PISO in icoFoam

PISO in icoFoam

- The `icoFoam` directory (`$FOAM_SOLVERS/incompressible/icoFoam`) consists of the following:

```
createFields.H  Make/  icoFoam.C
```

- The `Make` directory contains instructions for the `wmake` compilation command.
- `icoFoam.C` is the main file, and `createFields.H` is an inclusion file, which is included in `icoFoam.C`.

We have a look at a part of the description in `icoFoam.C`:

```
Description
    Transient solver for incompressible, laminar flow of Newtonian fluids.

    The solver uses the PISO algorithm ...
```

We will here discuss the PISO algorithm in general and the way it is done in `icoFoam`, for transient incompressible laminar flow of Newtonian fluids.

PISO in icoFoam: Governing equations

- The incompressible continuity and momentum equations are given by

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

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

  The kinematic viscosity, ν is constant for Newtonian flow.
  The kinematic pressure, p, is the static pressure divided by the constant density, ρ.
  In `icoFoam` cases we don't specify ρ, so we remember that we solve for *kinematic pressure*.
  Only pressure *gradient* affects the momentum eqs., so level of pressure is not important.
  The non-linear convection term is linearized and evaluated using Gauss' theorem, as `Foam::div(phi, U)`, where the face flux field `phi` is taken from the previous time step/iteration.

- Unknowns are $\mathbf{u}$ and p, but there is no pressure equation.
  The continuity equation imposes a scalar constraint on the momentum equation (since $\nabla \cdot \mathbf{u}$ is a scalar).
  We use the continuity and momentum equations to derive a pressure equation ...

PISO in icoFoam: Derivation of the pressure equation

- Discretize the linearized momentum equation, keeping the pressure gradient in its original form:

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

  Here, r is a source term (may have contributions from the discretized time-term).

- Introduce the $\mathbf{H}(\mathbf{u})$ operator:

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

  so that:

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

- Substitute this in the incompressible continuity equation ($\nabla \cdot \mathbf{u} = 0$) to get a pressure equation for incompressible flow:

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

  The R.H.S. is the continuity error of the velocity field without the pressure gradient.
  It is given by summing the fluxes through the faces, interpolated to the faces.

PISO in icoFoam: General algorithm (1/2)

(Issa, 1986; description from Versteeg and Malalasekera)

The original PISO algorithm consists of one predictor step and two corrector steps.
Here it is described with some details how it is done in OpenFOAM.

- Predictor step:
  - **Solve the discretized momentum equations using a guessed/intermediate pressure field to get intermediate velocity fields.**
    Remember: The face flux field `phi` fulfils continuity, but is frozen at the old time step.
- Corrector step 1:
  - The intermediate velocity fields will not fulfil continuity unless the guessed/intermediate pressure field used in the predictor step is correct. In OpenFOAM the pressure field should fulfil the pressure equation, derived using both the momentum and continuity equations. I.e.:, the first corrector step will:
    Solve the pressure equation and correct the velocities, including a consistent correction of the face flux field `phi`, which also represents the velocity field.
    Note: In the original algorithm there is a pressure *correction* equation (rather than a pressure equation), followed also by a pressure correction.

Continued ...

PISO in icoFoam: General algorithm (2/2)

(Issa, 1986; description from Versteeg and Malalasekera)

... continued

- Corrector step 2:
  - The first corrector step gives a new intermediate velocity field that satisfies continuity. However, the pressure equation was solved using the intermediate velocity field that did not fulfil continuity. The second corrector step should give the correct pressure field, based on a conservative velocity field, i.e.:
    Update and solve the discretized pressure equation again, using the original contributions from the discretized momentum equations, but using the updated velocities. Follow up with corrections of the velocity and face flux fields, as in the first corrector step.
    Note: The original PISO algorithm rather derives a second pressure correction equation, and also corrects the pressure a second time.

In the non-iterative PISO algorithm the velocity and pressure fields are considered solved after the two corrector steps (for sufficiently small time steps, i.e. tiny). In the iterative PISO algorithm the entire predictor-corrector procedure is repeated until convergence. This is necessary for large time steps (larger than tiny) or if the algorithm is used for steady-state.

PISO in icoFoam: Time loop

We will now have a look at how the PISO algorithm is implemented in `icoFoam.C`, looking only at the most important parts of the code. We will have a look at all of the code later.

