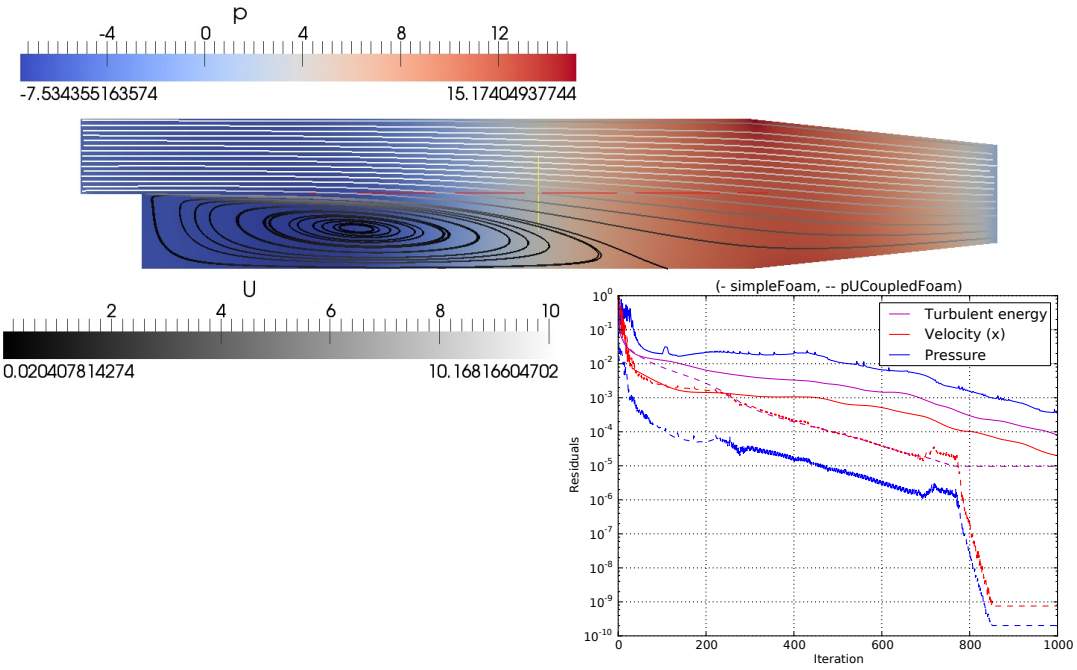
CHALMERS

CFD with OpenSource software 2012
Project 3

Block coupled calculations in OpenFOAM



Klas Jareteg

October 19, 2012

Block coupled calculations in OpenFOAM
Project within course: CFD with OpenSource software
Chalmers University of Techonology, 2012

Author:
Klas Jareteg
Department of Applied Physics
Division of Nuclear Engineering
Chalmers University of Technology
Email: klas.jareteg@chalmers.se
WWW: `http://klas.nephy.chalmers.se`

Abstract

This report describes the basic formulation and the OpenFOAM implementation of a block coupled solver strategy for finite volume calculations. In order to understand the implemented solvers, the existing block matrix class in OpenFOAM 1.6-ext is explained and exemplified, including the preserved sparsity structure, block matrix assembling and block matrix solvers.

The possibilities of the block coupled solution strategy is exemplified with the implementation of a pressure-velocity incompressible coupled steady-state solver. The newly implemented solver corresponds to the existing segregated solver `simpleFoam`. The implementation of the coupled solver requires not only a theoretical model but also a deeper insight in to the discretization procedure of OpenFOAM, which is described briefly and is of interest not only for the purpose of the block coupled solver, but also for general OpenFOAM knowledge.

The newly implemented solver is compared to the existing solver `simpleFoam`. The two dimensional version of the solver shows a major convergence rate increase both considering convergence per iteration and convergence per elapsed CPU time. The three dimensional solver is less beneficial, which is however partly an effect of the segregated turbulence model.

Cover: Pressure and velocity solution of the `pitzDaily` case with a comparative graph for the convergence of the coupled solver as compared to `simpleFoam`.

Contents

1 Introduction and overview

Many type of fluid mechanics problems and other partial differential equation (PDE) problems requires the solution of multiple coupled equations. Examples includes the Navier-Stokes equations, multi-group neutron kinetics and many more. Such problems have traditionally been solved in a segregated manner, resolving the couplings iteratively.

Alternative to the segregated solvers more integrated solvers, with implicit coupling between different variables, possibly allows faster convergence and faster algorithms. The coupled solver can potentially avoid resolving the coupled equations in an iterative manner. The general formulation block coupled system is given in section 2.

In order to utilize a such approach, solver routines and structures for a block coupled matrix format is necessary. One such coupled framework is the block coupled solver available with OpenFOAM 1.6-ext (`BlockLduMatrix`). This report is aimed at describing the idea and theory behind the block coupled solver available in OpenFOAM 1.6-ext, complemented by an example of an implementation of a pressure-velocity coupled solver. The basic structure of the block coupling in OpenFOAM is given in section 5.

In order not to duplicate the code existing in OpenFOAM, the use of the block coupled solver strategy should be complemented by the use of the existing OpenFOAM operators (including discretizations and mathematical operators) which can still be used for the major parts of the matrix assembling and discretization. Not least, the existing boundary conditions are general enough to serve for the implementation of the block coupled approach. Some notes on the OpenFOAM matrix structure, discretization, assembling and solving used in the implementation is described in section 4.

The previous mentioned parts are the foundation for the implementation of the pressure-velocity coupled solver which is the major focus in this work and described in section 6. A steady-state incompressible solver, utilizing existing turbulence modeling structure, has been implemented using the block coupling approach. The solver is benchmarked against the existing `simpleFoam` solver. The comparisons and benchmarking are described in section 7.

Final conclusions and outlook are given in section 8 where ideas of parallelization and further developments are discussed.

Throughout the report discussions of the OpenFOAM library is exemplified with code snippets. This is meant to encourage the interested reader to continue study the code, to get a deeper understanding of the discussed structures and implementations. Studying the code is the way to understand and be able to use the code, and also the way to get new ideas of what could be used for the work at hands.[1]

[1] In general the code snippets are taken from a late (2012) Git version of OpenFOAM-1.6-ext, and thus the included line numbers are primarily valid for this version of the code

2 Block coupled systems

To understand the basic idea of a block coupled solver, consider the analogy with even more basic CFD procedures. In general, fluid dynamics systems are described by PDEs with one ore more unknown fields (e.g. pressure) to be determined in some vector space (for pressure: four coordinates, including position (x,y,z) and time (t)). In order to determine the unknown field in a computational manner the vector space is discretized. For the positional coordinates this corresponds to a discretized mesh with cells. The differential equation is then discretized on this mesh, giving a set of coupled linear or non-linear equations. In the case of linear or linearized systems, such systems are usually not solved equation by equation but in a matrix system. This often allows a faster convergence through the use of more sophisticated mathematical techniques (as compared to direct solution using e.g. Jacobi iteration).

In the same manner, facing a problem described by multiple field equations we can instead of solving a matrix for one variable at a time solve multiple matrix systems describing the solution of all variables through all equations at once. Solving the equations one at a time (segregated approach) forces explicit coupling between the variables, and thus some iteration technique must be applied to resolve the coupling between the fields. If instead assembling a single matrix with all fields and all equations, implicit dependencies between the coupled equations are allowed.

2.1 Formulation

In mathematical notation let us consider a problem with two coupled unknown scalar fields. In the segregated solver we would solve two problems:

$$\mathbf{A}(y)x = a \tag{1}$$

$$\mathbf{B}(x)y = b \tag{2}$$

where $\mathbf{A}$ and $\mathbf{B}$ corresponds to matrices with sources a and b for unknowns x and y. Since the system is coupled $\mathbf{A}$ will depend on the present values of y even though y is not solved for, and vice versa for B. Solving the system both equations at once would correspond to:

$$\begin{bmatrix} \mathbf{A}(y) & 0 \\ 0 & \mathbf{B}(x) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} \tag{3}$$

This would however not help, as there is still no coupling between the equations in the formed matrix system. What is interesting in the coupled approach is the off-diagonal matrix positions (0 in eq. (3)). By removing the explicit linear dependencies of y from $\mathbf{A}$ and placing it in the off-diagonal position, an implicit dependence on y is achieved. Such that:

$$\begin{bmatrix} \mathbf{A}' & \mathbf{A}_y \\ \mathbf{B}_x & \mathbf{B}' \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} \tag{4}$$

where $\mathbf{A}'$ is the $\mathbf{A}$ matrix with the explicit linear y-dependence removed. Instead the matrix $\mathbf{A}_y$ is used to implicitly resolve the dependence on y.

Another way to formulate this system is:

$$\mathbf{C}z = c \tag{5}$$

where instead each element of the matrix $\mathbf{C}$ and the vectors c and z consist of a tensor of four elements and vectors of two elements respectively, i.e. in tensor notation:

$$\mathbf{C} = C_{i,j} = \begin{bmatrix} c_{i,j}^{a,a} & c_{i,j}^{a,b} \\ c_{i,j}^{b,a} & c_{i,j}^{b,b} \end{bmatrix}_{i,j} \tag{6}$$

$$c = c_i = \begin{bmatrix} a_i & b_i \end{bmatrix}^\top \tag{7}$$

$$z = z_i = \begin{bmatrix} x_i & y_i \end{bmatrix}^\top \tag{8}$$

In this way a system equivalent to eq. (4) can be formed. The structure of eq. (5) is what will be referred to as a block coupled system, as each element in the vectors and matrices consists of a block of elements.

2.2 Non-linear dependencies

At this point it must be noticed that a coupled approach formulated through the use of a block structure, although resolving linear couplings between the equations, still not resolves the non-linear couplings. One important example of this includes the momentum equation in the Navier-Stokes equations:

$$\nabla \cdot (\mathbf{U}\mathbf{U}) - \nabla(\nu \nabla \mathbf{U}) = -\frac{1}{\rho}\nabla p \tag{9}$$

If one considers this as an equation for velocity it is seen that there is a linear dependence (although through the gradient operator) on the pressure field. The linear dependence on the pressure can, and will in

3

6.1, be taken care of by the block coupling. However, there is also a non-linearity in the equation through the convective term which can not be resolved by the linear block coupling approach. In order to get a fully implicit method for this kind of linearity a non-linear solution strategy must be applied, which is not usually done in CFD.

3.1 Advantages and drawbacks

The primer benefit sought with a coupled equation system as exemplified above is to achieve faster convergence of the problem. As the newly formed coupled matrix will be larger the system will take longer time to solve, but if less (or no) iterations are needed the larger system can still be beneficial.

One major advantage of the block coupled approach as shown in eq. (5) is that the matrix sparsity structure will not change. That is the matrix will not have more elements, although each element consists of a vector (or more general a tensor).

The new formulation might change the matrix conditioning and the convergence properties other than the convergence rate. It is thus not given that the approach will work for all kinds of problems and space discretizations.

4 OpenFOAM matrix structure, assembling and solving

As already stated in the introduction it is a primary aim of this work to use as much of the existing OpenFOAM structure as possible. Thus, some notes on the existing matrix and discretization framework in OpenFOAM is given to help the reader with the background necessary for the implementation of the pressure-velocity coupled solver, described in section 6. Some of what is described is also described in the programmer's guide[1], which is the recommended starting point for learning the internals of OpenFOAM. The below description of OpenFOAM code is by no means meant to be complete, rather aimed at pointing out some needed parts of the OpenFOAM structure, as well as directions to basic classes studied for this work. The experienced foamer can likely skip this section.

4.1 `polyMesh`

The OpenFOAM mesh class `polyMesh` contains basic data of the mesh. The finite volume implementation in OpenFOAM is strictly face based, meaning that the computational cell consists of two cells (owner and neighbour), sharing a face, as is shown in figure 1. This is of importance to understand the discretization schemes, which will accordingly work on face shared by a cell pair.

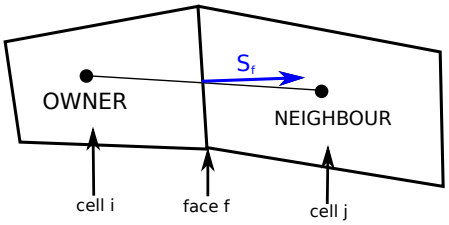


Figure 1: The computational cell of OpenFOAM, two cells sharing one face.

The `polyMesh` is inherited by the `fvMesh` which contains extra parts needed for the finite volume schemes including functions as:

`V()` Volumes of the cells. Numbered according to cell numbering.

`Sf()` Surface normals with magnitude equal to the area. Numbered according to face numbers.

Further general information is found in the programmers guide [1, section 2.3].

4.2 `lduMatrix` format

The `lduMatrix` is a basic square sparse matrix format used in OpenFOAM[2]. The matrix is stored in three arrays: the diagonal, the upper and the lower part. Each is stored as a `scalarField` as can be seen `lduMatrix.H`

```
85        //- Coefficients (not including interfaces)
86        scalarField *lowerPtr_, *diagPtr_, *upperPtr_;
```

Listing 1: `lduMatrix.H`

where the pointers for the `scalarFields` are setup. Following the structure of the mesh, the diagonal elements are numbered according to the cell numbers, and the off-diagonal elements are numbered according to faces. For each off-diagonal element, corresponding to the discretization of a face, the owners, neighbours and face normals can be found by the call:

```
1         const surfaceVectorField& Sf = p.mesh().Sf();
2         const unallocLabelList& owner = mesh.owner();
3         const unallocLabelList& neighbour = mesh.neighbour();
```

Listing 2: Surface normal, owner and neighbour for each face

Considering the sparse matrix:

$$\mathbf{A} = A_{i,j} \tag{10}$$

the upper part of the sparse matrix ($i > j$) corresponds to the contribution from cell j on cell i, and vice versa for the lower part. The element $A_{i,j}$ for $i > j$ is stored in the list of upper elements, and its index is the same as the face number the face shared between i and j. Further examples of the `lduMatrix` format can be found in [3].

