Chapter 1

centered.h Documentation

Version: 1.0 Updated: 2025-01-01

1.1 Introduction

This solver addresses the incompressible Navier-Stokes equations[4, 3]:

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u} \otimes \mathbf{u}) = \frac{1}{\rho}\left(-\nabla p + \nabla \cdot (2\mu \mathbf{D})\right) + \mathbf{a} \tag{1.1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{1.2}$$

where the deformation tensor is defined as $\mathbf{D} = \frac{1}{2}(\nabla \mathbf{u} + (\nabla \mathbf{u})^T)$.

For multiphase flows, the acceleration term $\mathbf{a}$ includes the effects of interfacial forces, such as surface tension. With the incompressibility constraint in equation 1.2, the velocity $\mathbf{u}$ and pressure gradient ∇p at the next time step can be determined using approximate projection methods. The discrete form is:

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} + \nabla \cdot (\mathbf{u} \otimes \mathbf{u})^{n+1/2} = \frac{1}{\rho^{n+1/2}}\left(-\nabla p^{n+1} + \nabla \cdot \left(2\mu^{n+1/2}\mathbf{D}^{**}\right)\right) + \mathbf{a}^{n+1/2} \tag{1.3}$$

$$\nabla \cdot \mathbf{u}^{n+1} = 0 \tag{1.4}$$

Here, the viscosity term is computed implicitly using the intermediate velocity $\mathbf{u}^{**}$.

The following subsections will provide a theoretical overview of each step, with practical program analysis presented in subsequent sections. Notably, in the following subsections, face-centered variables will be denoted by the subscript f, such as μ_f, to distinguish them from cell-centered variables. Readers are also referred to the excellent introduction written by Edoardo Cipriano [2], which has been a great inspiration for this documentation.

1.1.1 Algorithm Overview

To summarize, solving the Navier-Stokes equations involves four main steps: **Property Update, Advection, Viscosity, and Projection**, corresponding to distinct stages of the solution process. The first three steps involve solving the following equations:

$$\frac{c^{n+1/2} - c^{n-1/2}}{\Delta t} + \mathbf{F}(c, \mathbf{u}_f^n) = 0 \tag{1.5}$$

$$\rho^{n+1/2}\left[\frac{\mathbf{u}^{**} - \mathbf{u}^n}{\Delta t} + \nabla \cdot (\mathbf{u} \otimes \mathbf{u})^{n+1/2} - \mathbf{a}^{n-1/2} + \frac{\nabla p^n}{\rho^{n-1/2}}\right] = \nabla \cdot \left(2\mu_f^{n+1/2}\mathbf{D}^{**}\right) \tag{1.6}$$

$$\mathbf{u}^{***} = \mathbf{u}^{**} - \Delta t\left(\mathbf{a}^{n-1/2} - \frac{\nabla p^n}{\rho^{n-1/2}}\right) \tag{1.7}$$

Once $\mathbf{u}^{***}$ is computed, we obtain $\mathbf{u}^{n+1}$ by:

$$\mathbf{u}^{n+1} = \mathbf{u}^{***} + \Delta t\left(\mathbf{a}^{n+1/2} - \frac{\nabla p^{n+1}}{\rho^{n+1/2}}\right) \tag{1.8}$$

Using the incompressibility constraint:

$$\nabla \cdot \mathbf{u}^{n+1} = 0 \tag{1.9}$$

the pressure gradient ∇p^{n+1} can be computed by solving a Poisson equation in the **Projection** step.

Property Update

Equation 1.5 is the discrete form of advection equation for markers

$$\frac{\partial c}{\partial t} + \nabla \cdot (c\mathbf{u}_f^n) = 0 \tag{1.10}$$

the scheme of advection term $\mathbf{F}(c, \mathbf{u}_f^n)$ varies depending on the practical condition, the $\mathbf{u}_f^n$ here is the strict non-divergence face velocity obtained in the last final step (see section 1.1.4). If c denotes the volume of fluids and the header file `two-phase.h` is included, the geometric advection scheme in `vof.h` is activated. Otherwise, if c represents tracers, the BCG scheme is applied to solve the equation. For further details, refer to the `tracers.h` and `vof.h` documentation.

The density ρ and viscosity μ_f at the $n + 1/2$ time step are determined based on the distribution of c, this step is implemented in `two-phase-generic.h`. For those condition with large density ratio, there is option to smear the $\alpha_f = 1/\rho$ in the same header file, as it is considered in poisson equation solver. All these steps are achieved by event succession, note although the body force such as surface tension is associated with distribution of c, it is actually implemented in the **Projection** step also through succession of event 'acceleration'.

1.1.2 Advection

In this step, we merge the first two terms on the L.H.S. of Equation 1.6, solving:

$$\rho^{n+1/2}\left[\frac{\mathbf{u}^* - \mathbf{u}^n}{\Delta t} + \nabla\cdot(\mathbf{u}\otimes\mathbf{u})^{n+1/2}\right] = 0 \tag{1.11}$$

where $\mathbf{u}^*$ is the intermediate velocity. Given the face velocity $\mathbf{u}_f^{n+1/2}$, this equation is solved using functions provided by the header file `bcg.h`, following the BCG scheme. For further details on the BCG scheme, please refer to the `bcg.h` documentation.
The face velocity $\mathbf{u}_f^{n+1/2}$ is first computed using the BCG scheme and then projected to satisfy the non-divergence constraint, as required by the functions provided in `bcg.h`. Since these functions do not directly output values at $n+1/2$, this computation is performed separately in `centered.h`.

1.1.3 Diffusion

This step aims to compute $\mathbf{u}^{***}$. Starting from Equation 1.11, Equation 1.6 becomes:

$$\rho^{n+1/2}\left[\frac{\mathbf{u}^{**} - \mathbf{u}^*}{\Delta t} - \mathbf{a}^{n-1/2} + \frac{\nabla p^n}{\rho^{n-1/2}}\right] = \nabla\cdot\left(2\mu_f^{n+1/2}\mathbf{D}^{**}\right) \tag{1.12}$$

Here, $\mathbf{u}^{**}$ is computed implicitly, and $\mathbf{u}^{***}$ is subsequently determined by equation 1.7.

1.1.4 Projection

The final step enforces the non-divergence constraint. A key challenge is managing the conversion between cell-centered and face-centered values, as velocities are stored at cell centers, accelerations are stored at cell faces, and the projection function operates on face values.

From Cell-Centered to Face-Centered: Computing p^{n+1}

We compute the pressure p^{n+1} from:

$$\frac{\mathbf{u}_f^{n+1} - \mathbf{u}_f^{***}}{\Delta t} = -\frac{\nabla p^{n+1}}{\rho^{n+1/2}} + \mathbf{a}_f^{n+1/2} \tag{1.13}$$

$$\nabla\cdot\mathbf{u}_f^{n+1} = 0 \tag{1.14}$$

which simplifies to:

$$\nabla\left(\frac{\nabla p^{n+1}}{\rho^{n+1/2}}\right) = \nabla\left(\frac{\mathbf{u}_f^{***}}{\Delta t} + \mathbf{a}_f^{n+1/2}\right) \tag{1.15}$$

Using this relationship, the cell-centered pressure is computed with a multigrid solver applied to the face velocity:

$$\mathbf{u}_f^{****} = \frac{\mathbf{u}^{***}[] + \mathbf{u}^{***}[-1]}{2} + \Delta t\mathbf{a}_f^{n+1/2} \tag{1.16}$$

The divergence-free face velocity $\mathbf{u}_f^{n+1}$ is updated for computing the advection of tracers in the next time step.

From Face-Centered to Cell-Centered: Updating $\mathbf{u}^{n+1}$

With p^{n+1} computed, the cell-centered velocity $\mathbf{u}^{n+1}$ is updated:

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^{***}}{\Delta t} = -\frac{\nabla p^{n+1}}{\rho^{n+1/2}} + \mathbf{a}^{n+1/2} \tag{1.17}$$

To transform face-centered values to cell-centered values, the face-centered pressure gradient is computed and combined with the face-centered acceleration $\mathbf{a}_f^{n+1/2}$ to yield:

$$\mathbf{g}_f^{n+1} = \mathbf{a}_f^{n+1/2} + \frac{\nabla p^{n+1}}{\rho_f^{n+1/2}} \tag{1.18}$$

The cell-centered $\mathbf{g}^{n+1}$ is obtained by averaging:

$$\mathbf{g}^{n+1} = \frac{\mathbf{g}_f^{n+1}[1] + \mathbf{g}_f^{n+1}[]}{2} \tag{1.19}$$

Finally, the velocity is updated:

$$\mathbf{u}^{n+1} = \mathbf{u}^{***} + \Delta t \mathbf{g}^{n+1} \tag{1.20}$$

Approximate vs. Exact Projection

The approximate projection method, as initially described, pertains to an algorithm where the discrete Laplace operator does not precisely equal the discrete divergence of the discrete gradient, according to Almgren (2000)[1]. This situation arises exclusively on grids where only the cell-center is active, due to the misalignment caused by the face-centered vector. However, `Basilisk` circumvents this issue. Nonetheless, as previously discussed, while the face-centered velocity u_f maintains strict non-divergence, the cell-centered velocity u_c does not, as it is derived by subtracting the average of the updated pressure-acceleration at the face center. Consequently, this is why the projection method utilized by `Basilisk` is termed an approximate projection.

Bibliography

[1] A. S. Almgren, J. B. Bell, and W. Y. Crutchfield. "Approximate projection methods: Part I. Inviscid analysis". In: *SIAM J. Sci. Comput.* 22.4 (2000), pp. 1139–1159.

[2] E. Cipriano. *Introduction for centered.h.* `http://basilisk.fr/sandbox/ecipriano/doc/centered`. 2024.

[3] S. Popinet. "A quadtree-adaptive multigrid solver for the Serre–Green–Naghdi equations". In: *J. Comput. Phys.* 302 (2015), pp. 336–358.

[4] S. Popinet. "Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries". In: *J. Comput. Phys.* 190.2 (2003), pp. 572–600.

Chapter 2

double-projection.h Documentation

Version: 1.0 Updated: 2025-06-14

2.1 Introduction

This header file provides a method to update the velocity and pressure gradient separately, mitigating error accumulation in the pressure gradient due to a non-strictly divergence-free cell-centered velocity. This approach requires an additional projection step.

As discussed in the documentation for `centered.h`, the cell-centered velocity $\mathbf{u}^n$ is not strictly divergence-free. This is because it is computed by interpolating the source term (comprising acceleration and pressure gradient), which complements the strictly divergence-free face-centered velocity. Consequently, when updating the pressure gradient ∇p in the next time step by solving the Poisson equation for $\mathbf{u}^{***}$, defined as

$$\mathbf{u}^{***} = \mathbf{u}^n - \Delta t \nabla \cdot (\mathbf{u} \otimes \mathbf{u})^{n+1/2} + \frac{\Delta t}{\rho^{n+1/2}} \nabla \cdot \left(2\mu_f^{n+1/2} \mathbf{D}^{**} \right), \tag{2.1}$$

the non-divergence-free error in $\mathbf{u}^n$ accumulates, as $\mathbf{u}^{***}$ includes $\mathbf{u}^n$. As reported on the `Basilisk` website, this issue can be significant under certain conditions.

2.2 Solution Approach

To address this issue, we exclude $\mathbf{u}^n$ from $\mathbf{u}^{***}$ as therotically it satisfies $\nabla \cdot \mathbf{u}^n = 0$. The pressure gradient ∇p is then computed by solving

$$\nabla^2 p^{n+1} = \nabla \cdot \left[-\Delta t \nabla \cdot (\mathbf{u} \otimes \mathbf{u})^{n+1/2} + \frac{\Delta t}{\rho^{n+1/2}} \nabla \cdot \left(2\mu_f^{n+1/2} \mathbf{D}^{n} \right) \right]. \tag{2.2}$$

To maintain a divergence-free velocity field, an additional projection iteration is required. The velocity update process is now defined as follows:

$$\frac{\rho^{n+1/2}}{\Delta t}\left(\mathbf{u}^{**}-\mathbf{u}^{n}\right)+\rho^{n+1/2}\left[\nabla\cdot(\mathbf{u}\otimes\mathbf{u})^{n+1/2}+\frac{\nabla p^{n}}{\rho^{n-1/2}}\right]=\nabla\cdot\left(2\mu_{f}^{n+1/2}\mathbf{D}^{n+1/2}\right), \quad (2.3)$$

$$\nabla^{2}\delta p=\nabla\cdot\mathbf{u}^{**},\quad \mathbf{u}^{n+1/2}=\mathbf{u}, \quad (2.4)$$

$$\mathbf{u}^{n+1}=\mathbf{u}^{**}-\frac{\Delta t}{\rho^{n+1/2}}\nabla\delta p^{n}, \quad (2.5)$$

where δp is the temporary pressure gradient to correct the velocity field.

Chapter 3

bcg.h Documentation

Version: 2.0 Updated: 2025-06-04

3.1 Introduction and Background

`bcg.h` file contains two functions: *tracer_fluxes* and *advection* whose major purpose is to construct a solver for advective equation:

$$\frac{\partial \Phi}{\partial t} + (\mathbf{u} \cdot \nabla)\Phi = 0 \tag{3.1}$$

where Φ is the scalar and $\mathbf{u}$ is the velocity. The discrete time formulation of equation 3.1 reads:

$$\frac{\Phi^{n+1} - \Phi^n}{\Delta t} + \mathbf{A}^{n+\frac{1}{2}} = 0 \tag{3.2}$$

Where $\mathbf{A}^{n+\frac{1}{2}}$ is the abbreviation of advection term. With the conservative constraints $\nabla \cdot \mathbf{u} = 0$, the integral form reads:

$$\int_\Gamma A^{n+\frac{1}{2}} = \int_\Gamma [(\mathbf{u} \cdot \nabla)\Phi]^{n+\frac{1}{2}} = \int_\Gamma [\nabla \cdot (\mathbf{u}\Phi)]^{n+\frac{1}{2}} = \int_{\partial\Gamma} (\mathbf{u}^{n+\frac{1}{2}} \cdot \mathbf{n})\Phi^{n+\frac{1}{2}} \tag{3.3}$$

where $\mathbf{n}$ represents the normal direction of cell interface. For cartesian cell employed in Basilisk, the calculation turns out to be

$$\Delta A^{n+\frac{1}{2}} = \sum_d s_d u_{f,d}^{n+\frac{1}{2}} \Phi_{f,d}^{n+\frac{1}{2}} \tag{3.4}$$

Note in this documentation, the subscript f denotes face value stored in staggered mesh[2], otherwise it is cell-average value stored in center of the cell. Subscript d is the notation of specific face (e, w, n, s for 2D and e, s, n, s, f, b for 3D) for a single cell. In addition, readers may found algebraic notation without d such as u_f in the upcoming discussion which refers to the same thing. This is because in the most cases d is used to addressed spatial relationship between face and cell from cell-perspective, such relation shall vanish from face-perspective and so does d.

The following discussions are conducted under 2D condition and results of 3D is easily to obtain accordingly. s_d represents sign function which reads

$$s_d = \begin{cases} 1 & d = e, n \\ -1 & d = w, s \end{cases} \tag{3.5}$$

The problem now becomes how to obtain $u_f^{n+1/2}$ and $\Phi_f^{n+1/2}$. Basilisk employs BCG scheme to tackle such issue. The scheme, which is originally named by three authors[1], stems from the algorithm intending to apply second order Godunov method to incompressible flow. It obtains value at $n + \frac{1}{2}$ not by averaging but by considering the Taylor series, for each cell:

$$\tilde{\Phi}_{f,d}^{n+\frac{1}{2}} = \Phi^n + s_d \frac{\Delta}{2} \frac{\partial \Phi^n}{\partial x_d} + \frac{\Delta t}{2} \frac{\partial \Phi^n}{\partial t} + O(\Delta^2, \Delta t^2) \tag{3.6}$$

Where $x_d = x$ for $d = e, w$ and $x_d = y$ for $d = n, s$. Replacing $\frac{\partial \Phi^n}{\partial x_d}$ with Euler equation 3.1 yielding

$$\tilde{\Phi}_{f,d}^{n+\frac{1}{2}} = \Phi^n + [s_d \frac{\Delta}{2} - \frac{\Delta t}{2} u_d] \frac{\partial \Phi^n}{\partial x_d} - \frac{\Delta t}{2} u_o \frac{\partial \Phi^n}{\partial x_o} \tag{3.7}$$

where u_d is the cell-centered value on d direction (i.e. u_x for $d = e, w$, u_y for $d = n, s$). Subscript o represents directions other than d. Since aiming equation 3.1 is advection-only, a simple upwind scheme is therefore employed[4, 3] to adapt each term in equation 3.7:

$$\tilde{\Phi}_{f,d}^{n+\frac{1}{2}} = \Phi^n + [s_d \frac{\Delta}{2} - max(\frac{\Delta t}{2} u_d, 0)] \frac{\partial \Phi^n}{\partial x_d} - \frac{\Delta t}{2} u_o \overline{\frac{\partial \Phi^n}{\partial x_o}} \tag{3.8}$$

$$\overline{\frac{\partial \Phi^n}{\partial x_o}} = \begin{cases} (\Phi^n[0] - \Phi^n[-1])/\Delta & if\ u_o > 0 \\ (\Phi^n[1] - \Phi^n[0])/\Delta & if\ u_o < 0 \end{cases} \tag{3.9}$$

If not additionally declared, $\frac{\partial \Phi^n}{\partial x_d}$ is computed with central scheme. Given four face values from each cell, each face now has two values from its neighbor cell which are left value and right value in traditional Godunov method. Unlike the traditional method which obtains the final face value $\Phi_{f,d}$ by solving Riemann problem, the upwind scheme is again applied to determine $\Phi_{f,d}$ in current algorithm (the value is counted from each face hence the subscript d is omitted.)

$$\Phi_f^{n+1/2}[0] = \begin{cases} \tilde{\Phi_f}[0] & u_f^{n+1/2} < 0 \\ \tilde{\Phi_f}[-1] & u_f^{n+1/2} > 0 \\ \frac{1}{2}(\tilde{\Phi_f}[0] + \tilde{\Phi_f}[-1]) & u_f^{n+1/2} = 0 \end{cases} \tag{3.10}$$

where u_f is velocity on the same face and marker $[0], [-1]$ indicate the cell from which the face value is obtained. In mesh system of Basilisk, for face located at $(i - 1/2, j)$ the $[0]$ (resp. $[-1]$) represents cell positioned at (i, j) (resp. $(i-1, j)$). It is worth mentioning that the marker in current equation is associated with the direction. For example, if one calculates face value $\Phi_f^{n+1/2}$ on x direction, the marker in equation 3.10 indicates the relationship on x direction while those in equation 3.9 refers to relation on y direction. Above discussion is the original algorithm on the paper, compensation is made in real code which shall be introduced in the following sections.

3.2 Functions

3.2.1 tracer_fluxes

Given face velocity $u_f^{n+1/2}$, the face flux $u_f^{n+\frac{1}{2}}\Phi_f^{n+\frac{1}{2}}$ is computed and stored on staggered mesh.

Parameters

Name	Data type	Status	Option/Default	Representation (before/after)
f	scalar	unchanged	compulsory	Φ^n
uf	face vector	unchanged	compulsory	$u_f^{n+\frac{1}{2}}$
flux	face vector	**output**	compulsory	$u_f^{n+\frac{1}{2}}\Phi_f^{n+\frac{1}{2}}$
dt	double	unchanged	compulsory	Δt
src	scalar	unchanged	compulsory	a^n

Worth Mentioning Details

As discussed previously, specific face value on staggered mesh[2] can be located by its spatial relation with corresponding cell. Take cell at (i, j) as an example, figure 3.1 demonstrates notation of each face. Moreover such value can be iterated by calling macro $foreach\_face()$ in Basilisk. Thanks to this feature, the code can be constructed from face-perspective starting from equation 3.10 rather than iterating every cell and compare two values for single faces.

Notably, different from original algorithm, two compensations are made in performing equation 3.10 and equation 3.8 respectively. For the former one, the condition where $u_f^{n+1/2} = 0$ merges into the second condition, the filter now reads:

$$\Phi_f^{n+1/2}[0] = \begin{cases} \tilde{\Phi}_f[0] & u_f^{n+1/2} < 0 \\ \tilde{\Phi}_f[-1] & u_f^{n+1/2} \geq 0 \end{cases} \tag{3.11}$$

For the latter one, u_d is replaced by face value $u_f[]$ for simplicity in boundary condition settings. Consequently the upwind filter (i.e. the $max()$) is no longer needed in the second term of equation 3.8 since upwind scheme is automatically met owing to upwind selection originally conducted by equation 3.11.

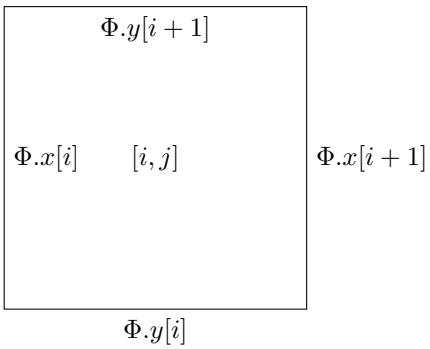


Figure 3.1: Example of staggered mesh.

Given $u_f^{n+1/2}$, the procedure to compute face flux now becomes:

1. Identify the upstream direction based on $u_f^{n+1/2}$.
2. Choose corresponding cell to compute face flux based on equation 3.10 and upstream direction obtained in previous step.
3. According to equation 3.8 3.9, add contribution from each direction and multiply the result by $u_f^{n+1/2}$ to obtain $u_f^{n+1/2}\Phi_f^{n+1/2}$.

Yet another detail worth mentioning is the source term a^n in current function. This is indeed preparation for solving non-Euler equation such as NS equation in 'centered.h'. Albeit compulsory here, such term is overall optional in the final function **advection** which shall be introduced in the upcoming section.

