Aalto University
School of Engineering
Mechanical Engineering

PISO vs. transient SIMPLE: A Comparison of Two Different Transient Solution Algorithms in Incompressible CFD with OpenFOAM and Python

Bachelor Thesis

18.04.2011

Tuomas Turunen

Aalto-yliopisto
Teknillinen korkeakoulu

Aalto University
School of Engineering
PO Box 14100, FI-00076 Aalto
http://www.aalto.fi

BACHELOR THESIS ABSTRACT

Author: Tuomas Turunen

Title: PISO vs. transient SIMPLE: A Comparison of Two Different Transient Solution Algorithms in Incompressible CFD with OpenFOAM and Python

Studyprogramme: Mechanical Engineering

Main subject: Aeronautical Engineering | Subject code: K3004

Supervisor: prof. Gary Marquis

Advisor: DI Mikko Auvinen

The goal of this work is to compare two transient solvers in incompressible viscous flow, one based on the SIMPLE- and the other on the PISO-algorithm. A case-study is performed with the help of Python scripting. At the end general guidelines for determining which solver is more adequate in a given flow case are presented.

Date: 18.04.2011 | Language: english | Pages: 21 + 6

Key words: CFD, PISO, SIMPLE, OpenFOAM

Contents

Symbols and Abbreviations

$\mathbf{A}^u$	matrix in the momentum equation
$\mathbf{A}^p$	matrix in the pressure equation
a_N	off-diagonal term of matrix $\mathbf{A}^u$
a_P	diagonal term of matrix $\mathbf{A}^u$
$\mathbf{b}$	vector in the pressure equation
$\mathbf{H}$	$-\sum_{N_f} a_N \mathbf{U}_N + \mathbf{src}(\mathbf{U})$
$\mathbf{n}_f$	normal vector of a cell face
N_f	number of faces in a cell
p	pressure
p_f	pressure on a cell face
u	velocity x-component
$\mathbf{U}$	velocity vector
$\mathbf{U}_f$	velocity vector on a cell face
$\mathbf{U}_{f,HS}$	velocity interpolated using some other than the upwind scheme
$\mathbf{U}_{f,UW}$	velocity using the upwind scheme
$\mathbf{U}_{f,\Delta}$	$\mathbf{U}_{f,HS} - \mathbf{U}_{f,UW}$
$\mathbf{U}_N$	velocity vector in a neighboring cells center
$\mathbf{U}_P$	velocity vector in a cell center
$\mathbf{U}^\star$	$\frac{\mathbf{H}}{a_P}$
S_f	surface area of a cell face
$\mathbf{src}$	vsource vector in the momentum equation
Δt	increment of time; time step
u	velocity x-component
v	velocity y-component
w	velocity z-component
V_P	volume of a cell around point P
x	x-coordinate
y	y-coordinate
z	z-coordinate

α_U	relaxation factor of the momentum equation
α_p	relaxation factor of pressure
ϕ	volume flux
ϕ_f	volume flux on a cell face
ϕ_{corr}	correction term of $\phi^\star$
$\phi^\star$	$\left(\mathbf{U}_f^\star \cdot \mathbf{n}_f\right) S_f + \phi_{corr}$
ν	kinematic viscosity

CFD	Computational Fluid Dynamics

1 Introduction

1.1 Motivation

Computational Fluid Dynamics (CFD) is an important tool in a variety of fields such as aeronautical engineering and turbomachinery as well as ship- and car design. The development of computer technology and numerical methods have made it possible to run simulations on regular PC's and CFD is increasingly employed as a design tool in engineering. To increase the feasibility of preliminary design with CFD, effective methods of running transient viscous flow computations are needed. Knowledge of the characteristics of existing solution methods gives a better starting point for selecting the most effective way of running a computation and thus save time and costs.

An open source CFD code opens the way for customizing and automating computations in a versatile manner. Being able to automatically vary parameters makes comparing cases with slight differences fast and effective. For example, the shape of an airplane, or of a part of it, could be optimized by running computations with slightly different shapes and comparing the results.

Indeed, at the present state of CFD all from grid generation to post-processing can be defined in advance so that no computer-human interaction is needed after the computation has been started. This is the case in this work, too. However, there is still much to be developed in the field since the automation of computations still imposes limitations. The subject is currently under research [1].

1.2 Objective

The goal of this work is to compare two different pressure based solution algorithms for transient flow with viscous effects included, one with PISO- [2] and one with SIMPLE-pressure correction algorithm [3]. This is done using an open source CFD library, OpenFOAM [4]. The solvers used are *transientSimpleFoam* and *icoFoam*. One case is computed with the PISO-based *icoFoam* and eight cases with the SIMPLE-based *transientSimpleFoam*. The PISO-based results are used as reference and the SIMPLE-based results are compared against them.

At the end of this work there will be a discussion of flow features that affect the applicability of the two solution methods. Some general guidelines on how to choose the more adequate solution method will be outlined.

1.3 Outline

The case studied is called the Taylor-Green vortex. It is a two-dimensional decaying vortex in a square domain. The case is transient and there is an analytical solution for it. It has been used for testing accuracy of numerical methods. Figure (1) shows the velocity magnitude and pressure fields according to the analytical result used in this work as a reference.

