

center.h 文档说明

Haochen Huang

西安交通大学 MFM 课题组

版本：3.02test

日期：2023 年 3 月 22 日

目录

1 理论背景	1
1.1 文件目的	2
1.2 理论背景综述	2
2 源代码解析	3
2.1 代码构成及结构简介	3
2.2 基础代码引用，边界条件设置，初始条件设置	3
2.3 对流项处理及对流方程计算	9
2.4 扩散项处理	11
2.5 速度修正项	13
3 附录 A：Basilisk 中 event 的执行顺序与特殊数据结构的赋值	15
3.1 event 执行顺序	16
3.2 特殊数据结构的赋值	17
4 附录 B：标准 Fractional Step Method 算法及误差	17
参考文献	17

摘要

本文为 basilisk 的头文件 center.h 的说明文档，在阅读时请结合 poisson.h、bcg.h、viscosity.h 等说明头文件。

2.02 版本更新，全面更正之前错误，添加附录解释加速度项更新及 Basilisk 执行规则，添加程序示意图，更新理论部分，更新程序注释。

3.02 版本更新确认了 Basilisk 中 event 的继承顺序，修改部分笔误

1. 理论背景

1.1 文件目的

本文件的目的是求解不可压 NS 方程，文件相关离散格式可以在相关文件中找到 [3][2]:

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u} \otimes \mathbf{u}) = \frac{1}{\rho} [-\nabla p + \nabla \cdot (2\mu \mathbf{D})] + \mathbf{a} \quad (1)$$

$$\nabla \mathbf{u} = 0 \quad (2)$$

1.2 理论背景综述

centered.h 大体上使用 Fractional Step Method 方法对不可压 NS 方程进行求解 [1] (注意是大体上，因为其并没有严格按照 FSM 进行求解器构建，但确实借用其思想)

首先对原方程1进行离散有:

$$\rho_{n+\frac{1}{2}} \frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} + \nabla \cdot (\mathbf{u}^{n+\frac{1}{2}} \otimes \mathbf{u}^{n+\frac{1}{2}}) = \nabla \cdot [\mu^{n+\frac{1}{2}} (\mathbf{D}^{n+1} + \mathbf{D}^n)] + [\mathbf{a}^{n+\frac{1}{2}} - \nabla p^{n+\frac{1}{2}}] \quad (3)$$

其中 $\mathbf{u}^{n+1}, \mathbf{u}^n$ 满足无散条件。

在式3中我们已知 $\mathbf{u}^n, \nabla p^{n-\frac{1}{2}}$ 而需要求解 $\mathbf{u}^{n+1}, \nabla p^{n+\frac{1}{2}}$ 。

对上式方程进行更改后得到:

$$\rho_{n+\frac{1}{2}} \frac{\mathbf{u}^* - \mathbf{u}^n}{\Delta t} + \nabla \cdot (\mathbf{u}^{n+\frac{1}{2}} \otimes \mathbf{u}^{n+\frac{1}{2}}) = \nabla \cdot [2\mu^{n+\frac{1}{2}} \mathbf{D}^*] + [\mathbf{a}^{n-\frac{1}{2}} - \nabla p^{n-\frac{1}{2}}] \quad (4)$$

注意 \mathbf{u}^* 并不需要满足无散条件，而此时方程中的加速项与压力梯度项均变为 $n - \frac{1}{2}$ 时层值。我们称 \mathbf{u}^* 为速度预测步，用式3减去4，忽略扩散项的差异可以得到:

$$\mathbf{u}^{n+1} = \mathbf{u}^* + \frac{\Delta t}{\rho_{n+\frac{1}{2}}} (\nabla p^{n-\frac{1}{2}} - \nabla p^{n+\frac{1}{2}} - \mathbf{a}^{n-\frac{1}{2}} + \mathbf{a}^{n+\frac{1}{2}}) \quad (5)$$

可以看到预测步与 $n + 1$ 时层速度之间修补加速度项后满足投影法；两式中扩散项之差作为改方法在时间上的误差记 ϵ 有

$$\epsilon = (\Delta t)^2 \mathbf{L} \left(\frac{1}{\rho} \mathbf{G} \left(\frac{\partial p}{\partial t} \right) \right) + \frac{1}{2} \Delta t \mathbf{L} \left(\frac{\partial \mathbf{u}}{\partial t} \right) \quad (6)$$

