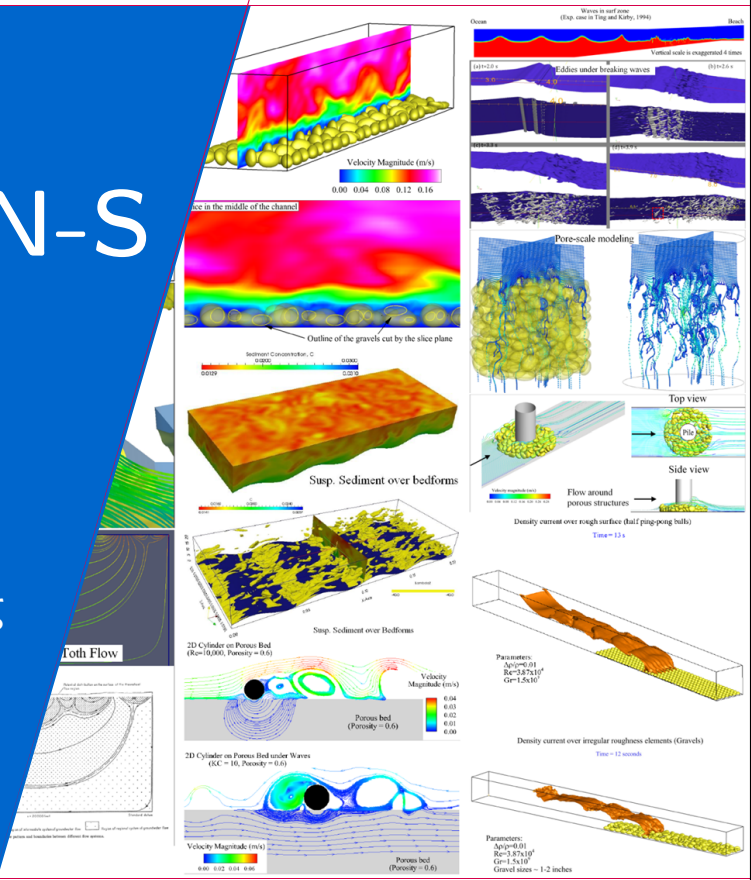
PENNSTATE

1855

CE597D Special Topics, Fall, 2014

Chapter 6: Solution of N-S Equations (Part I)

Xiaofeng Liu, Ph.D., P.E.
Assistant Professor
Department of Civil and Environmental Engineering
Pennsylvania State University
xliu@engr.psu.edu



Outline

General overview

What will be covered in this chapter?

- General overview of the pressure–velocity coupling
- Solution algorithms
  - **Segregated** algorithms: u, v, w and p fields are solved separately. Coupling between these variables are through velocity and pressure corrections.
    - Iterative algorithms: SIMPLE, PISO, etc.
    - Projection method
  - **Coupled** algorithms: All fields are solved in one shot!
    - Not covered in this course.
    - Can be achieved through the new *block* − *matrix*

Pressure and Velocity

Unsteady, 3D Navier-Stokes equations for incompressible flow:

$$\nabla \cdot \mathbf{u} = 0 \tag{1}$$

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

where $\nu = \mu/\rho$ is the kinematic viscosity. Note the definition of the pressure p here is a little different from the one for compressible flows, i.e., it has been divided by the density ρ.

Note:

- Four unknowns: $\mathbf{u}$ (three components) and p
- Four equations.
- Velocity and pressure are coupled
- There is no specific governing equation for pressure
- Instead, it requires that the solution for p must satisfy the continuity equation

Pressure and Velocity

Note (continue):

- Non-linearity of NS equations comes from the advection of momentum: $\nabla \cdot (\mathbf{uu})$
- Direct solution of this nonlinear equation is difficult and costly.
- Usually an iterative solution algorithm or an approximation using old time step values is used.
- For example, the nonlinear term can be linearized as

$$\nabla \cdot (\mathbf{uu}) \approx \nabla \cdot (\mathbf{u}^o \mathbf{u}^n)$$

  where $\mathbf{u}^o$ is the currently available solution or an initial guess and $\mathbf{u}^n$ is the "new" solution. The algorithm loops until $\mathbf{u}^o \approx \mathbf{u}^n$.
- Another example, it can be approximated as

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

  where $\mathbf{u}^{n-1}$ is the old time step value.

