EBAMRTools: EBChombo's Adaptive Refinement Library

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

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

June 28, 2001

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.

- Redistribute 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 [2]. 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 right 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,

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

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

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

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).

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

(7)

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

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

(8)

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

(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 interpolate 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 $ind^{-1}(\Omega^c)$.

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

(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 $\tilde{\varphi}^c$.

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

(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

Away from locations where the embedded boundary crosses the coarse-fine interface, we use the algorithm in [2].

To proceed from here we need to define the corner ghost cells region. Say we have two levels of refinement. Define Ω^f as that region covered by the finer level. Define Λ^f to be the problem domain at the finest refinement. Define G^d to be the grow operation that only grows a region in the coordinate direction d. Define the grow operator G to be the operator which grows a region by one cell in all coordinate directions. In three dimensions, this is

$$G(\Omega, 1) \equiv G^1(G^2(G^3(\Omega, 1), 1), 1)$$

The coarse-fine layer of ghost Ω^{cf} cells is defined to be

$$\Omega^{cf,f} = (G(\Omega^f, 1) - \Omega^f) \cap \Lambda^f$$

The ghost cells which are not corners Ω^e can be obtained by shifting Ω^f along coordinate directions:

$$\Omega e, f = (\bigcup_{d=1}^{D} G^d(\Omega^f, 1) - \Omega^f) \cap \Lambda^f$$

The corner ghost cells Ω^p are defined to be

$$\Omega^p = \Omega^{cf} - \Omega^e$$

Define C to be the pointwise coarsening operation and r to be the refinement ratio. We define the coarse-fine interface set on the coarse level $\Omega^{cf,c}$ to be the coarse cells which underly the fine ghost cells.

$$\Omega^{cf,c} = C(\Omega^{cf,f}, r)$$

Because of proper nesting requirements, we claim that $\Omega^{e,c} = C(\Omega^{e,f}, r)$ and $\Omega^{p,c} = C(\Omega^{e,f}, r)$ do not intersect and $\Omega^{cf,c} = \Omega^{e,c} \cap \Omega^{p,c}$.

We use the Johansen stencil for Dirichlet EB boundary conditions away from the coarse-fine boundary. When any VoF of the Johansen stencil is within a ghost cell on the coarse-fine interface, we drop to the least-squares stencil which has a smaller. The least-squares stencil still requires corner ghost cells be filled. In the absence of coarse-fine interfaces intersecting embedded boundaries, we only need to fill ghost cells which were not corners Ω^e. Since we are proposing to allow this intersection, we must interpolate to all of Ω^{cf}.

4.3.1 Interpolation to non-corner ghost cells $\Omega^{e,f}$

Away from points where the coarse-fine interface, we interpolate using QuadCFInterp. It uses one-sided differences to avoid using coarse data under finer data. See the AMRTools section of the Chombo design document for details. Though that is somewhat ideologically inconsistent with the following strategy near embedded boundaries, we feel that, as a proven technology, QuadCFInterp should be left alone.

This section deals only with quadratic interpolation near where the coarse-fine interface crosses the embedded boundary design document. The functional change from QuadCFInterp here is that we only need to do one-sided difference when covered cells are near. Because we are doing higher-order averaging to fill coarse data that is under finer data (see section 7.0.1), we can allow the coarse-fine interpolation stencil to reach under finer grids.

We present the natural extension of the regular grid description to embedded boundaries of quadratic coarse-fine interpolation. For each coordinate direction d, we compute values of ϕ in the set $\Omega_d^{e,f} = (G^{d}(\Omega^f, 1) - \Omega^f) \cap \Lambda^f$. Define the "valid" parts of the domain to be the parts of the domain whose volume fractions are greater than zero.

$$\Omega_i^{c,valid} = \{\boldsymbol{i} : \boldsymbol{i} \in \Omega^c \text{ and } \kappa_i > 0\}$$

To perform this interpolation, we first observe that, given $\boldsymbol{i} \in \tilde{\Omega}_k^f - \Omega^f$, there is a unique choice of $\pm$ and d, such that $\boldsymbol{i} \mp \boldsymbol{e}^d \in \Omega_k^f$. Having specified that choice, the interpolant is constructed in two steps