\mathbf{L}, \mathbf{G} 分别是拉普拉斯与梯度算子。

接下来依顺序求解方程4

首先启用 bcg.h 处理对流项，bcg.h 得目的为求解方程:

$$\mathbf{u}^{**} = \mathbf{u}^n - \Delta t \mathbf{A} \quad (7)$$

其中 \mathbf{A} 就是 $\nabla \cdot (\mathbf{u}^{n+\frac{1}{2}} \otimes \mathbf{u}^{n+\frac{1}{2}})$ 的 bcg 离散格式，由此我们就可以将对流项加入到非定常项中，4变为:

$$\rho_{n+\frac{1}{2}} \frac{\mathbf{u}^* - \mathbf{u}^{**}}{\Delta t} = \nabla \cdot [2\mu^{n+\frac{1}{2}} \mathbf{D}^*] + [\mathbf{a}^{n-\frac{1}{2}} - \nabla p^{n-\frac{1}{2}}] \quad (8)$$

再将非扩散项的源项汇入非定常项，最后得到

$$\rho_{n+\frac{1}{2}} \frac{\mathbf{u}^* - \mathbf{u}^{***}}{\Delta t} = \nabla \cdot [2\mu^{n+\frac{1}{2}} \mathbf{D}^*] \quad (9)$$

其中

$$\mathbf{u}^{***} = \mathbf{u}^{**} + \frac{\Delta t}{\rho_{n+\frac{1}{2}}} [\mathbf{a}^{n-\frac{1}{2}} - \nabla p^{n-\frac{1}{2}}] \quad (10)$$

由 viscosity.h 求解该方程，得到 \mathbf{u}^* 。由方程5我们填补加速项与已知压力梯度项 $\nabla p^{n-\frac{1}{2}}$ 可以得到

$$\mathbf{u}_{\text{new}}^* = \mathbf{u}^* - \frac{\Delta t}{\rho_{n+\frac{1}{2}}} (\nabla p^{n-\frac{1}{2}} - \mathbf{a}^{n-\frac{1}{2}} + \mathbf{a}^{n+\frac{1}{2}}) = \mathbf{u}^{n+1} + \nabla p^{n+\frac{1}{2}} \quad (11)$$

$$\nabla \mathbf{u}_{\text{new}}^* = \Delta p^{n+\frac{1}{2}} \quad (12)$$

2.1 代码构成及结构简介

1. 初始边界设置, 初始条件设置2.2
2. 对流项处理2.3
3. 扩散项处理2.4
4. 加速度更新2.5
5. 速度投影与压力梯度更新

```
graph TD
    centered(centered.h) --> advection(event advection_term)
    centered --> viscous(event viscous_term)
    centered --> acceleration(event acceleration)
    centered --> projection(event projection)
    poisson(poisson.h) -- "poisson方程求解, 速度无散化" --> projection
    bcg(bcg.h) -- "对流方程求解" --> advection
    bcg -- "对流项与非定常项合并" --> viscous
    viscosity(viscosity.h) -- "源项与非定常项合并" --> viscous
    viscosity -- "求解扩散方程" --> viscous
    acceleration -- "更新加速项" --> acceleration
    acceleration -- "消除n-1/2时层加速度与压力梯度项" --> acceleration
    projection -- "poisson方程求解, 速度无散化" --> poisson
    projection -- "更新速度场压力梯度场, 输出" --> projection
    main[主程序] -- "调用" --> centered
    main -- "调用" --> accel_event(event acceleration)
    accel_event -- "调用" --> acceleration
    accel_event -- "调用" --> projection
```

