university of groningen

University of Groningen

Master's Thesis Computer Science

---

# A software interface for fully implicit flow simulations on block-structured grids

---

*Author:*
Bob Dröge

*Supervisors:*
Dr. H. Bekker
Dr. ir. F.W. Wubs

February 2012

# Abstract

Recently, a new iterative domain-decomposition method has been developed which has been used to solve large linear systems arising from the spatial discretization of partial differential equations (such as the Navier-Stokes equations) by splitting them into many small subdomains. This method combines the advantages of direct and iterative solvers, which gives the method a lot of favorable properties.

We present a software interface for this solve method which is able to apply the solver to a wide range of problems in a distributed memory environment. Furthermore, our software interface does only require a (set of) partial differential equation(s), a suitable mesh and some parameters as input, so the user does not need to worry about subjects like discretization, solving equations and parallelization.

In this thesis, we give an overview of which mathematical and computational aspects are playing a role in typical fluid problems and how the software interface is built on top of some existing software packages. Finally, we show some results obtained by running the application on a computer cluster.

# Contents


**1 Introduction** **1**
- 1.1 Background . . . 1
- 1.2 Problem statement . . . 6
- 1.3 Thesis overview . . . 6

**2 Solver** **7**
- 2.1 Overview . . . 7
- 2.2 Implementation . . . 8

**3 Trilinos** **9**
- 3.1 Introduction . . . 9
- 3.2 Epetra . . . 9
- 3.3 CRS format . . . 10

**4 OpenFOAM** **12**
- 4.1 Introduction . . . 12
- 4.2 Discretization . . . 13
- 4.3 Capabilities . . . 14
- 4.4 Matrix structure . . . 14
- 4.5 Mesh . . . 14
- 4.6 Other dictionaries . . . 15
- 4.7 Dimensions . . . 19
- 4.8 File structure of OpenFOAM applications . . . 19
- 4.9 Example . . . 20

**5 Implementation** **22**
- 5.1 Idea . . . 22
- 5.2 Matrix conversion . . . 23
- 5.3 Explicit to implicit . . . 24

**6 Demonstration and results** **25**
- 6.1 Initial test results . . . 25
  - 6.1.1 Speedup on more cores . . . 29
- 6.2 Navier-Stokes equations . . . 30

**7 Conclusion** **33**
- 7.1 Future work . . . 34

**A Files of OpenFOAM's laplacianFoam example** **36**
A.1 laplacianFoam.C . . . . . 36
A.2 createFields.H . . . . . 38
A.3 write.H . . . . . 38
A.4 controlDict . . . . . 39
A.5 fvSolution . . . . . 41
A.6 transportProperties . . . . . 41

**B Dictionaries of Navier-Stokes example** **43**
B.1 0/p . . . . . 43
B.2 0/U . . . . . 44
B.3 constant/polyMesh/blockMeshDict . . . . . 45
B.4 constant/transportProperties . . . . . 46
B.5 system/controlDict . . . . . 47
B.6 system/fvSchemes . . . . . 48
B.7 system/fvSolution . . . . . 49

# Chapter 1

# Introduction

## 1.1 Background

An important field of mathematics and physics is fluid dynamics, which covers the study of the motion/flow of fluids. This involves nearly anything where some kind of gas or liquid in motion is involved.

For example, a typical application is the analysis of flow of air around (wings of) aircrafts in order to design the right shape of the wings and the aircraft itself. Of course, this example gives rise to a lot of other similar problems where air flows around objects. Furthermore, a very important, practical and daily returning problem is the prediction of the weather.

Typically, all these problems are described by one or more equations, which are often derived from some physical (conservation) laws, for example the conservation of mass, the conservation of momentum and the conservation of energy. These equations often include properties of the fluid, like its velocity, acceleration, pressure, temperature, etcetera. Some of these properties are described in terms of the motion itself: for example, the velocity of a motion is described in the change of the position over time; in mathematical terms, the velocity is the derivative of the position.

When, as in most cases, the equation contains derivatives of a certain unknown, we speak of a differential equation. More specifcally, since many of these variables change in each direction and in time, they usually have multiple derivatives describing all these changes for each direction (e.g. three derivatives for each direction when we consider a 3D problem) and over time. Each of these specific derivatives are called partial derivatives and differential equation involving such variables are called partial differential equations.

So suppose we are given a specific (partial) differential equation, inluding boundary/initial conditions, describing the flow of a fluid. How do we solve such an equation, i.e. how do we describe or predict the motion of the fluid? In some simple cases, we can solve the equation analytically, hence giving a function which describes the motion. This would of course be ideal.

But, unfortunately, in most of the typical applications, the equations describing the problem are too complex to be solved analytically, if possible at all. This is where a subfield of fluid dynamics turns up: computational fluid dynamics (CFD).

The focus of CFD is to use numerical methods to solve problems involving fluid dynamics. In other words, CFD tries to solve the aforementioned equations (which we cannot solve by hand) using some algorithm. In order to do this, the problem needs to be "converted" to a way such that it can be solved on a computer.

This "converting process" is called discretization. Where the original problem usually is defined on a continuous domain, computers only allow discrete problems because of its limited memory and processing time. So, an important part of the discretization process is to replace the initial, continuous domain by a discrete grid, which only describes the flow in its grid points. Hence, we need to come up with a grid size and define the distance between the grid points. If, as a simple example, our initial domain consists of a square area, we could come up with a grid of, for instance, $n \times n$ points, where $n$ is an integer.

Finding a suitable grid is not as easy as our example may suggests. The choice of the grid plays an important role in the final solution: obviously, a finer grid (i.e. a grid with more grid points) will give better results since it better approximates the real, continuous domain. On the other hand, a finer grid requires more memory and more calculations, hence it will take longer for the numerical method to find a solution. So, a good balance has to be found between between a fine and coarse grid and thus between speed and accuracy.

Furthermore, a domain may of course be more complex than the simple square area of our example. When we consider, for instance, a three-dimensional object like a car or the wing of an airplane, defining a grid will be far more complex.

Once we have a grid, we only define all variables/functions in our equation on the grid points. For some terms, this requires some additional work. For example, partial derivatives which describe changes of a variable in a certain direction, can be approximated by considering the changes in the neighbouring grid points.
In a similar way, we could do this for changes over time by looking at the value of points of the grid at previous time steps.
For all such terms, there are various mathods of how to approximate them using neighbouring points.

Another discretization step is the choice of a suitable time step. As we are calculating a numerical solution, we are only able to find the solution in a finite number of time values due to the aforementioned discrete nature of computers. So we need a certain time step which we use to hop through time. Every next step in time, we recalculate the solution of our equation on the grid. Again, by looking at solutions of previous time steps, we could use the change of these values to approximate derivatives describing changes in time.

Just as with the choice of the grid, also for the time step there exists a balance between speed and accuracy. Though a smaller time step gives better and more results, it will take longer to reach the end time.

The last thing we need in order to find a solution, is specifying boundary conditions. These conditions describe the initial state of the motion or conditions at the extremes of the domain (i.e. the boundaries).

So, let us combine everything we described so far and proceed to finding a solution. We start with some initial value for all variables in every grid point. Next, we continue to the next time step. Using all equations we should be able to calculate new values for all these variables. This process continues, until a certain time, state or other criterion is reached.

However, calculating solutions at every time step is not as easy as it may sound. Since there are usually many equations concerning many variables and many grid points, this is a complex and time-consuming job. To make this easier, we first linearize and combine all equations to a matrix equation $Ax = b$, where $x$ contains the unknowns in the grid points, $A$ its coefficients arising from the discretized equations and $b$ constant values. Linearisation is needed to approximate (non-linear) terms which contain, for instance, a product of two unknowns.

Important to notice is what the coefficients in the matrix $A$ represent. To illustrate their meaning, consider a (partial) differential equation involving the temperature $T$ discretized on a simple $10 \times 10$ grid where we number the grid points from left to right and from bottom to top. Then, our system of equations will look like:

$$\underbrace{\begin{pmatrix} \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \vdots & \vdots & \ddots & \vdots \\ \cdot & \cdot & \cdots & \cdot \end{pmatrix}}_{A \quad (100\times100)} \underbrace{\begin{pmatrix} T_1 \\ T_2 \\ \vdots \\ T_{100} \end{pmatrix}}_{x \quad (100\times1)} = \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{100} \end{pmatrix}}_{b \quad (100\times1)}$$

Now the $n$-th row in the matrix $A$ represents the discretized equation for $T_n$, i.e. the temperature in grid point $n$. As we mentioned before, this often involves neighbouring grid points. With the numbering we just introduced, this means we use the values of $T_{n-1}$ (left neighbour), $T_{n+1}$ (right neighbour), $T_{n-10}$ (bottom neighbour) and $T_{n+10}$ (top neighbour) in the equation for $T_n$. So, the coefficients of $A$ represent which neighbours influence the calculation of our unknown in a certain grid point and by which amount.
Though more complex/accurate discretizations might involve further neighbours, the number of influencing neighbours is usually still much smaller than the total number of grid points. From this we can make an important observation, namely that most values of each row, and hence of the entire matrix $A$, are zero. Such a matrix, which mainly consists of zeros, is called a sparse matrix.
For our simple example above with a $10 \times 10$ grid, we obtained a $100 \times 100$ matrix. In general, for $n \times n$ grids, the matrix will be at least $n^2 \times n^2$. For 3D problems or problems involving more equations, the matrix can be even larger. Therefore, storing and performing calculations with the full matrix would be

very costly. Fortunately, there exists various methods to store and perform operations on sparse matrices by only storing the non-zero elements and some information about their positions. Later on in this thesis, we will discuss some of these sparse matrix formats.

