AMR Godunov Unsplit Algorithm and Implementation

P. Colella
D. T. Graves
T. J. Ligocki
D. F. Martin
B. Van Straalen

Applied Numerical Algorithms Group
NERSC Division
Lawrence Berkeley National Laboratory
Berkeley, CA

August 8, 2003

Contents

Chapter 1

Algorithm

This section describes the numerical method for integrating systems of conservation laws (e.g., the Euler equations of gas dynamics) on an AMR grid hierarchy. This is done using an unsplit, second-order Godunov method.

1.1 Notation

Most of the notation used here is introduced in the Chombo design document [CGL+00]. The main exception to that is a notation using | symbols. For computations at cell centers the notation

$$CC = A \mid B \mid C$$

means that the 3-point formula A is used for CC if all cell centered values it uses are available, the 2-point formula B is used if current cell borders the high side of the physical domain (i.e., no high side value), and the 2-point formula C is used if current cell borders the low side of the physical domain (i.e., no low side value). For computations at face centers the analogous notation

$$FC = A \mid B \mid C$$

means that the 2-point formula A is used for FC if all cell centered values it uses are available, the 1-point formula B is used if current face coincides with the high side of the physical domain (i.e., no high side value), and the 1-point formula C is used if current face coincided with the low side of the physical domain (i.e., no low side value).

1.2 Multidimensional higher-order Godunov method

The methods developed here have their origins in Colella [Col90] and Saltzman [Sal94]. We are solving a hyperbolic system of equations of the form

$$\frac{\partial U}{\partial t} + \sum_{d=0}^{\mathbf{D}-1} \frac{\partial F^d}{\partial x^d} = S$$

We also assume there may be a change of variables $W = W(U)$ ($W \equiv$ "primitive variables") that can be applied to simplify the calculation of the characteristic structure of the equations. This leads to a similar system of equations in W.

$$\frac{\partial W}{\partial t} + \sum_{d=0}^{\mathbf{D}-1} A^d(W)\ \frac{\partial W^d}{\partial x^d} = S'$$
$$A^d = \nabla_U W \cdot \nabla_U F^d \cdot \nabla_W U$$
$$S' = \nabla_U W \cdot S$$

Note, this system is not in conservation form as the primitive variables, in general, are not conserved quantities.

1.2.1 Outline

Given $U_{\boldsymbol{i}}^n$ and $S_{\boldsymbol{i}}^n$, we want to compute a second-order accurate estimate of the fluxes: $F^{n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} \approx F^d(\boldsymbol{x}_0 + (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d)h, t^n + \frac{1}{2}\Delta t)$. The transformations $\nabla_U W$ and $\nabla_W U$ are functions of both space and time. We shall leave the precise centering of these transformations vague as this will be application dependent. In outline, the method is given as follows.

1. Transform to primitive variables, and compute slopes (the definition of $\Delta^d W_{\boldsymbol{i}}$ is given in section 1.2.2):

   $$\text{Given } W_{\boldsymbol{i}}^n = W(U_{\boldsymbol{i}}^n), \text{ compute } \Delta^d W_{\boldsymbol{i}}, \text{ for } 0 \le d < \mathbf{D}$$

2. Compute the effect of the normal derivative terms and the source term on the extrapolation in space and time from cell centers to faces. For $0 \le d < \mathbf{D}$,

   $$W_{\boldsymbol{i},\pm,d} = W_{\boldsymbol{i}}^n + \frac{1}{2}(\pm I - \frac{\Delta t}{h} A_{\boldsymbol{i}}^d) P_\pm(\Delta^d W_{\boldsymbol{i}}) \tag{1.1}$$
   $$A_{\boldsymbol{i}}^d = A^d(W_{\boldsymbol{i}})$$
   $$P_\pm(W) = \sum_{\pm\lambda_k > 0} (l_k \cdot W) r_k$$
   $$W_{\boldsymbol{i},\pm,d} = W_{\boldsymbol{i},\pm,d} + \frac{\Delta t}{2} \nabla_U W \cdot S_{\boldsymbol{i}}^n \tag{1.2}$$

where λ_k are eigenvalues of $A^d_{\boldsymbol{i}}$, and l_k and r_k are the corresponding left and right eigenvectors.

3. Compute estimates of F^d suitable for computing 1D flux derivatives $\frac{\partial F^d}{\partial x^d}$ using a Riemann solver for the interior, R, and for the boundary, R_B. Here, and in what follows, $\nabla_U W$ need only be first-order accurate, e.g., differ from the value at $U^n_{\boldsymbol{i}}$ by $O(h)$.

$$\begin{aligned} F^{\mathsf{1D}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} = & R(W_{\boldsymbol{i},+,d}, W_{\boldsymbol{i}+\boldsymbol{e}^d,-,d}, d) \\ & \mid R_B(W_{\boldsymbol{i},+,d}, (\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d)h, d) \\ & \mid R_B(W_{\boldsymbol{i}+\boldsymbol{e}^d,-,d}, (\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d)h, d) \end{aligned} \tag{1.3}$$

4. In 3D compute corrections to $W_{\boldsymbol{i},\pm,d}$ corresponding to one set of transverse derivatives appropriate to obtain $(1,1,1)$ diagonal coupling. In 2D skip this step.

$$W_{\boldsymbol{i},\pm,d_1,d_2} = W_{\boldsymbol{i},\pm,d_1} - \frac{\Delta t}{3h}\nabla_U W \cdot (F^{\mathsf{1D}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^{d_2}} - F^{\mathsf{1D}}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^{d_2}}) \tag{1.4}$$

5. In 3D compute fluxes corresponding to corrections made in the previous step. In 2D skip this step.

$$\begin{aligned} F_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^{d_1},d_2} = & R(W_{\boldsymbol{i},+,d_1,d_2}, W_{\boldsymbol{i}+\boldsymbol{e}^{d_1},-,d_1,d_2}, d_1) \\ & \mid R_B(W_{\boldsymbol{i},+,d_1,d_2}, (\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^{d_1})h, d_1) \\ & \mid R_B(W_{\boldsymbol{i}+\boldsymbol{e}^{d_1},-,d_1,d_2}, (\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^{d_1})h, d_1) \\ & d_1 \neq d_2,\ 0 \leq d_1, d_2 < \mathbf{D} \end{aligned} \tag{1.5}$$