2.2 基础代码引用，边界条件设置，初始条件设置

```
1 #include "run.h"
2 #include "timestep.h"
3 #include "bcg.h"//注释：对流方程迭代计算
4 #if EMBED
5 # include "viscosity-embed.h"
6 #else
```

```

7  # include "viscosity.h"//注释: 扩散方程迭代计算
8  #endif
9  /**
10 The primary variables are the centered pressure field p and the
11 centered velocity field u. The centered vector field
12 g will contain pressure gradients and acceleration terms.
13
14 We will also need an auxilliary face velocity field uf and
15 the associated centered pressure field pf. */
16
17 scalar p[];
18 vector u[], g[];
19 scalar pf[];
20 face vector uf[];
21
22 /**
23 In the case of variable density, the user will need to define both the
24 face and centered specific volume fields ( $\alpha$  and  $\alpha_c$ 
25 respectively) i.e.  $1/\rho$ . If not specified by the user, these
26 fields are set to one i.e. the density is unity.
27
28 Viscosity is set by defining the face dynamic viscosity  $\mu$ ; default
29 is zero.
30
31 The face field a defines the acceleration term; default is
32 zero.
33
34 The statistics for the (multigrid) solution of the pressure Poisson
35 problems and implicit viscosity are stored in *mgp*, *mgpf*, *mgu*
36 respectively.
37
38 If *stokes* is set to *true*, the velocity advection term
39  $\nabla \cdot (\mathbf{u} \otimes \mathbf{u})$  is omitted. This is a
40 reference to [Stokes flows](http://en.wikipedia.org/wiki/Stokes\_flow)
41 for which inertia is negligible compared to viscosity. */
42
43 (const) face vector mu = zerof, a = zerof, alpha = unityf;

```

```

44 (const) scalar rho = unity;
45 mgstats mgp, mgpf, mgu;
46 bool stokes = false;
47
48 /**
49  ## Boundary conditions
50
51  For the default symmetric boundary conditions, we need to ensure that
52  the normal component of the velocity is zero after projection. This
53  means that, at the boundary, the acceleration a must be
54  balanced by the pressure gradient. Taking care of boundary orientation
55  and staggering of a, this can be written */
56  //说明: 如 event acceleration 所示, uf 中包含了 a, 因此需要在对  $u_f^{n+1}$  校正时,
57  ↪ 需要对边界上的  $u_f$  减去 a 的值 (令  $\Delta p=a$ ) 以满足  $u_f=0$ 
58
59  #if EMBED
60  # define neumann_pressure(i) (alpha.n[i] ? a.n[i]*fm.n[i]/alpha.n[i] : ^I\
61            a.n[i]*rho[]/(cm[] + SEPS))
62  #else
63  # define neumann_pressure(i) (a.n[i]*fm.n[i]/alpha.n[i])
64  #endif
65
66  p[right] = neumann (neumann_pressure(ghost));
67  p[left]  = neumann (- neumann_pressure(0));
68
69  #if AXI
70  uf.n[bottom] = 0.;
71  uf.t[bottom] = dirichlet(0); // since uf is multiplied by the metric which
72  // is zero on the axis of symmetry
73  p[top]      = neumann (neumann_pressure(ghost));
74  #else // !AXI
75  # if dimension > 1
76  p[top]      = neumann (neumann_pressure(ghost));
77  p[bottom]   = neumann (- neumann_pressure(0));
78  # endif
79  # if dimension > 2
80  p[front]    = neumann (neumann_pressure(ghost));

```

```

80 p[back] = neumann (- neumann_pressure(0));
81 # endif
82 #endif // !AXI
83
84 /**
85  For [embedded boundaries on trees](/src/embed-tree.h), we need to
86  define the pressure gradient for prolongation of pressure close to
87  embedded boundaries. */
88
89 #if TREE && EMBED
90 void pressure_embed_gradient (Point point, scalar p, coord * g)
91 {
92     foreach_dimension()
93         g->x = rho[]/(cm[] + SEPS)*(a.x[] + a.x[1])/2.;
94 }
95 #endif // TREE && EMBED
96
97 /**
98  ## Initial conditions */
99
100 event defaults (i = 0)
101 {
102
103     CFL = 0.8; //注释：影响时间步长的选取
104
105     /**
106      The pressures are never dumped. */
107
108     p.nodump = pf.nodump = true;
109
110     /**
111      The default density field is set to unity (times the metric). */
112
113     if (alpha.x.i == unityf.x.i) {
114         alpha = fm;
115         rho = cm;
116     }