Pressure and Velocity

Derivation of the Pressure Equation:

- Now we go back to the NS equation. We can approximately solve for $\mathbf{u}$ using the momentum equation.
- For example:

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

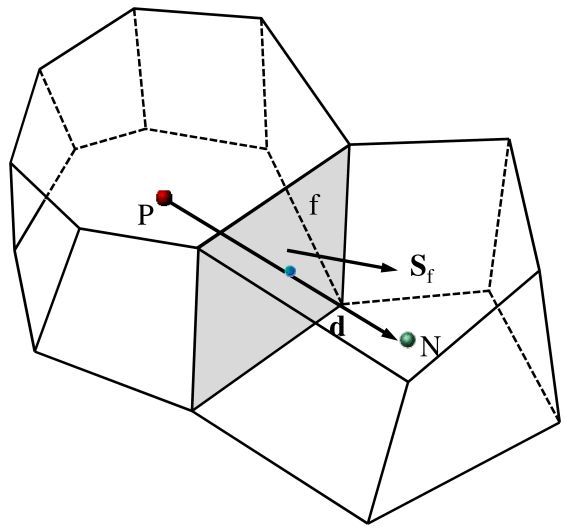
- But the problem is that we don't know p^n, i.e., the new pressure value
- Don't worry! We have another equation, the continuity equation to nail down the pressure value.
- In other words, the determination of the pressure is through the satisfaction of divergence free condition of the velocity field.

Pressure and Velocity

Derivation of the Pressure Equation:

- The linearized momentum equation is nothing but an unsteady advection-diffusion equation with some source term
- It can be discretized using the numerical methods we have learned so far.
- To derive an equation for p, the pressure gradient term will remain in the differential form.
- The discretised momentum equation for each control volume:

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



Pressure and Velocity

Derivation of the Pressure Equation:

- The discretised momentum equation for each control volume:

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

which can be re-written as

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

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

- The $\mathbf{H}(\mathbf{u})$ operator is introduced (implemented in OpenFOAM® too)
- $\mathbf{H}(\mathbf{u})$ contains the off-diagonal part of the momentum matrix and the source contribution:

$$\mathbf{H}(\mathbf{u}) = \mathbf{r} - \sum_N a_N \mathbf{u}_N$$

Pressure and Velocity

Derivation of the Pressure Equation:

- For each control volume, we have:

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

- In this equation, ∇p is yet to be determined.
- Divide both sides by a_P, we get

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

- Our aim is to have a $\mathbf{u}$ field satisfying the divergence free condition.
- The only adjustable term is the pressure gradient ∇p
- That is why the pressure is sometime called a Lagrange multiplier here.

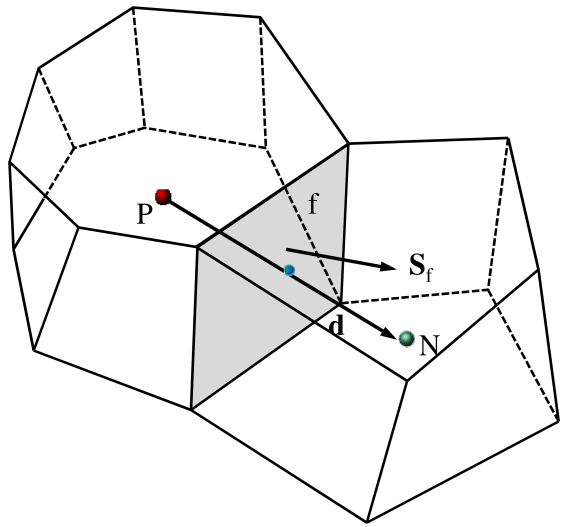
Pressure and Velocity

Derivation of the Pressure Equation:

- Substituting $\mathbf{u}_P$ into the continuity equation $\nabla \cdot \mathbf{u} = 0$, we have

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

- This is the **famous** Pressure Poisson Equation (PPE)
- To recap:
  - the momentum matrix is decomposed into the diagonal and off-diagonal parts
  - the diagonal part (from the CV) is in a_P
  - the off-diagonal part (from neighbors) is inside $\mathbf{H}(\mathbf{u}) = \mathbf{r} - \sum_N a_N \mathbf{u}_N$



