Advanced Process Simulation

CFD - Analysis of numerical solvers in openFOAM Extend 4.0

Written by Amos Fang

December 01, 2017

Computational Fluid Dynamics

- Difference between FEM and FVM
- Describe the linear solvers used for mesh computation
- Difference between the coupled and segregated solver
- Walkthrough the steps for pUCoupled solver using the `BlockLduMatrix` class
- Test the convergence for the coupled solver in openFoam Extend which uses the `fvMatrix` class
- Run convergence results comparison for the SIMPLE and pUCoupled foam solver cases

1 Computational Fluid Dynamics

Computational fluid dynamics is the use of computer-based simulations to analyze the profile of a system involving fluid flow, heat transfer or other associated phenomena. CFD analysis provides an understanding of the system of interest without having to carry out physical experiments that may incur high costs.

In this report, the goal is to compare two solver approaches to achieve solution convergence, with regard to time taken and performance (number of iterations required). The coupled solvers are available only in openFOAM Extend.

1.1 Numerical Solutions of partial differential equations

There are many commercial software packages used in CFD simulations, each employing different numerical approaches to perform computational modelling. The classification of these modelling tools are [1],

1) Finite Element Method
2) Finite Volume Method
3) Finite Difference Method
4) Spectral Methods

As the objective for each numerical modelling project differs widely from each other, different techniques as listed, are used to discretize the governing equations. Both the finite difference and spectral methods are out of scope in this discussion. The focus will be on the finite volume method.

Popular commercial software packages such as ANSYS Fluent uses the finite element method (FEM) for computer-aided simulations. The open source C++ toolbox used to develop customized numerical solvers, openFOAM, uses the finite volume method. The distinguishing feature of openFOAM is the object-oriented programming and operator overloading feature available in C++ to allow custom solvers be built.

1.2 Finite Volume Method

The accurate approximation of solutions to the governing equations describing the system is the goal of every simulation effort. Solution discontinuities lead to loss of numerical accuracy in traditional finite difference approaches. Accuracy is affected in domains where there are discontinuities in the differential equations and the solution does not hold (Figure 1).

In contrast with finite difference methods which make pointwise approximation at grid points after discretizing the PDEs, the finite volume method takes the integral form of the governing equations. The domain is divided into grid cells and the flux quantity q is approximated by taking the total integral of q over the cell volume (3D) or area (2D) to obtain the average $\bar{q}$ value. The average $\bar{q}$ is modified for each time step by the flux through the edges of each grid cell [2].

$$\frac{\partial}{\partial t}\int_{\Omega} q \; dx = f(q(x_2, t)) - f(q(x_1, t)) \tag{1}$$

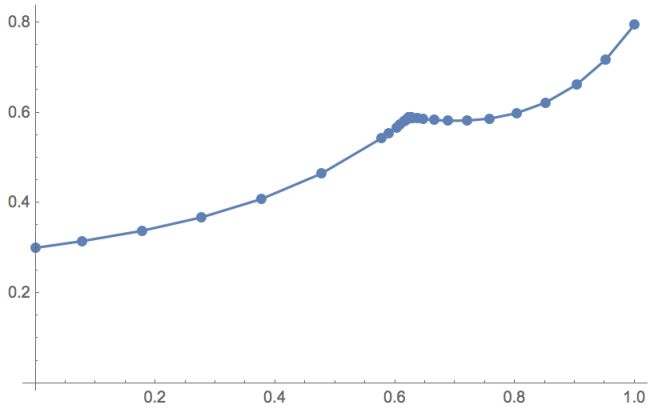

Figure 1: Discontinuity in numerical solutions (Source: *Wolfram Mathematics*)

where the grid cell region $\Omega \in [x_1, x_2]$. q is the flux quantity that is dependent on both location and time.

In the 3D simulation case, the Reynolds Transport equations may be written for the arbitrary conserved flux quantity q.

$$\frac{\partial}{\partial t}\int_{\Omega_0} q(\mathbf{x},t)\ d\mathbf{x} = \frac{\partial}{\partial t}\int_{\Omega} q(\mathbf{x},t)\ d\mathbf{x} + \int_{\partial\Omega} q(\mathbf{x},t)\ \mathbf{u}\cdot\mathbf{n}\ d\sigma \tag{2}$$

where $\mathbf{x}$ is the 3D domain space, σ is the surface area of the grid cell, Ω_0 and $\partial\Omega_0$ are the cell volume and boundary respectively at time 0. By divergence theorem, the outward flux through closed surface in Equation (2) may be written as a volume integral of the divergence of the flux in the closed region.

$$\underbrace{\frac{\partial}{\partial t}\int_{\Omega_0} q(\mathbf{x},t)\ d\mathbf{x}}_{\mathbf{S}} = \frac{\partial}{\partial t}\int_{\Omega} q(\mathbf{x},t)\ d\mathbf{x} + \cancel{\frac{\partial}{\partial t}}\int_{\Omega} \nabla\cdot\mathbf{q}(\mathbf{x},t)\ d\mathbf{x}\ \cancel{dt} \tag{3}$$

This conservation principle is applied to obtain numerical solutions in the finite volume method. If the flux quantity is not conserved, the equation must also contain source terms [2]. In Equation (4), the left hand side term is the source term, followed by the transient and divergence term on the right hand side.

By shrinking the control volume $\Omega \to 0$, the differential form of the conservation law is obtained.

$$\mathbf{S} = \frac{\partial q(\mathbf{x},t)}{\partial t} + \nabla\cdot\mathbf{q}(\mathbf{x},t) \tag{4}$$

In the finite volume method, the cell averaged quantity $q_i(t)$ is determined,

$$q_i(t) = \frac{1}{V_i}\int_{V_i} q(\mathbf{x},t)\ d\mathbf{x} \tag{5}$$

For cells with fixed number of faces, the volume averaged conservation law is [1],

$$\frac{\partial q_i}{\partial t} + \frac{1}{V_i}\sum_p \mathbf{q}_p \cdot \mathbf{n} = \frac{1}{V_i}\int_{V_i} \mathbf{S}\ d\mathbf{x} \tag{6}$$

1.3 C++ Implementation in openFOAM

All libraries in openFOAM and openFOAM Extend are written in C++.