```

```

117     else if (!is_constant(alpha.x)) {
118         face vector alphav = alpha;
119         foreach_face()
120             alphav.x[] = fm.x[];
121     }
122
123     /**
124     On trees, refinement of the face-centered velocity field needs to
125     preserve the divergence-free condition. */
126 //说明：对不同层级网格中变量和 embed 边界网格的插值，由细到粗或由粗到细
127
128 #if TREE
129     uf.x.refine = refine_face_solenoidal;
130
131     /**
132     When using [embedded boundaries](/src/embed.h), the restriction and
133     prolongation operators need to take the boundary into account. */
134
135 #if EMBED
136     uf.x.refine = refine_face;
137     foreach_dimension()
138         uf.x.prolongation = refine_embed_face_x;
139     for (scalar s in {p, pf, u, g}) {
140         s.restriction = restriction_embed_linear;
141         s.refine = s.prolongation = refine_embed_linear;
142         s.depends = list_add (s.depends, cs);
143     }
144     for (scalar s in {p, pf})
145         s.embed_gradient = pressure_embed_gradient;
146 #endif // EMBED
147 #endif // TREE
148 }
149
150
151 /**
152 We had some objects to display by default. */
153

```

```

154 event default_display (i = 0)
155     display ("squares (color = 'u.x', spread = -1);");
156
157 /**
158  After user initialisation, we initialise the face velocity and fluid
159  properties. */
160
161 double dtmax;
162
163 event init (i = 0)
164 {
165     trash ({uf});
166     foreach_face()
167         uf.x[] = fm.x[]*face_value (u.x, 0);
168
169     /**
170      We update fluid properties. */
171
172     event ("properties");
173
174     /**
175      We set the initial timestep (this is useful only when restoring from
176      a previous run). */
177
178     dtmax = DT;
179     event ("stability");
180 }
181
182 /**
183  ## Time integration
184
185  The timestep for this iteration is controlled by the CFL condition,
186  applied to the face centered velocity field  $u_f$ ; and the
187  timing of upcoming events. */
188
189 event set_dtmax (i++,last) dtmax = DT;
190

```



```

191 event stability (i++,last) {
192     dt = dtnext (stokes ? dtmax : timestep (uf, dtmax)); //根据限制条件设置最大时
        ↪ 间步
193 }
194
195 /**
196  If we are using VOF or diffuse tracers, we need to advance them (to
197  time  $t + \Delta t/2$ ) here. Note that this assumes that tracer fields
198  are defined at time  $t - \Delta t/2$  i.e. are lagging the
199  velocity/pressure fields by half a timestep. */
200  //说明: 为了与  $u$  形成时间上的交错, 初始各参数均被假定在  $-\Delta t/2$  时刻, ( $p$  应该也
        ↪ 应是被假定在  $-\Delta t/2$  时刻?)
201
202 event vof (i++,last);
203 event tracer_advection (i++,last);
204 event tracer_diffusion (i++,last);
205
206 /**
207  The fluid properties such as specific volume (fields  $\alpha$  and
208   $\alpha_c$ ) or dynamic viscosity (face field  $\mu_f$ ) -- at time
209   $t + \Delta t/2$  -- can be defined by overloading this event. */
210
211 event properties (i++,last);

```

2.3 对流项处理及对流方程计算

在本小节中我们对对流项进行处理, 用已知 \mathbf{u}^n 构建位于单元面上的面元速度 $\mathbf{u}_f^{n+\frac{1}{2}}$, 具体的推导公式在 bcg.h 中详细阐述, 后使用在 poisson.h 文件中构建的 projection 函数对所求得 $\mathbf{u}_f^{n+\frac{1}{2}}$ 进行无散化, 再带入 advection 函数中求解方程7, 从而与 \mathbf{u}^n 一同化为 \mathbf{u}^{**}

```

1 void prediction()
2 {
3     vector du;
4     foreach_dimension() {
5         scalar s = new scalar;
6         du.x = s;
7     }
8