Program Workflow

Starting Point
input:
$f = \Phi^n$ $uf = u_f^{n+\frac{1}{2}}$
$flux$(empty) dt=Δt
$src = \mathbf{g}^n$
gradient:
$g = \nabla$ $f = \nabla\Phi$

```
1 void tracer_fluxes (scalar f,
2                     face vector uf,
3                     face vector flux,
4                     double dt,
5                     (const) scalar src)
6 {
```

```
7   vector g[];
8   gradients ({f}, {g});
```

Compute un[a]:

$un = \frac{u_{f,d}^{n+1/2}\Delta t}{fm.x[i]\Delta}$

Identification of upstream direction

The notation of corresponding cell i is identified according to equation 3.11.

$s = s_d$

[a] A compensation is made here, see 3.2.1 for detailed information

```
1   foreach_face() {
2     double un = dt*uf.x[]/(fm.x[]*Delta + SEPS), s = sign(un);
3     int i = -(s + 1.)/2.;
```

Contribution of Each Direction

Main direction

$f2 = \Phi^n[i] + \frac{\Delta}{2}[s_d - \frac{\Delta t}{\Delta}u_{f,d}^{n+1/2}]\frac{\partial \Phi^n}{\partial x_d}[i] + \frac{\Delta t}{2}\frac{a^n[-1]+a^n[0]}{2}$

Traversal direction

compute u_o in equation 3.8:

$vn(wn) = (u_{f,o}^{n+1/2}[i,0] + u_{f,o}^{n+1/2}[i,1])/(fm.e[i,0] + fm.e[i,1])$

component of other direction:

The gradient $\overline{\frac{\partial \Phi^n}{\partial x_o}}$ is first computed by upwind scheme based on direction of u_o according to equation 3.9

$fyy(fzz) = \overline{\frac{\partial \Phi^n}{\partial x_o}}$

$f2 \mathrel{-}= \frac{\Delta t}{2}u_o\overline{\frac{\partial \Phi^n}{\partial x_o}}$

```
1     double f2 = f[i] + (src[] + src[-1])*dt/4. + s*(1. -
      ↪  s*un)*g.x[i]*Delta/2.;
2     #if dimension > 1
3     if (fm.y[i] && fm.y[i,1]) {
4       double vn = (uf.y[i] + uf.y[i,1])/(fm.y[i] + fm.y[i,1]);
5       double fyy = vn < 0. ? f[i,1] - f[i] : f[i] - f[i,-1];
```

```
      f2 -= dt*vn*fyy/(2.*Delta);
    }
    #endif
    #if dimension > 2
    if (fm.z[i] && fm.z[i,0,1]) {
      double wn = (uf.z[i] + uf.z[i,0,1])/(fm.z[i] + fm.z[i,0,1]);
      double fzz = wn < 0. ? f[i,0,1] - f[i] : f[i] - f[i,0,-1];
      f2 -= dt*wn*fzz/(2.*Delta);
    }
    #endif
```

Assemble

assemble the final result, current direction is d

flux.d=$u_{f,d}^{n+\frac{1}{2}}\Phi_{f,d}^{n+\frac{1}{2}}$

after the $foreach\_face$ macro, the face flux on every face is obtained.

```
    flux.x[] = f2*uf.x[];
  }
}
```

3.2.2 advection

Given face flux $u_f^{n+1/2}\Phi_f^{n+1/2}$, compute $\Phi^{n+1} = \Phi^n - \Delta t A^{n+1/2}$.

Parameters

Name	Data type	Status	Option/Default	Representation (before/after)
tracers	scalar*	update	compulsory	Φ^n/ $\Phi^n - \Delta t A^{n+\frac{1}{2}}$
u	face vector	unchanged	compulsory	$u_f^{n+\frac{1}{2}}$
dt	double	unchanged	compulsory	Δt
src	scalar*	unchanged	optional/NULL	$\mathbf{g}^n$

Worth Mentioning Details

The data type *scalar∗* indicates the input can be vector, and each components will be treated as *scalar* type data and are assembled into one *vector* data eventually.

Given the face flux $u_f^{n+1/2}\Phi_f^{n+1/2}$, according to equation 3.1, for each cell the Φ^{n+1} yielding:

$$\Phi^{n+1} = \Phi^n - \Delta t \sum_d s_d u_{f,d}^{n+\frac{1}{2}} \Phi_{f,d}^{n+\frac{1}{2}} \tag{3.12}$$

take $flux = u_f^{n+1/2}\Phi_f^{n+1/2}$ for cell located at (i, j) the above equation reads

$$\Phi^{n+1}[i,j] = \Phi^n[i,j] + \Delta t(flux.x[i] - flux.x[i+1] + flux.y[j] - flux.y[j+1]) \tag{3.13}$$

Program Workflow

Starting Point
input:
tracers $= \Phi^n$ *u* $= u_f^{n+\frac{1}{2}}$
dt$=\Delta t$ *src* $= a^n$

```
struct Advection {
  scalar * tracers;
  face vector u;
  double dt;
  scalar * src; // optional
};
void advection (struct Advection p)
{
  scalar * lsrc = p.src;
  if (!lsrc)
    for (scalar s in p.tracers)
      lsrc = list_append (lsrc, zeroc);
  assert (list_len(p.tracers) == list_len(lsrc));
```

↓

Fluxes Compute
Traversal each elements in *tracers* (if *tracers* is vector, then this step traversal component on every direction)
computation:
flux $= \Phi_f^{n+\frac{1}{2}} u_f^{n+\frac{1}{2}}$

```
scalar f, src;
for (f,src in p.tracers,lsrc) {
  face vector flux[];
  tracer_fluxes (f, p.u, flux, p.dt, src);
```

Update

$$tracers(\text{updated}) = \Phi^{n+1}[i,j] = \Phi^{n}[i,j] + \Delta t(flux.x[i] - flux.x[i+1] + flux.y[j] - flux.y[j+1])$$

```
  #if !EMBED
      foreach()
        foreach_dimension()
          f[] += p.dt*(flux.x[] - flux.x[1])/(Delta*cm[]);
  #else // EMBED
      update_tracer (f, p.u, flux, p.dt);//This is a function that
       ↪ induced by embed.h which conducts same procedure with
       ↪ special care taken for embed boundary.
  #endif // EMBED
    }

    if (!p.src)
      free (lsrc);
  }
```

3.3 Calculation of Face Centered Normal Velocity

Readers may find all the algorithms constructed in this headfile are based on one condition: $u_f^{n+1/2}$ are known. In fact when solving a NS equation, the face velocity $u_f^{n+1/2}$ is obtained in a similar method (i.e. BCG scheme) in the NS solver before calling the functions discussed in current doc[5]. The difference majorly manifests in two perspectives: 1) the criteria in equation 3.11 becomes face velocity computed by center scheme from adjacent cell-centered value and u_o in equation 3.9 is the cell-centered value; 2) the obtained face velocity is projected to satisfy no-divergence condition. The detailed discussion will be presented in documentation for 'centered.h'.

Bibliography

[1] J. B. Bell, P. Colella, and H. M. Glaz. "A second-order projection method for the incompressible Navier-Stokes equations". In: *J. Comput. Phys.* 85.2 (1989), pp. 257–283.

[2] F. H. Harlow and J. E. Welch. "Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface". In: *Phys. Fluids* 8.12 (1965), pp. 2182–2189.

[3] D. F. Martin. *An adaptive cell-centered projection method for the incompressible Euler equations*. 1998.

[4] D. F. Martin and P. Colella. "A cell-centered adaptive projection method for the incompressible Euler equations". In: *J. Comput. Phys.* 163.2 (2000), pp. 271–312.

[5] S. Popinet. "Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries". In: *J. Comput. Phys.* 190.2 (2003), pp. 572–600.

Chapter 4

poisson.h Documentation

Version: 1.0 Updated: 2025-06-07

4.1 Introduction and Background

`poisson.h` serves as toolbox which provides functions to construct V-cycle iteration solver for implicit equations. A specific one for solving poisson equation is constructed within the headfile as an example. We shall first introduce the constructing toolbox.

Assuming the governing equation can be written as

$$f(x) = y \tag{4.1}$$

where y is the known variable, x is the desired variable and f represents linear operator that satisfies

$$f(x_a + x_b) = f(x_a) + f(x_b) \tag{4.2}$$

Now consider the discrete form of operator $\hat{f}$ which takes all desired variable from every cell (suppose the total number of cell is n) to express the local known variable y_i then yields the implicit equation group

$$\hat{f}(x_1, x_2, x_3, \cdots, x_n) = y_i \quad i = 1, 2, \cdot, n \tag{4.3}$$

which can be solved by indirective iterative method such as Jacobi method, G-S method[1] *etc.* Moreover, constraints 4.2 provide another perspective to construct equation group. Use $x_1^e, x_2^e, \cdots, x_n^e$ to denote exact solution of equation 4.3 and $x_1^k, x_2^k, \cdots, x_n^k$ to represent result of kth iteration. Following equation 4.2 we have

$$\hat{f}(\delta x_1^k, \delta x_2^k, \cdots, \delta x_n^k) = RES^k \tag{4.4}$$

where $\delta x_i^k = x_i^e - x_i^k$, $RES^k = \hat{f}(x_1^e, x_2^e, \cdots, x_n^e) - \hat{f}(x_1^k, x_2^k, \cdots, x_n^k)$. The criterion of solution then becomes $|RES^k|_\infty < \epsilon$ where ϵ is a setting tolerance.

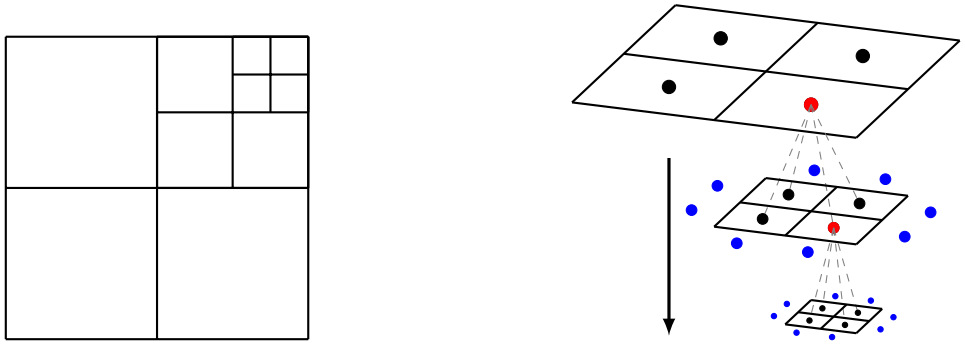
There are many techniques to accelerate the convergence of iteration, and multigrid method[4] may be one of the most famous which employs iterations on every layer of the mesh to reduce the residual of corresponding wavenumber. A similar methodology is applied by quadtree/octree in Basilisk. Take quadtree as an exapmle. Consider tree

architecture in figure 4.1a, the actual calculating rules for this problem is shown in 4.1b where ● represents leaf cells (the finest cell at this area and is not divided by higher level) and the value it carrying is the the final value shown in the result called active value. ● represents ghost cell served as boundary condition whose value is computed by bilinear interpolation. Finally ● represents value carried by parent cell. The parent cell, indicated by its name, will be divided into 4(8) children cells in finer layer (level in Basilisk).[3]

A single round of iteration is accomplished by two procedures. First, from highest level to lowest one, assign residual to each cell of current level which form the *R.H.S.* of equation 4.4. Second, starting from lowest level to the highest, obtain the result after few iterating (by Jacobi method or GS method) on current level and use it to compute initial value on next level. We shall first dive into second procedure which is more sophisticated.

Calculations happens at every level shown in figure 4.1b, when it comes to higher level the boundary condition is first set and then undergoes the iteration on cells at same level instead of whole domain. Moreover, the initial value on each level is obtained by prolongation (bilinear mostly) from previous mesh level.

In order to facilitate equation 4.4 we also need residual, which only exists at leaf cell, of every cell at each level. This procedure is achieve by restricting[2] (averaging mostly) value on 4(8) children cells, which is much simpler compared to bilinear that use in previous description.



(a) 2D quadtree example. (b) Calculation for each level.

Figure 4.1: Quadtree example. Arrow in (b) indicates calculating sequence.

After introducing the mesh architecture, we shall now step a little further to see the solver structure provided by 'poisson.h' and to perceive the overall workflow. figure 4.2 displays whole system as well as its workflow. As can be seen from the sketch, the whole solver consists of four functions, **mg_solve**, **mg_cycle**, **relax** and **residual**. Their nesting relating is shown by corresponding position, *e.g.* **relax** is inside **mg_cycle** while **residual** and **mg_cycle** locate inside **mg_solve** indicates that **relax** is called by **mg_cycle** and **mg_cycle** along with **residual** are directly called by **mg_solve**. Detailed workflow is also presented, after inputting $\mathbf{x}^0, \mathbf{y}$ before the residual actually meet the tolerance ϵ, **mg_solve** plays as a manager to make rest functions coordinate, $\mathbf{x}^k$ is conveyed between **mg_cycle** and **residual** to renew. Number behind each step represents the order within the loop. $\mathbf{x}^k, \mathbf{y}$ is first sent to residual to compute residual RES^k which served as parameter in **mg_cycle**. $\mathbf{x}^k$ and n are also taken into **mg_cycle** where n controls iteration number on each mesh level. $\mathbf{x}^{k+1}$ is obtained by first solving equation 4.4 for $\delta\mathbf{x}^k$ then

execute update

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \delta\mathbf{x}^{k+1} \tag{4.5}$$

Loop will break out either residual satisfies tolerance constraint or number of round exceed setting threshold. Readers may notice there is no parameters conveyed within **mg_cycle**, this is because relationship between **relax** and **mg_cycle** cannot be simply abstracted as 'linear' as depicted in this figure. Structure inside **mg_cycle** is demonstrate in figure 4.3 as described before residual is assigned to each level then relax is called at each level multiple times updating $\delta\mathbf{x}^k$ in the form (condition varies according to iteration method)

$$\delta x_i^{k+1} = F(\delta x_1^k, \delta x_2^k, \cdots, \delta x_{i-1}^k, \delta x_{i+1}^k, \cdots, \delta x_n^k, RES^k) \tag{4.6}$$

Back to **mg_solve**, readers may notice from figure 4.2 that all the function within, including **mg_solve** itself, are divided into three layer by dashed line and each layer is named by Roman number from top to bottom. Higher the layer, more irreplaceable the function is. Therefore, functions at III can be changed or altered based on one's purpose. In another word, users can choose their own **relax** and **residual** based on equation they cope with. The governing equation for `poisson.h` is

$$L(a) = \nabla \cdot (\alpha \nabla a) + \lambda a = b \tag{4.7}$$

where L is a linear operator. Based on above discussion such equation can be solved by multigrid solver only if one constructs appropriate **relax** and **residual** function. Another example is referred to headfile `viscosity.h` where same solver construction is used for totally different linear equation.

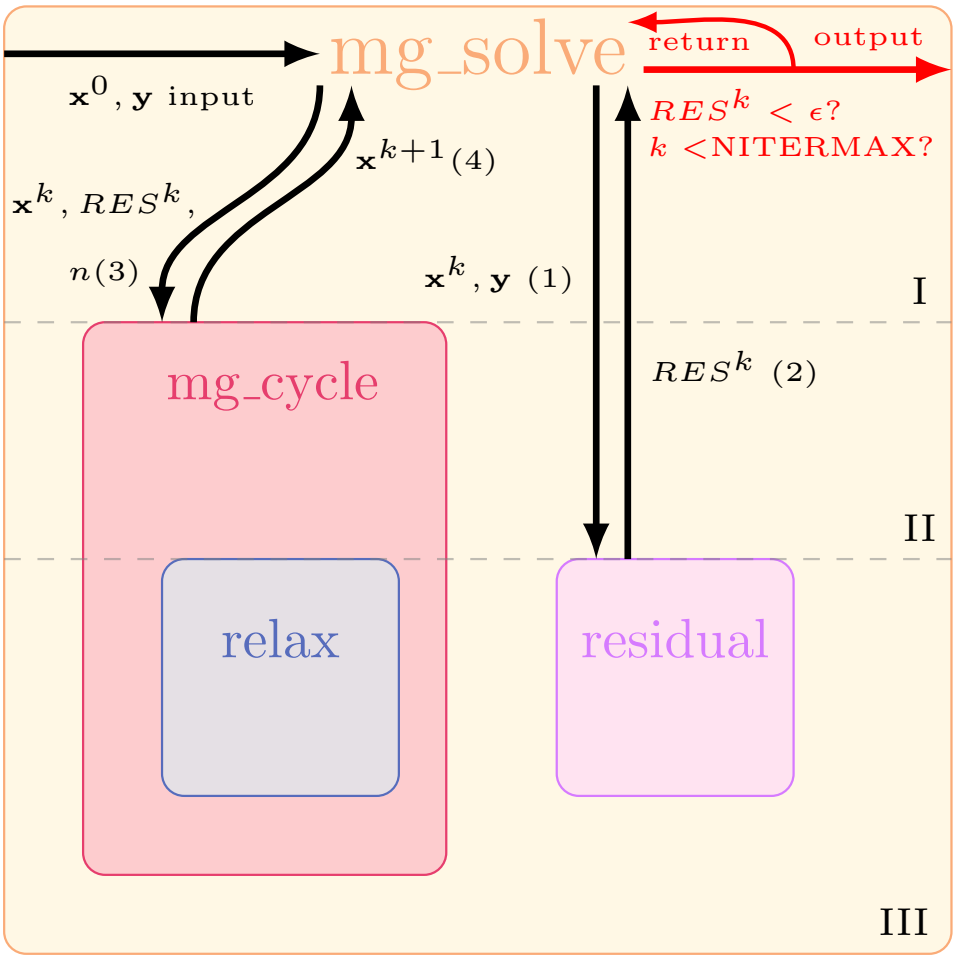


Figure 4.2: Architecture of the solver. Nested relationship of functions is indicated by box containing relationship *e.g.* **mg_cycle** contains **relax** but not **residual** indicates that **relax** is called in **mg_cycle** while **residual** is called in **mg_solve**.

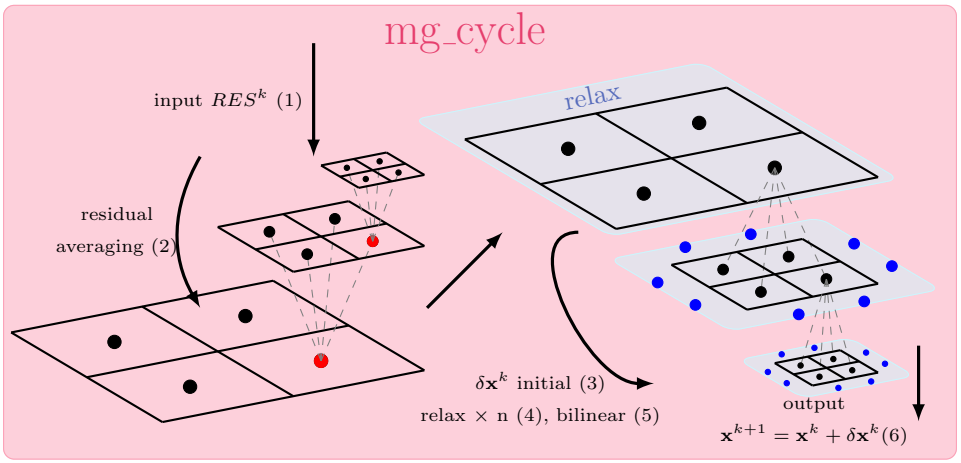


Figure 4.3: Combination between **mg_cycle** and **relax**. The 'round' of iteration described before is also demonstrated in a detailed way. **relax** herein is embed into every level of the mesh and is executed several times (depends on parameter n) on each level to accelerate convergence.

4.2 Multigrid Solver

As indicated in section 4.1, we here first introduce general structure of multigrid solver which consists of **mg_solve** and **mg_cycle**.

4.2.1 mg_cycle

Parameters

Name	Data type	Status	Option/Default	Representation (before/after)
a	scalar*	update	compulsory	$\mathbf{x}^k/\mathbf{x}^{k+1}$
res	scalar*	unchanged	compulsory	RES^k
da	scalar*	unchanged	compulsory	$\delta\mathbf{x}^k$
relax	void*	unchanged	compulsory	**relax**
data	void*	unchanged	compulsory	**Poisson** (struct defined below)
nrelax	int	unchanged	compulsory	n
minlevel	int	unchanged	compulsory	*minlevel*
maxlevel	int	unchanged	compulsory	*maxlevel*

Worth Mentioning Details

As described in section 4.1 and figure 4.3 function **mg_cycle** serves as a subcomponent to update the result. Details of such function have been explored before and shall not be repeated here.

Program Workflow

Starting Point
input:
$a = x^k$ $res = RES^k$ da=δx^k
relax=relax function *data* = Poisson structure
nrelax=times relax function applied to each level
minlevel=minimum of mesh level
maxlevel=maximum of mesh level

```
void mg_cycle (scalar * a, scalar * res, scalar * da,
               void (* relax) (scalar * da, scalar * res, int depth,
                ↪  void * data),
               void * data, int nrelax, int minlevel, int maxlevel)
{
```

RES restriction
To restrict residual from finer mesh to coarser mesh.
Entering of iterative
Iterate starting form lowest (coarsest mesh) level.

```
    restriction (res);
    minlevel = min (minlevel, maxlevel);
    for (int l = minlevel; l <= maxlevel; l++) {
```

Initial for Iteration
Set initial guess for each level. If is on the coarsest level, the initial guess is set to 0 otherwise the initial guess comes from bilinear interpolation of coarser level.

```
    if (l == minlevel)
      foreach_level_or_leaf (l)
        for (scalar s in da)
          foreach_blockf (s)
            s[] = 0.;
```

```
6     else
7       foreach_level (l)
8         for (scalar s in da)
9           foreach_blockf (s)
10            s[] = bilinear (point, s);
```

Relax on each level

Boundary condition (value of blue point in figure 4.3) is first calculated and assigned at certain level.

