used as a platform to test design patterns which enable plasma simulations models to be extended from single ion species to multiple species. The aims are to:

1. Evaluate code designs which enable single-species plasma simulations to be extended to multiple species, in terms of their flexibility and complexity of implementation. This can inform the design of models built on other frameworks, including finite element and spectral libraries.

2. Explore Domain Specific Languages (DSLs) for specifying complex multi-fluid models, and provide a platform for further development.

2 Introduction

An important feature of tokamak edge plasmas is that they consist of a mixture of different species. Physics studies of some phenomena, such as plasma turbulence, have frequently treated the plasma as if it consisted of a single ion species (eg. deuterium). In order to make predictions of phenomena important to the performance and design of the divertor, models must capture the interactions between plasma and neutral gas species (deuterium and tritium in a reactor), as well as radiation from impurity species (eg. Be, C, Ne, Ar, W). Efficient pumping of helium "ash" is also an important aspect of the design. In many cases none of these species can be treated as "trace", but are all coupled, so that all should be considered self-consistently in the same simulation. For each of these atomic species, the density and dynamics of multiple states may need to be considered, for example the ground state, multiple ionisation stages, and molecular species. In some cases it may also be necessary to track metastable states, such as vibrationally excited states or charged molecules. The result is a potentially large number of species types which must be solved for, together with a complex set of reactions between them.

If the number of species states to be solved for is relatively small, such as electrons, ions and a single atomic species as in Hermes-2 [2] and recent versions of STORM [3], then code can be written for each species. Unfortunately with this design the size of the code (number of lines) grows linearly with the number of species states, becoming increasingly error prone, difficult to test, and hard to maintain, as the number of species is increased. Experience with Hermes-2 indicated that this was not a viable path to multi-species simulations.

Some desirable features of a software design for this problem are:

1. The number and type of species to be solved for, and the reactions between them, should be **customisable**. Either at run-time with a user-friendly configuration language (DSL), or at compile time with minimal code changes.

2. The code must be **modular** such that it can be tested in small pieces. This makes finding and fixing errors more efficient than tests which run the whole of a large model.

3. Code modules must be **reusable**: Where the evolution equations for two species are the same, the same code should be used to perform those calculations. This is needed to limit growth of code size with number of species, and reduce the amount of code which needs to be tested.

4. The composition of these modules should be **reliable**: The configuration method (e.g. input DSL) should prevent unphysical or erroneous combinations of components, or these should be caught quickly at run-time. In particular the system design should prevent the following scenarios:

   - **Incorrect use**: A value is used in a way which is inconsistent, for example the units or normalisations are different.
   - **Set after use**: Calculating a quantity after it is used, so that it is either assumed to be zero (missing) or incorrect values are used (e.g uninitialised values, or values from a previous timestep).
   - **Update after use**: Similar to set after use, a term or quantity (e.g collision rate) is modified after it has been used. The result is inconsistent with what the user likely intended.
   - **Unused values**: A term which the user intends to be used, such as a source or sink, is calculated but never used.

In this report we describe two multi-fluid implementations, both built on top of BOUT++: In section 3 a 1D model developed from SD1D is described, which uses inheritance to define species behaviour, and a centralised system to coordinate interactions between species. In section 4 a more flexible code design is presented, which is capable of 1,2 or 3-dimensional multi-fluid simulations, and coordinates inputs and outputs of code components through a state object. The advantages and disadvantages of these approaches will be discussed. Finally in section 5.1 the Entity Component System (ECS) approach to solving similar problems is discussed, which was originally developed for game development.

3 Subclassing

In this section we describe an approach to building a multi-fluid simulation model using a class hierarchy, where object-oriented inheritance is used to adapt and specialise behaviour. This development started from the SD1D single-fluid code [4].

