



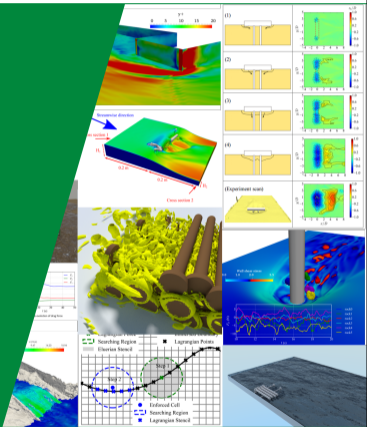
中国农业大学 流体机械与流体工程系

2024年春季《计算流体力学编程实践》

第四章 NS方程求解

徐云成

✉ ycxu@cau.edu.cn



2024年4月11日

三维非稳态不可压Navier-Stokes方程:

$$\nabla \cdot \mathbf{u} = 0 \quad (1)$$

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u}\mathbf{u}) = -\nabla p + \nu \nabla^2 \mathbf{u} \quad (2)$$

其中 p 与可压流体有所不同, 已除密度 ρ

- ▶ 4个未知量: \mathbf{u} (3个方向) 和 p
- ▶ 4个方程

- ▶ 压力与速度是耦合的
- ▶ 没有针对压力的控制方程
- ▶ 但要求压力解必须满足连续性方程
- ▶ NS方程的非线性来自于对流项
 $\nabla \cdot (\mathbf{u}\mathbf{u})$
- ▶ 直接求解困难, 一般用迭代求解或者上一时刻值进行估计, 例如

$$\nabla \cdot (\mathbf{u}\mathbf{u}) \approx \nabla \cdot (\mathbf{u}^{n-1}\mathbf{u}^n)$$



三维非稳态不可压Navier-Stokes方程:

$$\nabla \cdot \mathbf{u} = 0 \quad (1)$$

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u}\mathbf{u}) = -\nabla p + \nu \nabla^2 \mathbf{u} \quad (2)$$

其中 p 与可压流体有所不同, 已除密度 ρ

- ▶ 4个未知量: \mathbf{u} (3个方向) 和 p
- ▶ 4个方程

- ▶ 压力与速度是耦合的
- ▶ 没有针对压力的控制方程
- ▶ 但要求压力解必须满足连续性方程
- ▶ NS方程的非线性来自于对流项
 $\nabla \cdot (\mathbf{u}\mathbf{u})$
- ▶ 直接求解困难, 一般用迭代求解或者上一时刻值进行估计, 例如

$$\nabla \cdot (\mathbf{u}\mathbf{u}) \approx \nabla \cdot (\mathbf{u}^{n-1}\mathbf{u}^n)$$



NS方程的一种估计方法:

$$\frac{\mathbf{u}^n - \mathbf{u}^{n-1}}{\Delta t} + \nabla \cdot (\mathbf{u}^{n-1} \mathbf{u}^n) = -\nabla p^n + \nu \nabla^2 \mathbf{u}^n \quad (3)$$

- ▶ 但是我们不知道 n 时刻的压力 p^n ，也就是新的压力
- ▶ 针对这个问题，我们可以用连续性方程 $\nabla \cdot \mathbf{u} = 0$ 来确定压力
- ▶ 换言之，压力的确定需要满足无散度条件(divergence free condition)的速度场
- ▶ 动量方程实际上是非稳态的带有源项的对流扩散方程，可离散成

$$a_P \mathbf{u}_P + \sum_N a_N \mathbf{u}_N = \mathbf{r} - \nabla p \quad (4)$$



可重写成

$$a_P \mathbf{u}_P = \mathbf{r} - \sum_N a_N \mathbf{u}_N - \nabla p \quad (5)$$

$$a_P \mathbf{u}_P = \mathbf{H}(\mathbf{u}) - \nabla p \quad (6)$$

- ▶ $\mathbf{H}(\mathbf{u})$ 在 OpenFOAM[®] 中已定义
- ▶ $\mathbf{H}(\mathbf{u})$ 是指系数矩阵中非对角线部分和源项
- ▶ a_P 是指系数矩阵中对角线部分

