

Primer on Mara Computational Library

Andrew Liu Jonathan Zrake

May 19, 2018

1 Introduction and Software Architecture

1.1 Background

The objective of this primer is to introduce the main ideas of Mara and its functionality in a relatively self-contained manner. The goal is to make Mara's structure tractable and to point out the most important modules for basic computation.

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 physical time 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:

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

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 its 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)`.

Now that we've covered the basic flow of the program, we can delve deeper into the details of Cow and Mara modules and its syntax.

2 Review on Cow

2.1 Array

There are a few important Cow classes that will be reviewed in this section. Under each section, important 'using statements' are also listed since they sometimes could be hard to remember:

- `Cow::Shape3D`
 - `using Shape=std::array<int,5>;`
 - One has to first distinguish between a 5 component array and a 5 dimension array. 5 component array is a 1 dimensional array (vector) that looks like `{1,2,3,4,5}`; on the other hand, a 5 dimension array is something like `A[] [] [] [] []`
 - `Shape3D` is a helper class to make sure 3 component arrays return 5 component arrays
 - `class Shape3D` is a specific case of the Shape object we discussed. `Shape3D` contain methods that focus on the first 3 components of a Shape array, which has 5 components. `Shape3D` literally focuses on axes 0, 1 and 2 which in Cartesian coordinates is x, y and z.
 - Important features of `Shape3D` include the ability to initialize a 5 component array that represents a 5 dimension array via only passing in the first 3 numbers of a Shape array, which represents the gridpoints of the first 3 dimensions. This is given by the method: `Shape3D::Shape3D (int n1, int n2, int n3)`
 - If you don't want to be lazy, you can put in the whole 5 component Shape array, which is what this method allows: `Shape3D::Shape3D (Shape S)`. What this method does is the private variable `S` defined in the `class Shape3D` is filled with the individual components of the Shape `S` array passed.

- `void Shape3D::deploy(std::function<void(int i,int j,int k)> function) const` is a useful way of filling an array with `function(i,j,k)` without looping over 3-spatial axes. Shorthand for writing a triple for-loop in C++ and useful when trying to evaluate a function over a 3D grid.
- `Cow::Array`
 - using `using Array=Cow::Array`
 - `Array` constructors can take in `Shape`
 - `Array` constructors can take 5 numbers: `{n1, n2, n3, n4, n5}`, each representing the number of points in each dimension of the 5 dimensional array which will be created. `Cow` then computes `n1*n2*n3*n4*n5*sizeof(double)` and allocates that in memory.
 - Important note: if `Array {{100,1,1,5,1}}`, then `Array.size()`=`100*1*1*5*1=500`, which is the total number of cells allocated in memory.
 - Some useful short-hand: `P(i,j,k,q,1)` will get direct access to the memory of a 5-dimensional (nested) array at `P[i][j][k][q][1]`
 - if `P(i,j,k)` then get direct access to `P(i,j,k,0,0)`
 - The notation to initialize `Cow::Array` is `{{n1,n2,n3,n4,n5}}`, where `n1,n2,n3` are dimensions (of a grid usually) in direction 1,2 and 3 of your system. `n4` is usually the number of primitives/conserved variables your system has. In classical hydrodynamics, `n4=5`. `n5` is the so-called rank, which tells us for each primitive variable at a point on a mesh, how many components that primitive variable has. For Newtonian hydrodynamics rank is 1 since mass density/ total energy density is a scalar function and momentum1 density is also a scalar since we split each direction up into 3 equations, so `n5=1`. One can contrast that with EM fields where rank=3 since 3 components of EM field is used to describe one primitive locally at one meshpoint.
 - Now to apply this idea of a 5 dimensional array, as mentioned above, since `numComponent` in our hydrodynamic case in `ThermalConvection.cpp` is 5. If location `i,j,k` are specified within the grid, then `P(i,j,k,0-4,0)` contain the 5 primitive variables and is a 5 component 1D array (or a vector) corresponding to the primitive in at cell:`i,j,k`.
 - `P[some_Region]` is how one references/retrieves information from an array, where region contains strided region (discussed below).
 - `begin()` and `end()` methods works for `Array`, and gives iterator taking into account strides since region is specified.
- `Cow::Region`
 - using `Index=std::array<int,5>;`

- Empty region has no stride: lower=0, upper=0, stride=0. lower and upper are respectively the lower and upper index of a region. It can be positive or negative. Positive means it's an absolute region, negative or zero means index from that end (python-like notation when dealing with arrays).
 - A natural region has stride 1: lower=0, upper=0, stride=1.
 - A region literally describes a region with lower,upper,stride within an array, that's why this concept is natural. It is also important to consider stride, lower and upper because of how memory is allocated.
 - One useful method in Array class is Array[Region] which returns a reference to the region you specified
- Cow::Range
 - Like a 1-dimensional analog of Region where only concerned with 1 axis within an array

3 Mara

Next we will discuss some useful features of Mara that will help with everyday computation:

3.1 Mara

We've discussed some important features of the Mara.cpp file, now let's examine the header file which contains all of the useful abstract class definitions and virtual functions ready to be overridden.