(i) Interpolation in the direction orthogonal to $\boldsymbol{e}^d$. We compute

$$\boldsymbol{x} = \frac{\boldsymbol{i} + \frac{1}{2}\boldsymbol{u}}{r} - (\boldsymbol{i}^c + \frac{1}{2}\boldsymbol{u})$$

where $\boldsymbol{i}^c = \mathcal{C}_r(\boldsymbol{i})$. The real-valued vector $\boldsymbol{x}$ is the displacement of the cell center $\boldsymbol{i}$ on the fine grid from the cell center at $\boldsymbol{i}^c$ on the coarse grid, scaled by h^c.

$$\hat{\varphi}_{\boldsymbol{i}} = \varphi_{\boldsymbol{i}^c}^c + \sum_{d' \neq d}\left[(x_{d'}(D^{1,d'}\varphi^c)_{\boldsymbol{i}^c} + \frac{1}{2}(x_{d'})^2(D^{2,d'}\varphi^c)_{\boldsymbol{i}^c}) + \sum_{d'' \neq d, d'' \neq d'} x_{d'}x_{d''}(D^{d'd''}\varphi^c)_{\boldsymbol{i}^c}\right]$$

The second sum has only one term if $\mathbf{D} = 3$, and no terms if $\mathbf{D} = 2$.

(ii) Interpolation in the normal direction.

$$\tilde{\varphi}_{\boldsymbol{i}} = I_q^B(\varphi^f, \varphi^{c,valid}) \equiv 4a + 2b + c \ , \ \tilde{x}_d = x_d - \frac{1}{2}(r + 3)1$$

where a, b, c are computed to interpolate between the collinear data

$$((\boldsymbol{i} \pm \frac{1}{2}(n_{ref}^l - 1)\boldsymbol{e}^d)h, \hat{\varphi}_{\boldsymbol{i}}),$$

$$((\boldsymbol{i} \mp \boldsymbol{e}^d)h, \varphi_{\boldsymbol{i}\mp\boldsymbol{e}_d}^l),$$

$$((\boldsymbol{i} \mp 2\boldsymbol{e}^d)h, \varphi_{\boldsymbol{i}\mp 2\boldsymbol{e}_d}^l)$$

In (i), the quantities $D^{1,d'}\varphi^c, D^{2,d'}\varphi^c$ and $D^{d'd''}\varphi^c$ are difference approximations to $\frac{\partial}{\partial x_{d'}}, \frac{\partial^2}{\partial x_{d'}^2}$, and $\frac{\partial^2}{\partial x_{d'}\partial x_{d''}}$, respectively. $D^{1,d}\varphi$ must be accurate to $O(h^2)$, while the other two quantities need only be $O(h)$. The strategy for computing these quantities is to use only values in Ω^c_{valid} to compute these difference approximations. For the case of $D^{1,d'}\varphi, D^{2,d'}\varphi$, we use 3-point stencils, centered if possible, or shifted as required to consist of points on Ω^c_{valid}.

$$(D^{1,d'}\varphi)_{\boldsymbol{i}} = \begin{cases} \frac{1}{2}(\varphi^c_{\boldsymbol{i}+\boldsymbol{e}^{d'}} - \varphi^c_{\boldsymbol{i}-\boldsymbol{e}^{d'}}) & \text{if both } \boldsymbol{i} \pm \boldsymbol{e}^{d'} \in \Omega^c_{valid} \\ \pm\frac{3}{2}(\varphi^c_{\boldsymbol{i}\pm\boldsymbol{e}^{d'}} - \varphi^c_{\boldsymbol{i}}) \mp \frac{1}{2}(\varphi^c_{\boldsymbol{i}\pm2\boldsymbol{e}^{d'}} - \varphi^c_{\boldsymbol{i}\pm\boldsymbol{e}^{d'}}) & \text{if } \boldsymbol{i} \pm \boldsymbol{e}^{d'} \in \Omega^c_{valid}, \ \boldsymbol{i} \mp \boldsymbol{e}^{d'} \notin \Omega^c_{valid} \\ 0 & \text{otherwise} \end{cases}$$

