Incompressible Navier-Stokes Software Design

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

August 8, 2003

Contents

Chapter 1

Introduction

1.1 Purpose, goals, Objectives

The purpose of this project is to provide an adaptive mesh refinement (AMR) code capability for simulating multiphase low-Mach-number fluid dynamics in microgravity environments.

1.2 Statement of Scope

This software is intended to solve the incompressible Navier-Stokes equations using block-structured adaptive mesh refinement (AMR) techniques. This code incorporates refinement in time as well as space, is second-order in time and space, and maintains conservation and approximate freestream preservation at coarse-fine interfaces. The divergence constraint of incompressible flow is approximately maintained in a composite sense across the entire hierarchy of refined grids.

The software is written for use in both serial and parallel environments, using MPI on parallel machines.

User inputs generally consist of a subroutine defining the initial velocity profile, along with an inputs file which defines operating parameters such as number of cells in the domain, maximum number of refinement levels, etc.

Output of the code is generally in the form of some limited screen output (generally indicating the progress of the code) and hdf5 plotfiles, which are in a format accessible by ChomboVis, the Chombo data visualization tool.

Chapter 2

Algorithm Description

2.1 Formulation of the Problem

We are solving the incompressible constant-density Navier-Stokes equations with body forces:

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\vec{u} - \nabla p + \nu \nabla^2 \vec{u} + \vec{F}, \tag{2.1}$$

$$\nabla \cdot \vec{u} = 0, \tag{2.2}$$

$$\vec{u} = 0 \;\text{ on }\; \partial\Omega,$$

where $\vec{u}(\vec{x}, t)$ is the fluid velocity vector $(u, v)^T$, t is the time, and $p(\vec{x}, t)$ is the pressure. We also evolve an auxiliary passively transported scalar Λ, which we use to compute the freestream preservation correction.

$$\frac{\partial \Lambda}{\partial t} + \nabla \cdot (\vec{u}\Lambda) = 0 \tag{2.3}$$

2.2 Projection Formulation

We transform the constrained dynamics problem of equations (2.1) and (2.2) into an initial value problem through the use of the Hodge-Helmholtz decomposition. An arbitrary vector field $\vec{w}$ can be uniquely decomposed into two orthogonal components, one of which is divergence-free, the other the gradient of a scalar:

$$\vec{w} = \vec{w}_d + \nabla\phi$$

$$\nabla \cdot \vec{w}_d = 0$$

$$\Delta\phi = \nabla \cdot \vec{w} \;\text{ on }\; \Omega \tag{2.4}$$

$$\vec{w}_d \cdot \vec{n} = 0, \quad \frac{\partial \phi}{\partial \vec{n}} = \vec{w} \cdot \vec{n} \;\text{ on }\; \partial\Omega$$

$$\int_\Omega \vec{w}_d \cdot \nabla\phi \; d\vec{x} = 0.$$

This decomposition can be expressed in terms of an orthogonal projection $\mathbf{P} : \mathbf{P}(\vec{w}) = \vec{w}_d$, computed by solving (2.4) for $\nabla\phi$ and subtracting to obtain the divergence-free part. Formally,

$$\mathbf{P} = I - \nabla(\Delta)^{-1}\nabla\cdot$$

Using the projection operator, the constrained system (2.1)-(2.2) can be transformed into a pure evolution equation, with the constraint applied to the initial data:

$$\frac{\partial \vec{u}}{\partial t} = \mathbf{P}(-\vec{u}\cdot\nabla\vec{u} + \nu\nabla^2\vec{u} + \vec{F})$$

$$(\nabla\cdot\vec{u})(\cdot, t=0) = 0.$$

Chorin [Cho68] used this formulation as the starting point for a discretization of the incompressible flow equations. Following [BCG89], our algorithm is a predictor-corrector formulation in which we first compute an intermediate velocity field and project it onto the space of vectors which satisfy the divergence constraint. Updates to the scalar Λ are computed using a conservative update.

On a uniform grid with grid spacing h, the velocity $\vec{u}(\vec{x},t)$ is approximated by $\vec{u}_{i,j}(t) \approx \vec{u}(ih, jh, t)$. The scalar field $\Lambda(\vec{x},t)$ is likewise approximated as $\Lambda_{i,j}(t) \approx \Lambda(ih, jh, t)$. Then,

$$\vec{u}(t+\Delta t) = \mathbf{P}\Big(\vec{u}(t) - \Delta t(\vec{u}\cdot\nabla\vec{u})^H + \nu(\nabla^2\vec{u})^H + \vec{F}^H\Big) \tag{2.5}$$

$$\nabla p^H = \frac{1}{\Delta t}(\mathbf{I}-\mathbf{P})\Big(\vec{u}(t) - \Delta t(\vec{u}\cdot\nabla\vec{u})^H + \nu(\Delta\vec{u})^H + \vec{F}^H\Big) \tag{2.6}$$

$$\Lambda(t+\Delta t) = \Lambda(t) - \Delta t\nabla\cdot(\vec{u}\Lambda)^H,$$

where the superscript H indicates centering at the intermediate time $(t+\frac{1}{2}\Delta t)$. Following [Col90], $(\vec{u}\cdot\nabla\vec{u})^H$ and $(\nabla\cdot(\vec{u}\Lambda))^H$ are computed using a second-order upwind method. Eq. (2.5) can also be expressed in terms of the pressure gradient $(\nabla p^H)_{i,j} \approx \nabla p(ih, jh, t+\frac{\Delta t}{2})$, where ∇p^H is computed using (2.6):

$$\vec{u}(t+\Delta t) = \vec{u}(t) - \Delta t(\vec{u}\cdot\nabla\vec{u})^H + \Delta t\nu(\Delta\vec{u})^H + \Delta t\vec{F}^H - \Delta t(\nabla p)^H.$$

2.3 AMR Notation

Following [BC89], our adaptive mesh calculations are performed on a hierarchy of nested, cell-centered grids (Figure 2.1). At each AMR level $\ell = 0, ..., \ell_{max}$, the problem domain is discretized by a uniform grid Γ^ℓ with grid spacing h_ℓ. Level 0 is the coarsest level, while each level $\ell+1$ is a factor $n^\ell_{ref} = \frac{h_\ell}{h_{\ell+1}}$ finer than level ℓ; the refinement ratio n^ℓ_{ref} is an integer. Because refined grids overlay coarser ones, cells on different levels will represent the same geometric region in space. We identify cells at different levels which occupy

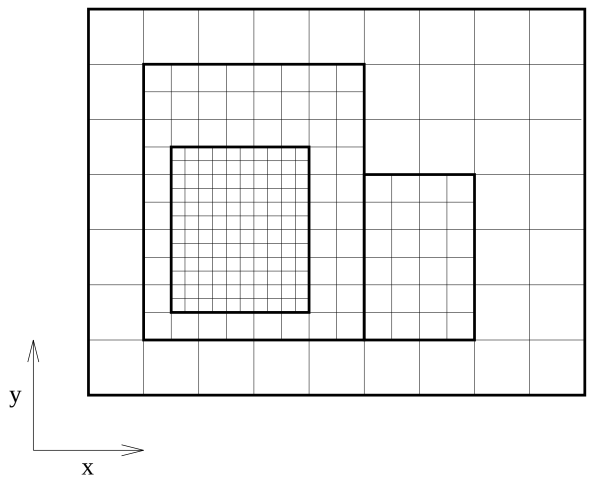

Figure 2.1: Block-structured local refinement. Note that refinement is by an integer factor and is organized into rectangular patches.

the same geometric regions by means of the coarsening operator $\mathcal{C}_r(i,j) = (\lfloor \frac{i}{r} \rfloor, \lfloor \frac{j}{r} \rfloor)$. In that case, $\{\mathcal{C}_r\}^{-1}\{(i,j)\}$ is the set of all cells in a grid r times finer that represent the same geometric region (in a finite volume sense) as the cell (i,j).

In the current implementation, the problem domain is a rectangle, and the refinement ratios are powers of two. Calculations are performed on a hierarchy of meshes $\Omega^\ell \subset \Gamma^\ell$, with $\Omega^\ell \supset \mathcal{C}_{n^\ell_{ref}}(\Omega^{\ell+1})$. Ω^ℓ is the union of rectangular patches (grids) with spacing h_ℓ; the block-structured nature of refinement is used in the implementation to simplify computations on the hierarchy of meshes. On the coarsest level, $\Omega^0 = \Gamma^0$. A cell on a level is either completely covered by cells at the next finer level, or it is not refined at all. Since we assume the solution on finer grids is more accurate, we distinguish between *valid* and *invalid* regions on each level. The valid region on a level is not covered by finer grid cells: $\Omega^\ell_{valid} = \Omega^\ell - \mathcal{C}_{n^\ell_{ref}}(\Omega^{\ell+1})$. The grids on each level satisfy a *proper nesting* condition [BC89]: no cell at level $\ell+1$ represents a geometric region adjacent to one represented by a valid cell at level $\ell-1$.

Likewise, $\Omega^{\ell,*}$ denotes the cell faces of level ℓ cells, while $\Omega^{\ell,*}_{valid}$ refers to the cell faces on level ℓ not covered by level $\ell+1$ faces. Note that the coarse-fine interface $\partial\Omega^{\ell+1,*}$ between levels ℓ and $\ell+1$ is considered to be valid on level $\ell+1$, but not on level ℓ. The coarsening operator also extends to faces: $\mathcal{C}_{n^\ell_{ref}}(\Omega^{\ell+1,*})$ is the set of level ℓ faces covered by level $\ell+1$ faces.

A *composite variable* is defined on the union of valid regions of all levels. Since we organize computation on a level-by-level basis, the invalid regions of each level also contain data, usually an approximation to the valid solution. A *level variable* is defined on the entire level Ω^ℓ (not just the valid region). For a cell-centered variable ϕ, the level variable ϕ^ℓ is defined on all of Ω^ℓ; the composite variable ϕ^{comp} is defined on the union of valid regions over all levels. We also define composite and level-based *vector fields*,

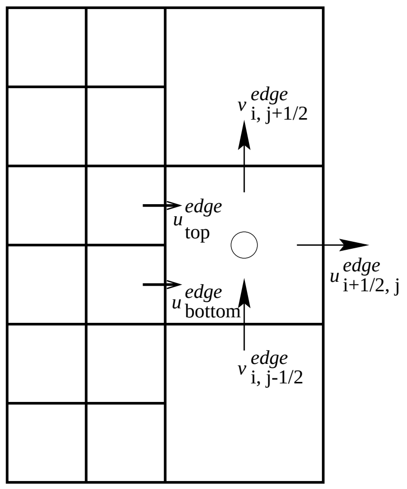

Figure 2.2: Sample two-dimensional coarse-fine interface with an face-centered vector field. Cell (i,j) (open circle) is to the right of the coarse-fine interface.

which are defined at normal cell faces. Like other face-centered variables, a composite vector field $\vec{u}^{face,comp}$ is valid on all faces not overlain by finer faces (Fig 2.2). Likewise, we define composite and level operators which operate on composite and level variables, respectively.

It is also necessary to transfer information from finer grids to coarser ones. We define $\langle \phi^{\ell+1} \rangle$ to be the appropriate cell-centered or face-centered arithmetic average of level $\ell+1$ data $\phi^{\ell+1}$ to the underlying coarser cells in level ℓ.

Divergence, Flux Registers, and Reflux-Divergence

The basic multilevel divergence is a cell-centered divergence of an face-centered vector field. If none of the faces of cell (i,j) are coarse-fine interfaces, we use a centered-difference divergence:

$$(D^{comp,\ell}\vec{u}^{face})_{\boldsymbol{i}} = \sum_{d=0}^{\mathbf{D}-1} \frac{u^{face}_{\boldsymbol{i}+\boldsymbol{e}^d,d} - u^{face}_{\boldsymbol{i}-\boldsymbol{e}^d,d}}{h_\ell}. \tag{2.7}$$

On the fine side of a coarse-fine interface, the stencil is unchanged, since the coarse-fine interface with the coarser level $\ell-1$ is a valid face in level ℓ. For cells on the coarse side of a coarse-fine interface, the coarse-grid vector on the coarse-fine interface is the average of the fine-grid vectors on that face. For the two-dimensional coarse-grid cell in Figure 2.2, the divergence operator is:

$$(D^{comp,\ell}\vec{u}^{face})_{i,j} = \frac{u^{face,\ell}_{i+\frac{1}{2},j} - \langle u^{face,\ell+1} \rangle_{i-\frac{1}{2},j}}{h_\ell} + \frac{v^{face,\ell}_{i,j+\frac{1}{2}} - v^{face,\ell}_{i,j-\frac{1}{2}}}{h_\ell}. \tag{2.8}$$

The level-operator divergence D^ℓ of a level variable $\vec{u}^{face,\ell}$ is defined by ignoring any finer levels and computing D^ℓ everywhere in Ω^ℓ using (2.7). Since the composite divergence on level ℓ depends on both level ℓ and level $\ell+1$ data, it may be written as $D^{comp,\ell}(\vec{u}^{face,\ell}, \vec{u}^{face,\ell+1})$; the level operator only depends on level ℓ data: $D^\ell(\vec{u}^{face,\ell})$.

