FVM implementation of NS Equation discretization in OpenFoam

Ajit Kumar
Shiv Nadar University
Email: *ajit.kumar@snu.edu.in*

May 18, 2018

1 Navier Stokes Equation and Finite Volume discretized equations

This notes explains OpenFoam implements two of the most widely used Finite Volume discretization of Navier Stoke's Equation: PISO & SIMPLE. The theory is picked up from Prof. Hrvoje Jasak's Phd thesis [1]. Since he is the key developer of `foam-extend`, I will walk through the `pisoFoam` and `simpleFoam` code as implemented in foam-extend-4.0.

Incompressible Navier Stokes equation is given by a pair of partial differential equations

$$\nabla \cdot \vec{u} = 0, \tag{1}$$

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

in field variables $\vec{u}$ and p which represent velocity and normalized pressure (pressure divided by fluid-density) field respectively. Eq.(1) represents conservation of mass where as Eq.(2) represents conservation of momentum. The first term in Eq.(2) will vanish for steady state problems. The symbol $\nu = \nu_f + \nu_T$ is fluid viscosity plus turbulent viscosity. For laminar flows, $\nu_T = 0$, where as in turbulent flows ν_T is calculated by solving additional equations such as $k - \omega$, $k - \epsilon$, etc.

Lets assume the flow domain is divided into N number of cells, and we are interested in finding fluid velocity $\vec{u}_i$ and pressure p_i at the centroid of each cell, $i \in \{1, 2, 3, ..., N\}$. Lets formally assemble these discrete variables into two big vector $\mathbf{U}$ and $\mathbf{p}$:

$$\mathbf{U} = \begin{bmatrix} \vec{u}_1 \\ \vec{u}_2 \\ \vdots \\ \vec{u}_N \end{bmatrix} \quad \mathbf{p} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_N \end{bmatrix}$$

In finite volume method, a discrete approximation of Eq.(1) and (2) is obtained by integrating each equation over every cell, occasionally transferring cell volume integration to cell boundary integration (using divergence theorem), and then using various interpolation schemes to approximate these integrals in terms of the discrete variables $\mathbf{U}$ and $\mathbf{p}$.

If we try to solve for $\mathbf{U}$ and $\mathbf{p}$ simultaneously, we come across linear equations which is not suitable for iterative linear solvers (See [2] section 15.1). To overcome this problem, algorithm like SIMPLE and PISO were proposed in which we do not solver for $\mathbf{U}$ and $\mathbf{p}$ simultaneously, but rather solved in a "segregated" fashion.

Before we start taking about SIMPLE and PISO, let me introduce one intermediate variable which may not play a huge role in abstraction of Navier Stokes equation but appears often in OpenFoam implementation of SIMPLE and PISO. It is the cell faces interpolation of fluid velocity, $\mathbf{Up}_f$. This is an intermediate variable and lets assume that we get a matrix M is known which maps $\mathbf{U}$ to $\mathbf{U}_f$.

$$\mathbf{U}_f = M\mathbf{U}$$

Because of divergence theorem, the finite volume form of continuity equation is written in terms of face centered fields. Momentum equations are discretized in terms of cell centric variables, and split into the diagonal and off-diagonal components to simplify the discussion of consistency and convergence of iterative schemes. OpenFoam assumes a discrete form of Equations (1) and (2) to have the form

$$C\overbrace{M\mathbf{U}}^{=\mathbf{U}_f} = \mathbf{0} \tag{3}$$

$$\mathbf{AU} - [\mathbf{L}(\mathbf{U})\mathbf{U} + \mathbf{X}] = -D\mathbf{p} \tag{4}$$

Here $\mathbf{L}(U)$ is matrix which depends on $\mathbf{U}$ (non linear because convection term in non-linear), $\mathbf{X}$ is known vector (coming from discretization of time derivative), $\mathbf{A}$ and D are known vectors of suitable orders. Eq. (3) is a discrete version of the continuity equation which essentially means that net volume flux across each cell is zero:

$$\sum_{\text{faces in cell}} \mathbf{U}_f \cdot \mathbf{S}_f = 0, \text{ for all cells}$$

Eq. (4) is a discrete version of the momentum equation. As mentioned earlier, solving Eqs. (3) and (4) is problematic because the resulting coefficient matrix is not suitable for iterative solver (See [2] section 15.1, try Exercise (1).

Both SIMPLE and PISO are iterative method and at each will involve solving for $\mathbf{U}^{n+1}$ and $\mathbf{p}^{n+1}$ assuming $\mathbf{U}^0, \mathbf{U}^1, ...\mathbf{U}^n$ and $\mathbf{p}^0, \mathbf{p}^1, ...\mathbf{p}^n$ are known. Here the superscript denote the time step or iteration step.

2 PISO and its OpenFoam implementation

As mentioned earlier, for each time step in PISO loop we assume $\mathbf{U}^0, \mathbf{U}^1, ...\mathbf{U}^n$ and $\mathbf{p}^0, \mathbf{p}^1, ...\mathbf{p}^n$ are known, and we aim to solve for $\mathbf{U}^{n+1}$ and $\mathbf{p}^{n+1}$. We do this by first finding an intermediate $\mathbf{U}^*$ using a so called *momentum predictor* step followed by *PISO corrections*.

