EBAMRTools: EBChombo's Adaptive Refinement Library

P. Colella
D. T. Graves
T. J. Ligocki
D. Modiano
B. Van Straalen

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

April 16, 2003

Contents

1 Introduction

This document is meant to discuss the different components of the EBAMRTools component of the EBChombo infrastructure for embedded boundary, block-structured adaptive mesh applications. The principal operations that these tools execute are as follows:

- Average a level's worth of data onto the next coarser level.
- Interpolate in a piecewise-linear fashion data from a coarser level to a finer level.
- Fill ghost cells at a coarse-fine interface with a second-order interpolation between the coarse and fine data.
- Fill ghost cells at a coarse-fine interface with data interpolated using a bilinear interpolation.
- Preserve multi-level conservation using refluxing.
- Redistibute mass differences between stable and conservative schemes.

After a discourse on the notational difficulties of embedded boundaries, we will discuss our algorithm for each of these tasks.

2 Notation

All these operations take place in a very similar context to that presented in [CGL+00]. For non-embedded boundary notation, refer to that document. The standard (i, j, k) is not sufficient here to denote a computational cell as there can be multiple VoFs per cell. We define $\boldsymbol{v}$ to be the notation for a VoF and $\boldsymbol{f}$ to be a face. The function $ind(\boldsymbol{v})$

produces the cell which the VoF lives in. We define $\boldsymbol{v}^+(f)$ to be the VoF on the high side of face f; $\boldsymbol{v}^-(f)$ is the VoF on the low side of face $\boldsymbol{f}$; $\boldsymbol{f}_d^+(v)$ is the set of faces on the high side of VoF v; $f_d^-(v)$ is the set of faces on the low side of VoF $\boldsymbol{v}$, where $d \in \{x, y, z\}$ is a coordinate direction (the number of directions is D). Also, we compose these operators to represent the set of VoFs directly connected to a given VoF: $v_d^+(v) = \boldsymbol{v}^+(f_d^+(\boldsymbol{v}))$ and $\boldsymbol{v}_d^-(v) = v^-(f_d^-(\boldsymbol{v}))$. The $<<$ operator shifts data in the direction of the left hand argument:

$$(\phi << \boldsymbol{e}^d)_{\boldsymbol{v}} = \phi_{v_d^+(v)} \tag{1}$$

We follow the same approach in the EB case in defining multilevel data and operators as we did for ordinary AMR. Given an AMR mesh hierarchy $\{\Omega^l\}_{l=0}^{lmax}$, we define the valid VoFs on level l to be

$$\mathcal{V}_{valid}^l = ind^{-1}(\Omega_{valid}^l) \tag{2}$$

and composite cell-centered data

$$\varphi^{comp} = \{\varphi^{l,valid}\}_{l=0}^{lmax}, \varphi^{l,valid} : \mathcal{V}_{valid}^l \to \mathbb{R}^m \tag{3}$$

For face-centered data,

$$\begin{array}{c} \mathcal{F}_{valid}^{l,d} = ind^{-1}(\Omega_{valid}^{l,\boldsymbol{e}^d}) \\ \vec{F}^{l,valid} = (F_0^{l,valid}, \dots, F_{D-1}^{l,valid}) \\ F_d^{l,valid} : \mathcal{F}_{valid}^{l,d} \to \mathbb{R}^m \end{array} \tag{4}$$

3 Conservative Averaging

Assume that there are two levels of grids Ω^c, Ω^f, with data defined on the fine grid and on the valid region of the coarse grid

$$\varphi^f : ind^{-1}(\Omega^f) \to \mathbb{R}\varphi^{c,valid} : ind^{-1}(\Omega_{valid}^c) \to \mathbb{R} \tag{5}$$

We assume that $\mathcal{C}_r(\tilde{\Omega}^f) \cap \Gamma^c \subset \Omega^c$. We want to replace the coarse data which is covered by fine data with the volume-weighted average of the fine data. This operator is used to average from finer levels on to coarser levels, or for constructing averaged residuals in multigrid iteration. We define the volume weighted average

$$\begin{array}{c} \varphi_{\boldsymbol{v_c}}^c = Av(\varphi^f, n_{ref})_{\boldsymbol{v_c}} \\ Av(\varphi^f) = \frac{1}{V^c} \sum\limits_{\boldsymbol{v_f} \in \mathcal{F}} V^f \varphi_{\boldsymbol{v_f}} \\ \mathcal{F} = \mathcal{C}_{n_{ref}}^{-1}(\boldsymbol{v_c}) \end{array} \tag{6}$$

4 Interpolation Operations

4.1 Piecewise Linear Interpolation

This method is primarily used to initialize fine grid data after regridding. Given a level array φ^c on Ω^c, we want to compute $I_{pwl}(\varphi)$ defined on an Ω^f properly nested in Ω^c. For the values on $C(\tilde{\Omega}^f)$, interpolate in a piecewise-linear fashion in space, using the values $\tilde{\varphi}^c$ (we assume that the coarse data already contains the average of the fine data as discussed in the last section).

$$\begin{gathered}\varphi^f_{\boldsymbol{v_f}} = \tilde{\varphi}^c_{\boldsymbol{v_c}} + \sum_{d=0}^{D-1} \left(\tfrac{(ind(\boldsymbol{v_f})_d + \frac{1}{2})}{n_{ref}} - ind(\boldsymbol{v_c}) + \tfrac{1}{2})\right)\Delta^d \cdot \varphi^c_{\boldsymbol{v_c}} \\ \text{where } \boldsymbol{v_c} \in ind^{-1}(\tilde{\Omega}^f - \Omega^f) \\ \boldsymbol{v_c} = \mathcal{C}_{n_{ref}}(\boldsymbol{v_f}).\end{gathered} \tag{7}$$

The slopes Δ^d are computed using minmod limiting as shown below:

$$\begin{gathered}\Delta^d W_{\boldsymbol{v_c}} = \delta^{minmod}(W_{\boldsymbol{v_c}}) | \delta^L(W_{\boldsymbol{v_c}}) | \delta^R(W_{\boldsymbol{v_c}}) | 0 \\ \delta^L(W_{\boldsymbol{v_c}}) = W_{\boldsymbol{v_c}} - (W^n_{\boldsymbol{v}<<-\boldsymbol{e}^d}) \\ \delta^R(W_{\boldsymbol{v_c}}) = (W^n_{\boldsymbol{v}<<\boldsymbol{e}^d}) - W_{\boldsymbol{v_c}}\end{gathered} \tag{8}$$

$$\delta^{minmod} = \left\{ \begin{array}{ll} min(|\delta_L|, |\delta_R|) \cdot sign(\delta_L + \delta_R) & \text{if } \delta_L \cdot \delta_R > 0 \\ 0 & \text{otherwise} \end{array} \right\} \tag{9}$$

The shift operator (denoted by $<<$) is defined using a simple average of connected values.