Now that we reduced our initial problem to a system of linear equations, finding a solution comes down to finding the solution $x$ of the system $Ax = b$. However, the larger and the finer our grid is, the more variables we have and the larger our system of equations become. Fortunately, there are various methods to find the solution $x$.
These methods, or solvers, can be divided into two different classes: direct solvers and iterative solvers. Iterative solvers start with an approximation of the solution $x$, try to estimate the difference between the approximation and the real result and then try to make a better approximation. This is repeated until the difference satisfies a given criterion and then the current estimation will be considered as the solution.
Direct solvers, on the other hand, use a method to find the solution in a limited number of foreknown steps. Usually, these methods split the matrix into a product of two matrices with a specific, triangular form. In this way, the solution can be more easily calculated, though obtaining these triangular matrices requires quite some calculations.
In comparison, direct solvers are very easy to use and very robust, while iterative solvers depend on many parameters and are usually far less robust. On the other hand, especially for larger matrices, iterative solvers are more memory and CPU efficient than direct solvers.

Everything we described so far, mainly concerned mathematics. There is, however, also an important computational aspect. As we already briefly mentioned before, the solving process is very time-consuming. Nowadays, most of the computers we have at home could do simple simulations over a small amount of time. But what if we want to do more complicated simulations, like predicting the weather or modelling an ocean? In these cases, even the fastest home computers lack speed and memory to do such simulations in reasonly amounts of time, i.e. within hours or days or even weeks.

There are some ways to speed up this process. A simple way might be to change parameters like the time step and the coarseness of the grid. However, this affects the accurateness of the solution, which we often do not want and is not feasible in case of, for instance, the aforementioned examples.

Fortunately, there are also computational ways to gain speed. Most of these methods involve parallel computing or parallelization, which is a way of doing computations simultaneously. Many of the currently available computers, or better said processors / central processing units (CPUs), already include two or more execution unites (cores). In this way, they are able to perform two or more instructions at once, which is one of the simplest forms of parallel computing. But, as said before, this still does not make home computers suitable for complex simulations. So, we need to take this way of parallel computing to higher levels.

A common way to do this is, is to connect multiple computers by some kind of network; this is a special kind of parellel computing called distributed computing. Again, this could be done in multiple ways, depending on the kind of network. For example, the famous supercomputer IBM Blue Gene is composed of a large number of subsystemts, each having its own CPU and memory. All these subsystems are connected by special, high-speed interconnect networks to communicate with each other. Typically, such a system contains far more than 1000 subsystems.

Another typical example of distributed computing and probably the most distributed one, is grid computing, where indivual computers are connected by and communicate over the internet. A famous example which uses this technology, is the FoldingHome project[1]. The goal of this project is to understand the (mis)folding of proteins and its related diseases. On the website of the project, everyone can download a client application and run it. The application periodically connects to the project servers to retrieve new instructions on which calculations it should perform. After it is done, it sends back the results and continues with the next calculations. This way, everyone can contribute to the project.

Whichever way of parallel computing is used, there is one problem. The application concerned, should be capable for parallel computing. When the application is written, one should take this into account and make the calculations suitable for parallel computing. Fortunately, there are a lot of methods/packages which help to divide calculations over multiple processors/computers.
However, besides making the application itself suitable to be run in parallel, the tasks it should perform should also be suitable to be executed parallel. Ideally, one may expect that an application runs twice as fast when there are twice as many processors, but in practice, this is rarely true. Often, applications contain parts which cannot be parallelized, for example when parts/calculations depend on each other and cannot be executed independently.

An important part of parallel computing is the communication. After all, each processor or each (sub)system should somehow get to know what to do. For example, suppose that we have a cluster of systems on which we want to run a parallellized application consisting of a lot of matrix operations. If we start the application on one of the systems, this system should first tell every other system exactly what to do. Besides, it should also send the necessary part of the (matrix) data to each of the corresponding systems. When the calculations are done, all results have to be gathered. All in all, this simple example already illustrates that a parallel application requires a lot of communication.
Fortunately, there is an application programming interface (API) which makes this communication much easier, the message passing interface (MPI). This communication protocol supports all different kinds of programming languages and is the de-facto standard for parallel computing on distributed memory systems, i.e. systems where each subsystem/processor has its own memory. MPI more or less allows all sorts of communications needed for a parallel application, from sending, receiving and combining data to synchronizing systems or obtaining

[1] http://folding.stanford.edu

information about, for example, the number of processors/systems.

To conclude this introduction, this Masters' project/thesis uses all of the concepts we have discussed so far. Now that we are a little bit familiar with each of them, we can continue to define the main problem of this thesis.

## 1.2 Problem statement

As we pointed out in the previous section, the solver is the important part of finding a solution of a partial differential equation. Recently, a new method has been developed [7, 6, 5] which has been used to solve large linear systems arising from the spatial discretization of partial differential equations by splitting them into many small subdomains. This method has some favourable properties, which we discuss later.

The main goal of this Master's project/thesis is to develop, test and discuss a software interface for this solver.

By implementing/reusing the solver and by constructing an interface for this solver, it should be possible to easily apply the method to a wide range of problems, defined on a simple shape, in a distributed memory environment. So, the user should only have to define a suitable problem, but does not need to worry about, for instance, parallelization.

We can divide the problem into some subproblems: first, we need a method or interface which allows for easy input of the problem, i.e. the concerning equations, boundary conditions, grid, etcetera. The next step is to discretize the problem to obtain a systems of equations of the form $Ax = b$. Then we are able to apply our parallellized solver to solve this system of equations. Since the solver makes use of special numerical toolkits (and hence, special structures of, for example, matrices), some converting needs to be done to pass the problem to the solver in a correct (sparse) format. Finally, as a demonstration of concept, the solver should then be used to solve one or two large-scale problems on a parallel computer.

## 1.3 Thesis overview

In chapter 2 we start by briefly discussing the given solver, which is the basis of our project. Next, in chapter 3 and 4 we discuss two important packages/-modules, OpenFOAM and Trilinos respectively, which we use in our software. Chapter 5 gives a detailed overview of our software interface, including some initial results and problems. In chapter 6 some demonstrations and results of our software interface and solver are given. Finally, in chapter 7, we conclude this thesis by summarizing the results we obtained and pointing out possible future work.

# Chapter 2

# Solver

As this project is mainly built upon an already existing solver, we first give a brief overview of the solver in this chapter. For a more detailed view, we refer to the corresponding papers by F.W. Wubs and J. Thies[7, 6] and the PhD thesis of J. Thies[5].

## 2.1 Overview

As we pointed out in the introduction, the most time-consuming part of a typical computational fluid dynamics problem is the step where a large linear system of equations has to be solved. Typically, such a CFD problem may consist of millions or more unknowns. This is why there exist many different types of solvers, suitable for many types of different problems.

Also, we already mentioned that all of these solvers can roughly be classified into two different types, direct solvers and iterative solvers. Where direct solvers are usually chosen because of their robustness and high accuracy, they tend to reach memory limits soon when increasing the size of the problem. Also, computing time may increase fast.
Iterative methods, on the other hands, are usually faster and less memory consuming than the direct methods, but they lack the robustness of the direct methods. Possibly, they do not converge and, hence, give an inaccurate result/-solution in certain cases.

The solver we are using, combines the best of both kinds of solvers: the robustness of direct solver and memory and computational efficiency of iterative solvers. Specifically, it has the following favourable properties:

- the solver is very robust, even close to the point where the solution becomes unstable;
- proven robustness for Stokes equation;
- a single parameter controls the fill-in (i.e. the entries of the matrix which change from zero to non-zero during the solving process) and convergence, making the method straightforward to use;

- the convergence rate is independent of the number of unknowns;
- it can be parallellized in a natural way.

In the remaining of this thesis, we will refer to this solver by its name HYMLS, which is an abbreviation for **hy**brid **m**ulti**l**evel **s**olver.

## 2.2 Implementation

HYMLS was first implemented in Matlab in a sequential/serial way. As mentioned before, a lot of speed could be gained by parallellizing it and running it on a parallel computer. Therefore, HYMLS was later rewritten in C++ by using the Trilinos package, which adds a lot of mathematical functionality. In particular, the linear algebra packages of Trilinos are very useful in this case as they support all different kinds of (sparse) matrix operations and manipulation which are needed to solve a system of linear equations and new functions can easily be implemented on top of existing ones. Furthermore, parallel computing is supported and can be enabled with one simple function call. In Chapter 3, we will have a closer look at Trilinos and its packages.
HYMLS is parallellized by splitting up the physical domain into subdomains and the matrix into submatrices, such that each of the resulting "subdomain matrices" can be handled by different processors.

Furthermore, the implementation of HYMLS is kept very general and not aimed at a specific type of discretization or partial differential equation. In principle, all input that is required to give a solution, is a matrix, some information on the structure of the problem (the type of equation, degrees of freedom, number of subdomains, etcetera) and a right-hand side vector. The only assumptions which are made is that the problem is defined on some grid, where each cell of the grid has the same number of unknowns and the unknowns are ordered per cell.