Use of template metaprogramming is prevalent in openFOAM in performing operations on fields and matrices. Metaprogramming means a program within "a program that manipulates code" [3]. A natural question is what is the benefit of metaprogramming in CFD computations? For matrix setup in the solver, function templates are written so that the same function can be used on **different data types**, because the matrix setup in openFOAM can be a vector, matrix or tensor with different sizes. This cannot be achieved with traditional programming that uses function overloading for different operation scenarios. Function overloading used in normal programming styles may be impractical with the variability and possibly large number of operations used in openFOAM computations.

In general, template metaprogramming expands on traditional programming styles to include custom data type for functions and classes. Metaprogramming styles allow computation to be done at run-time instead of compile time. To illustrate a widely used example in openFOAM, a technical explanation of how a wrapper class `tmp<class-name>` reduces peak memory for large tensorial objects is found in the link [2]. This may theoretically reduce computational burden by implementing safe memory management algorithm using `tmp<function-or-class-name>`.

Experienced foamers have advised users against making changes to the existing templates due to unknown dependencies. This is due to the lack of openFOAM Extend documentation and template revisions with each new update. But one with knowledge of these templates, could build a custom matrix structure for new solver algorithms. In the 2012 project report written by Klas Jareteg from Chalmers, he has created a `pUCoupledFoam` solver using the `BlockLduMatrix` **block coupled matrix structure** with the full code found in the Appendix C of his report [4]. This solver has, however, not been able to run on the latest version of openFoam Extend 4.0 because the code has not been maintained with software version updates over the years.

The solver is found in "`foam-extend-4.0>applications>solver>SOLVER_NAME`". The example for `simplefoam` is illustrated. Together with the solver `main` routine "`simplefoam.c`", the equation files such as "`UEqn.h`" and other files used by `simplefoam` are located in this directory. The code snippet for equation input in "`UEqn.h`" for the momentum equation, Equation (7), is shown in this example,

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \phi\mathbf{U} - \nabla \cdot \nu\nabla\mathbf{U} = -\frac{\nabla p}{\rho} \tag{7}$$

[1] Discretization of FVM - `https://perswww.kuleuven.be/~u0016541/Talks/finvol.pdf`

[2] `http://openfoamwiki.net/index.php/Snip_tmp_explained`

```
    tmp<fvScalarMatrix> UEqn
(
      fvm::ddt(U)
    + fvm::div(phi,U)
    - fvm::laplacian(nu,U)
   ==
    - fvc::grad(p)
);
```

In this solver case, `fvm` and `fvc` is the namespace for the finite volume method and calculus respectively. `fvm` is used for implicit discretization methods whereas `fvc` refers to explicit discretization typically used to solve the source terms. In this example, within the finite volume method namespace, the functions `ddt()`, `div()` and `lapacian()` are used for the respective mathematical operations. The `fvc::grad()` function is used for treatment of the source term.

Namespaces are widely used in openFOAM to prevent variable or function name conflicts [3]. For example, both `vector3::zero` and `vector4::zero` use the variable `zero` but they are not the same as the `vector3` is a zero vector size 3 and size 4 for `vector4`. This should not be confused with class member function specification in object oriented programming given that both cases use the `::` operator.

2 Numerical solution methods in openFOAM Extend

In a given CFD problem, the system is described by one or many governing equations. The number of unknown variables must be equal to the number of governing equations for a solution to be found for the problem. In the computational domain, each positional coordinate is a cell in a discretized mesh where the solution is found. The goal of the CFD exercise is to compute the solution vector for the cell meshes generated for the case geometry.

In the traditional direct solution method, a matrix with non-zero diagonals $\mathbf{A}$ is solved via a linear equation system $\mathbf{Ax} = \mathbf{b}$ [4] [5]. For each diagonal value in a_{ij} where $i = j$, the value of x_j is approximated until the solution converges. The Jacobi method solves for the ith equation in the matrix system in (9). This is done iteratively until all the variables in x_j is solved. For this discussion, $\mathbf{a_P}$ is used to denote the diagonal element in the $\mathbf{A}$ matrix which is the coefficient of the point cell of interest.

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \tag{8}$$

$$\mathbf{a}_\mathrm{p}\mathbf{x}_\mathrm{p} = \mathbf{b} \tag{9}$$

For almost all CFD cases, the system is more complicated and is described by multiple governing equations[5]. Thus, there are more than one unknown variables to be solved for in each grid cell. Multiple matrix

[3] Slide 220 of C++ Introduction to openFOAM (`http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2010/basicsOfC++.pdf`)

[4] The Jacobi Method - `http://mathworld.wolfram.com/JacobiMethod.html`

[5] also known as field equations

systems are used to solve these multiple equations. These matrix systems can either be implicit or explicit. In the explicit case, the solution variables are independent of each other at that given time step t_n. This solution vector at t_n is solved from that of the previous time step t_{n-1}. For implicit systems, the variables are dependent on each other at the given time step t_n.

The momentum equation, as mentioned in Equation (7), has two unknown solution variables $\mathbf{U}$ and p. p is linearly dependent on $\mathbf{U}$, suggesting linear coupling in this equation.

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \phi \mathbf{U} - \nabla \cdot \nu \nabla \mathbf{U} = -\frac{\nabla p}{\rho} \tag{10}$$

The continuity equation is,

$$\frac{\partial \rho \mathbf{U}}{\partial t} + \nabla \cdot \rho \mathbf{U} = 0 \tag{11}$$

2.1 Segregated Approach

In the segregated approach, one equation of the multiple matrix system is solved at a given time. Intuitively, that means the solution for the time step t_n will have to be found by some kind of iteration scheme; there are two unknowns that has to be solved for given time step t_n and only one value is predicted at a time.