Assume that the vector field $\vec{u}^{face,\ell}$ can be extended to all faces in $\Omega^{\ell,*}$, including those covered by the coarse-fine interface face $\partial\Omega^{\ell+1,*}$. The composite divergence $D^{comp}\vec{u}^{face,comp}$ on Ω^ℓ may then be expressed as the level-operator divergence D^ℓ along with a correction for the effects of the finer level $(\ell+1)$. To do this efficiently, we define a *flux register* $\delta\vec{u}^{\ell+1}$ defined on $\mathcal{C}_{n^\ell_{ref}}(\partial\Omega^{\ell+1,*})$, which stores the difference in the face-centered quantity $\vec{u}^{face}$ on the coarse-fine interface between levels ℓ and $\ell+1$. Notationally, $\delta\vec{u}^{\ell+1}$ belongs to the fine level $(\ell+1)$ because it represents information on $\partial\Omega^{\ell+1,*}$. However, it has coarse-level (ℓ) grid spacing and indexing.

We define the *reflux divergence* D^ℓ_R to be the D^ℓ stencil as applied to the face-centered vectors on the coarse-fine interface with level $\ell+1$; the general composite operator can then be expressed as:

$$(D^{comp,\ell}\vec{u}^{face})_{\boldsymbol{i}} = (D^\ell\vec{u}^{face,\ell})_{\boldsymbol{i}} + D^\ell_R(\delta\vec{u}^{\ell+1})_{\boldsymbol{i}},$$

$$\delta\vec{u}^{\ell+1} = \langle\vec{u}^{face,\ell+1}\rangle - \vec{u}^{face,\ell} \quad \text{on} \;\; \mathcal{C}_{n^\ell_{ref}}(\partial\Omega^{\ell+1,*}).$$

For the level ℓ cell $(\boldsymbol{i})$, D^ℓ_R can be defined as:

$$D^\ell_R(\delta\vec{u}^{\ell+1})_{\boldsymbol{i}} = \frac{1}{h_\ell}\sum_p \pm(\delta\vec{u}^\ell)_p,$$

where the sum is over the set of all faces of cell $\boldsymbol{i}$ which are also coarse-fine interfaces with level $\ell+1$, and the $\pm$ is $+$ if the face p is on the high side of cell $\boldsymbol{i}$, and - if p is on the low side. Note that D^ℓ_R only affects the set of level ℓ cells immediately adjacent to the coarse-fine interface with level $\ell+1$.

Gradient and Coarse-Fine Interpolation

The gradient is a face-centered, centered-difference gradient of a cell-centered variable ϕ. $G^{comp}\phi$ is a composite vector field, defined on all valid faces in the multilevel domain. On faces which are not coarse-fine interfaces,

$$\begin{aligned} G^{comp,\ell}(\phi)^x_{i+\frac{1}{2},j,k} &= \frac{\phi_{i+1,j,k}-\phi_{i,j,k}}{h_\ell} \\ G^{comp,\ell}(\phi)^y_{i,j+\frac{1}{2},k} &= \frac{\phi_{i,j+1,k}-\phi_{i,j,k}}{h_\ell}. \\ G^{comp,\ell}(\phi)^z_{i,j,k+\frac{1}{2}} &= \frac{\phi_{i,j,k+1}-\phi_{i,j,k}}{h_\ell}. \end{aligned} \tag{2.9}$$

To compute $G^{comp}\phi$ at a coarse-fine interface, we interpolate values for ϕ using both coarse- and fine-level values. We use a quadratic interpolation similar to that used in

[Min94] and adopted by [MC96, Min96, ABC$^+$98] to compute ϕ^I. For details of the quadratic coarse-fine interpolation, refer to the Chombo documentation, particularly for the class `QuadCFInterp`.

The level-operator gradient G^ℓ is defined by extending G^{comp} (which is only defined on $\Omega^{\ell,*}_{valid}$) to all faces in $\Omega^{\ell,*}$. Away from $\partial\Omega^{\ell,*}$ we use the grid-interior stencil (2.9), while on interfaces with a coarser level $\ell - 1$, the interpolation operator $I(\phi^\ell, \phi^{\ell-1})$ is used to compute ghost cell values to be used in (2.9).

The composite gradient on level ℓ, $G^{comp,\ell}$, is dependent on level ℓ and coarse-level $(\ell - 1)$ data: $G^{comp,\ell}(\phi^\ell, \phi^{\ell-1})$. Likewise, the level-operator gradient can be written $G^\ell(\phi^\ell, \phi^{\ell-1})$.

Laplacian

The Laplacian is defined as the divergence of the gradient:

$$L^{comp}\phi^{comp} = D^{comp}G^{comp}\phi^{comp} \tag{2.10}$$

$$L^\ell\phi^\ell = D^\ell G^\ell \phi^\ell. \tag{2.11}$$

On the interiors of grids, (2.10) and (2.11) reduce to the normal five-point second-order discrete Laplacian. On the fine side of a coarse-fine interface, the interpolation operator I fills ghost-cell values which are used in the five-point stencil. On the coarse side of a coarse-fine interface, (2.10) becomes:

$$L^{comp,\ell}\phi = L^\ell\phi^\ell + D^\ell_R(\delta G\phi^{\ell+1})$$

$$\delta G\phi^{\ell+1} = \langle G^{\ell+1}\phi\rangle - G^\ell\phi^\ell.$$

2.4 AMR timestepping

The algorithm is structured as a series of recursive updates on a single refinement level. The basic steps when updating level ℓ are:

1. Perform single-level update on level ℓ, including application of a level ℓ projection to the velocities to ensure that they are approximately divergence-free (π^ℓ is the approximation to the pressure computed using the level projection),

2. Initialize/update flux registers with coarse-fine interface information,

3. Recursive calls to update finer levels n_{ref} times with $\Delta t^{\ell+1} = \frac{\Delta t^\ell}{n^\ell_{ref}}$,

4. Synchronize composite multilevel solution.

At the beginning of a coarsest-level ($\ell = 0$) update, all levels in the AMR hierarchy are at the same time t^0. At the end of the recursive timestep for level 0, all levels in the AMR hierarchy have been advanced to time $t^0 + \Delta t^0$.

2.5 Multilevel Algorithm

In this section, we describe the complete recursive algorithm used to advance the level ℓ solution from time t^ℓ to time $t^\ell + \Delta t^\ell$. Implicit in this recursive algorithm is the subcycled advance of all finer levels to time $t^\ell + \Delta t^\ell$, including all necessary synchronization operations.

2.5.1 Variables

We start the level ℓ advance with the solution at time t^ℓ, which includes the velocity field $\vec{u}^\ell(\vec{x}, t^\ell) = (u^\ell, v^\ell)^T$, the freestream preservation scalar $\Lambda^\ell(\vec{x}, t^\ell)$, and the staggered-grid freestream preservation correction $\vec{u}_p$ from the most recent synchronization step, which has been extended to the invalid regions on level ℓ with $\langle \vec{u}_p^{\ell+1} \rangle$.

We also need flux registers to contain coarse-fine matching information. $\delta\vec{V}^\ell$ contains the normal and tangential (to the coarse-fine interface) momentum fluxes across the coarse-fine interface between level ℓ and the coarser level $\delta\Lambda^\ell$ contains the fluxes of the advected scalar Λ.

2.5.2 Level Advance

The basic update on a single AMR level updates the solution on level ℓ from time t^ℓ to time $t^\ell + \Delta t^\ell$. Any coarser levels have already been updated from time $t^{\ell-1}$ to time $t^{\ell-1} + \Delta t^{\ell-1}$, so boundary conditions for the level ℓ advance may be taken from the coarser level, since $t^{\ell-1} \leq t^\ell < (t^\ell + \Delta t^\ell) \leq (t^{\ell-1} + \Delta t^{\ell-1})$. In cases where a time is specified for the coarse-level data, this data will be computed by linear interpolating in time using the coarse-level solutions at times $t^{\ell-1}$ and $t^{\ell-1} + \Delta t^{\ell-1}$.

1. **Compute Advection Velocities**
   First, a set of staggered-grid advection velocities $\vec{u}^{AD,\ell}$ is computed.

   (a) Compute upwinded face-centered velocities
   Before the tracing and upwinding steps are performed, we fill a ring of ghost cells around each grid which is wide enough to complete the tracing stencil for all interior cells with appropriate solution values for $\vec{u}(t^\ell)$. If a level ℓ ghost cell lies in the interior of another level ℓ grid, solution values are copied from the other grid. If the ghost cell lies over a coarser grid's valid region, the coarse-grid solution $\vec{u}^{\ell-1}$ is interpolated in time and space, using conservative linear interpolation. Once ghost cells have been filled, computation of the staggered-grid $\vec{u}^{half,\ell}$ can be carried out. For the viscous source term in the tracing step, however, we use a quadratic interpolation coarse-fine boundary condition: $\vec{u}^\ell(t^\ell) = I(\vec{u}^\ell(t^\ell), \vec{u}^{\ell-1}(t^\ell))$. At physical boundaries, inviscid (slipwall) boundary conditions are used for the tracing step, while viscous (no-slip-wall) boundary conditions are used when computing the viscous source.

We follow the approach detailed in the Chombo `AMRGodunov` documentation for a general hyperbolic transport equation with a source term S:

$$\frac{dW}{dt} + \sum_{d=0}^{\mathbf{D}-1} A_d(W)\frac{dW_d}{dx_d} = S$$

In the case of the Navier-Stokes momentum equation, W is the velocity, the advection velocity is is A_d, and the source term is $G\pi + \nu L\vec{u}$.

First, we compute approximate face-centered advection velocities $\vec{u}^{face}$ by averaging the cell-centered $\vec{u}^n$ to faces:

$$\vec{u}^{face} = Av^{C\to E}\vec{u}^n.$$

Next, we use Taylor expansions to extrapolate normal velocities to cell faces at time $t^n + \frac{\Delta t}{2}$, using (2.1) to replace the time derivative in the expansion. For the $d-$component of velocity on the $x-$direction faces $(\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^0)$, the Taylor expansion from the left state becomes:

$$\begin{aligned}\tilde{u}^{L,n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0,d} &= u^n_{\boldsymbol{i}} + min[\frac{1}{2}(1 - u^{norm}_{\boldsymbol{i},0}\frac{\Delta t}{h}), \frac{1}{2}](u_x)_{\boldsymbol{i},d} \\ &\quad -\frac{\Delta t}{2h}\sum_{s=0,s\neq 0}^{D-1} u^n_{\boldsymbol{i},s}(\bar{u}_s)_{\boldsymbol{i},d} + \frac{\nu\Delta t}{2}(L^\ell u + F)_{\boldsymbol{i},d}\end{aligned}$$

$$u^{norm}_{\boldsymbol{i},d} = \frac{1}{2}(u^{face}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0,d} + u^{face}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^0,d})$$

where $(u_x)_{\boldsymbol{i},d}$ is the limited undivided centered difference in the $x-$direction of the $d-$component of velocity, and $(\bar{u}_s)_{\boldsymbol{i},d}$ is the undivided (and unlimited) upwinded transverse difference in the s direction of the $d-$component:

$$(\bar{u}_s)_{\boldsymbol{i},d} = \begin{cases} u^n_{\boldsymbol{i},d} - u^n_{\boldsymbol{i}-\boldsymbol{e}^s,d} & if\ u^n_{\boldsymbol{i},s} > 0, \\ u^n_{\boldsymbol{i}+\boldsymbol{e}^s,d} - u^n_{\boldsymbol{i},d} & if\ u^n_{\boldsymbol{i},s} < 0. \end{cases}$$

The sum $\sum_{s=0,s\neq 0}^{D-1}$ is over all transverse directions. Computing the right state is similar:

$$\begin{aligned}\tilde{u}^{R,n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0,d} &= u^n_{\boldsymbol{i}+\boldsymbol{e}^0,d} + max[\frac{1}{2}(-1 - u^{norm}_{\boldsymbol{i}+\boldsymbol{e}^0,0}\frac{\Delta t}{h}), -\frac{1}{2}](u_x)_{\boldsymbol{i}+\boldsymbol{e}^0,d} \\ &\quad -\frac{\Delta t}{2h}\sum_{s=0,s\neq 0}^{D-1} u^n_{\boldsymbol{i}+\boldsymbol{e}^0,s}(\bar{u}_s)_{\boldsymbol{i}+\boldsymbol{e}^0,d} + \frac{\nu\Delta t}{2}(L^\ell u + F)_{\boldsymbol{i}+\boldsymbol{e}^0,d}.\end{aligned}$$

Then, we choose the upwind state; for the $x-$direction faces we choose like this:

$$u^{n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0,d} = \begin{cases} \tilde{u}^{L,n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0,d} & if u^{face}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0,0} > 0 \\ \tilde{u}^{R,n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0,d} & if u^{face}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0,0} < 0 \\ \frac{1}{2}(\tilde{u}^{L,n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0,d} + \tilde{u}^{R,n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0,d}) & if u^{face}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0,0} = 0 \end{cases}$$

The pressure term is not included in this extrapolation because these velocities are projected with a face-centered projection.