6. Compute final corrections to $W_{\boldsymbol{i},\pm,d}$ due to the final transverse derivatives.

$$\text{2D:} \quad W^{n+\frac{1}{2}}_{\boldsymbol{i},\pm,d} = W_{\boldsymbol{i},\pm,d} - \frac{\Delta t}{2h}\nabla_U W \cdot (F^{\mathsf{1D}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^{d_1}} - F^{\mathsf{1D}}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^{d_1}}) \tag{1.6}$$
$$d \neq d_1,\ 0 \leq d, d_1 < \mathbf{D}$$

$$\begin{aligned} \text{3D:} \quad W^{n+\frac{1}{2}}_{\boldsymbol{i},\pm,d} = W_{\boldsymbol{i},\pm,d} & - \frac{\Delta t}{2h}\nabla_U W \cdot (F_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^{d_1},d_2} - F_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^{d_1},d_2}) \\ & - \frac{\Delta t}{2h}\nabla_U W \cdot (F_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^{d_2},d_1} - F_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^{d_2},d_1}) \end{aligned} \tag{1.7}$$
$$d \neq d_1 \neq d_2,\ 0 \leq d, d_1, d_2 < \mathbf{D}$$

7. Compute final estimate of fluxes.

$$\begin{aligned} F^{n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} = & R(W^{n+\frac{1}{2}}_{\boldsymbol{i},+,d}, W^{n+\frac{1}{2}}_{\boldsymbol{i}+\boldsymbol{e}^d,-,d}, d) \\ & \mid R_B(W^{n+\frac{1}{2}}_{\boldsymbol{i},+,d}, (\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d)h, d) \\ & \mid R_B(W^{n+\frac{1}{2}}_{\boldsymbol{i}+\boldsymbol{e}^d,-,d}, (\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d)h, d) \end{aligned} \tag{1.8}$$

8. Update the solution using the divergence of the fluxes.

$$U^{n+1}_{\boldsymbol{i}} = U^n_{\boldsymbol{i}} - \frac{\Delta t}{h}\sum_{d=0}^{\mathbf{D}-1}(F^{n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} - F^{n+\frac{1}{2}}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^d}) \tag{1.9}$$

1.2.2 Slope Calculation

We will use the 4th order slope calculation in Colella and Glaz [CG85] combined with characteristic limiting.

$$\begin{aligned} \Delta^d W_{\boldsymbol{i}} &= \zeta_{\boldsymbol{i}}\ \delta^{vL}(\Delta^d_4 W_{\boldsymbol{i}}, \Delta^d_- W_{\boldsymbol{i}}, \Delta^d_+ W_{\boldsymbol{i}}) \mid \Delta^d_2 W_{\boldsymbol{i}} \mid \Delta^d_2 W_{\boldsymbol{i}} \\ \Delta^d_4 W_{\boldsymbol{i}} &= \frac{2}{3}((W - \frac{1}{4}\Delta^d_2 W)_{\boldsymbol{i}+\boldsymbol{e}^d} - (W + \frac{1}{4}\Delta^d_2 W)_{\boldsymbol{i}-\boldsymbol{e}^d}) \\ \Delta^d_2 W_{\boldsymbol{i}} &= \delta^{vL}(\widetilde{\Delta}^d_2 W_{\boldsymbol{i}}, \Delta^d_- W_{\boldsymbol{i}}, \Delta^d_+ W_{\boldsymbol{i}}) \mid \Delta^d_- W_{\boldsymbol{i}} \mid \Delta^d_+ W_{\boldsymbol{i}} \\ \widetilde{\Delta}^d_2 W_{\boldsymbol{i}} &= \frac{1}{2}(W^n_{\boldsymbol{i}+\boldsymbol{e}^d} - W^n_{\boldsymbol{i}-\boldsymbol{e}^d}) \\ \Delta^d_- W_{\boldsymbol{i}} &= W^n_{\boldsymbol{i}} - W^n_{\boldsymbol{i}-\boldsymbol{e}^d}\ ,\ \Delta^d_+ W_{\boldsymbol{i}} = W^n_{\boldsymbol{i}+\boldsymbol{e}^d} - W^n_{\boldsymbol{i}} \end{aligned}$$

At domain boundaries, $\Delta^d_- W_{\boldsymbol{i}}$ and $\Delta^d_+ W_{\boldsymbol{i}}$ may be overwritten by the application to provide application dependent slopes at the boundaries (see section 2.3.4). There are two versions of the van Leer limiter $\delta^{vL}(\delta W_C, \delta W_L, \delta W_R)$ that are commonly used. One is to apply a limiter to the differences in characteristic variables.

1. Compute expansion of one-sided and centered differences in characteristic variables.

$$\begin{aligned} \alpha^k_C &= l^k \cdot \delta W_C \\ \alpha^k_L &= l^k \cdot \delta W_L \\ \alpha^k_R &= l^k \cdot \delta W_R \end{aligned}$$

2. Apply van Leer limiter

$$\alpha^k = \begin{cases} min(|\,\alpha^k_C\,|,\, 2|\,\alpha^k_L\,|,\, 2|\,\alpha^k_R\,|) & \text{if } \alpha^k_L \cdot \alpha^k_R > 0 \\ 0 & \text{otherwise} \end{cases}$$

3. $\delta^{vL} = \sum_k \alpha^k r^k$

Here, $l^k = l^k(W_{\boldsymbol{i}}^n)$ and $r^k = r^k(W_{\boldsymbol{i}}^n)$.

For a variety of problems, it suffices to apply the van Leer limiter componentwise to the differences. Formally, this can be obtain from the more general case above by taking the matrices of left and right eigenvectors to be the identity.

