vof.h Documentation

Haochen (Langford) Huang

August 20, 2025

Version: 1.0 Updated: 2025-06-07

Section 1

Introduction

`vof.h` aims to solve advection equation

$$\frac{\partial c_i}{\partial t} + \mathbf{u} \cdot \nabla c_i = 0 \tag{1}$$

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]

$$\frac{\partial t_{i,j}}{\partial t} + \mathbf{u} \cdot \nabla t_{i,j} = 0 \tag{2}$$

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 1 and 2 are carefully discussed in the second part.

Section 2

Preparation

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.

2.1.1 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	$t_{i,j}$

2.1.2 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.

2.1.3 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.;
}
```

2.2 vof_concentration_refine

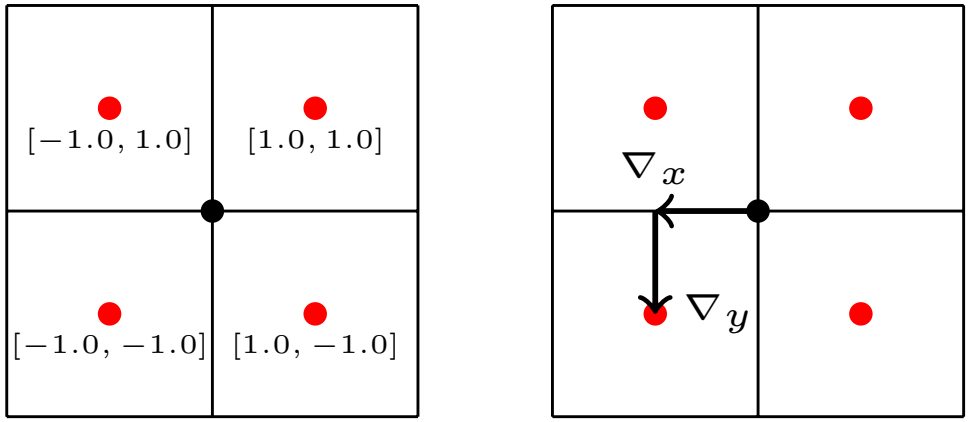
vof_concentration_refine defines the prolongation formula of VOF-concentration t when mesh is refined.

2.2.1 Parameters

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

2.2.2 Worth Mentioning Details

Basilisk employs *child* index to indicate spatial relation between parent and child cells, as displayed in figure 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 $foreach_child$, Basilisk will automatically transversal every child cells and *child* index is assigned with



(a) (b)

Figure 1: (a) Sketch for *child* index. (b) Sketch for volume-fraction-weighted linear interpolation.

corresponding value. As indicated by figure 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

$$t_{child} = c_{child}(\frac{t_{parents}}{c_{parents}} + \frac{\Delta}{4}(child.x\nabla_x t + child.y\nabla_y t)) \tag{3}$$

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.

2.2.3 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.))
    foreach_child()
      s[] = 0.;
```

Prolongation for Tracers

Tracer gradient assign:

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

Implement of equation 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:

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

```
  else {
    coord g;
    foreach_dimension()
      g.x = Delta*vof_concentration_gradient_x (point, f, s);
    double sc = s.inverse ? s[]/(1. - f[]) : s[]/f[], cmc = 4.*cm[];
    foreach_child() {
      s[] = sc;
      foreach_dimension()
        s[] += child.x*g.x*cm[-child.x]/cmc;
      s[] *= s.inverse ? 1. - f[] : f[];
    }
  }
}
```

Section 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.

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 1 is

$$\Delta^d c_i|_n^{n+1} + \int_\Omega \mathbf{u}_f \cdot \nabla c_i = 0 \tag{4}$$

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

$$\Delta^d c_i|_n^{n+1} + \int_{\partial\Omega} \mathbf{u}_f c_i - \int_\Omega c_i \nabla \cdot \mathbf{u}_f = 0 \tag{5}$$

the second term in equation 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 5 now reads

$$c_i^* = c_i^n - \frac{\Delta t}{\Delta}(\mathbf{F}_x[1] - \mathbf{F}_x[]) + \frac{\Delta t}{\Delta}\int_\Omega c_i \nabla_x \cdot \mathbf{u}_f \tag{6}$$

$$c_i^{n+1} = c_i^* - \frac{\Delta t}{\Delta}(\mathbf{F}_y[1] - \mathbf{F}_y[]) + \frac{\Delta t}{\Delta}\int_\Omega c_i \nabla_y \cdot \mathbf{u}_f \tag{7}$$

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 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 2b), reconstruction is applied between advection of every direction as indicated from shape changes between two figures.

