AMR Godunov Unsplit Algorithm and Implementation

P. Colella
D. T. Graves
T. J. Ligocki
D. F. Martin
B. Van Straalen [1]

Applied Numerical Algorithms Group
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA

January 25, 2008

[1]This work supported by the NASA Earth and Space Sciences Computational Technologies Program and by the U.S. Department of Energy: Director, Office of Science, Office of Advanced Scientific Computing Research under Contract DE-AC02-05CH11231.

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 [**?**]. 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 [**?**] and Saltzman [**?**]. We are solving a hyperbolic system of equations of the form

$$\frac{\partial U}{\partial t} + \sum_{d=0}^{\mathbf{D}-1} \frac{\partial F^d}{\partial x^d} = S \tag{1.1}$$

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.

$$\begin{gathered} \frac{\partial W}{\partial t} + \sum_{d=0}^{\mathbf{D}-1} A^d(W)\ \frac{\partial W^d}{\partial x^d} = S' \\ A^d = \nabla_U W \cdot \nabla_U F^d \cdot \nabla_W U \\ S' = \nabla_U W \cdot S \end{gathered} \tag{1.2}$$

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

Further note: The algorithm (and the software implementation) use the source term, S and/or S', to compute accurate fluxes which are used to advance U in time due to the hyperbolic portion of (1.1). U still needs to be updated to incorporate these source terms. Specifically, solving:

$$\frac{\partial U}{\partial t} = S$$

is left to the user. This may be solved directly/explicitly or using indirect/implicit methods but it must be included in the overall algorithm.

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 $0 \leq d < \mathbf{D}$,

$$\begin{aligned} W_{\boldsymbol{i},\pm,d} &= W^n_{\boldsymbol{i}} + \frac{1}{2}(\pm I - \frac{\Delta t}{h} A^d_{\boldsymbol{i}}) P_{\pm}(\Delta^d W_{\boldsymbol{i}}) \qquad (1.3) \\ A^d_{\boldsymbol{i}} &= A^d(W_{\boldsymbol{i}}) \\ P_{\pm}(W) &= \sum_{\pm\lambda_k > 0} (l_k \cdot W) r_k \\ W_{\boldsymbol{i},\pm,d} &= W_{\boldsymbol{i},\pm,d} + \frac{\Delta t}{2} \nabla_U W \cdot S^n_{\boldsymbol{i}} \qquad (1.4) \end{aligned}$$

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 $O(h)$.

$$\begin{aligned} F^{\mathsf{1D}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} &= R(W_{\boldsymbol{i},+,d}, W_{\boldsymbol{i}+\boldsymbol{e}^d,-,d}, d) \\ &\quad \mid R_B(W_{\boldsymbol{i},+,d}, (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d)h, d) \\ &\quad \mid R_B(W_{\boldsymbol{i}+\boldsymbol{e}^d,-,d}, (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d)h, d) \end{aligned} \qquad (1.5)$$

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.

$$W_{\boldsymbol{i},\pm,d_1,d_2} = W_{\boldsymbol{i},\pm,d_1} - \frac{\Delta t}{3h} \nabla_U W \cdot (F^{\mathsf{1D}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^{d_2}} - F^{\mathsf{1D}}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^{d_2}}) \qquad (1.6)$$

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

$$\begin{aligned} F_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^{d_1},d_2} &= R(W_{\boldsymbol{i},+,d_1,d_2}, W_{\boldsymbol{i}+\boldsymbol{e}^{d_1},-,d_1,d_2}, d_1) \\ &\quad \mid R_B(W_{\boldsymbol{i},+,d_1,d_2}, (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^{d_1})h, d_1) \\ &\quad \mid R_B(W_{\boldsymbol{i}+\boldsymbol{e}^{d_1},-,d_1,d_2}, (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^{d_1})h, d_1) \\ &\quad d_1 \neq d_2,\ 0 \leq d_1, d_2 < \mathbf{D} \end{aligned} \qquad (1.7)$$