Besides this generality of HYMLS, the implementation of HYMLS is also set up in a modular way. The solver consists of various parts, from decomposition and partitioning of the domain and variables to preconditioners and subsolvers which solve different kind of equations. The modularity of the implementation means that the main parts of the algorithm, e.g. the partioning method or the direct solver, are replaceable in an easy manner.

Finally, by using the Trilinos package, HYMLS makes use of existing code (mainly Trilinos) as much as possible. Since Trilinos is a well maintained and documented package, this is more favourable and less error prone than writing own code.

# Chapter 3

# Trilinos

Since our solver HYMLS uses a lot of Trilinos methods, we give a brief overview in this chapter of what Trilinos is and describe some of its capabilities and features.

## 3.1 Introduction

"The Trilinos Project is an effort to develop algorithms and enabling technologies within an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems. A unique design feature of Trilinos is its focus on packages."[1]

In other words, Trilinos is an open source collection existing of many packages which are building blocks for scientific applications. There are packages for

- basic linear algebra,
- preconditioning,
- solving (systems of) equations,
- optimization problems,
- automatic differentation,

and more. Most of these packages are written in C++ and, furthermore, designed for use on distributed memory parallel architectures.

Our solver uses many Trilinos functions to manipulate matrices and to do other linear algebra operations. The main Trilinos package which allows many of these features, is called Epetra, at which we have a look in the next section.

## 3.2 Epetra

Epetra is the basic Trilinos package, necessary for all basic linear algebra operations. For exmaple, it contains the methods for creating and manipulating

[1] http://trilinos.sandia.gov/

vectors, matrices and graphs, for both serial and parallel computing. Multiple kinds/structures of matrices are supported. Our solver mainly uses the sparse matrix class called `Epetra_CrsMatrix`, which stores matrices in a Compressed Row Storage format.

## 3.3 CRS format

The CRS format[2] is a way to store sparse matrices. The advantages of this format are that it does not store any unnecessary element and that it makes no assumptions about the sparsity structure of the matrix. However, the CRS format is not a very efficient one, compared to other sparse formats.

But since Trilinos, and hence our solver, makes use of this format, we will have to be sure to pass all (matrix) parameters from our interface to the solver in the right format. Therefore, we have a further look in the underlying structure of CRS matrices.

Basically, a matrix in CRS format is stored by three vectors. The first vector, say `values`, contains all non-zero matrix values in a row-wise fashion, i.e. first all non-zero elements of the first row, then all the ones of the second row, etcetera.
The second vector, say `col_ind`, stores all column indices of the corresponding values in the `val` vector, i.e. `col_ind(i)` gives the column index of `values(i)`. Finally, the third vector, say `row_ptr`, is used to reproduce which items belong to which row by storing the locations in the `values` vector which start a new row. By convention, one last element is added to this vector with the value $nnz + 1$, where $nnz$ is the number of non-zero elements in our matrix.

So, as a simple example, consider the next matrix:

$$\begin{pmatrix} 1 & 2 & 0 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 5 & 6 & 7 \\ 0 & 0 & 0 & 8 \end{pmatrix}$$

Suppose we want to convert this matrix to CRS format. Our `values` vector would simply consist of all non-zero elements, i.e. `values` $= \{1, 2, 3, 4, 5, 6, 7, 8\}$. Next, the `col_ind` vector stores the corresponding column indices, so it is given by $\{0, 1, 1, 2, 1, 2, 3, 3\}$, assuming that the first column index is 0. Finally, we need pointers to the values starting a new row: in our case, the values which start a new row are 1,3,5 and 8. In the `values` vector, these values are on positions 0, 2, 4 and 7, respectively. So, our $row\_ind$ vector is given by $\{0, 2, 4, 7, 8\}$.

Finally, let us have a short look at what we gain in terms of storage for a $n \times n$ matrix with $nnz$ non-zero elements. Both the `values` and the `col_ind` vector store information about the non-zero elements and, hence, are of length $nnz$. The `row_ptr` vector contains one element per row; so, including the one element extra which we mentioned before, it is of length $n + 1$. Hence, the number of values we have to store (i.e. the sum of the lengths of the three vectors) is given by $2nnz+n+1$. The original format, on the other hand, would require storing $n^2$ elements. Clearly, the more non-zero elements the matrix

contains, i.e. the sparser the matrix is, the less storage is required when storing the matrix in this format.

# Chapter 4

# OpenFOAM

In this chapter we give a brief overview of OpenFOAM, a software package which we will use in our interface.

## 4.1 Introduction

OpenFOAM is a more or less comparable software package as Trilinos, but more specfically intended for computational fluid dynamics. The main feature of OpenFOAM is its capability "to solve anything from complex fluid flows involving chemical reactions, turbulence and heat transfer, to solid dynamics and electromagnetics."[1]

Also, OpenFOAM is free and open source and contains a huge set of C++ modules. Though OpenFOAM also contains a lot of pre-configured solvers, we will not use these, as we already have our own solver. Then why use OpenFOAM?

The answer lies in the way OpenFOAM represents the equations to be solved. The OpenFOAM documentation shows the following convincing example. Consider the equation:

$$\frac{\partial \rho U}{\partial t} + \nabla \cdot \phi U - \nabla \cdot \mu \nabla U = -\nabla p.$$

In OpenFOAM, this equation is directly represented (and solved) by the following piece of code:

```
solve
(
    fvm::ddt(rho, U)
  + fvm::div(phi, U)
  - fvm::laplacian(mu, U)
    ==
  - fvc::grad(p)
);
```

[1] http://www.openfoam.com/

Trilinos, for example, does not offers such features. Hence, OpenFOAM seems an ideal software package to use in our interface for representing and discretizing equations. In the next sections, we give a more detailed view of the capabilities and features of OpenFOAM.

## 4.2 Discretization

In order to solve a given partial differential equation (PDE), OpenFOAM discretizes the PDE into a set of algebraic equations. Each term in the PDE is represented individually by a specific function in either one of the OpenFOAM classes `fvm` or `fvc`: functions of `fvm` (finite volume method) calculate implicit derivatives and return a special OpenFOAM matrix object, while functions of `fvc` (finite volume calculus) calculate explicit derivatives and return a special OpenFOAM field/mesh.

Both the `fvm` and `fvc` class offers functions for discretizing terms like a Laplacian, (second) time derivative, convection and source. But, in addition, the `fvc` class also has functions for divergence, gradient and curl terms.

Here, we immediately encounter a problem. Since we need to discretize our PDE into a matrix form in order to pass it to our solver, we want to make use of the `fvm` class. However, without the ability to use divergence and gradient terms, this would be useless. Hence, we need to find a way to convert the results of the `fvc` functions, which do offer support for these terms, into matrix form. How this is done, will be discussed later.

| Term description | Implicit/ Explicit | Text expression | Function |
|---|---|---|---|
| Laplacian | Imp/Exp | $\nabla^2\phi$<br>$\nabla \cdot \Gamma\nabla\phi$ | `laplacian(phi)`<br>`laplacian(Gamma, phi)` |
| Time derivative | Imp/Exp | $\frac{\partial\phi}{\partial t}$<br>$\frac{\partial\rho\phi}{\partial t}$ | `ddt(phi)`<br>`ddt(rho, phi)` |
| Second time derivative | Imp/Exp | $\frac{\partial}{\partial t}\left(\rho\frac{\partial\phi}{\partial t}\right)$ | `d2dt2(rho, phi)` |
| Convection | Imp/Exp | $\nabla \cdot (\psi)$<br>$\nabla \cdot (\psi\phi)$ | `div(psi, scheme)`<br>`div(psi, phi, word)`<br>`div(psi, phi)` |
| Divergence | Exp | $\nabla \cdot \chi$ | `div(chi)` |
| Gradient | Exp | $\nabla\chi$<br>$\nabla\phi$ | `grad(chi)`<br>`gGrad(phi)`<br>`lsGrad(phi)`<br>`snGrad(phi)`<br>`snGradCorrection(phi)` |
| Grad-grad squared | Exp | $\lvert\nabla\nabla\phi\rvert^2$ | `sqrGradGrad(phi)` |
| Curl | Exp | $\nabla \times \phi$ | `curl(phi)` |
| Source | Imp | $\rho\phi$ | `Sp(rho, phi)` |
| | Imp/Exp | | `SuSp(rho, phi)` |

**Table 4.1:** Main functions available in the `fvc` and `fvm` classes.

## 4.3 Capabilities

OpenFOAM is capable of solving any type of equation concerning complex flows involving chemical reactions, turbulence and heat transfer. But also, solvers for solid dynamics and electromagnetics are available.

To give an idea of which terms are available in the aforementioned `fvc` and `fvm` classes, Table 4.1, copied from the OpenFOAM Programmer's Guide[1], lists their main functions.

