centered.h 文档说明

Haochen Huang

版本：4.02test

日期：2023 年 8 月 3 日

目录

1 理论背景 2

1.1 文件目的 . . . 2

1.2 理论求解综述 . . . 2

1.3 APM 与 EPM . . . 3

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 . . . 10

2.4 扩散项处理 . . . 12

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 16

3.1 event 执行顺序 . . . 16

3.2 特殊数据结构的赋值 . . . 18

4 附录 B：标准 Fractional Step Method 算法及误差 18

参考文献 18

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

4.02 版本重新确认并更正了此前在理论部分出现的重大错误，并借由此从更深刻的角度重新理解相关求解器

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，文件相关离散格式请见 [5][4]:

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

$$\nabla \cdot \mathbf{u} = 0 \tag{2}$$

1.2 理论求解综述

本求解器使用 $Approximate\,Projection\,Method$(下称 APM) 对不可压缩 NS 方程进行求解:

对原方程1进行离散有:

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

理论上 $\mathbf{u}^{n+1}, \mathbf{u}^n$ 满足无散条件; 式3中已知条件为 $\mathbf{u}^n, \nabla p^n$ 而待解项为 $\mathbf{u}^{n+1}, \nabla p^{n+1}$。

与 MAC 相似 [3]，该方法的第一步都为计算不满足无散条件的中间步 $\mathbf{u}^*$:

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

注意 $\mathbf{u}^*$ 并不需要满足无散条件。此处求解器选择 APM 的 $pressure\,form$[2]，即在相应的投影之前需要对 $\mathbf{u}^*$ 进行相应的处理

$$\mathbf{u}^{n+1} = \mathscr{P}(\mathbf{u}^* + \frac{\Delta t}{\rho^{n+\frac{1}{2}}}(-\mathbf{a}^n + p^n)) \tag{5}$$

其中 $\mathscr{P}$ 代表投影算子；两式中扩散项之差作为改方法在时间上的误差记 ϵ 有

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

$\mathscr{L}, \mathscr{G}$ 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:

$$u^{**} = u^n - \Delta t \mathbf{A} \tag{7}$$

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:

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

再将非扩散项的源项汇入非定常项，最后得到

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

其中

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

由 viscosity.h 求解该方程，得到 $\mathbf{u}^*$。再由方程5填补加速项与已知压力梯度项 ∇p^n 得 u^*_{new}

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

又因为 $\mathbf{u}^{n+1}$ 满足无散条件，即可得到 poisson 方程

$$\nabla \mathbf{u^*_{new}} = \Delta p^{n+1} \tag{12}$$

由此更新速度 $\mathbf{u}^{n+1}$ 以及速度梯度项 ∇p^{n+1}

1.3 APM 与 EPM

如前文所述，本求解器算法为近似投影法 (APM) 而并非标准投影法 (EPM)，二者主要区别在与对拉普拉斯算子 $\mathscr{L}$ 的构造以及最后结果是否满足无散条件。将1改写为 (忽略加速度项)：

$$\frac{\partial \mathbf{u}}{\partial t}+\frac{1}{\rho}\nabla p=\frac{1}{\rho}[\nabla\cdot(2\mu\mathbf{D})]-\nabla\cdot(\mathbf{u}\otimes\mathbf{u}) \tag{13}$$

观察到方程右端为连续向量场，依照定义对其施加投影算子 $\mathscr{P}$ 有

$$\frac{\partial \mathbf{u}}{\partial t}=\mathscr{P}(\frac{1}{\rho}[\nabla\cdot(2\mu\mathbf{D})]-\nabla\cdot(\mathbf{u}\otimes\mathbf{u})) \tag{14}$$