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

$$\text{2D:} \quad W^{n+\frac{1}{2}}_{\boldsymbol{i},\pm,d} = W_{\boldsymbol{i},\pm,d} \; - \; \frac{\Delta t}{2h}\nabla_U W \cdot (F^{\text{1D}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^{d_1}} - F^{\text{1D}}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^{d_1}}) \qquad (1.8)$$
$$d \neq d_1, \; 0 \leq d, d_1 < \mathbf{D}$$

$$\text{3D:} \quad W^{n+\frac{1}{2}}_{\boldsymbol{i},\pm,d} = W_{\boldsymbol{i},\pm,d} \; - \; \frac{\Delta t}{2h}\nabla_U W \cdot (F_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^{d_1},d_2} - F_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^{d_1},d_2}) \qquad (1.9)$$
$$- \; \frac{\Delta t}{2h}\nabla_U W \cdot (F_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^{d_2},d_1} - F_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^{d_2},d_1})$$
$$d \neq d_1 \neq d_2, \; 0 \leq d, d_1, d_2 < \mathbf{D}$$

7. Compute final estimate of fluxes.

$$F^{n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} = R(W^{n+\frac{1}{2}}_{\boldsymbol{i},+,d}, W^{n+\frac{1}{2}}_{\boldsymbol{i}+\boldsymbol{e}^d,-,d}, d)$$
$$\mid \; R_B(W^{n+\frac{1}{2}}_{\boldsymbol{i},+,d}, (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d)h, d) \qquad (1.10)$$
$$\mid \; R_B(W^{n+\frac{1}{2}}_{\boldsymbol{i}+\boldsymbol{e}^d,-,d}, (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d)h, d)$$

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

$$U^{n+1}_{\boldsymbol{i}} = U^n_{\boldsymbol{i}} - \frac{\Delta t}{h}\sum_{d=0}^{\mathbf{D}-1}(F^{n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} - F^{n+\frac{1}{2}}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^d}) \qquad (1.11)$$

1.2.2 Slope Calculation

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

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

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 2.3.5). 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.

$$\alpha_C^k = l^k \cdot \delta W_C$$
$$\alpha_L^k = l^k \cdot \delta W_L$$
$$\alpha_R^k = l^k \cdot \delta W_R$$

2. Apply van Leer limiter

$$\alpha^k = \begin{cases} \min\{|\,\alpha_C^k\,|,\, 2|\,\alpha_L^k\,|,\, 2|\,\alpha_R^k\,|\} & \text{if } \alpha_L^k \cdot \alpha_R^k > 0; \\ 0 & \text{otherwise.} \end{cases}$$

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 component-wise to the differences. Formally, this can be obtained 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.

$$\zeta_{\boldsymbol{i}} = \begin{cases} \min\limits_{0 \le d < \mathbf{D}} \zeta_{\boldsymbol{i}}^d & \text{if } \sum_{d=0}^{\mathbf{D}-1} \Delta_1^d u_{\boldsymbol{i}}^d < 0 \\ 1 & \text{otherwise} \end{cases} \tag{1.12}$$
$$\zeta_{\boldsymbol{i}}^d = \min_3(\widetilde{\zeta}^d, d)_{\boldsymbol{i}}$$
$$\widetilde{\zeta}_{\boldsymbol{i}}^d = \eta(\Delta_1^d p_{\boldsymbol{i}},\, \Delta_2^d p_{\boldsymbol{i}},\, \min_3(K, d)_{\boldsymbol{i}})$$
$$\Delta_1^d p_{\boldsymbol{i}} = \frac{1}{2}(p_{\boldsymbol{i}+\boldsymbol{e}^d} - p_{\boldsymbol{i}-\boldsymbol{e}^d}) \mid p_{\boldsymbol{i}} - p_{\boldsymbol{i}-\boldsymbol{e}^d} \mid p_{\boldsymbol{i}+\boldsymbol{e}^d} - p_{\boldsymbol{i}}$$
$$\Delta_2^d p_{\boldsymbol{i}} = (\Delta_1^d p_{\boldsymbol{i}+\boldsymbol{e}^d} + \Delta_1^d p_{\boldsymbol{i}-\boldsymbol{e}^d}) \mid 2\Delta_1^d p_{\boldsymbol{i}} \mid 2\Delta_1^d p_{\boldsymbol{i}}$$

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 Artificial Viscosity

We add a small $O(h^2)$ diffusive term to the flux prior to the final conservative difference step. This "artificial viscosity" term serves to suppress instabilities occurring in multi-dimensional shocks that are nearly aligned with one of the coordinate directions; for a detailed discussion, see [**?**] and [**?**].