```

```

9   if (u.x.gradient)
10   foreach()
11     foreach_dimension() {
12   #if EMBED
13       if (!fs.x[] || !fs.x[1])
14       du.x[] = 0.;
15   else
16   #endif
17       du.x[] = u.x.gradient (u.x[-1], u.x[], u.x[1])/Delta; //gradient 是在
        ↳ common.h 中保存的每个 scalar 都具有的数据结构特别类型
18   }
19   else
20   foreach()
21     foreach_dimension() {
22   #if EMBED
23       if (!fs.x[] || !fs.x[1])
24       du.x[] = 0.;
25   else
26   #endif
27       du.x[] = (u.x[1] - u.x[-1])/(2.*Delta); //其实就是求该方向上的梯度
28   }
29
30   trash ({uf});
31   foreach_face() {
32       double un = dt*(u.x[] + u.x[-1])/(2.*Delta), s = sign(un);
33       int i = -(s + 1.)/2.;
34       uf.x[] = u.x[i] + (g.x[] + g.x[-1])*dt/4. + s*(1. -
        ↳ s*un)*du.x[i]*Delta/2.;
35   #if dimension > 1
36       if (fm.y[i,0] && fm.y[i,1]) {
37           double fyy = u.y[i] < 0. ? u.x[i,1] - u.x[i] : u.x[i] - u.x[i,-1];
38           uf.x[] -= dt*u.y[i]*fyy/(2.*Delta);
39       }
40   #endif
41   #if dimension > 2
42       if (fm.z[i,0,0] && fm.z[i,0,1]) {
43           double fzz = u.z[i] < 0. ? u.x[i,0,1] - u.x[i] : u.x[i] - u.x[i,0,-1];

```

```

44     uf.x[] -= dt*u.z[i]*fzz/(2.*Delta);
45 }
46 #endif
47 uf.x[] *= fm.x[];
48 }
49
50 delete ((scalar *){du});
51 }
52
53 /**
54  Advection term
55
56  We predict the face velocity field  $u_f$  at time  $t + \Delta t/2$  then project it to make
57   $\hookrightarrow$  it divergence-free. We can then use it to
58  compute the velocity advection term, using the standard
59  Bell-Collela-Glaz advection scheme for each component of the velocity
60  field. */
61 event advection_term (i++,last)
62 {
63     if (!stokes) {
64         prediction(); //注释: 预测步,  $u_f$  位于  $n+1/2$  时层
65         mgpf = project (uf, pf, alpha, dt/2., mgpf.nrelax); //注释:  $u_f$  无散化,  $pf$ 
66          $\hookrightarrow$  位于  $n+1/2$  时层
67         advection ((scalar *){u}, uf, dt, (scalar *){g}); //注释: 对流方程计算得到
68          $\hookrightarrow u^{**}$ 
69     }
70 }

```

2.4 扩散项处理

在得到 u^{**} 后我们将 $\nabla p^{n-\frac{1}{2}}$ 以及加速度表面张力项等汇入 u^{**} 见公式10

```

1  /**
2  ### Viscous term
3
4  We first define a function which adds the pressure gradient and
5  acceleration terms. */

```

```

6
7 static void correction (double dt)
8 {
9     foreach()
10         foreach_dimension()
11             u.x[] += dt*g.x[];
12 }
13
14 /**
15  The viscous term is computed implicitly. We first add the pressure
16  gradient and acceleration terms, as computed at time t, then call
17  the implicit viscosity solver. We then remove the acceleration and
18  pressure gradient terms as they will be replaced by their values at
19  time t + Δt. */
20
21 event viscous_term (i++,last)
22 {
23     if (constant(mu.x) != 0.) {
24         correction (dt); //构造 poisson 型方程的残差, 直接构造 u***
25         mgu = viscosity (u, mu, rho, dt, mgu.nrelax);
26         correction (-dt); //注意此时由于 viscosity 的计算 u[] 已经存储的是 u*, 让其
           ↪ 减去位于  $n - \frac{1}{2}$  的加速度项与压力梯度项
27     }
28
29     /**
30      We reset the acceleration field (if it is not a constant). */
31
32     if (!is_constant(a.x)) {
33         face vector af = a;
34         trash ({af});
35         foreach_face()
36             af.x[] = 0.; //刷新加速度项
37     }
38 }

```

通过操作 `mgu = viscosity (u, mu, rho, dt, mgu.nrelax);` 解方程9。

由此我们得解 \mathbf{u}^* 并在 `correction (-dt);` 中对其做 $\mathbf{u}^* + \frac{\Delta t}{\rho_{n+\frac{1}{2}}}(\nabla p^{n-\frac{1}{2}} - \mathbf{a}^{n-\frac{1}{2}})$