公式(6)两侧除以 a_P 可以得到

$$\mathbf{u}_P = \frac{\mathbf{H}(\mathbf{u})}{a_P} - \frac{\nabla}{a_P} p \quad (7)$$

我们要让速度场 \mathbf{u} 满足无散度条件 $\nabla \cdot \mathbf{u} = 0$, 带入得到

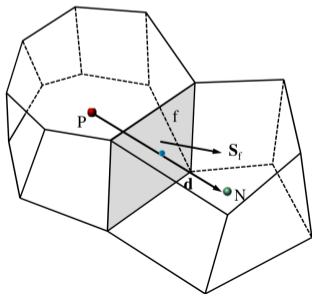
$$\nabla \cdot [(a_P)^{-1} \nabla p] = \nabla \cdot [(a_P)^{-1} \mathbf{H}(\mathbf{u})] \quad (8)$$

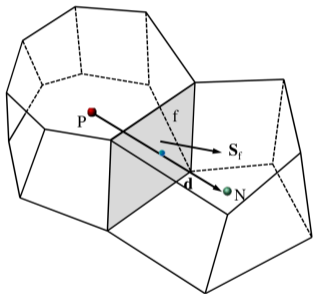
这就是著名的压力泊松方程 Pressure Poisson Equation (PPE)



- ▶ 泊松方程的解能保证速度场的无散度
- ▶ 那么在有限体积法中无散度(divergence free)到底有什么实际意义?
- ▶ 连续性方程可以离散为, 其中 $F = \mathbf{s}_f \cdot \mathbf{u}_f$ 是面通量 face flux

$$\int_V \nabla \cdot \mathbf{u} dV = \int_S \mathbf{u}_f \cdot \mathbf{n} dS = \sum_f \mathbf{s}_f \cdot \mathbf{u}_f = \sum_f F = 0 \quad (9)$$





- ▶ 通量 $F = s_f \cdot \mathbf{u}_f$ 的需要面上速度 \mathbf{u}_f
- ▶ \mathbf{u}_f 不能直接由相邻网格插值得到，因为这可能不满足 divergence free
- ▶ 为确保质量守恒，通量 F 需要从泊松方程得到

- ▶ 首先对离散后泊松方程进行体积分

$$\int_V \nabla \cdot [(a_P)^{-1} \nabla p] dV = \int_V \nabla \cdot [(a_P)^{-1} \mathbf{H}(\mathbf{u})] dV$$

- ▶ 高斯定理得到面积分

$$\int_S [(a_P)^{-1} (\nabla p)_f] \cdot \mathbf{n} dS = \int_S [(a_P)^{-1} \mathbf{H}(\mathbf{u})_f] \cdot \mathbf{n} dS$$

- ▶ 整理得到

$$\int_S \underbrace{[(a_P)^{-1} \mathbf{H}(\mathbf{u})_f - (a_P)^{-1} (\nabla p)_f]}_{\mathbf{u}_f} \cdot \mathbf{n} dS = 0$$

- ▶ 所以面通量 face flux

$$F = \mathbf{s}_f \cdot \mathbf{u}_f = (a_P)^{-1} \mathbf{s}_f \cdot \mathbf{H}(\mathbf{u})_f - (a_P)^{-1} \mathbf{s}_f \cdot (\nabla p)_f$$



- ▶ 首先对离散后泊松方程进行体积分

$$\int_V \nabla \cdot [(a_P)^{-1} \nabla p] dV = \int_V \nabla \cdot [(a_P)^{-1} \mathbf{H}(\mathbf{u})] dV$$

- ▶ 高斯定理得到面积分

$$\int_S [(a_P)^{-1} (\nabla p)_f] \cdot \mathbf{n} dS = \int_S [(a_P)^{-1} \mathbf{H}(\mathbf{u})_f] \cdot \mathbf{n} dS$$