In the segregated case, two matrix systems for a point cell in a given mesh with two coupled unknown variables are solved [4].

$$\mathbf{A}(p)\ \mathbf{U} = \mathbf{a} \tag{12}$$
$$\mathbf{B}(\mathbf{U})\ p = \mathbf{b} \tag{13}$$

As there is implicit coupling between the two variables, solving both at once in a combined matrix system in Equation (14) (sparse matrix) does not take into account of the coupling.

$$\begin{bmatrix} \mathbf{A}(p) & 0 \\ 0 & \mathbf{B}(\mathbf{U}) \end{bmatrix} \begin{bmatrix} \mathbf{U} \\ p \end{bmatrix} = \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix} \tag{14}$$

This leads to the segregated solver algorithm. The iterative procedure is carried out for each time step t_n. In openFOAM, a solver algorithm known as SIMPLE (Semi-Implicit Method for Pressure Linked Equations) is used for pressure-velocity computation. According to *Wikipedia*, this algorithm was developed by Prof Brian Spalding from Imperial College in the 1970s, and is widely used in numerical procedure of the Navier Stokes Equation.

The following steps are taken to analyze the SIMPLE algorithm for the pressure and velocity coupled equation system. The steps evaluated are taken with reference to the SIMPLE algorithm in openFOAM [6] [6].

[6] `http://openfoamwiki.net/index.php/OpenFOAM_guide/The_SIMPLE_algorithm_in_OpenFOAM`

2.1.1 Momentum equation

```
1     tmp<fvVectorMatrix> HUEqn
2     (
3         fvm::div(phi, U)
4       + turbulence->divDevReff()
5     );
```

The left-hand-side of the momentum equation (15) is defined, with addition of the turbulence modeling structure.

$$H(\mathbf{U}) = \nabla \cdot \phi\mathbf{U} + \frac{\partial \mathbf{U}}{\partial t} - \nabla \cdot \nu \nabla \mathbf{U} \tag{15}$$

Note that the $H(\mathbf{U})$ reference from Section 2.1.5 is the discretized form of the momentum equation, where it is a matrix.

2.1.2 Apply under-relaxation factor

```
1     const scalar UUrf = mesh.solutionDict().equationRelaxationFactor(U.name());
```

The under-relaxation factor for **U** is obtained from the user specified `relaxationFactor` parameter in the `fvSolution` file in the case directory [7]. Relaxation factors RF are used to maintain solution stability. An under-relaxation factor of less than 1 may stabilize the solution and prevent overshooting from the true solution[8] [6]. `UUrf` for this case stands for velocity underrelaxation factor.

2.1.3 Solve the momentum equation

```
1     solve
2     (
3         relax(HUEqn(), UUrf)
4      ==
5         -fvc::grad(p)
6     );
```

In this segment, pressure is computed. The left-hand-side of the momentum equation is solved for with the relaxation factor such that it equates to the source term `grad(p)`. **Pressure is first predicted by solving the momentum equation with a suitable relaxation factor.**

2.1.4 Update boundary conditions for pressure

```
1     p.boundaryField().updateCoeffs();
```

The boundary conditions for p in each iteration are updated.

[7] A case directory is where the case files such as the mesh, controlDict and schemes for the given case are stored.

[8] Convergence and relaxation discussion post - (https://www.researchgate.net/post/Anyone_familiar_with_convergence_and_under_relaxation_factors_in_fluent)

2.1.5 a_p and $\mathbf{U_p}$ are computed

```
    // Prepare clean 1/Ap without contribution from under-relaxation
    volScalarField rUA
    (
        "(1|A(U))",
        1/HUEqn().A()
    );

    // Store velocity under-relaxation point before using U for
    // the flux precursor
    U.storePrevIter();
```

Both $\mathbf{U_p}$ and the coefficient a_p are calculated for the next iteration step. $\mathbf{U_P}$ corresponds to the velocity field of the point cell and $\mathbf{a_p}$ refers to the diagonal coefficient of the $\mathbf{A}$ matrix ($\mathbf{p}$ refers to the point cell location).

$$a_p \mathbf{U}_p = H(\mathbf{U}) - \nabla p \tag{16}$$

$$\mathbf{U_p} = \frac{H(\mathbf{U})}{a_p} - \frac{\nabla p}{a_p} \tag{17}$$

where the discretized momentum equation is [9],

$$H(\mathbf{U}) = -\sum_n a_n \mathbf{U}_n + \underbrace{\frac{\mathbf{U}}{\Delta t}}_{\text{transient term}} \tag{18}$$

H contains the matrix coefficients of the neighbouring cells multiplied by their velocity components plus the transient term, as written in Equation (18).

2.1.6 Interpolate to compute the field

```
    U = rUA*HUEqn().H();
    HUEqn.clear();
    phi = fvc::interpolate(U) & mesh.Sf();
    adjustPhi(phi, U, p);
```

The field ϕ is interpolated at each face of the cell mesh. `rUA` is the inverse of the coefficient $\mathbf{A}$ matrix for the $\mathbf{U}$ equation system.

The discretized form of the continuity equation is expressed as the sum of flux through each face of the cell mesh,

$$\nabla \cdot \mathbf{U_p} = \sum_f \mathbf{S} \cdot \mathbf{U_f} = 0 \tag{19}$$

[9] note that $H(\mathbf{U})$ is the linearized equation of H coefficient matrix times the $\mathbf{U}$ vector

where $\mathbf{S}$ is the normal vector at each face of the mesh and $\mathbf{U_f}$ is the velocity component at each face of the mesh.