After setting the initial value and boundary condition, relaxation is then conduct on each level by employing **relax**. At the same time, $\delta\mathbf{x}$ is computed and restored in *da*.

```
1     boundary_level (da, l);
2     for (int i = 0; i < nrelax; i++) {
3       relax (da, res, l, data);
4       boundary_level (da, l);
5     }
6   }
```

Final update

$$\mathbf{x}^{k+1} = \mathbf{x}^{k} + \delta\mathbf{x}$$

```
1   foreach() {
2     scalar s, ds;
3     for (s, ds in a, da)
4       foreach_blockf (s)
5         s[] += ds[];
6   }
7 }
```

4.2.2 mg_solve

Parameters

Name	Data type	Status	Option/Default	Representation (before/after)
a	scalar*	update	compulsory	$\mathbf{x}^0/\mathbf{x}^{final}$
b	scalar*	unchanged	compulsory	$\mathbf{y}$
residual	scalar*	unchanged	compulsory	**residual**
relax	void*	unchanged	compulsory	**relax**
data	void*	unchanged	compulsory	**Poisson** (struct defined below)
nrelax	int	unchanged	optional/4	n
res	scalar*	unchanged	compulsory	RES
minlevel	int	unchanged	optional/0	$minlevel$
tolerance	double	unchanged	optional/10^{-3}	ϵ

Worth Mentioning Details

The variable δx is initialized in the current function and passed to **mg_cycle** for updates. According to the superposition principle, the boundary condition for δx must be homogeneous in order to satisfy the boundary conditions of the original variable x. This requirement is enforced in the function using the postfix '_homogeneous'. For more details, see the documentation on homogeneous boundary conditions.

Two optional components **residual** and **relax** are input as void pointer in this function. The return of this function is of self-defined data structure 'mgstats' which indeed contains information of the whole multigrid circle. The information includes *i*: the total number **mg_cycle** is employed inside **mg_solve**, *resb* and *resa*: maximum residual before/after the cycle.

Program Workflow

Initial Settings
NITERMAX=100 and *NITERMIN*=1 is the maximum and minimum times **mg_cycle** employed by **mg_solve**
Self-defined structure 'mgstats' serves as return value for all multigrid solver. It contains basic information about this circle.

```
int NITERMAX = 100, NITERMIN = 1;
double TOLERANCE = 1e-3 [*];

typedef struct {
  int i;              // number of iterations
  double resb, resa;  // maximum residual before and after the
   ↪  iterations
```

```
  double sum;          // sum of r.h.s.
  int nrelax;          // number of relaxations
  int minlevel;        // minimum level of the multigrid hierarchy
} mgstats;
```

Input

a = $\mathbf{x}^0$, *b*=$\mathbf{y}$, *nrelax* is the one used in **mg_cycle**
minlevel is the coarsest mesh level user would like to iterate.
tolerance=ϵ,
relax and *residual* are two pointers points to related function **residual** and **relax**.

```
mgstats mg_solve (scalar * a, scalar * b,
                  double (* residual) (scalar * a, scalar * b, scalar *
                   ↪  res, void * data),
                  void (* relax) (scalar * da, scalar * res, int depth,
                   ↪  void * data),
                  void * data = NULL,
                  int nrelax = 4,
                  scalar * res = NULL,
                  int minlevel = 0,
                  double tolerance = TOLERANCE)
{
```

Pointer Preparation

Note that 'list_clone' directly copy the data to the pointer address instead of point to the original address. Therefore *da* and *a* are two pointers with same data but different address while *pre* point to *res*. If *res* points NULL, *pres* will directly points to the same address. Note the change of *res* hereinafter shall not influence the NULL address of *pres*.

```
  scalar * da = list_clone (a), * pres = res;
  if (!res)
    res = list_clone (b);
```

Boundary and initial information settings

Homogeneous boundary conditions are applied to $\delta\mathbf{x}$. Return value s is initialized. $s.sum$ is the summation of R.H.S. of equation 4.7.

```
for (int b = 0; b < nboundary; b++)
  for (scalar s in da)
    s.boundary[b] = s.boundary_homogeneous[b];

mgstats s = {0};
double sum = 0.;
scalar rhs = b[0];
foreach (reduction(+:sum))
  sum += rhs[];
s.sum = sum;
s.nrelax = nrelax > 0 ? nrelax : 4;
```

Residual before mg_cycle

Optional function **residual** is called to calculate the residual before **mg_cycle**.

```
double resb;
resb = s.resb = s.resa = (* residual) (a, b, res, data);
```

Entering iteration

mg_cycle is called iteratively until the residual meets tolerance or iteration exceeds $NITERMAX$ (default by 100)

```
for (s.i = 0;
     s.i < NITERMAX && (s.i < NITERMIN || s.resa > tolerance);
     s.i++) {
  mg_cycle (a, res, da, relax, data,
            s.nrelax,
            minlevel,
            grid->maxdepth);
  s.resa = (* residual) (a, b, res, data);
```

Alteration of iteration inside mg_cycle
Based on the speed of convergence, times of **relax** applied on each level will inside **mg_cycle** be altered by changing *s.nrelax* (*nrelax* in **mg_cycle**).
End of iteration

```
#if 1
    if (s.resa > tolerance) {
      if (resb/s.resa < 1.2 && s.nrelax < 100)
        s.nrelax++;
      else if (resb/s.resa > 10 && s.nrelax > 2)
        s.nrelax--;
    }
#else
    if (s.resa == resb)
      break;
    if (s.resa > resb/1.1 && p.minlevel < grid->maxdepth)
      p.minlevel++;
#endif

    resb = s.resa;
  }
  s.minlevel = minlevel;
```

Warning and others
Once fail to converge, warning will be output through standard output. Extra allocated memory (*da* and *res* if not by default) then is freed. Since *res* point to a specific address no matter the input, *pres* serves as condition.

```
  if (s.resa > tolerance) {
    scalar v = a[0];
    fprintf (ferr,
             "WARNING: convergence for %s not reached after %d
             ↪  iterations\n"
             "  res: %g sum: %g nrelax: %d tolerance: %g\n", v.name,
             s.i, s.resa, s.sum, s.nrelax, tolerance), fflush (ferr);
  }
  if (!pres)
```

```
    delete (res), free (res);
  delete (da), free (da);

  return s;
}
```

4.2.3 Poisson-Helmholtz solver

Now we shall introduce the application for Poisson-Helmholtz equation of multigrid solver.

4.2.4 Poisson structure

Parameters

Name	Data type	Status	Option/Default	Representation (before/after)
a	scalar	unchanged	compulsory	$\mathbf{a}^0$
b	scalar	unchanged	compulsory	$\mathbf{b}$
alpha	face vector	unchanged	optional/1	α
lambda	scalar	unchanged	optional/0	λ
tolerance	double	unchanged	optional/10^{-4}	ϵ
nrelax	int	unchanged	optional/4	n
minlevel	int	unchanged	optional/1	n
res	scalar*	unchanged	optional/NULL	RES

Worth Mentioning Details

Such structure is built for conveying α and λ to solve equation 4.7. *res* serves as convergence monitor. anything concerned with embed is under construction.

Program Workflow

```
struct Poisson {
  scalar a, b;
  (const) face vector alpha;
  (const) scalar lambda;
  double tolerance;
  int nrelax, minlevel;
```

```
7   scalar * res;
8 #if EMBED
9   double (* embed_flux) (Point, scalar, vector, double *);
10 #endif
11 };
```

4.2.5 relax

Parameters

Name	Data type	Status	Option/Default	Representation (before/after)
al	scalar*	unchanged	compulsory	$\mathbf{a}^0$
bl	scalar*	unchanged	compulsory	$\mathbf{b}$
l	int	unchanged	compulsory	current level
data	void*	unchanged	compulsory	Poisson data structure

Worth Mentioning Details

Consider the discrete form of equation 4.7 using 5-points Laplacian operator in 2D

$$\frac{\alpha_{i+\frac{1}{2},j}\frac{a_{i+1,j}-a_{i,j}}{\Delta}-\alpha_{i-\frac{1}{2},j}\frac{a_{i,j}-a_{i-1,j}}{\Delta}}{\Delta}+\frac{\alpha_{i,j+\frac{1}{2}}\frac{a_{i,j+1}-a_{i,j}}{\Delta}-\alpha_{i,j-\frac{1}{2}}\frac{a_{i,j}-a_{i,j-1}}{\Delta}}{\Delta}+\lambda a_{i,j}=b \quad (4.8)$$

and arrange it into forms of equation 4.6 hence the expression of relaxation

$$a_{i,j}=\frac{\alpha_{i+\frac{1}{2},j}a_{i+1,j}+\alpha_{i-\frac{1}{2},j}a_{i-1,j}+\alpha_{i+\frac{1}{2},j+\frac{1}{2}}a_{i,j+1}+\alpha_{i,j-\frac{1}{2}}a_{i,j-1}-b\Delta^2}{\alpha_{i+\frac{1}{2},j}+\alpha_{i-\frac{1}{2},j}+\alpha_{i,j+\frac{1}{2}}+\alpha_{i,j-\frac{1}{2}}-\lambda\Delta^2} \quad (4.9)$$

By turning on/off macro 'JACOBI', user can choose type of iteration which employed in the solver. Once turning on, solver will implement Jacobi iteration with relaxation factor of $\frac{2}{3}$ i.e. $\mathbf{a}^{n+1}=(\mathbf{a}^n+2\mathbf{a}')$. Where $\mathbf{a}'$ is L.H.S. of equation 4.9 and will be stored in independent address named *c*. If such macro is turned off, *c* will be defined as additional pointer pointed to *a* and Gauss-Seidel method will be employed. Which indicates equation 4.6 is altered as

$$\delta x_i^{k+1}=F(\delta x_1^{k+1},\delta x_2^{k+1},\cdots,\delta x_{i-1}^{k+1},\delta x_{i+1}^{k},\cdots,\delta x_n^{k},RES^k) \quad (4.10)$$

and there is no use of relaxation factor.

Program Workflow

Initial settings and Input

a=$\mathbf{a}^0$, b=$\mathbf{b}$,$alpha$=α,$lambda$=λ

Note α and λ is conveyed through data structure 'Poisson' p

```
static void relax (scalar * al, scalar * bl, int l, void * data)
{
  scalar a = al[0], b = bl[0];
  struct Poisson * p = (struct Poisson *) data;
  (const) face vector alpha = p->alpha;
  (const) scalar lambda = p->lambda;
```

↓

Macro switch of Jacobi

If macro switch 'JACOBI' is active new independent scalar c is defined to store result of equation 4.9. Otherwise c will directly point to same address as a.

```
#if JACOBI
  scalar c[];
#else
  scalar c = a;
#endif
```

↓

Relaxation Function

Implementation of equation 4.9.

$$n = \alpha_{i+\frac{1}{2},j}a_{i+1,j} + \alpha_{i-\frac{1}{2},j}a_{i-1,j} + \alpha_{i+\frac{1}{2},j+\frac{1}{2}}a_{i,j+1} + \alpha_{i,j-\frac{1}{2}}a_{i,j-1} - b\Delta^2$$
$$d = \alpha_{i+\frac{1}{2},j} + \alpha_{i-\frac{1}{2},j} + \alpha_{i,j+\frac{1}{2}} + \alpha_{i,j-\frac{1}{2}} - \lambda\Delta^2$$

```
  foreach_level_or_leaf (l) {
    double n = - sq(Delta)*b[], d = - lambda[]*sq(Delta);
    foreach_dimension() {
      n += alpha.x[1]*a[1] + alpha.x[]*a[-1];
      d += alpha.x[1] + alpha.x[];
    }
```

↓

Embed Flux
TBD

```
#if EMBED
    if (p->embed_flux) {
      double c, e = p->embed_flux (point, a, alpha, &c);
      n -= c*sq(Delta);
      d += e*sq(Delta);
    }
    if (!d)
      c[] = 0., b[] = 0.;
    else
#endif // EMBED
```

Jacobi Implementation
L.H.S. of equation 4.9 is computed and stored in c. Once macro JACOBI is active, relaxation factor of $\frac{2}{3}$ is employed.

```
      c[] = n/d;
  }
#if JACOBI
  foreach_level_or_leaf (l)
    a[] = (a[] + 2.*c[])/3.;
#endif

#if TRASH
  scalar a1[];
  foreach_level_or_leaf (l)
    a1[] = a[];
  trash ({a});
  foreach_level_or_leaf (l)
    a[] = a1[];
#endif
}
```

4.2.6 residual

Parameters

Name	Data type	Status	Option/Default	Representation (before/after)
al	scalar*	unchanged	compulsory	$\mathbf{a}^k$
bl	scalar*	unchanged	compulsory	$\mathbf{b}$
resl	scalar*	unchanged	compulsory	RES^k
l	int	unchanged	compulsory	current level
data	void*	unchanged	compulsory	Poisson data structure

Worth Mentioning Details

The function aim to solve

$$RES^k = b_{i,j} - \frac{\alpha_{i+\frac{1}{2},j}\frac{a^k_{i+1,j}-a^k_{i,j}}{\Delta} - \alpha_{i-\frac{1}{2},j}\frac{a^k_{i,j}-a^k_{i-1,j}}{\Delta}}{\Delta} - \frac{\alpha_{i,j+\frac{1}{2}}\frac{a^k_{i,j+1}-a^k_{i,j}}{\Delta} - \alpha_{i,j-\frac{1}{2}}\frac{a^k_{i,j}-a^k_{i,j-1}}{\Delta}}{\Delta} - \lambda a^k_{i,j} \tag{4.11}$$

however careful consideration is needed when it comes to tree grid. Same problem has been fully discussed in section 2.3.2 of 'viscosity.h Documentation' and will not be repeated here.

The maximum residual is returned by **residual** and serves as criterion in **mg_solve** as described in section 4.2.

Program Workflow

Initial settings and Input

a=$\mathbf{a}^k$, *b*=$\mathbf{b}$,*alpha*=α,*lambda*=λ

Same as **relax** α and λ is conveyed through data structure 'Poisson' *p*. *res* is currently empty and *maxres* is the returned value that contains maximum residual.

```
1 static double residual (scalar * al, scalar * bl, scalar * resl, void *
  ↪  data)
2 {
3   scalar a = al[0], b = bl[0], res = resl[0];
4   struct Poisson * p = (struct Poisson *) data;
5   (const) face vector alpha = p->alpha;
6   (const) scalar lambda = p->lambda;
7   double maxres = 0.;
```

Conservative discretisation
Implementation of equation 4.11. Please check 'viscosity.h Documentation' for more details.

```
#if TREE
  face vector g[];
  foreach_face()
    g.x[] = alpha.x[]*face_gradient_x (a, 0);
  foreach (reduction(max:maxres), nowarning) {
    res[] = b[] - lambda[]*a[];
    foreach_dimension()
      res[] -= (g.x[1] - g.x[])/Delta;
```

Embed Flux
TBD

```
#if EMBED
    if (p->embed_flux) {
      double c, e = p->embed_flux (point, a, alpha, &c);
      res[] += c - e*a[];
    }
#endif // EMBED
```

Maximum Residual
$maxres$=$(RES)_{max}$ and will be returned eventually.

```
    if (fabs (res[]) > maxres)
      maxres = fabs (res[]);
  }
```

Implementation for Nontree Grid
Implementation for nontree grid, same scheme has only 1st order on tree grid.

```
#else
  foreach (reduction(max:maxres), nowarning) {
    res[] = b[] - lambda[]*a[];
    foreach_dimension()
      res[] += (alpha.x[0]*face_gradient_x (a, 0) -
                alpha.x[1]*face_gradient_x (a, 1))/Delta;
#if EMBED
    if (p->embed_flux) {
      double c, e = p->embed_flux (point, a, alpha, &c);
      res[] += c - e*a[];
    }
#endif
    if (fabs (res[]) > maxres)
      maxres = fabs (res[]);
  }
```

Final Return
Return maximum of residual.

```
#endif
  return maxres;
}
```

Bibliography

[1] P. Moin. *Fundamentals of engineering numerical analysis*. 2010.

[2] S. Popinet. "A quadtree-adaptive multigrid solver for the Serre–Green–Naghdi equations". In: *J. Comput. Phys.* 302 (2015), pp. 336–358.

[3] J. A. Van Hooft et al. "Towards adaptive grids for atmospheric boundary-layer simulations". In: *Bound.-Layer Meteorol.* 167 (2018), pp. 421–443.

[4] P. Wesseling. *Introduction to multigrid methods*. Tech. rep. 1995.

Chapter 5

viscosity.h Documentation

Version: 1.0 Updated: 2025-06-07

5.1 Introduction and Background

`viscosity.h` is constructed exclusively for NS solver 'centered.h' (see corresponding doc for more information.), whose purpose is to solve viscos equation implicitly using poisson equation solver built in 'poisson.h'. The governing equation is

$$\rho_{n+\frac{1}{2}}[\frac{\mathbf{u}^* - \mathbf{u}'}{\Delta t}] = \nabla \cdot [2\mu_{n+\frac{1}{2}}\mathbf{D}^*] \tag{5.1}$$

where $\mathbf{D}^* = [\nabla\mathbf{u}^* + (\nabla\mathbf{u}^*)^T]/2$, $\mathbf{u}'$ is known variable and $\mathbf{u}^*$ is the desired output.

Consider integral form of equation 5.1

$$\int_\Omega \rho_{n+\frac{1}{2}}[\frac{\mathbf{u}^* - \mathbf{u}'}{\Delta t}]dV = \int_\Omega \nabla \cdot [2\mu_{n+\frac{1}{2}}\mathbf{D}^*]dV = \int_{\partial\Omega} [2\mu_{n+\frac{1}{2}}\mathbf{D}^*] \cdot \mathbf{n}dS \tag{5.2}$$

The discrete form of this equation reads (component form is presented instead of tensor for sake of convenience).

$$\Delta\rho_{n+\frac{1}{2}}[\frac{u_i^* - u_i'}{\Delta t}] = \sum_{f=1}^{6}(n_f\mu_{n+\frac{1}{2}}(\frac{\partial u_j^*}{\partial x_i} + \frac{\partial u_i^*}{\partial x_j}))_f \quad i = x, y, z \quad j = normal(f) \tag{5.3}$$

where f in summation and subscript represents surfaces of single cell, j represents coordinate component of face normal disregarding negative or positive direction, n_f indicates whether the face normal is consistent with positive direction of coordinate, taking 2D cell as an example

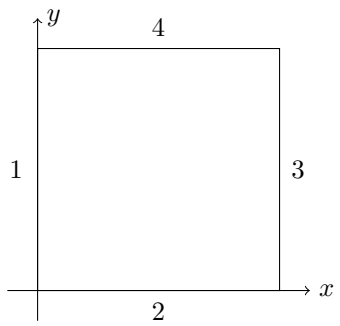


Figure 5.1: 2D cell example.

Unfold *R.H.S.* of equation 5.3 based on 2D cell depicted by figure 5.1 yields

$$R.H.S = [\mu_3(\frac{\partial u_x^*}{\partial x_i}+\frac{\partial u_i^*}{\partial x})_3-\mu_1(\frac{\partial u_x^*}{\partial x_i}+\frac{\partial u_i^*}{\partial x})_1+\mu_4(\frac{\partial u_y^*}{\partial x_i}+\frac{\partial u_i^*}{\partial y})_4-\mu_2(\frac{\partial u_y^*}{\partial x_i}+\frac{\partial u_i^*}{\partial y})_2] \quad i=x,y \tag{5.4}$$

Constructing derivative is simple for aligned direction but cumbersome for spliting direction.

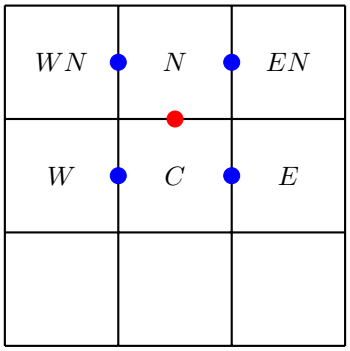


Figure 5.2: Sketch for derivative calculating.

Still take $2D$ cell as an example, as shown in figure 5.2, aligned direction derivative $e.g.(\frac{\partial u_y^*}{\partial y})_n$ (lower case indicates corresponding cell face) can be obtained by simply cnetral difference scheme

$$(\frac{\partial u_y^*}{\partial x_y})_n = \frac{u_N^* - u_C^*}{\Delta} \tag{5.5}$$

However, for splitting direction derivative $e.g.(\frac{\partial u_y^*}{\partial x})_n$ highlighted by red point above, there is no direct method to calculate the desired but average the ambient derivatives (highlighted by blue point), which are obtained following the direct method.

$$(\frac{\partial u_y^*}{\partial x})_n = \frac{(\frac{\partial u_y^*}{\partial x})_{wn} + (\frac{\partial u_y^*}{\partial x})_{en} + (\frac{\partial u_y^*}{\partial x})_w + (\frac{\partial u_y^*}{\partial x})_e}{4} \tag{5.6}$$

Now consider constructing x component of equation 5.3 for cell locates $(0,0)$. We herein denote x, y components of $\mathbf{u}$ as u, v but not u_x, u_y for sake of clarity. The $L.H.S$ can be directly expressed as

$$L.H.S = \Delta\rho_{n+\frac{1}{2}}[\frac{u_{0,0}^* - u_{0,0}'}{\Delta t}] \tag{5.7}$$

Then the first two terms of equation 5.4 which can be obtained directly are

$$R.H.SF2 = 2\mu_{(\frac{1}{2},0)}(\frac{u^*_{1,0} - u^*_{0,0}}{\Delta}) - 2\mu_{(-\frac{1}{2},0)}(\frac{u^*_{0,0} - u^*_{-1,0}}{\Delta}) \tag{5.8}$$

Finally the last two terms that is calculated by averaging yields

$$\begin{aligned} R.H.SL2 = \mu_{(0,\frac{1}{2})}[&\frac{u^*_{(0,1)} - u^*_{(0,0)}}{\Delta} + \frac{v^*_{(1,1)} - v^*_{(-1,1)} + v^*_{(1,0)} - v^*_{(-1,0)}}{4\Delta}] \\ -\mu_{(0,-\frac{1}{2})}[&\frac{u^*_{(0,0)} - u^*_{(0,-1)}}{\Delta} + \frac{v^*_{(1,0)} - v^*_{(-1,0)} + v^*_{(1,-1)} - v^*_{(-1,-1)}}{4\Delta}] \end{aligned} \tag{5.9}$$

Rearrange the equations we have

$$u^*_{(0,0)} = \frac{\frac{\Delta t}{\rho}(2\mu_{(\frac{1}{2},0)}u^*_{(1,0)} + 2\mu_{(-\frac{1}{2},0)}u^*_{(-1,0)} + \mathscr{A} - \mathscr{B}) + \Delta^2 u'_{(0,0)}}{\Delta^2 + \frac{\Delta t}{\rho}(2\mu_{(\frac{1}{2},0)} + 2\mu_{(-\frac{1}{2},0)} + \mu_{(0,\frac{1}{2})} + \mu_{(0,-\frac{1}{2})})} \tag{5.10}$$

where

$$\mathscr{A} = \mu_{(0,\frac{1}{2})}(u^*_{(0,1)} + \frac{v^*_{(1,1)} + v^*_{(1,0)} - v^*_{(-1,1)} - v^*_{(-1,0)}}{4}) \tag{5.11}$$

$$\mathscr{B} = \mu_{(0,-\frac{1}{2})}(-u^*_{(0,-1)} + \frac{v^*_{(1,-1)} + v^*_{(1,0)} - v^*_{(-1,-1)} - v^*_{(1,0)}}{4}) \tag{5.12}$$

Now, we obtain the implicit discrete linear expressions for desired valuable $\mathbf{u}^*$. Moreover, if we modify the governing equation into a more general form

$$\mathbf{u}' = \mathbf{u}^* - \frac{\Delta t \nabla \cdot [2\mu_{n+\frac{1}{2}}\mathbf{D}^*]}{\rho_{n+\frac{1}{2}}} = \mathscr{D}(\mathbf{u}^*) \tag{5.13}$$

here $\mathscr{D}$ is a linear operator that satisfies

$$RES^k = \mathbf{u}' - \mathscr{D}(\mathbf{u}^{*,k}) \tag{5.14}$$

$$\mathscr{D}(\delta\mathbf{u}^{*,k}) = RES^k \tag{5.15}$$

$$\mathbf{u}^{*,k+1} = \mathbf{u}^{*,k} + \delta\mathbf{u}^{*,k} \tag{5.16}$$

We denote by k the kth iterate approximating result of corresponding value. Then the multicycle solver constructed in 'poisson.h' can be applied to solve viscosity equation with specific relaxation and residual functions, which are built in this headfile.