$$\begin{aligned}
F^{\eta+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}_d} &= F^{\eta+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}_d} - K_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d}(U^n_{\boldsymbol{i}+\boldsymbol{e}^d} - U^\eta_{\boldsymbol{i}}) \\
K_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} &= K_0 \ \max(-(D\vec{u})_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d}, 0) \\
(D\vec{u})_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} &= (u^d_{\boldsymbol{i}+\boldsymbol{e}^d} - u^d_{\boldsymbol{i}})+ \\
&\qquad \sum_{d'\neq d} \frac{1}{4}((\Delta^{d'}_+ u^{d'})_{\boldsymbol{i}} + (\Delta^{d'}_- u^{d'})_{\boldsymbol{i}} + (\Delta^{d'}_+ u^{d'})_{\boldsymbol{i}+\boldsymbol{e}^d} + (\Delta^{d'}_- u^{d'})_{\boldsymbol{i}+\boldsymbol{e}^d})
\end{aligned}$$

For typical time-dependent calculations of shocks in gases, $K_0 = 0.1$.

1.4 Extension to PPM

We can extend this algorithm to the case of using the piecewise-parabolic method of Colella and Woodward [**?**] to perform the normal predictor step [**?**]. We begin by computing spatially extrapolated face-centered values at the low and high edges of the cells.

$$W_\pm = \frac{1}{2}(W^n_{\boldsymbol{i}\pm\boldsymbol{e}} + W^n_{\boldsymbol{i}}) \pm \frac{1}{6}(\Delta^d_2 W_{\boldsymbol{i}} - \Delta^d_2 W_{\boldsymbol{i}\pm\boldsymbol{e}}) \mid W^n_{\boldsymbol{i}} - \frac{1}{2}\Delta^d_2 W_{\boldsymbol{i}} \mid W^n_{\boldsymbol{i}} + \frac{1}{2}\Delta^d_2 W_{\boldsymbol{i}}$$
$$\alpha^k_\pm = l^k \cdot (W_\pm - W^n_{\boldsymbol{i}})$$

The van Leer slopes $\Delta^d_2 W$ can be limited component-wise, or by using limiting in characteristic variables. Similarly, there are two options for limiting the parabolic profile. One is to apply the PPM limiter to the characteristic variables $\alpha^k_\pm$: if $\alpha^k_+\alpha^k_- < 0$, then

$$\begin{aligned}
\alpha^k_+ &:= s \cdot \min\{s \cdot \alpha^k_+, -2s \cdot \alpha^k_-\} \quad \text{if } (\alpha^k_+)^2 > (\alpha^k_-)^2; \\
\alpha^k_- &:= s \cdot \min\{s \cdot \alpha^k_-, -2s \cdot \alpha^k_+\} \quad \text{otherwise.}
\end{aligned}$$

where $s = sign(\alpha^k_+ - \alpha^k_-)$. If $\alpha^k_+\alpha^k_- \geq 0$, then we set $\alpha^k_+, \alpha^k_- := 0$. An alternative approach is to apply the limiter above component-wise to the differences $W_\pm - W^n_{\boldsymbol{i}}$, and then compute the characteristic amplitudes $\alpha^k_\pm$. If appropriate, we also apply the flattening coefficients (1.12) to the parabolic profiles after the limiting for monotonicity has been applied: $\alpha^k_\pm := \alpha^k_\pm \cdot \zeta_{\boldsymbol{i}}$.

Finally, we use the PPM predictor to compute the normal predictor corresponding to

(1.3).

$$W_{\boldsymbol{i},\pm,d} = W^n_{\boldsymbol{i}} + \sum_k (\alpha^k_{\pm} + \frac{1}{2}\sigma^k_{\pm}(\pm(\alpha^k_- - \alpha^k_+) - (\alpha^k_- + \alpha^k_+)(3 - 2\sigma^k_{\pm})) \cdot r^k$$

$$\sigma^k_{\pm} = \begin{cases} \pm\lambda^k_d(W^n_{\boldsymbol{i}})\frac{\Delta t}{\Delta x} & \text{if } \pm\lambda^k_d(W^n_{\boldsymbol{i}}) > 0 \\ \max\{\pm\lambda^{\pm}(W^n_{\boldsymbol{i}}), 0\}\frac{\Delta t}{\Delta x} & \text{otherwise.} \end{cases}$$

Here $\lambda^{\{+,-\}}$ is the $\{$maximum , minimum$\}$ of the wave speeds over all of the wave families.