## 4.4 Matrix structure

Just like Trilinos, also OpenFOAM uses its own way to store matrices. Specifically, a matrix type/class called `lduMatrix` is used which contains three arrays to store the non-zero elements of the matrix: one for the **l**ower triangle elements, one for the **d**iagonal elements and one for the **u**pper triangle elements. Additionally, two addressing arrays, `lowerAddr` and `upperAddr`, are used to store the positions of the elements of the lower and upper triangle. The position (row, column) of each element can be retrieved in the following way:

| Value | Row | Column |
|---|---|---|
| diagonal(i) | i | i |
| upper(i) | upperAddr(i) | lowerAddr(i) |
| lower(i) | lowerAddr(i) | upperAddr(i) |

So, as a simple example, consider the following matrix:

$$\begin{pmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 5 & 0 \\ 0 & 6 & 7 & 8 \\ 0 & 0 & 9 & 10 \end{pmatrix}$$

This matrix is represented by the following arrays:

```
diagonal  = [1,4,7,10]
upper     = [2,5,8]
lower     = [3,6,9]
upperAddr = [0,1,2]
lowerAddr = [1,2,3]
```

Note that we, again, start indices of rows and columns at zero and that the `upper` and `lower` arrays store the values of the corresponding triangles in a row-wise order.

In order to convert the OpenFOAM matrix to the Trilinos CRS format, we have to use this addressing to find the positions of all elements and add them to the Trilinos matrix. How this is exactly done, is discussed in the next chapter.

## 4.5 Mesh

OpenFOAM also offers a way, including some utilities, to generate a mesh. By making a mesh dictionary, one can specify the characteristics of the mesh. The

principle is to decompose the domain into one or more 3D blocks. Edges of blocks can be straight lines, arcs or splines. Each edge is formed by joining two vertices. Finally, patches can be specified as faces of the blocks and can be used to define boundaries (and, hence, boundary conditions).

As a simple example, Listing shows the dictionary for a simple cuboid/box. The first four vertices can be considered as the vertices of a rectangle in the plane $z = 0$, while the following four vertices are the same rectangle but in the plane $z = 0.1$. In the next part, a block is defined over all eight vertices, which results in a cuboid or box. The parameters specify the number of cells in the x, y and z direction and the type of expansion in each direction. `simpleGrading` results in a uniform expansion.
Next, there are no edges specified. Because we do not need any curved edge, this is not necessary in our example.
Finally, each face of our single block is specified by using the vertex numbers; the vertex specified first, is number 0, the next one is vertex 1, etcetera.

When the dictionary is finished, the mesh can be generated by running the utility `blockMesh`. This will generate some files which store the exact coordinates of all points, faces, cells, etcetera, and will be used in the discretization process.

## 4.6 Other dictionaries

Besides a dictionary for defining a grid, OpenFOAM uses some additional dictionaries to define some other settings and parameters.

First, there is a "schemes" dictionary that defines which numerical schemes should be used for all different terms in the equation. A simple part of such a file is shown in Listing 4.2.

Next, there is a dictionary for specifying which solver and which corresponding parameters OpenFOAM should use to solve the equation. To demonstrate how this dictionary often looks like, a part of a simple example is given in Listing 4.3. However, since we are going to use our own solver, we will not really need this dictionary.

Furthermore, OpenFOAM uses a control dictionary which defines more general parameters, such as a start and end time, a time step, etcetera. As an example, (a part of) such a control dictionary is shown in Listing 4.4.

Finally, there should be a file which contains the initial values and boundary conditions for the particular problem/equation. Such a file should look like the one shown in Listing 4.5.

```
vertices
(
    (0 0 0)
    (10 0 0)
    (10 1 0)
    (0 1 0)
    (0 0 0.1)
    (10 0 0.1)
    (10 1 0.1)
    (0 1 0.1)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (50 20 1) simpleGrading (1 1 1)
);

edges
(
);

patches
(
    wall topWall
    (
        (3 7 6 2)
    )
     wall bottomWall
     (
              (1 5 4 0)
     )
    wall inlet
    (
        (0 4 7 3)
     )
     wall outlet
     (
        (2 6 5 1)
    )
    empty frontAndBack
    (
        (0 3 2 1)
        (4 5 6 7)
    )
);
```

**Listing 4.1:** Simple example of an OpenFOAM mesh dictionary file.

```
ddtSchemes
{
    default           Euler;
}

gradSchemes
{
    default           Gauss linear;
    grad(T)           Gauss linear;
}

laplacianSchemes
{
    default           none;
    laplacian(DT,T)   Gauss linear corrected;
}
```

**Listing 4.2:** Simple example of an OpenFOAM dictionary for choosing numerical schemes.

```
solvers
{
    T
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0;
    }
}
```

**Listing 4.3:** Simple example of an OpenFOAM dictionary for choosing a solver and its parameters.

```
application     laplacianFoam;
startFrom       latestTime;
startTime       0;
stopAt          endTime;
endTime         3;
deltaT          0.005;
writeControl    runTime;
writeInterval   0.1;
purgeWrite      0;
writeFormat     ascii;
writePrecision  6;
writeCompression uncompressed;
timeFormat      general;
timePrecision   6;
runTimeModifiable yes;
```

**Listing 4.4:** Simple example of an OpenFOAM control dictionary.

```
internalField   uniform 273;

boundaryField
{
    patch1
    {
        type            zeroGradient;
    }

    patch2
    {
        type            fixedValue;
        value           uniform 273;
    }

    patch3
    {
        type            zeroGradient;
    }

    patch4
    {
        type            fixedValue;
        value           uniform 573;
    }
}
```

**Listing 4.5:** Simple example of an OpenFOAM file which specifies initial values and boundary conditions.

## 4.7 Dimensions

OpenFOAM attaches a dimension to all field data and physical properties. In this way, a dimension check can be performed on every operation, to prevent meaningless operations. So, when one wants to define the value of a scalar, field, etcetera in a certain dictionary file, also the dimension should be given. This can be done by specifying the powers of each SI unit between square brackets. The order of the units is given in Table 4.2. As an example, consider:

$$[0 \;\; 2 \;\; -1 \;\; 0 \;\; 0 \;\; 0 \;\; 0].$$

Here, the 2 on the second position corresponds to $m^2$, while the $-1$ on the third position corresponds to $s^{-1}$. Hence, the dimension specified is $m^2/s$.

| No. | Property | SI unit |
|---|---|---|
| 1 | Mass | kilogram (kg) |
| 2 | Length | metre (m) |
| 3 | Time | second (s) |
| 4 | Temperature | Kelvin (K) |
| 5 | Quantity | kilogram-mole (kgmol) |
| 6 | Current | ampere (A) |
| 7 | Luminous intensity | candela (cd) |

**Table 4.2:** Specifying dimension in OpenFOAM.

## 4.8 File structure of OpenFOAM applications

We now know which basic files need to be present in order to create and run an OpenFOAM application. All of these files need to be on a specific place such that OpenFOAM can find them. Therefore, OpenFOAM uses a prescribed file structure, which every application should meet in order to run. This structure is displayed in Figure 4.1.

The root folder of the application contains, besides the main files of the application, three directories. First, there is a `system` directory that contains the dictionaries which specify the numerical schemes, solver settings and control dictionaries. Next, there is a constant directory which could contain files specifying physical properties. This directory also contains a subdirectory `polyMesh` where the grid (probably/usually generated by the `polyMesh` application) files are stored. Finally, there should be a time directory where the files with initial values and boundary conditions are stored. The name of the time directory should be the value of the start time, e.g. just `0`. The files inside it should have the name of the variable for which it contains data, e.g. `T`, `p`, etcetera.

When the application runs, OpenFOAM will create similar time directories for the solutions/values of all unknowns at each time step. Hence, again these directory will just carry the name of the current time. The files inside again have the name of the corresponding variable and contain the values of this variable at every grid point.

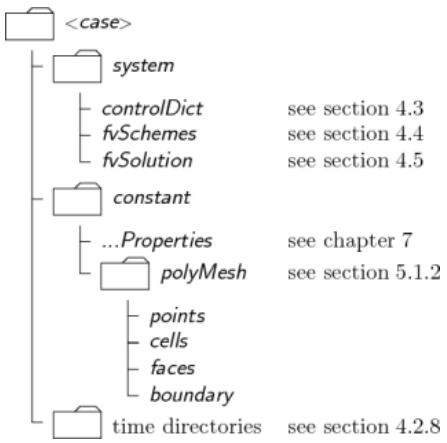



**Figure 4.1:** File structure for OpenFOAM applications.

## 4.9 Example

One of the simplest examples included in OpenFOAM, is the `laplacianFoam` application. The code and other relevant files of this application are given in Appendix A. We will shortly discuss the application.

Though the name of the application may suggest otherwise, the equation which is solved by this application is not simply Laplace's equation. Instead, the following equation is solved, which describes thermal diffusion in a solid:

$$\frac{\partial T}{\partial t} + \nabla^2 \left( D_T \cdot T \right) = 0$$

The main code which solves this equation, is given in `laplacianFoam.c` in Appendix A. As can be seen, the relevant code consists of only a few lines. The first lines of the code only exists of some includes which are necessary for each OpenFOAM application. All lines starting with `Info` are used to write output to the console. So, the actual code which solves the equation, can be reduced to:

```
while (runTime.loop()) {
  for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++) {
    solve( fvm::ddt(T) - fvm::laplacian(DT, T) );
  }
}
```

First, the `while` loop, loops over the time, as defined in the control dictionary (also given in Appendix A). Next, the `for` loop, is part of the Semi-Implicit Method for Pressure-Linked Equations (SIMPLE) algorithm[4] which solves the equation. The number of iterations of this loop, `nNonOrthCorr`, is specified in the solver dictionary (see `fvSolution` in Appendix A) and depends on the degree of non-orthogonality of the mesh. Finally, the code which actually solves the equation, is within this `for` loop and is, again, a quite straightforward representation of the partial differential equation.