4.3 `fvMatrix`

The `fvMatrix` inherits the `lduMatrix` and is the OpenFOAM matrix specialization for the finite volume method. The `fvMatrix` contains not only the diagonal, upper and lower elements as inherited from the `lduMatrix` but also a source (a right hand side of the equation) and a reference to the field to be solved for. Further there are a set of operators which are used in the solver implementations. Examples are seen e.g. in the velocity equation of the incompressible, turbulent, steady-state solver `simpleFoam`:

```
3     volScalarField AU = UEqn().A();
4     U = UEqn().H()/AU;
```

Listing 3: Part of `pEqn.H` in `simpleFoam`

where the `A`-operator corresponds to the diagonal coefficient of the `lduMatrix` divided by the volume for the corresponding cell. Further the `fvMatrix` class adds helper functions for taking care of the contributions from the boundary conditions. This will be discussed and used in section 6.

4.4 `fvm` and `fvc` discretization

In general a discretization scheme in the finite volume methodology can be described as implicit and explicit. In OpenFOAM separation of the two is implemented as two different namespaces; `fvm` for an implicit discretization and `fvc` for an explicit discretization. Not all operators are found in both namespaces, e.g. there is no implicit discretization scheme for the divergence operator implemented in OpenFOAM. In general the use of

such operator would lead to an undesirable matrix, ill-conditioned[4]. However, for the implementation of a block coupled pressure-velocity solver such operator will be needed.

The discretizations are used in the definition of the equations, as for example for the implementation of the pressure equation in `simpleFoam`:

```
12          fvScalarMatrix pEqn
13          (
14              fvm::laplacian(1.0/AU, p) == fvc::div(phi)
15          );
```

Listing 4: Part of `pEqn.H` in `simpleFoam`

where the LHS is an implicit discretization of a Laplacian operator of the pressure and the RHS is an explicit discretization of the divergence of the flux. The explicit discretizations will in general give fields corresponding to the cell structure (`volScalarField-/volVectorField`) or face structure (`surfaceScalarField/surfaceVectorField`). This is added to the implicit part of the matrix via the mathematical operators (+,-,*,/) as defined in the matrix classes.

A complete list of the existing operators and more elaborate discussion of the discretization procedures is given in the programmer's guide [1, section 2.4]. In order to understand the code implemented in the present work, it must be recognized that an explicit discretization will end as a source term, and any explicit dependencies will be taken as of the previous iteration, not updated during the solving of the matrix.

4.5 Discretization of boundary conditions

The OpenFOAM boundary conditions are defined on the patches, the set of faces, making up the external faces. In general the boundary conditions will be taken care of in the discretization operator implementations. The application programmer will merely have to specify that the boundary conditions should be corrected, i.e. re-calculated.

To understand how the contributions from the boundary conditions are included in the `fvMatrix` assembling, consider the access functions of the Dirichlet boundary condition `fixedValue`:

```
139             //- Return the matrix diagonal coefficients corresponding to the
140             //  evaluation of the value of this patchField with given weights
141             virtual tmp<Field<Type> > valueInternalCoeffs
142             (
143                 const tmp<scalarField>&
144             ) const;
145
146             //- Return the matrix source coefficients corresponding to the
147             //  evaluation of the value of this patchField with given weights
148             virtual tmp<Field<Type> > valueBoundaryCoeffs
149             (
150                 const tmp<scalarField>&
151             ) const;
152
153             //- Return the matrix diagonal coefficients corresponding to the
154             //  evaluation of the gradient of this patchField
155             virtual tmp<Field<Type> > gradientInternalCoeffs() const;
156
```

```
157             //- Return the matrix source coefficients corresponding to the
158             //  evaluation of the gradient of this patchField
159             virtual tmp<Field<Type> > gradientBoundaryCoeffs() const;
```

Listing 5: Part of `fixedValueFvPatchField.H`

As explained in the comments, the four functions are aimed to give the value and gradient contributions to the diagonal and source terms for each cell neighbouring a face in the presently described patch. Thus, when implementing a boundary condition the contribution of the boundary conditions can be included by calling these functions. E.g. in the matrix assembling in the implementation of a Gaussian convection scheme:

```
92      forAll(fvm.psi().boundaryField(), patchI)
93      {
94          const fvPatchField<Type>& psf = fvm.psi().boundaryField()[patchI];
95          const fvsPatchScalarField& patchFlux = faceFlux.boundaryField()[patchI];
96          const fvsPatchScalarField& pw = weights.boundaryField()[patchI];
97
98          fvm.internalCoeffs()[patchI] = patchFlux*psf.valueInternalCoeffs(pw);
99          fvm.boundaryCoeffs()[patchI] = -patchFlux*psf.valueBoundaryCoeffs(pw);
100     }
```

Listing 6: Part of `gaussConvectionScheme.C`

the contributions from the boundary is added to the diagonal and the source using the `value` functions. For comparison in the implementation of the Gaussian Laplacian scheme:

```
70      forAll(fvm.psi().boundaryField(), patchI)
71      {
72          const fvPatchField<Type>& psf = fvm.psi().boundaryField()[patchI];
73          const fvsPatchScalarField& patchGamma =
74              gammaMagSf.boundaryField()[patchI];
75
76          fvm.internalCoeffs()[patchI] = patchGamma*psf.gradientInternalCoeffs();
77          fvm.boundaryCoeffs()[patchI] = -patchGamma*psf.gradientBoundaryCoeffs();
78      }
```

Listing 7: Part of `gaussianLaplacianScheme.C`

where instead the `gradient` functions are used, as a gradient is left in the expression after applying Gauss theorem on the Laplacian term.

5 OpenFOAM block coupling implementation

This far, the basic idea of the block coupled matrix formulation has been given in theory and some OpenFOAM general discretization procedure and matrix assembling notas have been given. In this section the block coupled solver is studied. Section 5.1 discusses the implemented classes allowing the block coupled solver in general. In section 5.2, an example solver as implemented in OpenFOAM is discussed in detail.

5.1 Classes related to block coupling

In OpenFOAM 1.6-ext a block matrix class has been implemented. Except for the block coupled matrix class itself, a number of other classes have been implemented to enable templated matrix elements and solution vectors of other dimension than one (`scalar`) and three (`vector`).

`BlockLduMatrix`

The class `BlockLduMatrix` is a block matrix class, allowing templated matrix elements (as compared to `lduMatrix` which can ony have scalar elements). The class basically contains similar matrix elements as `lduMatrix`:

```
115             //- List of coupled interfaces
116             typename BlockLduInterfaceFieldPtrsList<Type>::Type interfaces_;
117
118             //- Coupled interface coefficients, upper
119             FieldField<CoeffField, Type> coupleUpper_;
120
121             //- Coupled interface coefficients, lower
122             FieldField<CoeffField, Type> coupleLower_;
```

Listing 8: `BlockLduMatrix.H`

Instead of `scalarField`, templated `CoeffField` are used for the matrix elements. At construction a standard `lduMesh` must be provided. The `lduMesh` addressing is used also for the block coupled matrix, and thus the sparsity structure is conserved.

`CoeffField`

The templated coefficient field allows a generic field, which is needed for the coupled matrix. Mathematical operators are implemented for the templated type. The `CoeffField` allows access to the elements of the field in multiple formats, among other:

```
182             //- Return as scalar field
183             scalarTypeField& asScalar();
184
185             //- Return as linear field
186             linearTypeField& asLinear();
187
188             //- Return as square field
189             squareTypeField& asSquare();
```

Listing 9: `CoeffField.H`

This gives different possibilities for the programmer to access references to the coefficients for assembling purposes.

`VectorN`

In order to construct other solution vectors than `scalar` or `vector` fields a class with arbitrary number of components is introduced. Using the basic `VectorN` class, specializations are done for the desired dimensions, e.g. for `vector4`:

```
50 typedef VectorN<scalar, 4> vector4;
51
52
53 //- Specify data associated with vector4 type is contiguous
54 template<>
55 inline bool contiguous<vector4>() {return true;}
```

Listing 10: `vector4.H`

Note that `vector3` has not been specialized, as this is supposed to be the same as the inherent `vector`. This is however not really the case as the access to the components of vectors is done by `.x(),.y(),.z()` for the `vector` and `.(0),.(1),.(2)` for a potential `vector3`. It is thus recommended to implement `vector3`, which is straightforward looking at e.g. the `vector4` class. Such implementation is needed e.g. for the two dimensional pressure-velocity coupled solver presented below.

5.2 Example solver `blockCoupledScalarTransport`

To establish an idea of the possibilities of the block matrix implementation in OpenFOAM, the solver `blockCoupledScalarTransportFoam` is studied. The accompanying tutorial case `blockCoupledSwirlTest` is presented briefly in Appendix B.

5.2.1 Theory

The theory behind the solver is a coupled two phase heat transfer problem described by [5, 6]:

$$\nabla \cdot (UT) - \nabla(D\nabla T) = \alpha(T_s - T) \tag{11}$$

$$-\nabla(DT_s\nabla D_s) = \alpha(T - T_s) \tag{12}$$

where the velocity field U is not solved for but prescribed, and T and T_s are the fluid and solid temperature respectively. The coupling between the equations is obvious.

5.2.2 Implementation

Considering the solver directory[2], it is seen that two temperature fields are constructed in the `createFields.H` corresponding to T and T_s. Further a combined block temperature field is created:

```
31      Info<< "Creating field blockT\n" << endl;
32      volVector2Field blockT
33      (
34          IOobject
35          (
36              "blockT",
37              runTime.timeName(),
38              mesh,
39              IOobject::NO_READ,
40              IOobject::NO_WRITE
41          ),
42          mesh,
43          dimensionedVector2(word(), dimless, vector2::zero)
44      );
```

Listing 11: `createFields.H`

The values from the two separate temperature fields are inserted in to the `volVector2Field` such that:

```
51      {
52          vector2Field& blockX = blockT.internalField();
53
54          blockMatrixTools::blockInsert(0, T.internalField(), blockX);
55          blockMatrixTools::blockInsert(1, Ts.internalField(), blockX);
56      }
```

Listing 12: `createFields.H`

[2] `$WM_PROJECT_DIR/applications/solvers/coupled/blockCoupledScalarTransportFoam`

such that an element a_i in `blockT` consists of:

$$a_i = \begin{bmatrix} T_i \\ T_{s,i} \end{bmatrix} \tag{13}$$

The internal field of `blockT` is initiated using the read values of `T` and `T_s`. This is done using the utilities found in the class `blockMatrixTools`. Furthermore, the velocity field and thermal diffusivities are read together with α (not shown above).

Considering now the solver itself, the first part consists of all the includes and the start of an iteration in SIMPLE style (not shown), followed by the creation of two standard equations:

```
74              fvScalarMatrix TEqn
75              (
76                  fvm::div(phi, T)
77                - fvm::laplacian(DT, T)
78               ==
79                   alpha*Ts
80                - fvm::Sp(alpha, T)
81              );
82
83              TEqn.relax();
84
85              fvScalarMatrix TsEqn
86              (
87                - fvm::laplacian(DTs, Ts)
88               ==
89                   alpha*T
90                - fvm::Sp(alpha, Ts)
91              );
92
93              TsEqn.relax();
```

Listing 13: Part of `blockCoupledScalarTransportFoam.C`

The equations are constructed with the coupling introduced in an explicit, segregated manner, as is seen from the first term on the right hand side. This is followed by the creation of a block coupled matrix:

```
95              // Prepare block system
96              BlockLduMatrix<vector2> blockM(mesh);
97
98              // Grab block diagonal and set it to zero
99              Field<tensor2>& d = blockM.diag().asSquare();
100             d = tensor2::zero;
101
102             // Grab linear off-diagonal and set it to zero
103             Field<vector2>& u = blockM.upper().asLinear();
104             Field<vector2>& l = blockM.lower().asLinear();
105             u = vector2::zero;
106             l = vector2::zero;
107
108             vector2Field& blockX = blockT.internalField();
109             // vector2Field blockX(mesh.nCells(), vector2::zero);
110             vector2Field blockB(mesh.nCells(), vector2::zero);
111
112             //- Inset equations into block Matrix
113             blockMatrixTools::insertEquation(0, TEqn, blockM, blockX, blockB);
```

```
114             blockMatrixTools::insertEquation(1, TsEqn, blockM, blockX, blockB);
```

Listing 14: Part of `blockCoupledScalarTransportFoam.C`

The introduced block matrix is templated, and thus the type is given; `vector2`. The equations can be inserted in its segregated form in to the block matrix using the utilities in `blockMatrixTools`. Note that also a source is constructed, again templated as a `vector2`.

```
169             vector2Field& blockX = blockT.internalField();
170             // vector2Field blockX(mesh.nCells(), vector2::zero);
171             vector2Field blockB(mesh.nCells(), vector2::zero);
172
173             //- Inset equations into block Matrix
174             blockMatrixTools::insertEquation(0, TEqn, blockM, blockX, blockB);
175             blockMatrixTools::insertEquation(1, TsEqn, blockM, blockX, blockB);
```

Listing 15: `blockCoupledScalarTransportFoam.C` part 3

At this stage the blocked equation `blockM` corresponds to the equation system (3), no implicit coupling has yet been utilized. Considering the implementation of the equations, the explicit terms `alpha*T` and `alpha*Ts` need to be subtracted from the source (`blockB`) and the coefficient (α) transfered to the diagonal block of the matrix:

```
116             //- Add off-diagonal terms and remove from Block source
117             forAll(d, i)
118             {
119                 d[i](0,1) = -alpha.value()*mesh.V()[i];
120                 d[i](1,0) = -alpha.value()*mesh.V()[i];
121
122                 blockB[i][0] -= alpha.value()*blockX[i][1]*mesh.V()[i];
123                 blockB[i][1] -= alpha.value()*blockX[i][0]*mesh.V()[i];
124             }
```