1.5 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. $\{t^l\} = \{t^{l-1} + k\Delta t^l : 0 \le k < n^l_{ref}\}$ The algorithm in [**?**] 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 $l > 0$
    $\delta F_d^l := \delta F_d^l + \frac{1}{n_{ref}^{l-1}} \langle F_d^l \rangle$ 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 $\mathcal{C}_{n_{ref}^l}(\Omega^{l+1})$
$U^l(t^l + \Delta t^l) := U^l(t^l + \Delta t^l) - \Delta t^l D_R(\delta F^{l+1})$
$t^l := t^l + \Delta t^l$
$n_{\text{step}}^l := n_{\text{step}}^l + 1$
if $(n_{\text{step}}^l = 0 \mod n_{\text{regrid}})$ and $(n_{\text{step}}^{l-1} \neq 0 \mod n_{\text{regrid}})$
    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 AMRGodunov code makes extensive use of the AMR time-dependent infrastructure contained in the Chombo libraries. 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.

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 that facilitate implementations of block-structured adaptive algorithms. For more detailed descriptions of these classes, see the Chombo documentation [**?**].

2.2.1 Global Data Structures

The important variables in the AMRGodunov code are in 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 Boxes (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.

All of these classes are further documented in the Chombo documentation [**?**].

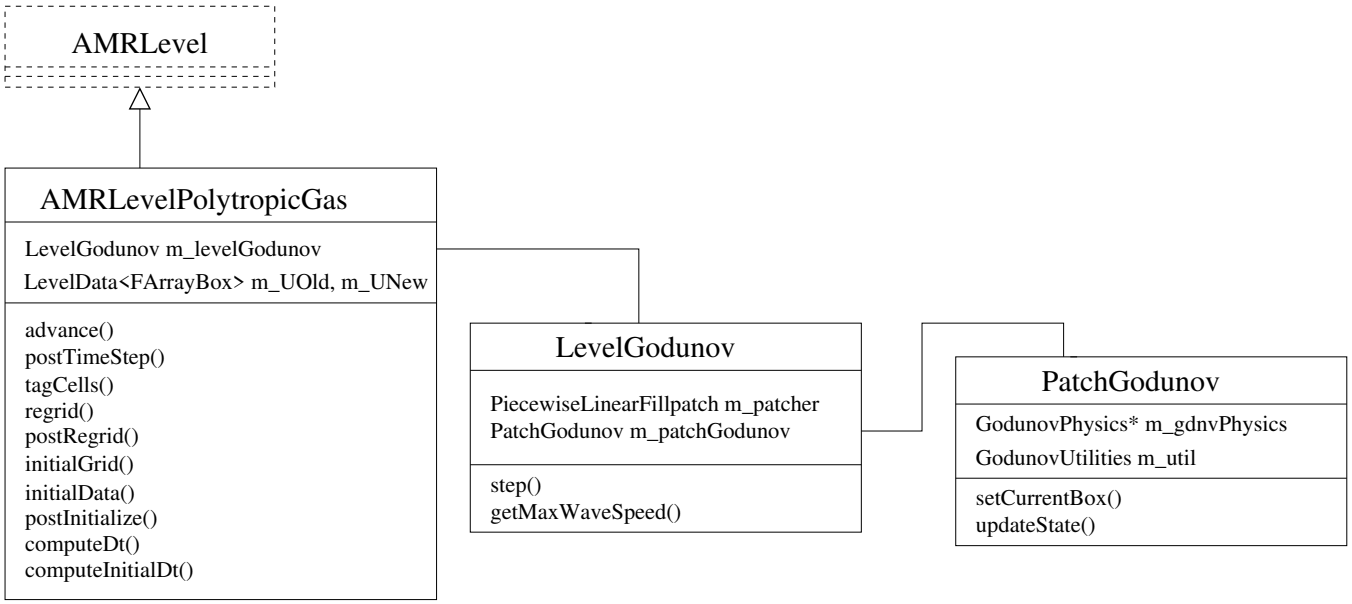

Figure 2.1: Software configuration diagram for the AMRGodunov code showing basic relationships between AMRGodunov classes and Chombo classes for the polytropic gas example.

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 initialization of 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 $\vec{U}(t^\ell + \Delta t^\ell)$). `AMRLevelPolytropicGas` also contains a `LevelGodunov` object as a member. The `LevelGodunov` class contains the functionality necessary for updating the conserved variables on a single level by one timestep. `LevelGodunov` contains as a member a `PatchGodunov` object, which in turn contains a `GodunovPhysics`-derived object. The `GodunovPhysics`-derived class contains the physics-dependent part of of the algorithm; for the polytropic gas example, this is the `PolytropicPhysics` class.