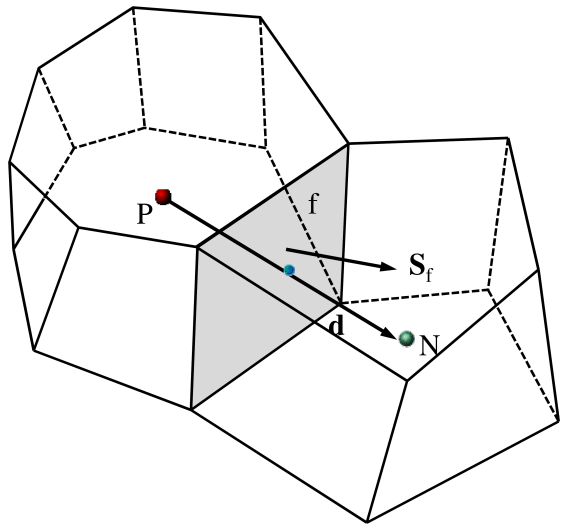
Pressure and Velocity

Conservative Fluxes

- The solution of PPE will guarantee divergence-free velocity since it is from the continuity equation
- But what exactly does **divergence-free** mean in the context of FVM?
- The continuity equation can be discretized using FVM as

$$\int_V \nabla \cdot \mathbf{u}\, dv = \int_S \mathbf{u}_f \cdot \mathbf{n}\, d\mathbf{S} = \sum_f \mathbf{s}_f \cdot \mathbf{u}_f = \sum_f F = 0$$

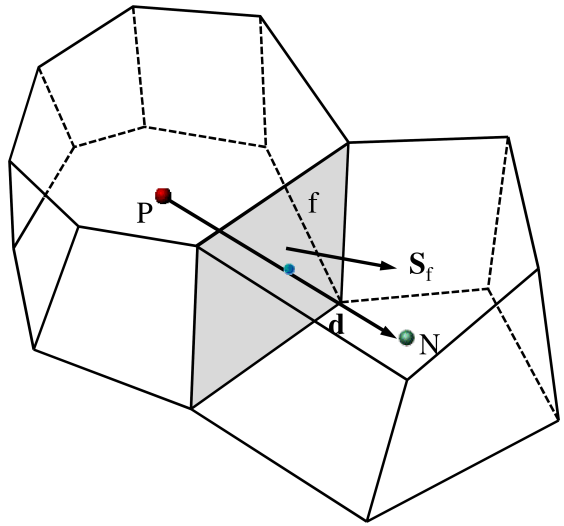
where $F = \mathbf{s}_f \cdot \mathbf{u}_f$ is the face flux, and subscript f denotes value on the face.



Pressure and Velocity

Conservative Fluxes

- The conservative flux F calls for the velocity value on the face $\mathbf{u}_f$.
- A naive way to get this face velocity is to do interpolation from cell center velocities.
- However, this is NOT the consistent way. It does NOT guarantee the divergence free condition $\sum_f F = 0$.
- A consistent way to construct the conservative flux is from the solution of PPE.



Pressure and Velocity

Conservative Fluxes:

- The descretization of PPE using FVM:

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

  Using Gauss's theorem, we get

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

- Re-arrange this equation, we get

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

- So the conservative flux should be:

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

Iterative solution algorithms

With the derived Pressure Poisson Equation (PPE), we will introduce two representative iterative solution algorithms for the N-S equations:

- SIMPLE
- PISO

SIMPLE Algorithm

- This is the earliest pressure-velocity coupling algorithm for steady problem
- Patankar and Spalding, 1972 (Imperial College London): Semi-Implicit Algorithm for Pressure-Linked Equations
- Iterative procedure:
  1. Guess the pressure field p^* (or from previous iteration)
  2. **Momentum predictor**: Solve the momentum equation using the guessed pressure.
  $$a_P \mathbf{u}_P = \mathbf{H}(\mathbf{u}) - \nabla p^*$$
  3. **Pressure correction**: Calculate the new pressure based on the predicted velocity field.
  $$\nabla \cdot \left[(a_P)^{-1} \nabla p \right] = \nabla \cdot \left((a_P)^{-1} \mathbf{H}(\mathbf{u}) \right)$$
  4. Assemble the conservative face F
  $$F = (a_P)^{-1} \mathbf{s}_f \cdot \mathbf{H}(\mathbf{u})_f - (a_P)^{-1} \mathbf{s}_f \cdot (\nabla p)_f$$
  5. Correct the velocity field using the new pressure
  $$\mathbf{u}_P = \frac{\mathbf{H}(\mathbf{u})}{a_P} - \frac{\nabla p}{a_P} \tag{4}$$
  6. Repeat to convergence

Under-Relaxation

Stability of the SIMPLE algorithm:

- The iterative algorithm (hopefully) generates converged series for $\mathbf{u}$ and p.
- However, it could diverge if the mesh quality is not good or the initial guess is far from the final solution.
- To improve the convergence, **under-relaxation** is commonly applied:

$$p^n = p^{n-1} + \alpha_P(p^{predicted} - p^{n-1})$$

and

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

where α_P and α_U are under-relaxation factors. They are usually less than unity.