Listing 16: Part of `blockCoupledScalarTransportFoam.C`

What now remains of the assembling is to take care of special boundaries, e.g. allowing the cyclic boundaries to be implicitly given in the block coupled system:

```
126             //- Transfer the coupled interface list for processor/cyclic/etc. boundaries
127             blockM.interfaces() = blockT.boundaryField().blockInterfaces();
128
129             //- Transfer the coupled interface coefficients
130             forAll(mesh.boundaryMesh(), patchI)
131             {
132                 if (blockM.interfaces().set(patchI))
133                 {
134                     Field<vector2>& coupledLower = blockM.coupleLower()[patchI].asLinear();
135                     Field<vector2>& coupledUpper = blockM.coupleUpper()[patchI].asLinear();
136
137                     const scalarField& TLower = TEqn.internalCoeffs()[patchI];
138                     const scalarField& TUpper = TEqn.boundaryCoeffs()[patchI];
139                     const scalarField& TsLower = TsEqn.internalCoeffs()[patchI];
140                     const scalarField& TsUpper = TsEqn.boundaryCoeffs()[patchI];
141
142                     blockMatrixTools::blockInsert(0, TLower, coupledLower);
143                     blockMatrixTools::blockInsert(1, TsLower, coupledLower);
144                     blockMatrixTools::blockInsert(0, TUpper, coupledUpper);
```

```
145                     blockMatrixTools::blockInsert(1, TsUpper, coupledUpper);
146                 }
147             }
```

Listing 17: Part of `blockCoupledScalarTransportFoam.C`

Finally the matrix system is solved using a block coupled solver. The calculated values of the temperatures are then transferred back to the original fields `T` and `Ts` and after finishing the correction iterations data is written and the SIMPLE loop is completed (not shown below):

```
149             //- Block coupled solver call
150             BlockSolverPerformance<vector2> solverPerf =
151                 BlockLduSolver<vector2>::New
152                 (
153                     word("blockVar"),
154                     blockM,
155                     mesh.solver("blockVar")
156                 )->solve(blockX, blockB);
157
158             solverPerf.print();
159
160             // Retrieve solution
161             blockMatrixTools::blockRetrieve(0, T.internalField(), blockX);
162             blockMatrixTools::blockRetrieve(1, Ts.internalField(), blockX);
163
164             T.correctBoundaryConditions();
165             Ts.correctBoundaryConditions();
```

Listing 18: Part of `blockCoupledScalarTransportFoam.C`

A few points are worth notice considering the above excursion of `blockCoupledScalarTransportFoam`:

- The templating of `BlockCoupLduMatrix` allows an arbitrary sized tensor at each matrix element. There are no limitations on the number of equations that could be coupled in this manner.
- The block matrix assembling can be simplified by the utilities in `blockMatrixTools`, also giving a starting point for implementing other black matrix operations.
- A set of special solvers has been implemented, allowing templating of the size of the tensor used for an element also in the solvers.

5.3 Alternative approach for equations of equal structure

Clifford [6] outlines another way to use the block coupled solver, utilizing regularities of the coupled system. In the given example N coupled equations of exact same structure is outlined, using discretization operators aimed for the block matrix class. This approach, although only useful for problems with equally structured equations, can give a cleaner and simpler assembling of the system.

6 Implementing pressure and velocity coupling

Solving the pressure-velocity coupling is relevant for the majority of all calculations performed by CFD. A large number of algorithms has been proposed for this purpose, including SIMPLE[7] and variations thereof.

The SIMPLE method basically solves the pressure-velocity coupling by estimating the velocity based on the present pressure field, followed by a pressure correction equation formulated through the continuity equation, and finally a momentum corrector step based on the new pressure field. Such algorithm segregates the solution of the pressure and velocity field, in many cases resulting in slow convergence.

For the purpose of potentially faster convergence more implicit solver procedures has been proposed. Such solver model will be discussed in 6.1. The OpenFOAM implementation will be outlined in section 6.2.

6.1 Coupled model

The theory proposed in this report is based primarily on the work by Darwish. et al[8]. The continuity and momentum equations will for the restriction of laminar, constant density and imcompressible flow read:

$$\nabla \cdot (\mathbf{U}) = 0 \tag{14}$$

$$\nabla \cdot (\mathbf{U}\mathbf{U}) - \nabla(\nu \nabla \mathbf{U}) = -\frac{1}{\rho}\nabla p \tag{15}$$

Considering a general discretization of the different terms in a finite volume manner we can write the above equations in the general form:

$$\sum_{\text{faces}} \mathbf{U}_\text{f} \cdot \mathbf{S}_\text{f} = 0 \tag{16}$$

$$\sum_{\text{faces}} [\mathbf{U}\mathbf{U} - \mu \nabla \mathbf{U}]_\text{f} \cdot \mathbf{S}_\text{f} = -\sum_{\text{faces}} P_\text{f} \mathbf{S}_\text{f} \tag{17}$$

where the Gauss theorem has been applied, allowing the gradient and divergence terms to be written in terms of surface integrals discretized to a sum over the control volume surfaces. This is equivalent to the OpenFOAM discretizations[1]. Note that the density has been included in the pressure such that:

$$\frac{p}{\rho} = P \tag{18}$$

In the SIMPLE algorithm, a pressure correction equation is retrieved from the momentum predictor inserted in to the continuity equation (for OpenFOAM specific implementation outline see [9]). In order to get a coupled equation system another approach must be used.

Using Rhie-Chow interpolation[10] in the semi-distretized form of the continuity equation (16) gives a second dependence for the pressure:

$$\sum_{\text{faces}} \left[\overline{\mathbf{U}_\text{f}} - \overline{\mathbf{D}_\text{f}}\left(\nabla P_\text{f} - \overline{\nabla P_\text{f}}\right] \cdot \mathbf{S}_\text{f} = 0 \tag{19}$$

where the overline indicates a value retrieved from interpolation of face values. For the pressure term ∇P cell gradient values should be interpolated to the face. Tannehill et al[11] states that the Rhie-Chow interpolation can somehow be seen as local correction of the pressure gradient used to calculate the face flux (i.e. the face velocity), which is how it will be used in this work.

The $\mathbf{D}_\text{f}$ operator corresponds to ratio of the cell volume and the central (diagonal) coefficient of the discretized version of the left hand side of the momentum equation (17), such that:

$$\mathbf{D}_\text{f} = \begin{bmatrix} \frac{\Omega_P}{a_p^x} & 0 & 0 \\ 0 & \frac{\Omega_P}{a_p^y} & 0 \\ 0 & 0 & \frac{\Omega_P}{a_p^z} \end{bmatrix} \tag{20}$$

where a_p^i corresponds to the i:th central component of the momentum equation left hand discretization. Simply rewriting eq. (19):

$$\sum_{\text{faces}} -\overline{\mathbf{D}_\text{f}} \nabla P_\text{f} \cdot \mathbf{S}_\text{f} + \sum_{\text{faces}} \overline{\mathbf{U}_\text{f}} \cdot \mathbf{S}_\text{f} = \sum_{\text{faces}} -\overline{\mathbf{D}_\text{f} \nabla P_\text{f}} \cdot \mathbf{S}_\text{f} \tag{21}$$

and treating the right hand side as a source term, and combining it with eq. (17) a coupled set of four equations (for 3D case) has been retrieved.

6.2 OpenFOAM implementation

In order to implement the coupled equation system (17) and (21) in an OpenFOAM style, the different terms will be considered separately. The structured equation approach as discussed briefly in section 5.3 can be immediately out-ruled as the system of equations are not having the same general structure. Instead, an approach similar to the discussed tutorial solver (section 5.2) is designed. First the setting up of the block matrix and the fields necessary is discussed, followed by the discretization of each term in eqs. (17) and (21).

The code, with same line numbering, can be seen in its complete in Appendix C. The discussed solver is aimed at three dimensional calculations. This means that the tensor elements in the block matrix will be of length 4 (three velocity components and pressure), with rank 2.

The three dimensional solver will be used also for two dimensional cases. The third dimension velocity residual will then immediately be very small. The drawback with this is that unnecessary calculations will be performed for the third direction, not leading to any valuable result. Thus, for high performance issues it is better to implement a specific 2D solver. Such implementation is in its self straightforward using a block matrix with

tensor elements of length 3 (two velocity components and pressure), and the code is thus not presented within this work.

6.2.1 Block matrix and source term

For the coupled calculation the solution vector x^P will consist of the three velocity components and the pressure such that:

$$x^P = x_l^P = \begin{bmatrix} u^P \\ v^P \\ w^P \\ P^P \end{bmatrix} \tag{22}$$

A solution vector corresponding to the mesh with each element consisting of a block with four elements is constructed after the construction of the pressure and velocity field:

```
117 // Block vector field for the pressure and velocity field to be solved for
118 volVector4Field pU
119 (
120     IOobject
121     (
122         "pU",
123         runTime.timeName(),
124         mesh,
125         IOobject::NO_READ,
126         IOobject::NO_WRITE
127     ),
128     mesh,
129     dimensionedVector4(word(),dimless,vector4::zero)
130 );
```

In order to get the initial values from the existing (separate) pressure and velocity fields the initial conditions are transferred to the new solution vector:

```
132 // Insert the pressure and velocity internal fields in to the volVector2Field
133 {
134     vector4Field blockX = pU.internalField();
135
136     // Separately add the three velocity components
137     for (int i=0; i<3;i++)
138     {
139         tmp<scalarField> tf = U.internalField().component(i);
140         scalarField& f = tf();
141         blockMatrixTools::blockInsert(i,f,blockX);
142     }
143
144     // Pressure is the 2nd component
145     scalarField& f = p.internalField();
146     blockMatrixTools::blockInsert(3,f,blockX);
147 }
```

The coupled solver will be formulated through the assembled block matrix system:

$$\mathbf{A}^P x^P + \sum_{\mathrm{F}\in\{\mathrm{N}\}} \mathbf{A}^F x^F = b^P \tag{23}$$

which corresponds to the fully discretized versions of eqs. (17) and (21). Each matrix element A^P or A^F is a tensor of rank 2 and length 4, with P a cell with neighbours F and with:

$$A^X = \left[a^X_{k,l}\right]_i \quad k,l \in \{u,v,w,p\}, \quad X \in \{P,F\} \tag{24}$$

In general $a^X_{k,l}$ gives the contribution from field component k to l.

Considering the general structure of the OpenFOAM matrix (`lduMatrix`, section 4.2), the diagonal coefficients will correspond to the A^P elements and the off-diagonal to the A^F elements.

The block matrix is constructed using:

```
188             // Matrix block
189             BlockLduMatrix<vector4> B(mesh);
```

and the diagonal and off-diagonal matrix elements can be retrieved in tensor form as:

```
191             // Diagonal is set separately
192             Field<tensor4>& d = B.diag().asSquare();
193
194             // Off-diagonal also as square
195             Field<tensor4>& u = B.upper().asSquare();
196             Field<tensor4>& l = B.lower().asSquare();
```

Similar to the `lduMatrix` the `BlockLduMatrix` does not contain any source, and will thus be constructed separately:

```
198             // Source term for the block matrix
199             Field<vector4> s(mesh.nCells(), vector4::zero);
```

At this point the general block structure is finished. The next step consists in constructing the matrix elements coefficients.

6.2.2 Discretizing the momentum equation

The momentum equation (17) will be discretized in two parts.

LHS The LHS is recognized from the implementation of `simpleFoam` such that:

```
182             tmp<fvVectorMatrix> UEqnLHS
183             (
184                 fvm::div(phi,U)
185               + turbulence->divDevReff(U)
186             );
```

Note that as compared to eq. (15) turbulence is introduced by calling the `divDivReff(U)` in the turbulence model. As the previous part is already formulated as a `fvVectorMatrix` the diagonal, upper, and lower coefficients can be immediately retrieved from the matrix:

```
202             tmp<scalarField> tdiag = UEqnLHS().D();
203             scalarField& diag = tdiag();
204             scalarField& upper = UEqnLHS().upper();
205             scalarField& lower = UEqnLHS().lower();
```

Usin the function `D()`, the boundary diagonal contribution will be automatically included, but for the source the the boundary contribution will need to be added:

```
211             // Add source boundary contribution
212             vectorField& source = UEqnLHS().source();
213             UEqnLHS().addBoundarySource(source, false);
```

RHS The RHS is less straightforward. In the segregated implementation of `simpleFoam` the RHS is taken explicitly. This is not feasible in the coupled approach as an implicit coupling is desired. Thus an implicit gradient operator (i.e in namespace `fvm`) is needed. As earlier stated such does not exist.