The solvers are run with help of Python-scripting. Reference [1] contains a script that handles the whole flow case. It runs the *icoFoam* solver in OpenFOAM and changes the mesh-density and divergence schemes. In addition to running the case, the script also

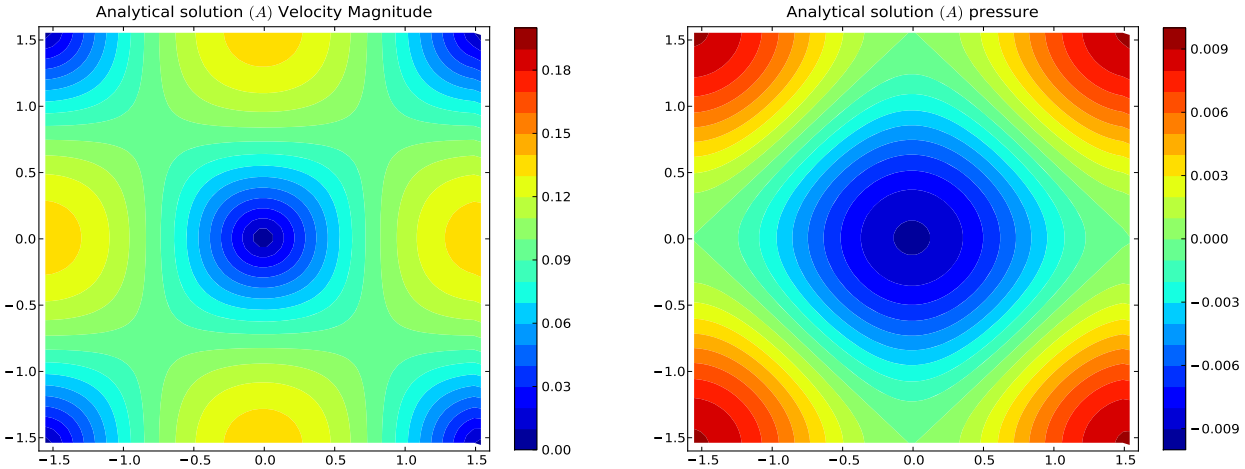

Figure 1 Velocity and Pressure Fields According to the Analytical Solution.

handles the pre- and post-processing of the case.

2 Flow Solvers

In this chapter, the governing equations of viscous transient flow will be presented. The equations form a set of unlinear partial differential equations.

To enable solving the set of equations with a computer, a discrete form of the equations needs to be presented. The discretizing treatments for different terms in the equations will be presented separately. The treatments will include the linearization of the convection term as well as the formulation of the transient and friction terms. Also a way to utilize the mass conservation law along with the momentum conservation law, the pressure equation, will be derived.

At the end of this chapter, the two computer algorithms that are used for solving the equations in this work, will be presented in detail. Their characteristics will be discussed and compared against one another.

2.1 Theoretical Background

A time-dependent incompressible laminar flow is governed by the Navier-Stokes equations and mass conservation, i.e. continuity, equation [6].

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

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

There are two main difficulties in these equations:

1) The convection term $\nabla \cdot (\mathbf{U}\mathbf{U})$ is unlinear because velocity is multiplied by itself and

2) the equations are coupled in U and p.

The first problem, unlinearity, will be handled by introducing a new flux variable [5]

$$\phi_f = \mathbf{U}_f \cdot \mathbf{n} S_f \ , \tag{3}$$

where $\mathbf{U}_f$ is velocity interpolated onto a cell face, n is the unit normal vector of the cell face and S_f the area of the cell surface The point is that the flux ϕ will be calculated using known velocities and thus the convection term will only contain a known scalar variable and one unknown term, the velocity being solved [6]. This treatment linearizes the convection term. For the linearized form, see section (2.2.3)

The fact that the transporting part of the convection term, ϕ, is calculated using known velocities means that information is lagged. If the error caused by the lagged information can not be neglected, an iterative solution method is needed. In that case the *known* values needed to calculate ϕ can be taken from the previous iteration round. Until the

velocity field no more changes from one iteration round to another, the convection term does not correspond to the true convection term, but after convergence the lagging is no more significant [6].

Because of the pressure-velocity coupling, the momentum equation can not be solved without an existing pressure field. Since the pressure field is, in general, not known, an initial pressure field needs to be estimated. However, if the estimate is not the true pressure field, the velocity solution from the momentum equation does not fulfill the continuity requirement [3].

This problem is dealt with by deriving a pressure equation. The momentum equation is manipulated so that new velocities are expressed in terms of known velocity and pressure values. By substituting the expression into the continuity equation, a new relation for velocity and pressure is obtained. This relation is the pressure equation and it is essentially equal to the mass conservation law. It is used to progressively correct the velocity field, so that ultimately also the continuity requirement is fulfilled. However, also the utilization of the pressure equation will lead to an iterative solution method [3].

Both of the mentioned problems, unlinearity and pressure-velocity coupling, result in the fact that pressure and velocity can not be solved at one time without great difficulty, and pose a need to iterate while solving the variables. In the following part of this chapter, two different iterative algorithms will be derived.

2.2 Discretizing the equations

Equations (1) and (2) are discretized in order to formulate a set of equations for a spatial domain represented by a grid. Both equations are written for each cell in the grid, which will result in a large set of equations. Due to the linearization practise described in Sec. (2.1), the equations can be represented as a system of linear equations, a matrix equation.