4.2 Piecewise-Linear Coarse-Fine Boundary Interpolation

In the next algorithm, we use the same linear interpolant but we also interpoalte in time between levels of time. We have the solution on the coarser level of refinement at two time levels, t_{Cold} and t_{Cnew}. We want to compute an extension $\tilde{\varphi}^f$ of φ^f on $\tilde{\Omega}^f = \mathcal{G}(\Omega^f, p) \cap \Gamma^f, p > 0$ that exists at time level t_F where $t_{Cold} < t_f < t_{Cnew}$. We assume that $\mathcal{C}_r(\tilde{\Omega}^f) \cap \Gamma^c C \Omega^c$. Extend $\varphi^{c,valid}$ to φ^c, defined on all of $ind^{-1}(\Omega^c)$.

$$\varphi^c_{\boldsymbol{v_c}} = Av(\varphi^f, n_{ref})_{\boldsymbol{v_c}}, \boldsymbol{v_c} \in ind^{-1}\mathcal{C}_{n_{ref}}(\Omega^f) \tag{10}$$

At both t_{Cold} and t_{Cnew}, for the values on $\tilde{\Omega}^f - \Omega^f$ compute a piecewise linear interpolant, using the values $\tilde{\varphi}^c$.

$$\begin{gathered}\tilde{\varphi}^f_{\boldsymbol{v_f}} = \tilde{\varphi}^f_{\boldsymbol{v_c}} + \sum_{d=0}^{D-1} \left(\tfrac{(ind(\boldsymbol{v_f})_d + \frac{1}{2})}{n_{ref}} - (ind(\boldsymbol{v_c}) + \tfrac{1}{2})\right)\Delta^d \cdot \varphi^c_{\boldsymbol{v_c}} \\ \text{where } \boldsymbol{v_c} \in ind^{-1}(\tilde{\Omega}^f - \Omega^f), \\ \boldsymbol{v_c} = \mathcal{C}_{n_{ref}}(\boldsymbol{v_f}).\end{gathered} \tag{11}$$

The slopes Δ^d are computed using minmod limiting as shown in equation 9. We then interpolate in time between the new and old interpolated values.

$$\varphi^f_{\boldsymbol{v_f},t_F} = \tilde{\varphi}^f_{\boldsymbol{v_f},t_{Cold}} + \frac{t_F - t_{Cold}}{t_{Cnew} - t_{Cold}}(\tilde{\varphi}^f_{\boldsymbol{v_f},t_{Cnew}} - \tilde{\varphi}^f_{\boldsymbol{v_f},t_{Cold}}) \tag{12}$$

This process should produce an interpolated value which has second-order error in both time and space.

4.3 Quadratic Coarse-Fine Boundary Interpolation

At VoFs where the embedded boundary crosses the coarse-fine boundary, we use the algorithm described in 4.1. On all other cells, we use the algorithm in [CGL$^+$00].

5 Redistribution

To preserve stability and conservation in embedded boundary calculations, we must redistribute a quantity of mass δM (the difference between stable and conservative updates) from irregular VoFs to their neighbors. This mass is normalized by h^D where h is the grid spacing on the level. We define $\eta_{\boldsymbol{v}}$ to be the set of neighbors (no farther away than the redistribution radius) which can be reached by a monotonic path. We then assign normalized weights to each of the neighbors $\boldsymbol{v}^{'}$ and divide the mass accordingly:

$$\delta M_{\boldsymbol{v}} = \sum_{\boldsymbol{v}' \in \eta_{\boldsymbol{v}}} w_{\boldsymbol{v},'} \kappa_{\boldsymbol{v}'} \delta M_{\boldsymbol{v}} \tag{13}$$

where

$$\sum_{\boldsymbol{v}' \in \eta_{\boldsymbol{v}}} w_{\boldsymbol{v},\boldsymbol{v}'} \kappa_{\boldsymbol{v}'} = 1 \tag{14}$$

We then update the solution U at the neighboring cells $\boldsymbol{v}^{'}$

$$U^l_{\boldsymbol{v}'} \mathrel{+}= w_{\boldsymbol{v},\boldsymbol{v}'} \delta M^l_{\boldsymbol{v}}. \tag{15}$$

This operation occurs at all $\boldsymbol{v} \in ind^{-1}(\Omega^l)$ without regard to valid or invalid regions. If the irregular cell is within the redistribution radius of a coarse-fine interface, we must account for mass that is redistributed across the interface.

5.1 Multilevel Redistribution Summary

We begin with $\delta M^l_{\boldsymbol{v}}, \boldsymbol{v} \in ind^{-1}\Omega^l$, the redistribution mass for level l.

Define the redistribution radius to be R^r. We define the coarsening operator to be $C_{N_{ref}}$ and the refinement operator to be $C^{-1}_{N_{ref}}$. We define the "growth" operator to be G. The operator which produces the Z^D index of a vof is ind and the operator to produces the VoFs for points in Z^D is ind^{-1}.

If $\boldsymbol{v}$ is part of the valid region, the redistribution mass is divided into three parts,

$$\begin{gathered}\delta M_{\boldsymbol{v}}^l = \delta^1 M_{\boldsymbol{v}}^l + \delta^2 M_{\boldsymbol{v}}^{l,l+1} + \delta^2 M_{\boldsymbol{v}}^{l,l-1}, \\ \boldsymbol{v} \in ind^{-1}(\Omega^{l,valid}).\end{gathered} \tag{16}$$

$\delta^1 M_{\boldsymbol{v}}^l$ is the part of the mass which is put onto the $\Omega^{l,valid}$. $\delta^2 M_{\boldsymbol{v}}^{l,l+1}$ is the part of the mass which is redistributed to $\Omega^l \cap C_{N_{ref}}(\Omega^{l+1})$ (the part of the level covered by the next finer level). $\delta M_{\boldsymbol{v}}^{l,l-1}$ is the part of the mass which is redistributed off level l.

If $\boldsymbol{v}$ is not part of the valid region, the redistribution mass is divided into two parts,

$$\begin{gathered}\delta M_{\boldsymbol{v}}^l = \delta^I M_{\boldsymbol{v}}^l + \delta M_{\boldsymbol{v}}^{l,l} \\ \boldsymbol{v} \in ind^{-1}(\Omega - \Omega^{l,valid}).\end{gathered} \tag{17}$$

$\delta^I M_{\boldsymbol{v}}^l$ is the portion of $\delta^l M_{\boldsymbol{v}}^l$ which is redistributed to other invalid VoFs of level l. $\delta^I M^P l, l\boldsymbol{v}$ is the portion of $\delta^l M_{\boldsymbol{v}}^l$ which is redistributed to valid VoFs of level l and must be removed later from the solution.

We must account for $\delta M_{\boldsymbol{v}}^{l,l-1}$, $\delta^2 M_{\boldsymbol{v}}^{l,l+1}$ and $\delta^3 M_{\boldsymbol{v}}^{l,l}$ to preserve conservation. $\delta^2 M_{\boldsymbol{v}}^{l,l+1}$ is added to the level $l+1$ solution. $\delta^2 M_{\boldsymbol{v}}^{l,l-1}$ is added to the level $l-1$ solution. $\delta^3 M_{\boldsymbol{v}}^{l,l}$ is removed from the level l solution.