One of the included files is `createFields.H`, which nearly every OpenFOAM application needs to define the variables used in the equation. The version for our example is also given in Appendix A and shows how the variables `T` and `DT` are defined. The first one is a `volScalarField`, which is a field defined at all cell centres of the mesh. Also, it is stated that the initial values of the field should be read from a file named `T`, as we discussed in a previous section.
Similarly, `DT` is defined as a constant and should be read from a dictionary file named `transportProperties`, which should be present in the `constant` directory as we also mentioned in a previous section. The contents of this file is also shown in Appendix A. Since `DT` represents the thermal diffusivity, its dimension is specified as $m^2/s$.

Another included header file, which is very similar to the previous one, is `write.H`. This file is generally used to define extra values/fields which should be stored at every time step in the corresponding time directory. In the case of our example (again, see Appendix A), this file states that the $x$, $y$ and $z$ components of the gradient of $T$ should be written to a file at every time step. Since these do no exist yet, they first have to be created: $\nabla T$ itself is a vector field as it contains components for each direction, so it is defined as a `volVectorField`. The components itself just contain scalars, hence they are defined as a `volScalarField`. So, after running the example application, each time step directory in the root folder of the application, should now contain four files: `gradTx`, `gradTy`, `gradTz` and, of course, `T` itself.

To conclude this example application, note that we did not list the code of all files. For example, the dictionaries for the mesh, initial values and numerical schemes are not listed in Appendix A. These files are not very relevant to demonstrate the example. Also, we do not discuss the results/output of the application, as we only want to demonstrate some example code. In a later chapter, we will look at results of our complete application.

# Chapter 5

# Implementation

Now that we have discussed all necessary and useful software, we will discuss the interface to our solver in this chapter. First, we give a brief overview of the idea of our interface. Next, we discuss the test results of the initial version and the problems we encountered. Finally, we describe how these problems are solved.

## 5.1 Idea

The main idea of our interface is a logical result of what we have discussed in the previous chapters. The big advantage of OpenFOAM, the direct representation of equations, could be used to define the equation/problem and to generate a suitable mesh. However, our solver is based on Trilinos, which uses a different representation/storage of matrices than OpenFOAM does. So, some kind of conversion is necessary to pass the results of the OpenFOAM discretization to our solver.

So, the blueprint of our software is as follows:

1. define the mesh in a mesh dictionary;
2. generate the mesh using the `blockMesh` utility;
3. define the equation in an OpenFOAM manner;
4. get the result of the discretization;
5. convert the result to a Trilinos (Epetra) CRS format;
6. call the solver with the converted parameters;
7. convert the solution back to OpenFOAM format in order to store and reuse the solution.

## 5.2 Matrix conversion

An important step in the software is to convert the OpenFOAM output to Trilinos format. We already discussed the details of this specific format in Chapter 3.

Fortunately, the relevant Trilinos package, Epetra, offers quite a lot functions to create a matrix in CRS format. More specific, there is a method to generate an empty CRS matrix, a method to insert a single element into the matrix and, finally, a method to indicate that the data entry is complete. Since also the OpenFOAM matrix class offers easy access to the elements of the matrix, the conversion can be easily done by following the next steps:

1. loop over the elements of the three arrays which store the OpenFOAM matrix (see previous chapter);
2. find the position (i.e. row and column) of the element by using the addressing arrays;
3. insert the element into the CRS matrix by specifying its value and position (row and column);
4. when finished, call the `FillComplete` method to indicate that all elements are inserted into the matrix.

Although the matrix is in the right sparse format now, we still have to make some changes in order to make the solver accept our matrix. As we mentioned in Chapter 2, the solver assumes that our unknowns are ordered per cell. OpenFOAM uses a different order, which we illustrate by a simple example. Suppose we have an equation involving the unknowns $u, v, w$ and $p$, which represent the three components of the flow velocity and the pressure, respectively. Furthermore, assume that we have a certain numbering of our $N$ cells and that the values of these unknowns in the $i$-th cell are given by $u_i, v_i, w_i, p_i$ for some integer $1 \leq i \leq N$. Now, consider our system of equations $Ax = b$ which we want to solve, where $x$ is our vector containing the unknowns. In OpenFOAM, this vector is now given by

$$(u_1, u_2, \ldots, u_N, v_1, v_2, \ldots, v_N, w_1, w_2, \ldots, w_N, p_1, p_2, \ldots, p_N)^T .$$

Instead, we want this vector to be ordered as

$$(u_1, v_1, w_1, p_1, u_2, v_2, w_2, p_2, \ldots, u_N, v_N, w_N, p_N)^T .$$

Obviously, we cannot just change the order of our vector $x$, without changing the matrix $A$ and right-hand side vector $b$. So, we first create an array `perm` which stores the new order of the rows of $x$. This order is calculated automatically, depending on the number of cells and number of unknowns. Once we have this array, we use the `Permutation` class in the Trilinos package EpetraExt to create a permutation matrix $P$, which is an identity matrix with its rows reordered. The position of the ones are determined by our `perm` array in the following way: $P(i, perm(i)) = 1$. Now, if we multiply this permutation matrix $P$ with our right-hand side vector $b$, we obtain a new permuted vector $b_P$ which has the right order for our solver. For our matrix $A$, we do not only have to permute

the rows, but also the columns, which is done by multiplying with both $P$ itself and its inverse/transpose $P^T$, hence: $A_P = PAP^T$.

Now we are ready to apply our solver on the permuted matrix $A_p$ and permuted right-hand side vector $b_P$. This yields a solution vector $x_P$, which we eventually can permute back to OpenFOAM format by multiplying with the inverse/transose of P, hence: $x = P^T x_P$.

## 5.3 Explicit to implicit

The final section of this chapter deals with an issue which prevents us from using certain operators in the equations to be solved, especially for more complex equations. As can be seen in Table 4.1, OpenFOAM cannot discretize every term implicitly. More concrete, all `fvc` functions are calculated explicitly and returned as an OpenFOAM `Field`, which is a data structure containing the calculated values of the unknown at every cell for the corresponing term/operator. For example, `fvc::grad(p)` returns a `Field` with the calculated gradient of the pressure at every cell of our mesh. Since we want the entire (set of) equation(s) in a matrix-vector equation, we do need every term to be discretized implicitly, which is a problem for the divergence and gradient terms. Therefore, we need to manually implement these functions.

To do this, we make use of the linearity of the gradient and divergence functions. For every linear function, a transformation matrix can be determined by applying the function to each vector of a standard basis and using the results as the columns of the transformation matrix.
In our case, this is somewhat more tedious, since the gradient function returns a vector (usually a tuple since OpenFOAM assumes every problem to be three-dimensional) of values for every point with the approximate values of all partial derivatives. Hence, if we calculate the gradient for a given vector, OpenFOAM returns a vector of tuples.
So instead, we generate seperate matrices for each of the components of the gradient by using the aforementioned property. In other words, if we want to build a matrix $G$ for our gradient function $g(\vec{v})$, we first consider the $\frac{\partial}{\partial x}$ part of the gradient to determine $G_x$:

$$G_x = \begin{pmatrix} g_x(\vec{e_1}) & g_x(\vec{e_2}) & \cdots & g_x(\vec{e_N}) \end{pmatrix},$$

where $\vec{e_i}$ is the $i$-th column of the identity matrix $I_N$ and $g_x(\vec{v})$ returns the approximation of $\frac{\partial v}{\partial x}$.
In a similar way we can determine $G_y$ and $G_z$, which gives us the implicit representation of our gradient by

$$G = \begin{pmatrix} G_x \\ G_y \\ G_z \end{pmatrix}.$$

Since the divergence is just the transpose of the gradient, its matrix can now simply be determined by

$$D = (G_x G_y G_z).$$

# Chapter 6

# Demonstration and results

In the previous chapters we demonstrated how our software interface is built and which problems had to be solved. Now we can show some results by specifying equations in OpenFOAM format, doing all necessary conversion steps to obtain a suitable Trilinos matrix and, finally, applying our solver.

We first try our implementation on a very simple equation and on a very small grid to demonstrate that the conversion between the different formats actually works. Next, we will expand this equation to larger grids. Finally, we will show an even larger problem with more degrees of freedom.

## 6.1 Initial test results

As a first attempt to test our implementation, we consider Laplace's equation

$$\nabla^2 T = 0$$

on the domain $[0,1] \times [0,1]$, with the following boundary conditions:

$$\begin{aligned} T(0,y) &= 0, \\ T(1,y) &= 0, \\ T(x,0) &= 0, \\ T(x,1) &= 1. \end{aligned}$$

In other words, we consider a unit square which is heated from the top.

First, we create a mesh which represents the domain. The code of our mesh dictionary is given in Listing 6.1. In OpenFOAM, every mesh should be 3D, so we specify a unit cube and choose, for example and simplicity, only ten cells in both the $x$ and $y$ direction and just one in the $z$ direction. Also, we set the top and bottom face of the unit cube to be "empty", which instructs OpenFOAM that these are the planes normal to our 2D surface and, hence, that it should only solve in two dimensions. We define the left, right and bottom boudaries as `fixedWalls`, while the boundary at the top is named `heater`.