5.2 Functions

5.2.1 Define

Parameters

Name	Data type	Status	Option	Representation (before/after)
mu	face vector	unchanged	compulsory	$\mu^{n+\frac{1}{2}}$
rho	scalar	unchanged	compulsory	$\rho^{n+\frac{1}{2}}$
dt	double	unchanged	compulsory	Δt

Program Workflow

Pre-Define

Define data structure Viscosity called in the total solver

mu=$\mu^{n+\frac{1}{2}}$ *rho*=$\rho^{n+\frac{1}{2}}$

dt=Δt

macro

macro defined here to set factor *lambda* especially for 'axi.h'. Otherwise, for cases in Cartesian mesh, the factor is set to be unity.

```
#include "poisson.h"

struct Viscosity {
  face vector mu;
  scalar rho;
  double dt;
};

#if AXI
# define lambda ((coord){1., 1. + dt/rho[]*(mu.x[] + mu.x[1] + \
                 mu.y[] + mu.y[0,1])/2./sq(y)})
#else // not AXI
# if dimension == 1
#   define lambda ((coord){1.})
# elif dimension == 2
#   define lambda ((coord){1.,1.})
# elif dimension == 3
#   define lambda ((coord){1.,1.,1.})
#endif
#endif
```

5.2.2 relax_viscosity

Parameters

Name	Data type	Status	Option	Representation (before/after)
a	scalar*	update	compulsory	$\delta\mathbf{u}^{*,k}/\delta\mathbf{u}^{*,k+1}$
b	scalar*	unchanged	compulsory	RES
dt	double	unchanged	compulsory	Δt
l	int	unchanged	compulsory	mesh level
data	struct Vsicosity	unchanged	compulsory	$\mu^{n+\frac{1}{2}}, \rho^{n+\frac{1}{2}}, \Delta t$

Worth Mentioning Details

The purpose of this function is to implement equation 5.10 and push forward iteration of G-S (or Jacobi) method for a single round on level l mesh. Note the object processed is $\delta\mathbf{u}^*$ defined previously instead of $\mathbf{u}^*$. Therefore, the equation it solves is equation 5.15 with given residual RES computed in another function.

The Basilisk self-defined macro `foreach_level_or_leaf(l)` takes l as its parameter and traverses each cell on level l or leaf cell (the cell that cannot be divided) whose level is less than l. Interested readers are referred to 'poisson.h Documentation' or Van Hooft *et.al.*[1] for more information about mesh heiarchy in Basilisk.

Program Workflow

Starting Point
input:
$a = \delta\mathbf{u}^*$ $b = RES$
l=*meshlevel*
$data = \mu^{n+\frac{1}{2}}, \rho^{n+\frac{1}{2}}, \Delta t$
data assignment
$\mu, \rho, \Delta t$ is extracted from *data*
u and r serve as pointer and point to $\delta\mathbf{u}^*, RES$ respectively
iteration type
The algorithm of iteration is selected by macro, if JACOBI is false, then a pointer w is created and points to u (a). Then all modification hereinafter to w is direct exert on u.

```
1 static void relax_viscosity (scalar * a, scalar * b, int l, void * data)
2 {
3   struct Viscosity * p = (struct Viscosity *) data;
4   (const) face vector mu = p->mu;
5   (const) scalar rho = p->rho;
6   double dt = p->dt;
```

```
7   vector u = vector(a[0]), r = vector(b[0]);
8
9   #if JACOBI
10    vector w[];
11  #else
12    vector w = u;
13  #endif
```

Relaxation Compute
implement of equation 5.10 for different dimension.
$w = \delta \mathbf{u}_{TBD}^{*,k+1}$

```
1   foreach_level_or_leaf (l) {
2     foreach_dimension()
3      w.x[] = (dt/rho[]*(2.*mu.x[1]*u.x[1] + 2.*mu.x[]*u.x[-1]
4   #if dimension > 1
5      + mu.y[0,1]*(u.x[0,1] +
6       (u.y[1,0] + u.y[1,1])/4. -
7       (u.y[-1,0] + u.y[-1,1])/4.)
8      - mu.y[]*(- u.x[0,-1] +
9          (u.y[1,-1] + u.y[1,0])/4. -
10         (u.y[-1,-1] + u.y[-1,0])/4.)
11  #endif
12  #if dimension > 2
13     + mu.z[0,0,1]*(u.x[0,0,1] +
14        (u.z[1,0,0] + u.z[1,0,1])/4. -
15        (u.z[-1,0,0] + u.z[-1,0,1])/4.)
16     - mu.z[]*(- u.x[0,0,-1] +
17         (u.z[1,0,-1] + u.z[1,0,0])/4. -
18         (u.z[-1,0,-1] + u.z[-1,0,0])/4.)
19  #endif
20     ) + r.x[]*sq(Delta))/
21  (sq(Delta)*lambda.x + dt/rho[]*(2.*mu.x[1] + 2.*mu.x[]
22  #if dimension > 1
23     + mu.y[0,1] + mu.y[]
24  #endif
25  #if dimension > 2
26     + mu.z[0,0,1] + mu.z[]
27  #endif
28                              ));
29  }
```

Update

If $JACOBI$ == $TRUE$

u=$\delta\mathbf{u}^{*,k+1}$ = $\frac{1}{3}\delta\mathbf{u}^{*,k}$ + $\frac{2}{3}\delta\mathbf{u}^{*,k+1}_{TBD}$

else (G − S iteration)

u=$\delta\mathbf{u}^{*,k+1}$ = $\delta\mathbf{u}^{*,k+1}_{TBD}$

a is modified along with u

```
#if JACOBI
  foreach_level_or_leaf (l)
    foreach_dimension()
      u.x[] = (u.x[] + 2.*w.x[])/3.;
#endif

#if TRASH
  vector u1[];
  foreach_level_or_leaf (l)
    foreach_dimension()
      u1.x[] = u.x[];
  trash ({u});
  foreach_level_or_leaf (l)
    foreach_dimension()
      u.x[] = u1.x[];
#endif
}
```

5.2.3 residual_viscosity

Parameters

Name	Data type	Status	Option	Representation (before/after)
a	scalar*	unchanged	compulsory	$\mathbf{u}^{*,k}$
b	scalar*	unchanged	compulsory	$\mathbf{u}'$
resl	scalar*	**output**	compulsory	$RES^k = \mathbf{u}' - \mathscr{D}(\mathbf{u}^{*,k})$
data	struct Vsicosity	unchanged	compulsory	$\mu^{n+\frac{1}{2}}, \rho^{n+\frac{1}{2}}, \Delta t$

Worth Mentioning Details

This function updates residual by solving equation 5.13 and returns the maximum component. Two computational methods are provided for tree and non-tree grid. The

first method for tree grid costs more computational resource compared with the second one due to extra traversal or cell face but preserves 2nd order on tree grid. While the second method is of 2nd order with caresian but 1st with tree grid.

Consider quadtree for 2D cases shown in figure 5.3 where coarser grid is of level l and finer grid is of level $l+1$. ■ represents example targert cell whose residual will be compute to display the difference between two methods. ● represents active point located at cell center where stores value for each cell. ● and ● are ghost cells which do not exist initially. ● is calculated by bilinear interpolation and is called as adjacent cell for finer cell while ● is obtained by averaging and is called from coarser cell. Moreover, cell faces highlighted by letters is another issue that worth mentioning. As can be seen from figure, target cell has five marked faces instead of four that face A is divided into subfaces A_1 and A_2 due to tree grid. However when called face value from centered cell perspective *i.e.* invoke value on face A from target cell, the value returned is the average of A_1 and A_2.

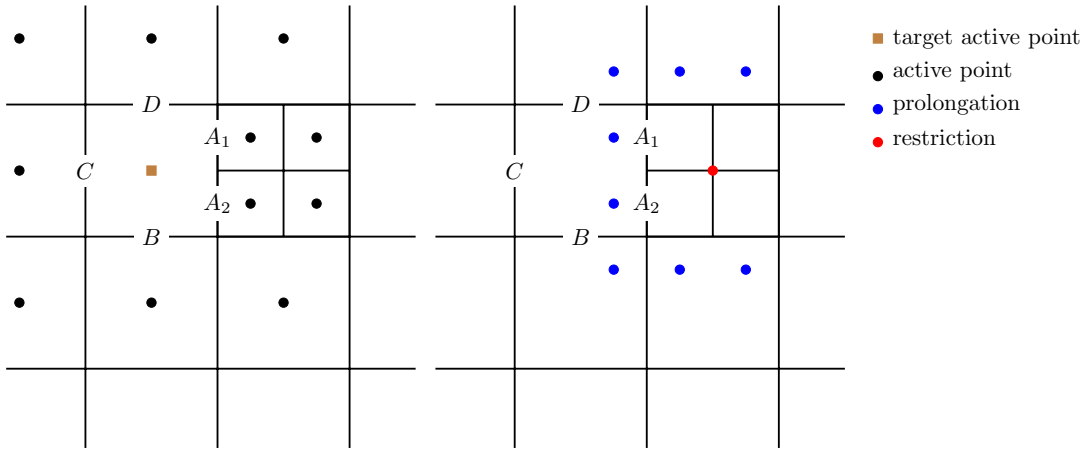


Figure 5.3: Sketch for 2D quadtree sample, active point and those ghost cells are depicted separately for clarity.

The first method which conserves 2nd order on tree grid caculates and stores value at each face. Therefore, ● as well as ● at level $l+1$ is used to compute desired value at face A_1 and A_2. Then value on target cell is obtained by linear combination of its face value. In conclusion, all marked points (● is used when compute face value on B, D) in figure 5.3 participate in determining target value.

However, second method employs direct way to caculate target value from cell centered stand. Therefore, target value is only decided by ● and ● on level l. Thus, applying second method to tree grid will lead to accuracy decrease. Otherwise, both methods act the same way on cartesian and the second method runs even faster.

Program Workflow

Starting Point
input:
$a = \mathbf{u}^{*,k}$ $b = \mathbf{u}'$
resl=RES^k
data = $\mu^{n+\frac{1}{2}}, \rho^{n+\frac{1}{2}}, \Delta t$
data assignment
$\mu, \rho, \Delta t$ is extracted from *data*
u, *r* and *res* serve as pointer and point to $\delta\mathbf{u}^{*,k}, \mathbf{u}', RES^k$ respectively.

```
static double residual_viscosity (scalar * a, scalar * b, scalar * resl,
          void * data)
{
  struct Viscosity * p = (struct Viscosity *) data;
  (const) face vector mu = p->mu;
  (const) scalar rho = p->rho;
  double dt = p->dt;
  vector u = vector(a[0]), r = vector(b[0]), res = vector(resl[0]);
  double maxres = 0.;
```

Tree Grid Update
ghost cell:
Set prolongation ghost cells for all components manually.
tree-flux compute
Compute flux *tau* on each face for three dimension based on equation 5.4.
tree-flux assmeble
Traverse all cell and assemeble $\mathbf{D}^*$ as *d* from cell centered perspective.
residual compute
Then the residual is updated by equation 5.13 and stored in *res*. The maximum of all components is returned through double data *maxres*.

```
#if TREE
  boundary ({u});

  foreach_dimension() {
    face vector taux[];
    foreach_face(x)
      taux.x[] = 2.*mu.x[]*(u.x[] - u.x[-1])/Delta;
    #if dimension > 1
      foreach_face(y)
        taux.y[] = mu.y[]*(u.x[] - u.x[0,-1] +
```

```
        (u.y[1,-1] + u.y[1,0])/4. -
        (u.y[-1,-1] + u.y[-1,0])/4.)/Delta;
    #endif
    #if dimension > 2
      foreach_face(z)
        taux.z[] = mu.z[]*(u.x[] - u.x[0,0,-1] +
        (u.z[1,0,-1] + u.z[1,0,0])/4. -
        (u.z[-1,0,-1] + u.z[-1,0,0])/4.)/Delta;
    #endif
    foreach (reduction(max:maxres)) {
      double d = 0.;
      foreach_dimension()
        d += taux.x[1] - taux.x[];
        res.x[] = r.x[] - lambda.x*u.x[] + dt/rho[]*d/Delta;
      if (fabs (res.x[]) > maxres)
        maxres = fabs (res.x[]);
    }
  }
```

NonTree Grid Update
residual compute

Then the residual is updated by equation 5.13 and stored in *res*. The maximum of all components is returned through double data *maxres*.

```
#else
  foreach (reduction(max:maxres))
    foreach_dimension() {
      res.x[] = r.x[] - lambda.x*u.x[] +
        dt/rho[]*(2.*mu.x[1,0]*(u.x[1] - u.x[])
        - 2.*mu.x[]*(u.x[] - u.x[-1])
  #if dimension > 1
        + mu.y[0,1]*(u.x[0,1] - u.x[] +
            (u.y[1,0] + u.y[1,1])/4. -
            (u.y[-1,0] + u.y[-1,1])/4.)
        - mu.y[]*(u.x[] - u.x[0,-1] +
            (u.y[1,-1] + u.y[1,0])/4. -
            (u.y[-1,-1] + u.y[-1,0])/4.)
        #endif
  #if dimension > 2
        + mu.z[0,0,1]*(u.x[0,0,1] - u.x[] +
          (u.z[1,0,0] + u.z[1,0,1])/4. -
```

```
18              (u.z[-1,0,0] + u.z[-1,0,1])/4.)
19           - mu.z[]*(u.x[] - u.x[0,0,-1] +
20               (u.z[1,0,-1] + u.z[1,0,0])/4. -
21               (u.z[-1,0,-1] + u.z[-1,0,0])/4.)
22           #endif
23                     )/sq(Delta);
24         if (fabs (res.x[]) > maxres)
25           maxres = fabs (res.x[]);
26       }
27   #endif
28     return maxres;
29   }
30
31   #undef lambda
```

5.2.4 viscosity

Parameters

Name	Data type	Status	Option/Default	Representation (before/after)
u	vector	update	compulsory	$\mathbf{u}'/\mathbf{u}^*$
mu	face vector	unchanged	compulsory	$\mu^{n+\frac{1}{2}}$
rho	scalar	unchanged	compulsory	$\rho^{n+\frac{1}{2}}$
dt	double	unchanged	compulsory	Δt
nrelax	int	unchanged	optional/4	$max\ of\ iteration$
res	scalar*	**output**	optional/NULL	RES

Worth Mentioning Details

The function to assemble all the tools built in this headfile or in 'poisson.h' and solve the governing equation equation 5.1. Details about implicit iteration solver construction used here are explored in 'poisson.h Documentation' which shall not be repeated again.

Program Workflow

Solver Construction
input:
$u = \mathbf{u}'$ $mu = \mu^{n+\frac{1}{2}}$ $rho = \rho^{n+\frac{1}{2}}$
$dt = \Delta t$ $res = empty$
$nrelax = iteration\ number$
data assignment
r is created to store $\mathbf{u}'$ and u now serves as $\mathbf{u}^*$ which will be updated.
Then mu and rho is restricted to all level and p
is defined as self made struct and to store $\mu, \rho, \Delta t$.
iteration solver
Solver is built using **mg_solve** from 'poisson.h'

```
trace
mgstats viscosity (vector u, face vector mu, scalar rho, double dt, int
 ↪  nrelax = 4, scalar * res = NULL)
{
  vector r[];
  foreach()
    foreach_dimension()
      r.x[] = u.x[];

  restriction ({mu,rho});
  struct Viscosity p = { mu, rho, dt };
  return mg_solve ((scalar *){u}, (scalar *){r}, residual_viscosity,
   ↪  relax_viscosity, &p, nrelax, res);
}
```

5.2.5 viscosity_explicit

Parameters

Name	Data type	Status	Option/Default	Representation (before/after)
u	vector	update	compulsory	$\mathbf{u}'/\mathbf{u}^*$
mu	face vector	unchanged	compulsory	$\mu^{n+\frac{1}{2}}$
rho	scalar	unchanged	compulsory	$\rho^{n+\frac{1}{2}}$
dt	double	unchanged	compulsory	Δt

Worth Mentioning Details

Different from implicit form shown in equation 5.1, the governing equation of this function is explicit

$$\rho_{n+\frac{1}{2}}[\frac{\mathbf{u}^* - \mathbf{u}'}{\Delta t}] = \nabla \cdot [2\mu_{n+\frac{1}{2}}\mathbf{D}'] \tag{5.17}$$

Then

$$\mathbf{u}^* = \mathbf{u}' + \frac{\Delta t \nabla \cdot [2\mu_{n+\frac{1}{2}}\mathbf{D}']}{\rho_{n+\frac{1}{2}}} \tag{5.18}$$

According to section 5.2.3, if a, b of **viscosity_residual** are set to be the same and equal to $\mathbf{u}'$ then the output yields

$$RES = \frac{\Delta t \nabla \cdot [2\mu_{n+\frac{1}{2}}\mathbf{D}']}{\rho_{n+\frac{1}{2}}} \tag{5.19}$$

Hence

$$\mathbf{u}^* = \mathbf{u}' + \frac{\Delta t \nabla \cdot [2\mu_{n+\frac{1}{2}}\mathbf{D}']}{\rho_{n+\frac{1}{2}}} = \mathbf{u}' + RES \tag{5.20}$$

Program Workflow

Solver Construction
input:
$u = \mathbf{u}'$ $mu = \mu^{n+\frac{1}{2}}$ $rho = \rho^{n+\frac{1}{2}}$
$dt = \Delta t$
explicit solver
$r = \frac{\Delta t \nabla \cdot [2\mu_{n+\frac{1}{2}}\mathbf{D}']}{\rho_{n+\frac{1}{2}}}$
u then updates to $\mathbf{u}^*$

```
trace
mgstats viscosity_explicit (vector u, face vector mu, scalar rho, double
 ↪  dt)
{
  vector r[];
  mgstats mg = {0};
  struct Viscosity p = { mu, rho, dt };
  mg.resb = residual_viscosity ((scalar *){u}, (scalar *){u}, (scalar
   ↪  *){r}, &p);
  foreach()
    foreach_dimension()
      u.x[] += r.x[];
```

```
11   return mg;
12 }
```

Bibliography

[1] J. A. Van Hooft et al. “Towards adaptive grids for atmospheric boundary-layer simulations”. In: *Bound.-Layer Meteorol.* 167 (2018), pp. 421–443.

Chapter 6

vof.h Documentation

Version: 1.0 Updated: 2025-06-07

6.1 Introduction

`vof.h` aims to solve advection equation

$$\frac{\partial c_i}{\partial t} + \mathbf{u} \cdot \nabla c_i = 0 \tag{6.1}$$

where c_i is the volume fraction. Additionally, Basilisk provides option to compute transport equation of diffusive tracer which is confined within specific phase e.g. ions in fluids whose governing equation reads[2]

$$\frac{\partial t_{i,j}}{\partial t} + \mathbf{u} \cdot \nabla t_{i,j} = 0 \tag{6.2}$$

where $t_{i,j} = f_j c_i$ or $tr = f_j(1 - c_i)$ depending on which side the tracer is confined. f_j is the concentration of the tracer.

The documentation is split into two parts: in the first part, the preparation to sovle the equation including gradient computation, prolongation in tree-grid and default event settings to implement prolongation are introduced. The solution of equation 6.1 and 6.2 are carefully discussed in the second part.

6.2 Preparation

6.2.1 vof_concentration_gradient

vof_concentration_gradient computes gradient for vof concentration using three-point scheme when given position is away from the surface and two-point scheme for those surface nearby cells.

Parameters

Name	Data type	Status	Option	Representation (before/after)
gradient	double	**output**	output	$\nabla t_{i,j}$
point	Point	unchanged	compulsory	position index
c	scalar	unchanged	compulsory	volume fraction c_j
s	scalar	unchanged	compulsory	$t_{i,j}$

Worth Mentioning Details

The gradient is calculated following a upwind-type two-point scheme when locates near the surface cell. In particular, such scheme is active if the volume fraction of only one adjacent cell is greater than 0.5. Otherwise a central three point scheme is used. Notably, the gradient is valid only if there are at least two out of adjacent cells, including current one, has fraction volume greater than 0.5.