Under-Relaxation

Stability of the SIMPLE algorithm:

- In practice momentum under-relaxation is implicit and pressure (elliptic equation) is under-relaxed explicitly.
- In the `simpleFoam` solver in OpenFOAM® , it has

  `UEqn.relax()`

  which does the following:

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

  It is not explicitly under-relaxed as `U.relax()`
- But for pressure, it is done explicitly `p.relax()`

Under-Relaxation

Stability of the SIMPLE algorithm:

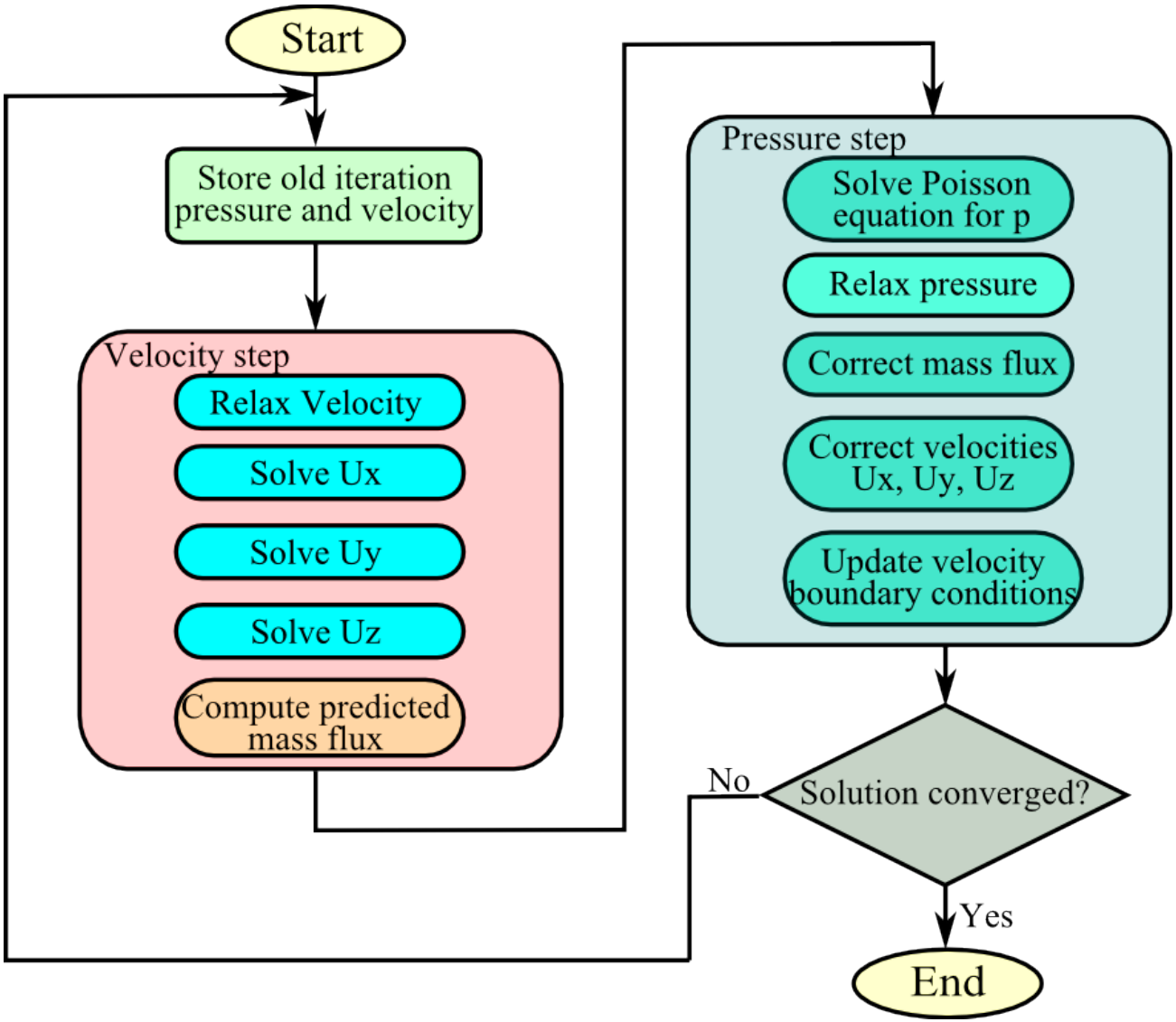
- α_P and α_U are the pressure and velocity under-relaxation factors. Some guidelines for choosing under-relaxation are

$$0 < \alpha_P \leq 1$$
$$0 < \alpha_U \leq 1$$
$$\alpha_P + \alpha_U \approx 1 or 1.1$$

- Patankar (1980) suggested: $\alpha_P = 0.5$, $\alpha_U = 0.8$
- OpenFOAM® default: $\alpha_P = 0.3$, $\alpha_U = 0.7$
- Unfortunately, the optimal values depend on the specific problem and mesh quality.
- Use with caution and always experimenting with different combinations.
- There are some researches in choosing the values automatically based on for example minimizing the global residual.