Considering the RHS as a separate problem:

$$\sum_{\text{faces}} P_\text{f} \mathbf{S}_\text{f} = 0 \tag{25}$$

it is seen that the pressure is desired at the faces. In order to get this OpenFOAM uses run-time chosen interpolation schemes. By setting up a such interpolation scheme:

```
218             // Interpolation scheme for the pressure weights
219             tmp<surfaceInterpolationScheme<scalar> >
220             tinterpScheme_
221             (
222                 surfaceInterpolationScheme<scalar>::New
223                 (
224                     p.mesh(),
225                     p.mesh().interpolationScheme("grad(p)")
226                 )
227             );
```

weights based on the present mesh can be calculated:

```
218             tmp<surfaceScalarField> tweights = tinterpScheme_().weights(p);
219             const surfaceScalarField& weights = tweights();
```

A specific weight is calculated for each face, and the value of the weight corresponds to the weight to be used for the owner cell of that face. The neighbour cell weight is simply found as:

$$w_N = 1 - w_P \tag{26}$$

In order to compute an implicit discretization of the pressure gradient, vector fields equivalent to diagonal, lower, upper and the source of such matrix is needed:

```
229             // Pressure gradient contributions - corresponds to an implicit
230             // gradient operator
231             tmp<vectorField> tpUv = tmp<vectorField>
232                 (
233                     new vectorField(upper.size(),pTraits<vector>::zero)
234                 );
235             vectorField& pUv = tpUv();
236             tmp<vectorField> tpLv = tmp<vectorField>
237                 (
238                     new vectorField(lower.size(),pTraits<vector>::zero)
239                 );
240             vectorField& pLv = tpLv();
```

```
241             tmp<vectorField> tpSv = tmp<vectorField>
242                 (
243                     new vectorField(source.size(),pTraits<vector>::zero)
244                 );
245             vectorField& pSv = tpSv();
246             tmp<vectorField> tpDv = tmp<vectorField>
247                 (
248                     new vectorField(diag.size(),pTraits<vector>::zero)
249                 );
250             vectorField& pDv = tpDv();
```

The matrix coefficients can then be calculated according to the weights:

```
256             for(int i=0;i<owner.size();i++)
257             {
258                 int o = owner[i];
259                 int n = neighbour[i];
260                 scalar w = weights.internalField()[i];
261                 vector s = Sf[i];
262
263                 pDv[o]+=s*w;
264                 pDv[n]-=s*(1-w);
265                 pLv[i]=-s*w;
266                 pUv[i]=s*(1-w);
267
268             }
```

Note that a diagonal contribution for the present face (`i`) will need to be added both for the owner and the neighbour cell. For the neighbour cell the surface normal will point in the opposite direction wherefore these contributions are added negatively. Two diagonal contributions are needed, in the lower coefficient vector for the contribution from the owner cell on the neighbour cell and in the upper coefficient vector for the contribution from the neighbour cell on the owner cell.

As no `fvMatrix` is used the helper functions `addBoundarySource/addBoundaryDiag` are not available, consequently the boundary contribution must be added manually:

```
270             // Get boundary condition contributions for pressure grad(P)
271             p.boundaryField().updateCoeffs();
272             forAll(p.boundaryField(),patchI)
273             {
274                 // Present fvPatchField
275                 fvPatchField<scalar> & fv = p.boundaryField()[patchI];
276
277                 // Retrieve the weights for the boundary
278                 const fvsPatchScalarField& pw = weights.boundaryField()[patchI];
279
280                 // Contributions from the boundary coefficients
281                 tmp<Field<scalar> > tic = fv.valueInternalCoeffs(pw);
282                 Field<scalar>& ic = tic();
283                 tmp<Field<scalar> > tbc = fv.valueBoundaryCoeffs(pw);
284                 Field<scalar>& bc = tbc();
285
286                 // Get the fvPatch only
287                 const fvPatch& patch = fv.patch();
288
289                 // Surface normals for this patch
290                 tmp<Field<vector> > tsn = patch.Sf();
291                 Field<vector> sn = tsn();
292
```

```
293                 // Manually add the contributions from the boundary
294                 // This what happens with addBoundaryDiag, addBoundarySource
295                 forAll(fv,facei)
296                 {
297                     label c = patch.faceCells()[facei];
298 
299                     pDv[c]+=ic[facei]*sn[facei];
300                     pSv[c]-=bc[facei]*sn[facei];
301                 }
302             }
```

Again, care has to be taken with the signs of the contributions. The contribution from the boundary face to the source term (`valueBoundaryCoeffs`) should be taken negative as this is moved to the source term.

At this stage the contributions from the LHS and RHS can be added to the block matrix. The coefficients that are set corresponds to $a_{u,u}$,$a_{v,v}$,$a_{w,w}$,$a_{p,u}$,$a_{p,v}$,$a_{p,w}$:

```
317             forAll(d,i)
318             {
319                 d[i](0,0) = diag[i];
320                 d[i](1,1) = diag[i];
321                 d[i](2,2) = diag[i];
322 
323                 d[i](0,3) = pDv[i].x();
324                 d[i](1,3) = pDv[i].y();
325                 d[i](2,3) = pDv[i].z();
326             }
327             forAll(l,i)
328             {
329                 l[i](0,0) = lower[i];
330                 l[i](1,1) = lower[i];
331                 l[i](2,2) = lower[i];
332 
333                 l[i](0,3) = pLv[i].x();
334                 l[i](1,3) = pLv[i].y();
335                 l[i](2,3) = pLv[i].z();
336             }
337             forAll(u,i)
338             {
339                 u[i](0,0) = upper[i];
340                 u[i](1,1) = upper[i];
341                 u[i](2,2) = upper[i];
342 
343                 u[i](0,3) = pUv[i].x();
344                 u[i](1,3) = pUv[i].y();
345                 u[i](2,3) = pUv[i].z();
346             }
347             forAll(s,i)
348             {
349                 s[i](0) = source[i].x()+pSv[i].x();
350                 s[i](1) = source[i].y()+pSv[i].y();
351                 s[i](2) = source[i].z()+pSv[i].z();
352             }
```

6.2.3 Discretizing the continuity equation

The continuity equation (21) will again be discretized in two parts.

Pressure terms For the pressure terms, one implicit and one explicit contribution is to be calculated.

The implicit part can be discretized with the implicit Laplacian operator (`fvm::laplacian`) with the `D`-operator computed as the inverse of the `A`-operator:

```
439             tmp<volScalarField> tA = UEqnLHS().A();
440             volScalarField& A = tA();
```

Note that the operator `A` is coded to use the `D` operator which in OpenFOAM is not the same as the operator described in eq. 20 (which is rather equivalent to the operator `DD`).

The explicit part corresponding to the RHS of eq. (21). Is to be interpolated to the faces. This can be achieved by first applying the explicit gradient operator on the pressure only and then using the explicit divergence operator on the product of the explict gradient and the inverse of the `A`-operator, which combined with the implicit part gives:

```
442             tmp<volVectorField> texp = fvc::grad(p);
443             volVectorField& exp = texp();
444             tmp<volVectorField> texp2 = exp/A;
445             volVectorField exp2 = texp2();
446
447             tmp<fvScalarMatrix> MEqnLHSp
448             (
449                 -fvm::laplacian(1/A,p)
450                 ==
451                 -fvc::div(exp2)
452             );
```

The different matrix parts are then extracted, and again the boundary contribution is taken care of:

```
454             // Add the boundary contributions
455             scalarField& pMdiag = MEqnLHSp().diag();
456             scalarField& pMupper = MEqnLHSp().upper();
457             scalarField& pMlower = MEqnLHSp().lower();
458
459             // Add diagonal boundary contribution
460             MEqnLHSp().addBoundaryDiag(pMdiag,0);
461
462             // Add source boundary contribution
463             scalarField& pMsource = MEqnLHSp().source();
464             MEqnLHSp().addBoundarySource(pMsource, false);
```

Velocity term For this term an implicit divergence discretization is needed. Again, this is not generally any desired operator and does not exist. The procedure to perform such discretization follows the same structure as the implicit discretization of the pressure gradient:

```
348             // Again an implicit version not existing, now the div operator
349             tmp<surfaceInterpolationScheme<scalar> >
350             UtinterpScheme_
351             (
352                 surfaceInterpolationScheme<scalar>::New
353                 (
354                     U.mesh(),
```

```
355                     U.mesh().interpolationScheme("div(U)(implicit)")
356                 )
357             );
358
359
360             // 1) Setup diagonal, source, upper and lower
361             tmp<vectorField> tMUpper = tmp<vectorField>
362                 (new vectorField(upper.size(),pTraits<vector>::zero));
363             vectorField& MUpper = tMUpper();
364
365             tmp<vectorField> tMLower = tmp<vectorField>
366                 (new vectorField(lower.size(),pTraits<vector>::zero));
367             vectorField& MLower = tMLower();
368
369             tmp<vectorField> tMDiag = tmp<vectorField>
370                 (new vectorField(diag.size(),pTraits<vector>::zero));
371             vectorField& MDiag = tMDiag();
372
373             tmp<vectorField> tMSource = tmp<vectorField>
374                 (
375                     new vectorField
376                     (
377                         source.component(0)().size(),pTraits<vector>::zero
378                     )
379                 );
380             vectorField& MSource = tMSource();
381
382             // 2) Use interpolation weights to assemble the contributions
383             tmp<surfaceScalarField> tMweights =
384                 UtinterpScheme_().weights(mag(U));
385             const surfaceScalarField& Mweights = tMweights();
386
387             for(int i=0;i<owner.size();i++)
388             {
389                 int o = owner[i];
390                 int n = neighbour[i];
391                 scalar w = Mweights.internalField()[i];
392                 vector s = Sf[i];
393
394                 MDiag[o]+=s*w;
395                 MDiag[n]-=s*(1-w);
396                 MLower[i]=-s*w;
397                 MUpper[i]=s*(1-w);
398             }
399
400             // Get boundary condition contributions for the pressure grad(P)
401             U.boundaryField().updateCoeffs();
402             forAll(U.boundaryField(),patchI)
403             {
404                 // Present fvPatchField
405                 fvPatchField<vector> & fv = U.boundaryField()[patchI];
406
407                 // Retrieve the weights for the boundary
408                 const fvsPatchScalarField& Mw =
409                     Mweights.boundaryField()[patchI];
410
411                 // Contributions from the boundary coefficients
412                 tmp<Field<vector> > tic = fv.valueInternalCoeffs(Mw);
413                 Field<vector>& ic = tic();
414                 tmp<Field<vector> > tbc = fv.valueBoundaryCoeffs(Mw);
415                 Field<vector>& bc = tbc();
416
```

```
417                 // Get the fvPatch only
418                 const fvPatch& patch = fv.patch();
419
420                 // Surface normals for this patch
421                 tmp<Field<vector> > tsn = patch.Sf();
422                 Field<vector> sn = tsn();
423
424                 // Manually add the contributions from the boundary
425                 // This what happens with addBoundaryDiag, addBoundarySource
426                 forAll(fv,facei)
427                 {
428                     label c = patch.faceCells()[facei];
429
430                     MDiag[c]+=cmptMultiply(ic[facei],sn[facei]);
431                     MSource[c]-=cmptMultiply(bc[facei],sn[facei]);
432                 }
433             }
```

In the same manner as for the momentum equation, the block matrix is then updated. This time with the terms $a_{u,p}$,$a_{v,p}$,$a_{w,p}$,$a_{p,p}$:

```
469             forAll(d,i)
470             {
471                 d[i](3,0) = MDiag[i].x();
472                 d[i](3,1) = MDiag[i].y();
473                 d[i](3,2) = MDiag[i].z();
474                 d[i](3,3) = pMdiag[i];
475             }
476             forAll(l,i)
477             {
478                 l[i](3,0) = MLower[i].x();
479                 l[i](3,1) = MLower[i].y();
480                 l[i](3,2) = MLower[i].z();
481                 l[i](3,3) = pMlower[i];
482             }
483             forAll(u,i)
484             {
485                 u[i](3,0) = MUpper[i].x();
486                 u[i](3,1) = MUpper[i].y();
487                 u[i](3,2) = MUpper[i].z();
488                 u[i](3,3) = pMupper[i];
489             }
490             forAll(s,i)
491             {
492                 s[i](3) =  MSource[i].x()
493                           +MSource[i].y()
494                           +MSource[i].z()
495                           +pMsource[i];
496             }
```

6.2.4 Solving the block matrix and retrieving the solution

As the block mesh elements has been assembled the system can now be solved:

```
501             BlockSolverPerformance<vector4> solverPerf =
502                 BlockLduSolver<vector4>::New
503                 (
504                     word("blockVar"),
505                     B,
```

```
506                     mesh.solver("blockVar")
507                 )->solve(pU,s);
508
509             solverPerf.print();
```

Finally the solution is transferred from the coupled solution vector to the separate fields, the boundary conditions are recalculated and possibly relaxation is applied:

```
515             tmp<scalarField> tUx = U.internalField().component(0);
516             scalarField& Ux = tUx();
517             blockMatrixTools::blockRetrieve(0, Ux, pU);
518             U.internalField().replace(0,Ux);
519
520             tmp<scalarField> tUy = U.internalField().component(1);
521             scalarField& Uy = tUy();
522             blockMatrixTools::blockRetrieve(1, Uy, pU);
523             U.internalField().replace(1,Uy);
524
525             tmp<scalarField> tUz = U.internalField().component(2);
526             scalarField& Uz = tUz();
527             blockMatrixTools::blockRetrieve(2, Uz, pU);
528             U.internalField().replace(2,Uz);
529
530             blockMatrixTools::blockRetrieve(3, p.internalField(), pU);
531
532             UEqnLHS.clear();
533
534             p.relax();
535
536             U.correctBoundaryConditions();
537             p.correctBoundaryConditions();
```

The turbulence is in this solver solved using the standard function `correct()` applied to the turbulence model.

7 Results of `pUCoupledFoam`

To investigate the correctness and performance of the coupled solver, comparisons to `simpleFoam` are done. Benchmarking is done for both the 2D and the 3D solver.

There are two solvers implemented for the `BlockLduMatrix`: GMRES and BiCGStab. In general the bi-conjugate gradient stabilized solver was found more prone to diverge, wherefore it is not used in this work. The dimension of the Krylov space (`nDirections`) in combination with the maximum number of iterations (`maxIter`) has a significant effect on the convergence rate of the coupled solver and will be tested separately in section 7.3.