Next, we specify the boundary conditions in the appropriate dictionary. Since we gave all our boundaries suitable names in the previous step, we can now easily define the conditions on each of these boundaries. Also, the initial temperature within the square area is defined. Listing 6.2 gives the complete dictionary.

All other dictionary files do not really matter in order to test our implementation and, furthermore, they are very similar to the ones we have shown in previous examples. Hence, we do not list them here.

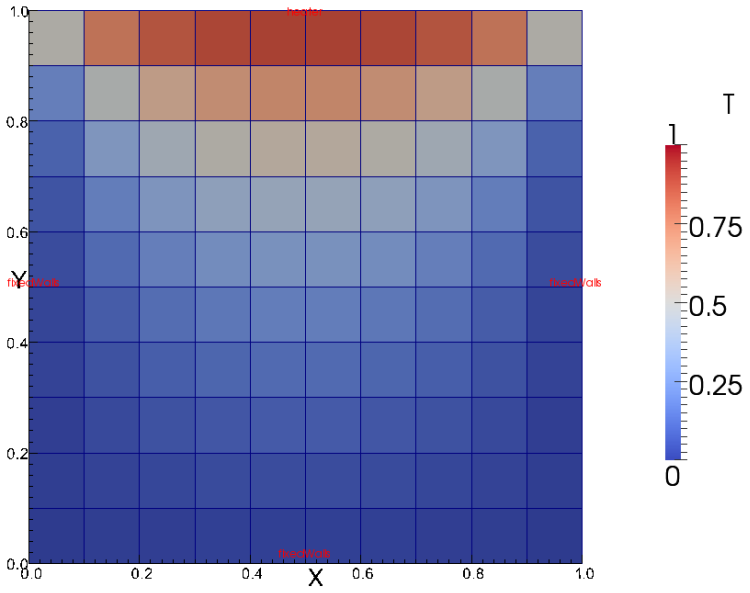



**Figure 6.1:** Numerical solution of Laplace's equation for a 10x10x1 mesh representing a unit square heated from the top.

Running our application for this example on a single core of a today's desktop computer, takes only several milliseconds due the very limited number of mesh cells and degrees of freedom. After running this code, the output is written to a file in OpenFOAM format. By using `paraFoam`, which is an OpenFOAM script that launches ParaView[1] and automatically loads an OpenFOAM generated solution, we are able to visualize our example in an easy manner. The result is shown in Figure 6.1. The result not only looks like a plausible solution; if we compare our solution to a solution generated by using an OpenFOAM solver, the norm of the difference is approximately $2.5 \cdot 10^{-6}$. Since the convergence tolerance of both solvers was $10^{-6}$, this difference is not significant. Therefore, we can conclude that our solution is correct and the conversion of our matrix must have succeeded.

[1] http://www.paraview.org/

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}
convertToMeters 1;

vertices
(
    (0 0 0)
    (1 0 0)
    (1 1 0)
    (0 1 0)
    (0 0 0.1)
    (1 0 0.1)
    (1 1 0.1)
    (0 1 0.1)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (10 10 1) simpleGrading (1 1 1)
);

patches
(
    wall fixedWalls
    (
        (0 4 7 3)
        (2 6 5 1)
        (1 5 4 0)
    )
    wall heater
    (
        (3 7 6 2)

    )
    empty frontAndBack
    (
        (0 3 2 1)
        (4 5 6 7)
    )
);