$$(D^{2,d'}\varphi)_{\boldsymbol{i}} = \begin{cases} \varphi^c_{\boldsymbol{i}+\boldsymbol{e}^{d'}} - 2\varphi^c_{\boldsymbol{i}} + \varphi^c_{\boldsymbol{i}-\boldsymbol{e}^{d'}} & \text{if both } \boldsymbol{i} \pm \boldsymbol{e}^{d'} \in \Omega^c_{valid} \\ \varphi^c_{\boldsymbol{i}} - 2\varphi^c_{\boldsymbol{i}\pm\boldsymbol{e}^{d'}} + \varphi^c_{\boldsymbol{i}\pm2\boldsymbol{e}^{d'}} & \text{if } \boldsymbol{i} \pm \boldsymbol{e}^{d'} \in \Omega^c_{valid}, \ \boldsymbol{i} \mp \boldsymbol{e}^{d'} \notin \Omega^c_{valid} \\ 0 & \text{otherwise} \end{cases}$$
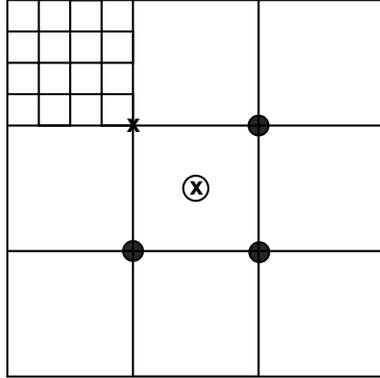
Figure 1: Mixed-derivative approximation illustration. The upper-left corner is covered by a finer level so the mixed derivative in the upper left (the uncircled x) has a stencil which extends into the finer level. We therefore average the mixed derivatives centered on the other corners (the filled circles) to approximate the mixed derivatives for coarse-fine interpolation in three dimensions.

In the case of $D^{d'd''}\varphi^c$, we use an average of all of the four-point difference approximations $\frac{\partial^2}{\partial x_{d'}\partial x_{d''}}$ centered at d', d'' corners adjacent to $\boldsymbol{i}$ such that all four points in the stencil are in Ω^c_{valid} (Figure 1)

$$(D^{d'd''}_{corner}\varphi^c)_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^{d'}+\frac{1}{2}\boldsymbol{e}^{d''}} = \begin{cases} \frac{1}{h^2}(\varphi_{\boldsymbol{i}+\boldsymbol{e}^{d'}+\boldsymbol{e}^{d''}} + \varphi_{\boldsymbol{i}} - \varphi_{\boldsymbol{i}+\boldsymbol{e}^{d'}} - \varphi_{\boldsymbol{i}+\boldsymbol{e}^{d''}}) & \text{if } [\boldsymbol{i}, \boldsymbol{i} + \boldsymbol{e}^{d'} + \boldsymbol{e}^{d''}] \subset \Omega^c_{valid} \\ 0 & \text{otherwise} \end{cases}$$

7

$$(D^{2,d'd''}\varphi^c)_{\boldsymbol{i}} = \begin{cases} \frac{1}{N_{valid}} \sum_{s'=\pm1} \sum_{s''=\pm1} (D^{d'd''}\varphi^c)_{\boldsymbol{i}+\frac{1}{2}s'\boldsymbol{e}^{d'}+\frac{1}{2}s''\boldsymbol{e}^{d''}} & \text{if } N_{valid} > 0 \\ 0 & \text{otherwise} \end{cases}$$

where N_{valid} is the number of nonzero summands. To compute (ii), we need to compute the interpolation coefficients a b, and c.

$$a = \frac{\hat{\varphi} - (r \cdot |x_d| + 2)\varphi_{\boldsymbol{i}\mp\boldsymbol{e}^d} + (r \cdot |x_d| + 1)\varphi_{\boldsymbol{i}\mp2\boldsymbol{e}^d}}{(r \cdot |x_d| + 2)(r \cdot |x_d| + 1)}$$
$$b = \varphi_{\boldsymbol{i}\mp\boldsymbol{e}^d} - \varphi_{\boldsymbol{i}\mp2\boldsymbol{e}^d} - a$$
$$c = \varphi_{\boldsymbol{i}\mp2\boldsymbol{e}^d}$$