3.1 Architecture

In this design, the model is divided into `Species`, which represent particular evolving atomic or ion species, and `Reactions`, which represent all interactions between species. `Species` hold the data representing the state of the system (density, temperatures etc.), and have no knowledge of other species or interactions between them; all interactions are represented by externally specified sources and sinks of particles, momentum and energy. `Reactions` here represent all interactions between species, including effects such as ionisation and also things like thermal forces which would not normally be considered to be "reactions". The interaction between `Species` and `Reactions` objects is orchestrated by a central object representing the whole system, which creates and contains lists of the participating species and reactions objects. This is illustrated in figure 1

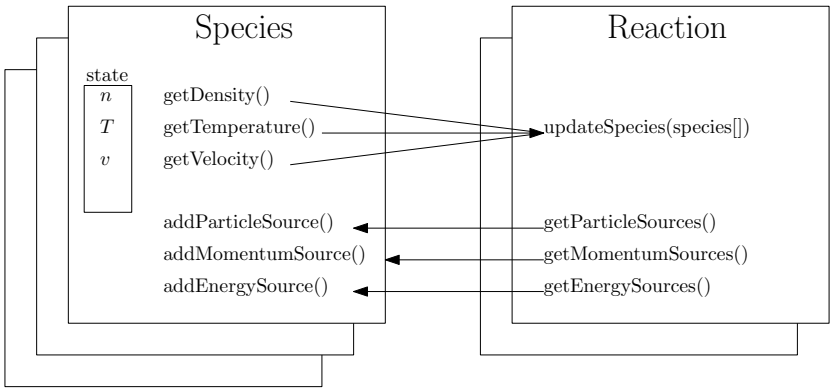

Figure 1: Subclassing architecture, in which `Species` objects describe plasma species, and `Reactions` represent interactions between them. All interactions are carried out through exchange of sources.

An implementation of this design is provided in SD1D-multifluid available at `https://github.com/boutproject/SD1D/tree/new-multifluid`. The evaluation of the time derivatives is done in the following steps:

1. Call the `evolve()` method for each species. The species updates its own density, velocity and temperature by taking values from the BOUT++ time integrator, which advances the entire system of all species syncronously.
2. Call the `updateSpecies()` method for each reaction, passing in the state of all species in the simulation. The reaction updates its own rates, using the state of all the species
3. The mass, momentum and energy sources and sinks are requested from each reaction, and added to the required species by passing the sources/sinks to the species objects.

Details of these steps are given below, starting with a description of the `Species` class.

3.1.1 Species

Every species has an atomic mass number and charge, and a density, temperature and velocity at every point in the domain. To distinguish between model equations for different types of species, the `Species` class can be subclassed. The `FluidSpecies` class evolves three equations for the density, momentum and pressure. These equations do not include coupling to other species, but are updated by a call to an `evolve()` function which is implemented for each `Species` class.

$$\frac{\partial n}{\partial t} = -\nabla\cdot\left[\mathbf{b}v_{||}n\right] \quad (1)$$

$$\frac{\partial}{\partial t}\left(\frac{3}{2}p\right) = -\nabla\cdot\mathbf{q}_e + v_{||}\partial_{||}p \quad (2)$$

$$\frac{\partial}{\partial t}\left(m_i n v_{||}\right) = -\nabla\cdot\left[m_i n v_{||}\mathbf{b}v_{||}\right] - \partial_{||}p \quad (3)$$

$$\mathbf{q}_e = \frac{5}{2}p\mathbf{b}v_{||} - \kappa_{||}\partial_{||}T_e \quad (4)$$

As with all BOUT++ models, the whole system is evolved together in time, so that the time integrator treats all time-evolving variables as part of a single state vector.

3.1.2 Reactions

The `evolve()` function for each species is not passed information on the other species present, and so cannot calculate interactions between species. The equations 1-4 for each species are therefore decoupled. To couple species together, additional terms need to be added to the right hand side of these equations.

To represent reactions, collisions, and other interactions between species, a set of `Reaction` classes are defined. Each reaction defines a method `updateSpecies()`, which is passed a collection (a C++ map) of all species in the simulation. The intention is that this function calculates all reaction rates, and exchanges of particles, momentum and energy between species.

Each reaction can also implement methods which return density, momentum and energy sources (`densitySources()`, `momentumSources()` and `energySources()`). These return C++ maps of `species` $\rightarrow$ `Field3D`, where a `Field3D` represents a spatially dependent scalar field, here representing the net source for the species due to the reaction.