Program Workflow

Starting Point
Input:
point: index information, *c* = c, *t* = t
Adjacent value assignment:
cl = $c[-1]$, *cc* = $c[]$, *cr* = $c[1]$
Inverse check:
To check in which phase the tracer exists.

```
foreach_dimension()
static double vof_concentration_gradient_x (Point point, scalar c,
 ↪  scalar t)
{
  static const double cmin = 0.5;
  double cl = c[-1], cc = c[], cr = c[1];
  if (t.inverse)
    cl = 1. - cl, cc = 1. - cc, cr = 1. - cr;
```

Gradient Compute
Local value check:
If *cc* < 0.5, return 0 otherwise checking *cr*.
Adjacent value check:
Check value of *cr* then the value of *cl*. If both value less than *cmin* then returning 0. If only one end is greater than *cmin*, compute baised gradient. Otherwise compute gradient using three-point scheme.

```
  if (cc >= cmin && t.gradient != zero) {
    if (cr >= cmin) {
      if (cl >= cmin) {
        if (t.gradient)
          return t.gradient (t[-1]/cl, t[]/cc, t[1]/cr)/Delta;
        else
          return (t[1]/cr - t[-1]/cl)/(2.*Delta);
      }
      else
        return (t[1]/cr - t[]/cc)/Delta;
    }
    else if (cl >= cmin)
      return (t[]/cc - t[-1]/cl)/Delta;
  }
  return 0.;
}
```

6.2.2 vof_concentration_refine

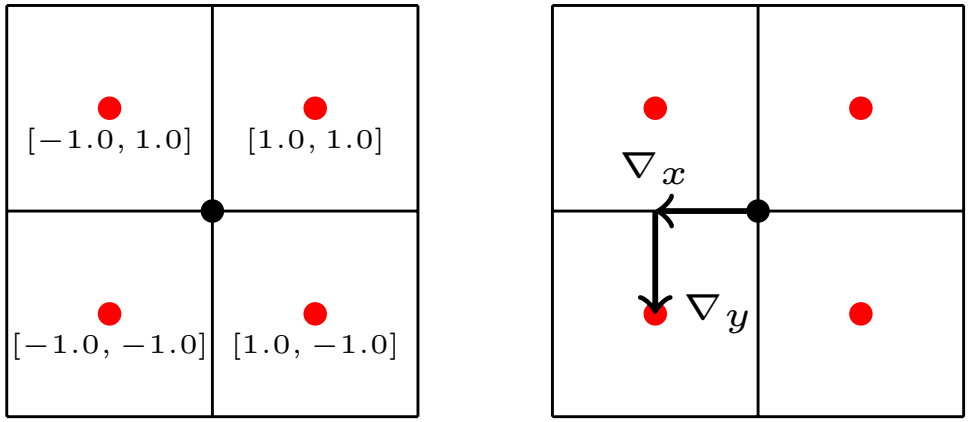
vof_concentration_refine defines the prolongation formula of VOF-concentration t when mesh is refined.

Parameters

Name	Data type	Status	Option	Representation (before/after)
point	Point	unchanged	compulsory	position index
s	scalar	unchanged	compulsory	$t_{i,j}$

Worth Mentioning Details

Basilisk employs *child* index to indicate spatial relation between parent and child cells, as displayed in figure 6.1a, the child cells with greater x (resp. y) coordinate are assigned with $child.x = 1$ (resp. $child.y = 1$) vice versa. When calling the macro $foreach_child$,



(a) (b)

Figure 6.1: (a) Sketch for *child* index. (b) Sketch for volume-fraction-weighted linear interpolation.

Basilisk will automatically transversal every child cells and *child* index is assigned with corresponding value. As indicated by figure 6.1b, given an active value, the prolongation is achieved by employing linear interpolation all the way to the center of child cell. Take 2D case as an example, the prolongation result t_{child} is obtained by

$$t_{child} = c_{child}(\frac{t_{parents}}{c_{parents}} + \frac{\Delta}{4}(child.x\nabla_x t + child.y\nabla_y t)) \tag{6.3}$$

where c is the fraction volume which is different for parent cell and child owing to reconstruction and $\Delta_x t, \Delta_y t$ are gradient computed by **vof_concentration_gradient** which has been detailed discussed in previous section.

Program Workflow

Starting Point
Input:
point: index information
$s = t, f = s.c = c$
Prolongation for void cells:
If the current cell is void i.e. does not contains tracers, the prolongation is directly assigned as 0.

```
#if TREE
static void vof_concentration_refine (Point point, scalar s)
{
  scalar f = s.c;
  if (cm[] == 0. || (!s.inverse && f[] <= 0.) || (s.inverse && f[] >=
  ↪ 1.))
```

```
6       foreach_child()
7         s[] = 0.;
```

Prolongation for Tracers

Tracer gradient assign:

$$g.d = \Delta \cdot \nabla_d t,\ d = x, y(, z)$$

Implement of equation 6.3:

First term of R.H.S.:

$$sc = s[] = \frac{t_{parents}}{c_{parents}}$$

Rest terms of R.H.S.:

$$s[] = s[] + \frac{\Delta}{4}(child.d \cdot g.d),\ d = x, y(, z)$$

Final assemble:

$$s[] = c_{child} \cdot s[]$$

```
1     else {
2       coord g;
3       foreach_dimension()
4         g.x = Delta*vof_concentration_gradient_x (point, f, s);
5       double sc = s.inverse ? s[]/(1. - f[]) : s[]/f[], cmc = 4.*cm[];
6       foreach_child() {
7         s[] = sc;
8         foreach_dimension()
9           s[] += child.x*g.x*cm[-child.x]/cmc;
10        s[] *= s.inverse ? 1. - f[] : f[];
11      }
12    }
13  }
```

6.3 Advection Solution

The exact solution is introduced in this section. Before diving into details, the conception of VOF method and major problem confronted in this method shall be discussed first.

6.3.1 VOF method

Generally speaking, there are two steps to accomplish VOF method i.e. advection of volume fraction and reconstruction of surface. The latter one is tackled in headfile

`geometry.h` and will not be covered in this document.

The integral form of equation 6.1 is

$$\Delta^d c_i|_n^{n+1} + \int_\Omega \mathbf{u}_f \cdot \nabla c_i = 0 \tag{6.4}$$

where Δ^d stands for the area (volume) of the cell. Using divergence theorem the equation turns into

$$\Delta^d c_i|_n^{n+1} + \int_{\partial\Omega} \mathbf{u}_f c_i - \int_\Omega c_i \nabla \cdot \mathbf{u}_f = 0 \tag{6.5}$$

the second term in equation 6.5 is the face flux $\mathbf{u}_f c_i = \mathbf{F}$. Consider the conservative constraint $\nabla \cdot \mathbf{u} = 0$, the third term is supposed to vanish. However, as will be discussed later, such term plays a critical role in the overall algorithm.

The advection is achieved by applying operator-split advection method[1] wherein the volume fraction is transport in each dimension. Take 2D case as an example, equation 6.5 now reads

$$c_i^* = c_i^n - \frac{\Delta t}{\Delta}(\mathbf{F}_x[1] - \mathbf{F}_x[]) + \frac{\Delta t}{\Delta}\int_\Omega c_i \nabla_x \cdot \mathbf{u}_f \tag{6.6}$$

$$c_i^{n+1} = c_i^* - \frac{\Delta t}{\Delta}(\mathbf{F}_y[1] - \mathbf{F}_y[]) + \frac{\Delta t}{\Delta}\int_\Omega c_i \nabla_y \cdot \mathbf{u}_f \tag{6.7}$$

Given interface and non-divergence face velocity $\mathbf{u}_f$ at t^n assuming the time step between t^n and t^{n+1} is Δt, the very next step is to obtain the face flux $\mathbf{F}$. The sketch of advection for specific cell is demonstrated by figure 6.2. The gray area represents volume of fraction and corresponding interface. The face flux $\mathbf{F}$ is directly obtained by the gray area within dashed rectangle whose width is $\mathbf{u}_f \Delta t$. To avoid non-conservation induced by transporting overlapped area (the blue area in figure 6.2b), reconstruction is applied between advection of every direction as indicated from shape changes between two figures.

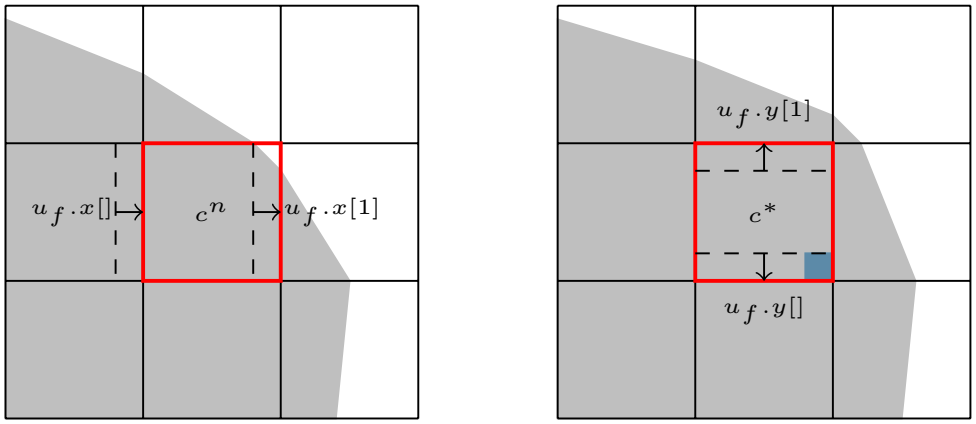
Yet there is still one constraint to satisfy: the volume fraction should be $0 \leq c_i \leq 1$ at any stage. However without additional care, such rule can be violated between direction switch. Take condition in figure 6.2a as an example. Assume $c^n = 0.9, u_f.x[] = 0.3, u_f.x[1] = 0.1$, for sake of simplicity the time step and cell width are set to be unity. Ignoring the third term in equation 6.6, $c^* = 1.1 > 1.0$ which of course break the constraint and leads to failure in surface reconstruction. An opposite condition may also occur where the volume fraction eventually becomes $c < 0$. The dilation term (third term in equation 6.5) is therefore introduced to cope with such issue[3].

The dilation term

Given the non-divergence of $\mathbf{u}_f$, the dilation term in equation 6.5 can be rewrite in discrete form as:

$$\int_\Omega c_i \nabla \cdot \mathbf{u}_f = g\Delta^{d-1} \sum_d (u_f.d[1] - u_f.d[]) \tag{6.8}$$

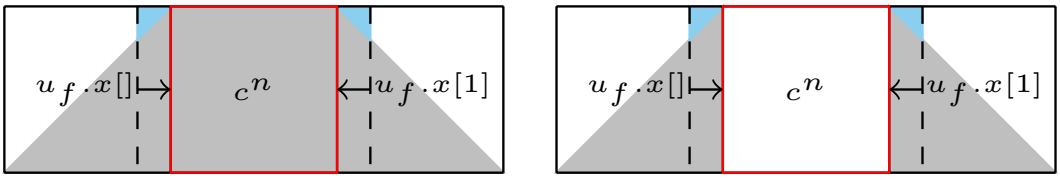
where g is an arbitrary value and can even vary in different cells or in different cycle of iteration. The main topic in this section is to find a proper form of g which can help to overcome the overfull issue depicted previously.



(a) (b)

Figure 6.2: Sketches for (a) advection on x direction (b) advection on y direction. The width of dashed rectangle is $\mathbf{u}_f \Delta t$

The condition where $g = 0$ i.e. no dilation has been fully discussed, consider now the condition where $g = 1$. Take condition in figure 6.3a as an example. The volume fraction for current cell is $c^n = 1$, given the input face velocity on both faces, the overfull occurs without dilation term. However after adding the dilation, according to equation 6.6 and 6.8 the input flux $\mathbf{F}$ (gray area confined by dashed rectangle) will be counteracted by dilation term (area of dashed rectangle when $g = 1$). The difference between (blue area) prevents the overfull issue.



(a) (b)

Figure 6.3: Sketch for dilation. The area highlighted by blue represents the area difference induced by adding dilation term

Nevertheless, the dilation can be harmful at certain circumstance. As shown by figure 6.3b where the volume fraction now becomes $c^n = 0$, with input face velocity the dilation now promotes clipping ($c < 0$) owing to area difference (blue area).

Coefficient g and CFL number

Basilisk follows work by Weymouth and Yue[4] and sets coefficient g as

$$g = c_c = \begin{cases} 1 & c^n \geq 0.5 \\ 0 & c^n < 0.5 \end{cases} \tag{6.9}$$

with CFL number constraint $\sum_d \frac{\Delta t}{\Delta} |u_f.d| < 0.5$ the method can preserve exact conservation.

Following the same work, given g in equation 6.9 the restriction for volume fraction reads

$$c \geq 0.5 \geq min(\frac{\Delta t}{\Delta}u_f[1], f) + \frac{\Delta t}{\Delta}(u_f[] - u_f[1]);\ 1 - c \geq 0.5 \geq \frac{\Delta t}{\Delta}u_f[] - max(0, \frac{\Delta t}{\Delta}u_f[1] - 1 + c) \tag{6.10}$$

$$c \geq 0.5 \geq \frac{\Delta t}{\Delta}(u_f[] - u_f[1]);\ 1 - c \geq 0.5 \geq \frac{\Delta t}{\Delta}(u_f[] - u_f[1]) \tag{6.11}$$

with CFL$<$ 0.5, the restriction can be easily satisfied therefore preserving the volume conservation. Note however such restriction is deduced based on the assumption that the volume fraction of current cell is the only accessible.

6.3.2 sweep_x

sweep_x aims to achieve advection equation 6.6 in each direction. Notably, similar method is also applied to the advection of tracer in which the dilation term is added.

Parameters

Name	Data type	Status	Option	Representation (before/after)
c	scalar	**updated**	compulsory	c^n/c^*
cc	scalar	unchanged	compulsory	c_c in equation 6.9
tcl	scalar	unchanged	compulsory	tracer coefficient t_c

Worth Mentioning Details

The dilation coefficient for tracer advection is defined as

$$t_c = \begin{cases} t_{ij}/c_i & c^n \geq 0.5 \\ t_{ij}/c_i & c^n < 0.5 \end{cases} \tag{6.12}$$

Note if inverse is true, the c_i changes as $(1-c_i)$. Different from volume fraction advection, the face flux for tracer is computed using 1D BCG scheme.

$$\mathbf{F}_t[] = u_f[] \cdot c_f[](\frac{t[]}{c^n[]} + \frac{\Delta}{2}[s_d - \frac{\Delta t}{\Delta}u_f[]]\frac{\partial t[]}{\partial x}) \tag{6.13}$$

Where the gradient is compute by interface biased scheme introduced in section 6.2.1 and s_d is the upwind coefficient (see 'bcg.h documentation' for more). c_f as demonstrated by

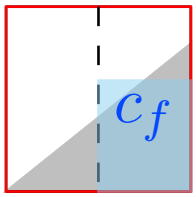


Figure 6.4: The sketch of partial fraction c_f.

figure 6.4 is in fact the average partial fraction of geometric flux which serves as a face coefficient on tracer flux. The advection for tracer now yields

$$t_d^* = t^n - \frac{\Delta t}{\Delta}(\mathbf{F}_{t,d}[1] - \mathbf{F}_{t,d}[]) + \frac{t_c \Delta t}{\Delta}(u_{f,d}[1] - u_{f,d}[]) \tag{6.14}$$

The function is replicated into **sweep_y** and **sweep_z** by macro *foreach_dimension*().

Program Workflow

Starting Point
Input:
c = c, *cc* = c_c, *tcl* = t_c
where *tcl* is computed in function **vof_advection**.
n and *alpha* are defined for the interface reconstruction and geometric advection.
Owing to 1D advection, the flux is stored in scalar *flux* instead of face vector data.

```
foreach_dimension()
static void sweep_x (scalar c, scalar cc, scalar * tcl)
{
  vector n[];
  scalar alpha[], flux[];
  double cfl = 0.;
```

Preparation for Tracer Advection
Variable defined:
For each tracer, two scalar types of data: *gf* and *tflux* are defined which represent tracer flux and tracer gradient respectively, and these are stored in corresponding lists.
Tracer gradient computation:
Compute gradient using function **vof_concentration_gradient_x** for every tracer.

```
  scalar * tracers = c.tracers, * gfl = NULL, * tfluxl = NULL;
  if (tracers) {
    for (scalar t in tracers) {
      scalar gf = new scalar, flux = new scalar;
      gfl = list_append (gfl, gf);
      tfluxl = list_append (tfluxl, flux);
    }
    foreach() {
```

```
      scalar t, gf;
      for (t,gf in tracers,gfl)
        gf[] = vof_concentration_gradient_x (point, c, t);
    }
  }
```

Computation of Geometric Advection Flux

Interface reconstruction:
Before computing the flux, the interface is reconstructed exclusively based on distribution of c.

Flux computation:
The face velocity *uf.x[]* is first calculated and stored in *un* as CFL number. After update the CFL number to its largest value, the partial fraction is computed by tool from `geometry.h` from 'upwinded' cell and stored in *cf*. The flux is therefore obtained as

$$flux = uf.x[] \times cf$$

```
  reconstruction (c, n, alpha);
  foreach_face(x, reduction (max:cfl)) {
    double un = uf.x[]*dt/(Delta*fm.x[] + SEPS), s = sign(un);
    int i = -(s + 1.)/2.;
#if EMBED
    if (cs[] >= 1.)
#endif
    if (un*fm.x[]*s/(cm[] + SEPS) > cfl)
      cfl = un*fm.x[]*s/(cm[] + SEPS);
    double cf = (c[i] <= 0. || c[i] >= 1.) ? c[i] :
      rectangle_fraction ((coord){-s*n.x[i], n.y[i], n.z[i]}, alpha[i],
      (coord){-0.5, -0.5, -0.5},
      (coord){s*un - 0.5, 0.5, 0.5});
    flux[] = cf*uf.x[];
```

Computation of Tracer Advection Flux
Preparation:
cf1 = c_f, *ci* = c[], check in which phase the tracer exists.
BCG flux computation:
Implementation of equation 6.13 for tracer-existing phase. Otherwise the tracer flux vanishes.
Clean up:
Clean up the tracer gradient and free the corresponding memory. Output warning if the CFL number exceeds the critical as discussed in section 6.3.1.

```
    scalar t, gf, tflux;
    for (t,gf,tflux in tracers,gfl,tfluxl) {
      double cf1 = cf, ci = c[i];
      if (t.inverse)
        cf1 = 1. - cf1, ci = 1. - ci;
      if (ci > 1e-10) {
        double ff = t[i]/ci + s*min(1., 1. - s*un)*gf[i]*Delta/2.;
        tflux[] = ff*cf1*uf.x[];
      }
      else
        tflux[] = 0.;
    }
  }
  delete (gfl); free (gfl);
  if (cfl > 0.5 + 1e-6)
    fprintf (ferr,
             "WARNING: CFL must be <= 0.5 for VOF (cfl - 0.5 = %g)\n",
             cfl - 0.5), fflush (ferr);
```

Volume Fraction Update (Non-embed)
Volume fraction update:
Implementation of equation 6.6 with dilation term.
Tracer update:
Implementation of equation 6.14. Note however, for advection of tracers the dilation term is optional and can be turned off by macro.

```
#if !EMBED
  foreach() {
    c[] += dt*(flux[] - flux[1] + cc[]*(uf.x[1] - uf.x[]))/(cm[]*Delta);
#if NO_1D_COMPRESSION
```

```
    scalar t, tflux;
    for (t, tflux in tracers, tfluxl)
      t[] += dt*(tflux[] - tflux[1])/(cm[]*Delta);
#else // !NO_1D_COMPRESSION
    scalar t, tc, tflux;
    for (t, tc, tflux in tracers, tcl, tfluxl)
      t[] += dt*(tflux[] - tflux[1] + tc[]*(uf.x[1] -
       ↪  uf.x[]))/(cm[]*Delta);
#endif // !NO_1D_COMPRESSION
  }
#else // EMBED
```

Volume Fraction Update (embed)
waiting for the latest version
Volume fraction update:
Tracer update:
Clean up:
Delete and free the memory of *tflux*.

```
  foreach()
    if (cs[] > 0.) {
      c[] += dt*(flux[] - flux[1] + cc[]*(uf.x[1] - uf.x[]))/Delta;
#if NO_1D_COMPRESSION
      for (t, tflux in tracers, tfluxl)
        t[] += dt*(tflux[] - tflux[1])/Delta;
#else // !NO_1D_COMPRESSION
      scalar t, tc, tflux;
      for (t, tc, tflux in tracers, tcl, tfluxl)
        t[] += dt*(tflux[] - tflux[1] + tc[]*(uf.x[1] - uf.x[]))/Delta;
#endif // !NO_1D_COMPRESSION
    }
#endif // EMBED

  delete (tfluxl); free (tfluxl);
}
```

6.3.3 vof_advection

The VOF advection along with tracer advection is assembled in function **vof_advection**.

Parameters

Name	Data type	Status	Option	Representation (before/after)
interfaces	scalar*	**updated**	compulsory	c_i^n/c_i^{n+1}
i	int	unchanged	compulsory	number of time step

Worth Mentioning Details

Direction switch is implemented by counting the time step i.

$$D = (i + d)\ mod\ 3 \quad d = 0, 1, 2 \cdots \tag{6.15}$$

Where $D = 0, 1, 2$ indicates x, y, z direction and shows up in specific sequence depending on i.