2.3 Class Hierarchy

For many hyperbolic conservation law applications, it is necessary only to implement the GodunovPhysics and PhysIBC interfaces for that application, leaving the remainder of the code unchanged. The principal AMRGodunov classes follow.

- AMRLevel<name>, the AMRLevel-derived class that 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 that will probably be common to many applications. This is where updates of U due to the source terms, S, need to be implemented.
- 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, a class that encapsulates the operations required to advance a solution on a single patch/grid. PatchGodunov owns a pointer to a GodunovPhysics-derived class. PatchGodunov also owns a GodunovUtilities object.
- GodunovUtilities, a class that handles operations common to many Godunov applications, such as slope calculations, construction of PPM interpolants, limiters, artificial viscosity coefficients, and flattening. These operations are independent of the details of the physical system to which the method is being applied, although not all of them are generally applicable: for example, artificial viscosity can be computed only for those systems in which the primitive variables include a vector velocity, i.e., continuum-mechanical systems.
- GodunovPhysics is a base class that provides an interface to the physics-dependent parts of the Godunov application. For many hyperbolic conservation law applications, it is necessary only to implement the GodunovPhysics and PhysIBC interfaces for that application, leaving the remainder of the code unchanged.
- PhysIBC, is a base class that encapsulates initial conditions and flux-based boundary conditions.

2.3.1 Class AMRLevel<name>

AMRLevel<name> is the AMRLevel-derived class with which the AMR class will directly interact. Its 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 functions that will probably be part of any implementation. These are documented here. The important data members of an 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 use during regridding, which fills newly-refined regions that were previously only covered by coarser data.

- `CoarseAverage m_coarse_average;`

  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 `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 to advance the hyperbolic portion of (1.1). If there are source terms, this is where the user can incorporate them in the advance of the conserved variables. The time step returned by this function is stored in member data `m_dtNew`.

- `void postTimeStep()`

  This function calls refluxing along the coarse-fine interface with the next finer level, and replaces the solution with an average of finer-level data in regions covered by the next finer level.

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

  This function changes the union of rectangles over which the data is defined. Where the old and new sets of rectangles intersect, solution data is copied from the existing data on this level. In 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_patchGodunov`, 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 on the given boxes. 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 GodunovPhysics* const a_godunovFactory,
              const int&                  a_normalPredOrder,
              const bool&                 a_useFourthOrderSlopes,
              const bool&                 a_usePrimLimiting,
              const bool&                 a_useCharLimiting,
              const bool&                 a_useFlattening,
              const bool&                 a_useArtificialViscosity,
              const Real&                 a_artificialViscosity,
              const bool&                 a_hasCoarser,
              const bool&                 a_hasFiner);
  ```

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

  - `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_godunovFactory: The factory for the problem specific physics and analysis of the PDE being solved, e.g., characteristic analysis. The GodunovPhysics class is described below. Note: this object is its own factory.
- a_normalPredOrder: The order of the normal predictor used during numerical integration. This must have a value of 1 (PLM) or 2 (PPM).
- a_useFourthOrderSlopes: If true, use a 4th-order slope computation. Otherwise use a 2nd-order slope computation.
- a_usePrimLimiting: If true, do slope limiting on the primitive variables. Note: Currently, simultaneous slope limiting of the primitive and characteristic variables is not supported.
- a_useCharLimiting: If true, do slope limiting on the characteristic variables. Note: Currently, simultaneous slope limiting of the primitive and characteristic variables is not supported.
- a_useFlattening: If true, do slope flattening. Note: This requires the enabling of 4th-order slope computations and some form of slope limiting.
- a_useArtificialViscosity: If true, apply artificial viscosity.
- a_artificialViscosity: The artificial viscosity coefficient used in applying artificial viscosity.
- 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 that exists between levels (e.g., flux registers) is needed.