4.3.2 Interpolation to corner ghost cells

We now discuss how we fill data on $\Omega^{p,f}$ the ghost cells over the coarse fine interface which cannot be reached from a single move in a coordinate direction. Define $\mathcal{D}$ to be the set of directions which have the requisite number of uncovered, single-valued cells from a corner cell $\boldsymbol{i}$. It is clear from the location of the corner which direction one needs to extrapolate from.

$$\mathcal{D}_{\boldsymbol{i}} = \{d : \kappa_{\boldsymbol{i}\pm\boldsymbol{e}} > 0 \text{ and } \kappa_{\boldsymbol{i}\pm2\boldsymbol{e}} > 0 \text{ and } \kappa_{\boldsymbol{i}\pm3\boldsymbol{e}} > 0\}$$

We also exclude from $\mathcal{D}$ any directions with a multivalued cell in the stencil. We define N_D to be the number of directions contained in $\mathcal{D}$.

$$\phi_{\boldsymbol{i}} = \frac{1}{N_D} \sum_{d\in\mathcal{D}} 3(\phi_{\boldsymbol{i}\pm\boldsymbol{e}^d} - \phi_{\boldsymbol{i}\pm2\boldsymbol{e}^d}) + \phi_{\boldsymbol{i}\pm3\boldsymbol{e}^d}$$

We must exercise some care here to ensure that our algorithm is independent of how we divide our region into rectangles. For this reason, after we do the above extrapolation, we do a cornerCopier exchange operation to fill corner cells that are covered by ghost cells of a neighboring fine grid. Finally we do an ordinary exchange operation to fill any ghost cells which are covered by the valid fine grid.

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 $\boldsymbol{v}'$

$$U_{\boldsymbol{v}'}^l \mathrel{+}= w_{\boldsymbol{v},\boldsymbol{v}'} \delta M_{\boldsymbol{v}}^l. \tag{15}$$

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 $\delta M_{\boldsymbol{v}}^l$, $\boldsymbol{v} \in ind^{-1}\Omega^l$, 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 $C_{N_{ref}}^{-1}$. 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,

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

$\delta^1 M_{\boldsymbol{v}}^l$ 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,

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

$\delta^I M_{\boldsymbol{v}}^l$ 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 redistribution mass $\delta^2 M^{l,l+1}$ to be

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

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.

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

The change in the fine solution is the given by

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

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 redistribution mass $\delta^2 M^{l,l-1}$ to be

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

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 redistribution mass $\delta^2 M^{l,l+1}$ to be

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

Define $\zeta_{v'_f}^2$ to be the unnormalized mass that goes to VoF v'_f. We distribute this mass to the VoF $v'_c = C_{N_{ref}}(v'_f)$.

$$\zeta_{v'_f}^2 = \zeta_{v'_c}^2 = h_f^D w_{v_f,v'_f} \kappa_{v'_f} \delta M_{v_f}^l \tag{23}$$

We define $\delta U_{v'_c}^{l-1}$ to be the change in the coarse solution density due to $\delta^w M_{v_f,v'_f}$:

$$\delta U_{v'_c}^{l-1} = \frac{\zeta_{v'_f}^2}{\kappa_{v'_c} h_c^D} \tag{24}$$

Substituting from above, we increment the coarse solution as follows

$$\begin{aligned}
U_{v'_c}^{l-1} &\mathrel{+}= \frac{\kappa_{v'_f}}{\kappa_{v'_c} N_{ref}^D} \delta M_{v_f}^l w_{v_f,v'_f} \\
v_f &\in ind^{-1}(\Omega^l - G(\Omega^l, -R^r)), \\
v'_f &\in \eta_{v_f} \cap ind^{-1}(C_{N_{ref}}^{-1}(\Omega^{l-1}) - \Omega^l) \\
v'_c &= C_{N_{ref}}(v'_f)
\end{aligned} \tag{25}$$