5.2 Coarse to Fine Redistribution

The mass going from coarse to fine is accounted for as follows. Recall that the mass we store is normalized by h_c^D where h_c is the grid spacing of the level of the source. Define h_f to be the grid spacing of the destination. For all VoFs $\boldsymbol{v_c} \in ind^{-1}(C_{N_{ref}}(G(\Omega^{l+1}, R^r) - \Omega^{l+1}))$, we define the coarse-to-fine redistribtuion mass $\delta^2 M^{l,l+1}$ to be

$$\begin{gathered}\delta^2 M_{\boldsymbol{v_c}}^{l,l+1} = \sum_{\boldsymbol{v_c'} \in S(\boldsymbol{v_c})} \delta M_{\boldsymbol{v_c}}^l w_{\boldsymbol{v_c},\boldsymbol{v_c'}} \kappa_{\boldsymbol{v_c'}} \\ S(\boldsymbol{v_c}) = \eta_{\boldsymbol{v_c}} \cap ind^{-1}(C_{N_{ref}}(\Omega^{l+1})).\end{gathered} \tag{18}$$

Define $\zeta_{\boldsymbol{v_c'}}^2$ to be the unnormalized mass that goes to VoF $\boldsymbol{v_c'}$. We distribute this mass to the VoFs $\boldsymbol{v_f'}$ that cover $\boldsymbol{v_c'}$ ($\boldsymbol{v_f'} \in C_{N_{ref}}^{-1}(\boldsymbol{v_c'})$) in a volume-weighted fashion.

$$\begin{gathered}\zeta_{\boldsymbol{v_c'}}^2 = h_c^D w_{\boldsymbol{v_c},\boldsymbol{v_c'}} \kappa_{\boldsymbol{v_c'}} \delta M_{\boldsymbol{v_c}}^l \\ \zeta_{\boldsymbol{v_f'}}^2 = \frac{\kappa_{\boldsymbol{v_f'}} h_f^D}{\kappa_c h_c^D} \zeta_{\boldsymbol{v_c'}}^2 \\ \zeta_{\boldsymbol{v_f'}}^2 = \kappa_{\boldsymbol{v_f'}} h_f^D w_{\boldsymbol{v_c},\boldsymbol{v_c'}} \delta M_{\boldsymbol{v_c}}^l\end{gathered} \tag{19}$$

The change in the fine solution is the given by

$$\begin{aligned}\delta U^{l+1}_{\boldsymbol{v_f'}} = \frac{\zeta^2_{\boldsymbol{v_f'}}}{\kappa_{\boldsymbol{v_f'}} h_f^D} &= \delta M^l_{\boldsymbol{v_c}} w_{\boldsymbol{v_c},\boldsymbol{v_c'}} \\ U^{l+1}_{\boldsymbol{v_f'}} &\mathrel{+}= \delta M^l_{\boldsymbol{v_c}} w_{\boldsymbol{v_c},\boldsymbol{v_c'}} \\ \boldsymbol{v_c} &\in ind^{-1}(C_{N_{ref}}(G(\Omega^{l+1}, R^r) - \Omega^{l+1})) \\ \boldsymbol{v_c'} &= \eta_{\boldsymbol{v_c}} \cap ind^{-1}(C_{N_{ref}}(\Omega^{l+1})) \\ \boldsymbol{v_f'} &\in C^{-1}_{N_{ref}}(\boldsymbol{v_c'})\end{aligned} \tag{20}$$

This can be interpreted as a piecewise-constant interpolation of the solution density.

5.3 Fine to Coarse Redistribution

The mass going from fine to coarse is accounted for as follows. Recall that the mass we store is normalized by h_f^D where h_f is the grid spacing of the level of the source. Define h_c to be the grid spacing of the destination. For all VoFs $\boldsymbol{v_f} \in ind^{-1}(\Omega^l - G(\Omega^l, -R^r))$, we define the fine-to-coarse redistribtuion mass $\delta^2 M^{l,l-1}$ to be

$$\begin{aligned}\delta^2 M^{l,l-1}_{\boldsymbol{v_f}} &= \sum_{\boldsymbol{v_f'} \in Q(\boldsymbol{v_f})} \delta M^l_{\boldsymbol{v_f}} w_{\boldsymbol{v_f},\boldsymbol{v_f'}} \kappa_{\boldsymbol{v_f'}} \\ Q(\boldsymbol{v_f}) &= \eta_{\boldsymbol{v_f}} \cap ind^{-1}(C^{-1}_{N_{ref}}(\Omega^{l-1}) - \Omega^l).\end{aligned} \tag{21}$$

For all VoFs $\boldsymbol{v_c} \in ind^{-1}(C_{N_{ref}}(G(\Omega^{l+1}, R^r) - \Omega^{l+1}))$, we define the coarse-to-fine redistribtuion mass $\delta^2 M^{l,l+1}$ to be

$$\begin{aligned}\delta^2 M^{l,l+1}_{\boldsymbol{v_c}} &= \sum_{\boldsymbol{v_c'} \in S(\boldsymbol{v_c})} \delta M^l_{\boldsymbol{v_c}} w_{\boldsymbol{v_c},\boldsymbol{v_c'}} \kappa_{\boldsymbol{v_c'}} \\ S(\boldsymbol{v_c}) &= \eta_{\boldsymbol{v_c}} \cap ind^{-1}(C_{N_{ref}}(\Omega^{l+1})).\end{aligned} \tag{22}$$

Define $\zeta^2_{\boldsymbol{v_f'}}$ to be the unnormalized mass that goes to VoF $\boldsymbol{v_f'}$. We distribute this mass to the VoF $\boldsymbol{v_c'} = C_{N_{ref}}(\boldsymbol{v_f'})$.

$$\zeta^2_{\boldsymbol{v_f'}} = \zeta^2_{\boldsymbol{v_c'}} = h_f^D w_{\boldsymbol{v_f},\boldsymbol{v_f'}} \kappa_{\boldsymbol{v_f'}} \delta M^l_{\boldsymbol{v_f}} \tag{23}$$

We define $\delta U^{l-1}_{\boldsymbol{v_c'}}$ to be the change in the coarse solution density due to $\delta^w M_{\boldsymbol{v_f},\boldsymbol{v_f'}}$:

$$\delta U^{l-1}_{\boldsymbol{v_c'}} = \frac{\zeta^2_{\boldsymbol{v_f'}}}{\kappa_{\boldsymbol{v_c'}} h_c^D} \tag{24}$$

Substituting from above, we increment the coarse solution as follows

$$\begin{aligned}U^{l-1}_{\boldsymbol{v_c'}} &\mathrel{+}= \frac{\kappa_{\boldsymbol{v_f'}}}{\kappa_{\boldsymbol{v_c'}} N^D_{ref}} \delta M^l_{\boldsymbol{v_f}} w_{\boldsymbol{v_f},\boldsymbol{v_f'}} \\ \boldsymbol{v_f} &\in ind^{-1}(\Omega^l - G(\Omega^l, -R^r)), \\ \boldsymbol{v_f'} &\in \eta_{\boldsymbol{v_f}} \cap ind^{-1}(C^{-1}_{N_{ref}}(\Omega^{l-1}) - \Omega^l) \\ \boldsymbol{v_c'} &= C_{N_{ref}}(\boldsymbol{v_f'})\end{aligned} \tag{25}$$

5.4 Coarse to Coarse Redistribution

The re-redistribution algorithm proceeds as follows. Given $\boldsymbol{v} \in ind^{-1}(C_{N_{ref}}(\Omega^{l+1})$, we define the re-redistribution mass $\delta^3 Ml, l$ to be

$$\begin{array}{c} \delta^3 M^{l,l}_{\boldsymbol{v}} = \sum\limits_{\boldsymbol{v}' \in T(\boldsymbol{v})} \delta M^l_{\boldsymbol{v}} w_{\boldsymbol{v},\boldsymbol{v}'} \kappa_{\boldsymbol{v}'} \\ T(\boldsymbol{v}) = \eta_{\boldsymbol{v}} \cap ind^{-1}(\Omega^l). \end{array} \tag{26}$$