In a general case a cell can have an arbitrary amount of neighbours and an arbitrary convex shape. Also the neighboring cells are present in the discretized form of each cell's equation. According to [7], the discrete form of equations (1) and (2) is written

$$\sum_f^{N_f} (\mathbf{U}_f \cdot \mathbf{n}_f) S_f = 0 \tag{4}$$

$$\frac{\Delta \mathbf{U}}{\Delta t} V_P + \sum_f^{N_f} \mathbf{U}_f (\mathbf{U} \cdot \mathbf{n})_f S_f - \sum_f^{N_f} (\nu \nabla \mathbf{U} \cdot \mathbf{n})_f S_f = -\sum_f^{N_f} p_f \mathbf{n}_f S_f \,, \tag{5}$$

where N_f is the number of cell faces. Applying the linearization (3) to the convection term, the momentum equation (5) becomes

$$\frac{\Delta \mathbf{U}}{\Delta t} V_P + \sum_f^{N_f} \mathbf{U}_f \phi_f - \sum_f^{N_f} (\nu \nabla \mathbf{U} \cdot \mathbf{n})_f S_f = -\sum_f^{N_f} p_f \mathbf{n}_f S_f \tag{6}$$

2.2.1 Momentum equation

The exact form of the matrix equation that results from the momentum equation depends at least on the discretization and interpolation methods used by the solver. However, it can always be written in the following general form:

$$\mathbf{A}^u \mathbf{U} = \mathbf{src}(\mathbf{U}) - \nabla p \ , \tag{7}$$

where $\mathbf{A}^u$ is the coefficient matrix of the momentum equation and $\mathbf{src}(\mathbf{U})$ is a *source* vector. ∇p is left out of the source vector to enable the manipulation presented in section (2.2.5).

From this point on the solvers differ from each other. In *transientSimpleFoam* with the SIMPLE pressure loop the diagonal terms of the matrix **A** are underrelaxed while in *icoFoam* with the PISO pressure loop no underrelaxation is applied. Thus, the momentum equation is solved in *icoFoam* in a form represented by equation (8).

The terms in matrix **A** need to be rearranged to enable further manipulation. The matrix is split into its diagonal and non-diagonal terms, a_P and a_N respectively. The following equation is obtained [6] [7]:

$$a_P \mathbf{U}_P + \sum_{N_f} a_N \mathbf{U}_N = \mathbf{src}(\mathbf{U}) - \nabla p \tag{8}$$

The underrelaxation of the momentum equation by a coefficient α_U is applied in the following way:

First, the diagonal terms a_P are divided in two parts:

$$a_P = \frac{1}{\alpha_U} a_P - \frac{(1-\alpha_U)}{\alpha_U} a_P \tag{9}$$

Then the left part - that is now larger than a_P since $0 \leq \alpha_U \leq 1$ - is included in matrix **A**. The right part is moved onto the right hand side of the equation and multiplied by a known velocity. Thus the momentum equation solved in *transientSimpleFoam* gets its final form [7].

$$\frac{a_P}{\alpha_U} \mathbf{U}_P + \sum_{N} a_N \mathbf{U}_N = \mathbf{src}(\mathbf{U}) - \nabla p + \frac{1-\alpha_U}{\alpha_U} a_P \mathbf{U}_P \tag{10}$$

The transient-, convection- and friction terms are divided between the matrix **A** and vector $\mathbf{src}(\mathbf{U})$. $\mathbf{src}(\mathbf{U})$ on the right-hand-side of the equation will contain known velocities and **A** the coefficients of the unknown velocities. The way in which the terms are divided is presented separately in chapters 2.2.2, 2.2.3 and 2.2.4

2.2.2 Transient term

The transient term can be first or second order accurate. In either way, it consists of the velocities that are being solved and older, known, velocities. For example the first order accurate implicite Euler method can be described in the following way:

$$\frac{\Delta \mathbf{U}}{\Delta t} V_P = \frac{\mathbf{U}^t - \mathbf{U}^{t-1}}{\Delta t}$$

In this case $\frac{\mathbf{U}^t}{\Delta t}$ would be included in the matrix as $\frac{1}{\Delta t}$ on the diagonal and $\frac{\mathbf{U}^{t-1}}{\Delta t}$ would be moved into the $\mathbf{src}(\mathbf{U})$ vector. In general, all terms involving known velocities will be included in $\mathbf{src}(\mathbf{U})$ and all terms that multiply the unknown velocity put into $\mathbf{A}^u$ [6].

2.2.3 Convection term

Introducing the definition of the flux, (3), the convection term can be expressed as:

$$\sum_f^{N_f} \mathbf{U}_f (\mathbf{U} \cdot \mathbf{n})_f \, S_f = \sum_f^{N_f} \phi_f \mathbf{U}_f \tag{11}$$

In OpenFOAM, ϕ is always defined on each cell face but $\mathbf{U}$ must be interpolated onto the cell faces for the summation in equation (11).

Each row in $\mathbf{A}$ represents one cell in the grid. The off-diagonal terms represent the neighboring cells' contribution to the cell's new velocity. The convection based terms in $\mathbf{A}$ are found by using the upwind interpolation scheme for $\mathbf{U}$. This means that $\mathbf{U}$-values are adopted from the cell that is located on the upwind side of the surface.
If, however, some other interpolation scheme is used for $\mathbf{U}$ the terms that differ from the upwind-scheme are included in $\mathbf{src}(\mathbf{U})$. Generally, the discretization practise of interpolating the velocity values onto the cell faces can be expressed as

$$\mathbf{U}_{f,HS} = \mathbf{U}_{f,UW} - (\mathbf{U}_{f,HS} - \mathbf{U}_{f,UW}) = \mathbf{U}_{f,UW} - \mathbf{U}_{f,\Delta} \, , \tag{12}$$

where the indices represent the following:

HS : Higher order scheme

UW : Upwind Scheme

Δ : Difference between the two schemes

In this general case the upwind based term multiplying $\mathbf{U}_{f,UW}$ is included in $\mathbf{A}$ and $\mathbf{U}_{f,\Delta}$ in $\mathbf{src}(\mathbf{U})$ along with the term multiplying it. The upwind scheme creates an easily solvable matrix and is therefore used as the basic scheme on which other schemes are built [6].