- ```
  Real step(LevelData<FArrayBox>&       a_U,
            LevelData<FArrayBox>&       a_flux[CH_SPACEDIM],
            LevelFluxRegister&          a_finerFluxRegister,
            LevelFluxRegister&          a_coarserFluxRegister,
            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_finerFluxRegister, a_coarserFluxRegister: The flux registers between this level and the next coarser (or finer) levels.
  ```

- a_S: Source terms from the right-hand side of the quasilinear form of system of PDEs being solved (integrated) - S' in equation (1.2). 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.

- `GodunovPhysics* getGodunovPhysicsPtr();`

  Return a pointer to the GodunovPhysics object used by the PatchGodunov member of this LevelGodunov.

2.3.3 Class PatchGodunov

The base class PatchGodunov provides an interface to LevelGodunov for managing the update of a single patch using the unsplit second-order Godunov method described above. It provides a top-level implementation of the algorithm by calling member functions in the GodunovUtilities class (which contains physics-independent components that make up the algorithm) and by calling member functions of the object pointed to by a_gdnvPhysicsPtr (which contains physics-dependent functions).

There are four types of grid variables that appear in the unsplit Godunov method in section 1.2: conserved variables, primitive variables, fluxes, and source terms, denoted 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. Redundant primitive variable quantities are often carried that parameterize the equation of state in order to avoid multiple calls to the equation-of-state function. Also, it is often convenient to split the fluxes 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 that the user will not need to change for a variety of physical problems.

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

```
                    const GodunovPhysics* const a_gdnvPhysicsPtr,
                    const int&                  a_normalPredOrder,
                    const bool&                 a_useFourthOrderSlopes,
                    const bool&                 a_usePrimLimiting,
                    const bool&                 a_useCharLimiting,
                    const bool&                 a_useFlattenping,
                    const bool&                 a_useArtificialViscosity,
                    const Real&                 a_artificialViscosity);
```

Set the domain and grid spacing.

- a_domain: The problem domain index space.
- a_dx: The grid spacing for this patch/grid.
- a_gdnvPhysicsPtr: A pointer to the object that supplies all the physics associated with the problem being solved.
- a_normalPredOrder: The order of the normal predictor used during numerical integration. This must have a value of 1 (PLM) or 2 (PPM).
- a_useFourthOrderSlopes: If true, use a 4th-order slope computation. Otherwise use a 2nd-order slope computation.
- a_usePrimLimiting: If true, do slope limiting on the primitive variables. Note: Currently, simultaneous slope limiting of the primitive and characteristic variables is not supported.
- a_useCharLimiting: If true, do slope limiting on the characteristic variables. Note: Currently, simultaneous slope limiting of the primitive and characteristic variables is not supported.
- a_useFlattening: If true, do slope flattening. Note: This requires the enabling of 4th-order slope computations and some form of slope limiting.
- a_useArtificialViscosity: If true, apply artificial viscosity.
- a_artificialViscosity: The artificial viscosity coefficient used in applying artificial viscosity.

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

  Set the current physical time of the problem.

  - a_time: The current physical time of the problem.

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

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

  - a_box: The box over which the conserved variables will 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 on each of the faces used to update the conserved variables (used for refluxing).
- a_maxWaveSpeed: The maximum wave speed for this patch/grid.
- a_S: Source terms from the right-hand side of the quasilinear form of system of PDEs being solved (integrated): S' in equation (1.2). 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_dt: The time step for this patch/grid.
- a_box: The box to be used for the computation/update.

- `GodunovPhysics* getGodunovPhysicsPtr();`

  Return a pointer to the GodunovPhysics object used by this object.

2.3.4 Class GodunovPhysics

GodunovPhysics is an interface class owned and used by PatchGodunov, through which a user specifies the physics of the problem. Most methods of the GodunovPhysics class are pure virtual. The user is expected to create a subclass of GodunovPhysics specific to the problem they are solving, and in that subclass implement all of these methods.

IMPORTANT NOTE: It is assumed that the characteristic analysis puts the smallest eigenvalue first and the largest eigenvalue last, and orders the characteristic variables accordingly.

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

  Set the initial and boundary condition pointer used by the integrator for the current level. This must be called for the class to be fully defined and usable.

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

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

  Compute the maximum wave speed of the state over the region.

  - a_U: The conserved state.
  - a_box: The region over which to calculate the max wave speed.

- `virtual GodunovPhysics* new_godunovPhysics() const = 0;`

  Factory method. Reproduce oneself and return a pointer to the new object.
  ```

- `virtual int numConserved() = 0;`

  Return the number of conserved variables being updated. This may be less than the total number of conserved variables.

- `virtual Vector<string> stateNames() = 0;`

  Return the names of all the conserved variables.