In the level redistribution step, we have added this mass to the solution density using equation 15. Re-redistribution is the process of removing it so that the solution is not modified by invalid regions

$$\begin{array}{c} U^l_{\boldsymbol{v}'} -= \delta M^l_{\boldsymbol{v}} w_{\boldsymbol{v},\boldsymbol{v}'} \\ \boldsymbol{v} \in ind^{-1}(C_{N_{ref}}(\Omega^{l+1})) \end{array} \tag{27}$$

6 Refluxing

First we describe the refluxing algorithm which, along with redistribution, preserves conservation at coarse-fine interfaces. The standard refluxing algorithm Given a level vector field F on Ω, we define a discrete divergence operator D as follows:

$$\begin{array}{c} \kappa_{\boldsymbol{v}}(D \cdot \vec{F}) = \frac{1}{h}(\sum\limits_{d=0}^{D-1}(\sum\limits_{\boldsymbol{f} \in \mathcal{F}^+_d(\boldsymbol{v})} \alpha_{\boldsymbol{f}} \tilde{F}_{\boldsymbol{f}} - \sum\limits_{\boldsymbol{f} \in \mathcal{F}^-_d(\boldsymbol{v})} \alpha_{\boldsymbol{f}} \tilde{F}_{\boldsymbol{f}}) + \alpha^B_{\boldsymbol{v}} F^B_{\boldsymbol{v}}) \\ \tilde{F}_{\boldsymbol{f}} = F_{\boldsymbol{f}} + \sum\limits_{d : d \neq dir(\boldsymbol{f})} |x_{\boldsymbol{f},d}|(F_{\boldsymbol{f} << sign(x_{\boldsymbol{f},d})\boldsymbol{e}^d} - F_{\boldsymbol{f}}), \end{array} \tag{28}$$

where $\kappa_{\boldsymbol{v}}$ is the volume fraction of VoF $\boldsymbol{v}$ and $\alpha_{\boldsymbol{f}}$ is the area fraction of face $\boldsymbol{f}$. Equation 28 consists of a summation of interpolated fluxes and a boundary flux. The flux interpolation is discribed in [JC98]. Let $\vec{F}^{comp} = \{\vec{F}^f, \vec{F}^{c,valid}\}$ be a two-level composite vector field. We want to define a composite divergence $D^{comp}(\vec{F}^f, \vec{F}^{c,valid})_{\boldsymbol{v}}$, for $\boldsymbol{v} \in V^c_{valid}$. We do this by extending $F^{c,valid}$ to the faces adjacent to $\boldsymbol{v} \in V^c_{valid}$, but are covered by $\mathcal{F}^f_{valid}$.

$$\begin{array}{c} < F^f_d >_{\boldsymbol{f}^c} = \frac{1}{(n_{ref})^{(\mathbf{D}-1)} \alpha_{\boldsymbol{f}^c}} \sum\limits_{\boldsymbol{f} \in \mathcal{C}^{-1}_{n_{ref}}(\boldsymbol{f}^c)} \alpha_{\boldsymbol{f}} F^f_d \\ \boldsymbol{f}^c \in ind^{-1}(\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d), \boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d \in \zeta^f_{d,+} \cup \zeta^f_{d,-} \\ \zeta^f_{d,\pm} = \{\boldsymbol{i} \pm \frac{1}{2}\boldsymbol{e}^d : \boldsymbol{i} \pm \boldsymbol{e}^d \in \Omega^c_{valid}, \boldsymbol{i} \in \mathcal{C}_{n_{ref}}(\Omega^f)\} \end{array} \tag{29}$$

Then we can define $(D \cdot \vec{F})_{\boldsymbol{v}}, \boldsymbol{v} \in \mathcal{V}^c_{valid}$, using the expression above, with $\tilde{F}_{\boldsymbol{f}} =< F^f_d >$ on faces covered by $\mathcal{F}^f$. We can express the composite divergence in terms of a level divergence, plus a correction. We define a flux register $\delta\vec{F}^f$, associated with the fine level

$$\begin{array}{c} \delta\vec{F}^f = (\delta F^f_{0,\dots} \delta F^f_{D-1}) \\ \delta F^f_d : ind^{-1}(\zeta^f_{d,+} \cup \zeta^f_{d,-}) \to \mathbb{R}^m \end{array} \tag{30}$$

If $\vec{F}^c$ is any coarse level vector field that extends $\vec{F}^{c,valid}$, i.e. $F_d^c = F_d^{c,valid}$ on $\mathcal{F}_{valid}^{c,d}$ then for $\boldsymbol{v} \in \mathcal{V}_{valid}^c$

$$D^{comp}(\vec{F}^f, \vec{F}^{c,valid})_{\boldsymbol{v}} = (D\vec{F}^c)_{\boldsymbol{v}} + D_R(\delta\vec{F}^c)_{\boldsymbol{v}} \tag{31}$$

Here $\delta\vec{F}^f$ is a flux register, set to be

$$\delta F_d^f =< F_d^f > -F_d^c \text{ on } ind^{-1}(\zeta_{d,+}^c \cup \zeta_{d,-}^c) \tag{32}$$

D_R is the reflux divergence operator. For valid coarse vofs adjacent to Ω^f it is given by

$$\kappa_{\boldsymbol{v}}(D_R\delta\vec{F}^f)_{\boldsymbol{v}} = \sum_{d=0}^{D-1}(\sum_{\boldsymbol{f}:\boldsymbol{v}=\boldsymbol{v}^+(\boldsymbol{f})} \delta F_{d,\boldsymbol{f}}^f - \sum_{\boldsymbol{f}:\boldsymbol{v}=\boldsymbol{v}^-(\boldsymbol{f})} \delta F_{d,\boldsymbol{f}}^f) \tag{33}$$

For the remaining vofs in $\mathcal{V}_{valid}^f$,

$$(D_R\delta\vec{F}^f) \equiv 0 \tag{34}$$

We then add the reflux divergence to adjust the coarse solution U^c to preserve conservation.

$$U_{\boldsymbol{v}}^c += \kappa_{\boldsymbol{v}}(D_R(\delta F))_{\boldsymbol{v}} \tag{35}$$

At coarse cells which are also irregular, this leaves unaccounted-for the quantity of mass δM^{Ref} given by

$$\delta M^{Ref} = (1-\kappa_{\boldsymbol{v}})(D_R(\delta F))_{\boldsymbol{v}} \tag{36}$$

This mass must be redistributed to preserve conservation:

$$\delta M_{\boldsymbol{v}}^{Ref,c} = \sum_{\boldsymbol{v}' \in \eta_{\boldsymbol{v}} - C(\mathcal{V}^{l,valid})} \kappa_{\boldsymbol{v}'} w_{\boldsymbol{v},\boldsymbol{v}'} \delta M_{\boldsymbol{v}}^{Ref,c} \tag{37}$$