5.4 Coarse to Coarse Redistribution

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

$$\begin{aligned}
\delta^3 M_v^{l,l} &= \sum_{v' \in T(v)} \delta M_v^l w_{v,v'} \kappa_{v'} \\
T(v) &= \eta_v \cap ind^{-1}(\Omega^l).
\end{aligned} \tag{26}$$

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

$$\begin{aligned}
U_{v'}^l &\mathrel{-}= \delta M_v^l w_{v,v'} \\
v &\in ind^{-1}(C_{N_{ref}}(\Omega^{l+1}))
\end{aligned} \tag{27}$$

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:

$$\begin{aligned}
\kappa_v(D \cdot \vec{F}) &= \frac{1}{h}\left(\sum_{d=0}^{D-1}\left(\sum_{f \in \mathcal{F}_d^+(v)} \alpha_f \tilde{F}_f - \sum_{f \in \mathcal{F}_d^-(v)} \alpha_f \tilde{F}_f\right) + \alpha_v^B F_v^B\right) \\
\tilde{F}_f &= F_f + \sum_{d:d \neq dir(f)} |x_{f,d}|(F_{f << sign(x_{f,d})e^d} - F_f),
\end{aligned} \tag{28}$$

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 described in [3]. 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 $\boldsymbol{v} \in V_{valid}^c$. We do this by extending $F^{c,valid}$ to the faces adjacent to $\boldsymbol{v} \in V_{valid}^c$, but are covered by $\mathcal{F}_{valid}^f$.

$$
\begin{aligned}
< F_d^f >_{\boldsymbol{f}^c} &= \frac{1}{(n_{ref})^{(\mathbf{D}-1)}} \sum_{\boldsymbol{f} \in \mathcal{C}_{n_{ref}}^{-1}(\boldsymbol{f}^c)} \alpha_{\boldsymbol{f}} F_d^f \\
\boldsymbol{f}^c &\in ind^{-1}(\boldsymbol{i} + \tfrac{1}{2}\boldsymbol{e}^d), \boldsymbol{i} + \tfrac{1}{2}\boldsymbol{e}^d \in \zeta_{d,+}^f \cup \zeta_{d,-}^f \\
\zeta_{d,\pm}^f &= \{\boldsymbol{i} \pm \tfrac{1}{2}\boldsymbol{e}^d : \boldsymbol{i} \pm \boldsymbol{e}^d \in \Omega_{valid}^c, \boldsymbol{i} \in \mathcal{C}_{n_{ref}}(\Omega^f)\}
\end{aligned}
\tag{29}
$$

Then we can define $(D \cdot \vec{F})_{\boldsymbol{v}}, \boldsymbol{v} \in \mathcal{V}_{valid}^c$, using the expression above, with $\tilde{F}_{\boldsymbol{f}} =< F_d^f >$ 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

$$
\begin{aligned}
\delta \vec{F}^f &= (\delta F_{0,...}^f \delta F_{D-1}^f) \\
\delta F_d^f &: ind^{-1}(\zeta_{d,+}^f \cup \zeta_{d,-}^f) \to \mathbb{R}^m
\end{aligned}
\tag{30}
$$

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

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

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

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

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.

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

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:

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

12

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

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

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

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

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:

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

7 Subcycling in time with embedded boundaries

We use the subcycling-in-time algorithm specified by Berger and Oliger [1] 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.

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

13

- Perform level redistribution of δM^l:

$$U^{new,l}_{\boldsymbol{v'}} += w_{\boldsymbol{v},\boldsymbol{v'}} \delta M^l_v$$
$$\boldsymbol{v'} \in \{\eta_{\boldsymbol{v}} \cap ind^{-1}(\Omega^l)\}$$
$$\sum_{\boldsymbol{v'} \in \eta_v} w_{\boldsymbol{v},\boldsymbol{v'}} \kappa_{\boldsymbol{v'}} = 1$$

$$(44)$$

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.