7.1 2D coupled solver for `pitzDaily`

The two dimensional coupled solver (based on `vector3`) is benchmarked using the standard `pitzDaily` case. The solver data for the coupled solver is seen in table 1. The solution and schemes dictionaries are given for this case in Appendix D.

Table 1: Solver configuration for the two dimensional coupled solver.

Solver	GMRES
Preconditioner	Cholesky
Convergence criteria	1e-9
Krylov space dimension (`nDirections`)	5
Max iterations	10
Underrelaxation p	1.0
Underrelaxation U	1.0
Underrelaxation k	0.7
Underrelaxation ϵ	0.7

The pressure can be seen in figure 2 for the solution of the block coupled solver. The pressure for $y = 0.0125$ is given in figure 3 for both the `simpleFoam` and `pUCoupledSolver`. A minor discrepancy is seen in the pressure whereas the velocity profiles coincide. Considering the convergence of the cases given in figure 4 it is seen that the pressure in the coupled solution is further converged, likely explaining the minor difference. The profiles confirm that the new solver has been correctly implemented.

A performance comparison can be seen in figure 4. It is seen that the number of iterations as well as the time to reach the same convergence is shorter for the block coupled solver. It should be noted that the convergence of k is in this case approximately converging as the velocity field.

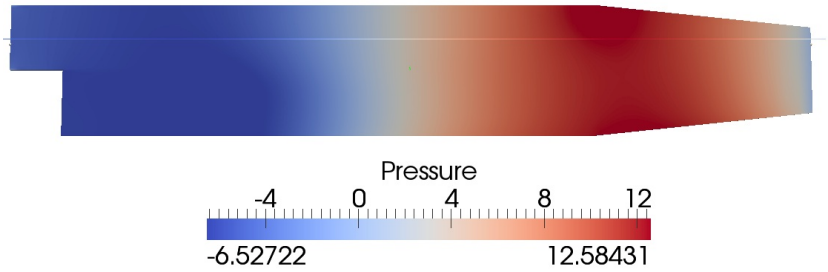


Figure 2: Pressure profile for solution of the `dailyPitz` using `pUCoupledFoam`.

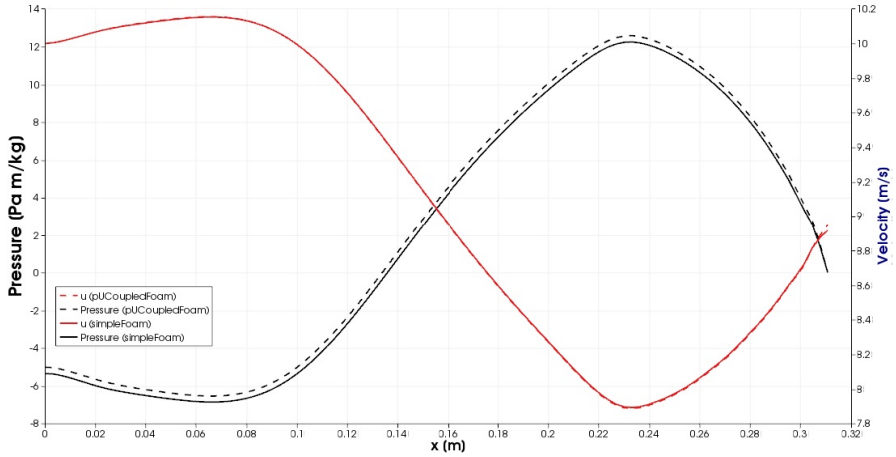
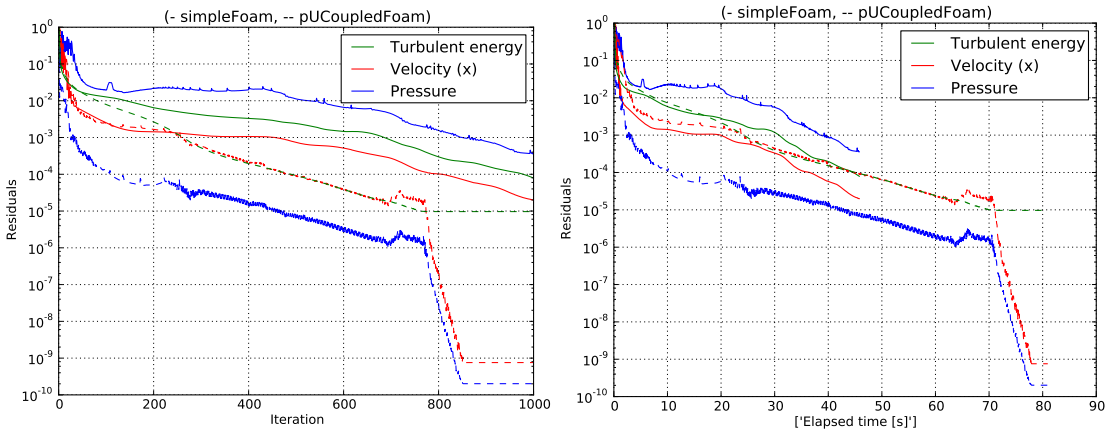


Figure 3: Pressure comparison for `simpleFoam` and `pUCoupledFoam` at line indicated in figure 2



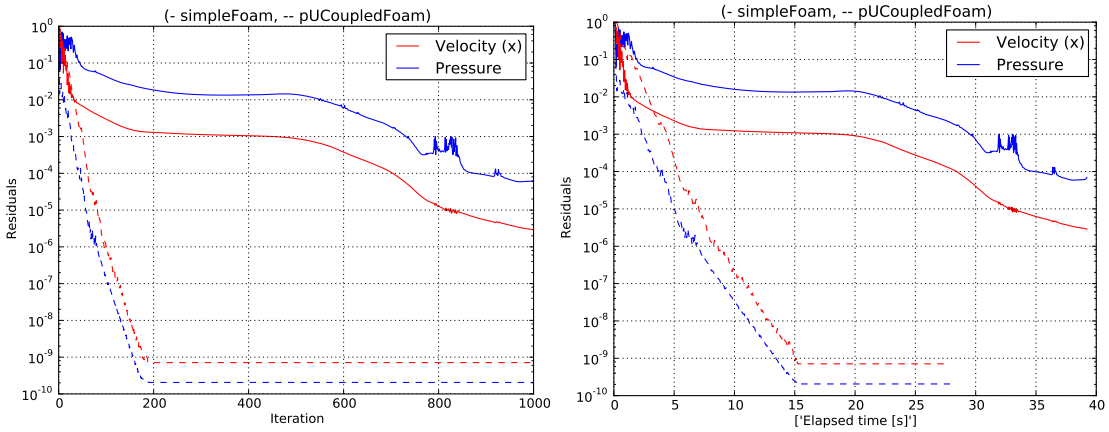
(a) Iterations **(b)** Elapsed time

Figure 4: Comparison of convergence for `simpleFoam` and `pUCoupledFoam`.

7.2 2D coupled solver for `pitzDaily` no turbulence

To eliminate the effect of the segregated turbulence model on the convergence rate, the `pitzDaily` case was run with no turbulence. Whereas this is not necessarily a correct approach from physics viewpoint, it will give a clearer comparison of the gain of the block coupled approach.

Figure 5 shows a comparison of the convergence for the laminar case. The convergence rate difference is larger as compared to the turbulent case (compare figure 4). This is expected since the turbulence model excluded, was a segregated solver slowing down the implemented block matrix. Figure 5b proposes a seven fold performance gain, considering the convergence of the pressure.



(a) Iterations **(b)** Elapsed time

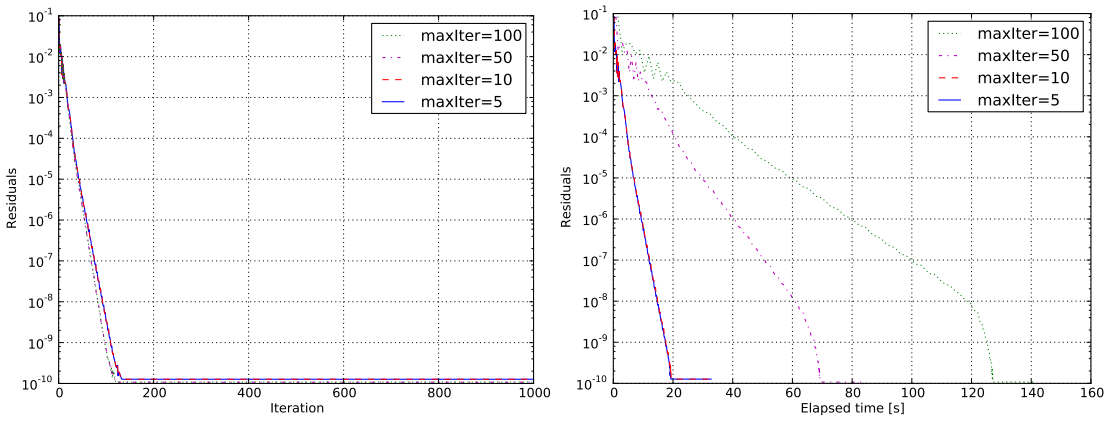
Figure 5: Comparison of convergence for `simpleFoam` and `pUCoupledFoam`.

7.3 Sensitivity analysis of `nDirections` and `maxIter` of block GMRES

For the choice of the parameters of the GMRES solver, a separate parameter study is performed. In figure 6 the convergence on `pitzDaily` without turbulence with the two dimensional coupled solver is compared for different maximum numbers of iterations. As can be seen from the figure increasing the number of iterations, leading to a more precise result within each time iteration, will not give a smaller number of time iterations. On the other hand, a higher number of maximum iterations will increase the time within each iteration and thus the convergence is slower.

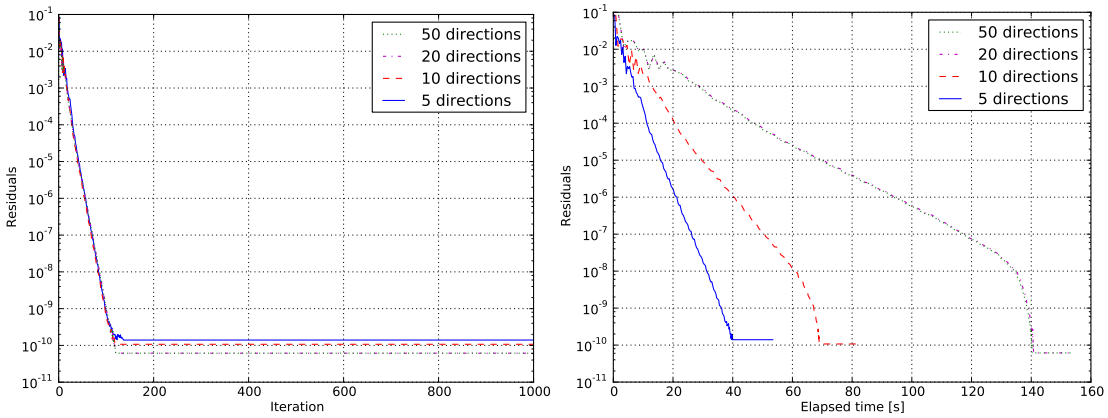
In the same manner, as displayed in figure 7b, increasing `nDirections` will not lead to a faster convergence in iterations, but rather a slower convergence in time.

This kind of results show that when introducing the block coupled solver, new parameters must be investigated in order to get the fastest possible convergence rate. The convergence rates will depend on the geometry and also the type of equations. As the solved equations are non-linear, there are still explicit parts introduced in the equation, and thus using a very high number of maximum iterations is not so beneficial.



(a) Iterations **(b)** Elapsed time

Figure 6: Comparison of convergence of the maximum number of iterations for GMRES solver. Elapsed time on horizontal axis.

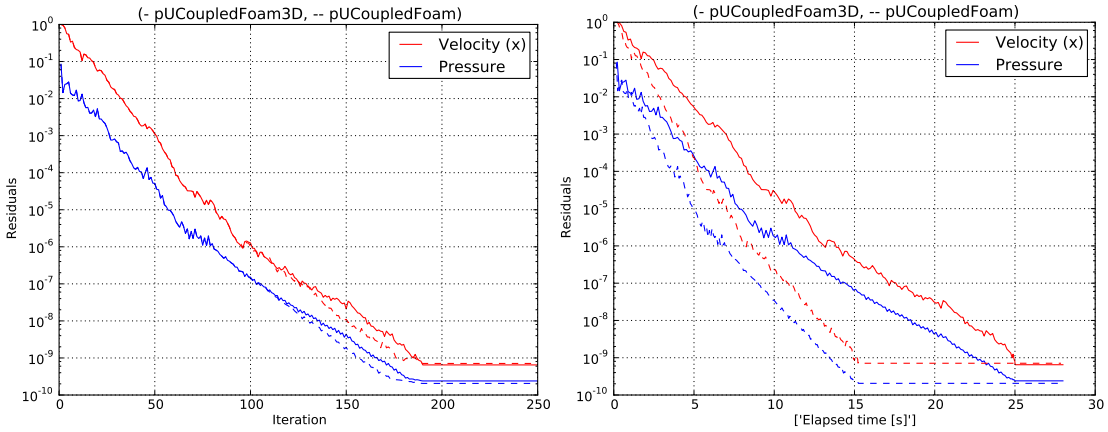


(a) Iterations **(b)** Elapsed time

Figure 7: Comparison of convergence of the Krylov space dimension (`nDirections`) for GMRES solver. Elapsed time on horizontal axis.

7.4 Using 3D solver for 2D problems

As discussed in the introduction of section 6.2, the 3D solver can be used for 2D calculations also, with a slight loss in performance. Such performance loss is outlined in figure 8. Using the higher dimensional solver for the lower dimensional case leads to a significantly lower performance. It is thus better to use a solver specialized for the dimension of the problem.