Finally, we give the algorithm for computing the flattening coefficient $\zeta_{\boldsymbol{i}}$. WE assume that there is a quantity corresponding to the pressure in gas dynamics (denoted here as p) which can act as a steepness indicator, and a quantity corresponding to the bulk modulus (denoted here as K, given as γp in a gas), that can be used to non-dimensionalize differences in p.

$$\begin{aligned}
\zeta_{\boldsymbol{i}} &= \begin{cases} \min\limits_{0\le d<\mathbf{D}} \zeta_{\boldsymbol{i}}^d & \text{if } \sum_{d=0}^{\mathbf{D}-1} \Delta_1^d u_{\boldsymbol{i}}^d < 0 \\ 1 & \text{otherwise} \end{cases} \\
\zeta_{\boldsymbol{i}}^d &= min_3(\widetilde{\zeta}^d, d)_{\boldsymbol{i}} \\
\widetilde{\zeta}_{\boldsymbol{i}}^d &= \eta(\Delta_1^d p_{\boldsymbol{i}},\, \Delta_2^d p_{\boldsymbol{i}},\, min_3(K,d)_{\boldsymbol{i}}) \\
\Delta_1^d p_{\boldsymbol{i}} &= \frac{1}{2}(p_{\boldsymbol{i}+\boldsymbol{e}^d} - p_{\boldsymbol{i}-\boldsymbol{e}^d}) \mid p_{\boldsymbol{i}} - p_{\boldsymbol{i}-\boldsymbol{e}^d} \mid p_{\boldsymbol{i}+\boldsymbol{e}^d} - p_{\boldsymbol{i}} \\
\Delta_2^d p_{\boldsymbol{i}} &= (\Delta_1^d p_{\boldsymbol{i}+\boldsymbol{e}^d} + \Delta_1^d p_{\boldsymbol{i}-\boldsymbol{e}^d}) \mid 2\Delta_1^d p_{\boldsymbol{i}} \mid 2\Delta_1^d p_{\boldsymbol{i}}
\end{aligned} \tag{1.10}$$

The functions min_3 and η are given below.

$$min_3(K,d)_{\boldsymbol{i}} = min(K_{\boldsymbol{i}+\boldsymbol{e}^d}, K_{\boldsymbol{i}}, K_{\boldsymbol{i}-\boldsymbol{e}^d}) \mid min(K_{\boldsymbol{i}}, K_{\boldsymbol{i}-\boldsymbol{e}^d}) \mid min(K_{\boldsymbol{i}+\boldsymbol{e}^d}, K_{\boldsymbol{i}})$$

$$\eta(\delta p_1, \delta p_2, p_0) = \begin{cases} 0 & \text{if } \frac{|\delta p_1|}{p_0} > d \text{ and } \frac{|\delta p_1|}{|\delta p_2|} > r_1 \\ 1 - \frac{\frac{|\delta p_1|}{|\delta p_2|} - r_0}{r_1 - r_0} & \text{if } \frac{|\delta p_1|}{p_0} > d \text{ and } r_1 \ge \frac{|\delta p_1|}{|\delta p_2|} > r_0 \\ 1 & \text{otherwise} \end{cases}$$

$$r_0 = 0.75,\ r_1 = 0.85,\ d = 0.33$$

1.3 Recursive AMR Update

We extend this method to an adaptive mesh hierarchy using the Berger-Oliger algorithm. We define

$$\{U^l\}_{l=0}^{l_{max}}, U^l : \Omega^l \to \mathbb{R}^m$$

$U^l = U^l(t^l)$. Here $\{t^l\}$ are a collection of discrete times that satisfy the temporal analogue of proper nesting. $\{t^l\} = \{t^{l-1} + k\Delta t^l : 0 \le k < n_{ref}^l\}$ The algorithm in [BC89] for advancing the solution in time is given in pseudo-code in figure 1.1. The discrete fluxes $\vec{F}$ are computed by using piecewise linear interpolation to define an extended solution on

$$\tilde{\Omega} = \mathcal{G}(\Omega^l, p) \cap \Gamma^l\ ,\ \tilde{U} : \tilde{\Omega} \to \mathbb{R}^m$$

$$\tilde{U}_{\boldsymbol{i}} = \begin{cases} U^l_{\boldsymbol{i}}(t^l) & \text{for } \boldsymbol{i} \in \Omega^l \\ I_{pwl}((1-\alpha)U^{l-1}(t^{l-1}) + \alpha\ U^{l-1}(t^{l-1} + \Delta t^{l-1}))_{\boldsymbol{i}} & \text{otherwise} \end{cases}$$

$$\alpha = \frac{t^l - t^{l-1}}{\Delta t^{l-1}}$$

and then computing fluxes for the advance as outlined in Section 1.2.

---

procedure advance (l)
$U^l(t^l + \Delta t^l) = U^l(t^l) - \Delta t D\vec{F}^l$ on Ω^l
if $l < l_{max}$
    $\delta F_d^{l+1} = -F_d^l$ on $\zeta_{+,d}^{l+1} \cup \zeta_{-,d}^{l+1}, d = 0, ..., \mathbf{D} - 1$
end if
if $l > 0$
    $\delta F_d^l := \frac{1}{n_{ref}^{l-1}} < F_d^l >$ on $\zeta_{+,d}^l \cup \zeta_{-,d}^l, d = 0, ..., \mathbf{D} - 1$
end if
for $q = 0, ..., n_{ref}^l - 1$
    advance$(l+1)$
end for
$U^l(t^l + \Delta t^l) = Average(U^{l+1}(t^l + \Delta t^l), n_{ref}^l)$ on $\mathcal{C}_{n_{ref}^l}(\Omega^{l+1})$
$U^l(t^l + \Delta t^l) := U^l(t^l + \Delta t^l) - \Delta t^l D_R(\delta F^{l+1})$
$t^l := t^l + \Delta t^l$
$n_{step}^l := n_{step}^l + 1$
if $(n_{step}^l = 0 \text{ mod } n_{regrid})$ and $(n_{step}^{l-1} \neq 0 \text{ mod } n_{regrid})$
    regrid(l)
end if

Figure 1.1: Pseudo-code description of the Berger-Colella AMR algorithm for hyperbolic conservation laws.

