vof.h documentation

Haochen(Langford) Huang

June 11, 2024

version:draft

1 Introduction

2 Function

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	∇t
point	Point	unchanged	complusory	data index $[i, j]$
c	scalar	unchanged	complusory	volume fraction c
s	scalar	unchange	complusory	t

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

1. Starting Point

A. input:

point: index information, $c = c$, $t = t$

B. adjacent value assignment:

$cl = c[-1]$, $cc = c[]$, $cr = c[1]$

C. 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;
```

2. Gradient Compute
A. local value check:
If *cc* < 0.5, return 0
otherwise checking *cr*.
B. 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.;
}
```

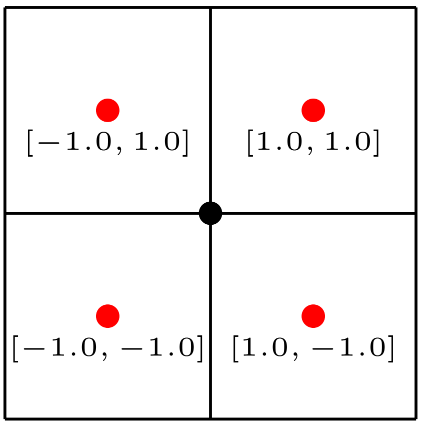
2.2 vof_concentration_refine

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

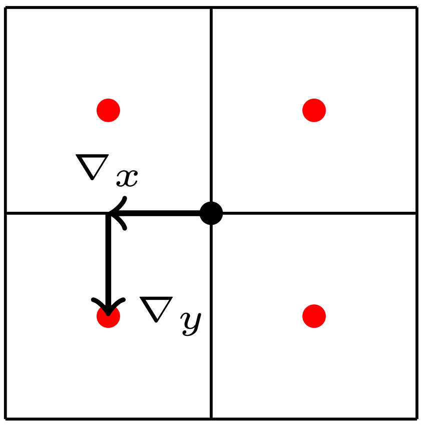
2.2.1 Parameters

Name	Data type	Status	Option	Representation (before/after)
point	Point	unchanged	complusory	data index $[i, j]$
s	scalar	unchange	complusory	t

2.2.2 Worth Mentioning Details



(a)



(b)

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

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 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

$$t_{child} = c_{child}(\frac{t_{parents}}{c_{parents}} + \frac{\Delta}{4}(child.x\nabla_x t + child.y\nabla_y t)) \tag{1}$$

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

1. Starting Point
A. input:
point: index information
$s = t,\ f = s.c = c$
B. 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.;
```

2. Prolongation for Tracers
A. tracer gradient assign:
$g.d = \Delta \cdot \nabla_d t,\ d = x, y(, z)$
B. implement of equation 1:
a. first term of R.H.S.
$sc = s[] = \frac{t_{parents}}{c_{parents}}$
b. rest terms of R.H.S.
$s[] = s[] + \frac{\Delta}{4}(child.d \cdot g.d),\ d = x, y(, z)$
c. 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[];
    }
  }
}
```

3 Readme

Normally, a documentation would consist of two major parts: Introduction & Backround and Function. The first part will introduce the purpose of the corresponding program and the governing equations it solved and other thing developers and users should be aware of *e.g.* in which method the program solve the overall problem. Pragmatic program will be explored line by line in the second part. It first contains a table to clearify all the parameters and their physical representatives as shown in Table.1. The highlighted row in the table indicates such paramter is either the output or has been updated. Second

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

Table 1: Referenc table of parameters.

subsection always concerns with detals and specific technique the function employed. Finally the third part is the workflow of the program.

Throughout documentation font *para* represents exact name of parameters and **function** represents exact name of the function.

Tikz inside text example: 🟫.

4 Program Workflow Example

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

```
void tracer_fluxes (scalar f,
                    face vector uf,
                    face vector flux,
                    double dt,
                    (const) scalar src)
{
  vector g[];
  gradients ({f}, {g});
```