The header file also contains detailed commenting regarding the uses of each class and method. The goal of virtual functions are used here is because we want to establish parent-child relationship to allow for overloading/overriding of methods to ensure maximum flexibility. Some important definition 'using statements' will also be included. We'll examine a few important abstract parent classes here:

- ConservationLaw
 - using StateArray = std::array<double, MARA_NUM_FIELDS>;
 - ConservationLaw::State is defined here, which is a struct containing arrays like primitive, conserved, flux, eigenvalues. This class will be inherited/expanded by the ConservationLaw.cpp.
 - ConservationLaw::Request is also defined here which contains simple data retrieval (get methods) such as getPrimitive or getConserved from a ConservationLaw::State; these are fleshed out in ConservationLaw.cpp
- BoundaryCondition

```

- enum class MeshLocation { vert, edge, face, cell };
- enum class MeshBoundary { left, right };
- The bread and butter of boundary condition is the virtual void
  apply (Cow::Array& A, MeshLocation location, MeshBoundary
  boundary, int axis, int numGuard) const = 0;.

```

3.2 MeshData

This is a section

3.3 BoundaryConditions

- advance takes meshdata and dt from ss
- ss->setboundarycondition sets bc
- each set of the advance process, boundary condition is applied through meshdata applyboundarycondition using applsimple
- applsimple uses apply
- apply method in TC overrides the virtual method in the parent class

3.4 ConservationLaws

The `ConservationLaw.h` file contains different classes tailored to different conservation laws examined. A commonly used one is `class NewtonianHydro : public ConservationLaw` which you can tell inherits publicly from the parent `class ConservationLaw` from `Mara.h`.

As you can see, this module provides specific implementation of the virtual methods defined in Mara's header file. Examples include `int getNumConserved() const override;`, a function to get the number of conserved variables which would vary from 5 to 8 when one goes from Newtonian hydrodynamics to MHD.

- Moving on to `ConservationLaw.cpp`, we see right off the bat, primitive names such as `#define RHO 0` are set to an array index to facilitate data retrieval later. This is given a fancy name called `fieldIndex`. Be on the lookout for this in functions like `std::string NewtonianHydro::getPrimitiveName(int fieldIndex) const`.
- Remember `Conservation::State` is defined in `Mara.h`, where it is a struct containing primitive/conserved arrays? It is used for example in `ConservationLaw::State NewtonianHydro::fromPrimitive(const Request& request, const double* P) const`.
- The request (see `ConservationLaw::Request` defined in `Mara.h`) and a primitive array `P` are passed into the function.

- A **State S** is generated, and populated with primitives and conserved variables contained within the input array P. First off, we know mass density, 3 components of velocity and Pressure: ρ, v_1, v_2, v_3, P comprise the primitive vector (5 component vector). Each one is given an index defined at the top of this file. This function takes these variables and generates conserved ones: $\rho \rightarrow \rho, v_1 \rightarrow \rho v_1$. E.g. in the `.cpp` file, the **State S**'s momentum flux in direction 1: `S.U[S11] = P[RH0] * P[V11]`. Eigenvalues of the characteristics are also stored in this state. For Newtonian hydro, we know there are 3 distinct ones $\{v - c, v, v + c\}$, where c is the speed of sound.
- An important function here are the diagnostic tools. One can use them to plot time series of mass, total energy to check the code's consistency. One example is the `std::vector<double> NewtonianHydro::makeDiagnostics (const State& state) const`, which you will need to pass a `ConservationLaw::State` in. The function creates an empty vector (of any size you want since it can contain as many diagnostics as you want), and it populates this vector with conserved variables.

The function is instrumental when trying to compute a time series of a volume-integrated variable. One thing of note is that this function represents the conserved variables within a particular cell, since volume-integrated quantities within a cell (defined as `celld` in `FieldOperator.cpp`):

$$\text{diagnostics} = \int_{\text{cell}} \begin{bmatrix} \rho \\ \rho v_1 \\ \rho v_2 \\ \rho v_3 \\ \frac{1}{2} \rho v^2 + \frac{P}{\gamma - 1} \end{bmatrix} dV$$

3.5 FieldOperator

- **FieldOperator** methods operate on the primitive/conserved variables. This module includes simple 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.
- One of the more important functions is `getCourantTimeStep()` and is computed here as well. `generateConserved()`, `getCourantTimeStep()`
- As mentioned in the previous subsection. An important diagnostic tool for sanity-checking the evolution of density-like thermodynamic quantities (i.e. entropy or total energy $e_T = e + \frac{1}{2} \rho v^2$, e is the internal energy per unit volume) 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.**

- `auto Treg = P.getRegion().withStride (3, law->getNumConserved());`
`Treg` is array `P`'s region taking into account the stride. Recall that `P` is a 5 dimensional array, `withStride` here takes axes 3 (the number of components in a primitive vector) and sets that number to the number of conserved variables. This is to ensure we're getting the right primitive values.
- `Preg` is just the `P` array after taking into account the strided region of consideration.
- `pit` and `vit` represent pointer/iterators to these primitive vectors and cell volume of a particular cell; as the iterator is incremented by 1, we move to the next cell.
- `celld = law->makeDiagnostics (state)` is what we described in the previous section and is a 5 component vector containing conserved variables at a particular cell location.
- This part: `diagnostics[n] += celld[n] * Vol` is looped over all the cells, hence summing the integrated to find the volume-integrated conserved variable of the whole mesh:

$$\text{Total diagnostic variable returned} = \sum_{\text{over mesh}} \text{diagnostics}$$

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 recur 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