Primer on Mara Computational Library

Andrew Liu Jonathan Zrake

May 18, 2018

1 Introduction and Software Architecture

1.1 Background

The objective of this primer is to introduce the main ideas of Mara and its funtionality in a relatively self-contained manner.

Mara is a multidimensional Godunov-type code that solves the equations of compressible gasdynamics and magnetohydrodynamics (MHD). (add link in footnotes or cite: https://github.com/jzrake/Mara2). Source code and visualization python modules available for pull request.

Let's examine the basic structure of the program:

- To begin the simulation, type:

  `./mara PROGRAM_NAME PARAMETERS`

  sample command:

  `./mara convect N=100 tfinal=100 cpi=0.25`

  in the command line; `PROGRAM_NAME` is the key name corresponding to key-value pairs listed in Mara.cpp inside int main block. All of the key-value pairs are specified as:

  `programs["regress-1d"]=std::make_unique<Hydro1DTestProgram>()`

  Please note `programs[...]` is a c++ `std::map` containing unique pointers to Mara programs, more on that later. Sample input include: `regress-1d`, `raid`, `pinch`, `kinetic`, `convect`.

- Run-time parameters can be specified in typical unix command format. For example, to specify the number of gridpoints used in a hydrodynamic simulation, `PARAMETERS` can include `N=1000`, representing 1000 gridpoints in the mesh, or `tfinal=100`, thereby setting the final physical time to 100s.

- Simulation results will be regularly stored in checkpoints in `./Mara/data` subdirectory. The interval of saving down checkpoints can be specified in the run-time parameters described above; `cpi=xxx`. xxx being the physicaltime interval you want between written checkpoints.

1.2 Architecture: in the context of ThermalConvection.cpp

- Mara files are modular, with the physics related files containing numerical scheme implementations, Mesh implementation of different geometries (Cartesian/Spherical), stored in `./Mara/src/Problems/`.
- Coded-up sample problems are stored in `./Mara/src/`
- Convenience or helper functions and general housekeeping code that ensures Mara runs smoothly are written in Cow; source code located in `./Mara/Cow/src/`, more on that in the next section
- The command-line arguments are passed into mara.cpp main in the usual `int argc, const char* argv[]` manner
- As mentioned above, the `program` map parses the command-line, looking for valid program names
- If valid program name found, it runs the program through the run function specified in the individual program files. Example: `int ThermalConvectionProgram::run (int argc, const char* argv[])` in ThermalConvection.cpp.
- Problem Customization Settings:
  - Code contained within program files are particular configurations of a problem. For example, looking at ThermalConvection.cpp, one can specify a particular `sourceTermsFunction`, which is a lambda which takes 3 spatial coordinates and an empty 5 component array (1 density, 3 momenta, 1 energy) containing the source terms at each specific cell gridpoint. With source term meaning the RHS of the set of Euler eq:

    $$\frac{\partial(\textbf{density})}{\partial t} + \nabla \cdot \textbf{flux} = \textbf{source}$$

    This allows flexibility to customize simulations on any sources.
  - Another important feature is the `initialData` lambda: the main use is to specify initial conditions. Notice `const double rho = 1.0;`. The initial profile of the gas is set to 1.0, and uniform across all the grid cells. We can also, customize `rho` to be any parametrizable function. A good example would be the hydrostatic profile.
- The problem file also assigns a few important pointers to important objects which are later used in the program (specifics of each are discussed in the next section). Generally speaking, pointers to mesh operators (`mo`), field operators (`fo`) and mesh data (`md`) (very important this one!) will be assigned.
- Mesh Customization Settings:

  - Perhaps the core of the simulation is the mesh geometry (`mg`)–whether cartesian or spherical is specified. `mg->setCellsShape` is used to set number of meshpoints in each direction while `mg->setLowerUpper` is used to set range in each dimension respectively. Pointer to mesh geometry, for example, can be changed to `auto mg = std::shared_ptr<MeshGeometry> (new SphericalMeshGeometry)` to accomodate different geometries.
  - Boundary cells are taken into account through `bs = Shape {{ 2, 0, 0, 0, 0 }}` (`bs` for boundary shape) for example. This array format will be discussed more in detail. But the main idea is that there are 2 "ghost cells" in the 1st-direction (x in cartesian) and none for the other 2nd and 3rd direction. Ignore the 4th and 5th 0's for now–they will mean something later.
  - Once boundary cells are set, we can take care of the rest of the boundary conditions through the problem-specific `apply` method. Each problem contains it's own inherited BC class,
    e.g. `class ThermalConvectionBoundaryCondition:public BoundaryCondition`.
    The apply method can override it's parent's `apply` and can customize BC specifications tailored to the problem. E.g. reflecting/outflow/inflow... boundary conditions at either endpoint (or write one yourself!)
  - This boundary shape array, along with the appropriate number of conserved variables and the geometry are passed into this **mesh data constructor**: `auto md = std::make_shared<MeshData> (mg->cellsShape(), bs, cl->getNumConserved());`, which initializes the mesh on which numerical simulations will be done. It's important for the program to know how many conserved variables there are to allocate the right amount of memory for the mesh/primitive arrays.