We increment the solution in the neighboring VoFs with their portion of δM^{Ref}:

$$\begin{gathered} U_{'}^c += \kappa_{\boldsymbol{v}'} w_{\boldsymbol{v},\boldsymbol{v}'} \delta M_{\boldsymbol{v}}^{Ref,c} \\ \boldsymbol{v}' \in \eta_{\boldsymbol{v}} - C(\mathcal{V}^{f,valid}) \end{gathered} \tag{38}$$

Time steps and other factors have been absorbed into the definition of δM. Unfortunately, we are not finished. In equation 38, some of the mass will be going back onto the fine grid

$$\delta M^{RR,c} += \delta M^{Ref} \sum_{\boldsymbol{v}' \in \eta_{\boldsymbol{v}} - \mathcal{V}^{c,valid}} \kappa_{\boldsymbol{v}} w_{\boldsymbol{v},\boldsymbol{v}'} \tag{39}$$

This mass must be accumulated at each fine time step. When the fine level has caught up with the coarse level in time, we adjust the fine solution to account for this mass:

$$\begin{gathered} U_{C^{-1}(\boldsymbol{v}')}^f += w_{\boldsymbol{v},\boldsymbol{v}'} \delta M_{\boldsymbol{v}}^{RR,c} \\ \boldsymbol{v}' \in \eta_{\boldsymbol{v}} - \mathcal{V}^{f,valid} \end{gathered} \tag{40}$$

7 Subcycling in time with embedded boundaries

We use the subcycling-in-time algorithm specified by Berger and Oliger [BO84] to advance an AMR solution in time. Embedded boundary synchronization substantially complicates Berger-Oliger timestepping. Here we present an overview of Berger-Oliger subcycling in time for adaptive mesh refinement in the context of embedded boundaries. Say we are solving the hyperbolic system of equations

$$\frac{\partial U}{\partial t} + \nabla \cdot F = 0 \tag{41}$$

in a domain discretized as described above. Here is an outline of the Berger-Oliger algorithm for this equation. First we perform the steps required to preserve stability and conservation in the presence of embedded boundaries.

- Compute fluxes F^l on $\mathcal{F}$.
- Compute the conservative and non-conservative solution updates ($D^C(F^l)$ and $D^{NCC}(F^l)$).
- Update the solution on the level:

$$U_{\boldsymbol{v}}^{new,l} = U_{\boldsymbol{v}}^{old,l} - \Delta t(\kappa D^{NC}(F^l)_{\boldsymbol{v}} + (1-\kappa) D^C(F^l)_{\boldsymbol{v}}), \quad \boldsymbol{v} \in ind^{-1}(\Omega^l) \tag{42}$$

- Initialize redistribution mass δM^l to be the mass left out in the previous step.

$$\begin{gathered} \delta M_{\boldsymbol{v}}^l = \Delta t \kappa_{\boldsymbol{v}} (1-\kappa_{\boldsymbol{v}})(D^{NC}(F^l)_{\boldsymbol{v}} - D^C(F^l)_{\boldsymbol{v}}) \\ \boldsymbol{v} \in ind^{-1} \mathcal{I}^l \end{gathered} \tag{43}$$

- Perform level redistribution of δM^l:

$$\begin{gathered} U_{\boldsymbol{v}'}^{new,l} \mathrel{+}= w_{\boldsymbol{v},\boldsymbol{v}'} \delta M_v^l \\ \boldsymbol{v}' \in \{\eta_{\boldsymbol{v}} \cap ind^{-1}(\Omega^l)\} \\ \sum_{\boldsymbol{v}' \in \eta_v} w_{\boldsymbol{v},\boldsymbol{v}'} \kappa_{\boldsymbol{v}'} = 1 \end{gathered} \tag{44}$$

Second we perform the steps required to preserve conservation across coarse-fine interfaces. We define δF to be flux registers and $\delta^2 M$ to be redistribution registers.

- We increment the flux register between this level and the next coarser level.

$$\begin{gathered} \delta F_{\boldsymbol{f}}^{l,l-1} \mathrel{+}= < F^l >_{\boldsymbol{f}} \Delta t^l \\ \boldsymbol{f} \in \partial(C(\mathcal{F}^{l-1})) \end{gathered} \tag{45}$$

- We initialize the flux register between this level and the next finer level.

$$\begin{gathered} \delta F_{\boldsymbol{f}}^{l+1,l} = < F^l >_{\boldsymbol{f}} \Delta t^l \\ \boldsymbol{f} \in \partial(\mathcal{F}^{l+1}) \end{gathered} \tag{46}$$

- Increment redistribution registers between this level and the next coarser level.

$$\delta^2 M^{l,l-1}_{\boldsymbol{v}} = \delta M^l_{\boldsymbol{v}} \boldsymbol{v} \in ind^{-1}(\mathcal{I}^l) \tag{47}$$

- Initialize redistribution registers with next finer level and the coarse-coarse ("re-redistribution") registers. for $\boldsymbol{v} \in ind^{-1}(\mathcal{I})^l$

$$\begin{gathered} \delta^2 M^{l,l+1}_{\boldsymbol{v}} = \delta M^l_{\boldsymbol{v}} \\ \delta^2 M^{l,l}_{\boldsymbol{v}} = -\delta M^l_{\boldsymbol{v}} \\ \delta^2 M^{l+1,l}_{\boldsymbol{v}} = 0 \end{gathered} \tag{48}$$

- Advance level $l+1$ solution to time $t^{new,l}$ (requires a minimum of n_{ref} time steps.

- Reflux a portion of the flux difference in equation 46 and save the extra mass into the appropriate redistribution register.

$$\begin{gathered} U^{new,l}_{\boldsymbol{v}} \mathrel{+}= \kappa D_R(\delta F^{l+1})_{\boldsymbol{v}} \\ \delta^2 M^{l,l+1}_{\boldsymbol{v}} \mathrel{+}= \kappa_{\boldsymbol{v}}(1-\kappa_{\boldsymbol{v}}) D_R(\delta F^{l+1})_{\boldsymbol{v}} \\ \delta^3 M^{l,l}_{\boldsymbol{v}} \mathrel{+}= \kappa_{\boldsymbol{v}}(1-\kappa_{\boldsymbol{v}}) D_R(\delta F^{l+1})_{\boldsymbol{v}} \end{gathered} \tag{49}$$

- Redistribute mass that was redistributed (in both directions) across coarse-fine interfaces.

$$\begin{gathered} U^{l+1}_{\boldsymbol{v'_f}} \mathrel{+}= \delta^2 M^{l,l+1}_{\boldsymbol{v_c}} w_{\boldsymbol{v_c},\boldsymbol{v'_c}} \\ \boldsymbol{v_c} \in ind^{-1}(C_{N_{ref}}(G(\Omega^{l+1}, R^r) - \Omega^{l+1})) \\ \boldsymbol{v'_c} = \eta_{\boldsymbol{v_c}} \cap ind^{-1}(C_{N_{ref}}(\Omega^{l+1})) \\ \boldsymbol{v'_f} \in C^{-1}_{N_{ref}}(\boldsymbol{v'_c}) \end{gathered} \tag{50}$$

$$\begin{gathered} U^{l-1}_{\boldsymbol{v'_c}} \mathrel{+}= \frac{\kappa_{\boldsymbol{v'_f}}}{\kappa_{\boldsymbol{v'_c}} N^D_{ref}} \delta^2 M^{l,l-1}_{\boldsymbol{v_f}} w_{\boldsymbol{v_f},\boldsymbol{v'_f}} \\ \boldsymbol{v_f} \in ind^{-1}(\Omega^l - G(\Omega^l, -R^r)), \\ \boldsymbol{v'_f} \in \eta_{\boldsymbol{v_f}} \cap ind^{-1}(C^{-1}_{N_{ref}}(\Omega^{l-1}) - \Omega^l) \\ \boldsymbol{v'_c} = C_{N_{ref}}(\boldsymbol{v'_f}) \end{gathered} \tag{51}$$