2.2.4 Friction term

In this case-study all friction based terms are included in $\mathbf{A}$ because the grid is completely orthogonal. However, if there are non-orthogonal features in the grid the terms caused by the non-orthogonality are included in $\mathbf{src}(\mathbf{U})$.

2.2.5 Pressure equation

In order to formulate the pressure equation, a new expression for velocity is required, from which the velocity can be solved when the pressure field is given. For this purpose the momentum equations (8) and (10) are manipulated.

First, the off-diagonal terms of the matrix $\mathbf{A}^u$, a_N, in equations (8) and (10) are moved onto the right-hand-side of the equations.

$$a_P\mathbf{U}_P = -\sum_{N_f} a_N\mathbf{U}_N + \mathbf{src}(\mathbf{U}) - \nabla p \tag{13}$$

Secondly, a new vector, $\mathbf{H}$, is introduced. Equations (8) and (10) vary due to the underrelaxation in *transientSimpleFoam*. Thus, $\mathbf{H}$ needs to be written separately for both solvers. The definition of $\mathbf{H}$ for *transientSimpleFoam* and *icoFoam* is presented in equations (14) and (15), respectively.

$$\text{SIMPLE: } \mathbf{H}(\mathbf{U},\phi) = -\sum_{N_f} a_N\mathbf{U}_N + \mathbf{src}(\mathbf{U}) + \frac{1-\alpha_U}{\alpha_U}a_P\mathbf{U}_P \tag{14}$$

$$\text{PISO: } \mathbf{H}(\mathbf{U},\phi) = -\sum_{N_f} a_N\mathbf{U}_N + \mathbf{src}(\mathbf{U}). \tag{15}$$

Substituting $\mathbf{H}$ in equations (14) and (15), into the corresponding momentum equations, (8) and (10), both of the equations can be rewritten in a general form, where the vector $\mathbf{H}$ varies according to the solver:

$$a_P\mathbf{U}_P = \mathbf{H}(\mathbf{U},\phi) - \nabla p \tag{16}$$

Finally, equation (16) can be solved for $\mathbf{U}$. In order to formulate the pressure equation, velocities on the cell surfaces are needed. The expressions for the velocity in a cell center and on a cell surface are presented in equations (17) and (18), respectively.

$$\mathbf{U}_P = \frac{\mathbf{H}(\mathbf{U},\phi)}{a_P} - \frac{\nabla p}{a_P} \tag{17}$$

$$\mathbf{U}_f = \left(\frac{\mathbf{H}(\mathbf{U},\phi)}{a_P}\right)_f - \left(\frac{1}{a_P}\right)_f (\nabla p)_f \tag{18}$$

The function of the pressure equation was to enable utilizing the continuity equation (1) with the momentum equation (2). Until now, only the momentum equation has been involved but at this point also the continuity equation is introduced. By substituting the new velocity on the cell surfaces (18) into equation (1) a new relation for velocity and pressure, the pressure equation, is obtained:

$$\sum_{N_f}\left[\left(\frac{1}{a_P}\right)_f(\nabla p)_f\right]\cdot\mathbf{n}_f S_f=\sum_{N_f}\left(\frac{\mathbf{H}(\mathbf{U},\phi)}{a_P}\right)_f\cdot\mathbf{n}_f S_f \tag{19}$$

In OpenFOAM, this equation is treated by introducing two variables:

$$\mathbf{U}^\star=\left(\frac{\mathbf{H}(\mathbf{U},\phi)}{a_P}\right) \tag{20}$$

$$\phi^\star=\left(\mathbf{U}_f^\star\cdot\mathbf{n}_f\right)S_f+\phi_{corr} \tag{21}$$

In the PISO-algorithm, the velocity used in the first pressure correction contains divergence because the continuity equation has at that point not yet been used. For this reason error terms will appear in the first pressure equation and can impair the solution in it's intermediate stage [2]. The ϕ_{corr} term in equation (20) accounts for the velocity divergence.

The flux ϕ can be written in two ways. It can be the inner product of the velocity interpolated onto the cell surface with the cell surface vector, $\mathbf{U}_f\cdot\mathbf{n}_f S_f$, or it can be an older corrected flux field. ϕ_{corr} corrects the interpolated velocity based flux by comparing an older flux, ϕ, and an older velocity interpolated onto the cell surface.

Substituting equations (20) and (21) into equation (19), the pressure equation for each cell can be written:

$$\sum_{f}^{N_f}\left[\left(\frac{1}{a_P}\right)_f(\nabla p)_f\right]\cdot\mathbf{n}_f S_f=\sum_{f}^{N_f}\phi_f^\star \tag{22}$$

By forming a similar equation for each cell, a matrix equation where the new pressure field can be solved is obtained. The final form of the pressure equation is presented in equation (23).

$$\mathbf{A}^p p=b\ , \tag{23}$$

where $\mathbf{A}^p$ is the coefficient matrix of the pressure equation, p contains the pressure values at each cell center and b the right-hand-side terms concerning each pressure value. Each row in equation (23) consists of equation (22) written for one cell in the grid. After solving for new p the velocity expression (18) can be substituted into the definition of ϕ (3). Thus new fluxes that exactly fulfill the mass conservation law (22) can be calculated. In other words, the flux ϕ, is corrected with the new pressure.

$$\phi_f=\phi_f^\star-\left(\left(\frac{1}{a_P}\right)(\nabla p)\right)_f\cdot\mathbf{n}_f S_f \tag{24}$$

At this point the solvers differ again. In *transientSimpleFoam* the new pressure is underrelaxed by a factor of α_p in the following way:

$$p^i = p^{i-1} + \alpha_p(p^i - p^{i-1}) \tag{25}$$

Only a part of the new pressure is used and the rest is taken from the previous iteration round. This increases computation stability. However, the underrelaxed pressure values do not fulfill the momentum equation before convergence. Therefore iterations are needed to reach convergence within a time step. When convergence is reached, the underrelaxation does no more change the pressure values.