Extrapolation of normal y- and $z-$direction face velocities is similar.

(b) **Project Advection Velocities**

Then, $\vec{u}^{half,\ell}$ is projected using a staggered-grid (MAC) projection to ensure that the advection velocities are divergence-free:

$$L^\ell \phi^\ell = D^\ell \vec{u}^{half,\ell} \tag{2.12}$$

$$\phi^\ell = I(\phi^\ell, \frac{\Delta t^\ell}{2}\pi^{\ell-1}).$$

The coarse-fine boundary condition on ϕ is designed to match ϕ^ℓ with the coarse-level pressure field. Then, the velocity field is corrected:

$$\vec{u}^{half,\ell} = \vec{u}^{half,\ell} - G^\ell \phi^\ell \tag{2.13}$$

$$\phi^\ell = I(\phi^\ell, \frac{\Delta t^\ell}{2}\pi^{\ell-1}).$$

Finally, $\vec{u}_p$ from the most recent synchronization is added to correct for freestream preservation errors. For a more detailed explanation of the freestream preservation technique used here, see [MC00].

$$\vec{u}^{AD,\ell} = \vec{u}^{half,\ell} + \vec{u}_p.$$

2. **Scalar Advection**

Once advection velocities $\vec{u}^{AD,\ell}$ have been computed, the scalar Λ^ℓ can be updated. As in the previous step, a ring of ghost cells around each grid is filled by either copying values from other level ℓ grids or by performing a conservative linear interpolation in time and space of coarse-level data. Then, computation of the advective fluxes $\vec{F}^{\Lambda,\ell}$, as well as the updated scalar $\Lambda^\ell(t^\ell + \Delta t^\ell)$, can be computed on a grid-by-grid basis. The update equation used is

$$\Lambda^\ell(t^\ell + \Delta t^\ell) = \Lambda^\ell(t^\ell) - \Delta t^\ell D^\ell(\vec{u}^{AD}\Lambda^{half,\ell}).$$

First, we predict face-centered upwinded values for $\Lambda^{n+\frac{1}{2}}$ in the same way as for the velocity predictor (step 1a). As before, we compute values for $\tilde{\Lambda}^{L,n+\frac{1}{2}}$ and $\tilde{\Lambda}^{R,n+\frac{1}{2}}$, and then choose the upwind value based on the local sign of $\vec{u}^{half}$. In the $x-$direction:

$$\tilde{\Lambda}^{L,n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0} = \Lambda^n_{i,j} + min[\frac{1}{2}(1 - u^{AD}_{\boldsymbol{i},0}\frac{\Delta t}{h}), \frac{1}{2}](\Lambda_x)_{\boldsymbol{i}} - \frac{\Delta t}{2h}\sum_{s=0, s\neq 0}^{D-1} u^{tan}_{\boldsymbol{i},s}(\bar{\Lambda}_s)_{\boldsymbol{i}}.$$

As before,

$$u^{tan}_{\boldsymbol{i},s} = \frac{1}{2}(v^{half}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^s,s} + u^{half}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^s,s})$$

$$(\bar{\Lambda}_s)_{\boldsymbol{i}} = \begin{cases} \Lambda^n_{\boldsymbol{i}} - \Lambda^n_{\boldsymbol{i}-\boldsymbol{e}^s} & if \;\; u^{tan}_{\boldsymbol{i},s} > 0 \\ \Lambda^n_{\boldsymbol{i}+\boldsymbol{e}^s} - \Lambda^n_{\boldsymbol{i}} & if \;\; u^{tan}_{\boldsymbol{i},s} < 0. \end{cases}$$

For the right state:

$$\tilde{\Lambda}^{R,n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0} = \Lambda^n_{\boldsymbol{i}+ebold^0} + max[\frac{1}{2}(-1-u^{AD}_{\boldsymbol{i}}\frac{\Delta t}{h}), -\frac{1}{2}](\Lambda_x)_{\boldsymbol{i}+\boldsymbol{e}^0} - \frac{\Delta t}{2h}\sum_{s=0,s\neq 0}^{D-1} u^{tan}_{\boldsymbol{i}+\boldsymbol{e}^0,s}(\bar{\Lambda}_s)_{\boldsymbol{i}+\boldsymbol{e}^0}$$

Then, choose the upwind state:

$$\Lambda^{n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0} = \begin{cases} \tilde{\Lambda}^{L,n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0} & if u^{face}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0,0} > 0 \\ \tilde{\Lambda}^{R,n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0} & if u^{face}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0,0} < 0 \\ \frac{1}{2}(\tilde{\Lambda}^{L,n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0} + \tilde{\Lambda}^{R,n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0,0}) & if u^{face}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^0,0} = 0 \end{cases}$$

Computing $\Lambda^{n+\frac{1}{2}}$ on the $y-$ and $z-$faces is similar. Then, we compute the fluxes:

$$F^{\Lambda,d}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} = u^{AD}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d,d}\Lambda^{n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} \tag{2.14}$$

Finally, the updated state Λ^{n+1} can be computed using a conservative update equation:

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

3. **Predict $\vec{u}^{half}$**

   Using the advection velocities $\vec{u}^{AD,\ell}$, the transverse components of the staggered-grid velocity field $\vec{u}^{half,\ell}$ are computed, using the same coarse-fine boundary conditions with the level $\ell - 1$ solution as in the original tracing step.

   First, we re-predict face-centered velocities as in the advection-velocity computation, this time using $\vec{u}^{half}$ rather than $Av^{C\to E}(\vec{u}^n)$, which was used for the advection velocity computation. We re-use the already-computed normal velocities $\vec{u}^{half}$ as predicted velocities. So, we only compute the tangential face-centered predicted velocities. In this case, however, we include the pressure gradient – for the $d-$component of velocity on the $s-$direction face (recall that we only recompute tangential velocities, so $s \neq d$),

$$\begin{aligned} u^{half}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^s,d} &= u^{half}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^s,d} - (G\phi)_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^s,d}. \\ &= u^{half}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^s,d} - \frac{\phi_{\boldsymbol{i}+\boldsymbol{e}^d+\boldsymbol{e}^s} + \phi_{\boldsymbol{i}+\boldsymbol{e}^d-\boldsymbol{e}^s}\phi_{\boldsymbol{i}-\boldsymbol{e}^d+\boldsymbol{e}^s} - \phi_{\boldsymbol{i}-\boldsymbol{e}^d-\boldsymbol{e}^s}}{4h}. \end{aligned}$$

4. **Compute Advective Terms**
   Using $\vec{u}^{half}$, we compute an approximation of the advection term $[(\vec{u}\cdot\nabla)\vec{u}]^{n+\frac{1}{2}}$. Note that we use convective differencing for this step, since the advection velocities are not generally discretely divergence-free, due to the effects of the freestream preservation correction.

   To compute the advective terms, first compute a cell-centered advection velocity $\vec{u}^{AD-CC}$:
$$\vec{u}^{AD-CC} = Av^{E\rightarrow C}\vec{u}^{AD}.$$

   Then, for the $d-$direction momentum equation, the advective term is:
$$[(\vec{u}\cdot\nabla)u]^{n+\frac{1}{2}}_{\boldsymbol{i},d} \;=\; \sum_{s=0}^{\mathbf{D}-1} u^{AD-CC}_{\boldsymbol{i},s}\frac{(u^{half}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^s,d} - u^{half}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^s,d})}{h} \tag{2.15}$$

   Note the distinction between $\vec{u}^{AD}$, the *advecting* velocity, and $\vec{u}^{half}$, the *advected* velocity.

5. **Compute $\vec{u}^{*,\ell}$**
   The intermediate velocity field $\vec{u}^{*,\ell}$ is then computed using the second-order TGA scheme. The quantities a, r_1, and r_2 are the values suggested in [TGA96]:
$$\begin{aligned} a &= 2-\sqrt{2}-\epsilon, \\ discr &= \sqrt{a^2-4a+2}, \\ r_1 &= \frac{2a-1}{a+discr}, \\ r_2 &= \frac{2a-1}{a-discr}, \end{aligned}$$

   where ϵ is a small quantity (we use 10^{-8}).

   First compute the diffused source term $\vec{f}^{*}$. This differs from the algorithm presented in [TGA96] because our source term is centered at the half time $t^\ell + \frac{\Delta t^\ell}{2}$, while the source term in the original reference is centered at the time t^ℓ. When computing $\vec{f}^{*}$, we use a higher-order extrapolation coarse-fine boundary condition on $\vec{f}$. Physical boundary conditions are the viscous velocity boundary conditions.
$$\begin{aligned} \vec{f} &= [(\vec{u}\cdot\nabla)\vec{u}]^{n+\frac{1}{2}}_{\boldsymbol{i}} + \vec{F}^H - G^{CC,\ell}\pi^{n-\frac{1}{2}} \\ \vec{f}^{*} &= \Delta t(I + (\frac{1}{2}-a)\Delta t\nu L^\ell)\vec{f} \end{aligned} \tag{2.16}$$

   Then, we compute the intermediate quantity $\vec{u}^\ell_e$:
$$(I - r_2\Delta t\nu L^\ell)\vec{u}^\ell_e = \vec{u}^\ell(t^\ell) + (1-a)\Delta t\nu L^\ell\vec{u}^\ell(t^\ell) + \vec{f}^{*}$$

At coarse-fine interfaces, we use a quadratic interpolation boundary condition to compute $\vec{u}_e^\ell$: $\vec{u}_e^\ell = I(\vec{u}_e^\ell, \vec{u}^{\ell-1}(t^\ell + (1-r_1)\Delta t^\ell)$, along with viscous velocity boundary conditions at physical boundaries. For details of the quadratic coarse-fine interpolation used, see the Chombo documentation for the `QuadCFInterp` class.

Finally, we solve for the approximation to the $\vec{u}^\ell(t^\ell + \Delta t^\ell)$ velocity field $\vec{u}^*$:

$$(I - r_1 \Delta t \nu L^\ell)\vec{u}^* = \vec{u}_e$$

The coarse-fine boundary condition for this solve is quadratic interpolation: $\vec{u}^{*,\ell} = I\Big(\vec{u}^{*,\ell}, \vec{u}^{\ell-1}(t^\ell + \Delta t^\ell)\Big)$

6. **Initialize/Update Momentum and Advective Flux Registers**
   Once the updates have been completed, the appropriate flux registers are updated to contain the mismatches between coarse and fine advective and momentum fluxes. Note that the pressure gradient term is present in $\vec{f}$:

   - if $(\ell < \ell_{max})$

$$\begin{aligned} \delta\vec{\Lambda}^{\ell+1} &= -\vec{u}^{AD} \cdot \Lambda^{half} \\ \delta\vec{V}^{\ell+1} &= -\vec{u}^{AD} \cdot \vec{u}^{half} - (\frac{1}{2} - a)\nu \Delta t^\ell G^\ell(\vec{f}) \\ &\quad + \nu G^\ell(r_1 \vec{u}^* + r_2 \vec{u}_e + (1-a)\vec{u}(t^\ell)) \qquad \text{on } \partial\Omega^{\ell+1} \end{aligned}$$

   - if $(\ell > 0)$

$$\begin{aligned} \delta\vec{\Lambda}^{\ell} &= \delta\vec{\Lambda}^{\ell} + \frac{1}{n_{ref}} \langle \vec{u}^{AD} \cdot \Lambda^{half} \rangle \\ \delta\vec{V}^{\ell} &= \delta\vec{V}^{\ell} + \frac{1}{n_{ref}} \langle \vec{u}^{AD} \cdot \vec{u}^{half} + (\frac{1}{2} - a)\nu \Delta t^\ell G^\ell(\vec{f}) \\ &\quad - \nu G^\ell(r_1 \vec{u}^* + r_2 \vec{u}_e + (1-a)\vec{u}(t^\ell)) \rangle \qquad \text{on } \partial\Omega^{\ell} \end{aligned}$$

7. **Project** $\vec{u}^{*,\ell} \rightarrow \vec{u}^\ell(t^\ell + \Delta t^\ell)$
   To complete the single-level portion of the level update, the intermediate velocity field $\vec{u}^{*,\ell}$ is projected using a level projection. We solve for the pressure, instead of the update to the pressure, so we first remove the lagged pressure gradient used to compute $\vec{u}^*$:

$$\vec{u}^{*,\ell} = \vec{u}^{*,\ell} + \Delta t^\ell G^{CC,\ell} \pi^\ell$$

   Then, compute the RHS for the projection:

$$RHS = \frac{1}{\Delta t} D^{CC,\ell}(\vec{u}^{*,\ell})$$

   When computing the divergence of the velocity for the level projection, we use a quadratic coarse-fine boundary condition on the velocity field: $\vec{u}^* = I(\vec{u}^*, \vec{u}^{\ell-1}(t^\ell +$

$\Delta t^\ell) + \Delta t^\ell G^{CC,\ell-1}\pi^{\ell-1})$ Subtraction of $G\pi$ from the coarse level velocity accounts for the fact that u^* is actually $\vec{u} - \Delta t^\ell G^\pi$.