2.1.7 Interpolate to compute p in pressure correction equation (26)

```
    // Non-orthogonal pressure corrector loop
    while (simple.correctNonOrthogonal())
    {
        fvScalarMatrix pEqn
        (
            fvm::laplacian(rUA, p) == fvc::div(phi)
        );

        pEqn.setReference(pRefCell, pRefValue);

        pEqn.solve();

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

At this step, the continuity equation is substituted into the momentum equation.

Using the similar analogy of velocity at the point cell in equation (17), the velocity at each face (f) of the mesh is,

$$\mathbf{U_f} = \left(\frac{H(\mathbf{U})}{a_p}\right)_f - \left(\frac{\nabla p}{a_p}\right)_f \tag{20}$$

Substitute (20) into (19),

$$\mathbf{U_f} = \left(\frac{H(\mathbf{U})}{a_p}\right)_f - \left(\frac{\nabla p}{a_p}\right)_f \tag{21}$$

$$\nabla \cdot \mathbf{U_p} = \sum_f \mathbf{S} \cdot \left(\left(\frac{H(\mathbf{U})}{a_p}\right)_f - \left(\frac{\nabla p}{a_p}\right)_f\right) = 0 \tag{22}$$

$$\nabla \cdot \mathbf{U_p} = \nabla \cdot \left(\frac{H(\mathbf{U})}{a_p} \right) - \frac{\nabla^2 p}{a_p} = 0 \tag{23}$$

$$\nabla \cdot \left(\frac{H(\mathbf{U})}{a_p} \right) - \frac{\nabla^2 p}{a_p} = \sum_f \mathbf{S} \cdot \left(\left(\frac{H(\mathbf{U})}{a_p} \right)_f - \left(\frac{\nabla p}{a_p} \right)_f \right) \tag{24}$$

$$\Rightarrow \nabla \cdot \left(\frac{H(\mathbf{U})}{a_p} \right) = \sum_f \mathbf{S} \cdot \left(\left(\frac{H(\mathbf{U})}{a_p} \right)_f \right) \tag{25}$$

$$\Rightarrow \frac{\nabla^2 p}{a_p} = \sum_f \mathbf{S} \cdot \left(\left(\frac{H(\mathbf{U})}{a_p} \right)_f \right) \tag{26}$$

Solve the pressure equation (26) and repeat for the prescribed number of corrector steps to obtain the corrected pressure value.

2.1.8 Continuity errors

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

Compute for continuity errors.

2.1.9 Apply momentum correction

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

// Momentum corrector
// Note: since under-relaxation does not change aU, H/a in U can be
// re-used.
U = UUrf*(U - rUA*fvc::grad(p)) + (1 - UUrf)*U.prevIter();
U.correctBoundaryConditions();
```

Compute the velocity for time step t_n with the underrelaxation factor (27). A similar analogy for `rUA*fvc::grad(p)` is like performing $\mathbf{A}^{-1}\mathbf{b}$ to find the solution at that iteration step. `U - rUA*fvc::grad(` can be thought as the $\mathbf{U}$ correction term before the underrelaxation factor is applied. Correct the boundary conditions velocity $\mathbf{U}$ with each iteration. The predicted $\mathbf{U}$ value at t_n step is computed in Equation (27)

$$\mathbf{U}_n = RF_U * \mathbf{U}_n + (1 - RF_U) * \mathbf{U}_{n-1} \tag{27}$$

2.1.10 Apply turbulence correction

```
turbulence->correct();
```

Similarly, correct the parameters for the turbulence property.

2.1.11 Conclusion

Check for convergence and repeat from the beginning until convergence criteria are satisfied. The result at each iteration for the variables, $\mathbf{U}$ and p, is the sum of the predicted and the corrected value.

There are three equations that are iterated to find the corrected values. They are the pressure, momentum(velocity) field and continuity equations. The correction of the turbulence parameters is more complicated, and will not be discussed here. In general, iteration uses more computational steps to achieve convergence and could use more computational time for some grids or cases [4]. In the next section, we will understand how the coupled system may be described by a block coupled matrix form used in numerical solution.

2.2 Block Coupled Approach

In the block coupled equation system, the solution vector is augmented to include all the variables in the point cell and neighboring cells as arranged in Figure 2 [10].

However, the matrix system in Equation (14) does not take into account of the implicit coupling between the two unknown variables. In the block coupled approach, off-diagonal terms are introduced to remove the explicit linear dependence of p from $\mathbf{A}$ and similarly for $\mathbf{B}$. The resulting matrix system from Equation (14) becomes,

$$\begin{bmatrix} \mathbf{A}' & \mathbf{A}_p \\ \mathbf{B}_\mathbf{U} & \mathbf{B}' \end{bmatrix} \begin{bmatrix} \mathbf{U} \\ p \end{bmatrix} = \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix} \tag{28}$$

where $\mathbf{a}$ and $\mathbf{b}$ hold the source terms.

When finite volume discretization is applied on the block coupled equation set, the solution vector in the point cell P is dependent on both the vector components in its cell as well as those of its neighboring cells N.

$$a_P \ \mathbf{x_P} + \sum_N a_N \ \mathbf{x_N} = \mathbf{b} \tag{29}$$

Alternatively, the block coupled matrix system for one point cell can be written as a tensorial product,

$$\mathbf{C}_\mathrm{P} \ \mathbf{z} = \mathbf{c} \tag{30}$$

$$\begin{bmatrix} c^{\mathrm{U,U}} & c^{\mathrm{U},p} \\ c^{p,\mathrm{U}} & c^{p,p} \end{bmatrix}_\mathrm{P} \begin{bmatrix} \mathbf{U} \\ p \end{bmatrix} = \begin{bmatrix} b^1_\mathrm{P} \\ b^2_\mathrm{P} \end{bmatrix} \tag{31}$$

[10] Block Coupled Simulations using openFOAM Slide 5- `http://www.personal.psu.edu/dab143/OFW6/Training/clifford_slides.pdf`

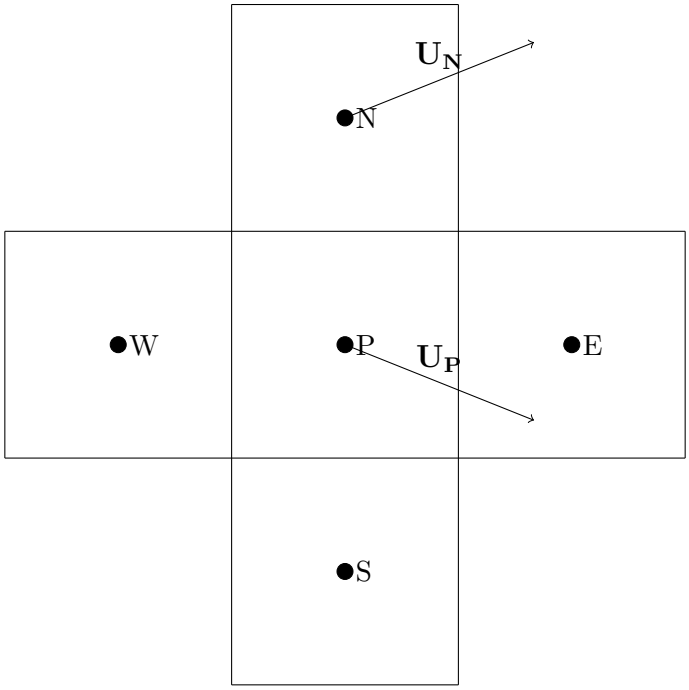

Figure 2: 2D finite volume discretization of block coupled equation set

$$\mathbf{C_P} = \begin{bmatrix} c^{\mathrm{U,U}} & c^{\mathrm{U},p} \\ c^{p,\mathrm{U}} & c^{p,p} \end{bmatrix}_{\mathrm{P}} \tag{32}$$