使用分布法实现该投影算子，即求解方程3首先得到 $\mathbf{u}^*$，再通过 $possion$ 求解器更新速度项与压力梯度项，即：

$$\mathscr{D}\mathscr{G}\phi=\mathscr{L}\phi=\mathscr{D}\mathbf{V} \tag{15}$$

其中 $\mathscr{D},\mathscr{G},\mathscr{L}$ 分别代表散度算子，梯度算子，拉普拉斯算子。EPM 与 APM 之间的差别就在于 $\mathscr{L}$ 的构建，对于 EPM，$\mathscr{L}_{EPM}=\mathscr{D}\mathscr{G}$ 即拉普拉斯算子并不单独存在而是由两离散的算子计算合成的，然而这样构成的拉普拉斯算子在实际的操作尤其是自适应网格等复杂网格中很难实现，是故诞生了 APM，即单独定义标准的 $\mathscr{L}$，而往往 $\mathscr{L}_{APM}\neq\mathscr{D}\mathscr{G}$，从而导致计算结果 $\mathbf{u}^{n+1}$ 并不能满足严格的无散条件，而是对标准无散速度的二阶近似，近似投影法 APM 也因此得名。

$\phi,\mathbf{V}$ 则分别代表余项以及相应的无散化矢量，根据不同需要 $\mathbf{V}$ 有不同选择 [1]，在 EPM 中 $\mathbf{V}$ 的选择并不影响最终结果，然而对于 APM，$\mathbf{V}$ 的选择会极大的影响最终的误差积累。本求解器在此处选取的形式为

$$\mathbf{V}=\mathbf{u}^*+\frac{\Delta t}{\rho^{n+\frac{1}{2}}}(\nabla p^n) \tag{16}$$

根据 [1]，此种模式在 APM 中相较其他选择具有最强的稳定性。

2. 源代码解析

2.1 代码构成及结构简介

同上文原理，代码共有五部分：

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

其中最后两部分合并编写，整体求解器示意图如下：

2.2 基础代码引用，边界条件设置，初始条件设置

相关情况设置于注释中并不做太多解释。(embed 边界的相关设置仍待查)