2.5 速度修正项

由于加速项为面元，而求得的 \mathbf{u}^* 为体元项，为了 poisson 方程求解方便，我们将所得速度项向单元面上插值，并更新补充 $n + \frac{1}{2}$ 的加速度项（如方程11，关于加速度项更新详见第3节附录），则此时 $\mathbf{u}_{\mathbf{f}}^*_{\text{new}}$ 满足

$$\mathbf{u}_{\mathbf{f}}^{n+1} = \mathbf{u}_{\mathbf{f}}^*_{\text{new}} - \frac{\Delta t}{\rho_{n+\frac{1}{2}}} \nabla p_{n+\frac{1}{2}} \quad (13)$$

即带入 poisson 求解器再进行求解，同时更新压力项 $p^{n+\frac{1}{2}}$

```
1  /**
2  ### Acceleration term
3
4  The acceleration term a needs careful treatment as many
5  equilibrium solutions depend on exact balance between the acceleration
6  term and the pressure gradient: for example Laplace's balance for
7  surface tension or hydrostatic pressure in the presence of gravity.
8
9  To ensure a consistent discretisation, the acceleration term is
10 defined on faces as are pressure gradients and the centered combined
11 acceleration and pressure gradient term g is obtained by
12 averaging.
13
14 The (provisionary) face velocity field at time  $t + \Delta t$  is
15 obtained by interpolation from the centered velocity field. The
16 acceleration term is added. */
17 //说明：基于 balance-force 方法，压力和表面张力两项在程序各方程中同时考虑。
18
19 event acceleration (i++,last)
20 {
21     trash ({uf});
22     foreach_face()
23         uf.x[] = fm.x[]*(face_value (u.x, 0) + dt*a.x[]); //此处为更新位于  $n + \frac{1}{2}$  时
24         ↪ 层的加速度项，并将其作为补充，填充速度项，从而获得速度预测步与压力梯度的
25         ↪ poisson 方程
26 }
27
28 /**
29 ## Approximate projection
```

```

28
29 This function constructs the centered pressure gradient and
30 acceleration field *g* using the face-centered acceleration field *a*
31 and the cell-centered pressure field *p*. */
32
33 void centered_gradient (scalar p, vector g)
34 {
35
36     /**
37     We first compute a face field  $g_f$  combining both
38     acceleration and pressure gradient. */
39
40     face vector gf[];
41     foreach_face()
42         gf.x[] = fm.x[]*a.x[] - alpha.x[]*(p[] - p[-1])/Delta;
43
44     /**
45     We average these face values to obtain the centered, combined
46     acceleration and pressure gradient field. */
47
48     trash ({g});
49     foreach()
50         foreach_dimension()
51             g.x[] = (gf.x[] + gf.x[1])/(fm.x[] + fm.x[1] + SEPS);
52 }
53
54 /**
55 To get the pressure field at time  $t + \Delta t$  we project the face
56 velocity field (which will also be used for tracer advection at the
57 next timestep). Then compute the centered gradient field *g*. */
58
59 event projection (i++,last)
60 {
61     mgp = project (uf, p, alpha, dt, mgp.nrelax);
62     centered_gradient (p, g);
63
64     /**

```

```

65  We add the gradient field *g* to the centered velocity field. */
66
67  correction (dt); //此处是对网格中心速度场进行源项附加（注意之前加速度操作及压力
    ↳ 梯度更新都是面元速度，单元中心速度自扩散项中去掉  $n - \frac{1}{2}$  时层的加速度与压力梯
    ↳ 度后再未更新）
68 }
69
70 /**
71  Some derived solvers need to hook themselves at the end of the
72  timestep. */
73
74  event end_timestep (i++, last);
75
76  /**
77  ## Adaptivity
78
79  After mesh adaptation fluid properties need to be updated. When using
80  [embedded boundaries](/src/embed.h) the fluid fractions and face
81  fluxes need to be checked for inconsistencies. */
82
83  #if TREE
84  event adapt (i++, last) {
85  #if EMBED
86      fractions_cleanup (cs, fs);
87      foreach_face()
88          if (uf.x[] && !fs.x[])
89              uf.x[] = 0.;
90  #endif
91      event ("properties");
92  }
93  #endif