$$\mathbf{z} = \begin{bmatrix} \mathbf{U} \\ p \end{bmatrix} \tag{33}$$

For a block mesh with a point cell P, east E and west W cells, the full tensorial product can be written as,

$$\begin{bmatrix} & \ddots & & & \\ \begin{pmatrix} c^{\mathrm{U,U}} & c^{\mathrm{U},p} \\ c^{p,\mathrm{U}} & c^{p,p} \end{pmatrix}_{\mathrm{E}} & \cdots & \begin{pmatrix} c^{\mathrm{U,U}} & c^{\mathrm{U},p} \\ c^{p,\mathrm{U}} & c^{p,p} \end{pmatrix}_{\mathrm{P}} & \cdots & \begin{pmatrix} c^{\mathrm{U,U}} & c^{\mathrm{U},p} \\ c^{p,\mathrm{U}} & c^{p,p} \end{pmatrix}_{\mathrm{W}} \\ & & & \ddots & \end{bmatrix} \begin{bmatrix} \begin{pmatrix} \mathbf{U} \\ p \end{pmatrix}_{\mathrm{E}} \\ \vdots \\ \begin{pmatrix} \mathbf{U} \\ p \end{pmatrix}_{\mathrm{P}} \\ \vdots \\ \begin{pmatrix} \mathbf{U} \\ p \end{pmatrix}_{\mathrm{W}} \end{bmatrix} = \begin{bmatrix} \begin{pmatrix} b^1_{\mathrm{E}} \\ b^2_{\mathrm{E}} \end{pmatrix}_{\mathrm{E}} \\ \vdots \\ \begin{pmatrix} b^1_{\mathrm{P}} \\ b^2_{\mathrm{P}} \end{pmatrix}_{\mathrm{P}} \\ \vdots \\ \begin{pmatrix} b^1_{\mathrm{W}} \\ b^2_{\mathrm{W}} \end{pmatrix}_{\mathrm{W}} \end{bmatrix} \tag{34}$$

Both the tensor matrices of the North and South neighbouring cells are omitted in (34).

The momentum and continuity equations are restated for easy reference to the steps taken to solve the block coupled system.

Momentum Equation

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \phi \mathbf{U} - \nabla \cdot \nu \nabla \mathbf{U} = -\frac{\nabla p}{\rho} \tag{35}$$

Continuity Equation

$$\frac{\partial \rho \mathbf{U}}{\partial t} + \nabla \cdot \rho \mathbf{U} = 0 \tag{36}$$

The `pUCoupledFoam` in Foam Extend 4.0, as attached in the Appendix Section, 5.2 is broadly summarized below,

- Initialize block matrix system, `Up`
- Find the explicit discretization (`fvc`) of the divergence of the field.
- Define the momentum equation like in the segregated solver case.
- Find the inverse of the $\mathbf{A}$ matrix for the $\mathbf{U}$ equation.
- openFOAM manages the convective term $\nabla \cdot \phi \mathbf{U}$ implicitly with the `fvm::SuSp(-divPhi, U)` term.
- When the momentum equation is constructed with the implicit terms, the relaxation factor is applied and the equation is stored in the `Up` block matrix structure.
- The pressure parts of the continuity equation are set up and stored in the `Up` block matrix.
- Assemble and insert the coupling terms, ∇p and $\nabla \cdot \mathbf{U}$.
- Solve the block coupled matrix system.
- The solution is transferred from the coupled solution vector to separate field and the boundary conditions are updated.
- The turbulence parameters are solved for and correction is applied with each iteration.
- Repeat cycle until convergence is reached.