$$\delta F^{l,l-1}_{\boldsymbol{f}} += \; <F^l>_{\boldsymbol{f}} \Delta t^l$$
$$\boldsymbol{f} \in \partial(C(\mathcal{F}^{l-1}))$$

$$(45)$$

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

$$\delta F^{l+1,l}_{\boldsymbol{f}} = \; <F^l>_{\boldsymbol{f}} \Delta t^l$$
$$\boldsymbol{f} \in \partial(\mathcal{F}^{l+1})$$

$$(46)$$

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

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

$$(47)$$

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

$$\delta^2 M^{l,l+1}_{\boldsymbol{v}} = \delta M^l_{\boldsymbol{v}}$$
$$\delta^2 M^{l,l}_{\boldsymbol{v}} = -\delta M^l_{\boldsymbol{v}}$$
$$\delta^2 M^{l+1,l}_{\boldsymbol{v}} = 0$$

$$(48)$$

- 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.

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

$$(49)$$

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

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

$$(50)$$

14

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

(51)

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

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

(52)

- Finally average down the finer solution where appropriate

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

(53)

7.0.1 $O(h^3)$ **Averaging**

The stencil for Dirichlet EB boundary conditions on a coarse level can reach under the fine level. Because of this, we need to average ϕ from the finer level to the coarser level before evaluating $L\phi$. We use a higher-order $(O(h^3))$ averaging operator because we need a more accurate value at a coarse location than averaging the fine values which cover the coarse cell would produce. Martin and Cartwright discuss this in detail. The standard averaging operator is second order accurate and the truncation error analysis works such that to avoid making $O(1)$ errors in the Laplacian on coarse cells near the fine grid, we need a third order estimate of the solution on regions covered by finer grids. We therefore use a modified averaging operator in which we eliminate term of the truncation error of the standard averaging operator. Consider a coarse cell at $\vec{i_c}$. The coarse cell is covered by fine cells and the refinement ratio is two, the fine grid spacing is h_f and the coarse grid spacing is h_c. Suppose we have a smooth function ϕ^e which exists at all points in space. Away from coarse-fine interfaces, the Laplacian is discretized in the standard way. In two dimensions, this discretization is:

$$(L\phi)_{i,j} = \frac{1}{h^2}(\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} - 4\phi_{i,j})$$

(54)

and in three dimensions

$$(L\phi)_{i,j} = \frac{1}{h^2}(\phi_{i+1,j,k} + \phi_{i-1,j,k} + \phi_{i,j+1,k} + \phi_{i,j-1,k} + \phi_{i,j,k+1} + \phi_{i,j,k-1} - 6\phi_{i,j,k})$$

(55)

At the coarse-fine interface, we interpolate values onto ghost cells which surround the union of rectangles that correspond to the level's domain and use equation 54 to calculate the Laplacian. We define the standard averaging operator A_S in two dimensions to be

$$(A_S(\phi^e))(h_c\vec{i_c}) = \frac{1}{4}\left(\begin{array}{c} \phi^e(h_f i_f, h_f j_f)+ \\ \phi^e(h_f(i_f + 1), h_f j_f)+ \\ \phi^e(h_f i_f, h_f(j_f + 1))+ \\ \phi^e(h_f(i_f + 1), h_f(j_f + 1)) \end{array} \right)$$

(56)

and in three dimensions to be

$$(A_S(\phi^e))(h_c\vec{i_c}) = \frac{1}{8}\begin{pmatrix} \phi^e(h_f i_f, h_f j_f, h_f k_f)+ \\ \phi^e(h_f(i_f+1), h_f j_f, h_f k_f)+ \\ \phi^e(h_f i_f, h_f(j_f+1,), h_f k_f)+ \\ \phi^e(h_f(i_f+1), h_f(j_f+1), h_f k_f)+ \\ \phi^e(h_f i_f, h_f j_f, h_f(k_f+1))+ \\ \phi^e(h_f(i_f+1), h_f j_f, h_f(k_f+1))+ \\ \phi^e(h_f i_f, h_f(j_f+1), h_f(k_f+1))+ \\ \phi^e(h_f(i_f+1), h_f(j_f+1), h_f(k_f+1)) \end{pmatrix} \tag{57}$$

where $\vec{i_f} = 2\vec{i_c}$. The truncation error τ of A_S is given by

$$\tau = \phi^e(h_c\vec{i_c}) - (A_S(\phi^e))(h_c\vec{i_c}) = \frac{h_f^2}{2}\nabla^2\phi_e(h_c\vec{i_c}) + O(h_f^3) \tag{58}$$