- Re-redistribute mass that was redistributed from invalid regions.

$$\begin{gathered} U^l_{\boldsymbol{v'}} \mathrel{-}= \delta^3 M^{l,l}_{\boldsymbol{v}} w_{\boldsymbol{v},\boldsymbol{v'}} \\ \boldsymbol{v} \in ind^{-1}(C_{N_{ref}}(\Omega^{l+1})) \end{gathered} \tag{52}$$

- Finally average down the finer solution where appropriate

$$U^{new,l}_{\boldsymbol{v}} =< U^{new,l+1} >, \quad \boldsymbol{v} \in ind^{-1} C_{N_{ref}}(\Omega^l + 1) \tag{53}$$

8 EBAMRTools User Interface

This section describes the various classes which implement the various algorithms described in the above section.

8.1 Class `EBCoarseAverage`

The `EBCoarseAverage` class is used to average from finer levels on to coarser levels, or for constructing averaged residuals in multigrid iteration. It averages fine data to coarse in a volume-weighted way (see equation 6). This class uses copying from one layout to another for communication. This class has as data a scratch copy of the data at the coarse level. The averaging operator is blocking due to the copy. The important functions of the `EBCoarseAverage` class are as follows:

-
  ```
  void define(const DisjointBoxLayout& dblFine,
              const DisjointBoxLayout& dblCoar,
              const EBISLayout& ebislFine,
              const EBISLayout& ebislCoar,
              const int& nref,
              const int& nvar);
  ```

  Define the stencils and internal data of the class. This must be called before the average function will work.

  - `dblFine`, `dblCoar`: The fine and coarse layouts of the data.
  - `ebislFine`, `ebislCoar`: The fine and coarse layouts of the geometric description.
  - `nref`: The refinement ratio between the two levels.
  - `nvar`: The number of variables contained in the data at each VoF.

-
  ```
  void average(LevelData<EBCellFAB>& coarData,
               const LevelData<EBCellFAB>& fineData,
               const Interval& variables);
  ```

  Average the fine data onto the coarse data over the intersection of the coarse layout with the coarsened fine layout.

  - `coarData`: The data over the coarse layout.
  - `fineData`: The data over the fine layout. Fine and coarse data must have the same number of variables.
  - `variables`: The variables to average. Those not in this range will be left alone. This range of variables must be in both the coarse and fine data.

8.2 Class `EBPWLFineInterp`

The `EBPWLFineInterp` class is used to interpolate in a piecewise-linear fashion coarse data onto fine layouts (see equation 7). This is primarily a useful class for regridding. It contains stencils and slopes over the coarse level and uses copy for communication. This makes its interpolate function blocking. The important functions of `EBPWLFineInterp` are as follows:

-
  ```
  void define(const DisjointBoxLayout& dblFine,
              const DisjointBoxLayout& dblCoar,
              const EBISLayout& ebislFine,
              const EBISLayout& ebislCoar,
              const Box& domainCoar,
              const int& nref,
              const int& nvar);
  ```

  Define the stencils and internal data of the class. This must be called before the `interpolate` function will work.

  - `dblFine, dblCoar`: The fine and coarse layouts of the data.
  - `ebislFine, ebislCoar`: The fine and coarse layouts of the geometric description.
  - `nref`: The refinement ratio between the two levels.
  - `nvar`: The number of variables contained in the data at each VoF.

-
  ```
  void interpolate(LevelData<EBCellFAB>& fineData,
                   const LevelData<EBCellFAB>& coarData,
                   const Interval& variables);
  ```

  Interpolate the fine data from the coarse data over the intersection of the fine layout with the refined coarse layout.

  - `fineData`: The data over the fine layout.
  - `coarData`: The data over the coarse layout.
  - `variables`: The variables to interpolate. Those not in this range will be left alone. This range of variables must be in both the coarse and fine data.

8.3 Class `EBPWLFillPatch`

Given coarse data at old and new times, during subcycling in time, we need to interpolated ghost data onto a fine data set at a time between the old and new coarse times. The `EBPWLFillPatch` class is used to interpolate fine data over the ghost region that is not covered by other fine grids. Data is simply copied from other fine grids where it is available. Only one layer of ghost cells is filled.

-
  ```
  void define(const DisjointBoxLayout& dblFine,
              const DisjointBoxLayout& dblCoar,
              const EBISLayout& ebislFine,
              const EBISLayout& ebislCoar,
              const Box& domainCoar,
              const int& nref,
              const int& nvar);
  ```

Define the stencils and internal data of the class. This must be called before the `interpolate` function will work.

  - `dblFine, dblCoar`: The fine and coarse layouts of the data.
  - `ebislFine, ebislCoar`: The fine and coarse layouts of the geometric description.
  - `nref`: The refinement ratio between the two levels.
  - `nvar`: The number of variables contained in the data at each VoF.

-
  ```
  void interpolate(LevelData<EBCellFAB>& fineData,
                   const LevelData<EBCellFAB>& coarDataOld,
                   const LevelData<EBCellFAB>& coarDataNew,
                   const Real& coarTimeOld,
                   const Real& coarTimeNew,
                   const Real& fineTime,
                   const Interval& variables);
  ```

  Interpolate the indicated fine data variables from the coarse data on ghost cells which overlay a coarse-fine interface. Copy fine data onto ghost cells where appropriate (using `LevelData::exchange`). Only one layer of ghost cells is filled.

  - `fineData`: The data over the fine layout.
  - `coarDataOld, coarDataNew`: The data over the coarse layout at the old and new times. Fine and coarse data must have the same number of variables.
  - `coarTimeOld, coarTimeNew`: The values of the old and new time of the coarse data. The old time must be smaller than the new time.
  - `fineTime`: The time at which the fine data exists. This time must be between the old and new coarse time.

8.4 Class `RedistStencil`

The `RedistStencil` class holds the stencil at every irregular VoF in a layout. The default weights that the stencil holds are volume weights. The class does allow the flexibility to redefine these weights. The weights correspond to $w_{\boldsymbol{v},\boldsymbol{v}'}$ in equations 37 and 44.

-
  ```
  void define(const DisjointBoxLayout& dbl,
              const EBISLayout& ebisl,
              const Box& domain,
              const int& redistRadius);
  ```

  Define the internals of the `RedistStencil` class.

  - `dbl`: The layout of the data.

  - `ebisl`: The layout of the geometric description.
  - `domain`: The computational domain at this level of refinement.
  - `nvar`: The number of variables contained in the data at each VoF.

-
  ```
  void resetWeights(const LevelData<EBCellFAB>& modifier,
                    const int& ivar)
  ```
  Modify the weights in the stencil by multiplying by the inputs in a normalized way.
  - `weights`: Relative weights at each VoF in the stencil. For instance, if one were to want to set the weighting to be mass weighting then `modifier(v, ivar)` would contain the density at VoF `v`.

-
  ```
  const BaseIVFAB<VoFStencil>&
  operator[] (const DataIndex& datInd) const
  ```
  Returns the redistribution stencil at every irregular point in input `Box` associated with this `DataIndex`.