Chapter 2

Interface

2.1 Architecture Diagram

The `AMRGodunovUnsplit` code makes extensive use of the AMR time-dependent infrastructure contained in the Chombo libararies. A basic schematic of the class relationships between `Chombo` and `AMRGodunov` classes is depicted in Figure 2.1. Where appropriate, the particular implementation for a polytropic gas will be referenced (for example, the `AMRLevelPolytropicGas` and `PatchPolytropicGas` classes).

2.2 Data Design

The AMR unsplit hyperbolic (AMRGodunov) code makes extensive use of the Chombo C++ libraries. The important data structures used in this application are all provided by Chombo, as are many of the utilities which facilitate implementations of block-structured adaptive algorithms. For more detailed descriptions of these classes, see the Chombo documentation [CGL+00].

2.2.1 Global Data Structures

The important variables in the AMRGodunov code are the conserved variable vector $\vec{U}$. These variables are contained in container classes provided by `Chombo`.

2.2.1.1 Chombo Container Classes

A logically rectangular region in space is defined by a `Box`. Cell-centered data on an individual `Box` is generally contained in an `FArrayBox`.

A set of disjoint `Box`'s (generally corresponding to all the grids at a single refinement level) is defined by a `DisjointBoxLayout`. Data on a `DisjointBoxLayout` is generally contained in a `LevelData`, which is a templated container class to facilitate computations on disjoint unions of rectangles.

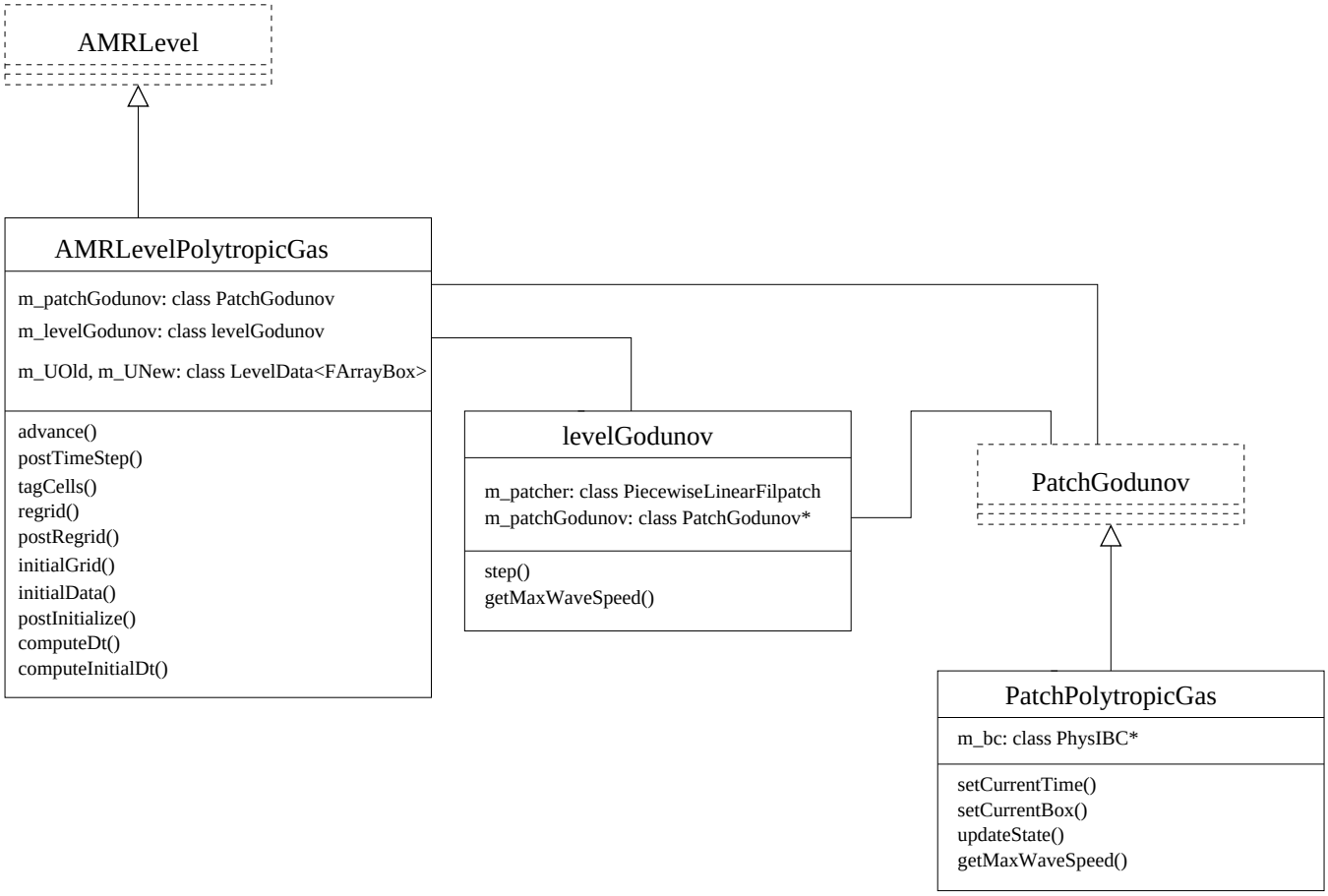

Figure 2.1: Software configuration diagram for the AMRGodunov code showing basic relationships between AMRGodunov classes and Chombo classes for the polytropic gas example.

All of these classes are further documented in the Chombo documentation [CGL+00].

2.2.1.2 Time-dependent AMR

The basic structure for the code is provided by the `Chombo AMRTimeDependent` library. The `AMR` class manages the global recursive timestep, along with initializing the hierarchy of grids and other functionality involving data on more than one level of the AMR grids.

The `AmrLevel` class manages data and functionality for a single AMR level, including the single-level advance. The `AMRLevelPolytropicGas` class is derived from the `AmrLevel` class and contains the functionality specific to the polytropic gas algorithm.