Then, solve for the level pressure π

$$\begin{aligned} L^\ell \pi^\ell &= RHS \\ \pi^\ell &= I(\pi^\ell, \pi^{\ell-1}), \end{aligned} \tag{2.17}$$

where $\pi^{\ell-1}$ in the coarse-fine boundary condition is the most recent $\pi^{\ell-1}$, which is treated as piecewise constant in time throughout the subcycled level ℓ timesteps. The correction is then applied to the velocity field:

$$\vec{u}^\ell(t^\ell + \Delta t^\ell) = \vec{u}^{*,\ell} - \Delta t G^\ell \pi^\ell$$

$$\pi^\ell = I(\pi^\ell, \pi^{\ell-1}).$$

2.5.3 Recursive update of finer levels

Once the single-level advance has been completed, if a finer level $\ell + 1$ exists, it is then updated n^ℓ_{ref} times with a timestep of $\Delta t^{\ell+1} = \frac{1}{n^\ell_{ref}}\Delta t^\ell$. This brings all levels finer than level ℓ to time $t^\ell + \Delta t^\ell$.

2.5.4 Synchronization

If a finer level $\ell + 1$ exists, we now synchronize level ℓ with all finer levels. We denote the time at which this synchronization takes place as $t^{sync} = t^\ell + \Delta t^\ell$. The coarsest level which has reached t^{sync} is denoted as ℓ_{base}; all levels finer than and including ℓ_{base} are synchronized at once. In practice, we check to see if the current level has reached the new time of the coarser level, $(t^{\ell-1} + \Delta t^{\ell-1})$. If so, we drop down to the coarser level. We also denote $\Delta t^{\ell_{base}}$ as Δt^{sync}, the time interval over which the synchronization is taking place.

1. **Refluxing and Averaging fine solution**
   First, the finer-level solutions are averaged down to underlying coarse grids and a refluxing operation is performed to correct coarse-level fluxes. For the passively-advected scalar Λ, this is simply the refluxing correction for conservation:

   $$[\Lambda]^\ell := [\Lambda]^\ell - \Delta t^\ell D_R(\delta[\Lambda]^{\ell+1})$$

   For the case of small viscosity, the velocity may be corrected explicitly in the same manner:

   $$\vec{u}(t^{sync}) := \vec{u}(t^{sync}) - \Delta t^\ell D_R(\delta\vec{V}^{\ell+1})$$

However, in the more general case, the refluxing must be implicit for stability. In this case, we compute a correction with a multilevel elliptic solve for all levels $\ell \geq \ell_{base}$:

$$(I - \nu \Delta t^{\ell_{base}} L^{comp}) \delta \vec{u} = \Delta t^{\ell} D_R(\delta \vec{V}^{\ell+1})$$

$$\delta \vec{u}^{\ell_{base}} = I(\delta \vec{u}^{\ell_{base}}, 0).$$

Note that the scaling factor for the right-hand-side varies by level, while Δt^{sync} is used as the coefficient for all levels for the elliptic solve. The correction is added to the velocity field for all levels $\ell \geq \ell_{base}$:

$$\vec{u} = \vec{u} + \delta \vec{u}$$

2. **Composite Projection**
   Then, a multilevel sychronization projection is applied in the same way as in [MC00], solving for the synchronization correction e_{sync}. The appropriate physical boundary conditions for e_{sync} are the homogeneous form of the boundary conditions applied to the level pressure π in the level projection. For solid walls, this is a homogeneous Neumann boundary condition. First, solve for e_{sync}

   $$L^{Comp} e_{sync} = D^{CC,Comp} \vec{u}(t^{sync}) \quad \text{for} \ell > \ell_{\text{base}}$$

   Coarse-fine boundary conditions for e_{sync} are quadratic interpolation: $e_s^{\ell_{base}} = I(e_s^{\ell_{base}}, \Delta t^{\ell_{base}} e_s^{\ell_{base}-1})$
   Then, correct the velocity field:

   $$\vec{u} = \vec{u} - G^{CC,comp} e_s$$

   The correction is then stored for future use by rescaling it to be a pressure:

   $$e_{sync} = \frac{1}{\Delta t^{\ell_{base}}} e_{sync}$$

3. **Freestream Preservation Correction**
   Finally, the freestream preservation correct ion $\vec{u}_p$ is computed, using the same approach as in [MC00]. First, solve

   $$L^{comp} e_{\Lambda}^{\ell} = \frac{(\Lambda - 1)}{\Delta t^{\ell_{base}}} \eta \quad \text{for} \quad \ell \geq \ell_{\text{base}}$$

   The coarse-fine boundary condition for e_Λ is quadratic interpolation: $e^{\ell} = I(e_{\Lambda}^{\ell}, 0)$
   The correction is then computed and stored for future use:

   $$u_p = G^{comp} e_{\Lambda}$$

2.6 Regridding and Pressure Initialization

Before the initial time step for a level ℓ, we will need to come up with initial values for π. This will also be necessary after regridding. After regridding, we will also need to recompute $\vec{u}_p$, the volume discrepancy correction, for the new grid configuration. We will define ℓ_{base} as the finest *unchanged* level. Note that for the initial grid set-up and initialization, ℓ_{base} would be -1. n_{passes} is the number of passes we make in order to initialize things. We have used $n_{passes} = 1$. The pressure re-initialization step is similar to a non-subcycled timestep.

2.6.1 Project new velocity field

First, the new velocity field is projected using the multilevel composite projection to ensure that the new velocity field is divergence-free.

2.6.2 Compute level pressures π

Use a series of non-subcycled level advances with $\widetilde{\Delta t} = \frac{\Delta t^{\ell_{max}}}{2}$ to compute the level pressures π. In principle, since the level pressure is needed for an accurate computation of $\vec{u}^*$, this computation is performed iteratively, with n_{passes} for the initialization in order to compute π accurately. In practice, we have generally set n_{passes} to be one, since the initialization computation is expensive.

First, we compute $\widetilde{\vec{u}}^*$ as in a normal timestep, using the pressure gradient term if it is available. For the face-centered projection of the advection velocities, if a coarse-level π is not available, then we use a coarse-fine boundary condition of $\phi^\ell = I(\phi^\ell, \phi^{\ell-1})$. Otherwise, all of the other boundary conditions for the initialization timesteps are the same as are used in a regular advance.

Then, we project $\widetilde{\vec{u}}^*$ to compute π, first solving

$$L^\ell \pi^\ell = \frac{1}{\widetilde{\Delta t}} D^{CC,\ell}(\widetilde{u^*})$$

$$\pi^\ell = I(\pi^\ell, \pi^{\ell-1}(\tilde{t})$$

Then, $\widetilde{\vec{u}}^*$ is corrected for use as a boundary condition for any finer-level initialization:

$$\widetilde{\vec{u}^{new}} := \widetilde{\vec{u}}^* - \widetilde{\Delta t} G^{CC,\ell} \pi^\ell$$

2.6.3 Initialize $\vec{u}_p$

To initialize the freestream-preservation correction, we first initalize Λ in any newly-refined regions using conservative linear interpolation. Then, we solve for e_Λ:

$$L^{comp} e^\ell_\Lambda = \frac{\eta}{\Delta t^{\ell_{base}}}(\Lambda - 1) \;\; \text{for} \;\; \ell \geq \ell_{base},$$

$$e_\Lambda^\ell = I(e_\Lambda^\ell, 0),$$

and compute and store the correction field:

$$\vec{u}_p = G^{comp} e_\Lambda$$

Note that this need not be done at the initial step, since it is assumed that no freestream preservation errors have been introduced yet, and $\vec{u}_p$ may be set to 0.

2.7 Pseudocode Description of Algorithm

2.7.1 Notation

$\vec{u}_{\boldsymbol{i}}^n$	$(u_{\boldsymbol{i}}^n, v_{\boldsymbol{i}}^n)$ – cell centered velocities at time n
$\vec{u}_G^{half}$	$(u_{i+\frac{1}{2},j}^{n+\frac{1}{2}}, v_{i,j+\frac{1}{2}}^{n+\frac{1}{2}})$ – Godunov predicted velocites (unprojected)
$\vec{u}^{half}$	MAC projected $\vec{u}_G^{half} = \vec{u}_G^{half} - \nabla\phi$
$\vec{u}_{ad}^{half}$	advection velocity $= \vec{u}^{half} + \vec{u}_p$
$\vec{u}_p$	∇e_Λ – velocity correction due to volume discrepancy stuff
$\vec{F}$	body force on fluid ($\vec{F}^H$ is centered at the half-time)
ϕ	MAC correction
$\pi_{\boldsymbol{i}}^{n-\frac{1}{2}}$	pressure from level projection
e_s	pressure correction from sync projection
e_Λ	correction due to volume discrepancy stuff
$[s]_{\boldsymbol{i}}^n$	Advected scalars: ρ, Λ, s, etc
Λ	advected scalar for freestream preservation – initially $= 1$
$\delta[s]^\ell, \delta\vec{V}^\ell$	Hyperbolic flux registers for level ℓ (defined on fine level) for scalars and velocities
G^ℓ	face-centered (MAC) gradient on level ℓ.
D^ℓ	face-centered (MAC) divergence on level ℓ
L^ℓ	Laplacian on level ℓ (same as L^{nf} in previous implementations)
G^{comp}	composite face-centered (MAC) gradient
D^{comp}	composite face-centered (MAC) divergence
L^{comp}	composite Laplacian
$G^{CC,\ell}$	cell-centered gradient on level ℓ
$D^{CC,\ell}$	cell-centered divergence on level ℓ
$G^{CC,comp}$	composite cell-centered gradient
$D^{CC,comp}$	composite cell-centered divergence
ν	coefficient of viscosity
a	coefficients for viscous solve: $a = 2 - \sqrt{2} - \epsilon$
r_1	coefficient for viscous solve: $r_1 = \frac{2a-1}{a+(a^2-4a+2)^{\frac{1}{2}}}$
r_2	coefficient for viscous solve: $r_2 = \frac{2a-1}{a-(a^2-4a+2)^{\frac{1}{2}}}$

2.7.2 Timestep for level ℓ

At time $t = t^n$, we have $\vec{u}^n_{\boldsymbol{i}}, \Lambda^n_{\boldsymbol{i}}, \vec{u}_p$. For a level ℓ, the timestep looks like:

1. Compute advection velocities:

   (a) Predict $\vec{u}^{half}_G = u^{n+\frac{1}{2}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d,d}$ (viscous Godunov Box)
   C/F BC: Conservative interpolation from level $\ell - 1$ (FilPatch) for tracing
   C/F BC: $\vec{u}^\ell(t^\ell) = I(\vec{u}^\ell(t^\ell), \vec{u}^{\ell-1}(t^\ell))$ for viscous source term $L^\ell\vec{u}^\ell(t^\ell)$.
   Physical BC: inviscid slipwall for tracing
   Physical BC: viscous BC for viscous source term

   (b) MAC projection: $L\phi = D^\ell\vec{u}^{half}_G$,
   C/F BC: $\phi^\ell = I(\phi^\ell, \frac{\Delta t}{2}\pi^{n-\frac{1}{2},\ell-1}) \quad \text{on} \quad \partial\Omega^\ell - \partial\Omega$
   Physical BC: Homogeneous Neumann for ϕ.