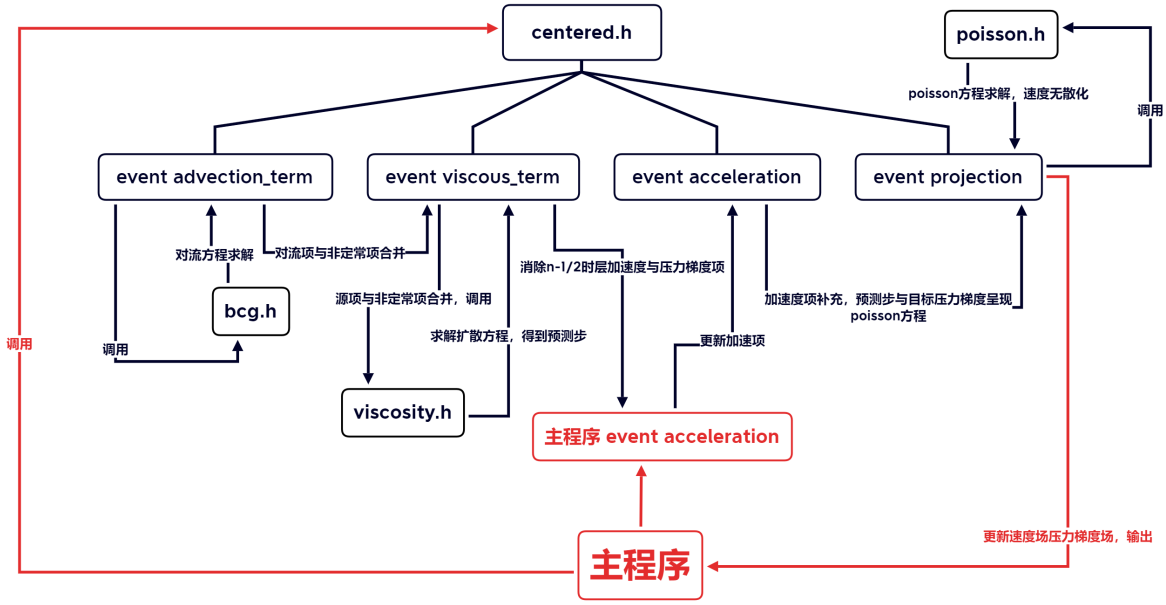

图 1: centered.h 头文件中主要函数关系

```
#include "run.h"
#include "timestep.h"
#include "bcg.h"//注释：对流方程迭代计算
#if EMBED
# include "viscosity-embed.h"
#else
# include "viscosity.h"//注释：扩散方程迭代计算
#endif
/**
The primary variables are the centered pressure field p and the
centered velocity field u. The centered vector field
g will contain pressure gradients and acceleration terms.

We will also need an auxilliary face velocity field u_f and
the associated centered pressure field p_f. */

scalar p[];
vector u[], g[];
scalar pf[];
face vector uf[];

/**
```

```
In the case of variable density, the user will need to define both the
face and centered specific volume fields ($\alpha$ and $\alpha_c$
respectively) i.e. $1/\rho$. If not specified by the user, these
fields are set to one i.e. the density is unity.

Viscosity is set by defining the face dynamic viscosity $\mu$; default
is zero.

The face field $\mathbf{a}$ defines the acceleration term; default is
zero.

The statistics for the (multigrid) solution of the pressure Poisson
problems and implicit viscosity are stored in *mgp*, *mgpf*, *mgu*
respectively.

If *stokes* is set to *true*, the velocity advection term
$\nabla\cdot(\mathbf{u}\otimes\mathbf{u})$ is omitted. This is a
reference to [Stokes flows](http://en.wikipedia.org/wiki/Stokes_flow)
for which inertia is negligible compared to viscosity. */

(const) face vector mu = zerof, a = zerof, alpha = unityf;
(const) scalar rho = unity;
mgstats mgp, mgpf, mgu;
bool stokes = false;

/**
## Boundary conditions

For the default symmetric boundary conditions, we need to ensure that
the normal component of the velocity is zero after projection. This
means that, at the boundary, the acceleration $\mathbf{a}$ must be
balanced by the pressure gradient. Taking care of boundary orientation
and staggering of $\mathbf{a}$, this can be written */
//说明：如 event acceleration 所示，uf 中包含了 a，因此需要在对 uf^{n+1} 校正时，
↪ 需要对边界上的 uf 减去 a 的值（令\Delta p=a）以满足 uf=0

#if EMBED
```

```
# define neumann_pressure(i) (alpha.n[i] ? a.n[i]*fm.n[i]/alpha.n[i] :^^I\
                  a.n[i]*rho[]/(cm[] + SEPS))
#else
# define neumann_pressure(i) (a.n[i]*fm.n[i]/alpha.n[i])
#endif

p[right] = neumann (neumann_pressure(ghost));
p[left]  = neumann (- neumann_pressure(0));

#if AXI
uf.n[bottom] = 0.;
uf.t[bottom] = dirichlet(0); // since uf is multiplied by the metric which
                             // is zero on the axis of symmetry
p[top]    = neumann (neumann_pressure(ghost));
#else // !AXI
#  if dimension > 1
p[top]    = neumann (neumann_pressure(ghost));
p[bottom] = neumann (- neumann_pressure(0));
#  endif
#  if dimension > 2
p[front]  = neumann (neumann_pressure(ghost));
p[back]   = neumann (- neumann_pressure(0));
#  endif
#endif // !AXI

/**
For [embedded boundaries on trees](/src/embed-tree.h), we need to
define the pressure gradient for prolongation of pressure close to
embedded boundaries. */

#if TREE && EMBED
void pressure_embed_gradient (Point point, scalar p, coord * g)
{
  foreach_dimension()
    g->x = rho[]/(cm[] + SEPS)*(a.x[] + a.x[1])/2.;
}
#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
```

```
  prolongation operators need to take the boundary into account. */