- `virtual int numFluxes() = 0;`

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

- `virtual int numPrimitives() = 0;`

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

- ```
  virtual void charAnalysis(FArrayBox&       a_dW,
                            const FArrayBox& a_W,
                            const int&       a_dir,
                            const Box&       a_box) = 0;
  ```

  Transform `a_dW` from primitive to characteristic variables.

  IMPORTANT NOTE: It is assumed that the characteristic analysis puts the smallest eigenvalue first and the largest eigenvalue last, and orders the characteristic variables accordingly.

  - `a_dW`: On input, contains the increments of the primitive variables. On output, contains the increments in the characteristic variables.
  - `a_W`: The state in primitive variables.
  - `a_dir`: Spatial direction.
  - `a_box`: The box over which the calculation is carried out.

- ```
  virtual void charSynthesis(FArrayBox&       a_dW,
                             const FArrayBox& a_W,
                             const int&       a_dir,
                             const Box&       a_box) = 0;
  ```

  Transform `a_dW` from characteristic to primitive variables.

  IMPORTANT NOTE: It is assumed that the characteristic analysis puts the smallest eigenvalue first and the largest eigenvalue last, and orders the characteristic variables accordingly.

  - `a_dW`: On input, contains the increments of the characteristic variables. On output, contains the increments in the primitive variables.
  - `a_W`: The state in primitive variables.

- a_dir: Spatial direction.
- a_box: The box over which the calculation is carried out.

- `virtual void charValues(FArrayBox&       a_lambda,`
  `const FArrayBox& a_W,`
  `const int&       a_dir,`
  `const Box&       a_box) = 0;`

  Compute the characteristic values (eigenvalues).

  IMPORTANT NOTE: It is assumed that the characteristic analysis puts the smallest eigenvalue first and the largest eigenvalue last, and orders the characteristic variables accordingly.

  - a_lambda: Eigenvalues of a_W.
  - a_W: The state in primitive variables.
  - a_dir: Spatial direction.
  - a_box: The box over which the calculation is carried out.

- `virtual void computeUpdate(FArrayBox&       a_dU,`
  `FluxBox&         a_F,`
  `const FArrayBox& a_U,`
  `const FluxBox&   a_WHalf,`
  `const bool&      a_useArtificialViscosity,`
  `const Real&      a_artificialViscosity,`
  `const Real&      a_currentTime,`
  `const Real&      a_dx,`
  `const Real&      a_dt,`
  `const Box&       a_box);`

  Compute the increment in the conserved variables from face variables. Compute dU = dt*dU/dt, the change in the conserved variables over the time step. The fluxes returned are suitable for use in refluxing. This has a default implementation but can be redefined as needed.

  - a_dU: The update to the conserved variables.
  - a_F: The fluxes associate with a_dU.
  - a_U: The initial conserved variable values.
  - a_WHalf: The extrapolated state in primitive variables at faces.
  - a_useArtificialViscosity: If true, apply artificial viscosity.
  - a_artificialViscosity: The artificial viscosity coefficient used in applying artificial viscosity.
  - a_currentTime: The current simulation time.
  - a_dx: The grid spacing for this patch/grid.
  - a_dt: The time step for this patch/grid.
  - a_box: The box over which the calculation is carried out.

- `virtual void getFlux(FArrayBox&       a_flux,`
  `const FArrayBox& a_WHalf,`
  `const int&       a_dir,`
  `const Box&       a_box);`

  Compute the fluxes from primitive variables on a face. This has a default implementation which throws an error. The method is here so that the default implementation of `computeUpdate` can use it and the user can supply it. It has an implementation, so if the user redefines `computeUpdate`, they aren't force to implement `getFlux`, which is used only by the default implementation of `computeUpdate`.

  - `a_flux`: The output fluxes.
  - `a_WHalf`: The extrapolated state in primitive variables at faces.
  - `a_dir`: Spatial direction.
  - `a_box`: The box over which the calculation is carried out.

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

  Add to (increment) the source terms given the current state.

  - `a_S`: On input, `a_S` contains the current source terms from the right-hand side of the quasilinear form of system of PDEs being solved (integrated): S' in equation (1.2). On output, `a_S` has had any additional source terms (based on the current state, `a_W`) added to it.
  - `a_W`: The state in primitive variables.
  - `a_box`: The box over which the calculation is carried out.