- ▶ 整理得到

$$\int_S \underbrace{[(a_P)^{-1} \mathbf{H}(\mathbf{u})_f - (a_P)^{-1} (\nabla p)_f]}_{\mathbf{u}_f} \cdot \mathbf{n} dS = 0$$

- ▶ 所以面通量 face flux

$$F = \mathbf{s}_f \cdot \mathbf{u}_f = (a_P)^{-1} \mathbf{s}_f \cdot \mathbf{H}(\mathbf{u})_f - (a_P)^{-1} \mathbf{s}_f \cdot (\nabla p)_f$$



- ▶ 首先对离散后泊松方程进行体积分

$$\int_V \nabla \cdot [(a_P)^{-1} \nabla p] dV = \int_V \nabla \cdot [(a_P)^{-1} \mathbf{H}(\mathbf{u})] dV$$

- ▶ 高斯定理得到面积分

$$\int_S [(a_P)^{-1} (\nabla p)_f] \cdot \mathbf{n} d\mathbf{S} = \int_S [(a_P)^{-1} \mathbf{H}(\mathbf{u})_f] \cdot \mathbf{n} d\mathbf{S}$$

- ▶ 整理得到

$$\int_S \underbrace{[(a_P)^{-1} \mathbf{H}(\mathbf{u})_f - (a_P)^{-1} (\nabla p)_f]}_{\mathbf{u}_f} \cdot \mathbf{n} d\mathbf{S} = 0$$

- ▶ 所以面通量 face flux



- ▶ 首先对离散后泊松方程进行体积分

$$\int_V \nabla \cdot [(a_P)^{-1} \nabla p] dV = \int_V \nabla \cdot [(a_P)^{-1} \mathbf{H}(\mathbf{u})] dV$$

- ▶ 高斯定理得到面积分

$$\int_S [(a_P)^{-1} (\nabla p)_f] \cdot \mathbf{n} d\mathbf{S} = \int_S [(a_P)^{-1} \mathbf{H}(\mathbf{u})_f] \cdot \mathbf{n} d\mathbf{S}$$

- ▶ 整理得到

$$\int_S \underbrace{[(a_P)^{-1} \mathbf{H}(\mathbf{u})_f - (a_P)^{-1} (\nabla p)_f]}_{\mathbf{u}_f} \cdot \mathbf{n} d\mathbf{S} = 0$$

- ▶ 所以面通量 face flux

$$F = \mathbf{s}_f \cdot \mathbf{u}_f = (a_P)^{-1} \mathbf{s}_f \cdot \mathbf{H}(\mathbf{u})_f - (a_P)^{-1} \mathbf{s}_f \cdot (\nabla p)_f$$



由压力泊松方程，我们可以引入2个NS方程中常用的迭代求解算法：

- ▶ SIMPLE
- ▶ PISO



- ▶ 这是求解稳态问题最早的压力-速度耦合算法
- ▶ Semi-Implicit Algorithm for Pressure-Linked Equations (英国帝国理工大学, Patankar and Spalding, 1972)



1. 估计压力场 p^* (或者用上一迭代步的值)
2. 预测动量 Momentum predictor: 利用估计的压力求解动量方程

$$a_P \mathbf{u}_P = \mathbf{H}(\mathbf{u}) - \nabla p^*$$

3. 压力修正 Pressure correction: 根据预测速度场计算新的压力

$$\nabla \cdot [(a_P)^{-1} \nabla p] = \nabla \cdot [(a_P)^{-1} \mathbf{H}(\mathbf{u})]$$

4. 计算面通量

$$F = (a_P)^{-1} \mathbf{s}_f \cdot \mathbf{H}(\mathbf{u})_f - (a_P)^{-1} \mathbf{s}_f \cdot (\nabla p)_f$$