mergePatchPairs
(
);
```

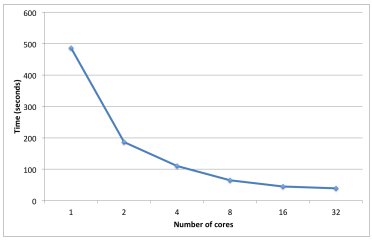
**Listing 6.1:** Mesh dictionary for a unit square heated from the top.

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      T;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 0 0 1 0 0 0];

internalField   uniform 0;

boundaryField
{
    fixedWalls
    {
        type            fixedValue;
        value           uniform 0;
    }

    heater
    {
        type            fixedValue;
        value           uniform 1;
    }

    defaultFaces
    {
        type            empty;
    }
}

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

**Listing 6.2:** Boundary conditions for a unit square heated from the top.

### 6.1.1 Speedup on more cores

Now that we have a working basic example, we can easily extend it to a finer mesh by simply editing the mesh dictionary file. For example, let us again consider Laplace's equation, but this time on a 64x64x64 mesh representing the unit cube. The boundary condition is expanded to three dimensions by heating an entire face of the cube instead of the edge of the unit square.




| Number of cores | Time (seconds) |
|---|---|
| 1 | 486 |
| 2 | 186 |
| 4 | 110 |
| 8 | 65 |
| 16 | 45 |
| 32 | 39 |

**Figure 6.2:** Computation times for Laplace's equation on a 64x64x64 mesh using different numbers of cores.

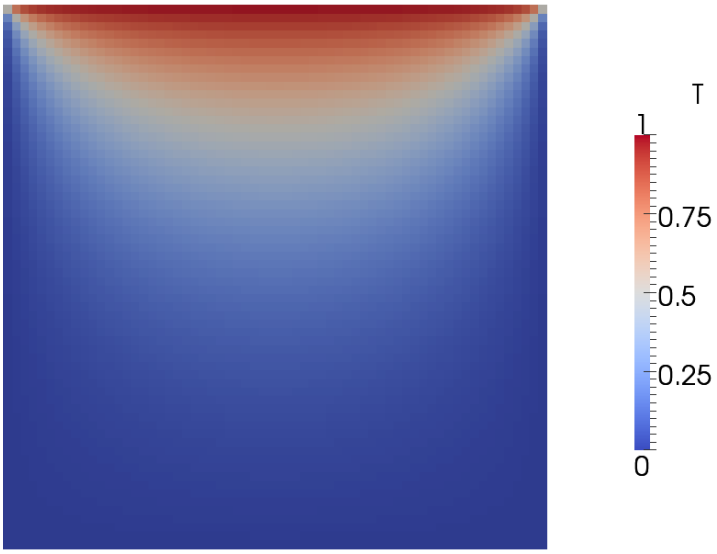



**Figure 6.3:** Two-dimensional slice through the middle of the unit cube of the numerical solution of Laplace's equation in three dimensions on a 64x64x64 mesh.

Our vector of unknowns now consists of $64^3 = 262144$ variables and solving the system of equations on a single core of standard desktop computer now takes several minutes, which gives a good opportunity to see how our application behaves when we add more cores. To do this, we make use of a computer cluster consisting of over 200 nodes, each having at least 12 AMD Opteron cores. The computation times for different numbers of cores are shown in Figure 6.2. As

we can observe from this graph, the speedup is initially even better than linear, which is probably due to the larger size of the (accumulated) caches. Increasing the number of cores to four or eight still gives acceptable speedups which are just below the (ideal) linear speedup. If we keep increasing the number of cores, the computation time seems to converge; now things like the communication, disk operations, initialization and the sequential parts of the software become a bottleneck. Also, the communication becomes relatively slower in this case, since we have to use multiple nodes to get more than 12 cores and inter-node communication is slower than intra-node communication.

Obviously, the result is the same in all cases. Figure 6.3 shows a two-dimensional part of the result by taking a 64x64x1 slice through the middle of the unit cube. As we can see from this figure, the result is quite similar but more accurate in comparison with the example of the previous section. This may have been expected, since we are solving almost the same problem, but on a finer and three-dimensional mesh.

## 6.2 Navier-Stokes equations

In the previous section we demonstrated a few simple examples, we now continue with a more complex problem by considering the incompressible Navier-Stokes equations which describe the flow of Newtonian fluids:

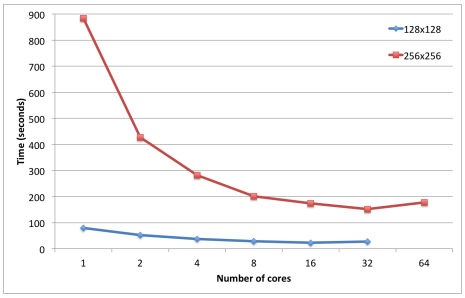
$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} &= -\nabla p + \nu \nabla^2 \mathbf{u} + f, \\ \nabla \cdot \mathbf{u} &= 0. \end{aligned}$$

The unknowns of the equations are denoted by $\mathbf{u}$, which is the vector containing velocities in all space dimensions, and $p$, which is the pressure. Furthermore, $\nu$ is a constant denoting the kinematic viscosity, while $f$ is used to to denote external forces.

For our test cases, we assume $\nu = 0.01$ and $f = 0$. The non-linear term $\mathbf{u} \cdot \nabla \mathbf{u}$ is linearized by using the flux field $\phi$ and calculating $\nabla(\phi \mathbf{u})$ instead, which is a common way to handle this term in OpenFOAM applications.
Our domain consists of the two-dimensional lid-driven cavity, which also is included in the OpenFOAM demo application `icoFoam`. The top boundary of our square cavity moves in the $x$-direction with a speed of $1m/s$, while the other boundaries are stationary. We assume the pressure to be zero on all walls. Since these equations are time dependent, we also have to choose a suitable time step and start and end times. For this example, we choose our time step $\delta t = 0.01$, the start time $t_{\text{start}} = 0$ and the end time $t_{\text{end}} = 0.1$, which means we have to do 10 solves.
Furthermore, as we now have three unknowns per mesh cell, $u$, $v$ and $p$, our system becomes much larger if we would choose the same number of cells as in the previous examples. Also, the computation times and memory usage increase, so we will limit ourselves to slightly coarser mesh of 128x128 and 256x256, using a subdomain size of 4x4 when running the solver. The dictionary files in which all these parameters are defined, are listed in Appendix B.

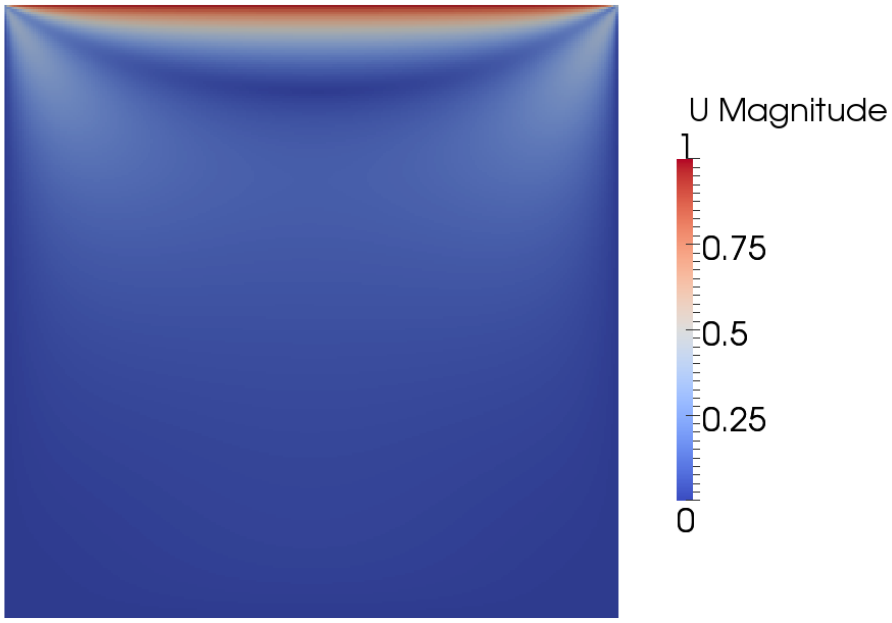
We have run these test cases on different numbers of cores, the timings for both cases are shown in Figure 6.4. In both cases, we did not include the additional time to calculate the implicit gradient matrix using the method discussed in Chapter 5. As this matrix can be read from a file and does not change in time, we only have to calculate it once per mesh.
The timings show more or less comparable behaviour as the timings from our previous Laplace examples. At first the speedup is almost or even better than ideal, but starts to decrease when we keep adding cores and ultimately we even see that the performance starts to drop when add even more cores. Another interesting observation is the difference between the different cases: for each number of cores, the difference in speed is about a factor 10, which corresponds quite well to factor 12 difference in unknowns between both cases.

The result at $t = 0.1$ for the 256x256 mesh is shown in Figure 6.5, which shows the magnitude of our velocity vector in every cell of the mesh (Figure 6.5(a)) and the stream lines as generated by the Stream Tracer filter of ParaFoam (Figure 6.5(b)). Using the OpenFOAM demo application `icoFoam`, which uses a built-in conjugate gradient based OpenFOAM solver, one can reproduce similar results.
In [5], this problem is also solved by computing the stationary solution for this low Reynolds number problem immediately. However, the computational complexity of the linear solver does currently not depend on the time step, so the results here predict the behavior for an "infinite" time step (used in the stationary case) well.

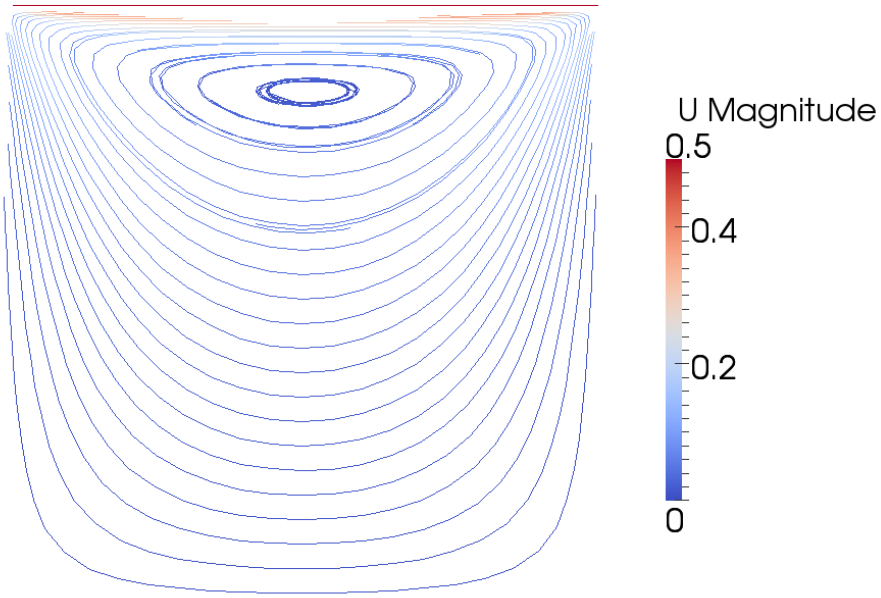



| Cores | Time (seconds) | |
|---|---|---|
| | 128x128 | 256x256 |
| 1 | 80 | 884 |
| 2 | 52 | 427 |
| 4 | 37 | 282 |
| 8 | 29 | 201 |
| 16 | 23 | 174 |
| 32 | 27 | 152 |
| 64 | - | 178 |

**Figure 6.4:** Computation times for Laplace's equation on a 64x64x64 mesh using different numbers of cores.




(a) Magnitude of the velocity vector.




(b) Streamlines generated by ParaFoam's Stream Tracer filter.

**Figure 6.5:** Result at $t = 0.1$ for our Navier-Stokes example on a 256x256 mesh.

# Chapter 7

# Conclusion

In this thesis we presented an application which is able to find a numerical solution for equation(s) describing the flow on a domain by splitting the entire process into two different parts. As we discussed in the introduction, this entire process can roughly be divided into a discretization part and a solve part. Our application basically glues together two different software packages, which both handle one of these parts.

First, the description of both the flow and the domain itself should be discretized in order to be handled by a computer. This entire step is done using the OpenFOAM software package. As we discussed in Chapter 4, we chose to use OpenFOAM because of the very easy and straightforward manner to enter partial differential equations without having to manually implement, for instance, differential schemes for each different equation. Furthermore, OpenFOAM includes a tool to generate meshes, by defining the properties of the mesh in a text file, like the boundaries, number of cells in each direction, etcetera. Also, boundary conditions for each variable and all kinds of parameters can be entered in a special text files.

When this step has finished, we are left with a lot of linear equations which describe how our numerical approximation at the next time step depends on the previous one. Though OpenFOAM also does include a few different solvers which are able to perform this step, we want to make use of a custom solver which has favorable properties over the more common solvers of OpenFOAM. Also, we want these equations to be contained into a single block matrix, such that we can solve all equations at the same time. This is, at least currently, not (yet) supported by OpenFOAM.

So, the second main part, i.e. to solve all equations obtained by the discretization process, is handled by our two-level version of the HYMLS solver. As we have shown in Chapter 2, this solver combines the advantages of iterative and direct solvers and is written using the software package Trilinos.

Now, in order to glue both parts together, our application needs to do some important tasks as we discussed in Chapter 5. One of them is the conversion of OpenFOAM structures and implementations to a suitable input for HYMLS.

Unfortunately, OpenFOAM is not really prepared for these kind of situations: though the documentation is in general quite good, it does lack details about the internally used classes to store things like matrices, equations, solutions and mesh information. So, getting the right information out of OpenFOAM and into a different ordered Trilinos-based matrix-vector equation is quite tedious. Furthermore, our application needs to add methods to allow implicit discretizations of gradient and divergence operators. As these are usually only necessary in equations involving block matrices, OpenFOAM lacks an implementation for these operators and only includes explicit ones. In Chapter 5, we presented a straightforward method to use the explicit methods to construct a matrix corresponding to an implicit discretization.

Finally, in Chapter 6 we have shown some results of our application for both Laplace's equations and Navier-Stokes equations. Besides the actual solutions, which are more or less the same as the outputs of similar OpenFOAM demo applications, we have also shown the timings on different numbers of cores. We concluded that the scaling of the application is quite good, up to a certain number of cores depending on the size of the problem.

## 7.1 Future work

As we have demonstrated and discussed, the application currently works well. However, there are some aspects which can be improved or extra functionalities could be added.

First of all, the method discussed in Chapter 5 which allows implicit discretizations for the gradient and divergence operators is currently not very fast, especially when the size of the mesh increases. In order to speed it up, one could use something like a coloring method to reduce the number of explicit gradient calculations which are necessary to build the matrix. Also, these explicit gradient calculations do not depend on each other, hence they could be parallelized to get even more speed gain.

A second possible extension is to include support for future releases of OpenFOAM. As OpenFOAM is still under active development, new functionalities are added regularly. For example, a possible interesting and important functionality which is already said to be included in a future release by the main developer of OpenFOAM [3], is the support for block matrices.

Finally, support for future versions of HYMLS may be added, since HYMLS is also still being developed and extended. As an example, recently a new multilevel version was already released. Just like future releases, it should be not too complex to update the version of the solver and make use of new features, unless the solver changes drastically.

# Bibliography


[1] *OpenFOAM Programmer's Guide*, July 2009. Version 1.6.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[3] Hrvoje Jasak. Openfoam: Future developments. http://powerlab.fsb.hr/ped/kturbo/OpenFOAM/WorkshopZagrebJan2006/FutureDevelopments.pdf.

[4] S. V. Patankar and D. B. Spalding. A Calculation Procedure for Heat, Mass and Momentum Transfer in Three-Dimensional Parabolic Flows. *Int. J. Heat Mass Transfer*, 15:1787–1972, 1972.

[5] Jonas Thies. *Scalable algorithms for fully implicit ocean models*. PhD thesis, University of Groningen, 2011.

[6] Jonas Thies and Fred Wubs. Design of a parallel hybrid direct/iterative solver for cfd problems. In Bob Werner, editor, *Proceedings 2011 Seventh IEEE International Conference on eScience, 5-8 December 2011, Stockholm, Sweden*, pages 387–394. Conference Publications Services, 2011. ISBN 978-0-7695-4597-4.

[7] Fred W. Wubs and Jonas Thies. A robust two-level incomplete factorization for (navier–)stokes saddle point matrices. *SIAM. J. Matrix Anal. and Appl.*, 32:1475–1499, 2011.

# Appendix A

# Files of OpenFOAM's laplacianFoam example

## A.1 laplacianFoam.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 1991-2010 OpenCFD Ltd.
     \\/     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 3 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, see <http://www.gnu.org/licenses/>.

Application
    laplacianFoam

Description
    Solves a simple Laplace equation, e.g. for thermal diffusion in a solid.

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