In terms of python-like pseudo-code, the main loop of the SD1D multifluid implementation, which calculates the time derivatives of all evolving quantities, consists of:

```
for species in species_list:
  species.evolve()

for reaction in reaction_list:
  # Calculate reaction rates using all species
  reaction.updateSpecies(species_list)

  for species_name, source in reaction.densitySources():
    # Check that the species is in species_list
```

```
    species_list[species_name].addDensity(source)

  for species_name, source in reaction.momentumSources():
    species_list[species_name].addMomentum(source)

  for species_name, source in reaction.energySources():
    species_list[species_name].addEnergy(source)
```

3.2 Advantages and disadvantages

In terms of the desirable characteristics described in the introduction (section 2 (with subjective score)):

1. **Customisable** (medium): The species and reactions to be created can be specified at run-time: A list of species and a list of reactions are built during initialisation, and iterated over. The interaction between species is however limited to a single round, meaning that some interactions which depend on multiple steps (e.g. calculating electromagnetic fields, or collision rates, then using those quantities in a further calculation) are difficult to customise.

2. **Modular** (low): Each species can be largely tested in isolation, because it is self-contained and all interactions are through sources which can be specified as part of the test. `Reactions` similarly don't depend on each other, though do depend on Species. This provides some coarse grained modularity, but is not sufficient to simplify testing: The `Species` objects are however complex internally, and are coupled to the time integrator, making them hard to test with frameworks like GoogleTest. Furthermore creating `Species` objects to test `Reactions` can be quite complex.

3. **Reusable** (medium): The code for `Species` is quite reusable; the code to solve the fluid moment equations is the same for all species. Where there are species-specific terms, these can be added as `Reactions` (even though they only involve one species). The code for `Reactions` is however harder to reuse, and so there are many implementations of essentially the same reaction but using different rates. This is something which could likely

be improved, using inheritance for classes of reactions, but needs further design work.

4. **Reliable** (high): The coordination of interactions between species in a single centralised place enables a number of useful checks to be performed: The code can ensure that sources and sinks are always balanced between species. The simple interaction between components involving a single interaction (species state to reactions, then reaction sources to species) means there is little room for unexpected behaviour, but also limited ability to incorporate more complex multi-step calculations.

The two main disadvantages of this structured approach are that

1. The dependencies between components are very restricted. Complex interactions between different parts of the code should be minimised, but the scheme used here with only two steps (species evolve, reaction update) is overly restrictive: There is no way to, for example, modify the calculation of the collision frequency or electric field, and have that affect other calculations.

2. The class hierarchy of `Species` isn't really used: Only a single species type (`FluidSpecies`) is actually used, and all modifications to that species are represented as reactions. Thus composition of behaviour turns out to be more useful than inheritance.

Building on this experience, a different design which emphasises composition was developed for Hermes-3, described in the following section.

4 State-Command pattern

Following experience with the inheritance pattern used in SD1D-multifluid, a more abstract and general software architecture has been developed. This was done so that a wider variety of interactions and effects could be incorporated, than were possible with a single pass from species state to reactions. One of the key aspects considered was how to enable mutual dependencies between code components, while keeping code complexity manageable. This design has been

implemented in Hermes-3 [5], which has been developed starting from an existing Hermes-2 single-fluid model [2] with a manual hosted on ReadTheDocs [6] which describes in detail the code implemented. It is capable of 1-, 2-, and 3-dimensional simulations, and wide range of plasma models, configured by an input text file.

The following sections describe the overall architecture (sec 4.1), and discuss the advantages and disadvantages of this approach (sec 4.2).

4.1 Architecture

The main features of the state-command design are:

- A flexible store or database, into which values (e.g. spatially dependent fields like densities, temperatures) can be inserted and later retrieved. In Hermes-3 this is a nested dictionary strucure (a tree), using C++ `variant` to enable different data types to be stored. A schema defines a convention for where values are stored, for example `state["species"]["h+"]["density"]` would be the number density of hydrogen ions.
- A collection of composable model components, which set and use values in the store. For example there is a component which evolves an equation for the density, another component which evolves an equation for pressure. These components are configured when they are created, so that the same code is used to evolve every species which needs that component. Every component can access the whole state, so some perform calculations for a single species, while others perform calculations involving multiple species (e.g. collisions, sheath boundary conditions).

The flow of information carried by a state through a sequence of components is shown in figure 2.