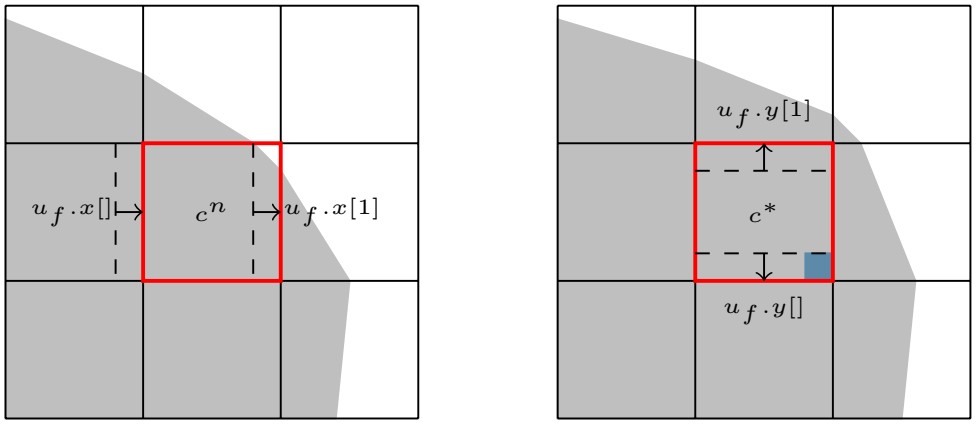
Yet there is still one constraint to satisfy: the volume fraction should be $0 \leq c_i \leq 1$ at any stage. However without additional care, such rule can be violated between direction switch. Take condition in figure 2a as an example. Assume $c^n = 0.9, u_f.x[] = 0.3, u_f.x[1] = 0.1$, for sake of simplicity the time step and cell width are set to be unity. Ignoring the third term in equation 6, $c^* = 1.1 > 1.0$ 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 5) is therefore introduced to cope with such issue[3].

3.1.1 The dilation term

Given the non-divergence of $\mathbf{u}_f$, the dilation term in equation 5 can be rewrite in discrete form as:

$$\int_\Omega c_i \nabla \cdot \mathbf{u}_f = g\Delta^{d-1} \sum_d (u_f.d[1] - u_f.d[]) \tag{8}$$

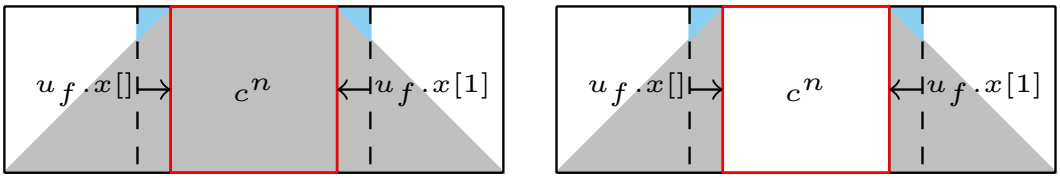
where g is an arbitrary value and can even vary in different cells or in different cycle of iteration. The main topic in this section is to find a proper form of g which can help to overcome the overfull issue depicted previously.



(a) (b)

Figure 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 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 and 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 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 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).

3.1.2 Coefficient g and CFL number

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

$$g = c_c = \begin{cases} 1 & c^n \geq 0.5 \\ 0 & c^n < 0.5 \end{cases} \tag{9}$$

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 9 the restriction for volume fraction reads

$$c \geq 0.5 \geq min(\frac{\Delta t}{\Delta}u_f[1], f) + \frac{\Delta t}{\Delta}(u_f[] - u_f[1]);\ 1 - c \geq 0.5 \geq \frac{\Delta t}{\Delta}u_f[] - max(0, \frac{\Delta t}{\Delta}u_f[1] - 1 + c) \tag{10}$$

$$c \geq 0.5 \geq \frac{\Delta t}{\Delta}(u_f[] - u_f[1]);\ 1 - c \geq 0.5 \geq \frac{\Delta t}{\Delta}(u_f[] - u_f[1]) \tag{11}$$

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.