   (c) Correct predicted velocities: $\vec{u}^{half} = \vec{u}^{half}_G - G^\ell\phi$
   C/F BC: $\phi^\ell = I(\phi^\ell, \frac{\Delta t}{2}\pi^{n-\frac{1}{2},\ell-1}) \quad \text{on} \quad \partial\Omega^\ell - \partial\Omega$
   Physical BC: Quadratic extrapolation of ϕ

   (d) $\vec{u}^{half}_{AD} = \vec{u}^{half} + \vec{u}_p$
   $(\vec{u}_p = G^{comp}e_\Lambda)$

2. Advance advected quantities:

   (a) Trace states using $\vec{u}^{half}_{AD}$: $\Lambda^{half} \approx \Lambda^{n+\frac{1}{2}}_{faces}$
   C/F BC: Conservative interpolation (FilPatch)

   (b) Advance: $\Lambda^{n+1}_{\boldsymbol{i}} = \Lambda^n_{ij} - \Delta t D^\ell \cdot (\vec{u}_{AD} \cdot \Lambda^{half})$

   (c) Update flux registers:
   - if $(\ell < \ell_{max})$ $\quad \delta\Lambda^{\ell+1} = -\vec{u}^{half}_{AD} \cdot \Lambda^{half} \quad \text{on } \partial\Omega^{\ell+1}$
   - if $(\ell > 0)$ $\quad \delta\Lambda^\ell = \delta\Lambda^\ell + \frac{1}{n_{ref}}\langle\vec{u}^{half}_{AD} \cdot \Lambda^{half}\rangle \quad \text{on } \partial\Omega^\ell$

3. Compute $\vec{u}^*$:

   (a) As in (2a), trace $\vec{u}^{half}$

   (b) Compute diffused source term:
   $\vec{f}^* = [-Av^{E\rightarrow C}(\vec{u}^{half}_{AD}) \cdot (D^\ell\vec{u}^{half}) + \vec{F}^H - G^{CC,\ell}\pi^{n-\frac{1}{2}}]$
   $\vec{f} = \Delta t(I + (\frac{1}{2} - a)\Delta t\nu L^\ell)\vec{f}^*$
   C/F BC: HO extrap for $\vec{f}^*$
   PhysBC: viscous velocity BC's?

(c) intermediate solve:
$(I - r_2 \Delta t \nu L^\ell)\vec{u}_e^\ell = \vec{u}^\ell(t^\ell) + (1-a)\Delta t \nu L^\ell \vec{u}^\ell(t^\ell) + \vec{f}$
C/F BC: $\vec{u}_e^\ell = I(\vec{u}_e^\ell, \vec{u}^{\ell-1}(t^\ell + (1-r_1)\Delta t^\ell)$
PhysBC: viscous Velocity BC's

(d) solve for u^*:
$(I - r_1 \Delta t \nu L^\ell)\vec{u}^* = \vec{u}_e^\ell$
C/F BC: $\vec{u}^{*,\ell} = I\Big(\vec{u}^{*,\ell}, \vec{u}^{\ell-1}(t^\ell + \Delta t^\ell)\Big)$

(e) Update velocity flux registers

- if $(\ell < \ell_{max})$ $\quad \delta\vec{V}^{\ell+1} = -\vec{u}_{AD}^{half} \cdot \vec{u}^{half}$
$- (\frac{1}{2} - a)\nu \Delta t^\ell G^\ell(f^*)$
$+ \nu G^\ell(r_1 \vec{u}^* + r_2 \vec{u}_e + (1-a)\vec{u}(t^\ell)) \quad \text{on } \partial\Omega^{\ell+1}$
- if $(\ell > 0)$ $\quad \delta\vec{V}^\ell = \delta\vec{V}^\ell + \frac{1}{n_{ref}}\langle \vec{u}_{AD}^{half} \cdot \vec{u}^{half}$
$+ (\frac{1}{2} - a)\nu \Delta t^\ell G^\ell(f^*)$
$- \nu G^\ell(r_1 \vec{u}^* + r_2 \vec{u}_e + (1-a)\vec{u}(t^\ell))\ \rangle \quad \text{on } \partial\Omega^\ell$

4. Project $\vec{u}^* \Rightarrow \vec{u}^\ell(t^\ell + \Delta t^\ell)$

(a) Solve $L^\ell(\pi^\ell) = \frac{1}{\Delta t} D^{CC,\ell}\Big(\vec{u}^* + \Delta t G^{CC,\ell}\pi^\ell\Big)$

C/F BC: $\pi^\ell = \mathrm{I}(\pi^\ell, \pi^{\ell-1}) \quad \text{on} \quad \partial\Omega^\ell - \partial\Omega$
C/F BC: $\vec{u}^* = I(\vec{u}^*, \vec{u}^{\ell-1}(t^\ell + \Delta t^\ell) + \Delta t^\ell G^{CC,\ell-1}\pi^{\ell-1})$
Physical BC: Homogeneous Neumann on π^ℓ

(b) Update level velocity: $\vec{u}^{new,\ell} = \vec{u}^{*,\ell} - \Delta t G^{CC,\ell}\pi^\ell$
C/F BC: $\pi^\ell = I(\pi^\ell, \pi^{\ell-1}) \quad \text{on} \quad \partial\Omega^\ell - \partial\Omega$

5. Advance level $\ell + 1$ n_{ref} times with $\Delta t^{\ell+1} = \frac{1}{n_{ref}}\Delta t^\ell$.

6. Synchronization: if $(\ell < \ell_{max})$

(a) Reflux for conservation:

- $\Lambda^\ell = \Lambda^\ell - \Delta t^\ell D_R(\delta\Lambda^{\ell+1})$
- Solve: $(I - \nu \Delta t^{\ell_{base}} L^{comp})\delta\vec{u} = \Delta t^\ell D_R(\delta\vec{V}^{\ell+1})$

(b) Sync projection to satisfy elliptic matching conditions:
CC sync solve: ℓ_{base} = coarsest level at time $t = t^{new}$. This now looks a lot like the initial velocity projection

i. $RHS = D^{CC,comp}\vec{u}^{new}$:
For $\ell = \ell_{max}, \ell_{base}, -1$

A. $\vec{u}^{\ell}_{face} = Av^{C\to E}(\vec{u}^{\ell})$
C/F BC: $\vec{u}^{\ell}$ = Extrap($\vec{u}^{\ell}$) $\quad \text{on} \quad \partial\Omega^{\ell}$

B. $RHS^{\ell} = D^{\ell}\vec{u}^{\ell}_{face}$

C. Update flux registers:

- if $(\ell < \ell_{max}) \qquad \delta\vec{u}^{\ell+1} = \delta\vec{u}^{\ell+1} - (\Delta x^{\ell})\vec{u}^{\ell}_{Face} \qquad \text{on } \partial\Omega^{\ell+1}$
- if $(\ell > 0) \qquad \delta\vec{u}^{\ell} = n_{ref}\Delta x^{\ell}\langle\vec{u}^{\ell}_{Face}\rangle \qquad \text{on } \partial\Omega^{\ell}$

D. if $(\ell < \ell_{max}) \qquad RHS^{\ell} = RHS^{\ell} + D_R(\delta\vec{u}^{\ell+1})$

ii. Composite Solve: $L^{Comp}e_s = RHS$
Physical BC: Homogeneous Neumann
C/F BC: $e_s^{\ell_{base}} = I(e_s^{\ell_{base}}, \Delta t^{\ell_{base}} e_s^{\ell_{base}-1}) \quad \text{on } \partial\Omega^{\ell_{\text{base}}}$

iii. Correct Velocities: $\vec{u} = \vec{u} - G^{CC,comp}e_s$
C/F BC: $e_s^{\ell_{base}} = I(e_s^{\ell_{base}}, \Delta t^{\ell_{base}} e_s^{\ell_{base}-1}) \quad \text{on } \partial\Omega^{\ell} - \partial\Omega$
Physical BC: Quadratic Extrapolation of e_s

- for $\ell = \ell_{max}, 0, -1$

  A. $\vec{u}^{\ell} = \vec{u}^{\ell} - G^{CC,comp}e_s$

  B. Store correction for future use: $Ge_s = \frac{1}{\Delta t^{sync}}G^{CC,comp}e_s$

  C. $e_s = \frac{1}{\Delta t^{\ell_{base}}}e_s$

(c) Mac Sync:

i. Solve $L^{comp}e^{\ell}_{\Lambda} = \frac{(\Lambda-1)}{\Delta t^{\ell_{base}}}\eta \quad$ for $\ell \geq \ell_{base}$
C/F BC: $e^{\ell} = I(e^{\ell}_{\Lambda}, 0) \;\; on \;\; \partial\Omega^{\ell}$

ii. Compute and store correction for future use:
$u_p = G^{comp}e_{\Lambda}$

2.7.3 Regridding and Pressure Initialization

Before the initial time step for a level ℓ, we will need will need to project the velocity field to ensure that it is divergence-free, and to compute initial values for π. This will also be necessary after regridding. After regridding, we will also need to recompute $\vec{u}_p$, the volume discrepancy correction, for the new grid configuration. We will define ℓ_{base} as the finest *unchanged* level. Note that for the initial grid set-up and initialization, ℓ_{base} would be (-1). n_{passes} is the number of passes we make in order to initialize things. We have used $n_{passes} = 1$.

1. Initial Velocity Projection:

(a) For $\ell = \ell_{max}, 0, -1$

i. $\vec{u}^{\ell}_{Face} = Av^{C\to E}(\vec{u}^{\ell})$
C/F BC: $\vec{u}^{\ell} = I(\vec{u}^{\ell}, \vec{u}^{\ell-1}) \;\; \text{on } \partial\Omega^{\ell}$

ii. $RHS^{\ell} = D^{CC,\ell}\vec{u}^{\ell}_{Face}$

iii. Update flux Registers:
   - if $(\ell < \ell_{max})$ $\quad \delta\vec{u}^{\ell+1} = \delta\vec{u}^{\ell+1} - (\Delta x^\ell)\vec{u}^\ell_{Face} \quad \text{on } \partial\Omega^{\ell+1}$
   - if $(\ell > 0)$ $\quad \delta\vec{u}^\ell = n_{ref}\Delta x^\ell \langle \vec{u}^\ell_{Face}\rangle \quad \text{on } \partial\Omega^\ell$

iv. if $(\ell < \ell_{max})$ $\quad RHS^\ell = RHS^\ell + D_R(\delta\vec{u}^{\ell+1})$

(b) Composite Solve: $L^{Comp}\phi = RHS$
Physical BC: Homogeneous Neumann

(c) Correct Velocities: $\vec{u} = \vec{u} - G^{CC,Comp}\phi$
C/F BC: $\phi^\ell = I(\phi^\ell, \phi^{\ell-1}) \quad \text{on } \partial\Omega^\ell - \partial\Omega$
Physical BC: Quadratic Extrapolation of ϕ

- for $\ell = \ell_{max}, 0, -1$
  i. $\vec{u}^\ell = \vec{u}^\ell - G^{CC,comp}\phi$

(d) Average Down for consistency: $\vec{u}^\ell = Avg(\vec{u}^{\ell+1}) \qquad \text{on } \mathcal{C}_{n^\ell_{ref}}(\Omega^{\ell+1})$

2. Initialize pressure fields:
for $n = 1$, n_{passes}:

   (a) $\widetilde{\Delta t} = \frac{\Delta t^{\ell max}}{2}$

   (b) For $\ell = \ell_{base} + 1, \ell_{max}$

      i. Compute $\widetilde{u^*}$ as in normal timestep (only without pressure corrections.)
      C/F BC (for MAC): $\phi^\ell = I(\phi^\ell, \phi^{\ell-1})$ if $n = 1$
      C/F BC (for MAC): $\phi^\ell = I(\phi^\ell, \frac{\widetilde{\Delta t}}{2}(\pi^{\ell-1}))$ otherwise

      ii. Solve $L^\ell\pi^\ell = \frac{1}{\widetilde{\Delta t}}D^{CC,\ell}(\widetilde{u^*})$
      C/F BC: $\pi^\ell = I(\pi^\ell, \pi^{\ell-1}(\tilde{t} + e_s^{\ell-1})$
      Physical BC: Homogeneous Neumann

      iii. Correct $\widetilde{\vec{u}^*}$: $\widetilde{\vec{u}^{new}} = \widetilde{\vec{u}^*} - \widetilde{\Delta t}G^{CC,\ell}\pi^\ell$
      C/F BC: $\pi^\ell = I(\pi^\ell, \pi^{\ell-1} + e_s^{\ell-1}) \quad \text{on } \ \partial\Omega^\ell - \partial\Omega$

   (c) If $(\ell_{base} > -1)$

      i. Compute $\widetilde{u^*}^{\ell_{base}}$ as in normal timestep (this time <u>using</u> pressure corrections)

      ii. Solve $L^\ell\widetilde{\pi}^{\ell_{base}} = \frac{1}{\widetilde{\Delta t}}D^{CC,\ell}(\widetilde{u^*}^{\ell_{base}})$
      C/F BC: $\widetilde{\vec{u}^*} = I(\widetilde{\vec{u}^*}, \widetilde{\vec{u}}^{\ell-1}(t^\ell + \Delta t^\ell) + \Delta t^\ell G^{CC,\ell-1}\pi^{\ell-1})$
      C/F BC: $\widetilde{\pi}^{\ell_{base}} = I(\widetilde{\pi}^{\ell_{base}}, \pi^{\ell_{base}-1}(\tilde{t}))$ on $\partial\Omega^{\ell_{base}} - \partial\Omega$
      Physical BC: Homogeneous Neumann

      iii. Correct $\widetilde{\vec{u}^*}$: $\widetilde{\vec{u}^{new}} = \widetilde{\vec{u}^*} - \widetilde{\Delta t}G^{CC,\ell}\pi^\ell$
      C/F BC: $\pi^\ell = I(\pi^\ell, \pi^{\ell-1}) \quad \text{on } \ \partial\Omega^\ell - \partial\Omega$

3. Volume Discrepancy initialization (if not initial step)

(a) Solve $L^{comp} e^{\ell}_{\Lambda} = \frac{(\Lambda-1)}{\Delta t^{\ell_{base}}} \eta \;\; \text{for} \;\; \ell \geq \ell_{base}$
C/F BC: $e^{\ell}_{\Lambda} = I(e^{\ell}_{\Lambda}, 0) \text{ on } \partial\Omega^{\ell}$

(b) Store correction for future use: $Ge^{\ell}_{\Lambda} = G^{comp} e_{\Lambda}$

Chapter 3

System Architecture and Component-level Design

3.1 Architecture Diagram

The incompressible Navier-Stokes code makes extensive use of the AMR time-dependent infrastructure contained in the Chombo libararies. A basic schematic of the class relationships between `Chombo` and `AMRINS` classes is depicted in Figure 3.1. For a more detailed description of the inter-relationships between AMRINS and Chombo classes, see the AMRINS-Chombo reference manual.