(a) Iterations **(b)** Elapsed time

Figure 8: Comparison of convergence of the two dimensional and the three dimensional coupled solver for the two dimensional `pitzDaily` case without turbulence.

7.5 Internal flow 3D case

In order to benchmark the three dimensional version of the coupled solver an internal flow case was produced. The Geometry can be seen in figure 9. The mesh is constructed from a single block using the `cellSet` utilities to select a set of the mesh. The underralaxation factors and solver specifications can be seen in table 2.

Table 2: Solver configuration for the three dimensional coupled solver.

Block solver	GMRES
Preconditioner	Cholesky
Convergence criteria	1e-9
Krylov space dimension (`nDirections`)	10
Max iterations	10
Underrelaxation p	1.0
Underrelaxation U	1.0
Underrelaxation k	0.7
Underrelaxation ϵ	0.7

Pressure and velocity profiles can be seen for horizontal lines parallel to the x-axis in in figures 10 and 11. The coupled solver again gives close to the same result as `simpleFoam`. Again the explanation of the discrepancy is differing convergence (compare figure 12).

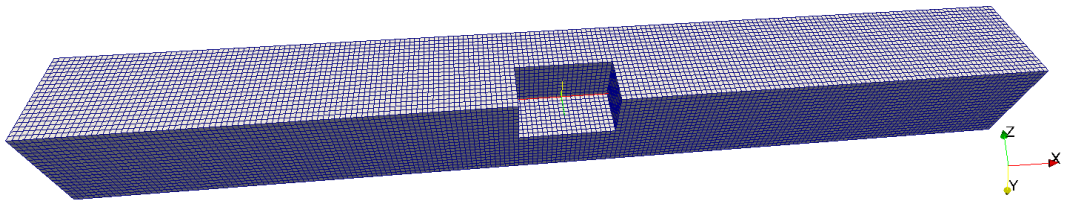


Figure 9: Geometry and mesh for the 3D channel with a flow obstacle placed at the center. The mesh consists of in total 624000 hexahedral elements.

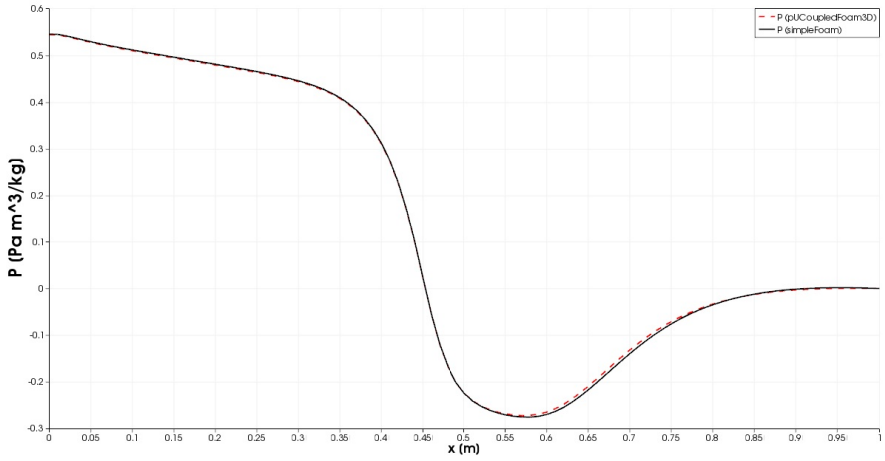


Figure 10: Pressure profile comparison for $(y,z) = (0.075,0.075)$.

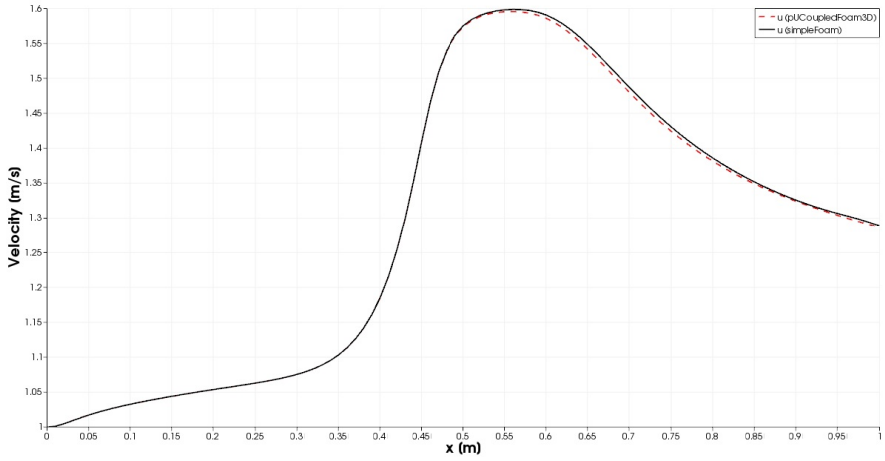
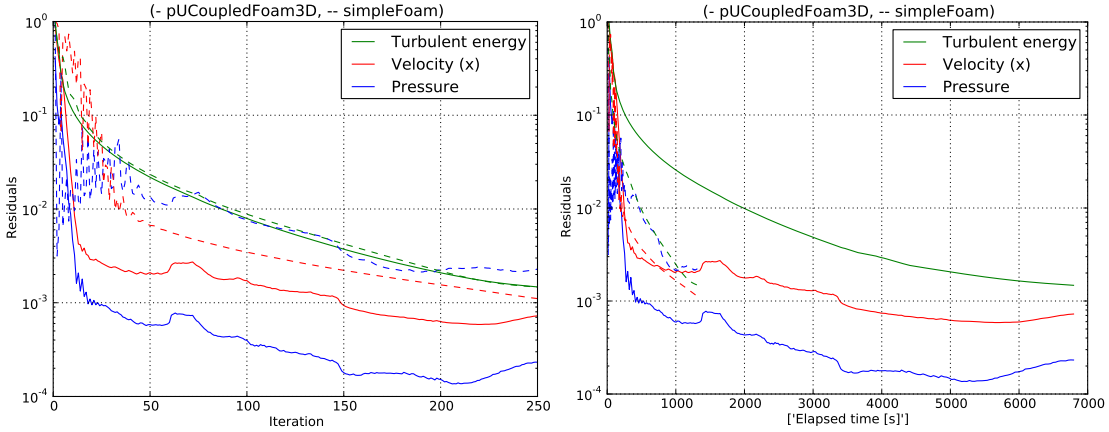


Figure 11: Velocity profile comparison for $(y,z) = (0.075,0.075)$.

A performance comparison can be seen in figure 12 and figure 13. The gain of the block coupled approach is less obvious in this case as compared to the `pitzDaily` cases. Considering figure 13a there is still a non-significant gain in the convergence per number of iterations, except for the turbulent energy (k) which converges in the same number of iterations. Comparing per time (figure 13b) it is seen that the convergence of k is slower, which is again only another way to show that the number of iterations limits the convergence of the turbulence model, not the convergence of the pressure and velocity field. Thus, eventually the turbulence will limit the convergence of also the pressure and the velocity and the gain from the block coupled solver is lost.

The dependence on the turbulence is also seen in the large change in convergence rate (the slope in the figures) occurring after approximately 12 iterations. Although, this could likely depend on the geometry, any such change is not seen in the laminar case (compare figure 5).



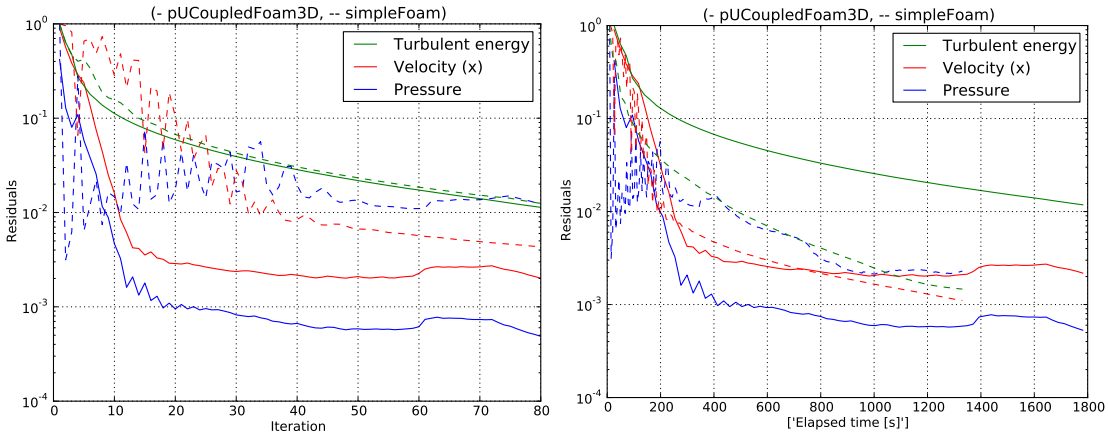
(a) Iterations **(b)** Elapsed time

Figure 12: Comparison of convergence of `simpleFoam` and `pUCoupledFoam3D` (250 iterations).

7.6 `motorBike` 3D case

Except for the `pitzDaily` case there is a second standard case for `simpleFoam`; `motorBike`. The case was tested with the coupled solver, and was found divergent. Also performing a few initial iterations by `simpleFoam` followed by the coupled solver was found divergent.

Studying the result of the few iterations succesful before reach a floating point exception it was seen that the pressure and the velocity diverged close to some of the most distorted cells close to the fender. The mesh for this region is shown in figure 14. As seen the cells are highly skewed, which causes trouble in the coupled solver. Possibly using limited schemes could remedy the troubles.



(a) Iterations **(b)** Elapsed time

Figure 13: Comparison of convergence of `simpleFoam` and `pUCoupledFoam3D` (50 iterations).

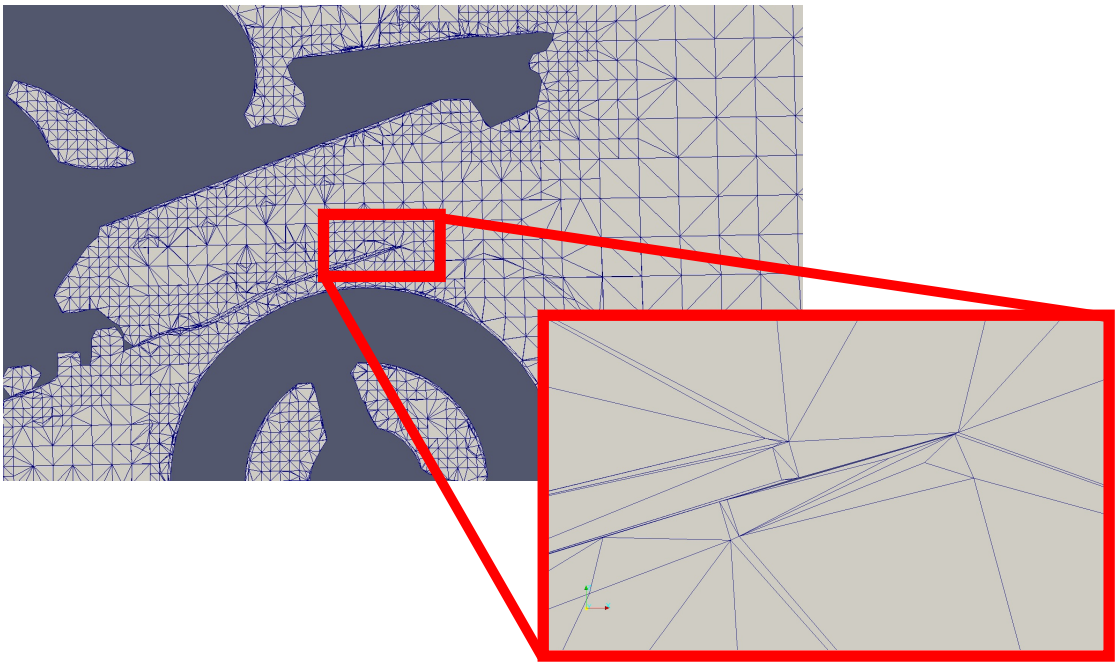


Figure 14: Parts of the mesh generated by `snappyHexMesh` in the `motorBike` case.

8 Conclusions and outlook

In this work a pressure-velocity coupled solver for steady-state turbulent flow was implemented in two and three dimensions. The solver was shown to give the same results as `simpleFoam` for both the 2D and 3D cases. Large gains in performance was seen for the 2D case, both in the turbulent and, even more, in the laminar case. For the 3D case the performance gain was smaller. The turbulence fields were seen to limit the convergence of the case, and thus also limiting the full potential of the coupled solver.

Although convergence was not reached for the `motorBike` case, this does not necessarily say so much about the algorithm, as the low quality of the mesh is apparent.

The work gives not only an insight in the OpenFOAM matrix and discretization procedures but also a clear indication of how the block matrix structure can be beneficially used. Although not all of the framework existed, considering e.g. implict discretizations of gradient and divergence operators, the OpenFOAM code generality allows for relatively fast and straightforward implementation of advanced solvers.

Future, further developments could include:

Coupling turbulence
: In order to make the dependence on the turbulence smaller some coupled approach also in the turbulence could be advantageous. Many models consist of a set of two equations which are interdepedent with potential performance gain from coupled solutions.

Transient modelling
: Including also transient calculations would give a new possible benefit from the coupled solver. As the pressure and velocity fields converges together, something similar to the PIMPLE algorithm is immediately achieved without any iteration. This can potentially give major time benefits.

Parallelization
: The present implementation does not allow for parallelization. Parallelization in combination with block matrix has been performed by Clifford [6], and is possible.