2.2.2 Internal Software Data Structures

For the polytropic gas example, the `AMRLevel`-derived class `AMRLevelPolytropicGas` contains the primary data fields necessary to update the solution on one AMR level, in particular the old- and new-time conserved variable fields ($\vec{U}(t^\ell)$ and $\vec{U}(t^\ell + \Delta t^\ell)$). Each `AMRLevelPolytropicGas` object also contains a `PatchGodunov`-derived object which contains the physics-dependent part of the algorithm; for the polytropic gas example, this is the `PatchPolytropicGas` class, which contains the functionality to perform updates on a single logically rectangular patch (which is dependent on the physics of the problem being solved). Also, every `AMRLevelPolytropicGas` also contains a `levelGodunov` class as a member object. This `levelGodunov` member contains the functionality necessary for updating the conserved variables on a single level by one timestep, using the physics-specific `PatchGodunov`-derived class (in this case, a `PatchPolytropicGas` object).

2.3 Class Hierarchy

The principal AMRGodunovUnsplit classes follow.

- `AMRLevel<name>`, the `AMRLevel`-derived class which is driven by the AMR class. This class is application/problem dependent but is included here to document some of the data members and functions which will probably be common to many applications.
- `LevelGodunov`, a class owned by `AMRLevel<name>`. `LevelGodunov` advances the solution on a level and can exist outside the context of an AMR hierarchy. This class makes possible Richardson extrapolation for error estimation (not currently implemented).
- PatchGodunov, is a base class which encapsulates the operations required to advance a solution on a single patch/grid.
- `PhysIBC`, is a base class which encapsulates initial conditions and flux-based boundary condtions.

2.3.1 Class AMRLevel<name>

AMRLevel<name> is the AMRLevel-derived class with which the AMR class will directly interact. It's user interface is therefore constrained by the AMRLevel interface. It is also an application/problem dependent portion of the code but there are important data members and function which will probably be part of any implementation. These are documented here. The important data members of the AMRLevel<name> class are as follows:

- `LevelData<FArrayBox> m_UOld, m_UNew;`

  The conserved variables at old and new times. Both need to be kept because subcycling in time requires temporal interpolation.

- `Real m_cfl, m_dx;`

  CFL number and grid spacing for this level.

- `FineInterp m_fineInterp;`

  Interpolation operator for refining data during regridding that were previously only covered by coarser data.

- `CoarseAverage m_coarse_average;`

  This is the averaging operator which replaces data on coarser levels with the average of the data on this level where they coincide in space.

The AMRLevel<name> implementation of the AMRLevel currently does the following for each of the important interface functions:

- `Real advance()`

  This function advances the conserved variables by one time step. It calls the `LevelGodunov::step` function. The time step returned by that function is stored in a member data, `m_dtNew`.

- `void postTimeStep()`

  This function calls refluxing from the next finer level and replaces its solution with an average from the next finer level where they coincide.

- `void regrid(const Vector<Box>& a_newGrids)`

  This function changes the union of rectangles over which the data is defined. At places where the two sets of rectangles intersect, the data is copied from the previous set of rectangles. At places where there was only data from the next coarser level, piecewise linear interpolation is used to fill in the data.

- `void initialData()`

  In this function the initial state is filled by calling the initial condition member data of `m_pathGodunov`, namely `getPhysIBC()->initialize()`.

- `void computeDt()`

  This function returns the time step stored during the `advance()` call, `m_dtNew`.

- `void computeInitialDt()`

  This function calculates the time step using the maximum wavespeed returned by a `LevelGodunov::getMaxWaveSpeed` call. Given the maximum wavespeed, w, the initial time step multiplier, K, and the grid spacing at this level, h, then the initial time step, Δt, is given by:

  $$\Delta t = K\frac{h}{w}. \tag{2.1}$$

- `DisjointBoxLayout loadBalance(const Vector<Box>& a_grids)`

  Calls the Chombo load balancer to create a load balanced layout. This is returned.

2.3.2 Class `LevelGodunov`

`LevelGodunov` is a class owned by `AMRLevel<name>`. `LevelGodunov` advances the solution on a level and can exist outside the context of an AMR hierarchy. This class makes possible Richardson extrapolation for error estimation. The important functions of the public interface of `LevelGodunov` are:

- ```
  void define(const DisjointBoxLayout&  a_thisDisjointBoxLayout,
              const DisjointBoxLayout&  a_coarserDisjointBoxLayout,
              const ProblemDomain&      a_domain,
              const int&                a_refineCoarse,
              const Real&               a_dx,
              const PatchGodunov* const a_patchGodunovFactory,
              const bool&               a_hasCoarser,
              const bool&               a_hasFiner);
  ```

  Define the internal data structures. For the coarsest level, an empty DisjointBoxLayout is passed in for coaserDisjointBoxLayout.

  - `a_thisDisjointBoxLayout, a_coarserDisjointBoxLayout`: The layouts at this level and the next coarser level. For the coarsest level, an empty `DisjointBoxLayout` is passed in for `coarserDisjointBoxLayout`.
  - `a_domain`: The problem domain on this level.
  - `a_refineCoarse`: The refinement ratio between this level and the next coarser level.
  - `a_dx`: The grid spacing on this level.
  - `a_patchGodunovFactory`: The factory for the integrator which can advance each patch/grid a time step. Boundary conditions and initial conditions are also encapsulated in this object. Note: this object is its own factory.
  ```

- a_hasCoarser, a_hasFiner: This level has a coarser (or finer) level. These are used when coarser or finer levels are needed or when data which exists between levels (e.g., flux registers) is needed.