```

最后将所有数据进行更新，一个时层的 NS 方程即求解完毕。

3. 附录 A：Basilisk 中 event 的执行顺序与特殊数据结构的赋值

centered.h 中加速度项的更新并没有在程序中明示，实际上的更新发生在调用 centered.h 头文件的主程序中，以 Basilisk 官网中的 Bubble rising in a large tank 为例 <http://basilisk.fr/src/examples/bubble.c>，可以发现主程序部分中，有与 centered.h 头文件中同名的 event acceleration：

```

1 event acceleration (i++) {
2     face vector av = a;
3     foreach_face(y)
4         av.y[] -= 1.;
5 }

```

此处也正是整体算法中，加速度更新的操作；下面从两个方向分析该代码

3.1 event 执行顺序

基于官网 *BasiliskC* <http://basilisk.fr/Basilisk%20C#event-inheritance> 中的相关介绍以及代码实验，现对 Basilisk 中同名 event 的继承执行顺序进行概述

1. 程序进行编译运行时所有的头文件将会被直接复制打开,所有同名的 event 视为一组 (group) 将会被同时执行,其相较于其他名称 event 被执行顺序取决于该组 event 中最先出现的那个位置
2. 组内 event 的执行顺序和出现顺序,即最先执行在源程序中出现最晚的 event

为了测试 event 执行顺序编写以下主程序及头文件

主程序示例：

```

1 #include "eventttest1.h"
2 // #include "eventttest2.h"
3 #include "run.h"
4
5 #define MAXTIME 1
6
7
8
9 int main()
10 {
11     run();
12 }
13
14 event scriptA(i=0; i<=MAXTIME; i++)
15 {
16     fprintf(stdout,"script1\n");
17 }
18
19 event scriptB(i=0; i<=MAXTIME; i++)

```



```

20 {
21     fprintf(stdout, "script2\n");
22 }

```

头文件示例:

```

1 event test1A(i++,last)
2 {
3     fprintf(stdout, "test1A\n");
4 }
5
6 event scriptA(i++,last)
7 {
8     fprintf(stdout, "test1B\n");
9 }

```

由上述规则可知，在运行 Bubble.c 时，主程序中名为 acceleration 的 event 会率先被忽略，而在头文件中同名 event 执行时被唤醒，并首先执行主程序 acceleration 编写内容，再执行头文件中的 acceleration。

3.2 特殊数据结构的赋值

在主程序的 acceleration 中有操作 `face vector av = a;`，这里并不是赋值操作，而是指针地址传递，其中 `av, a` 均为 `face vector*` 故我们在之后对 `av` 进行的任何操作，其实质上都是在对 `a` 进行各种同等操作。

类似的，所有 Basilisk 自带的数据结构，直接对名称使用等于号，都是对内存地址进行传递，而并非赋值。

4. 附录 B：标准 Fractional Step Method 算法及误差

在前文中提到，centered.h 大体上运用了 FSM 方法，而其不同之处便在于扩散项的离散，标准方法的离散为：

$$\rho_{n+\frac{1}{2}} \frac{\mathbf{u}^* - \mathbf{u}^{***}}{\Delta t} = \nabla \cdot [\mu^{n+\frac{1}{2}} (\mathbf{D}^* + \mathbf{D}^n)] \quad (14)$$

而误差为：

$$\epsilon = \frac{1}{2} (\Delta t)^2 \mathbf{L} \left(\frac{1}{\rho} \mathbf{G} \left(\frac{\partial p}{\partial t} \right) \right) \quad (15)$$

参考文献

- [1] John Kim and Parviz Moin. “Application of a fractional-step method to incompressible Navier-Stokes equations”. In: *Journal of computational physics* 59.2 (1985), pp. 308–323.

- [2] Stéphane Popinet. “An accurate adaptive solver for surface-tension-driven interfacial flows”. en. In: *Journal of Computational Physics* 228.16 (Sept. 2009), pp. 5838–5866. issn: 00219991. doi: [10 . 1016 / j . jcp . 2009 . 04 . 042](https://doi.org/10.1016/j.jcp.2009.04.042). url: <https://linkinghub.elsevier.com/retrieve/pii/S002199910900240X> (visited on 07/05/2022).
- [3] Stéphane Popinet. “Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries”. In: *Journal of computational physics* 190.2 (2003), pp. 572–600.