- scheduler and logger are pointers passed into `maraMainLoop` in Mara.cpp. Scheduler arranges tasks to be performed in sequence and logger cout's essential output when simulation run.
- The writer then writes the data before returning maraMainLoop.
- Logger is for performing `std::cout` into our console
- The program file ends with: `return maraMainLoop (status, timestep, condition, advance, *scheduler, *logger)`, the first 4 arguments are callback lambdas defined above in the program and the last two are dereferenced scheduler and logger pointers which allow allocated memory to persist in between loop.
- `maraMainLoop` loops over time and advances the simulation forward in time according to `advance` lambda and exits when `condition` lambda becomes false.
- A common misconception is to think that program files do all the heavy-lifting computations in one file. Instead, the iterations forward in time is

done when the program file returns to Mara.cpp, where computations are advanced until simulations are completed. The time-step integrator can be specified in the individual program files through `ss=std::make_shared<MethodOfLinesTVD>` member functions. (`ss` short for solution scheme). Important `ss` member function include `ss->setBoundaryCondition(bc) and ss->setRungeKuttaOrder(2)` for starters. And a few important housekeeping member functions to ensure the correct mesh and field operator are specified `ss->setMeshOperator(mo);` `ss->setFieldOperator(fo)`.

2 Review on Cow

2.1 Array

- `using Shape = std::array<int, 5>;`
- `using Index = std::array<int, 5>;`
- `using using Array = Cow::Array`
- Region
- Range

2.2 Variant

This is a section

3 Mara

This is a section

3.1 Mara

This is a section

3.2 MeshData

This is a section

3.3 BoundaryConditions

This is a section

3.4 ConservationLaws

This is a section

3.5 FieldOperator

- Field operators on the other hand perform operators on the primitive/conserved variables. This module includes convenience functions like `getConservationLaw()` which returns a shared pointer to the conservation law used. Other convenience functions include converting primitive variable arrays into conserved variables via `void generateConserved (Array::Reference P, Array::Reference U) const` which takes an array containing primitives P and puts the conserved variables into an empty array U. Courant timesteps are computed here as well. Perhaps an important diagnostic tool for sanity-checking the evolution of density-like thermodynamic quantities (i.e. entropy or total energy $e_T = e + \frac{1\rho v^2}{2}$) or momentum density in 3 dimensions $\rho\vec{v}$ in the Euler equations, is the `std::vector<double> volumeIntegratedDiagnostics (Array::Reference P, Array::Reference V) const` function. This returns volume integral (not volume average) of named diagnostics which are defaulted to 'total_energy' or 'mass' but one can customize this to ensure code is giving sensible results, *P and V contain primitives and cell volume for individual grid cells respectively throughout the whole mesh.*

3.6 MeshOperator

- Mesh operators are operators which Mara performs on the mesh grid itself (just spatial dimensions, doesn't touch primitive/conserved data in each cell). Common member functions include `divergence` and curl. One important one is `measure`, which returns the 1, 2, or 3 dimensional measure of the given mesh locations: length of edges, area of faces, or volume of cells. **Can talk about generate or generate Sourceterms HERE!!!!!!!!!!**

3.7 Scheduler

- scheduler and logger are pointers that are passed through back into the maraMainLoop in the Mara.cpp file. Main function of scheduler is to put together a list of tasks to be completed at certain points when the code runs. They could include writing checkpoints or saving down time series data to ensure code is running properly and thereby optimizing data retrieval. An example of scheduler is `scheduler->schedule (taskTimeSeries, TaskScheduler::Recurrence (user["tsi"]), "time_series");`. `taskTimeSeries` is a callback lambda which the function takes as input, thereby telling the scheduler what function to use to perform a specific task at certain periods of time during runtime. What determines *when* the task is performed depends solely on the **recurrence rule**. This literally means, when will the task/function recurr in the program via some hard and fast rule. In this case, `TaskScheduler::Recurrence (user["tsi"])` does just that using `user["tsi"]` which is specified at runtime. **SPLIT LOGGER INTO SEPARATE PARAGRAPH**

3.8 Problems

This is a section

3.9 RiemannSolvers

This is a section

3.10 SolutionSchemes

This is a section