SIMPLE algorithm in OpenFOAM

The SIMPLE algorithm has been implemented in OpenFOAM® :



SIMPLE algorithm in OpenFOAM

The SIMPLE algorithm has been implemented in OpenFOAM® : `simpleFoam`:

- Store the pressure and velocity calculated at the previous iteration required for under-relaxation.
- The calling sequence is inside the *simple.loop()* function call, where *simple* is an object of *simpleControl* class. Inside loop() function:

```
storePrevIterFields();
```

- Define the equation for U

```
tmp<fvVectorMatrix> UEqn
(
    fvm::div(phi, U)
  + turbulence->divDevReff(U)
  ==
    fvOptions(U)
);
```

  tmp is used to reduce peak memory.
- Under-relax the equation for U

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

SIMPLE algorithm in OpenFOAM

- Solve the momentum predictor

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

- Calculate the a_p coefficient and calculate U

```
volScalarField rAU(1.0/UEqn().A());
volVectorField HbyA("HbyA", U);
HbyA = rAU*UEqn().H();
UEqn.clear();
```

- Calculate the flux

```
surfaceScalarField phiHbyA("phiHbyA",
                    fvc::interpolate(HbyA) & mesh.Sf());
adjustPhi(phiHbyA, U, p);
```

SIMPLE algorithm in OpenFOAM

- Define and solve the pressure equation and repeat for the prescribed number of non-orthogonal corrector steps

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

pEqn.setReference(pRefCell, pRefValue);

pEqn.solve();
```

- Correct the flux

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

- Calculate continuity errors

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

SIMPLE algorithm in OpenFOAM

- Under-relax the pressure for the momentum corrector and apply the correction

```
p.relax();
U = HbyA - rAU*fvc::grad(p);
U.correctBoundaryConditions();
```

- Check for convergence and repeat from the beginning until convergence criteria are satisfied. This check is inside the function call *simple.loop()*.

SIMPLE algorithm in OpenFOAM

What does convergence mean in the SIMPLE algorithm?

- It means both the pressure and the velocity (three components) are converged from iteration to iteration.
- The convergence criteria are specified in `controlDict`