#if EMBED
  uf.x.refine = refine_face;
  foreach_dimension()
    uf.x.prolongation = refine_embed_face_x;
  for (scalar s in {p, pf, u, g}) {
    s.restriction = restriction_embed_linear;
    s.refine = s.prolongation = refine_embed_linear;
    s.depends = list_add (s.depends, cs);
  }
  for (scalar s in {p, pf})
    s.embed_gradient = pressure_embed_gradient;
#endif // EMBED
#endif // TREE
}


/**
We had some objects to display by default. */

event default_display (i = 0)
  display ("squares (color = 'u.x', spread = -1);");

/**
After user initialisation, we initialise the face velocity and fluid
properties. */

double dtmax;

event init (i = 0)
{
  trash ({uf});
  foreach_face()
    uf.x[] = fm.x[]*face_value (u.x, 0);

  /**
```

```
  We update fluid properties. */

  event ("properties");

  /**
  We set the initial timestep (this is useful only when restoring from
  a previous run). */

  dtmax = DT;
  event ("stability");
}

/**
## Time integration

The timestep for this iteration is controlled by the CFL condition,
applied to the face centered velocity field $\mathbf{u}_f$; and the
timing of upcoming events. */

event set_dtmax (i++,last) dtmax = DT;

event stability (i++,last) {
  dt = dtnext (stokes ? dtmax : timestep (uf, dtmax));//根据限制条件设置最大时间步
}

/**
If we are using VOF or diffuse tracers, we need to advance them (to
time $t+\Delta t/2$) here. Note that this assumes that tracer fields
are defined at time $t-\Delta t/2$ i.e. are lagging the
velocity/pressure fields by half a timestep. */
//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？)

event vof (i++,last);
event tracer_advection (i++,last);
event tracer_diffusion (i++,last);
```

```
205
206 /**
207 The fluid properties such as specific volume (fields α and
208 α_c) or dynamic viscosity (face field μ_f) -- at time
209 t + Δt/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])
```

```
      du.x[] = 0.;
    else
#endif
      du.x[] = (u.x[1] - u.x[-1])/(2.*Delta);//其实就是求该方向上的梯度
    }

  trash ({uf});
  foreach_face() {
    double un = dt*(u.x[] + u.x[-1])/(2.*Delta), s = sign(un);
    int i = -(s + 1.)/2.;
    uf.x[] = u.x[i] + (g.x[] + g.x[-1])*dt/4. + s*(1. -
    ↪  s*un)*du.x[i]*Delta/2.;
    #if dimension > 1
    if (fm.y[i,0] && fm.y[i,1]) {
      double fyy = u.y[i] < 0. ? u.x[i,1] - u.x[i] : u.x[i] - u.x[i,-1];
      uf.x[] -= dt*u.y[i]*fyy/(2.*Delta);
    }
    #endif
    #if dimension > 2
    if (fm.z[i,0,0] && fm.z[i,0,1]) {
      double fzz = u.z[i] < 0. ? u.x[i,0,1] - u.x[i] : u.x[i] - u.x[i,0,-1];
      uf.x[] -= dt*u.z[i]*fzz/(2.*Delta);
    }
    #endif
    uf.x[] *= fm.x[];
  }

  delete ((scalar *){du});
}

/**
Advection term

We predict the face velocity field $\mathbf{u}_f$ at time $t+\Delta t/2$ then project it to make
↪  it divergence-free. We can then use it to
compute the velocity advection term, using the standard
Bell-Collela-Glaz advection scheme for each component of the velocity
```

```
field. */

event advection_term (i++,last)
{
  if (!stokes) {
    prediction();//注释：预测步，uf 位于 n+1/2 时层
    mgpf = project (uf, pf, alpha, dt/2., mgpf.nrelax);//注释：uf 无散化，pf
    ↪ 位于 n+1/2 时层
    advection ((scalar *){u}, uf, dt, (scalar *){g});//注释：对流方程计算得到
    ↪ u**
  }
}
```