3.2 Data Design

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

3.2.1 Global Data Structures

The important variables in the AMRINS code are the velocities, and possibly the pressure field. These variables are contained in container classes provided by `Chombo`.

Chombo Container Classes

A logically rectangular region in space is defined by a `Box`. Cell-centered data on an individual `Box` is generally contained in an `FArrayBox`; face-centered data on a `Box` is generally contained by a `FluxBox`, which contains one face-centered `FArrayBox` for each space dimension.

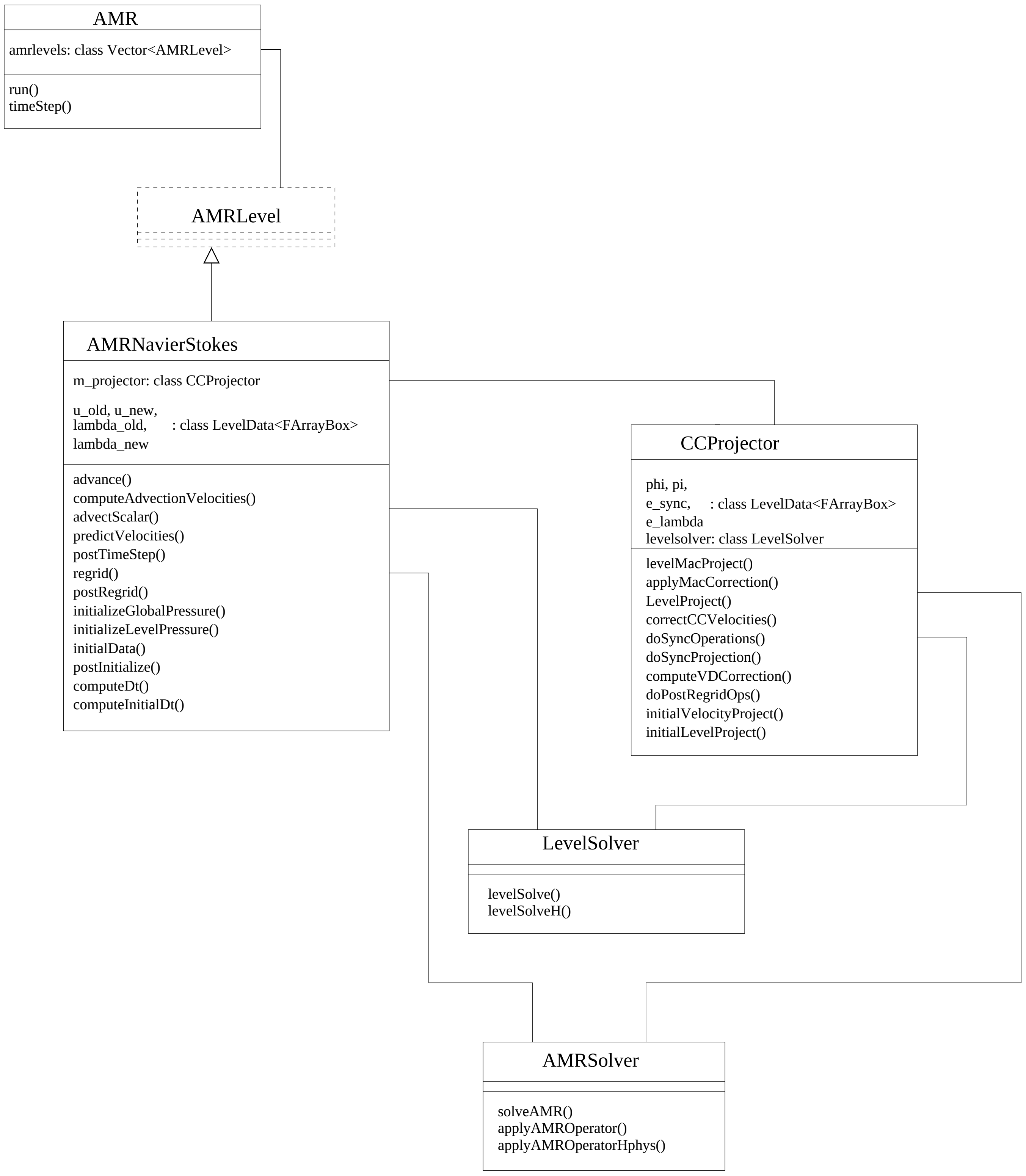

Figure 3.1: Software configuration diagram for the AMRINS code showing basic relationships between AMRINS classes and Chombo classes.

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

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

Time-dependent AMR

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

The AmrLevel class manages data and functionality for a single AMR level, including the single-level advance. The AMRNavierStokes class is derived from the AmrLevel class and contains the functionality specific to the Navier-Stokes solution algorithm.

3.2.2 Internal Software Data Structures

The AMRNavierStokes class contains the primary data fields necessary to update the solution on one AMR level, in particular the old- and new-time velocity fields ($\vec{u}(t^\ell)$ and $\vec{u}(t^\ell + \Delta t^\ell)$), and the old- and new-time auxiliary scalar fields ($\Lambda(t^\ell)$ and $\Lambda(t^\ell + \Delta t^\ell)$. Each AMRNavierStokes also contains a CCProjector class as a member object. This CCProjector member contains all the data and functionality necessary for enforcing the divergence constraint on the given AMR level, including the freestream preservation correction. The CCprojector class contains the ϕ and π pressure fields, along with the synchronization correction e_{sync} and the freestream preservation correction quantities e_Λ and $\vec{u}_p$, as well as the methods used to compute and access these quantities.

3.3 Basic Design

In the Chombo time-dependent infrastructure, the AMR class manages the data on the multilevel hiearchy and the subcycled timestepping, basic grid generation and regridding operations, along with data input and output. The AMRLevel class is an abstract base class which contains the data for a single level, along with the functionality for updating the solution on a single level. For a detailed description of the AMR and AMRLevel classes, see the Chombo documentation.

The AMRNavierStokes class is derived from AMRLevel and contains the data and functionality for implementing the incompressible algorithm described in Chapter 2. This includes managing the single-level update in Section 2.5.2, the specific synchronization algorithm for the Navier-Stokes equations detailed in Section 2.5.4, and the initialization steps in Section 2.6.

The AMRNavierStokes class uses the CCProjector class, which manages all of the algorithm pieces which enforce the incompressibility constraint, including the face-centered projection used in the advection velocity computation, the single-level and composite cell-centered projections, and the freestream preservation correction. The `CCProjector` class also contains all of the data used for these purposes (ϕ, π, e_Λ, and $\vec{u}_p$). In the current design, each `AMRNavierStokes` object contains a `CCProjector` object as a protected member. Data flow between the `AMRNavierStokes` and `CCProjector` classes is fairly straightforward. Velocity fields which need to be projected are passed from the `AMRNavierStokes` class to the associated `CCProjector` class, where the projection is performed and the velocity field is corrected in place. In the case of the freestream preservation correction, the composite Λ field is passed to the `CCProjector::computeVDCorrection` function, where the freestream preservation correction is computed. Where the algorithm requires a pressure field, gradients of the pressure field, or the freestream preservation correction, these are supplied by public access functions of the `CCProjector` class.

3.4 The class `AMRNavierStokes`

- `Real advance()`
  This function manages the entire single-level advance described in Section 2.5.2, advancing the solution on the current AMR level ℓ (`AMRNavierStokes::m_level`) from time t^ℓ to time $t^\ell + \Delta t^\ell$. It does not advance finer levels or perform synchronization. Returns the maximum safe timestep for the next level advance.

- `void computeAdvectionVelocities(LevelData<FluxBox>& a_adv_vel)` This function, called by `advance`, manages the computation of the face-centered advection velocities (step #1 in Section 2.5.2). This includes the tracing step and the calling the face-centered projection function (`CCProjector::levelMacProject`). The freestream preservation correction is then added to the resulting face-centered velocity field, and the advection velocity field $\vec{u}^{AD,\ell}$ is returned.
  Outputs:

  - `a_adv_vel` – face-centered divergence-free advection velocities.

-
  ```
  void advectScalar(LevelData<FArrayBox>& a_new_scal,
                    LevelData<FArrayBox>& a_old_scal,
                    LevelData<FluxBox>& a_adv_vel,
                    LevelFluxRegister* a_crseFluxRegPtr,
                    LevelFluxRegister& a_fineFluxReg,
                    Real a_dt)
  ```
  This function, called by `advance`, manages the scalar advection step (step #2 in Section 2.5.2. The scalar at the new time, $\Lambda^\ell(t^\ell + \Delta t^\ell)$ is returned. Also, the flux registers for Λ are updated.
  Inputs:

  - `a_old_scal` – scalar at old time

  - `a_old_scal` – scalar at new time item `a_adv_vel` – face-centered advection velocities
  - `a_dt` – time step

  Outputs:

  - `a_new_scal` – scalar at new time (old time + Δt).
  - `a_crseFluxRegPtr` – flux register to store fluxes along the interface between this level and the next coarser level
  - `a_fineFluxReg` – flux register to store fluxes along the interface between this level and the next finer level.

-
  ```
  void predictVelocities(LevelData<FArrayBox>& a_uDelU,
                         LevelData<FluxBox>& a_advVel)
  ```
  This function, called by `advance`, manages steps # 3 and 4 in Section 2.5.2; it handles the tranverse-velocity tracing step, and returns $[(\vec{u}\cdot\nabla)\vec{u}]^{n+\frac{1}{2}}$. Also, flux registers are incremented with the advective part of the fluxes.
  Inputs:

  - `a_advVel` – face-centered advection velocities

  Outputs

  - `a_uDelU` – avective term (approximation to $[(\vec{u}\cdot\nabla)\vec{u}]^{n+\frac{1}{2}}$).

- `void computeUStar(LevelData<FArrayBox>& a_uStar)`
  This function, called by `advance`, manages step #5 in Section 2.5.2, setting up and computing the two elliptic solves per velocity component necessary to do the viscous update. The momentum flux registers are also updated with the viscous fluxes computed during this process. Because the algorithms used for the viscous update are generally applicable in a variety of contexts, the viscous updates computed in this function will be factored out into a set of viscous solver classes in the next code design cycle.
  Inputs:

  - `a_uStar` – $\vec{u}^{old} + \Delta t[(\vec{u}\cdot\nabla)\vec{u}]^{n+\frac{1}{2}}$ (advective update to velocity field)

  Outputs:

  - `a_uStar` – approximation to new-time velocity (including the viscous terms)

- `void postTimeStep()`
  The `AMRNavierStokes::postTimeStep` function, called by `AMR::timeStep` function, manages the synchronization steps described in Section 2.5.4. This includes the (implicit or explicit) refluxing steps, calling the `CCProjector::doSyncOperations` function, and averaging fine-level data onto covered regions of coarser grids.