A global perspective of the time loop, including only important parts, is given by (c.f. `pisoFoam.C`):

```
while (runTime.loop())
{
    {
        #include "UEqn.H"
        while (piso.correct())
        {
            #include "pEqn.H"
        }
    }
}
```

Here `UEqn.H` corresponds to the momentum predictor, and `pEqn.H` corresponds to the corrector step(s). The user can choose the number of corrector steps. In `cavity/system/fvSolution`:

```
PISO { nCorrectors     2; }
```

This corresponds to the two corrector steps discussed before.

PISO in icoFoam: Momentum predictor

The momentum predictor step (`UEqn.H`) is implemented as:

```
fvVectorMatrix UEqn
(
    fvm::ddt(U)
  + fvm::div(phi, U)
  - fvm::laplacian(nu, U)
);
if (piso.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));
}
```

The convective term is linearized using face flux field `phi`, from previous time step.
The pressure gradient is excluded from `fvVectorMatrix UEqn`, since we later need to ask `UEqn` for the $\mathbf{H}(\mathbf{u})$ operator *without the pressure gradient*.
The user can choose if the momentum predictor should be done. It is done by default, but a switch can be added in `cavity/system/fvSolution`:

```
PISO { momentumPredictor   true; //false; //on; //off; }
```

PISO in icoFoam: Corrector step(s) (1/4)

The first part of the corrector step (`pEqn.H`) is implemented as:

```
while (piso.correct())
{
    volScalarField rAU(1.0/UEqn.A());
    volVectorField HbyA(constrainHbyA(rAU*UEqn.H(), U, p));
    surfaceScalarField phiHbyA
    (
        "phiHbyA",
        fvc::flux(HbyA)
      + fvc::interpolate(rAU)*fvc::ddtCorr(U, phi)
    );

    adjustPhi(phiHbyA, U, p);

    // Update the pressure BCs to ensure flux consistency
    constrainPressure(p, U, phiHbyA, rAU);

    //// CONTENTS IN NEXT SLIDE ////
}
```

- Calculate velocity field without pressure gradient, `HbyA`, as $(a_P^{\mathbf{u}})^{-1}\mathbf{H}(\mathbf{u})$ with corrected bc's (`constrainHbyA`). The `UEqn` discretization is used (`UEqn.A()` and `UEqn.H()`).
- Calculate face flux of `HbyA` (with time scheme correction of $(a_P^{\mathbf{u}})^{-1}$).
- Enforce global conservation of `phiHbyA` and coherent pressure bc's.
- Continued ...

PISO in icoFoam: Corrector step(s) (2/4)

The second part of the corrector step (`pEqn.H`) is implemented as:

```
while (piso.correct())
{
    //// CONTENTS IN PREVIOUS SLIDE ////
    while (piso.correctNonOrthogonal())
    {
        fvScalarMatrix pEqn
        (
            fvm::laplacian(rAU, p) == fvc::div(phiHbyA)
        );
        pEqn.setReference(pRefCell, pRefValue);
        pEqn.solve(mesh.solver(p.select(piso.finalInnerIter())));
        if (piso.finalNonOrthogonalIter())
        {
            phi = phiHbyA - pEqn.flux();
        }
    }
    #include "continuityErrs.H"
    U = HbyA - rAU*fvc::grad(p);
    U.correctBoundaryConditions();
}
```

- Use the discretized mom. eq. and intermediate velocity field to solve the pressure equation: $\nabla \cdot \left[(a_P^{\mathbf{u}})^{-1} \nabla p\right] = \nabla \cdot \left[(a_P^{\mathbf{u}})^{-1} \mathbf{H}(\mathbf{u})\right]$
- Correct the face fluxes consistent with the discretized pressure equation.
- Correct the velocity field and make sure that the velocity bc's are still as set in the case.
- Continued ...