Coupled and cyclic boundaries
: In the tutorial case dissected in section 5.2.2 special care was given to coupled interfaces. This was not implemented for the pressure-velocity solver. Such extension would be necessary for certain applications, as rotating system, etc.

Code cleaning
: To make coupled calculations easier to implement more implicit operators could beneficially be implemented. To extract the code for the implicit gradient and divergence operators implemented in this work could be a first step.

References

[1] *OpenFOAM. Programmers Guide*. Version 2.0.0. OpenFOAM Foundation, 2011.

[2] Hrvoje Jasak. *Five basic classes in OpenFOAM*. June 2010. URL: http://web.student.chalmers.se/groups/ofw5/Advanced_Training/FiveBasicClasses.pdf.

[3] OpenFOAM Wiki. *OpenFOAM guide/Matrices in OpenFOAM*. 2012. URL: http://openfoamwiki.net/index.php/OpenFOAM_guide/Matrices_in_OpenFOAM.

[4] *Gradient operator implicit discretization*. CFD-Online. 2012. URL: http://www.cfd-online.com/Forums/openfoam-solving/59717-gradient-operator-implicit-discretization.html.

[5] Henrik Rusche and Hrvoje Jasak. *Implicit solution techniques for coupled multi-field problems Block Solution, Coupled Matrices*. June 2010.

[6] Ivor Clifford. *Block-Coupled Simulations Using OpenFOAM*. June 2011.

[7] S.V. Patankar and D.B. Spalding. "A calculation procedure for heat, mass and momentum transfer in three dimensional parabolic flows". In: *International Journal of Heat and Mass Transfer* 15 (Dec. 1972), pp. 1787–1806.

[8] M. Darwish, I. Sraj, and F. Moukalled. "A coupled finite volume solver for the solution of incompressible flows on unstructured grids". In: *Journal of Computational Physics* 228 (2009), pp. 180–201.

[9] OpenFOAM Wiki. *The SIMPLE algorithm in OpenFOAM*. Last visited: 2012-09-27. Mar. 2010. URL: http://openfoamwiki.net/index.php/The_SIMPLE_algorithm_in_OpenFOAM.

[10] C.M. Rhie and W.L. Chow. "A numerical study of the turbulent flow past an isolated airfoil with trailing edge separation". In: *AIAA Journal* 21 (1983), pp. 1525–1532.

[11] John C. Tannehill, Dale A. Anderson, and Richard H. Pletcher. *Computational Fluid Mechanics and Heat Transfer*. second edition. Taylor & Francis, Washington, USA, 997.

[12] Julia Springer et al. *A coupled pressure based solution algorithm based on the Volume-of-Fluid approach for two or more immiscible fluids*. June 2010.

[13] L. Mangani and C. Bianchini. *A coupled finite volume solver for the solution of laminar/turbulent incompressible and compressible flows*. June 2010.

A Other sources on OpenFOAM block coupling

In order to grasp the idea of the `blockCoupled` matrix format a few presentations and articles are worth mentioning, most originated from the OpenFOAM workshops or similar:

[5]**Henrik Rusche and Hrvoje Jasak. *Implicit solution techniques for coupled multi-field problems Block Solution, Coupled Matrices.* June 2010**
Basic introduction to the idea of region coupling and block coupling for CFD simulations. Describes with a few examples how it has been implemented in OpenFOAM.

[6]**Ivor Clifford. *Block-Coupled Simulations Using OpenFOAM.* June 2011**
Gives theory of the block coupled solver, including solver strategy and matrix classes. Outlines two different ways to use the functionality; either manually setting the off-diagonal (coupled) terms, or by using pre-existing assembler routines block coupled equations of equal structure.

[12]**Julia Springer et al. *A coupled pressure based solution algorithm based on the Volume-of-Fluid approach for two or more immiscible fluids.* June 2010**
Example of Volume-of-Fluid (VOF) calculations performed, discretizing the phase equations and the pressure equations to a block coupled matrix.

Further, other works have presented the idea of block coupled solutions, without necessary explicitly using the `BlockLduMatrix` matrix format, such as:

[13]**L. Mangani and C. Bianchini. *A coupled finite volume solver for the solution of laminar/turbulent incompressible and compressible flows.* June 2010**
A general introduction to pressure-velocity coupling with block solvers. Gives multiple examples of comparisons of segregated and coupled solutions, showing a major benefit in the number of iterations.

[8]**M. Darwish, I. Sraj, and F. Moukalled. "A coupled finite volume solver for the solution of incompressible flows on unstructured grids". In: *Journal of Computational Physics* 228 (2009), pp. 180–201**
A more detailed description of coupled pressure and velocity calculations for incompressible, laminar flow independent of OpenFOAM. Theoretical base for [13].

B Tutorial case for `blockCoupledScalarTransportFoam`

A simple tutorial test case is implented in `blockCoupledSwirlTest`. The system consists of a 2D rectangular area with a structured mesh and prescribed velocity field of the fluid, as shown in Figure 15 The resulting temperatures can be seen in Figure 16.

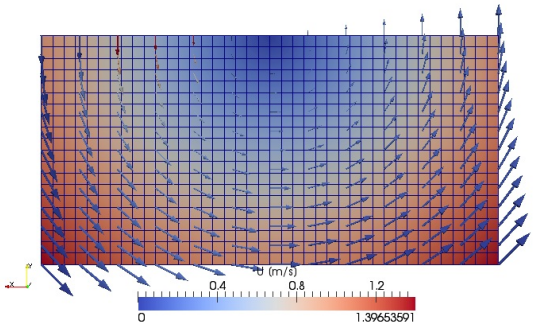


Figure 15: Geometry for tutorial test case of the block coupled solver. Displaying velocity magnitude in background with direction in glyphs.

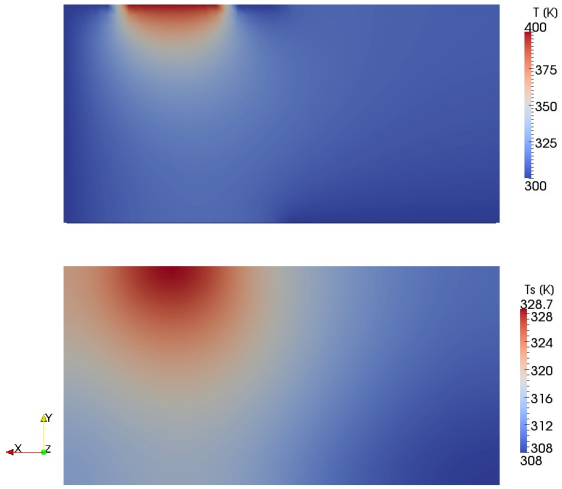


Figure 16: Temperature distributions for the solid and fluid temperatures.

Benchmarking and comparison

This case has been compared to a segregated solver showing beneficial results for the block coupled solver algorithm[5]. The number of iterations to reach convergence was decreased by a factor of two approximately.

C Complete code for `pUCoupledFoam`

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright held by original author
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM 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 2 of the License, or (at your
    option) any later version.

    OpenFOAM 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 OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Application
    pvCoupledFoam

Description
    Steady-state solver for incompressible, turbulent flow, with implicit
    coupling between pressure and velocity achieved by BlockLduMatrix
    Turbulence is in this version solved using the existing turbulence
    structure.
Auther
    Klas Jareteg, 2012-09-27-2012-10-16
Version
    0.6

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

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


// Add the support for arbitry vector size types
#include "VectorNFieldTypes.H"
#include "volVectorNFields.H"

// Add the support for block matrices
#include "blockLduSolvers.H"
#include "blockVectorNMatrices.H"

// Add the utilities written for blockMatrix transformations
#include "blockMatrixTools.H"