2.4 扩散项处理

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

```
/**
### Viscous term

We first define a function which adds the pressure gradient and
acceleration terms. */

static void correction (double dt)
{
  foreach()
    foreach_dimension()
      u.x[] += dt*g.x[];
}

/**
The viscous term is computed implicitly. We first add the pressure
gradient and acceleration terms, as computed at time t, then call
the implicit viscosity solver. We then remove the acceleration and
pressure gradient terms as they will be replaced by their values at
time t + Δt. */

event viscous_term (i++,last)
```

```
{
  if (constant(mu.x) != 0.) {
    correction (dt);//构造 poisson 型方程的残差，直接构造 u***
    mgu = viscosity (u, mu, rho, dt, mgu.nrelax);
    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 − 1/2 的加速度项与压力梯度项
  }

  /**
  We reset the acceleration field (if it is not a constant). */

  if (!is_constant(a.x)) {
    face vector af = a;
    trash ({af});
    foreach_face()
      af.x[] = 0.;//刷新加速度项
  }
}
```

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 $\mathbf{u}^*_{\mathbf{f\ new}}$ 满足

$$\mathbf{u}_{\mathbf{f}}^{\mathbf{n+1}} = \mathbf{u}^*_{\mathbf{f\ new}} - \frac{\Delta t}{\rho_{n+\frac{1}{2}}}\nabla p_{n+\frac{1}{2}} \tag{17}$$

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

```
/**
### Acceleration term

The acceleration term a needs careful treatment as many
equilibrium solutions depend on exact balance between the acceleration
term and the pressure gradient: for example Laplace's balance for
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+Δt 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+1/2 时
        ↪ 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
        ↪ poisson 方程
24  }
25
26  /**
27  ## 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;
```

```
  /**
  We average these face values to obtain the centered, combined
  acceleration and pressure gradient field. */

  trash ({g});
  foreach()
    foreach_dimension()
      g.x[] = (gf.x[] + gf.x[1])/(fm.x[] + fm.x[1] + SEPS);
}

/**
To get the pressure field at time t + Δt we project the face
velocity field (which will also be used for tracer advection at the
next timestep). Then compute the centered gradient field *g*. */

event projection (i++,last)
{
  mgp = project (uf, p, alpha, dt, mgp.nrelax);
  centered_gradient (p, g);

  /**
  We add the gradient field *g* to the centered velocity field. */

  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 − 1/2 时层的加速度与压力梯度后再未更新）
}

/**
Some derived solvers need to hook themselves at the end of the
timestep. */

event end_timestep (i++, last);

/**
## 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 中最先出现的那一个的位置

2. 组内 event 的执行顺序和出现顺序相反，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：

```
#include "eventtest1.h"
//#include "eventtest2.h"
#include "run.h"

#define MAXTIME 1



int main()
{
  run();
}

event scriptA(i=0; i<=MAXTIME; i++)
{
  fprintf(stdout,"script1\n");
}

event scriptB(i=0; i<=MAXTIME; i++)
{
  fprintf(stdout,"script2\n");
}
```

头文件示例：

```
event test1A(i++,last)
{
  fprintf(stdout,"test1A\n");
}

event scriptA(i++,last)
{
  fprintf(stdout,"test1B\n");
}
```

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：

$$\rho_{n+\frac{1}{2}}\frac{\mathbf{u}^{*}-\mathbf{u}^{***}}{\Delta t}=\nabla\cdot[\mu^{n+\frac{1}{2}}(\mathbf{D}^{*}+\mathbf{D}^{\mathbf{n}})] \tag{18}$$

而误差为：

$$\epsilon=\frac{1}{2}(\Delta t)^{2}\mathbf{L}(\frac{1}{\rho}\mathbf{G}(\frac{\partial p}{\partial t})) \tag{19}$$

参考文献

[1] Ann S Almgren, John B Bell, and William Y Crutchfield. “Approximate projection methods: Part I. Inviscid analysis”. In: *SIAM Journal on Scientific Computing* 22.4 (2000), pp. 1139–1159.

[2] Dimitris Drikakis and William Rider. *High-resolution methods for incompressible and low-speed flows*. Springer Science & Business Media, 2005.

[3] Francis H Harlow and J Eddie Welch. “Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface”. In: *The physics of fluids* 8.12 (1965), pp. 2182–2189.

[4] Stéphane Popinet. “An accurate adaptive solver for surface-tension-driven interfacial flows”. In: *Journal of Computational Physics* 228.16 (2009), pp. 5838–5866.

[5] 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.