8.5 Class `EBLevelRedist`

The `EBLevelRedist` class performs mass redistibution in an embedded boundary context. The algorithm for this is described in section 5. At irregular cells in a level described by a union of rectangles, mass to be redistibuted is stored incrementally (one `Box` at a time, with a ghost width equal to the redistribution radius). `EBLevelRedist` is then used to increment a solution by the stored redistribution mass. The redistribution radius is a constant static member of the class. The important functions of `EBLevelRedist` are as follows:

-
  ```
  void define(const DisjointBoxLayout& dbl,
              const EBISLayout& ebisl,
              const Box& domain,
              const int& nvar)
  ```
  Define the internals of the `EBLevelRedist` class. Buffers are made at every irregular cell including ghost buffers at a width of the redistribution radius. Sets values at all buffers to zero.
  - `dbl`: The layout of the data.
  - `ebisl`: The layout of the geometric description.
  - `domain`: The computational domain at this level of refinement.
  - `nvar`: The number of variables contained in the data at each VoF.

-
  ```
  void resetWeights(const LevelData<EBCellFAB>& modifier,
                    const int& ivar)
  ```
  Modify the weights in the stencil by multiplying by the inputs in a normalized way.

  - `weights`: Relative weights at each VoF in the stencil. For instance, if one were to want to set the weighting to be mass weighting then `modifier(v, ivar)` would contain the density at VoF `v`.

-
  ```
  void storeMass(const BaseIVFAB<Real>& massDiff,
                 const DataIndex& datInd,
                 const Interval& variables);
  ```

  Store the input mass difference in the internal buffers of the class by incrementing the buffer with the mass difference.

  - `massDiff`: Conserved values to store in registers.
  - `datInd`: The index of the `Box` in the input `DisjointBoxLayout` to which `massDiff` corresponds].
  - variables: The variables to store. These must fit within zero and the number of variables input to the define function.

-
  ```
  void setToZero();
  ```

  Set the internal buffer to zero.

-
  ```
  void redistribute(LevelData<EBCellFAB>& solution,
                    const Interval& variables);
  ```

  Redistribute the data contained in the internal buffers $U_{\boldsymbol{v}'} \mathrel{+}= w_{\boldsymbol{v},\boldsymbol{v}'} \delta M_{\boldsymbol{v}}$.

  - `solution`: Solution to increment.
  - variables: The variables to increment.

8.6 Class `EBFluxRegister`

The `EBFluxRegister` class performs refluxing in an embedded boundary context. The algorithm for this is described in section 6. The important functions of `EBFluxRegister` are as follows:

-
  ```
  void define(const DisjointBoxLayout& dblFine,
              const DisjointBoxLayout& dblCoar,
              const EBISLayout& ebislFine,
              const EBISLayout& ebislCoar,
              const Box& domainCoar,
              const int& nref,
              const int& nvar);
  ```

  Define the internals of the `EBFluxRegister` class. Buffers are made at every irregular cell including ghost buffers at a width of the redistribution radius. Sets values at all buffers to zero.

  - `dblFine, dblCoar`: The fine and coarse layouts of the data.
  - `ebislFine, ebislCoar`: The fine and coarse layouts of the geometric description.
  - `nref`: The refinement ratio between the two levels.
  - `nvar`: The number of variables contained in the data at each VoF.

- `void setToZero();`

  Set the registers to zero.

-
  ```
  void incrementCoarseRegular(
                  const EBFaceFAB& coarseFlux,
                  const Real& scale,
                  const DataIndex& coarsePatchIndex,
                  const Interval& variables,
                  const int& dir);
  void incrementCoarseIrregular(
                  const BaseIFFAB<Real>& coarseFlux,
                  const Real& scale,
                  const DataIndex& coarsePatchIndex,
                  const Interval& variables,
                  const int& dir);
  ```

  Increments the register with data from `coarseFlux`, multiplied by `scale` (α): δF^f_d +=αF^c_d, for all of the d-faces where the input flux (defined on a single rectangle) coincide with the d-faces on which the flux register is defined. `coarseFlux` contains fluxes in the `dir` direction for the grid `dblCoar[coarsePatchIndex]`. Only the registers corresponding to the low faces of `dblCoarse[coarsePatchIndex]` in the `dir` direction are incremented (this avoids double-counting at coarse-coarse interfaces. of the flux register.

  - `coarseFlux` : Flux to put into the flux register. This is not `const` because its box is shifted back and forth - no net change occurs.
  - `scale` : Factor by which to multiply `coarseFlux` in flux register.
  - `coarsePatchIndex` : Index which corresponds to the box in the coarse solution from which `coarseFlux` was calculated.
  - `variables` : The components to put into the flux register.
  - `dir` : Direction of the faces upon which the fluxes live.

-
  ```
  void incrementFineRegular(
                  const EBFaceFAB& fineFlux,
                  const Real& scale,
                  const DataIndex& finePatchIndex,
                  const Interval& variables,
  ```

```
                const int& dir,
                const Side::LoHiSide& sd);
void incrementFineIrregular(
                const BaseIFFAB<Real>& fineFlux,
                const Real& scale,
                const DataIndex& finePatchIndex,
                const Interval& variables,
                const int& dir,
                const Side::LoHiSide& sd);
```

Increments the register with the average over each face of data from `fineFlux`, scaled by `scale` (α): $\delta F^f_d \mathrel{+}= \alpha < F^f_d >$, for all of the d-faces where the input flux (defined on a single rectangle) cover the d-faces on which the flux register is defined. `fineFlux` contains fluxes in the `dir` direction for the grid `dbl[finePatchIndex]`. Only the register corresponding to the direction `dir` and the side `sd` is initialized. `srcInterval` and `dstInterval` are as above.

- `fineFlux` : Flux to put into the flux register. This is not `const` because its box is shifted back and forth - no net change occurs.
- `scale` : Factor by which to multiply `fineFlux` in flux register.
- `finePatchIndex` : Index which corresponds to which box in the `LevelData<FArrayBox>` solution from which `fineFlux` was calculated.
- `variables` : The `Interval` of components of the flux register into which the flux data is put.
- `dir` : Direction of faces upon which fluxes live.
- `sd` : Side of the fine face where coarse-fine interface lies.

• ```
  void reflux(LevelData<EBCellFAB>& uCoarse,
              const Interval& variables,
              const Real& scale);
  ```

  Increments `uCoarse` with the reflux divergence of the contents of the flux register, scaled by `scale` ($\alpha$): $U^c \mathrel{+}= \alpha D_R(\delta \vec{F})$.

  - `uCoarse` : The solution that gets modified by refluxing.
  - `variables`: gives the `Interval` of components of the flux register that correspond to the components of `uCoarse`.
  - `scale` : Factor by which to scale the flux register.

• ```
  void incrementRedistRegister(EBCoarToFineRedist& register,
                               const Interval& variables);
  ```

  Increments redistribution register with left-over mass from reflux divergence as in equation 49: $\delta^2 M^{l,l+1}_{\boldsymbol{v}} \mathrel{+}= \kappa_{\boldsymbol{v}}(1-\kappa_{\boldsymbol{v}}) D_R(\delta F^{l+1})_{\boldsymbol{v}}$.