With the new pressure values, also the velocity field can be corrected explicitly using equations (17) and (20).

$$\mathbf{U}_P = \mathbf{U}^\star - \frac{\nabla p}{a_P}$$

After this point either the pressure correction procedure is repeated or the computation will proceed to a new time step. The corrected velocities will be used in the right-hand-side of the equations in further computations.

Solver description; *icoFoam*:

i: Time step index

j: PISO loop index

TIME Step:

1. Increment time: $t^n = t^{n-1} + \Delta t$.
2. Build the matrix equation (8) and solve for U:
3. **PISO Loop:**

   (a) Rearrange the momentum equation to the form presented in equation (18) :

   $$\mathbf{U}_f^{i,j} = \left(\frac{\mathbf{H}(\mathbf{U}^{i,j-1}, \phi^{i,0})}{a_P}\right)_f - \left(\frac{1}{a_P}\right)_f (\nabla p^{i,j-1})_f$$

   (b) Introduce variables $\mathbf{U}^\star$ and $\phi^\star$ according to equations (20) and (21):

   $$\mathbf{U}^{\star,i,j} = \left(\frac{\mathbf{H}(\mathbf{U}^{i,j-1}, \phi^{i,0})}{a_P}\right)$$
   $$\phi_f^{\star i,j} = \left(\mathbf{U}_f^{\star i,j-1} \cdot \mathbf{n}_f\right) S_f + \phi_{corr}$$

   (c) Formulate the matrix equation for p using equation (23):

   $$\sum_f^{N_f} \left[\left(\frac{1}{a_P}\right)_f (\nabla p^{i,j})_f\right] \cdot \mathbf{n}_f S_f = \sum_f^{N_f} \phi_f^{\star i,j}$$

   (d) Correct the flux ϕ using equation (24):

   $$\phi_f^{i,j} = \phi_f^{\star i,j} - \left(\left(\frac{1}{a_P}\right)(\nabla p^{i,j})\right)_f \cdot \mathbf{n}_f S_f$$

   (e) Correct the velocity field explicitely using equation (17) written with the definition of $U^\star$ (20):

   $$\mathbf{U}_P^{i,j} = \mathbf{U}_P^{\star,i,j} - \frac{1}{a_P} \nabla p^{i,j}$$

   (f) Start the PISO-loop again (stage 3a) if predefined tolerances are not yet met or go to (1)

Solver description; *transientSimpleFoam*:

i: Time step index

j: SIMPLE loop index

TIME Step:

1. Increment time: $t^n = t^{n-1} + \Delta t$.
2. **SIMPLE Loop:**

   (a) Build the matrix equation (10) and solve for $\mathbf{U}^i$

   (b) Introduce variables $\mathbf{U}^\star$ and $\phi^\star$ according to equations 20 and 21:

   $$\mathbf{U}^{\star,i,j} = \left(\frac{\mathbf{H}(\mathbf{U}^{i,j-1}, \phi^{i,0})}{a_P/\alpha_U}\right)$$
   $$\phi_f^{\star i,j} = \left(\mathbf{U}_f^{\star i,j} \cdot \mathbf{n}_f\right) S_f$$

   (c) Build the pressure equation (22) and solve new pressure p^i:
   $$\sum_f^{N_f} \left[\left(\frac{1}{a_P}\right)_f (\nabla p^{i,j})_f\right] \cdot \mathbf{n}_f S_f = \sum_f^{N_f} \phi_f^{\star i,j}$$

   (d) Correct the flux field that fulfills mass conservation according to equation (24):
   $$\phi_f^{i,j} = \phi_f^{\star i,j} - \left(\left(\frac{1}{a_P}\right)(\nabla p^i)\right)_f \cdot \mathbf{n}_f S_f$$

   (e) Underrelax the new pressure value according to equation (25):

   $$p^i = p^{i-1} + \alpha_p(p^i - p^{i-1})$$

   (f) Correct the velocity field according to equation (17):
   $$\mathbf{U}_P = \mathbf{U}_P^{\star,i,j} - \frac{\nabla p}{a_P/\alpha_U}$$

   (g) Return to (2a) or continue to (1)

2.3 Comparing the algorithms

In this chapter, the two solution algorithms are compared against each other. Their characteristics are discussed mostly in the computational effort point of view. Some preliminary conclusions on what characteristics benefit one and what characteristics the other solver are drawn. These conclusions are based on the solvers' formulations derived earlier in this chapter and information in literature.

In *icoFoam*, a time step consists of one implicit velocity computation and a set of explicit pressure-velocity computation loops, the PISO-loops [6]. During each PISO-loop, a pressure field that fulfills the continuity requirement is computed and used for correcting the flux and velocity fields, ϕ and $\mathbf{U}$. In *icoFoam* the computed velocity and pressure solutions are close to the exact solution after two PISO-loops. The temporal errors of velocity and pressure caused by the solution method, not time-discretization, are of magnitudes $O(\Delta t^4)$ and $O(\Delta t^3)$, respectively. Any further PISO-loop improves the accuracy by one factor. Since the error induced by the temporal discretization remains constant, $O(\Delta t^3)$ with a second order scheme, there is no need for many more PISO-loops because they would no more improve the accuracy of the computation [2].