2.1 Momentum predictor

In this predictor step, presently known values of velocity and pressure: $\mathbf{U}^n$, $\mathbf{p}^n$, are used to construct the coefficient $\mathbf{L}(\mathbf{U}^n)$, the vector $\mathbf{X}(\mathbf{U}^n)$. The matrices $\mathbf{A}$ and D are constructed from mesh and runtime chosen interpolation schemes. And we construct an equation for the predictor $\mathbf{U}^*$.

$$\mathbf{AU}^* - \underbrace{[\mathbf{L}(\mathbf{U}^n)\mathbf{U}^* + \mathbf{X}(\mathbf{U}^n]}_{=H(\mathbf{U}^*)} = -D\mathbf{p}^n \tag{5}$$

OpenFoam creates Eq.(5) by first creating the LHS in a manner which mimics the LHS of Eq. (2):

```
fvVectorMatrix UEqn
(
    fvm::ddt(U)
  + fvm::div(phi, U)
  + turbulence->divDevReff()
);
```

Listing 1: LHS of the discretized momentum equation

Lets dissect Listing 1. An object `UEqn` of type `fvVectorMatrix` is constructed to solve for the `volVectorField U`. `fvm::ddt(Up)` is a finite volume discretization of $\frac{\partial \mathbf{U}}{\partial t}$. In the term `fvm::div(phi,U)`, `phi = linearInterpolate(U) & mesh.Sf()` is a `surfaceScalarField` which denotes volume flux through cell faces. So, `fvm::div(phi,U)` is mimicking $\nabla \cdot (\mathbf{U}^n \mathbf{U})$, convection of unknown momentum density $\mathbf{U}$ being transported by currently known velocity field $\mathbf{U}^n$. The `turbulence->divDevReff()` is actually implemented as

```
fvm::laplacian(nuEff(), U) -fvc::div(nuEff()*dev(T(fvc::grad(U))))
```

where `nuEff()` represents effective viscosity which is calculated as fluid viscosity by turbulence viscosity, $\nu_{\text{eff}} = \nu + \nu_T$. The second term I believe represents explicit treatment of devioteric stress tensor which should have been vanished in incompressible flows, right?

The equation is closed by constructing the rhs of Eq. (5) using current value of pressure $\mathbf{p}^n$ by the command `fvc::grad(p)`:. Before that coefficient matrix `UEqn` is also relaxed. I don't know the purpose of that at this point. TODO: Check why and how of `EUqn.relax()`.

```
UEqn.relax();

if (piso.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));
}
```

The `solve(...)` function solves for `U`. Don't forget, this is $\mathbf{U}^*$ in our abstraction. To get equation in the form of Eq. (4), we will need to extract $\mathbf{A}$ and $\mathbf{H}$ from `UEqn`. These can be done (but not done this way) as

```
volScalarField a = UEqn.A();
volScalarField H = UEqn.H();
```

2.2 PISO Correction loop

```
// --- PISO loop
while (piso.correct())
   {
```

Recall that we have only have a predicted value $\mathbf{U}^*$. The momentum equation can be rephrased as

$$\mathbf{U}^{n+1} = \frac{\mathbf{H}(\mathbf{U}^{n+1})}{\mathbf{A}} - \frac{D}{A}\mathbf{p}^{n+1} \tag{6}$$

The PISO corrector loop approximates $\mathbf{H}(\mathbf{U}^{n+1}) \approx \mathbf{H}(\mathbf{U}^*)$ to get

$$\mathbf{U}^{n+1} = \frac{H(\mathbf{U}^*)}{A} - \frac{D}{A}\mathbf{p}^{n+1} \tag{7}$$

The field $\frac{\mathbf{H}(\mathbf{U}^*)}{A}$ is constructed as

```
volScalarField rUA = 1.0/UEqn.A();
volVectorField HbyA = rUA*UEqn.H();
```

Eq. (7) is substituted in Eq. (3) to get an equation for $\mathbf{p}^{n+1}$

$$\frac{CMD}{A}\mathbf{p}^{n+1} = \frac{CM}{A}\mathbf{H}(\mathbf{U}^*) \tag{8}$$

Eq. (7) is implemented and solved for $\mathbf{p}^{n+1}$

```
  phi = (fvc::interpolate(U) & mesh.Sf())
         + fvc::ddtPhiCorr(rUA, U, phi); // TODO: This is not fully understood
                                         // I think by most Foamers

  adjustPhi(phi, U, p);                  // TODO: check its usage as well

  // Non-orthogonal pressure corrector loop
  while (piso.correctNonOrthogonal())
  {
    // Pressure corrector

    fvScalarMatrix pEqn
    (
    fvm::laplacian(rUA, p) == fvc::div(phi)
    );

    pEqn.setReference(pRefCell, pRefValue);
    pEqn.solve
    (
    mesh.solutionDict().solver
    (
    p.select(piso.finalInnerIter())
    )
    );
```

```
    if (piso.finalNonOrthogonalIter())
    {
      phi -= pEqn.flux();
    }
  }

  #include "continuityErrs.H"
```

and $\mathbf{U}^{n+1}$ is updated using Eq. (7)

```
 U -= rUA*fvc::grad(p);
 U.correctBoundaryConditions();
```

Because the equations are non-linear, the newly obtained $\mathbf{U}^{n+1}$ are used to create new $\mathbf{H}(\mathbf{U}^{n+1})$, which further feds into the equation for p^{n+1}. Process repeats till convergence is reached.

```
} // end PISO loop
```

And solution is updated and recorded after convergence in PISO loop.

```
  turbulence->correct();  // solve for turbulence model
                          // to get nut

  runTime.write();        // write converged solution on files
} // end rutime while loop loop
```

3 Exercise

1. [TODO] Conduct a numerical experiment to investigate the issues that arise when trying to solve Eqs. (3)-(4) simultaneously.

2. [TODO] I think, for incompressible flows, the devioteric stress tensor operation of velocity field need not be present. Study a case by removing that term and see whether that affects to solutions or not.

References

[1] Hrvoje Jasak. Error analysis and estimation for finite volume method with applications to fluid flow. Technical report, 1996.

[2] F Moukalled, L Mangani, M Darwish, et al. *The finite volume method in computational fluid dynamics*. Springer, 2016.