- `void regrid(const Vector<Box>& a_new_grids)`
  This function, called by the `AMR::regrid` function, sets up the `AMRNavierStokes` class after a regridding operation has modified the grids. Data storage is redefined to fit the new grids, and data is filled in from existing grids or interpolated from coarser grids to fill newly-refined regions.
  Inputs:
  - `a_new_grids)` – new grids for this level
- `void postRegrid(int a_lBase)`
  This function, called at the end of the `AMR::regrid` function, finishes the initialization process after `AMRNavierStokes::regrid` has been called on all levels. The initialization processm, described in Section 2.6, includes a composite projection of the newly initialized velocity to ensure that it is divergence-free (`CCProjector::initialVelocityProject`), calling `CCProjector::doPostRegridOps`, and calling `AMRNavierStokes::initializeGlobalPressure`.
  Inputs:
  - `a_lBase` – coarsest unchanged level in the regridding operation.
- `void initializeGlobalPressure()`
  The `AMRNavierStokes::initializeGlobalPressure` function, called by the `AMRNavierStokes::postRegrid` and `AMRNavierStokes::postInitialize` functions, manages the pressure initialization steps described in Section 2.6.2. This function is called from the base level for the regridding operation, and in turn calls `AMRNavierStokes::initializeLevelPressure` for each newly initialized level.
- `void initializeLevelPressure(Real a_currentTime, Real a_dtInit)`
  The `AMRNavierStokes::initializeLevelPressure` function manages the pressure initialization described in Section 2.6.2 for a single AMR level. It essentially performs a single-level timestep (similar to `AMRNavierStokes::advance`, computing $\vec{u}^*$ and then projecting it to get an approximation to the pressure π.
  Inputs:
  - `a_currentTime` – current time of solution.
  - `a_dtInit` – timestep to use for initialization timestep.
- `void initialData()`
  This function, called by `AMR::initialGrid`, initializes the data on a single AMR level. In particular, the freestream preservation marker quantity Λ is initialized to 1, and the initial velocity field is defined.
- `void postInitialize()`
  This function, also called by `AMR::initialGrid`, is similar to the `postRegrid`

function, implementing the initialization process described in Section 2.6. It manages the initial velocity projection (`CCProjector::initialVelocityProject`), and calls the `initializeGlobalPressure` function. Since this function is called at the initial time in the problem, it is assumed that there is no need to compute a freestream preservation correction, since no freestream preservation errors have been introduced yet.

- `Real computeDt()`
  This function, called from `AMR::timeStep` and from `AMRNavierStokes::computeInitialDt`, computes the timestep at which a given level may be advanced. It does this by computing the maximum stable timestep based on an advective CFL condition, then multiplies that timestep by the CFL number σ.

- `Real computeInitialDt()`
  This function, called from `AMR::setupForNewAMRRun`, computes the timestep which may be used for the initial timestep. In the current implementation, this function calls `AMRNavierStokes::computeDt`, and then reduces the timestep by a safety factor.

3.5 The class `CCProjector`

The `CCProjector` class encapsulates all the functionality related to enforcing the incompressibility constraint, including all single-level and composite projections, and the functionality for the freestream preservation correction.

Member data of this class includes the correction for the advection-velocity projection (ϕ), the pressure from the level projection π, the synchronization projection correction e_s, and the data for the freestream preservation correction (e_Λ and $\vec{u}_p$).

-
  ```
  void levelMacProject(LevelData<FluxBox>& a_uEdge,
                       Real a_oldTime, Real a_dt)
  ```
  Projects the face-centered advection velocities using a face-centered projection (step # 1b in Section 2.5.2). Computes correction using a LevelSolve, then corrects the velocities in place by calling `CCProjector::applyMacCorrection`. The velocity field being projected is assumed to be at the half-time $t^\ell + \frac{1}{2}\Delta t^\ell$.
  Inputs:
  - `a_uEdge` – face-centered velocity field to project.
  - `a_oldTime` – solution time at beginning of timestep (t^ℓ).
  - `a_dt` – timestep (Δt^ℓ).

  Outputs:
  - `a_uEdge` – face-centered velocity field is corrected in place.

-
  ```
  void applyMacCorrection(LevelData<FluxBox>& a_uEdge,
                          Real CFscale)
  ```

  Computes the face-centered gradient of the correction field ϕ and subtracts it from the face-centered velocity field.
  Inputs:

  - `a_uEdge` – face-centered velocity field being corrected.
  - `a_CFscale` – scaling of coarse-level pressure field for boundary conditions. In general, this is set to $\frac{1}{2}\Delta t^{\ell}$.

  Outputs:

  - `a_uEdge` – face-centered velocity field corrected in place.

-
  ```
  void LevelProject(LevelData<FArrayBox>& a_velocity,
                    LevelData<FArrayBox>* a_crseVelPtr,
                    const Real a_newTime, const Real a_dt)
  ```

  Performs the single-level cell-centered projection (Step # 7 in Section 2.5.2) Computes correction using a LevelSolve, then corrects velocities in place by calling `CCProjector::correctCCVelocities`.
  Inputs:

  - `a_velocity` – cell-centered velocity field.
  - `a_crseVelPtr` – pointer to coarser-level velocity field at same time as `a_velocity`. (if it exists).
  - `a_newTime` – time at which velocity field is centered.
  - `a_dt` – time step (Δt^{ℓ}).

  Outputs:

  - `a_velocity` – cell-centered velocity field corrected in place.

-
  ```
  void correctCCVelocities(LevelData<FArrayBox>& a_velocity,
                           const Real a_scale) const
  ```

  Computes the cell-centered gradient of the level pressure π, then applies this correction to the cell-centered velocity field. $\vec{u} = \vec{u} + scale(G^{\ell}\pi)$.
  Inputs:

  - `a_velocity` – cell-centered velocity field.
  - `a_scale` – scaling to apply to $G^{\ell}\pi$.

  Outputs:

  - a_velocity – cell-centered velocity field corrected in place.

-
  ```
  void doSyncOperations(Vector<LevelData<FArrayBox>* >& a_velocity,
                        Vector<LevelData<FArrayBox>* >& a_lambda,
                        const Real a_newTime, const Real a_dtSync)
  ```

  Called from AMRNavierStokes::postTimeStep, this function allocates the AMR-Solvers for the multilevel synchronization solves and then calls CCProjector::doSyncProjection and CCProjector::computeVDCorrection. **Inputs:**

  - a_velocity – cell-centered multilevel velocity field.
  - a_lambda – multievel Λ field.
  - a_newTime – time at which synchronization is being applied.
  - a_dtSync – time step over which synchronization is being applied.

  Outputs:

  - a_velocity – cell-centered multilevel velocity field (synchronization projection is applied in place).

-
  ```
  void doSyncProjection(Vector<LevelData<FArrayBox>* >& a_velocity,
                        const Real a_newTime,
                        const Real a_dtSync,
                        AMRSolver& a_solver)
  ```

  Called from CCProjector::doSyncOperations, this function performs the multilevel synchronization projection (step #2 in Section 2.5.4), to ensure that the velocity field is divergence-free in a composite sense. This function uses the AMRSolver class to do the multilevel elliptic solve.
  Inputs:

  - a_velocity – multilevel cell-centered velocity field.
  - a_newTime – time at which velocity field is centered.
  - a_dtSync – time step for coarsest level being synchronized.
  - a_solver – AMRSolver to use for projection.

  Outputs:

  - a_velocity – velocity field projected in place.

-
  ```
  void computeVDCorrection(Vector<LevelData<FArrayBox>* >& a_lambda,
                           const Real a_newTime,
                           const Real a_dtSync,
                           AMRSolver& a_solver)
  ```

This function computes the freestrean preservation correction (step $\#$ 3 in Section 2.5.4), solving for e_Λ and computing $\vec{u}_p$ which will be used to correct future advection velocities.
Inputs:

- `a_lambda` – multilevel Λ field.
- `a_newTime` – time at which Λ is centered.
- `a_dtSync` – time step of coarsest level being synchronized.
- `a_solver` – `AMRSolver` object to use for elliptic solve.

-
  ```
  void doPostRegridOps(Vector<LevelData<FArrayBox>* >& a_velocity,
                       Vector<LevelData<FArrayBox>* >& a_lambda,
                       const Real a_dt, const Real a_time)
  ```

  This function, called from `AMRNavierStokes::postRegrid`, calling the `CCProjector::initialVelo` function to ensure that the newly initialized post-regridding velocity field is divergence-free, and then calling `CCProjector::computeVDCorrection` to re-initialize the freestream-preservation correction.
  Inputs:

  - `a_velocity` – multilevel velocity field.
  - `a_lambda` – multilevel Λ field.
  - `a_dt` – timestep of coarsest level being re-initialized.
  - `a_time` – current time.

  Outputs:

  - `a_velocity` – multilevel velocity field projected in place.

-
  ```
  void initialVelocityProject(Vector<LevelData<FArrayBox>* >& a_velocity,
                              AMRSolver& a_solver,
                              bool a_homogeneousCFBC = true)
  ```

  This function applies a composite projection to a multilevel vector field and corrects it in place to return a divergence-free vector field. If $\ell_{base} > 0$, only levels $\ell \geq \ell_{base}$ are corrected.
  Inputs:

  - `a_velocity` – multilevel cell-centered velocity field
  - `a_solver` – `AMRSolver` to use for multilevel solves.
  - `a_homogeneousCFBC` – if true, use a coarse correction boundary condition set to 0 (only relevant if the base-level is not 0), which is the case for all applications of this projection in the current algorithm.

Outputs:

  - `a_velocity` – multilevel velocity field corrected in place.

-
  ```
  void doInitialSyncOperations(Vector<LevelData<FArrayBox>* >& a_vel,
                               Vector<LevelData<FArrayBox>* >& a_lambda,
                               const Real a_newTime,
                               const Real a_dtSync)
  ```

  If we need to do the synchronization operations at initialization time (compute synchronization correction e_{sync} and freestream preservation correction e_{Λ}, this function computes them.
  Inputs:

  - `a_vel` – multilevel cell-centered velocity field.
  - `a_lambda` – multilevel Λ field.
  - `a_newTime` – current time.
  - `a_dtSync` – time step being used for initialization.

  Outputs:

  - `a_vel` – multilevel velocity field corrected in place.

-
  ```
  void initialSyncProjection(Vector<LevelData<FArrayBox>* >& a_velocity,
                             const Real a_newTime,
                             const Real a_dtSync,
                             AMRSolver& a_solver)
  ```

  At this point, simply a wrapper around `CCProjector::doSyncProjection`.

-
  ```
  void initialLevelProject(LevelData<FArrayBox>& a_velocity,
                           LevelData<FArrayBox>* a_crseVelPtr,
                           const Real a_oldTime,
                           const Real a_newTime);
  ```

  Called by `AMRNavierStokese::initializeLevelPressure`, performs the level projection for the initial pressure computation.

3.6 Important Chombo functionality used in Navier-Stokes computation

The `AMRINS` code relies heavily on the `Chombo` elliptic solvers, The `AMRSolver` class solves an elliptic equation on a hierarchy of AMR levels, while the `LevelSolver` class solves an elliptic equation on a single AMR level, but possibly with boundary condtions from a coarser AMR level. These classes are in the `Chombo AMRElliptic` library.

- `void LevelSolver::levelSolve(LevelData<FArrayBox>& phi,`
  ```
                                 const LevelData<FArrayBox>* phic,
                                 const LevelData<FArrayBox>& rhs,
                                 bool initializePhiToZero=true)
  ```

  Do a multigrid-based elliptic solve on a single AMR level. This function is called by `CCProjector::levelMacProject`, `CCProjector::LevelProject`, SpaceDim times by `AMRNavierStokes::computeUStar`, and also by `AMRNavierStokes::initializeLevelPre`

- `void AMRSolver::solveAMR(Vector<LevelData<FArrayBox> *>& a_phiLevel,`
  ```
                           const Vector<LevelData<FArrayBox> *>& a_rhsLevel)
  ```

  Do a multigrid-based composite elliptic solve over a set of AMR levels, possibly only a subset of the entire AMR hierarchy (for example, synchronization solves with $\ell_{base} > 0$). The specific algorithm used is described in [CGL+00] This function is called by `AMRNavierStokes::postTimeStep` (for the implicit refluxing step), `CCProjector::initialVelocityProject`, `CCProjector::computeVDCorrection`, and by `CCProjector::doSyncProjection`.

Chapter 4

User Interface Design

The AMRINS code is designed to be run from a command-line environment using an inputs file which defines various options such as the problem size, number of refinement levels, refinement ratios, etc. The inputs file is read by the code through the use of the `Chombo ParmParse` utility, which is documented in the Chombo users document.

Also, a user will generally provide a FORTRAN subroutine which initializes the velocity field to the initial conditions for the problem being run. Support is also provided for defining an initial vorticity distribution, from which an initial velocity field is computed.

Depending on the verbosity level specified in the inputs file, the AMRINS code will generally produce screen output detailing its progress through the run. In addition, if specified in the inputs file, hdf5 checkpoint files (for restarting) and plotfiles (for data analysis) are produced at specified intervals. Plotfile formats are compatible with `ChomboVis`.

The general syntax for running the AMRINS code is as follows, where `ns*.ex` is the executable file:
ns*.ex [inputs_file]

4.1 User Interface Components – Input files

The input files used by the Navier-Stokes codes consist of single-line variable definitions, in the format:
prefix.variableName = variableDefinition

The prefix is a way of segmenting the inputs file by the type of input data. In the AMRINS example inputs file, there are four prefixes in use: `main`, `ns`, `projection`, and `physBC`. The `main` prefix denotes inputs to the general problem scope, such as the number of cells on the base grid, refinement ratios, etc. The `ns` prefix denotes inputs to the general Navier-Stokes solver classes. The `projection` prefix is for inputs to the `CCProjector` class (generally, these are not necessary, since the CCProjector class's defaults should be appropriate for most situations. Inputs to the projection class generally do things like turning off the synchronization projection, tuning the coefficients for the freestream-preservation solve, etc. The `physBC` inputs are used to define the physical boundary condition types. If periodic boundary conditions are defined (in the `main` inputs), then these will have no effect, since periodic boundary conditions are not considered to be physical boundary conditions in `Chombo`.