In contrast to *icoFoam*, in *transientSimpleFoam* the momentum equation is underrelaxed and the new velocity solution does not correspond to the true velocity before convergence. This is the one most significant reason for the fact that, unlike in *icoFoam*, in *transientSimpleFoam* convergence does not occur after a few pressure-velocity iterations, i.e. SIMPLE-loops. Thus the required number of SIMPLE-loops is always greater than the required number of PISO-loops. This implies that the solution process with *transientSimpleFoam* requires more computational effort than with *icoFoam*, unless *transientSimpleFoam* benefits from other aspects.

One advantage of underrelaxation is, that it permits the utilization of a larger time step. A larger time step affects the required computational effort and evokes contrast between the solvers in at least two ways:

- In order to cover a specified time range in fewer time steps such that fewer time loops are required. This can reduce the computational effort of *transientSimpleFoam* in comparison with *icoFoam*.
- With larger time steps, the unlinear effects become more significant. This demands more iteration to account for the error caused by the lagged information, which increases the required computational effort per time step in *transientSimpleFoam*.

It is important to underline that the net effect of the mentioned features depends on the case.

3 The Case-Study: Taylor-Green Vortex

In this chapter the case-study will be presented. In section (3.1), the flow case is introduced in general. Earlier contributions made with the same case are brought out and the parameters that are varied in this study are clarified. The computation is performed fully automated with the help of Python-scripting. The way the case is set up, solved and post-processed is discussed in more detail in section (3.2). Section (3.3) covers the results from the computation and some of the information in Figures (3)-(11) is pointed out. In section (3.4) the results are interpreted and some conclusions are drawn based on the interpretation.

3.1 General

Setting up the computation includes creating a correct grid and setting the initial and boundary conditions for the problem. OpenFOAM's input files are manipulated to define the discretization schemes, relaxation factors and the number of SIMPLE- or PISO-loops.

For post-processing, the analytical solution is calculated, the computed OpenFOAM solutions are compared with it and the maximum and average errors of velocity are calculated. Also velocity magnitude- and pressure contours, velocity-vector fields and velocity and pressure values along the line $y = 0$ are plotted.

This script presented in reference [1] is used for computing the *icoFoam* case in this study. The script used for running the *transientSimpleFoam* computations is a modified version of the original script. In addition to divergence schemes and mesh density, it also varies the number of SIMPLE-loops within a time step (nCorrectors), the time step length and the relaxation factor of velocity. Each of the new variable parameters will be given two different values and all combinations will be computed. Thus eight *transientSimpleFoam* cases are run with each grid. Four mesh densities are used but the divergence scheme, QUICK, is kept constant in all cases in this study. However, the option to vary it is preserved in the script. The following values are given to the variable parameters:

mesh size	(8x8)	(16x16)	(32x32)	(64x64)

nCorrectors	30	80
time step	0.01 s	0.05 s
α_U	0.7	0.8

3.2 Structure

3.2.1 Setting Up and Running the Computation

Computing the cases with *transientSimpleFoam* is performed with five Python `for`-loops, one for each variable parameter. For all combinations of variables, a new folder is created.

First in each case folder, a baseline case consisting of the basic OpenFOAM case structure is copied into the folder. This forms a basis to the case. After copying the basic case, the grid is refined according to the current grid loop. Both of these tasks are performed with the help of the `PyFoam` library presented in reference [8]. `PyFoam` is a library that has been developed to help scripting OpenFOAM with Python.

`PyFoam` has an application for copying the basic OpenFOAM case structure, *pyFoamCloneCase.py*, that is used for copying the baseline case. Refining the grid is performed with a `PyFoam`'s class, BlockMesh, that contains a member, `refineMesh`. `refineMesh` manipulates a dictionary, *blockMeshDict*, used by OpenFOAM's own mesher utility, `blockMesh`, to set the correct mesh density. When the *blockMeshDict* has been manipulated, `blockMesh` is run in the script [1].

Setting the rest of the variable parameters requires manipulating OpenFOAM's set-up files. Divergence scheme is set in the */system/fvSchemes* dictionary, the number of SIMPLE-loops and relaxation factors are set in *system/fvSolution* and time step in *system/controlDict*. All of these manipulations are performed by using standard Python functions.

Initial conditions are set with the help of a utility presented in reference [9], `funkySetFields`. It makes it possible to easily set a non-uniform initial condition on a patch using a dictionary, *system/funkySetFields*. Initial conditions are the same in all cases so the dictionary is included in the baseline case and `funkySetFields` is run by the script in each case folder.

The computation is run by calling the solver in the script. Both solvers read the dictionaries *system/fvSchemes*, *system/fvSolution*, *system/controlDict*, *constant/transportProperties*, *constant/turbulenceProperties*, *0/U* and *0/p*.

3.2.2 Post-processing the Computation

For computing the error between the computed and analytical solutions, a custom OpenFOAM utility presented in reference [1], *analyticalSolutionTaylorVortex*, has been developed. It computes the error at each cell center after reading the computation results and computing the correct values according to the analytical solution.

To see the role of the varied parameters with different mesh densities, the maximum and average errors are extracted from the data. A temporary file corresponding to each case is saved in the *results/*-folder. Each of the files contains mesh- and error data from a computation with one combination of parameters. That data is plotted by the subscript *tools/plotError.py*. After the data is plotted the data files are deleted.

OpenFOAM's own post-processing utility, `sample`, is used for extracting data from lines $y = 0$ and $x = 0$ as well as in the part of $x-y$-plane that the mesh covers. `Sample` reads the *system/sampleDict* dictionary in which the mentioned sets are defined.