5. 利用新计算压力修正速度场

$$\mathbf{u}_P = \frac{\mathbf{H}(\mathbf{u})}{a_P} - \frac{\nabla p}{a_P}$$



1. 估计压力场 p^* (或者用上一迭代步的值)
2. 预测动量 Momentum predictor: 利用估计的压力求解动量方程

$$a_P \mathbf{u}_P = \mathbf{H}(\mathbf{u}) - \nabla p^*$$

3. 压力修正 Pressure correction: 根据预测速度场计算新的压力

$$\nabla \cdot [(a_P)^{-1} \nabla p] = \nabla \cdot [(a_P)^{-1} \mathbf{H}(\mathbf{u})]$$

4. 计算面通量

$$F = (a_P)^{-1} \mathbf{s}_f \cdot \mathbf{H}(\mathbf{u})_f - (a_P)^{-1} \mathbf{s}_f \cdot (\nabla p)_f$$

5. 利用新计算压力修正速度场

$$\mathbf{u}_P = \frac{\mathbf{H}(\mathbf{u})}{a_P} - \frac{\nabla p}{a_P}$$



1. 估计压力场 p^* (或者用上一迭代步的值)
2. 预测动量 Momentum predictor: 利用估计的压力求解动量方程

$$a_P \mathbf{u}_P = \mathbf{H}(\mathbf{u}) - \nabla p^*$$

3. 压力修正 Pressure correction: 根据预测速度场计算新的压力

$$\nabla \cdot [(a_P)^{-1} \nabla p] = \nabla \cdot [(a_P)^{-1} \mathbf{H}(\mathbf{u})]$$

4. 计算面通量

$$F = (a_P)^{-1} \mathbf{s}_f \cdot \mathbf{H}(\mathbf{u})_f - (a_P)^{-1} \mathbf{s}_f \cdot (\nabla p)_f$$

5. 利用新计算压力修正速度场

$$\mathbf{u}_P = \frac{\mathbf{H}(\mathbf{u})}{a_P} - \frac{\nabla p}{a_P}$$



1. 估计压力场 p^* (或者用上一迭代步的值)
2. 预测动量 Momentum predictor: 利用估计的压力求解动量方程

$$a_P \mathbf{u}_P = \mathbf{H}(\mathbf{u}) - \nabla p^*$$

3. 压力修正 Pressure correction: 根据预测速度场计算新的压力

$$\nabla \cdot [(a_P)^{-1} \nabla p] = \nabla \cdot [(a_P)^{-1} \mathbf{H}(\mathbf{u})]$$

4. 计算面通量

$$F = (a_P)^{-1} \mathbf{s}_f \cdot \mathbf{H}(\mathbf{u})_f - (a_P)^{-1} \mathbf{s}_f \cdot (\nabla p)_f$$

5. 利用新计算压力修正速度场

$$\mathbf{u}_P = \frac{\mathbf{H}(\mathbf{u})}{a_P} - \frac{\nabla p}{a_P}$$



1. 估计压力场 p^* (或者用上一迭代步的值)
2. 预测动量 Momentum predictor: 利用估计的压力求解动量方程

$$a_P \mathbf{u}_P = \mathbf{H}(\mathbf{u}) - \nabla p^*$$

3. 压力修正 Pressure correction: 根据预测速度场计算新的压力

$$\nabla \cdot [(a_P)^{-1} \nabla p] = \nabla \cdot [(a_P)^{-1} \mathbf{H}(\mathbf{u})]$$

4. 计算面通量

$$F = (a_P)^{-1} \mathbf{s}_f \cdot \mathbf{H}(\mathbf{u})_f - (a_P)^{-1} \mathbf{s}_f \cdot (\nabla p)_f$$

5. 利用新计算压力修正速度场

$$\mathbf{u}_P = \frac{\mathbf{H}(\mathbf{u})}{a_P} - \frac{\nabla p}{a_P}$$



1. 估计压力场 p^* (或者用上一迭代步的值)
2. 预测动量 Momentum predictor: 利用估计的压力求解动量方程

$$a_P \mathbf{u}_P = \mathbf{H}(\mathbf{u}) - \nabla p^*$$

3. 压力修正 Pressure correction: 根据预测速度场计算新的压力

$$\nabla \cdot [(a_P)^{-1} \nabla p] = \nabla \cdot [(a_P)^{-1} \mathbf{H}(\mathbf{u})]$$