```
SIMPLE
{
    ...
    residualControl
    {
        p               1e-5;
        U               1e-5;
        nuTilda         1e-5;
    }
}
```

- The initial residual should be below the specified tolerance for ALL variables.
- This convergence should not be confused with the convergence of the linear system solver for each of the filed variable.

SIMPLE algorithm in OpenFOAM

What does convergence mean in the SIMPLE algorithm?

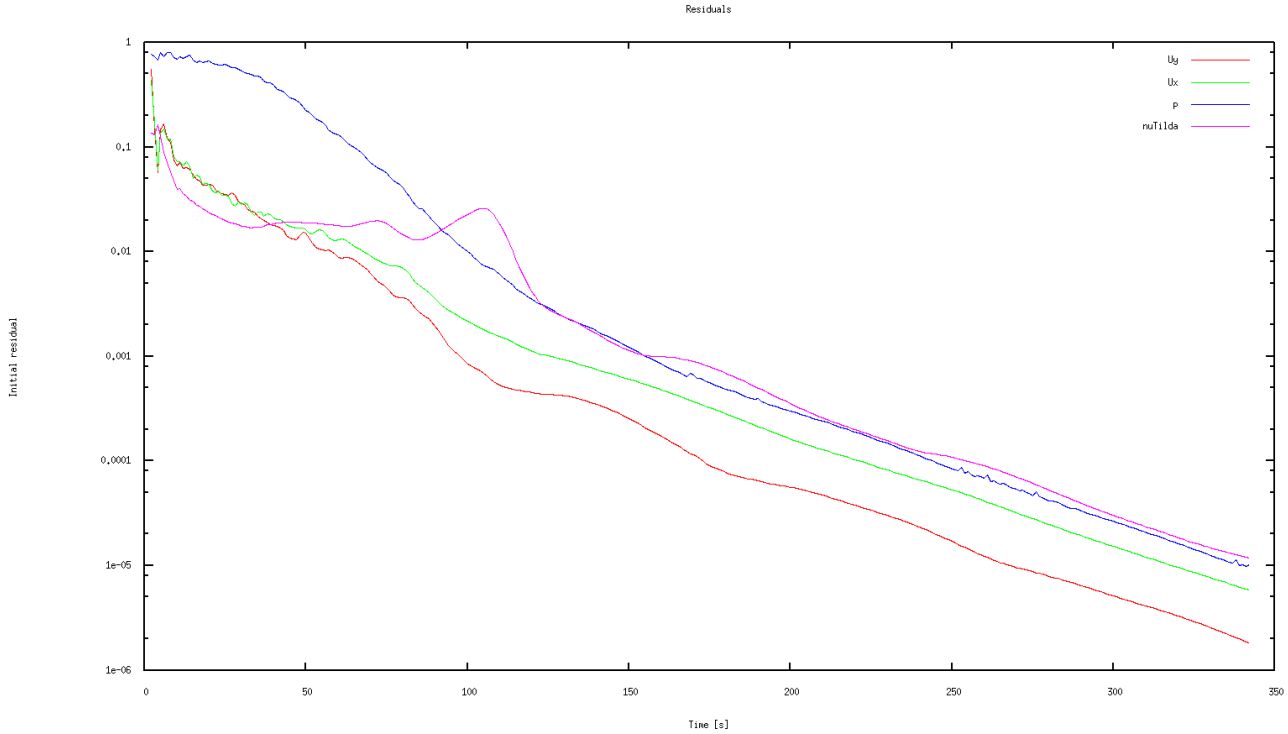


Figure: Example residual plot for SIMPLE algorithm (using PyFOAM)

PISO Algorithm

Pressure Correction Equation

- PISO: Pressure Implicit with Splitting of Operator; by Issa in 1986.
- SIMPLE algorithm:
  - Solves the momentum predictor using a guessed pressure field.
  - Then a PPE is solved to satisfy continuity.
  - But the resulted pressure is still not accurate since the r.h.s of the PPE is from an estimated velocity
  - It is like an infinity loop: that's why we need iterations.
- As a result, the pressure in SIMPLE after the first pressure corrector has two parts
  - Physical part which we need
  - A non-physical part which is used to satisfy the continuity at current iteration
- In SIMPLE, the purpose of iteration is to eliminate the non-physical part and converge to the physical solution.
- In SIMPLE iteration, the momentum equation and PPE are solved in turn; $\mathbf{u}$ and p need to be under-relaxed separately to converge.

PISO Algorithm

The idea of PISO: Freeze the momentum equation but do multiple pressure iterations

- Navier-Stokes equation system contains two couplings:
  - Non-linear $\mathbf{u} - \mathbf{u}$ coupling, i.e., convection term
  - Linear $\mathbf{u}$-p coupling

$$\nabla \cdot \mathbf{u} = 0 \tag{5}$$

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u}\mathbf{u}) = -\nabla p + \nu \nabla^2 \mathbf{u} \tag{6}$$

- PISO is based on this assumption:
  - When *Co* number (time-step) is small, the pressure velocity coupling is much stronger than the non-linear coupling