Program Workflow

Starting Point
Input:
interface = c_i, *i* = i
Since data type of *interface* is scalar* the advection interface can be multiple.
Preparation:
cc = c_c and *tcl* is the scalar list contains t_c for each tracer.

```
void vof_advection (scalar * interfaces, int i)
{
  for (scalar c in interfaces) {
    scalar cc[], * tcl = NULL, * tracers = c.tracers;
```

Tracer Settings
Definition of tracer dilation coefficient:
tc = t_c is the tracer dilation coefficient and is stored in list *tcl* for each kind of tracer.
TREE-grid arrangement for tracer:
The setting is the same as the one in function **vof_concentration_refine**
to make sure the refine and prolongation use the conservative method.

```
    for (scalar t in tracers) {
#if !NO_1D_COMPRESSION
```

```
      scalar tc = new scalar;
      tcl = list_append (tcl, tc);
#endif // !NO_1D_COMPRESSION
#if TREE
      if (t.refine != vof_concentration_refine) {
        t.refine = t.prolongation = vof_concentration_refine;
        t.restriction = restriction_volume_average;
        t.dirty = true;
        t.c = c;
      }
#endif // TREE
    }
```

↓

Computation of Dilation Coefficient
Implement of equation 6.9 and 6.12.

```
    foreach() {
      cc[] = (c[] > 0.5);
#if !NO_1D_COMPRESSION
      scalar t, tc;
      for (t, tc in tracers, tcl) {
        if (t.inverse)
          tc[] = c[] < 0.5 ? t[]/(1. - c[]) : 0.;
        else
          tc[] = c[] > 0.5 ? t[]/c[] : 0.;
      }
#endif // !NO_1D_COMPRESSION
    }
```

↓

Direction Switch and Default Calling
Direction switch:
As discussed in previous section, the direction switch is achieved by counting the steps i. The list stores tracer coefficient is cleaned.
Default calling:
The function **vof_advection** is called on every interface at each time step.

```
    void (* sweep[dimension]) (scalar, scalar, scalar *);
    int d = 0;
    foreach_dimension()
      sweep[d++] = sweep_x;
    for (d = 0; d < dimension; d++)
      sweep[(i + d) % dimension] (c, cc, tcl);
    delete (tcl), free (tcl);
  }
}

event vof (i++)
  vof_advection (interfaces, i);
```

6.4 Draft

If all the volume fraction is known, flux $\mathbf{F}$ on each face is

$$min(u_f[], c[-1]) \geq F[] \geq max(0, u_f[] - \bar{c}[-1]) \quad u_f[] \geq 0 \tag{6.16}$$

$$-max(0, -u_f[] - \bar{c}[]) \geq F[] \geq -min(-u_f[], c[]) \quad u_f[] < 0 \tag{6.17}$$

The flux difference $\Delta F_d[] = F_d[] - F_d[1]$ therefore is

$$min(u_f[], c[-1]) - max(0, u_f[1] - \bar{c}[]) \geq \Delta F[] \geq max(0, u_f[] - \bar{c}[-1]) - min(u_f[1], c[]) \quad u_f[] > 0 \tag{6.18}$$

$$min(u_f[], c[-1]) + min(-u_f[1], c[1]) \geq \Delta F[] \geq max(0, u_f[] - \bar{c}[-1]) + max(0, -u_f[1] - \bar{c}[1]) \quad u_f[] > 0 \tag{6.19}$$

$$-max(0, -u_f[] - \bar{c}[]) - max(0, u_f[1] - \bar{c}[]) \geq \Delta F[] \geq -min(-u_f[], c[]) - min(u_f[1], c[]) \quad u_f[] < 0 \tag{6.20}$$

$$-max(0, -u_f[] - \bar{c}[]) + min(-u_f[1], c[1]) \geq \Delta F[] \geq -min(-u_f[], c[]) + max(0, -u_f[1] - \bar{c}[1]) \quad u_f[] < 0 \tag{6.21}$$

Bibliography

[1] T. Gretar, S. Ruben, and S. Zaleski. *Direct numerical simulations of gas-liquid multiphase flows*. 2011.

[2] J. M. Lopez-Herrera et al. "Electrokinetic effects in the breakup of electrified jets: A volume-of-fluid numerical study". In: *Int. J. Multiph. Flow* 71 (2015), pp. 14–22.

[3] R. Scardovelli and S. Zaleski. "Interface reconstruction with least-square fit and split Eulerian–Lagrangian advection". In: *Int. J. Numer. Methods Fluids* 41.3 (2003), pp. 251–274.

[4] G. D. Weymouth and D. K-P. Yue. "Conservative volume-of-fluid method for free-surface simulations on Cartesian grids". In: *J. Comput. Phys.* 229.8 (2010), pp. 2853–2865.

Chapter 7

heights.h Documentation

Version: 2.0 Updated: 2025-06-08

7.1 Introduction

`heights.h`, together with `parabola.h`, serves as a toolbox for `curvature.h` to compute surface curvature in multiphase flow. The aim of the file is to allocate a value representing the distance to the surface for every cell. This value is named 'height', hence the name of the header file. Before diving into details, several conceptual definitions are introduced with a 1D example.

7.1.1 Surface

For each cell, if a 'coherent surface' occurs within a certain distance, the cell is assigned a valid height value. Otherwise, an invalid data value 'nodata' is allocated. A 'coherent surface' is defined as a process where the color function (volume fraction in the VOF method) c changes from 0 to 1 or vice versa.

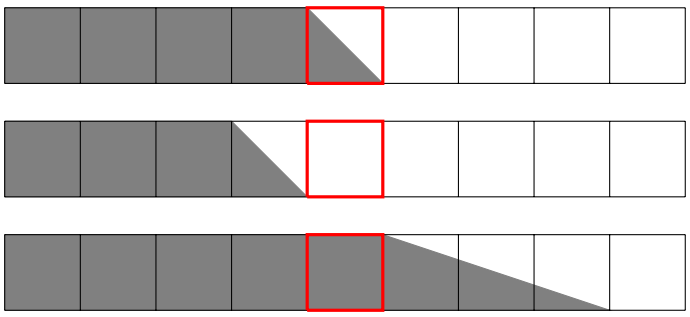


Figure 7.1: Graphical representation of a coherent surface. The cell highlighted by a red square represents the current cell. The grey color indicates volume fraction.

As shown in figure 7.1, height values are valid for the current cell (highlighted by a red square) when, within a certain distance, cells fully immersed in the color function (represented by grey) and those with a 0 value both occur. In contrast, figure 7.2 demonstrates cases where a 'coherent surface' is not observed, so 'nodata' is assigned.

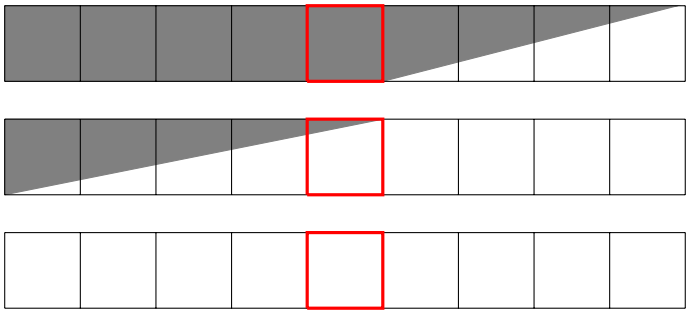


Figure 7.2: Graphical representation of a non-coherent surface. The cell highlighted by a red square represents the current cell. The grey color indicates volume fraction.

7.1.2 Height Value and Zero-Value Point

The height value is essentially a summary of the color function, with each cell's value represented at its center. To obtain a specific value, the position where the value is 0 (the zero-value point) must first be defined. Consider the condition illustrated in figure 7.3, where the surface spans two cells with color function values of 0.6 and 0.1. The zero-value point is located 0.7 units away from the last occupied cell, as indicated by the blue dashed line in figure 7.3. The value of each cell is then calculated based on the distance between its center and the zero-value point, as shown below the figure. The positive direction points toward the inner side of the surface.

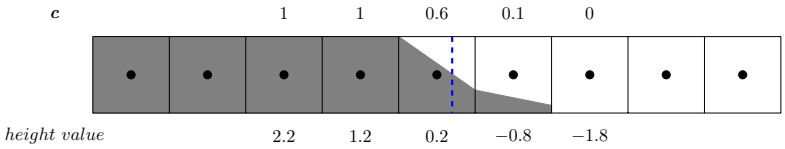


Figure 7.3: Graphical representation of the zero-value point. The blue dashed line indicates the exact position of the zero-value point. Numbers at the top show the color function of each cell, while those below display the corresponding height value at the center of each cell, marked by a black dot.

7.1.3 Direction of Surface

Besides the example in figure 7.3, another possibility exists where the surface extends in the opposite direction (from right to left in this example). To distinguish these cases, the zero-value point is assigned a value of 20 for cases shown in figure 7.4. Typically, only cells within a 5.5-unit distance from the zero-value point have valid height values. Thus, the range of height values is $[-5, 5]$ or $[15, 25]$, depending on the surface direction.

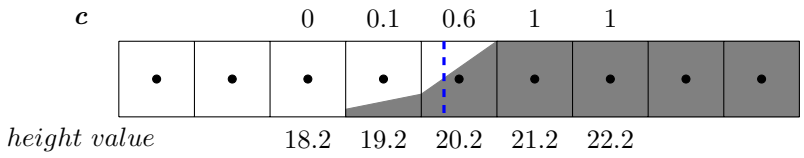


Figure 7.4: Graphical representation of the zero-value point. The blue dashed line indicates the exact position of the zero-value point. Numbers at the top show the color function of each cell, while those below display the corresponding height value at the center of each cell, marked by a black dot.

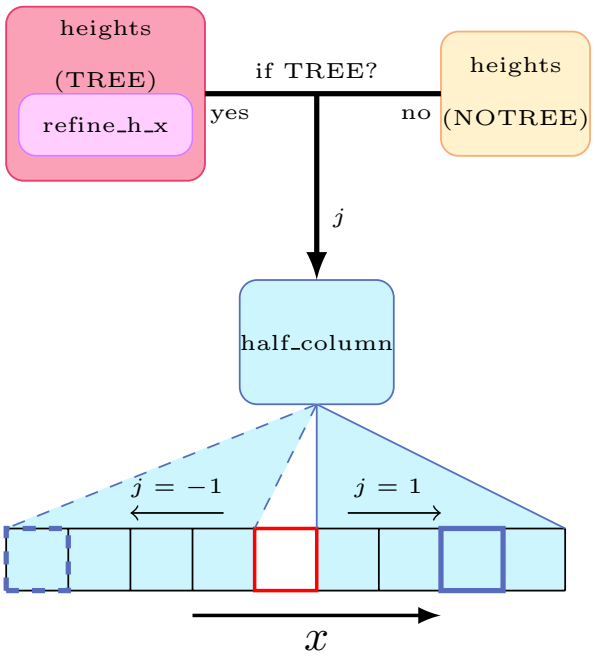


Figure 7.5: Overall configuration of the current header file. The red square highlights the current cell, and the blue square represents cells scanned by **half_column**.

7.2 Functions

7.2.1 Overall Configuration

Before introducing parameters and workflows, a brief overview of the configuration of the three main functions discussed in the following sections is provided. As shown in figure 7.5, **heights** acts as a controller, issuing commands to **half_column**, which integrates volume fractions in neighboring cells. After iterating over the eight adjacent cells (four in both positive and negative directions), the height value is assigned to the current cell. Special care is taken for tree grids, where the additional function **refine_h_x** is used to prolongate the color function.

7.2.2 height and orientation

Worth Mentioning Details

These two functions are 'tricky' functions. As discussed in section 7.1, to distinguish surface orientation, the final height value has two ranges: $[-5, 5]$ or $[15, 25]$. The **height** function is a reverse function that takes the height value as input, removes the orientation disguise, and returns the true distance between the current cell and the zero-value point, with an output range of $[-10, 10]$. In contrast, **orientation** returns the surface orientation based on the height value. To save memory, these tricky functions are implemented as 'inline' functions, meaning their code is substituted directly when called.

Program Workflow

Starting Point for height
Input: *H* (height value).
The disguise value *HSHIFT* is 20. By adding or removing this value, the true interfacial distance is revealed.

```
#define HSHIFT 20.

static inline double height (double H) {
  return H > HSHIFT/2. ? H - HSHIFT : H < -HSHIFT/2. ? H + HSHIFT : H;
}
```

Starting Point for orientation
Input: *H* (height value).
Returns `TRUE` or `FALSE` based on the disguise value.
Two layers of ghost values are set on every boundary.

```
static inline int orientation (double H) {
  return fabs(H) > HSHIFT/2.;
}

#define BGHOSTS 2
```

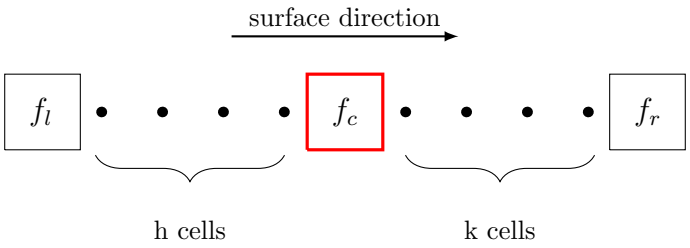


Figure 7.6: Sketch of position switching between cells. f_c, f_r, f_l represent the color function values of the current cell (red square) and its right/left neighbors, respectively.

7.2.3 half_column

Based on the previous discussion, **half_column** plays a major role in height computation by scanning neighboring cells and returning the surface status and height value. Due to its complexity, the algorithm is explained part by part.

Parameters

Name	Data type	Status	Option/Default	Representation (before/after)
point	Point	unchanged	compulsory	current cell position
c	scalar	unchanged	compulsory	c (volume fraction)
h	vector	**output**	compulsory	**h**
cs	scalar	unchanged	compulsory	$c[2 * j]$
j	int	unchanged	compulsory	j (direction control)

Purpose of the Function

half_column has two purposes:

1. To check whether a coherent surface (defined in section 7.1) exists within eight adjacent cells.

2. If a coherent surface exists, compute the corresponding height value by integrating the color function of each cell.

There are three possible situations for the current cell: $f_c = 1$, $f_c = 0$, or $f_c \in (0, 1)$, where f_c is the color function value. For the first two cases, the threshold for a coherent surface is finding a neighbor with $f = 0$ or $f = 1$, respectively. For the third case (a surface cell), both directions must be iterated to find a pair of opposite (full and empty) cells on each side.

Once a coherent surface is found, the remaining task is to calculate the height value. The key is understanding position switching between cells. As shown in figure 7.6, where f_c, f_r, f_l represent the color function values of the current cell and its neighbors, following the surface direction (defined as a vector pointing from the inner to the outer side of the surface, e.g., figure 7.3), the relationship between these values is:

$$f_c - (k + 1) = f_r \tag{7.1}$$

$$f_c + (h + 1) = f_l \tag{7.2}$$

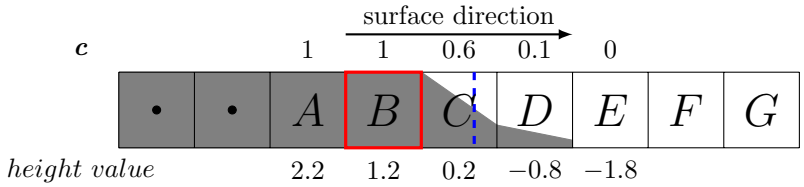


Figure 7.7: Same example as in figure 7.3 with two example stencils A, B. The red square highlights the final full cell next to the surface.

Among all cells, the easiest height value to compute is that of the final full cell close to the surface (hereinafter called the final cell). Figure 7.7 revisits the example from figure 7.3, highlighting the final cell and surface direction. Based on section 7.1, the height value of the final cell is computed by integrating the color function along the surface direction until the first empty cell (cell E):

$$h_B = h_B + h_C + h_D + h_E - 0.5 = 1 + 0.6 + 0.1 + 0 - 0.5 = 1.2 \tag{7.3}$$

The four terms on the right-hand side represent the color function values of cells B, C, D, and E. The -0.5 accounts for the fact that the height value is determined from the cell's center, not its left boundary. Using position switching, the height value of every cell can be obtained. For example, the height value of cell F is $h_F = h_B - 4 = -2.8$, and that of cell A is $h_A = h_B + 1 = 2.2$.

Configuration of the Function

Figure 7.9 details the configuration of **half_column**, which consists of three layers. As shown in figure 7.5 and section 7.2, **half_column** is called twice: first for the negative direction ($j = -1$), then for the positive direction ($j = 1$). Except for the second layer ('iteration'), the other two layers are adjusted based on the direction.

Parameter ***S*** records the process status: if a surface is found, the height computation is complete ($S = -1$); if one end is found, $S = 1$ or 0; if no end is found, $S \in (0, 1)$. Since ***S*** is defined within **half_column** and cleared when the function ends, a method is needed to preserve results from the negative direction. Basilisk achieves this by encoding the status and height result ***H*** into ***h[]*** and decoding it at the start of the second cycle ($j = 1$).

In the overall process, decoding (if needed) occurs with initial settings in the first layer. Height values are calculated and stored as ***H*** in the second layer by checking neighboring cells, with ***S*** updated accordingly. The output ***h[]*** is processed in the final layer, either encoded or directly output, depending on the direction. Detailed explanations follow.

First Layer

The initial status ***S*** is derived from the volume fraction of the current cell (highlighted by a red square) and can be 0, 1, or f (empty, full, or surface cell). In the second cycle ($j = 1$), an additional decoding step restores the status and height from the previous cycle into ***stats.s*** and ***stats.h***. Four decoding scenarios are possible:

Figure 7.8: The situation that the second layer occasion D aims to tackle.

1. $h[] = 300$, $(stat.s, stat.h) = (-1, \text{nodata})$: Non-surface cells ($c[] = 1$ or 0) that fail to find a coherent surface, or surface cells that fail to find an empty/full cell in the first cycle.

2. $90 \leq h[] < 190$, $(stat.s, stat.h) = (0, h[] - 100)$: Surface cells that find an empty cell in the first cycle.

3. $190 \leq h[]$, $(stat.s, stat.h) = (-1, h[] - 200)$: Surface cells that find a full cell in the first cycle.

4. $-10 \leq h[] < 90$, $(stat.s, stat.h) = (-1, h[])$: Empty/full cells that find a coherent surface in the first cycle.

For scenarios 2 and 3, S and H are set to $stats.s$ and $stats.h$ to continue searching for the other end. Otherwise, S and H retain their initial values.

Second Layer

As shown in figure 7.9, there are five categories of scenarios leading to four types of results, stemming from three types of S rather than the current cell's volume fraction:

A $0 < S = f < 1$: Find an empty/full cell, S switches to 0/1, resulting in condition I/II. One end of a coherent surface is found.

B $S = 1$: Find an empty cell, indicating a coherent surface. S switches to -1, completing the iteration (condition III).

C $S = 0$: Find a full cell, indicating a coherent surface. S switches to -1, completing the iteration (condition III).

D $S = 1$ or $S = 0$: Scan a surface cell and return to one with the same status as S. Since the program scans only four neighboring cells, this suggests failure to find a coherent surface in this direction, addressing cases like figure 7.8. This leads to condition IV, with S unchanged.

Others : All other cases fail to find a coherent surface or at least one end, leaving S unchanged and leading to condition IV.

Height value accumulation occurs in this layer, implicitly shown in figure 7.9. Following the protocol in section 7.2.3, the height value is computed and adjusted by adding $HSHIFT = 20$ if the surface direction opposes the coordinate, as discussed in section 7.2.1.

Third Layer

For the negative direction ($j = -1$), the output status and height value are encoded into $h[]$ in this layer. The four conditions from the second layer are grouped into three categories:

A Condition IV, encoded as $h[] = 300$: Cases failing to find a coherent surface, marked as 'inconsistent'.

B Condition III, not encoded, $h[] = H$: Cases successfully finding a coherent surface.

C Conditions I and II, encoded as I: $h[] = H + 100$, II: $h[] = H + 200$: Surface cells finding an empty/full cell as one end of a coherent surface.

For the positive direction ($j = 1$), the third layer serves as the final output. Before assigning $h[]$, two scenarios update the data as $(stats.s, stats.h) = (S, H)$:

1. Condition V: Surface cell as the current cell, finding one end in the first cycle.
2. Condition III: Coherent surface found in the second cycle, with the second cycle's height value less than the first cycle's, i.e., $H < stats.h$.

The second scenario ensures that when a cell has two valid height values, the smaller one is used. Since $nodata$ is a large integer in Basilisk, this also covers cases failing in the first cycle but succeeding in the second.

The final height value $h[]$ is assigned based on $(stats.s, stats.h)$, as shown in figure 7.9.

Program Workflow

First Layer: Initial Setup
Input: $point$, c, h, cs, j.
Initialize $S = f$, $H = f$, where ci and a are used in the second layer. Iterates over each dimension to compute $h.x$, $h.y$, $h.z$.

```
1  static void half_column (Point point, scalar c, vector h, vector cs, int
   ↪  j)
2  {
3    const int complete = -1;
4    foreach_dimension() {
5      double S = c[], H = S, ci, a;
```

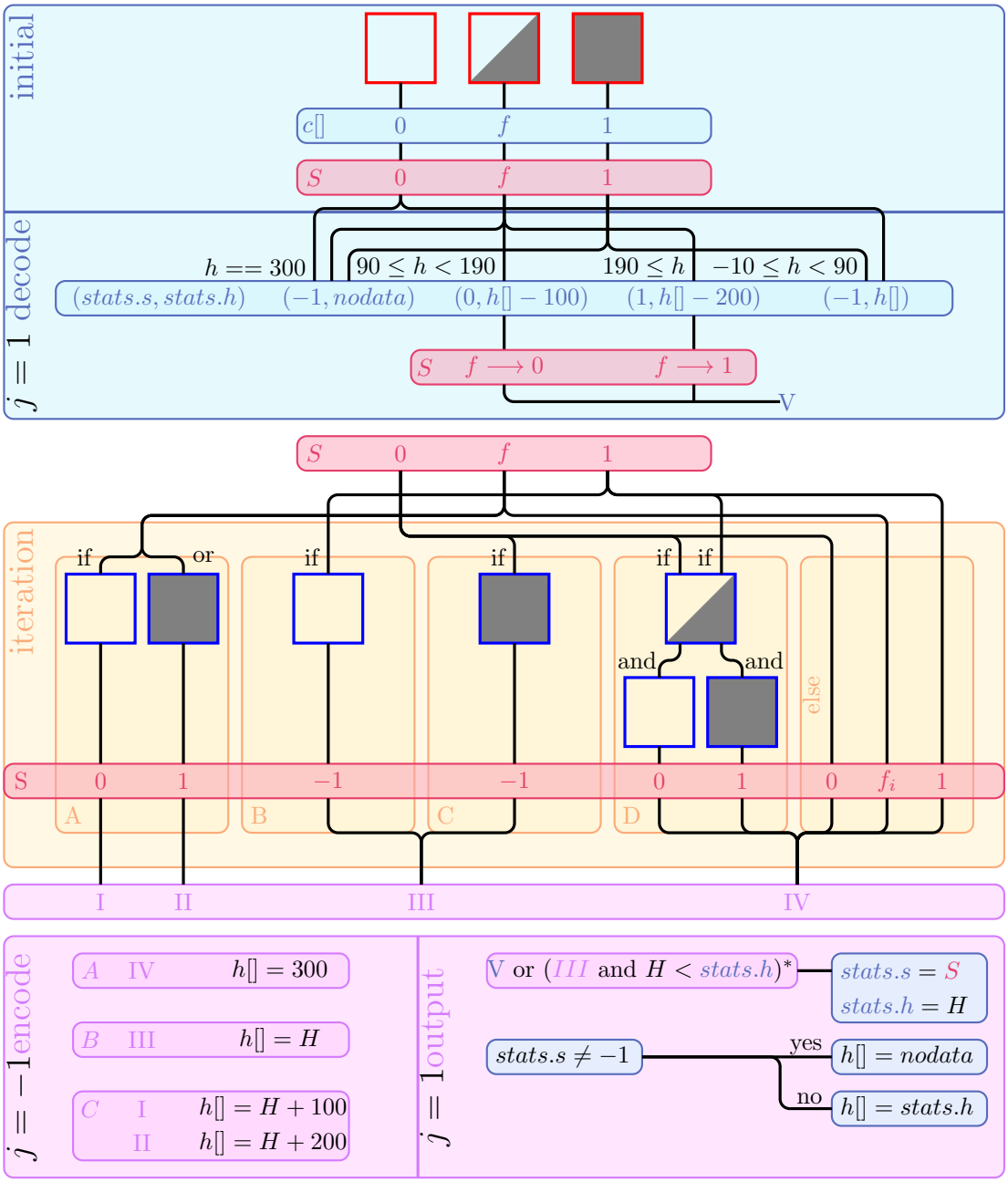


Figure 7.9: Detailed description of **half_column**. For simplicity, some tricky treatments are hidden. For instance, the judgment in the third layer highlighted by $*$ compares surface distance rather than direct values of H and $stats.h$. Thus, the actual judgment is $fabs(height(H)) < fabs(height(stats.h))$.