An important distinction between this design and one with a shared global state is that the state is an object which is passed to components in a user-defined order. This has two advantages:

1. It facilitates unit testing, because the inputs to the components can be

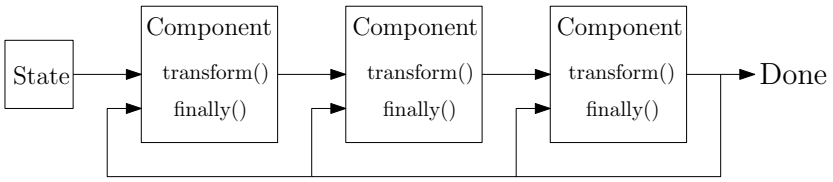

Figure 2: State-Command pattern: The `State` of the simulation system is passed through a sequence of `Components` in two passes. In the first pass (`transform()` calls), components can modify the state; in the second pass (`finally()` calls) the state cannot be modified, but is used to update component internal states.

precisely specified with no hidden side-channels or large setup/teardown procedures.

2. Access to the state can be controlled, making the logic of the program easier to follow. In Hermes-3 each component is passed the state twice: The first time the state is mutable, and the component can insert values into it. After all components have been called in this way, the state is "frozen", and passed to each component again as `const` and so cannot be modified. Each component can then use this final state to update its own internal state, such as time derivatives to be passed to the time integrator. This means that components can depend on each other, but all modifications must be in the first function (called `transform()`), and in the second function (called `finally()`) all components can assume that the state will not subsequently change[1].

4.2 Advantages and disadvantages

In terms of the desirable characteristics described in the introduction (section 2), with subjective scoring:

1. **Customisable** (medium): The species and reactions to be created can be specified at run-time: A list of components is built at run-time, and can in principle represent a system of equations for an arbitrary number

[1] This could be used to enable all component `finally()` functions to be called in parallel, but this is not currently done in Hermes-3.

of species. Components can interact with each other in complex, though not arbitrarily complex ways in order to simplify debugging.

2. **Modular** (high): The inputs to the components can be quite straightforwardly created in tests, and because the components don't interact with global variables or other shared state, the number of different tests required to cover all possible code paths is minimised. As a result of this, Hermes-3 has a collection of 50 unit tests (using GoogleTest) which cover most of the code.

3. **Reusable** (high): The code is broken up into smaller pieces than the SD1D-multifluid (subclassing) design, aiding reuse of components. Improvements to how the components and their configuration are specified in Hermes (a Domain Specific Language of sorts) make reusing components for reactions quite straightforward.

4. **Reliable** (low-medium): The flexible data structure used to store the state does not by itself provide many checks to ensure consistency and correctness. As implemented in Hermes-3, the user could specify an incorrect ordering of components, since dependencies are not explicitly specified in a way that would enable automatic ordering.

To mitigate some potential reliability issues ("set after use" and "update after use"), the setting and getting of values in the state is further controlled: If a component requests (`get`) a value from the state, then by default that value cannot be further modified. The assumption is that the component will treat the value as final, rather than a preliminary value which might be modified later by another component. In order to get a value which might be modified later, the more verbose `getNonFinal` function can be used.

An important corner case is components such as boundary conditions, where the value of the boundary cells is modified. The solution described above doesn't distinguish between cells in the boundary and in the domain, and therefore doesn't catch modifications after the boundary condition is applied. We therefore need to distinguish between boundary conditions and values in the domain; at any point in the calculation one or both of these could be considered "final".

The design used in Hermes-3 is that each value has two flags which can be set: `final-domain` means that the values in the domain are final and cannot be

modified; `final` means that all cells are final and cannot be modified. There are three ways to get values from the store: `get()` means the caller assumes the value everywhere (including boundaries) is final; `getNoBoundary()` means the boundary cells are not assumed final, and `getNonFinal()` means all values may be further modified later. There are also two ways to set values: `set()` can modify any cells, and checks that neither `final` nor `final-domain` flags have been set; `setBoundary()` modifies only the boundary, and checks that `final` flag has not been set.

After making these correctness improvements, failure modes described in section 2 are considered here:

- **Incorrect use**: The meaning and units for each quantity are not enforced automatically, but follow a documented schema. Automatic enforcement of compatible units is usually difficult because the equations being solved are made dimensionless. Further improvements might be possible if code and data were annotated, so that the type of data expected can be checked (e.g. is this data a rate, a density, a temperature?)
- **Set after use**: This failure is mostly prevented by marking values as final when they are used. This includes data which is missing (and so may be assumed to be zero in the consuming component), so that setting a value after it is used results in an error.
- **Update after use**: The same comments apply as for "set after use".
- **Unused values**: This is currently not prevented. It would be possible to check after all components have been run that all values have been used. There are cases where components can set values which are not used in the particular (perhaps simplified) model that the user is solving.