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

  Compute the solution to the Riemann problem on each `a_dir` face in `a_box`.

  - `a_WStar`: Riemann problem solution.
  - `a_WLeft`: Solution on the left side of the discontinuity.
  - `a_WRight`: Solution on the right side of the discontinuity.
  - `a_W`: The state in primitive variables.
  - `a_time`: The solution time.
  - `a_dir`: Spatial direction.
  - `a_box`: The box over which the calculation is carried out.

- `virtual void postNormalPred(FArrayBox& a_dWMinus,`
  `FArrayBox& a_dWPlus,`
  `const FArrayBox& a_W,`
  `const Real& a_dt,`
  `const Real& a_dx,`
  `const int& a_dir,`
  `const Box& a_box) = 0;`

  Perform post-processing of values for normal predictor. This is done, for example, to add any spatial derivatives that are not accounted for in the characteristic analysis, such as the Stone correction in MHD. Also, bounding the ranges of primitive variables must be done here.

  - `a_dWMinus`: Extrapolated solution on the low side of the cell.
  - `a_dWPlus`: Extrapolated solution on the high side of the cell.
  - `a_W`: Cell-centered solution value at the beginning of the time step.
  - `a_dt`: The time step for this patch/grid.
  - `a_dx`: The grid spacing for this patch/grid.
  - `a_dir`: Spatial direction.
  - `a_box`: The box over which the calculation is carried out.

- `virtual void quasilinearUpdate(FArrayBox& a_AdWdx,`
  `const FArrayBox& a_wHalf,`
  `const FArrayBox& a_W,`
  `const Real& a_scale,`
  `const int& a_dir,`
  `const Box& a_box) = 0;`

  Compute the partial update based on upwind differencing to the primitive variables using derivatives in the `a_dir` direction, for example in equations (1.6), (1.8), and (1.9).

  - `a_AdWdx`: On output, the upwind difference estimate of $\tau A_d \frac{\partial W}{\partial x_d}$.
  - `a_wHalf`: Solution to the Riemann problem at adjacent cell faces in the d direction.
  - `a_W`: Cell-centered primitive values that are being corrected.
  - `a_scale`: Scale factor τ.
  - `a_dir`: Spatial direction.
  - `a_box`: The cell-centered box over which the calculation is carried out.

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

  Compute primitive variables from conserved variables.

  - `a_W`: On output, the primitive variables.

  - a_U: The conserved variables.
  - a_box: The region over which the calculation is carried out.

- `virtual Interval velocityInterval() = 0;`

  Return the interval of component indices of the velocities within the primitive variables. Used for slope flattening (slope computation) and computing the divergence of the velocity (artificial viscosity).

- `virtual int pressureIndex() = 0;`

  Return the component index of the pressure within the primitive variables. Used for slope flattening (slope computation).

- `virtual Real smallPressure() = 0;`

  Return a value that is used by slope flattening to limit (away from zero) the absolute value of a slope in the `pressureIndex()` component (slope computation).

- `virtual int bulkModulusIndex() = 0;`

  Return the component index within the primitive variables for the bulk modulus. Used for slope flattening (slope computation) used as a normalization to measure shock strength.

2.3.5 Class `PhysIBC`

`PhysIBC` is an interface class owned and used by `PatchGodunov`, and through which a user specifies the initial and boundary of conditions of the particular problem. These boundary conditions are flux-based. `PhysIBC` contains as member data the grid 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 for this patch/grid.

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

  This is a factory method. It allocates and returns the pointer to a new `PhysIBC` object.

- `virtual void initialize(LevelData<FArrayBox>& a_U) = 0;`

  Fill the input with the initial conserved variable state of the problem.
  ```

  - a_U: The conserved variables.

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

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

  - a_WGdnv: The primitive variables over the face-centered box. The values in the array located along the boundary faces of the domain are replaced with boundary values.
  - a_Wextrap: The extrapolated values 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 normal direction 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 normal direction.
  - 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) = 0;
  ```

  Apply artificial viscosity to the fluxes of the conserved variables at the boundaries.
  ```

– a_F: The fluxes over the box. The values in the array along the boundary faces of the domain are updated by applying the artificial viscosity at the boundaries.

– a_U: The conserved variables.
– a_divVel: The face-centered divergence of the cell-centered velocity.
– a_dir: The normal direction.
– a_time: The physical time of the problem — for time-varying boundary conditions.