2.3 Caveats

The SIMPLE foam is one of the most basic solver built in openFOAM. Even SIMPLE foam is updated with every version (last update was in 2016) and users should understand that the updates may affect their project that is compiled in a previous version of openFOAM Extend (no technical support is provided for software issues). Also, cases built with openFOAM may not work on openFOAM Extend as they are two separate entities developing newer versions of their software. If required to work across software platforms, cases will have to be reconfigured. Looking up at the openFOAM documentation on a particular solver can provide some insights on the algorithm but may not give you an accurate understanding about how your solver actually works in the current version. Also note there are no documentations for coupled solvers, which were developed solely on openFOAM Extend.

```
        // Two equivalent equations defined differently in openFOAM and openFOAM Extend

        // openFoam
        tmp<fvVectorMatrix> UEqn
    (
      fvm::div(phi, U) - fvm::laplacian(nu, U)
    );

        // openFoam Extend
        tmp<fvVectorMatrix> HUEqn
    (
        fvm::div(phi, U)
      + turbulence->divDevReff()
    );

    // In openFOAM Extend, turbulence -> divDevReff is

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

For this example, ν is the kinematic viscosity of the fluid which appears in the diffusive term of the momentum equation. The diffusive term is defined in the turbulence modeling function in openFOAM Extend.

3 Relaxation Factor and Convergence Criteria

The following recommendations on the selection of relaxation factor and convergence criteria in this section is obtained from ResearchGate [6].

3.1 Relaxation and Convergence Criteria

CFD numerical solvers typically use one or more iteration procedures to achieve the convergence criteria. These iteration methods are also often used with relaxation procedures. Under-relaxation is used to achieve numerically stable results when all the flow equations are implicitly coupled together. An example of implicitly coupled equations is the coupled pressure and velocity variables found in the momentum and continuity equations as explained in the earlier parts. Over-relaxation[11] speeds up convergence of pressure-velocity iteration to satisfy an incompressible flow condition.

There are three ways to define the convergence criteria in openFOAM, like in most CFD numerical solver packages.

- Absolute tolerance: This is the minimum residual value we want to achieve at the end of the iterations. Iterations stop when the residuals fall below this value.

[11] not covered in the scope of this report

- Relative tolerance: This tolerance is multiplied by the initial residual. When the current residual is lower than this value, the solver stops iterating.
- Maximum number of iterations: This is the maximum number of iterations the solver will perform regardless whether convergence is achieved.

3.2 Choosing Relaxation Criteria

The amount of over or under-relaxation can affect numerical computation results. Too much relaxation can cause numerical instabilities, but too little could slow down convergence. A poorly chosen convergence criteria can also lead to either poor results (too loose) or excessive computational times (too tight).

Finding the sweet spot for the relaxation factor and convergence criteria can be a difficult task in CFD simulations. There are no heuristics or formula to choose these parameters. Though the cases in openFOAM have preset relaxation factors and convergence criteria, trial-and-error adjustments are usually made to find the best results possible.

4 Applying `pUCoupledFoam` to pitzDaily tutorial case

To compare the performance of the 2D block coupled and segregated solvers, the `pitzDaily` case was configured to run under as similar conditions as possible for both solvers.

Table 1: Segregated solver configuration

	Foam Extend 4.0 `simpleFoam`
Solver (Pressure)	PCG
Preconditioner (Pressure)	DIC
Solver (Others)	BiCGStab
Preconditioner (Others)	DILU
Convergence criteria	1×10^{-9}

The coupled solver configuration implemented in Foam Extend 4.0 is compared with the version created by Jareteg in 2012 in Table 2.

Table 2: 2D block coupled solver configuration comparison

	Foam Extend 4.0 `pUCoupledFoam`	**2012 `blockLduMatrix` `pUCoupledFoam`**
Solver	GMRES	GMRES
Preconditioner	Cholesky	Cholesky
Convergence criteria	1×10^{-9}	1×10^{-9}
Krylov space dimension (`nDirections`)	5	5
Max iterations	300	10
Underelaxation p	0.7	1.0
Underelaxation U	0.7	1.0
Underelaxation k	0.7	0.7
Underelaxation ϵ	0.7	0.7

As shown in Table 2, the max iterations for the current simulations of the `pUCoupledFoam` was attempted at 300 for the GMRES solver, an increase from 10 used in the 2012 version because poor solution was obtained for the block coupled solver when the iteration was kept at 10. This has resulted in significantly more time taken than `simpleFoam` to complete the simulation cycle.

5 Appendix

5.1 pUCoupledFoam

The purpose of this section is to conduct a gap analysis of the pUCoupledFoam solver written by Klas Jareteg in openFoam Extend 1.6 and the default pUCoupledFoam found in the latest release of openFoam Extend 4.0.

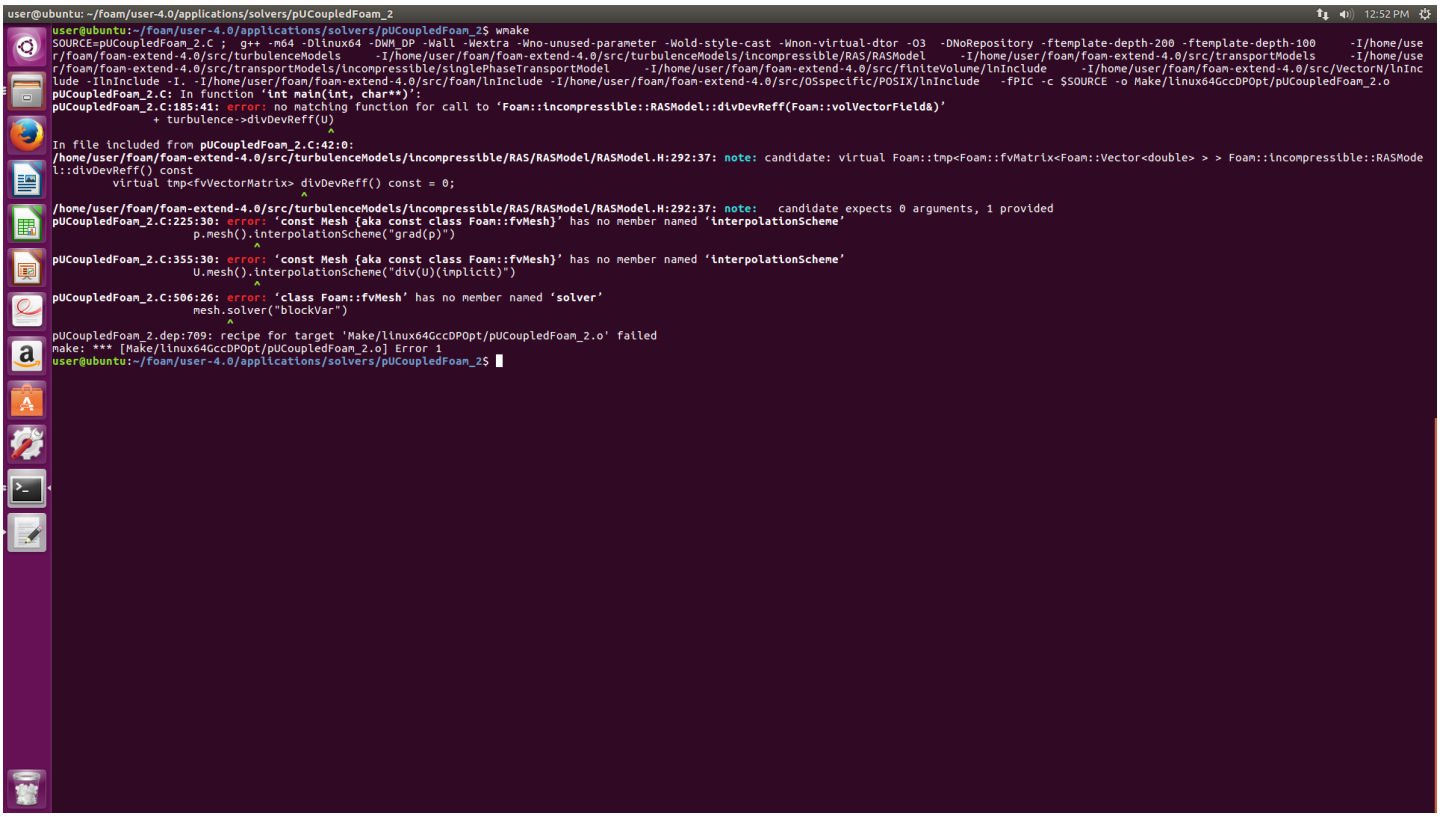

Figure 3: Compiling Klas Jareteg pUCoupledFoam on openFoam Extend 4.0 produced errors

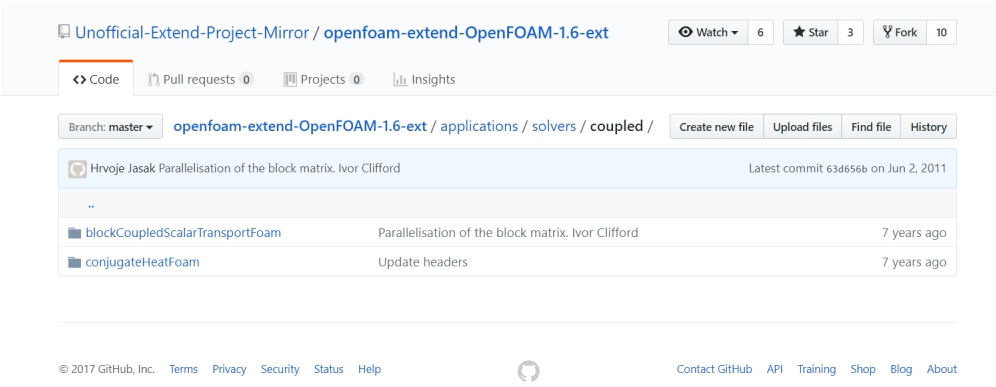

Figure 4: `coupled` solver directory in openFOAM Extend 1.6

It was also found that he developed the `pUCoupledFoam` during the period when the solver was not available in the openFOAM Extend 1.6. Therefore, this leads to the question whether the `pUCoupledFoam` using the `blockLdUMatrix` he has built performs similarly as the default one now available in openFoam Extend 4.0. This will require an analysis of how the `pUCoupledFoam` works in openFOAM Extend 4.0. The following header files are added in `pUCoupledFoam` in Extend version 4.0.