PISO in icoFoam: Corrector step(s) (3/4)

The second part of the corrector step (`pEqn.H`) is implemented as:

```
while (piso.correct())
{
    //// CONTENTS IN PREVIOUS SLIDE ////
    while (piso.correctNonOrthogonal())
    {
        fvScalarMatrix pEqn
        (
            fvm::laplacian(rAU, p) == fvc::div(phiHbyA)
        );
        pEqn.setReference(pRefCell, pRefValue);
        pEqn.solve(mesh.solver(p.select(piso.finalInnerIter())));
        if (piso.finalNonOrthogonalIter())
        {
            phi = phiHbyA - pEqn.flux();
        }
    }
    #include "continuityErrs.H"
    U = HbyA - rAU*fvc::grad(p);
    U.correctBoundaryConditions();
}
```

- The user can specify a number of non-orthogonal corrector steps in `cavity/system/fvSolution`:
  `PISO { nNonOrthogonalCorrectors 0; }`
  OpenFOAM has a few *explicit* implementations of discretization, such as for non-orthogonal correction and higher-order schemes. Iterations are needed to take those fully into account.
- Continued ...

PISO in icoFoam: Corrector step(s) (4/4)

The second part of the corrector step (`pEqn.H`) is implemented as:

```
while (piso.correct())
{
    //// CONTENTS IN PREVIOUS SLIDE ////
    while (piso.correctNonOrthogonal())
    {
        fvScalarMatrix pEqn
        (
            fvm::laplacian(rAU, p) == fvc::div(phiHbyA)
        );
        pEqn.setReference(pRefCell, pRefValue);
        pEqn.solve(mesh.solver(p.select(piso.finalInnerIter())));
        if (piso.finalNonOrthogonalIter())
        {
            phi = phiHbyA - pEqn.flux();
        }
    }
    #include "continuityErrs.H"
    U = HbyA - rAU*fvc::grad(p);
    U.correctBoundaryConditions();
}
```

- Why is `phi` not corrected outside the while loop, instead of at the final loop inside the loop? Hint: Scope of variables.
- In the next slide the consistent `phi` flux correction (`pEqn.flux()`) is derived.

PISO in icoFoam: Conservative face fluxes

(Acknowledgements to Professor Hrvoje Jasak)

- Here we derive the conservative face fluxes used in `pEqn.flux()` in the previous slide.
- Discretize the continuity equation:

$$\nabla \cdot \mathbf{u} = \sum_f \mathbf{s}_f \cdot \mathbf{u} = \sum_f F$$

where F is the face flux, $F = \mathbf{s}_f \cdot \mathbf{u}$.

- Substitute the expression for the velocity ($\mathbf{u}_P = (a_P^{\mathbf{u}})^{-1}(\mathbf{H}(\mathbf{u}) - \nabla p)$), yielding

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

- The first term on the R.H.S. appears during the discretization of the pressure Laplacian ($\nabla \cdot \left[(a_P^{\mathbf{u}})^{-1}\nabla p\right]$), for each face:

$$(a_P^{\mathbf{u}})^{-1}\mathbf{s}_f \cdot \nabla p = (a_P^{\mathbf{u}})^{-1}\frac{|\mathbf{s}_f|}{|\mathbf{d}|}(p_N - p_P) = a_N^P(p_N - p_P)$$

where $|\mathbf{d}|$ is the distance between the owner and neighbour cell centers, and $a_N^P = (a_P^{\mathbf{u}})^{-1}\frac{|\mathbf{s}_f|}{|\mathbf{d}|}$ is the off-diagonal matrix coefficient in the pressure Laplacian. For the fluxes to be fully conservative, they must be completely consistent with the assembly of the pressure equation (i.e. non-orthogonal correction).

PISO in icoFoam: Rhie & Chow interpolation

(Acknowledgements to Dr. Fabian Peng-Kärrholm and Professor Hrvoje Jasak)

- When using a colocated FVM formulation it is necessary to use a special interpolation to avoid unphysical pressure oscillations.
- OpenFOAM uses an approach 'in the spirit of Rhie & Chow', but it is not obvious how this is done. Fabian presents a discussion on this in his PhD thesis, and here is the summary of the important points:
  - In the explicit source term `fvc::div(phiHbyA)` of the pressure equation, `phiHbyA` does **not include any effect of the pressure**.
  - `rAU` does **not include any effect of pressure** when solving the pressure equation and finally correcting the velocity.
  - The Laplacian term, `fvm::laplacian(rAU, p)`, of the pressure equation uses the value of the gradient of p on the cell faces. The gradient is calculated using **neighbouring cells**, and not neighbouring faces.
  - `fvc::grad(p)` is calculated **from the cell face values** of the pressure.