4. 计算面通量

$$F = (a_P)^{-1} \mathbf{s}_f \cdot \mathbf{H}(\mathbf{u})_f - (a_P)^{-1} \mathbf{s}_f \cdot (\nabla p)_f$$

5. 利用新计算压力修正速度场

$$\mathbf{u}_P = \frac{\mathbf{H}(\mathbf{u})}{a_P} - \frac{\nabla p}{a_P}$$



SIMPLE 算法的稳定性问题:

- ▶ 迭代算法总是期望得到收敛的速度和压力
- ▶ 但是, 如果网格质量不好或者初始条件不好, 计算有可能发散(diverge)
- ▶ 为提高收敛性, 通常会使用亚松弛 under-relaxation

$$p^n = p^{n-1} + \alpha_P(p^{\text{predicted}} - p^{n-1})$$
$$\mathbf{u}^n = \mathbf{u}^{n-1} + \alpha_U(\mathbf{u}^{\text{predicted}} - \mathbf{u}^{n-1})$$

其中 α_P 和 α_U 是松弛因子, 通常小于1



SIMPLE 算法的稳定性问题:

- ▶ 迭代算法总是期望得到收敛的速度和压力
- ▶ 但是, 如果网格质量不好或者初始条件不好, 计算有可能发散(diverge)
- ▶ 为提高收敛性, 通常会使用亚松弛 under-relaxation

$$p^n = p^{n-1} + \alpha_P(p^{\text{predicted}} - p^{n-1})$$
$$\mathbf{u}^n = \mathbf{u}^{n-1} + \alpha_U(\mathbf{u}^{\text{predicted}} - \mathbf{u}^{n-1})$$

其中 α_P 和 α_U 是松弛因子, 通常小于1



SIMPLE 算法的稳定性问题:

- ▶ 迭代算法总是期望得到收敛的速度和压力
- ▶ 但是, 如果网格质量不好或者初始条件不好, 计算有可能发散(diverge)
- ▶ 为提高收敛性, 通常会使用亚松弛 under-relaxation

$$p^n = p^{n-1} + \alpha_P(p^{\text{predicted}} - p^{n-1})$$
$$\mathbf{u}^n = \mathbf{u}^{n-1} + \alpha_U(\mathbf{u}^{\text{predicted}} - \mathbf{u}^{n-1})$$

其中 α_P 和 α_U 是松弛因子, 通常小于1



SIMPLE 算法的稳定性问题:

- ▶ 实际操作中，动量的亚松弛计算是隐性的，压力的亚松弛计算是显性的
- ▶ 在OpenFOAM[®] 中的simpleFoam:

```
UEqn.relax();
```

这一步隐性亚松弛计算实际上是:

$$\frac{a_P}{\alpha_U} \mathbf{u}_P = \mathbf{H}(\mathbf{u}) - \nabla p^* + \frac{1 - \alpha_U}{\alpha_U} a_P \mathbf{u}_P^* \quad (10)$$

- ▶ 压力是进行显性亚松弛计算 `p.relax()`;