First Layer: Decode (Part 1)
If the condition is inconsistent (condition IV), set $state.h = -1$ and begin a new iteration.

```
typedef struct { int s; double h; } HState;
HState state = {0, 0};
if (j == 1) {
  if (h.x[] == 300.)
    state.s = complete, state.h = nodata;
```

First Layer: Decode (Part 2)
Decode cases other than condition IV. Since s is an integer, only cases with $h[] \geq 90$ (conditions II and III) yield non-zero s, restoring previous status as $(S, H) = (state.s, state.h)$. Otherwise (condition III), the iteration uses the original initial values.

```
  else {
    int s = (h.x[] + HSHIFT/2.)/100.;
    state.h = h.x[] - 100.*s;
    state.s = s - 1;
  }
  if (state.s != complete)
    S = state.s, H = state.h;
}
```

Second Layer: Iteration (Part A)
Scan four neighboring cells. If $1 > S > 0$ and an empty or full cell is found ($ci = 0$ or $ci = 1$), update S and modify H based on its position relative to the zero-value point.

```
for (int i = 1; i <= 4; i++) {
  ci = i <= 2 ? c[i*j] : cs.x[(i - 2)*j];
  H += ci;
  if (S > 0. && S < 1.) {
    S = ci;
    if (ci <= 0. || ci >= 1.) {
```

```
7           H -= i*ci;
8           break;
9         }
10      }
```

Second Layer: Iteration (Parts B and C)
Part B: For full current cells finding an empty cell, stop iteration, set S to complete, and compute height as discussed.
Part C: For empty current cells finding a full cell.

```
1       else if (S >= 1. && ci <= 0.) {
2         H = (H - 0.5)*j + (j == -1)*HSHIFT;
3         S = complete;
4         break;
5       }
6       else if (S <= 0. && ci >= 1.) {
7         H = (i + 0.5 - H)*j + (j == 1)*HSHIFT;
8         S = complete;
9         break;
10      }
```

Second Layer: Iteration (Part D)
If a surface cell is scanned but no coherent surface is found, stop the iteration.

```
1       else if (S == ci && modf(H, &a))
2         break;
3     }
```

Third Layer: Encoding (Part A)
Inconsistent surfaces (condition IV) are encoded as $h[] = 300$.

```
    if (j == -1) {
      if (S != complete && ((c[] <= 0. || c[] >= 1.) || (S > 0. && S <
      ↪  1.)))
        h.x[] = 300.; // inconsistent
```

Third Layer: Encoding (Part B)
Complete cases (condition III) assign *h[]* directly as *H*.

```
      else if (S == complete)
        h.x[] = H;
```

Third Layer: Encoding (Part C)
Partial heights (conditions I and II) are encoded
based on whether an empty or full cell was found.

```
      else
        h.x[] = H + 100.*(1. + (S >= 1.));
    }
```

Third Layer: Output (Part D)
For certain cases, update *state.s* and *state.h*. See section 7.2.3 for details.

```
    else { // j = 1
      if (state.s != complete ||
          (S == complete && fabs(height(H)) < fabs(height(state.h))))
        state.s = S, state.h = H;
```

Third Layer: Output (Part E)
Assign *h[]* based on *state.s* and *state.h*.

```
      if (state.s != complete)
        h.x[] = nodata;
      else
        h.x[] = (state.h > 1e10 ? nodata : state.h);
    }
  }
}
```

7.2.4 column_propagation

The **column_propagation** function supplements **half_column** by assigning height values to cells within $5.5R$ of the zero-value point that received 'nodata' from **half_column**. For example, cell G in figure 7.7 is valid and should have a height value of -3.8, but it receives 'nodata' due to failing to find a coherent surface in its four neighbors. **column_propagation** addresses this issue.

Parameters

Name	Data type	Status	Option/Default	Representation (before/after)
h	vector	**updated**	compulsory	**h**

Worth Mentioning Details

Similar to **half_column**, this function iterates two neighbors in both directions and updates the height value based on the spatial relation with the smallest height cell.

Program Workflow

Propagation
Input: $h = \mathbf{h}$.
Search four neighbors for valid height values, then update the current cell's value. The inline function **height** is called to reveal the true distance to the zero-value point.

```
static void column_propagation (vector h)
{
  foreach (serial) // not compatible with OpenMP
    for (int i = -2; i <= 2; i++)
      foreach_dimension()
        if (fabs(height(h.x[i])) <= 3.5 &&
            fabs(height(h.x[i]) + i) < fabs(height(h.x[])))
          h.x[] = h.x[i] + i;
}
```

7.2.5 heights for Non-Tree Grid

As discussed in section 7.2.1, **heights** controls **half_column** and **column_propagation** to assign height values to valid cells. The non-tree grid version is presented first, sharing the same structure as the tree grid version.

Parameters

Name	Data type	Status	Option/Default	Representation (before/after)
c	scalar	unchanged	compulsory	color function c
h	vector	**updated**	compulsory	**h**

Worth Mentioning Details

Since the default boundary settings provide only two cells, an additional vector s is created to support iteration over four cells in both directions. The value stored at position $[x_0, y_0]$ for s is:

$$s.x[x_0, y_0] = c[x_0 + 2j, y_0] \tag{7.4}$$

where j indicates the direction (see figure 7.5). Each component of s represents a value translation in the corresponding direction.

Program Workflow

Starting Point
Input: $c = c$, $h = \mathbf{h}$.
Boundary Settings: Ensure s has the same boundary settings as c.
Direction Iteration: j represents the current direction, with $j = -1$ (resp. 1) indicating negative (resp. positive) direction.
Value Assignment for s: Implement equation 7.4.

```
#if !TREE
trace
void heights (scalar c, vector h)
{
  vector s[];
  foreach_dimension()
    for (int i = 0; i < nboundary; i++)
      s.x.boundary[i] = c.boundary[i];
  for (int j = -1; j <= 1; j += 2) {
    foreach()
      foreach_dimension()
        s.x[] = c[2*j];
```

Height Value Assignment
Call half_column: Assign height values for each cell in every direction.
Call column_propagation: Propagate height values to ensure cells within $5.5R$ have valid height values.

```
    foreach (overflow)
      half_column (point, c, h, s, j);
  }
  column_propagation (h);
}
```

Chapter 8

myc2d.h,myc.h Documentation

Version: draft Updated: 2025-07-15

8.1 Introduction

'MYC' stands for the Mixed Young's Centered scheme, introduced by Scardovelli et al. [2]. It is used to compute the interface normal vector $\mathbf{n}$ at a given cell, based on the volume fraction c in the current cell and its neighboring cells (9 cells in 2D and 27 cells in 3D).

In the following sections, I will first introduce the basic concept of representing an interface with a linear function, followed by the algorithm that enables the mixed scheme to determine the normal direction in 2D cases. This will then be extended to the 3D scenario.

8.2 2D scenario

8.2.1 Linear Representation of the Interface

Consider an interfacial cell on a 2D plane, as shown in figure 8.1. Let the normal direction of the interface be denoted by $\mathbf{n} = (n_x, n_y)$. It is straightforward to show that the interface can be represented by the linear equation

$$n_x X + n_y Y = \alpha, \tag{8.1}$$

where α is a constant that determines the position of the interface.

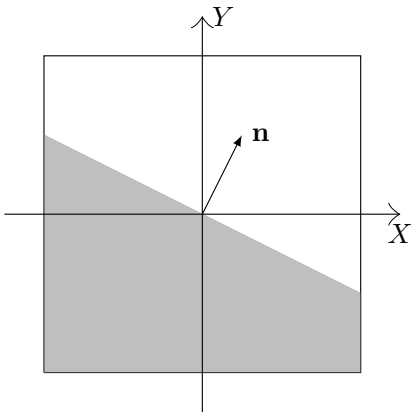


Figure 8.1: Linear representation of the interface, with normal direction $\mathbf{n} = (n_x, n_y)$.

This equation indicates that the linear interface within a cell is fully characterized once the triplet (n_x, n_y, α) is known.

If we treat Y as a function of X, the equation takes the slope-intercept form

$$sign(n_y)Y = -\frac{n_x}{|n_y|}X + \frac{\alpha}{|n_y|} = m_y X + b_y, \tag{8.2}$$

where $m_y = -\frac{n_x}{|n_y|}$ and $b_y = \frac{\alpha}{|n_y|}$.

Similarly, X can be expressed as a function of Y, with slope $m_x = -\frac{n_y}{|n_x|}$. The normal direction obtained from equation 8.2 can thus be written in the form $(-m_x, , \text{sign}(n_y))$ (or equivalently, $(\text{sign}(n_x), , -m_y)$). This form is derived by transforming equation 8.2 back into the original form of the interface representation, as shown in equation 8.1.

With the interface representation clarified, we now introduce the centered scheme, Young's scheme, and finally the mixed scheme. As will be shown, the centered scheme estimates the interface normal direction through the form of equation 8.2, while Young's scheme directly computes the normal vector (n_x, n_y).

8.2.2 Centered columns scheme

For the linear equation 8.2, the most straightforward way to compute the slope is by selecting an arbitrary pair of points on the line, denoted as (X_1, Y_1) and (X_2, Y_2), and calculating

$$m_x = \frac{Y_2 - Y_1}{X_2 - X_1}. \tag{8.3}$$

The centered columns scheme adopts the same idea but operates under discrete conditions. This naturally raises several questions:

1. Given a value of X, how can we determine the corresponding output Y in a discrete setting?

2. Since there are two slope estimates, which one should be selected?

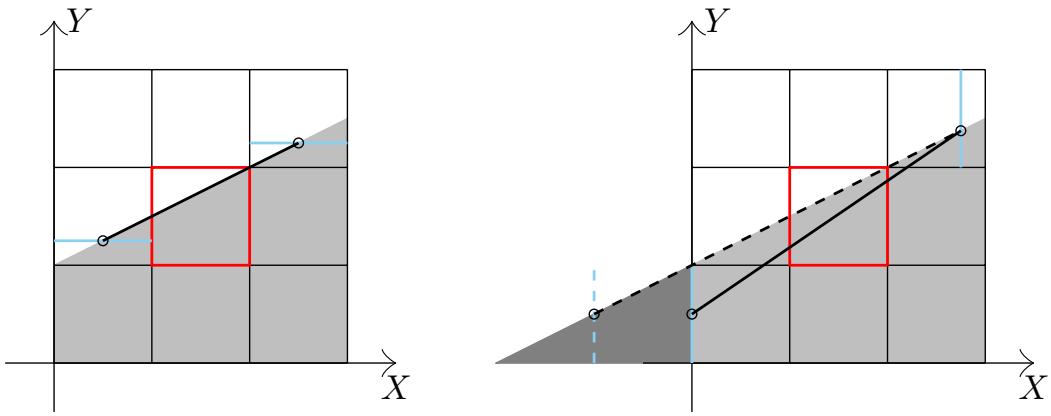


Figure 8.2: Sketch of the computation process for the centered scheme. The red cell highlights the current cell where the interfacial slope is being computed. The blue lines represent the summation of volume fractions across columns, and the black solid line shows the resulting slope obtained by the centered scheme. In figure (a), the scheme accurately captures the interface when it spans across opposite boundaries. By contrast, figure (b) presents the same scenario but from a complementary perspective. The result reveals a significant discrepancy between the computed and actual slopes. Note that the dashed and solid lines are intentionally reversed to facilitate comparison with the true slope of the interface.

Figure 8.2a illustrates the first question by showing how the centered scheme computes the interfacial slope. Similar to the height function method described in the `heights.h` documentation, the value of Y at each column is approximated by summing the volume fractions c of three adjacent cells aligned along the same X coordinate. Once two neighboring values (marked as empty circles) are obtained, the slope at the current cell (highlighted in red) is computed using a centered finite difference. The result is indicated by the solid black line.

Let the current cell be denoted by index $(0, 0)$. The slope in figure 8.2a is then given by:

$$m_x = \frac{1}{2}\left(\sum_{j=-1}^{1} c_{1,j} - \sum_{j=-1}^{1} c_{-1,j}\right) \tag{8.4}$$

Since $m_x = -\frac{n_y}{|n_x|}$, it follows that $\text{sign}(n_x) = -\text{sign}(m_x)$, and similarly, $\text{sign}(n_y) = -\text{sign}(m_y)$. This dual representation for the same interface raises the second question: which of the two directional estimates gives the more accurate result?

Figure 8.2b presents the same scenario but treats Y as the independent variable and X as the dependent output. In this case, the slope computed by the centered scheme deviates significantly from the true interface, as part of the left column (shown in dark gray) is omitted. For a perfectly linear interface, the centered scheme yields accurate results only when the interface intersects the left and right boundaries of the current cell. However, when the interface crosses adjacent boundaries (e.g., left and top) or spans the top and bottom boundaries, the accuracy deteriorates due to partial volume loss, as

illustrated in figure 8.2b. In other words, the smaller the absolute value of the slope, the more accurate the estimate [1].

Based on this observation, the normal direction is chosen as $(-m_x, sign(m_y))$ when $|m_y| > |m_x|$, and as $(sign(m_x), -m_y)$ otherwise.

8.2.3 Young's Scheme

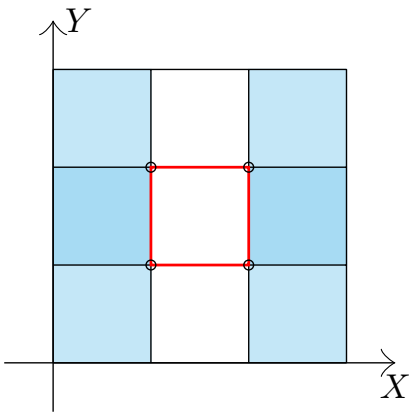


Figure 8.3: Illustration of Young's scheme. The interface normal is obtained by averaging the normals at four cell corners, indicated by empty circles. Blue shading marks the cells involved in the final algebraic expression, with dark blue showing overlaps.

Young's scheme estimates the interface normal vector $\mathbf{n}$ by averaging the normals computed at four cell corners. As shown in figure 8.3, the normal at each corner is evaluated using central differences of the surrounding cell values. For example, considering the target cell with index $(0, 0)$, the x-component of the normal at corner $(1/2, 1/2)$ is given by

$$n_{x,(1/2,1/2)} = \frac{1}{2\Delta}\left(c_{1,1} + c_{1,0} - c_{0,1} - c_{0,0}\right) \tag{8.5}$$

The average x-component of the normal over all four corners is then computed as

$$n_x = \frac{1}{4}\left(n_{x,(1/2,1/2)} + n_{x,(-1/2,1/2)} + n_{x,(1/2,-1/2)} + n_{x,(-1/2,-1/2)}\right) \tag{8.6}$$

Alternatively, a more compact expression for n_x can be obtained by directly summing the contributions from relevant cells:

$$n_x = \frac{1}{8}\left(C_1 - C_{-1}\right) \tag{8.7}$$

where

$$C_1 = c_{1,1} + 2c_{1,0} + c_{1,-1}, \quad C_{-1} = c_{-1,1} + 2c_{-1,0} + c_{-1,-1} \tag{8.8}$$

as indicated by the blue regions in figure 8.3. The same procedure applies for computing n_y. Note that the coefficient $\frac{1}{8}$ may be omitted or replaced, provided both components of $\mathbf{n}$ are scaled consistently.

8.2.4 Mixed Young's and Centered scheme

In the previous two sections, two normals—$\mathbf{n}_Y$ and $\mathbf{n}_C$—were computed separately, with the subscripts denoting the respective schemes. The MYC scheme unifies these by introducing a criterion to select the more accurate normal. In summary, the criterion favors the normal with the larger absolute value. While $\mathbf{n}_C$ is already normalized, $\mathbf{n}_Y$ must be normalized consistently—based on the selected $\mathbf{n}_C$—to enable a fair comparison.

Consider the example shown in figure 8.4, where the interface has a slope of 3/4. For simplicity, the cut-cell areas are labeled locally. Although the interface intersects adjacent cell boundaries, the omitted area (shaded in dark gray) under the current frame is significantly smaller than in the alternative case, suggesting that the resulting normal is more accurate—specifically, $|n_{C,x}| = \frac{35}{48}$. However, the normal obtained using Young's scheme is $|n_{Y,x}| = \frac{25}{35}$, which is not only favored by the MYC criterion due to its larger magnitude, but also closer to the exact value, making it in fact more accurate than the centered scheme in this case. This improvement stems from the averaging process in Young's scheme, which reduces the influence of the omitted area that would otherwise tend to reduce the magnitude of the normal.

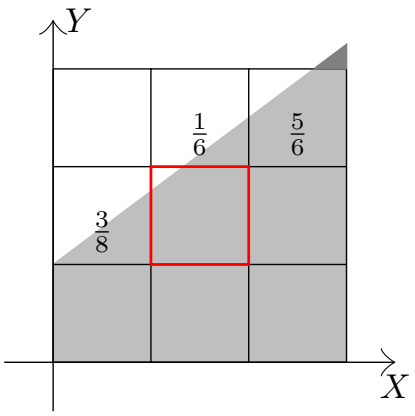


Figure 8.4: An example illustrating the difference between the two schemes. The red cell denotes the target interfacial cell, with the interface having a slope of 3/4. For simplicity, the cut-cell areas are labeled locally within each intersected cell.

8.2.5 myc (2D version)

Parameters

Name	Data type	Status	Option/Default	Representation (before/after)
point	Point	unchanged	compulsory	index (x, y)
c	scalar	unchanged	compulsory	c

Program Workflow

Starting Point

ix: flag for normal selection in Centered Scheme;
c_t,c_b,c_r,c_l: volume summation of four columns;
mx0,my0:normal components obtained by Centered scheme;*mx1,my1*: normal components obtained by Young's; *mm1,mm2* swap value.

```
  coord mycs (Point point, scalar c)
{
  int ix;
  double c_t,c_b,c_r,c_l;
  double mx0,my0,mx1,my1,mm1,mm2;
```

Centered Scheme Computation

To facilitate equation 8.4, note here *mx0*=$-m_x$, *my0*=$-m_y$

```
  c_t = c[-1,1] + c[0,1] + c[1,1];
  c_b = c[-1,-1] + c[0,-1] + c[1,-1];
  c_r = c[1,-1] + c[1,0] + c[1,1];
  c_l = c[-1,-1] + c[-1,0] + c[-1,1];

  mx0 = 0.5*(c_l - c_r);
  my0 = 0.5*(c_b - c_t);
```

Centered Scheme Criterion

Select the normal computed by the centered scheme according to the criterion described in section 8.2.2, and unify the remaining components accordingly. The flag *ix* is used to indicate which normal has been selected.

```
  if (fabs(mx0) <= fabs(my0)) {
    my0 = my0 > 0. ? 1. : -1.;
    ix = 1;
  }
  else {
```

```
      mx0 = mx0 > 0. ? 1. : -1.;
      ix = 0;
    }
```

Young's Scheme Computation

Facilitation of equation 8.8, the ***NOT_ZERO*** is introduced to avoid division by zero.

```
    mm1 = c[-1,-1] + 2.0*c[-1,0] + c[-1,1];
    mm2 = c[1,-1] + 2.0*c[1,0] + c[1,1];
    mx1 = mm1 - mm2 + NOT_ZERO;
    mm1 = c[-1,-1] + 2.0*c[0,-1] + c[1,-1];
    mm2 = c[-1,1] + 2.0*c[0,1] + c[1,1];
    my1 = mm1 - mm2 + NOT_ZERO;
```

Mixed Centered and Young's Scheme Criterion

If $ix = 0$, i.e., the normal $(-m_x, \text{sign}(m_y))$ is selected as the final output of the centered scheme, then $\mathbf{n}Y$ is normalized using the value of ny, Y—and vice versa. The final output is determined by the criterion described in section 8.2.4 which compares the absolute value of corresponding subcomponent and sored the larger pair in (***mx0***,***my0***).

```
    if (ix) {
    mm1 = fabs(my1);
    mm1 = fabs(mx1)/mm1;
    if (mm1 > fabs(mx0)) {
      mx0 = mx1;
      my0 = my1;
    }
  }
  else {
    mm1 = fabs(mx1);
    mm1 = fabs(my1)/mm1;
    if (mm1 > fabs(my0)) {
      mx0 = mx1;
```

```
      my0 = my1;
    }
  }
```

Final Output

The final normal, denoted as **n**, will be normalized such that $n_x + n_y = 1$, to align with the form used in 3D. The reason for this normalization will be explained in the following section.

```
  mm1 = fabs(mx0) + fabs(my0);
  coord n = {mx0/mm1, my0/mm1, 0};

  return n;
}
```

Bibliography

[1] Eugenio A. et al. “Interface reconstruction with least-squares fit and split advection in three-dimensional Cartesian geometry”. In: *J. Comput. Phys.* 225.2 (2007), pp. 2301–2319.