- ```
  Real step(LevelData<FArrayBox>&       a_U,
            LevelData<FArrayBox>&       a_flux[CH_SPACEDIM],
            LevelFluxRegister&          a_coarserFluxRegister,
            LevelFluxRegister&          a_finerFluxRegister,
            const LevelData<FArrayBox>& a_S,
            const LevelData<FArrayBox>& a_UCoarseOld,
            const Real&                 a_TCoarseOld,
            const LevelData<FArrayBox>& a_UCoarseNew,
            const Real&                 a_TCoarseNew,
            const Real&                 a_time,
            const Real&                 a_dt);
  ```

  Advance the solution at this timeStep for one time step.

  - a_U: The current solution at this level which will be advanced by a_dt to a_time.
  - a_flux: A SpaceDim array of face-centered LevelData<FArrayBox>s which may be used to pass face-centered data (such as fluxes) back and forth from the function.
  - a_coarserFluxRegister, a_finerFluxRegister: The flux registers between this level and the next coarser (or finer) levels.
  - a_S: Source terms from the RHS of the system of PDEs being solved/integrated. If there are no source terms a_S should be null constructed and not defined (i.e. a_S's define() function should not called).
  - a_UCoarseOld, a_TCoarseOld: The solution at the next coarser level at the old time, a_TCoarseOld.
  - a_UCoarseNew, a_TCoarseNew: The solution at the next coarser level at the new time, a_TCoarseNew.
  - a_time: The time to which to advance the current solution. This should be between a_TCoarseOld and a_TCoarseNew.
  - a_dt: The time step at this level.

- ```
  Real getMaxWaveSpeed(const LevelData<FArrayBox>& a_U);
  ```

  Return the maximum wave speed of the input a_U (the conserved variables) for purposes of limiting the time step.

2.3.3 Class PatchGodunov

The base class PatchGodunov provides a skeleton for the application-dependent pieces of a second-order unsplit Godunov method. The virtual functions are called by LevelGodunov, which manages the overall assembly of the second-order unsplit fluxes. As part of PatchGodunov, we provide some member functions (slope, flattening), that we expect to be useful across applications, but which require either virtual functions or parameter information to be supply by the user.

There are three types of grid variables that appear in the unsplit Godunov method in section (1.2): conserved variables, primitive variables, fluxes, and source terms, denoted below by U, W, F, and S, respectively. It is often convenient to have the number of primitive variables and fluxes exceed the number of conserved variables. In the case of primitive variables, redundant quantities are carried that parameterize the equation of state in order to avoid multiple calls to that the equation of state function. In the case of fluxes, it is often convenient to split the flux for some variables into multiple components, e.g., dividing the momentum flux into advective and pressure terms. The API given here provides the flexibility to support various possibilities.

The following virtual functions are part of the public interface. Some have default implementations which the user will not need to change for a variety of physical problems.

- `virtual void define(ProblemDomain& a_domain,`
  `const Real& a_dx);`

  Set the domain and grid spacing.

  - a_domain: The problem domain index space.
  - a_dx: The grid spacing.

- `virtual void setPhysIBC(PhysIBC* a_bc);`

  Set the initial and boundary condtion pointer used by the integrator for the current level.

  - a_bc: The initial and boundary condition object for the current level.

- `virtual PhysIBC* getPhysIBC() const;`

  Get (return) the initial and boundary condtion pointer used by the integrator.

- `virtual void setSlopeParameters(bool a_fourthOrderSlopes,`
  `bool a_flattening);`

  Set the parameters used for slope computations.

  - a_fourthOrderSlopes: If true then compute 4th order slopes otherwise compute 2nd order slopes.

- a_flattening: If a_fourthOrderSlopes is true and this is true then compute and apply slope flattening. It is illegal for a_fourthOrderSlopes to be false and a_flattening to be true.

- `virtual bool useFourthOrderSlopes();`

  This returns true if 4th order slopes are being computed and false if 2nd order slopes are being computed.

- `virtual bool useFlattening();`

  This returns true slope flattening is being computed and applied.

- ```
  virtual void setArtificialViscosity(bool a_useArtificialViscosity,
                                      Real a_artificialViscosity);
  ```

  Set the parameters used for artificial viscosity.

  - a_useArtificialViscosity: If true then artificial viscosity is applied.
  - a_artificialViscosity: The artificial viscosity coefficient used in applying artificial viscosity.

- `virtual bool useArtificialViscosity();`

  This returns true if artificial viscosity is being used.

- `virtual Real artificialViscosityCoefficient();`

  This returns the value of the artificial viscosity coefficient being used to apply artificial viscosity.

- `virtual PatchGodunov* new_patchGodunov() = 0;`

  This is a factory method - this object is its own factory. It returns a pointer to new PatchGodunov object with its boundary condtions defined. In addition to that the slope parameters and the artificial viscosity parameters also need to be defined.

- `virtual void setCurrentTime(const Time& a_time);`

  Set the current time.

  - a_time: The current time.

- `virtual void setCurrentBox(const Box& a_currentBox);`

  Set the box over which the conserved variables with be updated for this patch/grid.

  - a_box: The box over which the conversed variables with be updated.
  ```

- `virtual void updateState(FArrayBox&       a_U,`
  `FArrayBox        a_F[SPACEDIM],`
  `Real&            a_maxWaveSpeed,`
  `const FArrayBox& a_S,`
  `const Real&      a_dt,`
  `const Box&       a_box);`

  Update the conserved variables, return the fluxes used for this, and the maximum wave speed in the updated solution.

  - `a_U`: The conserved variables to be updated.
  - `a_F[]`: The fluxes each of the faces used of update the conserved variables (used for refluxing).
  - `a_maxWaveSpeed`: The maximum wave speed for this patch/grid.
  - `a_S`: The source terms - if there are no source terms this should be a null constructed object.
  - `a_dt`: The time step for this patch/grid.
  - `a_box`: The box to be used for the computation/update.

- `virtual Real getMaxWaveSpeed(const FArrayBox& a_U,`
  `const Box&       a_box) = 0;`

  Return the maximum wave speed on this patch/grid.

  - `a_U`: The conserved variables.
  - `a_box`: The box to be used for the computation.

- `virtual int numConserved() const = 0;`

  Return the number of conserved variables.

- `virtual Vector<string> stateNames();`

  Return the names of the variables. A default implementation is provided that puts in generic names, i.e., "variable#" where "#" ranges from 0 to `numConserved()` -1.