5 Alternatives

In the previous sections (3 and 4) two designs have been described and implemented using the BOUT++ framework to produce flexible plasma models which can be configured at run-time based on input settings. There are other software designs which aim to solve similar issues, which have been developed

within scientific computing and in other fields. Here we briefly describe some of these.

5.1 Entity Component Systems

Entity Component Systems (ECS) are a design pattern commonly used in game development. It is intended to describe a set of "Entities" which represent things which have defined sets of behaviour, and can interact with each other.

To define the behaviour of thse entities, an ECS enables components to be defined and composed, rather than defining a class hierarchy to represent different types of behaviour. These ECS systems also enable components to be iterated over in an efficient manner, by managing the mapping of components to contiguous memory structures.

A prominent and high-performance implementation of an ECS is EnTT [7]. Like the state-command pattern, it offers flexibility through composition rather than inheritance, and considerable run-time configurability.

For high performance scientific simulation codes, it is debatable whether run-time configuration is essential or even beneficial: Once the simulation is set up, it is performing the same set of operations repeatedly, calculating time derivatives given different system states. Run-time configuration enables the user (scientist) to modify the equations without recompiling, but means that errors are only caught at runtime, which might have been caught quicker at compile time. Compile-time configuration of the equations solved might enable more optimisations, since conditionals can be known and optimised out by the compiler. It is possible that Just-In-Time (JIT) compilation might offer the best of both worlds; since the operations are the same with different data, run-time analysis of the performance might enable real-time tuning to identify bottlenecks and optimise throughput.

5.2 Task graphs

The majority of the code lines and computation cost of a simulation consists of calculating the time derivatives of a system of equations, turning a PDE into an

ODE which can be then integrated by standard explicit or implicit ODE solvers. The outputs (the time derivatives) depend on quantities such as collision times and flows, which in turn may depend on calculations of electromagnetic fields or boundary conditions. The whole calculation can be viewed as a task graph.

Task graphs are a set of tasks, each with inputs and outputs; the outputs from some tasks act as the input to other tasks. This is quite a general way of thinking about computations, and how to break them up into pieces. An advantage from a performance perspective is that these tasks can be automatically scheduled, and in some cases moved between CPU and GPU depending on available resources. Examples of task-based systems include StarPU [8] and TaskFlow[9] which can use SYCL as an implementation language for cross-platform programming.

6 Conclusions

Two multi-fluid plasma models have been implemented on top of the BOUT++ framework, using different architectures to achieve the flexibility and code reuse needed for practical multi-fluid implementations. The desirable qualities of such architectures have been discussed, and the implementations evaluated against them. Lessons learned include that composition is a better model than inheritance, and that plasma models quite naturally fit into a model of interconnected tasks. Ideas for further refinement, and alternative directions to explore have been suggested.

7 References

[1] BOUT++ contributors. BOUT++ manual. `https://bout-dev.readthedocs.io/`.

[2] Ben Dudson, Jarrod Leddy, Hasan Muhammed. Hermes-2 hot ion drift-reduced model. `https://github.com/bendudson/hermes-2`.

[3] L. Easy, F. Militello, T. Nicholas, J. Omotani, F. Riva, N. Walkden, UKAEA. Hermes-2 hot ion drift-reduced model. `https://github.com/boutproject/STORM`.

[4] Ben Dudson et al. SD1D SOL and Divertor model in 1D. https://github.com/boutproject/SD1D.

[5] Ben Dudson. Hermes-3. `https://github.com/bendudson/hermes-3`.

[6] Ben Dudson. Hermes-3 manual. `https://hermes3.readthedocs.io`.

[7] Michele Caini. EnTT: Gaming meets modern C++. https://github.com/skypjack/entt, 2017-2021.

[8] INRIA. StarPU - A Unified Runtime System for Heterogeneous Multicore Architectures. https://starpu.gitlabpages.inria.fr/.

[9] Taskflow - A General-purpose Parallel and Heterogeneous Task Programming System. https://github.com/taskflow/taskflow.