3.2 sweep_x

sweep_x aims to achieve advection equation 6 in each direction. Notably, similar method is also applied to the advection of tracer in which the dilation term is added.

3.2.1 Parameters

Name	Data type	Status	Option	Representation (before/after)
c	scalar	**updated**	compulsory	c^n/c^*
cc	scalar	unchanged	compulsory	c_c in equation 9
tcl	scalar	unchanged	compulsory	tracer coefficient t_c

3.2.2 Worth Mentioning Details

The dilation coefficient for tracer advection is defined as

$$t_c = \begin{cases} t_{ij}/c_i & c^n \geq 0.5 \\ t_{ij}/c_i & c^n < 0.5 \end{cases} \tag{12}$$

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.

$$\mathbf{F}_t[] = u_f[] \cdot c_f[](\frac{t[]}{c^n[]} + \frac{\Delta}{2}[s_d - \frac{\Delta t}{\Delta}u_f[]]\frac{\partial t[]}{\partial x}) \tag{13}$$

Where the gradient is compute by interface biased scheme introduced in section 2.1 and s_d is the upwind coefficient (see 'bcg.h documentation' for more). c_f as demonstrated by figure 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

$$t_d^* = t^n - \frac{\Delta t}{\Delta}(\mathbf{F}_{t,d}[1] - \mathbf{F}_{t,d}[]) + \frac{t_c \Delta t}{\Delta}(u_{f,d}[1] - u_{f,d}[]) \tag{14}$$

The function is replicated into **sweep_y** and **sweep_z** by macro $foreach\_dimension()$.

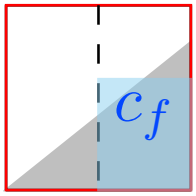


Figure 4: The sketch of partial fraction c_f.

3.2.3 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[]* × *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 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 3.1.2.

```
    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 with dilation term.
Tracer update:
Implementation of equation 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);
}
```

3.3 vof_advection

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

3.3.1 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

3.3.2 Worth Mentioning Details

Direction switch is implemented by counting the time step i.

$$D = (i + d)\ mod\ 3 \quad d = 0, 1, 2 \cdots \tag{15}$$

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

3.3.3 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 9 and 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);
```

Section 4

Draft

If all the volume fraction is known, flux **F** on each face is

$$min(u_f[], c[-1]) \geq F[] \geq max(0, u_f[] - \bar{c}[-1]) \quad u_f[] \geq 0 \tag{16}$$

$$-max(0, -u_f[] - \bar{c}[]) \geq F[] \geq -min(-u_f[], c[]) \quad u_f[] < 0 \tag{17}$$

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

$$min(u_f[], c[-1]) - max(0, u_f[1] - \bar{c}[]) \geq \Delta F[] \geq max(0, u_f[] - \bar{c}[-1]) - min(u_f[1], c[]) \quad u_f[] > 0 \tag{18}$$

$$min(u_f[], c[-1]) + min(-u_f[1], c[1]) \geq \Delta F[] \geq max(0, u_f[] - \bar{c}[-1]) + max(0, -u_f[1] - \bar{c}[1]) \quad u_f[] > 0 \tag{19}$$

$$-max(0, -u_f[] - \bar{c}[]) - max(0, u_f[1] - \bar{c}[]) \geq \Delta F[] \geq -min(-u_f[], c[]) - min(u_f[1], c[]) \quad u_f[] < 0 \tag{20}$$

$$-max(0, -u_f[] - \bar{c}[]) + min(-u_f[1], c[1]) \geq \Delta F[] \geq -min(-u_f[], c[]) + max(0, -u_f[1] - \bar{c}[1]) \quad u_f[] < 0 \tag{21}$$

References

[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.