For plotting velocity magnitude-, pressure- and error contours, there is a subscript *tools/plotContours.py*. It reads the data extracted by the `sample`- utility, creates the con-

tours and saves the produced pictures. The subscript, *tools/plotVectors.py*, plots velocity vector fields. Also this subscript reads the data extracted by the `sample`-utility.

To compare results from computations with different parameter combinations but with a given mesh, two plots are created: One plot presents the *v*-velocity component, the other the pressure values, against the *x*-coordinate at $y = 0$. This is performed by two subscripts, *tools/collectLinesData.py* and *tools/plotLines.py*. The first subscript reads the velocity and pressure data extracted by the `sample`-utility. It gathers the data and saves it into temporary files so that each file contains data from computation with one of the eight parameter combinations, with all mesh densities. The latter subscript then reads the temporary files and plots the data. All pictures are automatically moved into *figures/* -folder.

Finally, the subscript *tools/appendix.py* creates a file that contains chosen pictures in *.tex*-format. It adds the grid convergence data from computation with each set of parameters as well as from the *icoFoam* computation. The produced file forms the appendix of this work.

3.3 Results

In this chapter some of the information in the figures in the Appendix is pointed out and some general observations are made. The information will be interpreted and conclusions drawn in chapter (3.4).

3.3.1 Errors across the Grid

The upmost subfigures in Figures (3)-(11) show the errors in the velocity *v*-component and pressure along the line $y = 0$. Next, the contents of those subfigures will be described in three groups. The results from the *icoFoam* computation is covered separately and the *transientSimpleFoam* computations with the smaller time step and the *transientSimpleFoam* computations with the larger time step separately.

icoFoam

Figure (3) shows that the results from the *icoFoam* computations follow the analytical solution with all mesh densities. There is visible oscillation with all of the mesh densities. The oscillation reduces when the grid is refined.

***transientSimpleFoam*; smaller time step**

All *transientSimpleFoam* computations with the smaller time step give almost identical results with each other. In *v*-velocity there is no noticable deviation from the analytical solution with any of the meshes.

The pressure solutions vary more. They oscillate around the analytical solution so that the oscillation reduces when the grid is refined. The deviations from the analytical solution are small.

***transientSimpleFoam*; larger time step**

In all of the *transientSimpleFoam* computations with the larger time step, every grid gives results that visually differ from the analytical solution.
With 80 SIMPLE-loops and α_U = 0.8, all results are similar to each other but they visually differ from the analytical solution. In computations with other parameter combinations, the errors are approximately equal to these errors, except for the cases with the finest grid. The deviations from the analytical solution with the finest grid in both pressure and *v*-velocity, are many times as large as with the coarser grids.

3.3.2 Maximum Errors

Figure (2) shows the maximum velocity magnitude error in the *transientSimpleFoam* and *icoFoam* computations. The upper subfigure represents the *transientSimpleFoam* computations. The errors from computations with each combination of parameters (different curves) are plotted against mesh density (horizontal axis). The lower set of lines contains the results represent computations with the smaller time step. In the upper set of lines the greater time step was used.
In the lower subfigure there is only one line. It is the result from the *icoFoam* computations. The results from the *transientSimpleFoam* computations are discussed first.

transientSimpleFoam

With the two coarsest meshes the only parameter affecting the error is time step. With the second finest mesh, also the relaxation factors have influence on the error. The greater α_U gives better accuracy but only if the number of SIMPLE-loops is 30. With 80 SIMPLE-loops both underrelaxation factors give equal error.

With the finest mesh, the differences between different parameter combinations are emphasized. With the smaller time step only the case with 30 SIMPLE-loops and relaxation factor $\alpha_U = 0.7$ gives a greater error when compared with other cases with the same time step.
With the greater time step, however, all four cases give deviatign errors. In these cases, more SIMPLE-loops and a greater relaxation factor give a smaller error.

With the larger time step of 0.05 s computations with the finest mesh result in greater errors than computations with the second finest mesh. This can be seen with all variable combinations, except for the case with $\alpha_U = 0.8$ and 80 SIMPLE-loops.
With the smaller time step only the computation with the smaller $\alpha_U = 0.7$ and 30 SIMPLE-loops gives greater errors with the finest mesh than with the second finest mesh.

icoFoam

The errors in the *icoFoam* computations are of the same magnitude as in the *transientSimpleFoam* computations with the same time step, Δt = 0.05 s. The error order of magnitude in the *icoFoam* computations decreases with increasing mesh density, but not linearly, as it does in the most accurate cases with *transientSimpleFoam*.

3.4 Discussion and Conclusions

In this chapter, the results described in Sec. (3.3) will be interpreted. First some conclusions will be drawn based on the effect of different parameters on the results with a given mesh density. After that, the mesh density's effect on how the parameters affect the results will be considered. All of these conclusions concern a SIMPLE-based solver. Finally, there will be discussion on in what kind of cases the benefits of SIMPLE- and PISO - based solvers can be exploited.

3.4.1 Effect of the Parameters with a Given Mesh Density

In this subsection the effect of the varied parameters with a given mesh density are studied. Thus, only the results from the *transientSimpleFoam* computations can be considered. The results show that, in a SIMPLE-based solver, if the number of SIMPLE-loops has any effect on the results, increasing the number improves the results. A smaller relaxation factor and a larger time step increase the significance of SIMPLE-loops. Similarly, if the relaxation factor has an effect on the results, a greater factor improves them. A larger time step and less SIMPLE-loops increase the significance of the relaxation factor. As expected, a smaller time step gives always better results than a larger one. Next, the reasons behind these findings will be discussed. Each of the varied parameter will be considered individually.