- `virtual int numFluxes() const = 0;`

  Return the number of flux variables. This can be greater than the number of conserved variables if addition fluxes/face-centered quantities are computed.

The following virtual functions are not part of the public interface. Many are provided by the user to implement portions of the computation specific to the physical problem being solved. Some have default implementations which, in many cases, may not change from one physical problem to the next.

- `virtual bool isDefined() const;`

  Return true if the object has been completely defined. This means that the following have all been called: `define()`, `setPhysIBC()`, `setSlopeParameters()`, `setArtificialViscosity()`, `setCurrentTime()`, and `setCurrentBox()`.

- `virtual int numPrimitives() const = 0;`

  Return the number of primitive variables. This may be greater than the number of conserved variables if derived/redundant quantities are also stored for convenience.

- `virtual int numSlopes() const = 0;`

  Return the number of slopes to be used in the calculation. Only slopes corresponding to primitive variables in the interval 0 to `numSlopes()` -1 are computed and only primitive variables in that interval are updated using the slopes.

- ```
  virtual void consToPrim(FArrayBox&       a_W,
                          const FArrayBox& a_U,
                          const Box&       a_box) = 0;
  ```

  Compute the primitive variables given the conserved variables.

  - `a_W`: The primitive variables.
  - `a_U`: The conserved variables.
  - `a_box`: The box to be used for the computation.

- ```
  virtual void computeFlattening(FArrayBox&       a_flattening,
                                 const FArrayBox& a_W,
                                 const Box&       a_box);
  ```

  Computes the flattening coefficient (1.10) and return it.

  - `a_flattening`: The flattening coefficient.
  - `a_W`: The primitive variables.
  - `a_box`: The box to be used for the computation.

- ```
  virtual void slope(FArrayBox&       a_dW,
                     const FArrayBox& a_W,
                     const FArrayBox& a_flattening,
                     const int&       a_dir,
                     const Box&       a_box);
  ```

  Compute the slope, `a_dW`, of the primitive variables, `a_W`, using the algorithm described in (1.2.2). 2nd or 4th order slopes are computed, with or within flattening depending on user supplied parameters. The function `applyLimiter()` is used to do slope limiting - it has a default implementation but this can be changed by the user.
  ```

- `a_dW`: The slopes.
- `a_W`: The primitive variables.
- `a_flattening`: The flattening coefficient.
- `a_dir`: The direction in which to compute the slopes.
- `a_box`: The box to be used for the computation.

- ```
  virtual void normalPred(FArrayBox&       a_WMinus,
                          FArrayBox&       a_WPlus,
                          const FArrayBox& a_W,
                          const FArrayBox& a_dW,
                          const Real&      a_scale,
                          const int&       a_dir,
                          const Box&       a_box) = 0;
  ```

  Extrapolate the primitive variables in the minus and plus direction using the slopes, as in (1.1).

  - `a_WMinus, a_WPlus`: The extrapolated primitive variables.
  - `a_W`: The original primitive variables.
  - `a_dW`: The primitive variable slopes.
  - `a_scale`: The scaling for the extrapolation, $\frac{\Delta t}{h}$.
  - `a_dir`: The direction in which to do the extrapolation.
  - `a_box`: The box to be used for the computation.

- ```
  virtual void incrementWithSource(FArrayBox&       a_W,
                                   const FArrayBox& a_S,
                                   const Real&      a_scale,
                                   const Box&       a_box);
  ```

  Increment the primitive variables using by the source term, as in (1.2). The default implementation doesn't change the primitive variables.

  - `a_W`: The primitive variables.
  - `a_S`: The source term.
  - `a_scale`: The scaling for the extrapolation, $\frac{\Delta t}{2}$.
  - `a_box`: The box to be used for the computation.

- ```
  virtual void riemann(FArrayBox&       a_F,
                       const FArrayBox& a_WLeft,
                       const FArrayBox& a_WRight,
                       const int        a_dir,
                       const Box&       a_box) = 0;
  ```
  ```

Given left and right states at a face, compute a Riemann problem and generate fluxes at the faces, as in (1.3, 1.5, 1.8).

  - a_F: The computed fluxes.
  - a_WLeft, a_WRight: The left and right states
  - a_dir: The direction normal.
  - a_box: The box to be used for the computation.

- ```
  virtual void updatePrim(FArrayBox&       a_WMinus,
                          FArrayBox&       a_WPlus,
                          const FArrayBox& a_F,
                          const Real       a_scale,
                          const int        a_dir,
                          const Box&       a_box) = 0;
  ```

  Given the conservative fluxes in a given direction, update the extrapolated primitive variables, as in (1.4, 1.6, 1.7).

  - a_WMinus, a_WPlus: The extrapolated primitive variables.
  - a_F: The conservative fluxes.
  - a_dW: The primitive variable slopes.
  - a_scale: The scaling for the extrapolation - this varies depending on which update is occurring.
  - a_dir: The direction in which to do the extrapolation.
  - a_box: The box to be used for the computation.

- ```
  virtual void artificialViscosity(FArrayBox&       a_F,
                                   const FArrayBox& a_U,
                                   const FArrayBox& a_divVel,
                                   const int        a_dir,
                                   const Box&       a_box);
  ```

  Update the fluxes using artificial viscosity in a given direction.

  - a_F: The conservative fluxes.
  - a_U: The conservative variables.
  - a_divVel: The divergence of the velocity in direction a_dir.
  - a_dir: The direction of the fluxes.
  - a_box: The box to be used for the computation.

- `virtual void updateCons(FArrayBox&       a_U,`
  `const FArrayBox& a_F,`
  `const Real&      a_scale,`
  `const int&       a_dir,`
  `const Box&       a_box) = 0;`

  Update the conserved variables using fluxes in a given direction, as in (1.9).

  - `a_U`: The conserved variables.
  - `a_F`: The conservative fluxes.
  - `a_scale`: The scaling for the extrapolation, $\frac{\Delta t}{h}$.
  - `a_dir`: The direction of the fluxes.
  - `a_box`: The box to be used for the computation.