```
#include "fvCFD.H"
#include "fvBlockMatrix.H"
#include "singlePhaseTransportModel.H"
#include "RASModel.H"
```

In the "`fvBlockMatrix.H`" file, the following header files were added.

```
#include "BlockLduSystem.H"
#include "fvMatrices.H"
#include "blockLduSolvers.H"
```

Therefore, the conclusion is that there is a high possibility that the block coupled approach used `pUCoupledFoam` in Extend version 4.0 performs similarly as the one which was written in 2012 as both used the `BlockLduSolvers` approach. However, Jareteg's work in 2012 could be a source of inspiration for others to build new CFD coupled solvers as his code follows through the steps of executing a coupled solver using more basic openFoam functions. It is difficult to use the existing default solvers as an analogy to create new ones as the developer has in some cases, created very specialized and high level foam functions for a single purpose. There are also parts in the default solver that are still under development or are not efficient or flexible. Since the scope of this report is to conduct research on solvers in openFOAM, the technical aspects of creating a new coupled solver shall not be pursued.

5.2 **pUCoupledFoam** from openFOAM Extend 4.0

```
    /*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | foam-extend: Open Source CFD
   \\    /   O peration     | Version:     4.0
    \\  /    A nd           | Web:         http://www.foam-extend.org
     \\/     M anipulation  | For copyright notice see file Copyright
-------------------------------------------------------------------------------
License
    This file is part of foam-extend.

    foam-extend is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation, either version 3 of the License, or (at your
    option) any later version.

    foam-extend is distributed in the hope that it will be useful, but
    WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with foam-extend.  If not, see <http://www.gnu.org/licenses/>.

Application
    pUCoupledFoam

Description
    Steady-state solver for incompressible, turbulent flow, with implicit
    coupling between pressure and velocity achieved by fvBlockMatrix.
    Turbulence is in this version solved using the existing turbulence
    structure.

Authors
    Klas Jareteg, Chalmers University of Technology,
    Vuko Vukcevic, FMENA Zagreb.

\*---------------------------------------------------------------------------*/

#include "fvCFD.H"
#include "fvBlockMatrix.H"
#include "singlePhaseTransportModel.H"
#include "RASModel.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

int main(int argc, char *argv[])
{

#   include "setRootCase.H"
```

```
#    include "createTime.H"
#    include "createMesh.H"
#    include "createFields.H"
#    include "initContinuityErrs.H"
#    include "initConvergenceCheck.H"

    Info<< "\nStarting time loop\n" << endl;
    while (runTime.loop())
    {
#        include "readBlockSolverControls.H"
#        include "readFieldBounds.H"

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

        p.storePrevIter();

        // Initialize the Up block system (matrix, source and reference to Up)
        fvBlockMatrix<vector4> UpEqn(Up);

        // Assemble and insert momentum equation
            volScalarField divPhi
    (
        "divPhi",
        fvc::div(phi)
    );

    // Momentum equation
    {
    fvVectorMatrix UEqn
    (
        fvm::div(phi, U)
      + turbulence->divDevReff()
    );

    rAU = 1.0/UEqn.A();

    // Insert the additional components. Note this will destroy the H and A

    UEqn += fvm::SuSp(-divPhi, U) + divPhi*U;
    UEqn.relax();

    UpEqn.insertEquation(0, UEqn);

    }
        // Assemble and insert pressure equation

        surfaceScalarField presSource
(
    "presSource",
    fvc::interpolate(rAU)*
    (fvc::interpolate(fvc::grad(p)) & mesh.Sf())
);
```