- Therefore, we can repeat a number of pressure correctors without doing momentum predictor
- In between the momentum predictors in PISO, velocity is only passively updated after each pressure corrector step
- PISO can be seen as an extension of SIMPLE, with more than one pressure correction steps to enhance the continuity.

PISO Algorithm

The idea of PISO:

- The original PISO algorithm calls for two pressure correctors.
- In reality you can have more than two. But it makes no sense to have too many pressure corrections because the momentum equation is frozen.
- Since multiple pressure correctors are performed with one momentum predictor, pressure does not need to be under-relaxed.
- On the other side, the velocity can be under-relaxed.

PISO Algorithm

PISO algorithm steps:

- **momentum predictor**: Predict velocity by solving the momentum equation with the pressure p^* from previous corrector or time step

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

- **pressure correction**: Calculate the new pressure by solving the PPE

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

- Assemble the conservative face F using the new pressure

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

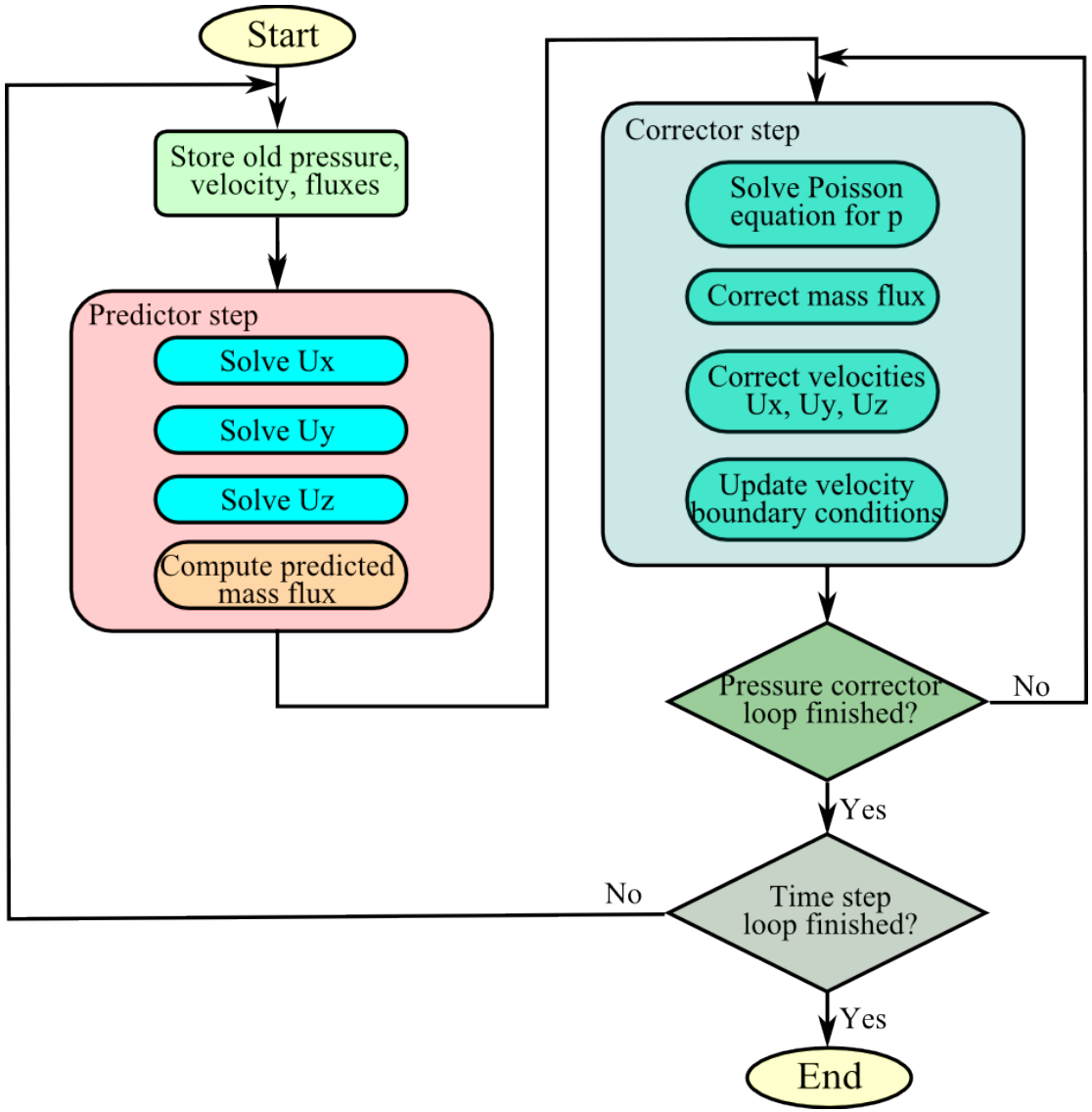
- Correct the velocity field using the new pressure

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

- Repeat from the pressure correction step for the specified PISO corrector steps and then proceed to a new time-step

PISO Algorithm

PISO algorithm scheme diagram



PISO algorithm in OpenFOAM

The PISO algorithm is implemented in OpenFOAM® as follows (Details can be found in the for example `pisoFoam` standard solver provided with OpenFOAM):

- Define the equation for U

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

- Solve the momentum predictor

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

PISO algorithm in OpenFOAM

- Calculate the a_p coefficient and calculate U

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

- Calculate the flux