[2] R. Scardovelli and S. Zaleski. “Interface reconstruction with least-square fit and split Eulerian–Lagrangian advection”. In: *Int. J. Numer. Methods Fluids* 41.3 (2003), pp. 251–274.

Chapter 9

embed-tree.h Documentation

Version: draft Updated: 2025-06-09

9.1 Introduction

This header file provides restriction, prolongation, and refinement operations under a tree grid for values used in `embed.h`. The content is divided into two parts: the refinement of cut-cell values, i.e., *cs* and *fs*, and the restriction/prolongation of values on cut-cells and cut-faces.

9.2 Refinement of Embed-Associated Values

9.2.1 embed_fraction_refine

The **embed_fraction_refine** function calculates the volume fraction contained by each child cell.

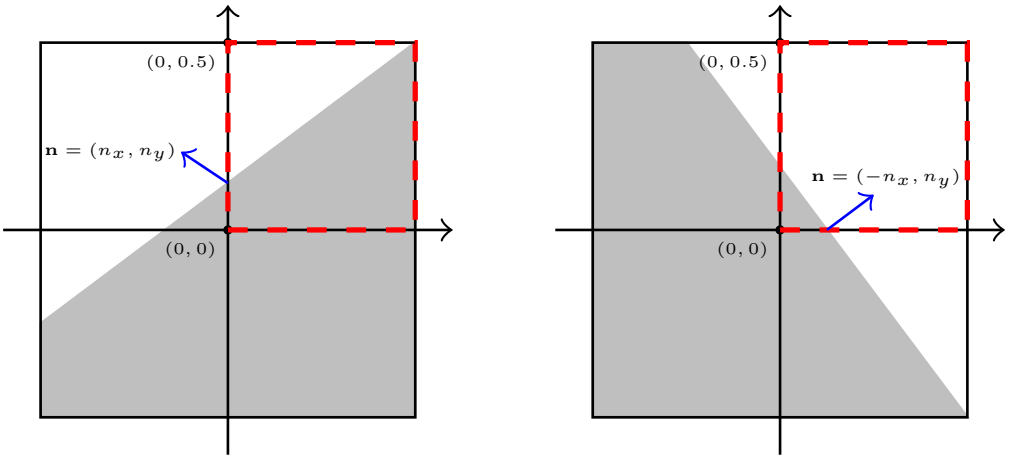
Worth Mentioning Details

For cells containing an interface, given the function **rectangle_fraction** from `geometry.h` and interface information $\mathbf{n}$, α, the volume fraction inside each child cell can be obtained. As shown in figure 9.1a, under a 2D condition, the area inside the child cell highlighted by a red dashed line is calculated.

Instead of switching the computational domain (i.e., the red dashed cell), the program iterates between cells by switching the sign of the normal direction, as indicated in figure 9.1b.

Parameters

Name	Data type	Status	Option	Representation (before/after)
point	Point	unchanged	compulsory	position index
cs	scalar	unchanged	compulsory	volume fraction *cs*



(a) (b)

Figure 9.1: Refinement of embed fraction *cs*. By switching the component of the normal direction **n**, the program rotates the interface, thereby iterating the volume fraction of each subcell.

Program Workflow

Starting Point
Input: $cs = cs$, the volume fraction, $cc = cs[]$.
Non-Interfacial Cell: For full or empty cells, the volume fraction of child cells is directly inherited from the parent cell.

```
1  static void embed_fraction_refine (Point point, scalar cs)
2  {
3    double cc = cs[];
4    if (cc <= 0. || cc >= 1.) {
5      foreach_child()
6        cs[] = cc;
7    }
```

↓

Refinement of Volume Fraction on Interfacial Cell
Interface Information: $n = \mathbf{n}$, $alpha = \alpha$.
Volume Fraction Refinement: After iterating each child cell using the macro `foreach_child`, the computational domain is confined with coordinates *a* and *b* (the red square in figure 9.1). An additional normal direction *nc* is defined to switch between child cells (see 'vof.h' documentation for the definition of index *child.x*). The volume fraction for each child cell is then obtained.

```
    coord n = facet_normal (point, cs, fs);
    double alpha = plane_alpha (cc, n);

    foreach_child() {
      static const coord a = {0.,0.,0.}, b = {.5,.5,.5};
      coord nc;
      foreach_dimension()
        nc.x = child.x*n.x;
      cs[] = rectangle_fraction (nc, alpha, a, b);
    }
  }
}
```

9.2.2 embed_face_fraction_refine

Given the face vector *fs* on mesh level N, the function **embed_face_fraction_refine** returns *fs* on the finer mesh level $N+1$. The computation of face fractions is achieved by first calculating the fractions of 'inner' faces, followed by 'boundary' faces, under both 2D and 3D conditions. For clarity, the implementation for 2D cases is introduced first, followed by 3D cases. However, the program is organized based on computations for 'inner' or 'boundary' faces.

Note that the function is duplicated by the macro `foreach_dimension`, and each instance is responsible for face fraction computation in one direction. The following explanation uses the x-direction as an example.

Moreover, for a cut-cell (*cs* $\in (0,1)$), the interface is defined by:

$$\mathbf{n} \cdot \mathbf{x} = \alpha \tag{9.1}$$

where $\mathbf{n}$ is the normal direction of the interface, $\mathbf{x}$ represents the coordinates of any point on the interface, and α is a constant. This equation is critical for identifying different conditions in the following sections.

2D Condition

First, consider the fine faces contained inside the cell, as shown by A and B in figure 9.2.

The initial step in computing face fractions is to determine whether the interface intersects the inner fine faces. Given $\alpha = \alpha_1$ and $\mathbf{n} = (n_x, n_y)$ for the interface, and a coordinate system with $(0,0)$ at the cell's center, the lower and upper boundaries are lines passing through $(0, 0.5)$ and $(0, -0.5)$, respectively. According to equation 9.1, if:

$$|2\alpha_1| < |n_y| \tag{9.2}$$

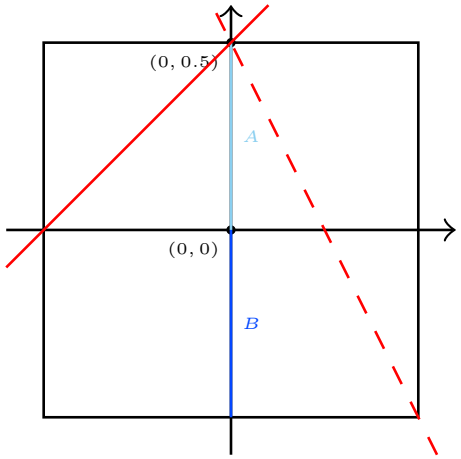


Figure 9.2: Sketch of inner fine faces A, B. The two red lines represent example threshold values for the interfaces.

the interface intersects the inner fine faces; otherwise, both face fractions are 0 or 1, depending on the interface's direction. If equation 9.2 holds, the intersection point $(0, y_i)$ between the interface and the y-axis is $y_i = \frac{\alpha}{n_y}$. The face fractions for faces A and B can then be computed accordingly.

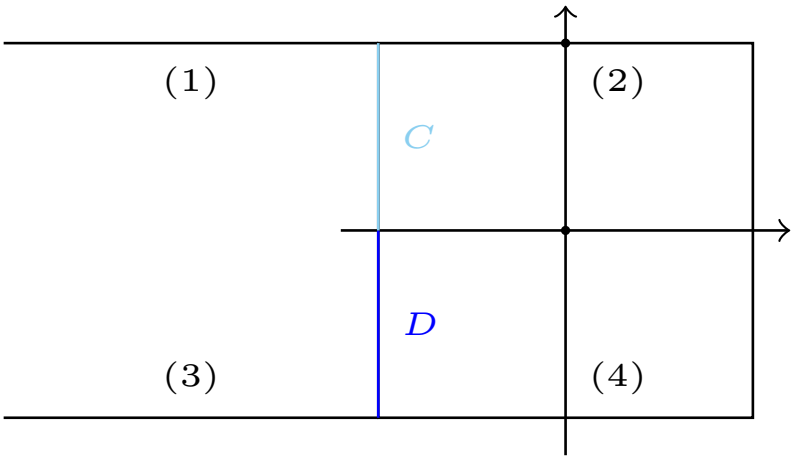


Figure 9.3: Sketch of boundary fine faces C, D. The numerical labels mark the corresponding transverse faces.

For boundary fine faces, a similar process is followed: first, identify the intersection point (if it exists), then compute the fine face fraction. Unlike the inner faces, boundary face fractions can be accessed directly without checking equation 9.2. The interface's orientation is determined by examining the corresponding transverse faces, as shown by labels (1–4) in figure 9.3. Combined with the face fraction on the original face, the fine face fraction is obtained.

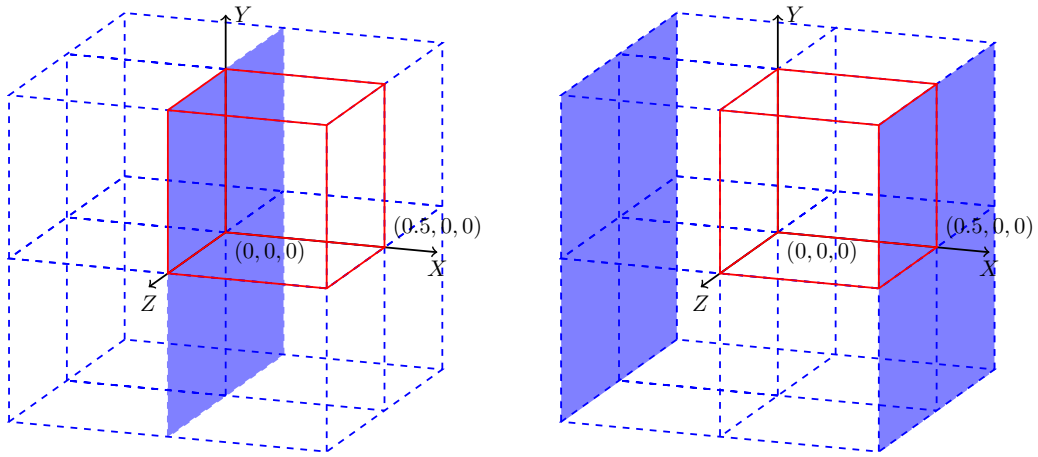


Figure 9.4: Inner fine face (a) and boundary fine face (b) under 3D conditions. The red contour highlights the cell where projection takes place.

3D Condition

The fine face fractions for both inner and boundary faces are computed directly using 2D projections from a 3D interface.

For the inner face, as shown in figure 9.4a, the fine face fraction is computed from a 2D projection in the red-contoured cell. Iteration among the four subcells is achieved by rotating the entire interface, as described in section (unfinished). Given $\mathbf{n} = (n_x, n_y, n_z)$ and $\alpha = \alpha_1$ for the example interface in figure 9.5a, the face fraction on the inner fine face is obtained by calculating the volume fraction of the reconstructed interface shown in figure 9.5b. Since volume calculations are supported by tools in `geometry.h`, the problem reduces to reconstructing the interface and determining its $\mathbf{n}$ and α.

The reconstructed interface is perpendicular to the projection plane and intersects the same line as the original interface on the projection plane, with $\mathbf{n} = (0, n_y, n_z)$ and $\alpha = \alpha_1$.

Similarly, the fine face fraction on the boundary face is computed by projection onto the boundary plane. As shown in figure 9.5c, the reconstructed plane must pass through the intersection line on the boundary face (blue line) and the red dashed line. Given an arbitrary point set $(0.5, y_1, z_1)$ and $(0, y_1, z_1)$, the $\alpha = \alpha_2$ of the reconstructed interface satisfies:

$$0.5n_x + y_1 n_y + z_1 n_z = \alpha_1 \tag{9.3}$$

$$y_1 n_y + z_1 n_z = \alpha_2 \tag{9.4}$$

$$\alpha_2 = \alpha_1 - 0.5n_x \tag{9.5}$$

With the normal direction $(0, n_y, n_z)$, the face fraction can be calculated.

Worth Mentioning Details

In addition to geometric formulas, the function **fine** is called multiple times to access data on the finer mesh. The index for the face vector of a subcell is shown in figure 9.6,

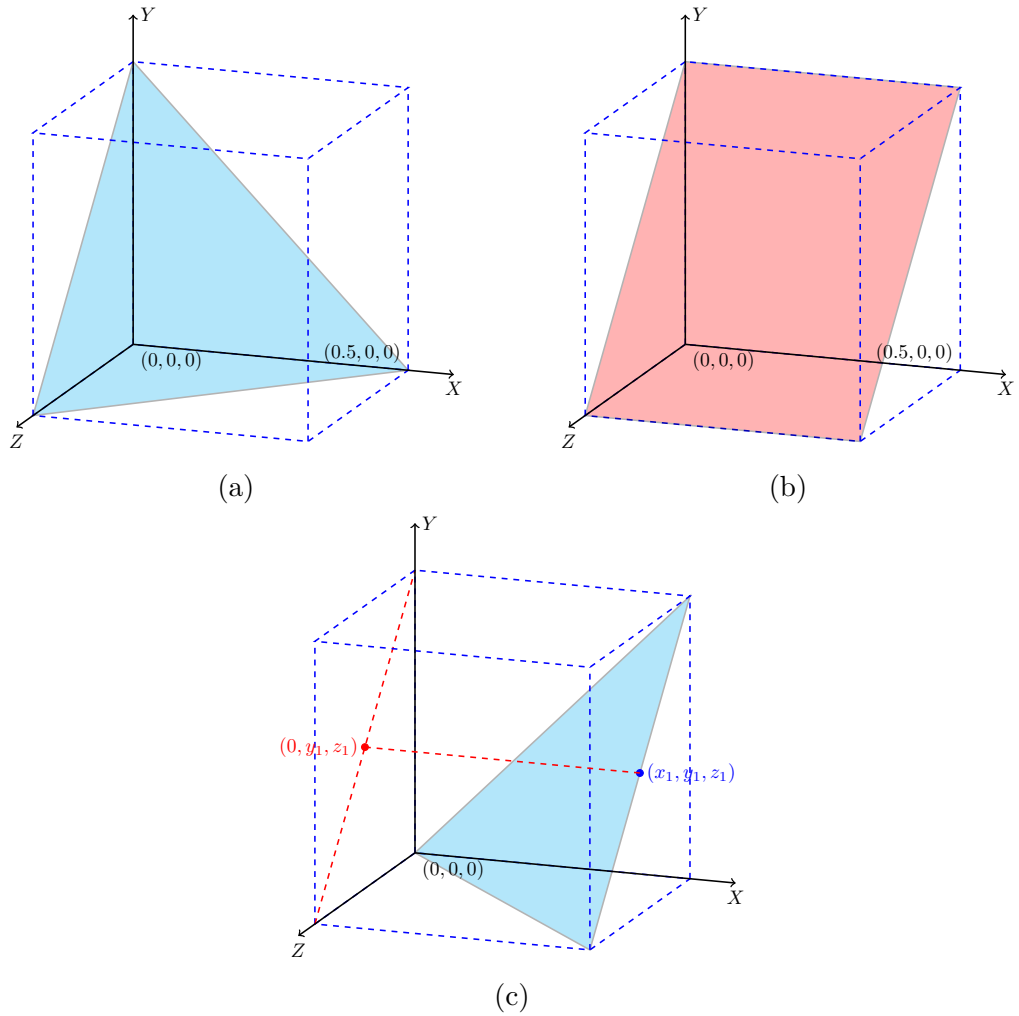


Figure 9.5: Zoom-in view of the highlighted cell in figure 9.4. (a) The original interface, (b) the projected interface for the inner face in figure 9.4a, and (c) the original interface for the boundary fine face in figure 9.4b.

which can be directly extended to 3D conditions.

Parameters

Name	Data type	Status	Option	Representation (before/after)
point	Point	unchanged	compulsory	position index
s	scalar	unchanged	compulsory	volume fraction *cs*

Program Workflow

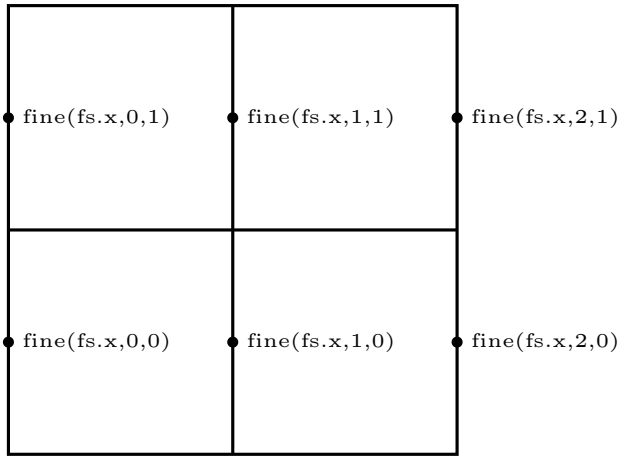


Figure 9.6: Index for **fine**.

Starting Point

Input: $s = cs$, face vector attributed with volume fraction, i.e., $fs = s.v$.
Full & Empty Cell: For full and empty cells, the face fraction is directly assigned the parent value. For boundary fine face fractions, a check ensures consistency with neighboring cells.

```
foreach_dimension()
static void embed_face_fraction_refine_x (Point point, scalar s)
{
  vector fs = s.v;
  if (cs[] <= 0. || cs >= 1.) {
    for (int j = 0; j <= 1; j++)
      for (int k = 0; k <= 1; k++)
        fine(fs.x,1,j,k) = cs[];
    for (int i = 0; i <= 1; i++)
      if (!is_refined(neighbor(2*i-1)) && neighbor(2*i-1).neighbors &&
          (is_local(cell) || is_local(neighbor(2*i-1))))
        for (int j = 0; j <= 1; j++)
          for (int k = 0; k <= 1; k++)
            fine(fs.x,2*i,j,k) = fs.x[i];
  }
```

Computation of Inner Fine Face Fraction
Interfacial Information: Since the current cell contains an interface, obtain $n = \mathbf{n}$, $alpha = \alpha$.
2D Condition: For interfaces satisfying equation 9.2, which cut through the inner face, compute the fine face fraction using the intersection $y_i = \frac{\alpha}{n_y}$. Otherwise, the fine face fraction is full or empty.
3D Condition: As discussed, compute the fine face fraction from the volume fraction of the reconstructed interface with $\mathbf{n}_{re} = (0, n_y, n_z)$, $\alpha_{re} = \alpha$. Iteration between subcells is implemented by rotating the original cell.

```
  else {
    coord n = facet_normal (point, cs, fs);
    double alpha = plane_alpha (cs[], n);
#if dimension == 2
    if (2.*fabs(alpha) < fabs(n.y)) {
      double yc = alpha/n.y;
      int i = yc > 0.;
      fine(fs.x,1,1 - i) = n.y < 0. ? 1. - i : i;
      fine(fs.x,1,i) = n.y < 0. ? i - 2.*yc : 1. - i + 2.*yc;
    }
    else
      fine(fs.x,1,0) = fine(fs.x,1,1) = alpha > 0.;
#else // dimension == 3
    for (int j = 0; j <= 1; j++)
      for (int k = 0; k <= 1; k++)
        if (!fine(cs,0,j,k) || !fine(cs,1,j,k))
          fine(fs.x,1,j,k) = 0.;
        else {
          static const coord a = {0.,0.,0.}, b = {.5,.5,.5};
          coord nc;
          nc.x = 0., nc.y = (2.*j - 1.)*n.y, nc.z = (2.*k - 1.)*n.z;
          fine(fs.x,1,j,k) = rectangle_fraction (nc, alpha, a, b);
        }
#endif // dimension == 3
```

Computation of Boundary Fine Face Fraction
Full/Empty Cell Detection: For full/empty cells, assign fine face fractions as 1/0.
2D Condition: Follow the procedure described to compute boundary fine face fractions.
3D Condition: Compute boundary fine face fractions using equation 9.5 for α_{re}.

```
    for (int i = 0; i <= 1; i++)
      if (neighbor(2*i-1).neighbors &&
          (is_local(cell) || is_local(neighbor(2*i-1)))) {
        if (!is_refined(neighbor(2*i-1))) {
          if (fs.x[i] <= 0. || fs.x[i] >= 1.)
            for (int j = 0; j <= 1; j++)
              for (int k = 0; k <= 1; k++)
                fine(fs.x,2*i,j,k) = fs.x[i];
          else {
#if dimension == 2
            double a = fs.y[0,1] <= 0. || fs.y[2*i-1,1] <= 0. ||
              fs.y[] >= 1. || fs.y[2*i-1] >= 1.;
            if ((2.*a - 1)*(fs.x[i] - 0.5) > 0.) {
              fine(fs.x,2*i,0) = a;
              fine(fs.x,2*i,1) = 2.*fs.x[i] - a;
            }
            else {
              fine(fs.x,2*i,0) = 2.*fs.x[i] + a - 1.;
              fine(fs.x,2*i,1) = 1. - a;
            }
#else // dimension == 3
            for (int j = 0; j <= 1; j++)
              for (int k = 0; k <= 1; k++) {
                static const coord a = {0.,0.,0.}, b = {.5,.5,.5};
                coord nc;
                nc.x = 0., nc.y = (2.*j - 1.)*n.y, nc.z = (2.*k -
                 ↪  1.)*n.z;
                fine(fs.x,2*i,j,k) =
                  rectangle_fraction (nc, alpha - n.x*(2.*i - 1.)/2., a,
                   ↪  b);
              }
#endif // dimension == 3
          }
        }
```

Confirmation of Empty Cell

Ensure the face fraction is 0 for empty cells.

```
        for (int j = 0; j <= 1; j++)
        #if dimension > 2
          for (int k = 0; k <= 1; k++)
        #endif
            if (fine(fs.x,2*i,j,k) && !fine(cs,i,j,k))
              fine(fs.x,2*i,j,k) = 0.;
      }
  }
}
```