The sample inputs file used for the vortex ring example used for evaluating the performance of this code is in Figure 4.1. Note also that the # character denotes a comment, and the rest of the line is skipped. A description of each of the lines in this file follows:

1. `main` inputs:

   - max_step – the maximum number of timesteps to compute
   - max_time – the maximum time to which the solution will be advanced . The code will stop whenever it reaches `max_time` or `max_step`.
   - num_cells – the number of cells in the base (level 0) mesh in each coordinate direction.
   - max_level – the maximum AMR level allowed in the computation.
   - ref_ratio – a list of the refinement ratios between levels. The first value is the refinement ratio between level 0 and level 1, the second is the refinement ratio between levels 1 and 2, and so on.
   - regrid_interval – the number of timesteps on each level between regridding.
   - block_factor – a grid-generation parameter which specifies the amount that each box is guaranteed to be coarsenable. For example, a block factor of 8 means that each grid box (and therefore each level) is guaranteeed to be coarsenable by at least a factor of 8. A larger block factor will produce "blockier" grids, which may cover the refined area less efficiently, but will also result in grids with better multigrid performance.
   - max_grid_size – a grid-generation parameter specifying the maximum allowable length in any dimension of an individual grid box. Grid boxes larger than the maximum size will be broken up into smaller boxes.
   - fill_ratio – a grid-generation parameter controlling how efficiently refined grids will cover the tagged regions. A higher fill ratio means that fewer untagged cells will be contained in the refined grids.
   - grid_buffer_size – a grid-generation parameter specifying the required nesting radius for proper nesting of refined grids.
   - checkpoint_interval – the number of level 0 timesteps between checkpoint files. A value of -1 means that no checkpoint files are generated.
   - checkPrefix – the string used as a base for the checkpoint filenames.
   - plot_interval – the number of level 0 timesteps between plot files. A negative value means that no plot files are generated. A value of 0 means that plot files are generated at the initial and final times. For any non-negative interval, a plotfile will also be generated at the final step.

```
main.max_step = 10    #max number of timesteps to compute
main.max_time = 16.0  #stop time
main.num_cells =  32 32 48  #base level domain
main.max_level = 2
main.ref_ratio = 4 4 4
main.regrid_interval = 4 4
main.block_factor = 8
main.max_grid_size = 32
main.fill_ratio = 0.8
main.grid_buffer_size = 1
main.checkpoint_interval = 50
main.checkPrefix = chk.
main.plot_interval = 10
main.plotPrefix = plt.
main.cfl = 0.5
main.verbosity = 2  #higher number means more verbose
#main.gridfile = grids.dat.128

main.is_periodic = 0 0  1

ns.init_shrink = 1.0
ns.tag_vorticity = 1
ns.vorticity_tagging_factor = 0.005

#initial grids
ns.specifyInitialGrids = 0
#ns.initialGridFile = grids.init
ns.initVelFromVorticity = 1

ns.viscosity = 0.000001

ns.num_scalars = 0

#inputs to the projection
#projection.doSyncProjection = 1
#projection.applyFreestreamCorrection = 1
#projection.eta = 0.9

# this is physical BC info
# 0 = solidWall, 1=inflow, 2=outflow, 3=symmetry, 4=noShear
physBC.lo = 4 4 4
physBC.hi = 4 4 4
```

Figure 4.1: Inputs file for vortex rings example

- plotPrefix – the string used as a base for the plotfile filenames.
- cfl – The Courant-Friedrich-Lewy (CFL) number to use when computing timesteps.
- verbosity – defines how much text output the run will generate. A higher number results in more output about what the code is doing.
- gridfile – If a fixed hierarchy run is being done, this file contains the refined-grid configuration to use. If this variable is defined, then no regridding will occur, and the hierarchy of refined grids will be fixed for the length of the run.
- is_periodic – A list of 0's and 1's which specifies whether to use periodic boundary conditions in each direction. The first entry is the x-direction, the second, the y-direction, and the third the z-direction (if in 3-D). A value of 1 specifies a periodic boundary condition, while 0 specifies a non-periodic boundary condition. The default is non-periodic boundary conditions.

2. `ns` inputs

   - init_shrink – the safety factor by which to multiply the initial timestep.
   - tag_vorticity – if 1, tag for refinement based on the magnitude of the vorticity.
   - vorticity_tagging_factor – Regridding parameter. Cells will be tagged for refinement if the undivided vorticity (vorticity*dx) is greater than the vorticity tagging factor.
   - specifyInitialGrids – If this is true (1), then the initial grid configuratoin will be specified (using the initialGridFile variable), but then adaptive regridding will be done as the solution progresses.
   - initialGridFile – if the specifyInitialGrids variable is 1, then this variable specifies the file containing the initial grid configuration.
   - initVelFromVorticity – if this variable is 1, then the initial velocity field is initialized from an initial vorticity field (generally initialized in the INITVORT subroutine in PROB_F.ChF).
   - viscosity – the kinetic viscosity ν of the fluid.
   - num_scalars – the number of additional passively advected and diffused scalars. These scalars /footnotePhil, should I completely remove any sign of having any advected/diffused scalars?. The default is 0.

3. `projection` inputs

   - doSyncProjecton – if 1, then apply multilevel synchronization projection. Default is 1 (apply the projection)

- applyFreestreamCorrection – if 1, then apply the freestream preservation correction to the face-centered advection velocities. The default is 1 (apply the correction)
- eta – the scaling factor used when computing the freestream preservation correction, and must be between 0 and 1. A larger value will result in a stronger correction, which will correct freestream preservation errors more quickly. The default is 0.9.

4. `physBC` inputs – the physBC inputs specify which physical boundary conditions to use as a series of integers, one integer for each coordinate direction. The integers correspond to physical boundary conditins as follows: 0 = solid wall boundary conditions, 1 = inflow boundary conditions, 2 = outflow boundary conditions, 3 = symmetry boundary conditions, and 4 = no-shear boundary conditions.

    - lo – the boundary conditions to use on the low side of the domain in each coordinate direction.
    - hi – the boundary conditions to use on the high side of the domain in each coordinate direction.

4.2 User Interface Components – FORTRAN files

Data initialization is generally done using a FORTRAN subroutine, which may be provided by the user by editing or replacing the `PROB_F.ChF` file in the executable directory. For initializing the velocity field, there are two possiblities. The velocity may be initialized directly, or a vorticity field may be initialized, and then a velocity field is computed from the initial vorticity field. If a velocity field is being defined analytically, then the `INITVEL` subroutine should be modified or replaced. If an initial vorticity field is being specified, this is set in the `INITVORT` subroutine. For initialization from a vorticity field, the "ns.initVelFromVorticity" variable in the inputs file must be set to 1 as well. The `PROB_F.ChF` file is compiled and linked along with the rest of the `AMRINS` code.

4.3 User Interface Components – Outputs

The AMRINS code generates hdf5 plotfiles which are readable with ChomboVis. Plotfile generation is managed by the `AMR::writePlotFile()` function, which calls the `AMRNavierStokes::writePlotLevel` and `AMRNavierStokes::writePlotHeader` functions. The `AMRNavierStokes::writePlotHeader` function writes out hdf5 file configuration information which is specific for the AMRINS application. The `AMRNavierStokes::writePlotLevel` function manages the data output for a single AMR level, making use of the `Chombo` hdf5 functionality in the `BoxTools` library.

The AMRINS code can also generate checkpoint files for restarting the computation. Checkpoint files generally contain more data than plotfiles, since they contain all the data needed to restart a computation, including pressure data from the `CCProjector` class. Checkpoint files are in hdf5 format, and are generated by the `AMR::writeCheckpointFile()` function, which calls the `AMRNavierStokes::writeCheckpointHeader` and `AMRNavierStokes::writeCheckpointLevel` functions. The `AMRNavierStokes::writeCheckpointHeader` function write out necessary configuration information to the checkpoint file, while the `AMRNavierStokes::writeCheckpointLevel` function writes all the data for the AMR level, and calls the `CCProjector::writeCheckpointLevel` function, which in turn writes out data from the `CCProjector` class.

Chapter 5

Checklist

- General Architecture Items

  1. *Is the system environment defined, including hardware, software, and external systems?*
  2. *Does the architecture clearly differentiate the problem domain, the user interface, task management, and data management? If not, are the reasons explained and justified?*
  3. *Does the architecture take into account the target environments? MPP? Clusters?*
  4. *Will the architecture accommodate any likely changes?*

- General Design Items

  1. *Are all the major components described?*
     Yes, except for the components of the `Chombo` libraries, which are described in [CGL+00].
  2. *Are there a sufficient number of diagrams and descriptions for all major aspects of the design to be fully understood?*
     I think so, but what do you, the reader, think?
  3. *Has the dataflow among the components been described?*
     Yes; the dataflow between `AMRNavierStokes` and `CCProjector` is outlined.
  4. *Are all the major algorithms identified?*
     A pseudocode description of the algorithm is provided in 2.7. Elliptic solver algorithms and other implementation details of `Chombo` classes may be found in [CGL+00]

- Modularity and Component Design

1. *Is the decomposition of system components logical and efficient?*
   Yes, although some refactoring of the `AMRNavierStokes` class is planned, in order to make it more manageable in size and also to facilitate reuse of its parts.
2. *Are module/component boundaries well defined, including functional interfaces?*
   Yes.
3. *Does the design minimize the number of component connections and interactions*
   Yes. As mentioned above, some refactoring is planned, which will complicate the number of components, and therefore the number of component interactions, but this will be an enhancement.
4. *Have all shared data and resources between components been described?*
   Yes.

- Dynamic System States

  1. *Area all significant system states/phases and events captured in the design?*
     To be honest, I think so, but I'm not completely sure what is meant by this question.

- Data Structures

  1. *Are all major data structures described?*
  2. *Is the conceptual view of composite data elements and objects documented? This applies to hierarchical data structures where inheritance might be used.*

- Functions and Routines

  1. *Are inputs to routines necessary and sufficient to perform the required operation?*
     Yes.
  2. *Do routines clearly state how the output is derived from input and shared data?*
     Yes, routines are fairly extensively commented.
  3. *Are all outputs produced by the routine being used?*
     Yes.

- Is there a strategy described for handling

  1. *Special states? (e.g abnormal termination, error recovery, losing power, etc.)*
     Using the checkpoint/restart capability allows one to restart the code from an intermediate point, in the event of a system failure. Also, `assert`'s are

used liberally in the code to attempt to catch implementation and algorithm problems.

2. *Failure of the system? (e.g. process termination, system recovery, etc.)*
   Same as #1.
3. *Memory management? Does it include memory use estimates?*
   Yes, memory usage is tracked. Maximum memory usage and memory leaks are reported.
4. *Shared resource management? Are the modules that use the shared resources indicated?*
   I don't think this is applicable to this code, other than possibly the use of hdf5 for data output?
5. *I/O? (file read/write, socket communication, user inputs, etc.)*
   Data output and restarting from a checkpoint are facilitated by the use of hdf5. User inputs are generally in the form of an inputs file, which is processed through the `Chombo` `ParmParse` utility.

Bibliography

[ABC$^+$98] A. S. Almgren, J. B. Bell, P. Colella, L. H. Howell, and M. Welcome. A conservative adaptive projection method for the variable density incompressible Navier-Stokes equations. *Journal of Computational Physics*, 142(1):1–46, May 1998.

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

[BCG89] J. B. Bell, P. Colella, and H. M. Glaz. A second-order projection method for the incompressible Navier-Stokes equations. *J. Comput. Phys.*, 85:257–283, 1989.

[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.

[Cho68] A. J. Chorin. Numerical solutions of the Navier-Stokes equations. *Math. Comp.*, 22:745–762, 1968.

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

[MC96] D. F. Martin and K. L. Cartwright. Solving Poisson's equation using adaptive mesh refinement. *Technical Report UCB/ERI M96/66 UC Berkeley*, 1996.

[MC00] D Martin and P Colella. A cell-centered adaptive projection method for the incompressible Euler equations. *J. Comput. Phys.*, 2000.

[Min94] Michael Minion. *Two Methods for the Study of Vortex Patch Evolution on Locally Refined Grids*. PhD thesis, U.C. Berkeley, May 1994.

[Min96] Michael L. Minion. A projection method for locally refined grids. *J. Comput. Phys.*, 127(1):158–178, Aug. 1996.

[TGA96] E.H. Twizell, A.B. Gumel, and M.A. Arigu. Second-order, l_0-stable methods for the heat equation with time-dependent boundary conditions. *Advances in Computational Mathmatics*, 6:333–352, 1996.