```
surfaceScalarField phiHbyA
(
    "phiHbyA",
    (fvc::interpolate(HbyA) & mesh.Sf())
  + fvc::ddtPhiCorr(rAU, U, phi)
);

adjustPhi(phiHbyA, U, p);
```

PISO algorithm in OpenFOAM

- Define and solve the pressure equation and repeat for the prescribed number of non-orthogonal corrector steps

```
fvScalarMatrix pEqn
(
  fvm::laplacian(rUA, p) == fvc::div(phiHbyA)
);
pEqn.setReference(pRefCell, pRefValue);
pEqn.solve();
```

PISO algorithm in OpenFOAM

- Correct the conservative face flux from the new pressure

```
    if (nonOrth == nNonOrthCorr)
    {
        phi = phiHbyA - pEqn.flux();
    }
```

- Calculate continuity errors

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

- Explicitly correct the velocity field from the new pressure

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

- Repeat from the calculation of a_p for the prescribed number of PISO corrector steps.

PIMPLE algorithm in OpenFOAM

OpenFOAM® also comes with PIMPLE algorithm:

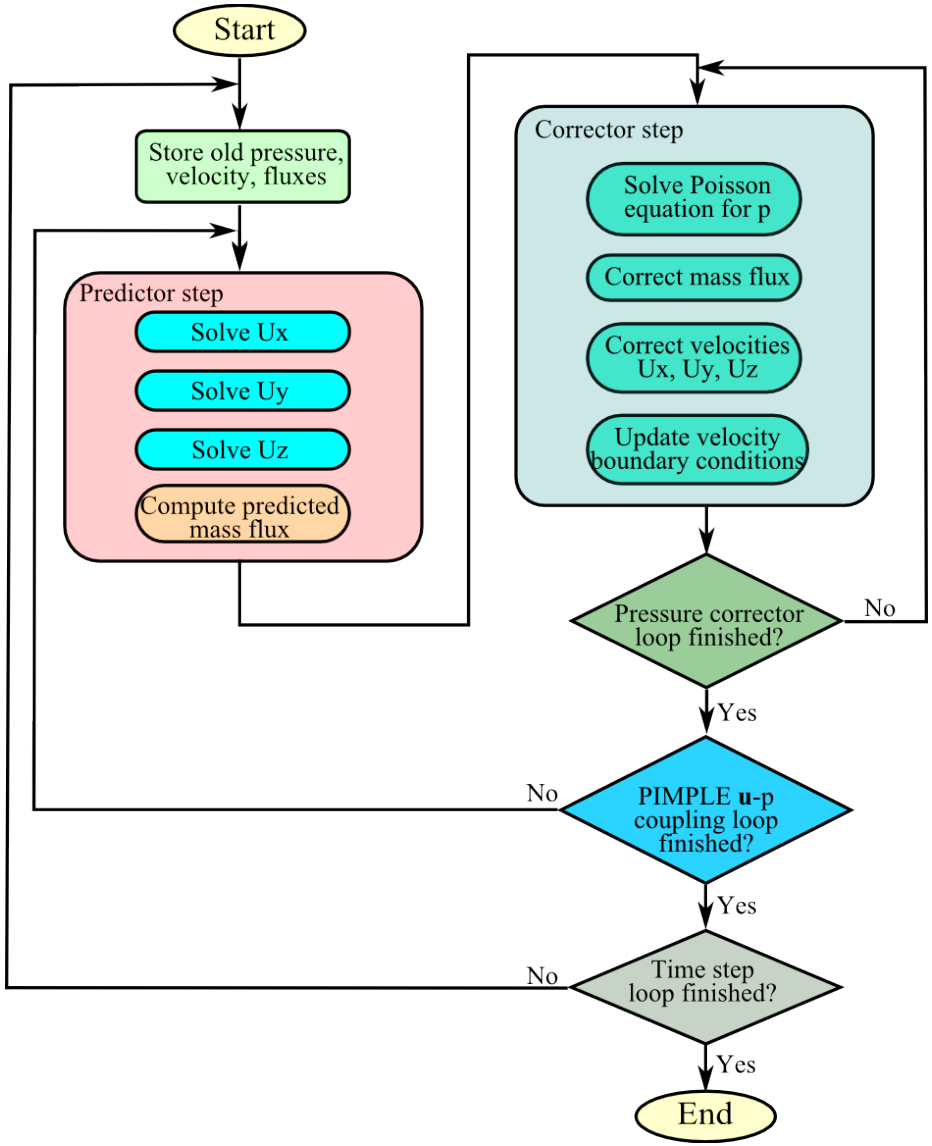
- Large time-step transient solver for incompressible, flow using the PIMPLE (merged PISO-SIMPLE) algorithm.
- It has two loops: inner (PISO corrector) and outer corrector to ensure convergence and tighter coupling

```
PIMPLE
{
    nOuterCorrectors 2;    //for outer corrector
    nCorrectors      2;    //for inner PISO corrector
    nNonOrthogonalCorrectors 0;
    ...
}
```

- As a result, the small time step limitation of PISO is relaxed.
- If `nOuterCorrectors` $= 1$, it will operate in PISO mode.

PIMPLE algorithm in OpenFOAM

OpenFOAM® also comes with PIMPLE algorithm:



Questions?