- ▶ 一般选取原则

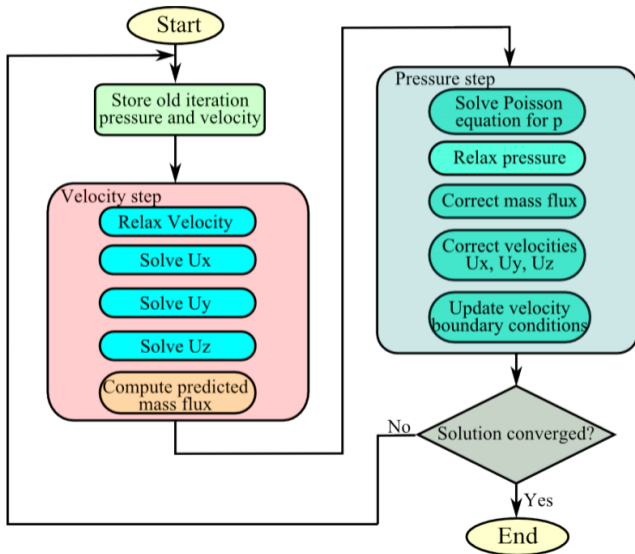
$$0 < \alpha_P \leq 1$$

$$0 < \alpha_U \leq 1$$

$$\alpha_P + \alpha_U \approx 1 \text{ or } 1.1$$

- ▶ Patankar(1980)推荐: $\alpha_P = 0.5, \alpha_U = 0.8$
- ▶ OpenFOAM[®] 默认: $\alpha_P = 0.3, \alpha_U = 0.7$
- ▶ 最优松弛因子取决于具体网格质量, 因问题而异
- ▶ 如果计算容易发散, 其中一种选择是多试几组松弛因子





- ▶ 在调用函数`simple.loop(runTime)`时，同时运行了`storePrevIterFields()`；
这是用于存储上一迭代步的压力速度，用于求解亚松弛计算
- ▶ 定义速度方程

```
tmp<fvVectorMatrix> tUEqn  
(  
    fvm::div(phi, U)  
    + MRF.DDt(U)  
    + turbulence->divDevSigma(U)  
    ==  
    fvOptions(U)  
);
```

`tmp`用于减小内存使用峰值



▶ 速度方程的亚松弛计算

```
UEqn.relax();
```

▶ 求解动量方程

```
solve(UEqn == -fvc::grad(p));
```

▶ 计算系数 a_P 和 U

```
volScalarField rAU(1.0/UEqn.A());  
volVectorField HbyA(constrainHbyA(rAU*UEqn.H(), U, p));
```

▶ 计算由于 $(a_P)^{-1} \mathbf{s}_f \cdot \mathbf{H}(\mathbf{u})_f$ 造成的部分面通量

```
surfaceScalarField phiHbyA("phiHbyA", fvc::flux(HbyA));  
...  
adjustPhi(phiHbyA, U, p);
```



- ▶ 定义并求解压力方程

```
fvScalarMatrix pEqn  
(  
    fvm::laplacian(rAtU(), p) == fvc::div(phiHbyA)  
);
```

```
pEqn.setReference(pRefCell, pRefValue);
```

```
pEqn.solve();
```

- ▶ 通过增加压力梯度部分 $(a_P)^{-1} \mathbf{s}_f \cdot (\nabla p)_f$ 来修正面通量

```
phi = phiHbyA - pEqn.flux();
```

- ▶ 计算质量守恒误差/连续性误差

```
#include "continuityErrs.H"
```



- ▶ 对压力进行亚松弛计算，并进行速度修正 Momentum corrector

```
// Explicitly relax pressure for momentum corrector  
p.relax();
```

```
// Momentum corrector  
U = HbyA - rAtU()*fvc::grad(p);  
U.correctBoundaryConditions();
```

- ▶ 检查是否满足收敛条件，该步骤在`simple.loop(runTime)`中进行



SIMPLE算法中收敛是什么意思？

- ▶ 速度（三个方向）和压力在迭代时不再发生变化
- ▶ 收敛条件是在system/fvSolutions中定义

```
SIMPLE
```

```
{
```

```
    ...
```

```
    residualControl
```

```
    {
```

```
        p                1e-2;
```

```
        U                1e-3;
```

```
        "(k|epsilon|omega|f|v2)" 1e-3;
```

```
    }
```

```
}
```

- ▶ **注意：**这里的收敛性和线性方程求解器的收敛性不同



PISO: Pressure Implicit with Splitting of Operator (Issa, 1986)