- `virtual void finalUpdate(FArrayBox&       a_U,`
  `const FArrayBox& a_F,`
  `const Real&      a_scale,`
  `const int&       a_dir,`
  `const Box&       a_box) = 0;`

  Peform final update of the conserved variables `a_U`. In many cases, this function simply calls `updateCons`.

  - `a_U`: The conserved variables.
  - `a_F`: The conservative fluxes.
  - `a_scale`: The scaling for the extrapolation, $\frac{\Delta t}{h}$.
  - `a_dir`: The direction of the fluxes.
  - `a_box`: The box to be used for the computation.

- `virtual Interval velocityInterval() const = 0;`

  Return the interval of component indices of the primitive variables corresponding to the velocities. This is used by `computeFlattening()` and `divVel()`.

- `virtual int pressureIndex() const = 0;`

  Return the component index of the primitive variable corresponding to the pressure. This is used by `computeFlattening()`.

- `virtual int bulkModulusIndex() const = 0;`

  Return the component index of the primitive variable corresponding to the bulk modulus. This is used by `computeFlattening()` to normalization shock strength.

- `virtual void applyLimiter(FArrayBox&       a_dW,`
  `const FArrayBox& a_dWLeft,`
  `const FArrayBox& a_dWRight,`
  `const int        a_dir,`
  `const Box&       a_box);`

  Given the center difference and the left and right differences in a direction, apply a slope limiter to generate final slopes. A default implementation is provided which implements a van Leer limiter without characteristic analysis, see section (1.2.2). The user can change this by implementing their own version of this function. Called by the default implementation of `slope()`.

  - `a_dW`: The center difference on input and limited slopes on output.
  - `a_dWLeft, a_dWRight`: The left and right differences.
  - `a_dir`: The direction of the differences.
  - `a_box`: The box to be used for the computation.

- `virtual void divVel(FArrayBox&       a_divVel,`
  `const FArrayBox& a_W,`
  `const int        a_dir,`
  `const Box&       a_box);`

  Compute the face centered divergence of the velocity.

  - `a_divVel`: The face centered divergence of the velocity.
  - `a_W`: The primitive variables (including the velocities).
  - `a_dir`: The direction normal.
  - `a_box`: The box to be used for the computation.

2.3.4 Class `PhysIBC`

`PhysIBC` is an interface class owned and used by PatchGodunov through which a user specifies the initial and boundary of conditions of their particular problem. These boundary conditions are flux-based. `PhysIBC` contains as member data the mesh spacing (`Real m_dx`) and the domain of computation (`ProblemDomain m_domain`). This object serves as its own factory. The important user functions of `PhysIBC` are as follows.

- `virtual void define(const ProblemDomain& a_domain`
  `const Real&         a_dx);`

  Define the internals of the class.

  - `a_domain`: The problem domain.
  - `a_dx`: The grid spacing.

- `virtual PhysIBC* new_physIBC() = 0;`

  This is a factory method. It returns a new `PhysIBC` object.

- ```
  virtual void fluxBC(FArrayBox&            a_F,
                      const FArrayBox&      a_W,
                      const FArrayBox&      a_Wextrap,
                      const int&            a_dir,
                      const Side::LoHiSide& a_side,
                      const Real&           a_time) = 0;
  ```

  Return the flux boundary condtion on the boundary of the domain.

  - `a_F`: The fluxes over the box. This values in the array that correspond to the boundary faces of the domain are to be replaced with boundary values fluxes.
  - `a_Wextrap`: The extrapolated value of the primitive variables to the `a_side` of the cells in direction `a_dir`. This data is cell-centered.
  - `a_W`: The primitive variables at the start of the time step. This data is cell-centered.
  - `a_dir, a_side`: The direction normal and the side of the domain where the boundary condition fluxes are needed.
  - `a_time`: The physical time of the problem - for time varying boundary conditions.

- ```
  virtual void setBdrySlopes(FArrayBox&       a_dW,
                             const FArrayBox& a_W,
                             const int&       a_dir,
                             const Real&      a_time) = 0;
  ```

  The boundary slopes are already set to one sided difference approximations on entry. If this function doesn't change them they will be used for the slopes at the boundaries.

  - `a_dW`: The slopes over the box.
  - `a_W`: The primitive variables at the start of the time step.
  - `a_dir`: The direction normal.
  - `a_time`: The physical time of the problem - for time varying boundary conditions.

- ```
  virtual void artViscBC(FArrayBox&       a_F,
                         const FArrayBox& a_U,
                         const FArrayBox& a_divVel,
                         const int&       a_dir,
                         const Real&      a_time);
  ```
  ```

Apply artificial viscosity to the fluxes of the conserved variables at the boundaries. The default implementation does nothing to the fluxes.

  - `a_F`: The fluxes over the box. This values in the array that correspond to the boundary faces of the domain are to be updated applying the artificial viscosity at the boundaries.
  - `a_U`: The conserved variables.
  - `a_divVel`: The face centered divergence of the velocity.
  - `a_dir`: The direction normal.
  - `a_time`: The physical time of the problem - for time varying boundary conditions.

- `virtual void initialize(LevelData<FArrayBox>& a_U);`

  Fill the input with the intial conserved variables values of the problem.

  - `a_U`: The conserved variables.

Bibliography

[BC89] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys.*, 82(1):64–84, May 1989.

[CG85] P. Colella and H. M. Glaz. Efficient solution algorithms for the Riemann problem for real gases. *J. Comput. Phys.*, 59:264, 1985.

[CGL+00] P. Colella, D. T. Graves, T. J. Ligocki, D. F. Martin, D. Modiano, D. B. Serafini, and B. Van Straalen. Chombo Software Package for AMR Applications - Design Document. unpublished, 2000.

[Col90] Phillip Colella. Multidimensional upwind methods for hyperbolic conservation laws. *J. Comput. Phys.*, 87:171–200, 1990.

[Sal94] Jeff Saltzman. An unsplit 3d upwind method for hyperbolic conservation laws. *J. Comput. Phys.*, 115:153–168, 1994.