```
#include "fvCFD.H"


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

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

#   include "setRootCase.H"

#   include "createTime.H"
#   include "createMesh.H"
#   include "createFields.H"

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

    Info<< "\nCalculating temperature distribution\n" << endl;

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

#       include "readSIMPLEControls.H"

        for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
        {
            solve
            (
                fvm::ddt(T) - fvm::laplacian(DT, T)
            );
        }

#       include "write.H"

        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }

    Info<< "End\n" << endl;

    return 0;
}


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

## A.2 createFields.H

```
Info<< "Reading field T\n" << endl;

volScalarField T
(
    IOobject
    (
        "T",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);


Info<< "Reading transportProperties\n" << endl;

IOdictionary transportProperties
(
    IOobject
    (
        "transportProperties",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ,
        IOobject::NO_WRITE
    )
);


Info<< "Reading diffusivity DT\n" << endl;

dimensionedScalar DT
(
    transportProperties.lookup("DT")
);
```

## A.3 write.H

```
if (runTime.outputTime())
{

    volVectorField gradT = fvc::grad(T);


    volScalarField gradTx
```

```
        (
            IOobject
            (
                "gradTx",
                runTime.timeName(),
                mesh,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            gradT.component(vector::X)
        );


        volScalarField gradTy
        (
            IOobject
            (
                "gradTy",
                runTime.timeName(),
                mesh,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            gradT.component(vector::Y)
        );


        volScalarField gradTz
        (
            IOobject
            (
                "gradTz",
                runTime.timeName(),
                mesh,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            gradT.component(vector::Z)
        );


        runTime.write();
    }
```

## A.4 controlDict

```
/*--------------------------*- C++ -*----------------------------*\
| =========                 |                                     |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox|
```

```
|  \\    /   O peration    | Version:  1.7.1                     |
|   \\  /    A nd          | Web:      www.OpenFOAM.com          |
|    \\/     M anipulation |                                     |
\*---------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application     laplacianFoam;

startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         3;

deltaT          0.005;

writeControl    runTime;

writeInterval   0.1;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression uncompressed;

timeFormat      general;

timePrecision   6;

runTimeModifiable yes;


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

## A.5 fvSolution

```
/*--------------------------*- C++ -*----------------------------*\
| =========                 |                                     |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox|
|  \\    /   O peration     | Version:  1.7.1                     |
|   \\  /    A nd           | Web:      www.OpenFOAM.com          |
|    \\/     M anipulation  |                                     |
\*-------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
    T
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0;
    }
}

SIMPLE
{
    nNonOrthogonalCorrectors 2;
}


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

## A.6 transportProperties

```
/*--------------------------*- C++ -*----------------------------*\
| =========                 |                                     |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox|
|  \\    /   O peration     | Version:  1.7.1                     |
|   \\  /    A nd           | Web:      www.OpenFOAM.com          |
|    \\/     M anipulation  |                                     |
\*-------------------------------------------------------------*/
FoamFile
{
```

```
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

DT              DT [ 0 2 -1 0 0 0 0 ] 4e-05;


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

# Appendix B

# Dictionaries of Navier-Stokes example

## B.1 0/p

```
/*--------------------------*- C++ -*----------------------------*\
| =========                 |                                     |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox|
|  \\    /   O peration     | Version:  1.7.1                     |
|   \\  /    A nd           | Web:      www.OpenFOAM.com          |
|    \\/     M anipulation  |                                     |
\*-------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    movingWall
    {
        type            fixedValue;
        value           uniform 0;
    }

    fixedWalls
    {
```

```
        type            fixedValue;
        value           uniform 0;
    }

    frontAndBack
    {
        type            empty;
    }
}

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

## B.2 0/U

```
/*--------------------------*- C++ -*----------------------------*\
| =========                 |                                    |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox|
|  \\    /   O peration     | Version:  1.7.1                    |
|   \\  /    A nd           | Web:      www.OpenFOAM.com         |
|    \\/     M anipulation  |                                    |
\*-------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{
    movingWall
    {
        type            fixedValue;
        value           uniform (1 0 0);
    }

    fixedWalls
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }

    frontAndBack
```

```
    {
        type            empty;
    }
}

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

## B.3 constant/polyMesh/blockMeshDict

```
/*--------------------------*- C++ -*----------------------------*\
| =========                 |                                     |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox|
|  \\    /   O peration     | Version:  1.7.1                     |
|   \\  /    A nd           | Web:      www.OpenFOAM.com          |
|    \\/     M anipulation  |                                     |
\*-------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

convertToMeters 0.1;

vertices
(
    (0 0 0)
    (1 0 0)
    (1 1 0)
    (0 1 0)
    (0 0 0.1)
    (1 0 0.1)
    (1 1 0.1)
    (0 1 0.1)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (256 256 1) simpleGrading (1 1 1)
);

edges
(
);

boundary
```

```
(

    movingWall
    {
        type wall;
        faces
        (
            (3 7 6 2)
        );
    }
    fixedWalls
    {
        type wall;
        faces
        (
            (0 4 7 3)
            (2 6 5 1)
            (1 5 4 0)
        );
    }
    frontAndBack
    {
        type empty;
        faces
        (
            (0 3 2 1)
            (4 5 6 7)
        );
    }
);

mergePatchPairs
(
);

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

## B.4 constant/transportProperties

```
/*--------------------------*- C++ -*----------------------------*\
| =========                 |                                     |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox|
|  \\    /   O peration     | Version:  1.7.1                     |
|   \\  /    A nd           | Web:      www.OpenFOAM.com          |
|    \\/     M anipulation  |                                     |
\*-------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
```

```
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

nu              nu [ 0 2 -1 0 0 0 0 ] 0.01;

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

## B.5 system/controlDict

```
/*--------------------------*- C++ -*----------------------------*\
| =========                 |                                     |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox|
|  \\    /   O peration     | Version:  1.7.1                     |
|   \\  /    A nd           | Web:      www.OpenFOAM.com          |
|    \\/     M anipulation  |                                     |
\*-------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application     navierStokesFoam;

startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         0.1;

deltaT          0.01;

writeControl    runTime;

writeInterval   0.01;

purgeWrite      0;

writeFormat     ascii;
```

```
writePrecision  6;

writeCompression uncompressed;

timeFormat      general;

timePrecision   6;

runTimeModifiable yes;


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

## B.6 system/fvSchemes

```
/*--------------------------*- C++ -*----------------------------*\
| =========                 |                                     |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox|
|  \\    /   O peration     | Version:  1.7.1                     |
|   \\  /    A nd           | Web:      www.OpenFOAM.com          |
|    \\/     M anipulation  |                                     |
\*-------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

ddtSchemes
{
    default         Euler;
}

gradSchemes
{
    default         Gauss upwind phi;
}

divSchemes
{
    default         none;
    div(phi,U)      Gauss linear corrected;
}
```

```
laplacianSchemes
{
    default         none;
    laplacian(nu,U) Gauss linear corrected;
}

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

## B.7 system/fvSolution[1]

```
/*--------------------------*- C++ -*----------------------------*\
| =========                 |                                     |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox|
|  \\    /   O peration     | Version:  1.7.1                     |
|   \\  /    A nd           | Web:      www.OpenFOAM.com          |
|    \\/     M anipulation  |                                     |
\*------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
    p
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0;
    }

    U
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-05;
        relTol          0;
    }
}

PISO
```

[1]Note: though the contents of this dictionary is not really important (as we are using our own solver), this file should exist in order to run any OpenFOAM application.

```
{
    nCorrectors     2;
    nNonOrthogonalCorrectors 0;
    pRefCell        0;
    pRefValue       0;
}


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