- ▶ 回顾SIMPLE
 - 使用估计的压力场求解动量方程
 - 求解压力泊松方程以满足连续性要求
 - 但是得到的压力仍然不准确，因为压力泊松方程的右侧仍旧使用估计的速度
 - 这就是为什么需要进行迭代
- ▶ SIMPLE算法第一次压力修正后由两部分：
 - 物理部分，我们所需要的
 - 非物理部分，用于当前步满足连续性要求
- ▶ SIMPLE迭代的目的是消除非物理意义部分，最终得到具有物理意义的解的收敛
- ▶ SIMPLE迭代中，动量方程和泊松方程是依次轮番求解， U 和 p 需要分开进行亚松弛计算



PISO的思想是固定动量方程迭代直至收敛，多进行几次压力修正迭代

- ▶ NS方程具有两种耦合：
 - 非线性 $\mathbf{u} - \mathbf{u}$ 耦合，例如对流项
 - 线性 $\mathbf{u} - p$ 耦合

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

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u}\mathbf{u}) = -\nabla p + \nu \nabla^2 \mathbf{u}$$

- ▶ PISO基于以下假设：
 - 当库郎数（时间步长）很小时， $\mathbf{u} - p$ 耦合比非线性耦合更加强烈
- ▶ 因此对于一次动量预测，需要重复多次压力修正
- ▶ PISO可以看做是SIMPLE的一种拓展



- ▶ 原始PISO算法是进行2次压力修正
- ▶ 实际上可以进行超过2次的压力修正，但无需进行太多次，因为动量方程不变
- ▶ 由于进行了多次压力修正，压力不需要进行亚松弛，但是速度仍需要亚松弛



1. 预测动量 Momentum predictor: 利用估计的压力（上一时间步长）求解动量方程

$$a_P \mathbf{u}_P = \mathbf{H}(\mathbf{u}) - \nabla p^*$$

2. 压力修正 Pressure correction: 根据预测速度场计算新的压力

$$\nabla \cdot [(a_P)^{-1} \nabla p] = \nabla \cdot [(a_P)^{-1} \mathbf{H}(\mathbf{u})]$$

3. 计算面通量

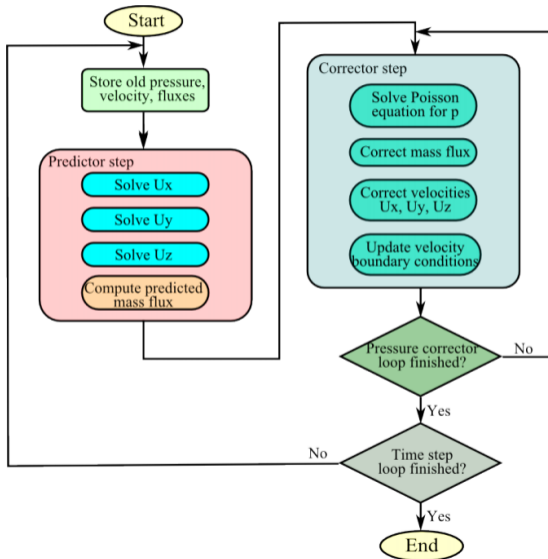
$$F = (a_P)^{-1} \mathbf{s}_f \cdot \mathbf{H}(\mathbf{u})_f - (a_P)^{-1} \mathbf{s}_f \cdot (\nabla p)_f$$

4. 利用新计算压力修正速度场

$$\mathbf{u}_P = \frac{\mathbf{H}(\mathbf{u})}{a_P} - \frac{\nabla p}{a_P}$$

5. 重复压力修正步骤，然后再进行新的时间步长





- ▶ 定义速度方程(增加了时间项 $fvm::ddt(U)$)

```
fvVectorMatrix UEqn
(
    fvm::ddt(U) + fvm::div(phi, U)
    + MRF.DDt(U)
    + turbulence->divDevSigma(U)
    ==
    fvOptions(U)
);
```



```
Info<< "\nStarting time loop\n" << endl;

while (runTime.loop())
{
    Info<< "Time = " << runTime.timeName() << nl << endl;

    #include "CourantNo.H"

    // Pressure-velocity PISO corrector
    {
        #include "UEqn.H"

        // --- PISO loop
        while (piso.correct())
        {
            #include "pEqn.H"
        }
    }

    laminarTransport.correct();
    turbulence->correct();

    runTime.write();

    Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
        << " ClockTime = " << runTime.elapsedClockTime() << " s"
        << nl << endl;
}

Info<< "End\n" << endl;
```