// Surface interpolation schemes to get the weights
#include "surfaceInterpolationScheme.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

```
60
61 int main(int argc, char *argv[])
62 {
63 #   include "setRootCase.H"
64 #   include "createTime.H"
65
66 Info<< "Create mesh for time = "
67     << runTime.timeName() << Foam::nl << Foam::endl;
68 Info<<" Mesh name: "<<fvMesh::defaultRegion<<endl;
69 Foam::fvMesh mesh
70 (
71     Foam::IOobject
72     (
73         Foam::fvMesh::defaultRegion,
74         //"fluid",
75         runTime.timeName(),
76         runTime,
77         Foam::IOobject::MUST_READ
78     )
79 );
80
81 Info << "Reading field p\n" << endl;
82 volScalarField p
83 (
84     IOobject
85     (
86         "p",
87         runTime.timeName(),
88         mesh,
89         IOobject::MUST_READ,
90         IOobject::AUTO_WRITE
91     ),
92     mesh
93 );
94
95 Info << "Reading field U\n" << endl;
96 volVectorField U
97 (
98     IOobject
99     (
100         "U",
101         runTime.timeName(),
102         mesh,
103         IOobject::MUST_READ,
104         IOobject::AUTO_WRITE
105     ),
106     mesh
107 );
108
109 #include "createPhi.H"
110
111 singlePhaseTransportModel laminarTransport(U, phi);
112 autoPtr<incompressible::RASModel> turbulence
113 (
114     incompressible::RASModel::New(U, phi, laminarTransport)
115 );
116
117 // Block vector field for the pressure and velocity field to be solved for
118 volVector4Field pU
119 (
120     IOobject
121     (
```

```
122         "pU",
123         runTime.timeName(),
124         mesh,
125         IOobject::NO_READ,
126         IOobject::NO_WRITE
127     ),
128     mesh,
129     dimensionedVector4(word(),dimless,vector4::zero)
130 );
131 
132 // Insert the pressure and velocity internal fields in to the volVector2Field
133 {
134     vector4Field blockX = pU.internalField();
135 
136     // Separately add the three velocity components
137     for (int i=0; i<3;i++)
138     {
139         tmp<scalarField> tf = U.internalField().component(i);
140         scalarField& f = tf();
141         blockMatrixTools::blockInsert(i,f,blockX);
142     }
143 
144     // Pressure is the 2nd component
145     scalarField& f = p.internalField();
146     blockMatrixTools::blockInsert(3,f,blockX);
147 }
148 
149 // ============================================================================//
150 
151     Info<< "\nStarting time loop\n" << endl;
152     while (runTime.loop())
153     {
154         Info<< "Time = " << runTime.timeName() << nl << endl;
155 
156         p.storePrevIter();
157         U.storePrevIter();
158 
159         const surfaceVectorField& Sf = p.mesh().Sf();
160         const unallocLabelList& owner = mesh.owner();
161         const unallocLabelList& neighbour = mesh.neighbour();
162 
163         {
164             vector4Field blockX = pU.internalField();
165 
166             // Separately add the three velocity components
167             for (int i=0; i<3;i++)
168             {
169                 tmp<scalarField> tf = U.internalField().component(i);
170                 scalarField& f = tf();
171                 blockMatrixTools::blockInsert(i,f,blockX);
172             }
173 
174             // Pressure is the 2nd component
175             scalarField& f = p.internalField();
176             blockMatrixTools::blockInsert(3,f,blockX);
177 
178 
179 // ============================================================================//
180 // Velocity equation (LHS) matrix
181 // ============================================================================//
182             tmp<fvVectorMatrix> UEqnLHS
183             (
```

```
184                 fvm::div(phi,U)
185               + turbulence->divDevReff(U)
186             );
187
188             // Matrix block
189             BlockLduMatrix<vector4> B(mesh);
190
191             // Diagonal is set separately
192             Field<tensor4>& d = B.diag().asSquare();
193
194             // Off-diagonal also as square
195             Field<tensor4>& u = B.upper().asSquare();
196             Field<tensor4>& l = B.lower().asSquare();
197
198             // Source term for the block matrix
199             Field<vector4> s(mesh.nCells(), vector4::zero);
200
201             // Add the boundary contributions for the velocity equation
202             tmp<scalarField> tdiag = UEqnLHS().D();
203             scalarField& diag = tdiag();
204             scalarField& upper = UEqnLHS().upper();
205             scalarField& lower = UEqnLHS().lower();
206
207             // Add diagonal boundary contribution
208             // This is automatically done when you do UEqnLHS().D();
209             //UEqnLHS().addBoundaryDiag(diag,0);
210
211             // Add source boundary contribution
212             vectorField& source = UEqnLHS().source();
213             UEqnLHS().addBoundarySource(source, false);
214
215 // ===========================================================================//
216 // Pressure gradient matrix
217 // ===========================================================================//
218             // Interpolation scheme for the pressure weights
219             tmp<surfaceInterpolationScheme<scalar> >
220             tinterpScheme_
221             (
222                 surfaceInterpolationScheme<scalar>::New
223                 (
224                     p.mesh(),
225                     p.mesh().interpolationScheme("grad(p)")
226                 )
227             );
228
229             // Pressure gradient contributions - corresponds to an implicit
230             // gradient operator
231             tmp<vectorField> tpUv = tmp<vectorField>
232                 (
233                     new vectorField(upper.size(),pTraits<vector>::zero)
234                 );
235             vectorField& pUv = tpUv();
236             tmp<vectorField> tpLv = tmp<vectorField>
237                 (
238                     new vectorField(lower.size(),pTraits<vector>::zero)
239                 );
240             vectorField& pLv = tpLv();
241             tmp<vectorField> tpSv = tmp<vectorField>
242                 (
243                     new vectorField(source.size(),pTraits<vector>::zero)
244                 );
245             vectorField& pSv = tpSv();
```

```
246             tmp<vectorField> tpDv = tmp<vectorField>
247                 (
248                     new vectorField(diag.size(),pTraits<vector>::zero)
249                 );
250             vectorField& pDv = tpDv();
251 
252             // 2) Use interpolation weights to assemble the contributions
253             tmp<surfaceScalarField> tweights = tinterpScheme_().weights(p);
254             const surfaceScalarField& weights = tweights();
255 
256             for(int i=0;i<owner.size();i++)
257             {
258                 int o = owner[i];
259                 int n = neighbour[i];
260                 scalar w = weights.internalField()[i];
261                 vector s = Sf[i];
262 
263                 pDv[o]+=s*w;
264                 pDv[n]-=s*(1-w);
265                 pLv[i]=-s*w;
266                 pUv[i]=s*(1-w);
267 
268             }
269 
270             // Get boundary condition contributions for pressure grad(P)
271             p.boundaryField().updateCoeffs();
272             forAll(p.boundaryField(),patchI)
273             {
274                 // Present fvPatchField
275                 fvPatchField<scalar> & fv = p.boundaryField()[patchI];
276 
277                 // Retrieve the weights for the boundary
278                 const fvsPatchScalarField& pw = weights.boundaryField()[patchI];
279 
280                 // Contributions from the boundary coefficients
281                 tmp<Field<scalar> > tic = fv.valueInternalCoeffs(pw);
282                 Field<scalar>& ic = tic();
283                 tmp<Field<scalar> > tbc = fv.valueBoundaryCoeffs(pw);
284                 Field<scalar>& bc = tbc();
285 
286                 // Get the fvPatch only
287                 const fvPatch& patch = fv.patch();
288 
289                 // Surface normals for this patch
290                 tmp<Field<vector> > tsn = patch.Sf();
291                 Field<vector> sn = tsn();
292 
293                 // Manually add the contributions from the boundary
294                 // This what happens with addBoundaryDiag, addBoundarySource
295                 forAll(fv,facei)
296                 {
297                     label c = patch.faceCells()[facei];
298 
299                     pDv[c]+=ic[facei]*sn[facei];
300                     pSv[c]-=bc[facei]*sn[facei];
301                 }
302             }
303 
304 // ============================================================================//
305 // Assemble momentum equation
306 // ============================================================================//
307             // Assemble the momentum equation contributions
```

```
308             forAll(d,i)
309             {
310                 d[i](0,0) = diag[i];
311                 d[i](1,1) = diag[i];
312                 d[i](2,2) = diag[i];
313 
314                 d[i](0,3) = pDv[i].x();
315                 d[i](1,3) = pDv[i].y();
316                 d[i](2,3) = pDv[i].z();
317             }
318             forAll(l,i)
319             {
320                 l[i](0,0) = lower[i];
321                 l[i](1,1) = lower[i];
322                 l[i](2,2) = lower[i];
323 
324                 l[i](0,3) = pLv[i].x();
325                 l[i](1,3) = pLv[i].y();
326                 l[i](2,3) = pLv[i].z();
327             }
328             forAll(u,i)
329             {
330                 u[i](0,0) = upper[i];
331                 u[i](1,1) = upper[i];
332                 u[i](2,2) = upper[i];
333 
334                 u[i](0,3) = pUv[i].x();
335                 u[i](1,3) = pUv[i].y();
336                 u[i](2,3) = pUv[i].z();
337             }
338             forAll(s,i)
339             {
340                 s[i](0) = source[i].x()+pSv[i].x();
341                 s[i](1) = source[i].y()+pSv[i].y();
342                 s[i](2) = source[i].z()+pSv[i].z();
343             }
344 
345 // ============================================================================//
346 // Create implicit velocity (LHS) for continuity equation
347 // ============================================================================//
348             // Again an implicit version not existing, now the div operator
349             tmp<surfaceInterpolationScheme<scalar> >
350             UtinterpScheme_
351             (
352                 surfaceInterpolationScheme<scalar>::New
353                 (
354                     U.mesh(),
355                     U.mesh().interpolationScheme("div(U)(implicit)")
356                 )
357             );
358 
359 
360             // 1) Setup diagonal, source, upper and lower
361             tmp<vectorField> tMUpper = tmp<vectorField>
362                 (new vectorField(upper.size(),pTraits<vector>::zero));
363             vectorField& MUpper = tMUpper();
364 
365             tmp<vectorField> tMLower = tmp<vectorField>
366                 (new vectorField(lower.size(),pTraits<vector>::zero));
367             vectorField& MLower = tMLower();
368 
369             tmp<vectorField> tMDiag = tmp<vectorField>
```

```
370                 (new vectorField(diag.size(),pTraits<vector>::zero));
371             vectorField& MDiag = tMDiag();
372
373             tmp<vectorField> tMSource = tmp<vectorField>
374                 (
375                     new vectorField
376                     (
377                         source.component(0)().size(),pTraits<vector>::zero
378                     )
379                 );
380             vectorField& MSource = tMSource();
381
382             // 2) Use interpolation weights to assemble the contributions
383             tmp<surfaceScalarField> tMweights =
384                 UtinterpScheme_().weights(mag(U));
385             const surfaceScalarField& Mweights = tMweights();
386
387             for(int i=0;i<owner.size();i++)
388             {
389                 int o = owner[i];
390                 int n = neighbour[i];
391                 scalar w = Mweights.internalField()[i];
392                 vector s = Sf[i];
393
394                 MDiag[o]+=s*w;
395                 MDiag[n]-=s*(1-w);
396                 MLower[i]=-s*w;
397                 MUpper[i]=s*(1-w);
398             }
399
400             // Get boundary condition contributions for the pressure grad(P)
401             U.boundaryField().updateCoeffs();
402             forAll(U.boundaryField(),patchI)
403             {
404                 // Present fvPatchField
405                 fvPatchField<vector> & fv = U.boundaryField()[patchI];
406
407                 // Retrieve the weights for the boundary
408                 const fvsPatchScalarField& Mw =
409                     Mweights.boundaryField()[patchI];
410
411                 // Contributions from the boundary coefficients
412                 tmp<Field<vector> > tic = fv.valueInternalCoeffs(Mw);
413                 Field<vector>& ic = tic();
414                 tmp<Field<vector> > tbc = fv.valueBoundaryCoeffs(Mw);
415                 Field<vector>& bc = tbc();
416
417                 // Get the fvPatch only
418                 const fvPatch& patch = fv.patch();
419
420                 // Surface normals for this patch
421                 tmp<Field<vector> > tsn = patch.Sf();
422                 Field<vector> sn = tsn();
423
424                 // Manually add the contributions from the boundary
425                 // This what happens with addBoundaryDiag, addBoundarySource
426                 forAll(fv,facei)
427                 {
428                     label c = patch.faceCells()[facei];
429
430                     MDiag[c]+=cmptMultiply(ic[facei],sn[facei]);
431                     MSource[c]-=cmptMultiply(bc[facei],sn[facei]);
```

```
                }
            }

// ========================================================================//
// Create explicit and implict pressure parts for continuity equation
// ========================================================================//
            // Pressure parts of the mass equation
            tmp<volScalarField> tA = UEqnLHS().A();
            volScalarField& A = tA();

            tmp<volVectorField> texp = fvc::grad(p);
            volVectorField& exp = texp();
            tmp<volVectorField> texp2 = exp/A;
            volVectorField exp2 = texp2();

            tmp<fvScalarMatrix> MEqnLHSp
            (
                -fvm::laplacian(1/A,p)
                ==
                -fvc::div(exp2)
            );

            // Add the boundary contributions
            scalarField& pMdiag = MEqnLHSp().diag();
            scalarField& pMupper = MEqnLHSp().upper();
            scalarField& pMlower = MEqnLHSp().lower();

            // Add diagonal boundary contribution
            MEqnLHSp().addBoundaryDiag(pMdiag,0);

            // Add source boundary contribution
            scalarField& pMsource = MEqnLHSp().source();
            MEqnLHSp().addBoundarySource(pMsource, false);

// ========================================================================//
// Assemble mass equation
// ========================================================================//
            forAll(d,i)
            {
                d[i](3,0) = MDiag[i].x();
                d[i](3,1) = MDiag[i].y();
                d[i](3,2) = MDiag[i].z();
                d[i](3,3) = pMdiag[i];
            }
            forAll(l,i)
            {
                l[i](3,0) = MLower[i].x();
                l[i](3,1) = MLower[i].y();
                l[i](3,2) = MLower[i].z();
                l[i](3,3) = pMlower[i];
            }
            forAll(u,i)
            {
                u[i](3,0) = MUpper[i].x();
                u[i](3,1) = MUpper[i].y();
                u[i](3,2) = MUpper[i].z();
                u[i](3,3) = pMupper[i];
            }
            forAll(s,i)
            {
                s[i](3) =  MSource[i].x()
                          +MSource[i].y()
```

```
494                             +MSource[i].z()
495                             +pMsource[i];
496             }
497
498 // ============================================================================//
499 // Solve the block matrix
500 // ============================================================================//
501             BlockSolverPerformance<vector4> solverPerf =
502                 BlockLduSolver<vector4>::New
503                 (
504                     word("blockVar"),
505                     B,
506                     mesh.solver("blockVar")
507                 )->solve(pU,s);
508
509             solverPerf.print();
510
511 // ============================================================================//
512 // Retrieve the solution and update for next iteration
513 // ============================================================================//
514
515             tmp<scalarField> tUx = U.internalField().component(0);
516             scalarField& Ux = tUx();
517             blockMatrixTools::blockRetrieve(0, Ux, pU);
518             U.internalField().replace(0,Ux);
519
520             tmp<scalarField> tUy = U.internalField().component(1);
521             scalarField& Uy = tUy();
522             blockMatrixTools::blockRetrieve(1, Uy, pU);
523             U.internalField().replace(1,Uy);
524
525             tmp<scalarField> tUz = U.internalField().component(2);
526             scalarField& Uz = tUz();
527             blockMatrixTools::blockRetrieve(2, Uz, pU);
528             U.internalField().replace(2,Uz);
529
530             blockMatrixTools::blockRetrieve(3, p.internalField(), pU);
531
532             UEqnLHS.clear();
533
534             p.relax();
535
536             U.correctBoundaryConditions();
537             p.correctBoundaryConditions();
538
539         }
540
541         phi = fvc::interpolate(U) & mesh.Sf();
542         turbulence->correct();
543         runTime.write();
544
545         Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
546             << "  ClockTime = " << runTime.elapsedClockTime() << " s"
547             << nl << endl;
548
549     }
550
551     Info<< "End\n" << endl;
552
553     return 0;
554 }
```

Listing 19: `pUCoupledFoam`

D Case specifications for benchmarks

```
/*---------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM Extend Project: Open Source CFD        |
|  \\    /   O peration     | Version:  1.6-ext                               |
|   \\  /    A nd           | Web:      www.extend-project.de                 |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{

    blockVar
    {
        solver GMRES;
        preconditioner Cholesky;
        nDirections 5;

        tolerance 1e-09;
        relTol  0;

        minIter 1;
        maxIter 10;
    }


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

```
60          relTol              0.1;
61      };
62      R
63      {
64          solver              PBiCG;
65          preconditioner      DILU;
66          tolerance           1e-05;
67          relTol              0.1;
68      };
69      nuTilda
70      {
71          solver              PBiCG;
72          preconditioner      DILU;
73          tolerance           1e-05;
74          relTol              0.1;
75      };
76  }
77
78  SIMPLE
79  {
80      nNonOrthogonalCorrectors 0;
81  }
82
83  relaxationFactors
84  {
85      p                   1.0;
86      U                   1.0;
87      k                   0.7;
88      epsilon             0.7;
89  }
90
91  // ************************************************************************* //
```

Listing 20: `fvSolution` for and `pUCoupledFoam` for the coupled case benchmark. For `simpleFoam` underralaxation is applied to `p`(0.3) and `U`(0.7).

```
1  /*--------------------------------*- C++ -*----------------------------------*\
2  | =========                      |                                           |
3  | \\      /  F ield              | OpenFOAM Extend Project: Open Source CFD  |
4  |  \\    /   O peration          | Version:  1.6-ext                         |
5  |   \\  /    A nd                | Web:      www.extend-project.de           |
6  |    \\/     M anipulation       |                                           |
7  \*---------------------------------------------------------------------------*/
8  FoamFile
9  {
10     version     2.0;
11     format      ascii;
12     class       dictionary;
13     object      fvSchemes;
14 }
15 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
16
17 ddtSchemes
18 {
19     default steadyState;
20 }
21
22 gradSchemes
23 {
24     default          Gauss linear;
25     grad(p)          Gauss linear;
```

```
26      grad(U)         Gauss linear;
27  }
28
29  divSchemes
30  {
31      default         none;
32      div(phi,U)      Gauss upwind;
33      div(phi,k)      Gauss upwind;
34      div(phi,epsilon) Gauss upwind;
35      div(phi,R)      Gauss upwind;
36      div(R)          Gauss linear;
37      div(phi,nuTilda) Gauss upwind;
38      div((nuEff*dev(grad(U).T()))) Gauss linear;
39      div((grad(p)|A(U))) Gauss linear;
40  }
41
42  laplacianSchemes
43  {
44      default         none;
45      laplacian(nuEff,U) Gauss linear corrected;
46      laplacian((1|A(U)),p) Gauss linear corrected;
47      laplacian(DkEff,k) Gauss linear corrected;
48      laplacian(DepsilonEff,epsilon) Gauss linear corrected;
49      laplacian(DREff,R) Gauss linear corrected;
50      laplacian(DnuTildaEff,nuTilda) Gauss linear corrected;
51  }
52
53  interpolationSchemes
54  {
55      default         linear;
56      interpolate(U)  linear;
57      div(p)          linear;
58  }
59
60  snGradSchemes
61  {
62      default         corrected;
63  }
64
65  fluxRequired
66  {
67      default         no;
68      p;
69  }
70
71  // ************************************************************************* //
```

Listing 21: `fvSchemes` for `simpleFoam` and `pUCoupledFoam` cases.