  - `register`: Coarse to fine register that must be incremented ($\delta^2 M^{l,l+1}$).
  - `variables`: Array indicies to be incremented.

8.7 Class `EBCoarToFineRedist`

The `EBCoarToFineRedist` class stores and redistributes mass that must move from the coarse solution to the fine solution The important functions of `EBCoarToFineRedist` are as follows:

-
  ```
  void define(const DisjointBoxLayout& dblFine,
              const DisjointBoxLayout& dblCoar,
              const EBISLayout& ebislFine,
              const EBISLayout& ebislCoar,
              const Box& domainCoar,
              const int& nref,
              const int& nvar);
  ```

  Define the internals of the class.

  - `dblFine`, `dblCoar`: The fine and coarse layouts of the data.
  - `ebislFine`, `ebislCoar`: The fine and coarse layouts of the geometric description.
  - `nref`: The refinement ratio between the two levels.
  - `nvar`: The number of variables contained in the data at each VoF.
  - `weightModifier`: Multiplier to stencil weights (density if you want mass weighting). If this is NULL, use volume weights.
  - `weightModVar` Variable number of weight modifier.

-
  ```
  void resetWeights(const LevelData<EBCellFAB>& modifier,
                    const int& ivar)
  ```

  Modify the weights in the stencil by multiplying by the inputs in a normalized way.

  - `weights`: Relative weights at each VoF in the stencil. For instance, if one were to want to set the weighting to be mass weighting then `modifier(v, ivar)` would contain the density at VoF `v`.

-
  ```
  void setToZero();
  ```

  Set the registers to zero.

-
  ```
  void increment(BaseIVFAB<Real>& coarMass,
                 const DataIndex& coarPatchIndex,
                 const Interval& variables);
  ```

  Increment the registers by the mass difference in coarMass as shown in the second part equation 49.

  - `coarMass`: The mass difference to add to the register.
  - `coarPatchIndex`: The index to the box on the coarse grid.
  - `variables`: The variables in the register to increment.

-
  ```
  void redistribute(LevelData<EBCellFAB>& fineSolution,
                    const Interval& variables);
  ```
  Redistribute the data contained in the internal buffers $U^{new,l+1}_{\boldsymbol{v}^f} {+}{=} w_{\boldsymbol{v},\boldsymbol{v}'} \delta^2 M^{l,l+1}_{\boldsymbol{v}},\ \boldsymbol{v}^f \in C^{-1}_{nref}(\boldsymbol{v})$

  - `fineSolution`: Solution to increment.
  - variables: The variables to increment.

8.8 Class `EBFineToCoarRedist`

The `EBFineToCoarToRedist` class stores and redistributes mass that must go from the fine to the coarse grid. The important functions of `EBFineToCoarRedist` are as follows:

-
  ```
  void define(const DisjointBoxLayout& dblFine,
              const DisjointBoxLayout& dblCoar,
              const EBISLayout& ebislFine,
              const EBISLayout& ebislCoar,
              const Box& domainCoar,
              const int& nref,
              const int& nvar);
  ```
  Define the internals of the class.

  - `dblFine, dblCoar`: The fine and coarse layouts of the data.
  - `ebislFine, ebislCoar`: The fine and coarse layouts of the geometric description.
  - `nref`: The refinement ratio between the two levels.
  - `nvar`: The number of variables contained in the data at each VoF.
  - `weightModifier`: Multiplier to stencil weights (density if you want mass weighting). If this is NULL, use volume weights.
  - `weightModVar` Variable number of weight modifier.

-
  ```
  void resetWeights(const LevelData<EBCellFAB>& modifier,
                    const int& ivar)
  ```
  Modify the weights in the stencil by multiplying by the inputs in a normalized way.

  - `weights`: Relative weights at each VoF in the stencil. For instance, if one were to want to set the weighting to be mass weighting then `modifier(v, ivar)` would contain the density at VoF `v`.

- `void setToZero();`

  Set the registers to zero.

-
  ```
  void increment(BaseIVFAB<Real>& fineMass,
                 const DataIndex& finePatchIndex,
                 const Interval& variables);
  ```

  Increment the registers by the mass difference in fineMass as shown in equation 49.

  - `fineMass`: The mass difference to add to the register.
  - `finePatchIndex`: The index to the box on the fine grid.
  - `variables`: The variables in the register to increment.

-
  ```
  void redistribute(LevelData<EBCellFAB>& coarSolution,
                    const Interval& variables);
  ```

  Redistribute the data contained in the internal buffers $U^{new,l}_{\boldsymbol{v}'} += w^{fc}_{\boldsymbol{v},\boldsymbol{v}'}\delta^2 M^{l+1,l}_{\boldsymbol{v}}$

  - `fineSolution`: Solution to increment.
  - variables: The variables to increment.

8.9 Class `EBCoarToCoarRedist`

The `EBCoarToCoarToRedist` class stores and redistributes mass that was redistributed to the coarse grid that is covered by the fine grid and now must be corrected. This is the notorious "re-redistribution" process. The important functions of `EBCoarToCoarRedist` are as follows:

-
  ```
  void define(const DisjointBoxLayout& dblFine,
              const DisjointBoxLayout& dblCoar,
              const EBISLayout& ebislFine,
              const EBISLayout& ebislCoar,
              const Box& domainCoar,
              const int& nref,
              const int& nvar);
  ```

  Define the internals of the class.

  - `dblFine`, `dblCoar`: The fine and coarse layouts of the data.
  - `ebislFine`, `ebislCoar`: The fine and coarse layouts of the geometric description.
  - `nref`: The refinement ratio between the two levels.
  - `nvar`: The number of variables contained in the data at each VoF.

-
  ```
  void resetWeights(const LevelData<EBCellFAB>& modifier,
                    const int& ivar)
  ```
  Modify the weights in the stencil by multiplying by the inputs in a normalized way.
  - `weights`: Relative weights at each VoF in the stencil. For instance, if one were to want to set the weighting to be mass weighting then `modifier(v, ivar)` would contain the density at VoF `v`.

-
  ```
  void setToZero();
  ```
  Set the registers to zero.

-
  ```
  void increment(BaseIVFAB<Real>& coarMass,
                 const DataIndex& finePatchIndex,
                 const Interval& variables);
  ```
  Increment the registers by the mass difference in coarMass as shown in equation 49.
  - `coarMass`: The mass difference to add to the register.
  - `coarPatchIndex`: The index to the box on the fine grid.
  - `variables`: The variables in the register to increment.

-
  ```
  void redistribute(LevelData<EBCellFAB>& coarSolution,
                    const Interval& variables);
  ```
  Redistribute the data contained in the internal buffers $U^{new,l}_{\boldsymbol{v}'} \mathrel{+}= w_{\boldsymbol{v},\boldsymbol{v}'}\delta^2 M^{l,l}_{\boldsymbol{v}}$
  - `coarSolution`: Solution to increment.
  - variables: The variables to increment.

References

[BO84] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.*, 53:484–512, March 1984.

[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.

[JC98] Hans Johansen and Phillip Colella. A cartesian grid embedded boundary method for Poisson's eqaution on irregular domains. *J. Comput. Phys.*, 1998.