Away from the embedded boundary, we define the modified averaging operator A_M to be A_S with the leading order in the truncation error subtracted off:

$$(A_M(\phi_f))_{\vec{i_c}} = A_S(\phi_f)_{\vec{i_c}} - \frac{h_f^2}{2}L(\phi_f)_{\vec{i_c}} \tag{59}$$

Near the embedded boundary, we extrapolate to $O(h^2)$ from fine cells to the coarse cell and average the result.

8 EBAMRTools User Interface

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

8.1 Classes `EBCoarseAverage/EBCoarsen`

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. `EBCoarsen` does the same thing with the same interface, but averages to $O(h^3)$ and is not conservative. The important functions of the `EBCoarseAverage/EBCoarsen` classes are as follows:

- `void define(const DisjointBoxLayout& dblFine,`
  `const DisjointBoxLayout& dblCoar,`
  `const EBISLayout& ebislFine,`
  `const EBISLayout& ebislCoar,`

16

```
                    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 $w_{v,v'}$ 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 redistribution 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 redistributed 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 $U_{v'} \mathrel{+}= w_{v,v'} \delta M_v$.

  - – `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 de-
    scription.
  - – `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): $\delta F_d^f += \alpha F_d^c$, 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): $\delta F_d^f += \alpha < F_d^f >$, 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` (α): $U^c += \alpha D_R(\delta \vec{F})$.

  - `uCoarse` : The solution that gets modified by refluxing.

  - `variables`: gives the `Interval` of components of the flux register that cor-
    respond 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: $\delta^2 M_v^{l,l+1} += \kappa_v(1 - \kappa_v)D_R(\delta F^{l+1})_v$.

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

  - `variables`: Array indices 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 $U^{new,l+1}_{\boldsymbol{v}^f} \mathrel{+}= w_{\boldsymbol{v},\boldsymbol{v}'}\delta^2 M^{l,l+1}_{\boldsymbol{v}},\ \ \boldsymbol{v}^f \in C^{-1}_{nref}(\boldsymbol{v})$

  – `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 $U_{\boldsymbol{v}'}^{new,l} \mathrel{+}= w_{\boldsymbol{v},\boldsymbol{v}'}^{fc} \delta^2 M_{\boldsymbol{v}}^{l+1,l}$

  - `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 $U_{\boldsymbol{v'}}^{new,l} += w_{\boldsymbol{v},\boldsymbol{v'}} \delta^2 M_{\boldsymbol{v}}^{l,l}$

  – `coarSolution`: Solution to increment.

  – `variables`: The variables to increment.

8.10 **Class** `EBQuadCFInterp`

This class interpolates to ghost cells over the coarse-fine interface with $O(h^3)$ error.

- `EBQuadCFInterp(const DisjointBoxLayout&      a_dblFine,`
  `const DisjointBoxLayout&      a_dblCoar,`
  `const EBISLayout&             a_ebislFine,`
  `const EBISLayout&             a_ebislCoar,`
  `const ProblemDomain&          a_domainCoar,`
  `const int&                    a_nref,`
  `const int&                    a_nvar,`
  `const LayoutData<IntVectSet>& a_cfivs);`

  Define the interpolation object.

- `void`
  `interpolate(LevelData<EBCellFAB>&       a_fineData,`
  `const LevelData<EBCellFAB>& a_coarData,`
  `const Interval&             a_variables);`

  Interpolate to the ghost cells of the fine data to $O(h^3)$.

References

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

[2] P. Colella, D. T. Graves, N.D. Keen, T. J. Ligocki, D. F. Martin, P.W. McCorquodale, D. Modiano, P.O. Schwartz, T.D. Sternberg, and B. Van Straalen. Chombo Software Package for AMR Applications - Design Document. unpublished, 2000.

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