- ▶ PIMPLE: PISO-SIMPLE, 大时间步长、不可压流体求解器
- ▶ 拥有两个循环: 内循环inner(PISO corrector)和外循环 outer corrector, 以确保收敛性和更好的耦合
- ▶ system/fvSolutions

```
PIMPLE
```

```
{
```

```
    nOuterCorrectors      2;      //for outer corrector
```

```
    nCorrectors           1;      //for inner PISO corrector
```

```
    nNonOrthogonalCorrectors 0;
```

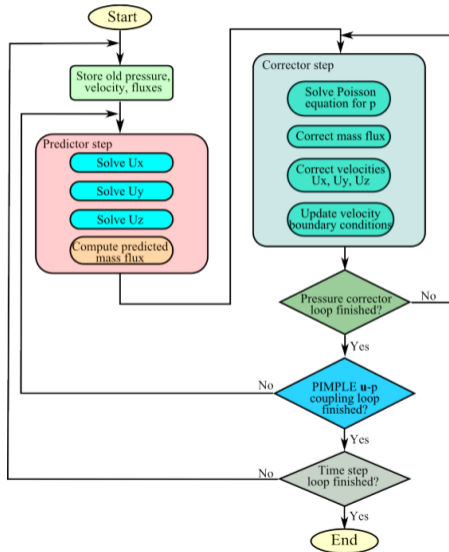
```
}
```

- ▶ 如果nOuterCorrectors=1, 就相当于PISO模式



OpenFOAM[®] 中的PIMPLE算法

28/30



SIMPLE vs PISO vs PIMPLE

9/30

```
Info<< "\nStarting time loop\n" << endl;
while (simple.loop(runTime))
{
    Info<< "Time = " << runTime.timeName() << nl << endl;

    // --- Pressure-velocity SIMPLE corrector
    {
        #include "UEqn.H"
        #include "pEqn.H"
    }

    laminarTransport.correct();
    turbulence->correct();

    runTime.write();

    Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
        << " ClockTime = " << runTime.elapsedClockTime() << " s"
        << nl << endl;
}
Info<< "End\n" << endl;
```

```
Info<< "\nStarting time loop\n" << endl;
while (runTime.loop())
{
    Info<< "Time = " << runTime.timeName() << nl << endl;

    #include "CourantNo.H"

    // Pressure-velocity PISO corrector
    {
        #include "UEqn.H"

        // --- PISO loop
        while (piso.correct())
        {
            #include "pEqn.H"
        }
    }

    laminarTransport.correct();
    turbulence->correct();

    runTime.write();

    Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
        << " ClockTime = " << runTime.elapsedClockTime() << " s"
        << nl << endl;
}
Info<< "End\n" << endl;
```

```
Info<< "\nStarting time loop\n" << endl;
while (pimple.run(runTime))
{
    #include "readDyMControls.H"

    if (LTS)
    {
        #include "setRDeltaT.H"
    }
    else
    {
        #include "CourantNo.H"
        #include "setDeltaT.H"
    }

    runTime++;

    Info<< "Time = " << runTime.timeName() << nl << endl;

    // --- Pressure-velocity PIMPLE corrector loop
    while (pimple.loop())
    {
        #include "UEqn.H"

        // --- Pressure corrector loop
        while (pimple.correct())
        {
            #include "pEqn.H"
        }

        if (pimple.turbCorr())
        {
            laminarTransport.correct();
            turbulence->correct();
        }
    }

    runTime.write();

    Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
        << " ClockTime = " << runTime.elapsedClockTime() << " s"
        << nl << endl;
}
Info<< "End\n" << endl;
```



Thank you.

欢迎私下交流，请勿私自上传网络，谢谢！