```
102
103 fvScalarMatrix pEqn
104 (
105   - fvm::laplacian(rAU, p)
106  ==
107   - fvc::div(presSource)
108 );
109
110 pEqn.setReference(pRefCell, pRefValue);
111
112 UpEqn.insertEquation(3, pEqn);
113
114
115         // Assemble and insert coupling terms
116         {
117     // Calculate grad p coupling matrix. Needs to be here if one uses
118     // gradient schemes with limiters.  VV, 9/June/2014
119     BlockLduSystem<vector, vector> pInU(fvm::grad(p));
120
121     // Calculate div U coupling.  Could be calculated only once since
122     // it is only geometry dependent.  VV, 9/June/2014
123     BlockLduSystem<vector, scalar> UInp(fvm::UDiv(U));
124
125     // Last argument in insertBlockCoupling says if the column direction
126     // should be incremented. This is needed for arbitrary positioning
127     // of U and p in the system. This could be better. VV, 30/April/2014
128     UpEqn.insertBlockCoupling(0, 3, pInU, true);
129     UpEqn.insertBlockCoupling(3, 0, UInp, false);
130 }
131
132
133         // Solve the block matrix
134         maxResidual = cmptMax(UpEqn.solve().initialResidual());
135
136         // Retrieve solution
137         UpEqn.retrieveSolution(0, U.internalField());
138         UpEqn.retrieveSolution(3, p.internalField());
139
140         U.correctBoundaryConditions();
141         p.correctBoundaryConditions();
142
143         phi = (fvc::interpolate(U) & mesh.Sf()) + pEqn.flux() + presSource;
144
145 #       include "continuityErrs.H"
146
147 {
148     // Bound the pressure
149     dimensionedScalar p1 = min(p);
150     dimensionedScalar p2 = max(p);
151
152     if (p1 < pMin || p2 > pMax)
153     {
```

```
154             Info<< "p:␣" << p1.value() << "␣" << p2.value()
155                 << ".␣␣Bounding." << endl;
156 
157             p.max(pMin);
158             p.min(pMax);
159             p.correctBoundaryConditions();
160         }
161 
162         // Bound the velocity
163         volScalarField magU = mag(U);
164         dimensionedScalar U1 = max(magU);
165 
166         if (U1 > UMax)
167         {
168             Info<< "U:␣" << U1.value() << ".␣␣Bounding." << endl;
169 
170             volScalarField Ulimiter = pos(magU - UMax)*UMax/(magU + smallU)
171                 + neg(magU - UMax);
172             Ulimiter.max(scalar(0));
173             Ulimiter.min(scalar(1));
174 
175             U *= Ulimiter;
176             U.correctBoundaryConditions();
177         }
178     }
179 
180             p.relax();
181 
182             turbulence->correct();
183             runTime.write();
184 
185             Info<< "ExecutionTime␣=␣" << runTime.elapsedCpuTime() << "␣s"
186                 << "␣␣ClockTime␣=␣" << runTime.elapsedClockTime() << "␣s"
187                 << nl << endl;
188 
189             // Check convergence
190     if (maxResidual < convergenceCriterion)
191     {
192         Info<< "reached␣convergence␣criterion:␣" << convergenceCriterion << endl;
193         runTime.writeAndEnd();
194         Info<< "latestTime␣=␣" << runTime.timeName() << endl;
195     }
196         }
197 
198         Info<< "End\n" << endl;
199 
200         return 0;
201     }
```

5.3 Configuration of **fvSolution** for simpleFoam

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.0                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
    p
    {
        solver           PCG;
        preconditioner   DIC;
        tolerance        1e-06;
        relTol           0.01;
    }
    U
    {
        solver           BiCGStab;
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0.1;
    }
    k
    {
        solver           BiCGStab;
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0.1;
    }
    epsilon
    {
        solver           BiCGStab;
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0.1;
    }
    R
    {
        solver           BiCGStab;
        preconditioner   DILU;
```

```
        tolerance        1e-05;
        relTol           0.1;
    }
    nuTilda
    {
        solver           BiCGStab;
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0.1;
    }
}

SIMPLE
{
    nNonOrthogonalCorrectors 0;

    residualControl
    {
        p               1e-9;
        U               1e-9;
        "(k|epsilon)"   1e-9;
    }
}

relaxationFactors
{
    fields
    {
        p               0.7;
    }

    equations
    {
        U               0.7;
        k               0.7;
        epsilon         0.7;
        R               0.7;
        nuTilda         0.7;
    }
}

cache
{
    grad(U);
    grad(p);
    grad(k);
    grad(omega);
    grad(epsilon);
}
// ************************************************************************* //
```

5.4 Configuration of pUCoupledFoam **fvSolution**

```
/*---------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.0                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
    Up
    {
        solver GMRES;
        preconditioner Cholesky;

        tolerance 1e-09;
        relTol  0.0;

        minIter 1;
        maxIter 300;
        nDirections 5;
    }

    p
    {
        solver           PCG;
        preconditioner   DIC;
        tolerance        1e-06;
        relTol           0.01;
    }
    U
    {
        solver           BiCGStab;
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0.1;
    }
    k
    {
        solver           BiCGStab;
        preconditioner   DILU;
        tolerance        1e-05;
```

```
        relTol          0.1;
    }
    epsilon
    {
        solver          BiCGStab;
        preconditioner  DILU;
        tolerance       1e-05;
        relTol          0.1;
    }
    R
    {
        solver          BiCGStab;
        preconditioner  DILU;
        tolerance       1e-05;
        relTol          0.1;
    }
    nuTilda
    {
        solver          BiCGStab;
        preconditioner  DILU;
        tolerance       1e-05;
        relTol          0.1;
    }
}

blockSolver
{
    convergence 1e-6;

    pRefCell 0;
    pRefValue 0;
}

fieldBounds
{
    p      -5e4 5e4;
    U      500;
}
relaxationFactors
{

    equations
    {
        p               0.7;
        U               0.7;
        k               0.7;
        epsilon         0.7;
        R               0.7;
        nuTilda         0.7;
    }
}
```

References

[1] T. Gerya, *Introduction to Numerical Geodynamic Modeling*. Cambridge University Press, 2010.

[2] R. J. LeVeque, "Finite Volume Methods for Hyperbolic Problems (Cambridge Texts in Applied Mathematics)," 2002.

[3] B. D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. 2004.

[4] K. Jareteg, "Block coupled calculations in OpenFOAM," p. 52, 2012.

[5] E. W. Black, Noel; Moore, Shirley; Weisstein, "Jacobi Method."

[6] ResearchGate, "Convergence and Relaxation Factor in ANSYS FLUENT."