SIMPLE-loops

The fact that a computation with more SIMPLE-loops always results in a or equal error as a computation with less SIMPLE-loops, implies that reaching a sufficient convergence within each time step is required to reach a maximum accuracy determined by other factors. However, after a specific number of SIMPLE-loops, i.e. after the sufficient convergence is reached, increasing the number does not improve the results.

Time step

A smaller time step reduces the effect of SIMPLE-loops on the error which is at least due to the fact that unlinear effects are not so significant with a smaller time step. The unlinear effects are greater with a larger time step [6] and, to account for them and to reach the mentioned sufficient convergence within each time step, more iterations are required.

Relaxation factors

The improving effect of a larger relaxation factor on the result is due to the fact that a larger relaxation factor accelerates the convergence of the computation. However, it is so only if the computation does not become unstable due to the larger relaxation factor. Thus, no such conclusion can be made that increasing the relaxation factor will always improve the results, because at a too large value the computation may diverge [10].

3.4.2 Effect of Mesh Density

The error order of magnitude decreases in most of the *transientSimpleFoam* computations linearly with an increasing mesh density. In some cases, however, this is not the case. It is seen that the denser the mesh, the more critical other parameters affecting convergence become for the results.

The results indicate that on a coarser grid smaller number of SIMPLE-loop iterations is required to reach a sufficient convergence within a time step. However, as the mesh density is increased, reaching this level requires a greater number of iterations. This is because of the relaxation factors. The iterative solution procedure within each time step can be described as marching in pseudo-time. Smaller cells correspond to a smaller pseudo-time step and thus cause slower convergence and a need for more SIMPLE-iterations. [11]

If a multigrid algorithm is used in solving the matrix equations, the time required for solving the equations increases linearly with the number of cells [10]. In other words, the time required by each SIMPLE-loop grows linearly. Since also the required number of SIMPLE-loops increases, the total computational effort required to reach good results increases faster than in a linear manner.

A larger relaxation factor results in a faster convergence, regardless of mesh density. Thereby it improves the results. However, this is the case only if the relaxation factor does not cause the computation to diverge. Whether the mesh density affects the role of relaxation factors in keeping the computation stable, requires further considerations and is beyond the scope of this study.

In the *icoFoam* results similar phenomena can be seen as in the *transientSimpleFoam* results. The error order of magnitude does not drop linearly when the mesh is refined in an exponential manner. It might do so if the time step was shortened. The nature of the PISO-algorithm is that the temporal errors decrease more effectively when time step is shortened than the spatial errors when the grid is refined [2]. However, these *icoFOAM* results were computed with three PISO-loops which is much less than the 30 SIMPLE-loops used in *transientSimpleFoam*. The function of PISO-loops is different from the function of SIMPLE-loops and increasing the number of PISO-loops would probably not fix the problem. Since no underrelaxation is applied in the PISO-algorithm, the time step remains the only parameter that can be used for achieving the accuracy dictated by the spatial discretization.

3.4.3 Benefits of PISO- and SIMPLE- based solvers

All findings mentioned in the two preceding subchapters imply that a solver with a SIMPLE algorithm requires a sufficient convergence within each time step. If the tolerance is not met, the initial conditions for the following time step are not precise enough and the error will be accumulated. All of the varied parameters have influence on if the sufficient tolerance is met. They have the following role in the iteration process:

- A smaller time step reduces the unlinear effects and thus reduces the iteration required to reach the sufficient tolerance that assures good results.
- A smaller relaxation factor postpones convergence and thus causes a need for more iteration rounds.
- A finer mesh slows down the converging process and thus more iteration within a time step is required to reach given tolerances.
- With more SIMPLE-loops, tighter tolerances are reached.

Dispite of the fully implicit discretization in *icoFOAM*, the stability of the computation is impaired by the intermediate steps in the solution procedure. The pressure corrections are of explicit nature. This imposes time step limitations that are difficult to formulate in a precise manner [2]. Since no underrelaxation is applied in *icoFOAM*, time step provides the only means to account for the unlinear effects and to adjust the accuracy with a given mesh.

The potential benefit of the SIMPLE-algorithm in comparison with the PISO-algorithm is, that underrelaxation and the number of SIMPLE-loops can be used to adjust the computation to account for problems induced by a longer time step. In this perspective the problem of choosing one of these solvers can be turned into the problem of deciding if the longer time step offsets the extra costs caused by adjusting the computation with underrelaxation and the number of SIMPLE-loops. Thus, the following conclusion can be drawn.

Using a SIMPLE-based solver can be recommended in cases where unlinear effects are not dominating. In such cases a longer time step can be utilized and thus a specified time range can be covered with less computational effort with a SIMPLE- than with a PISO-based solver. However, if the unlinear effects are great or the mesh is large enough, a SIMPLE-based solver requires strict tolerances i.e. many SIMPLE-loops within each time step, which cancels out the benefits achieved with the longer time step.

Because a PISO-based solver lacks underrelaxation, time step is the only way to improve accuracy and the required time step is always relatively short. Since the accuracy of a PISO-based solver is at it maximum with very few PISO-loops, a PISO-based solver is a good solver choise for cases with strong unlinearity.

4 Error Discussion and Further Subjects for Study

In this study three parameters were varied. There are many more parameters that affect the computation and its convergence, for example discretization methods. Changing these parameters would lead to different results. However, the phenomena seen in the results would probably not change considerably.

In this case-study very simple, orthogonal, meshes were used. Thus, it remains unclear how the solvers behave with more complex meshes with non-orthogonal terms included. Non-orthogonalities will produce new terms into the equations and thus affect the required computational effort. This study, however, does not tell if they affect one solver more than the other and if they do, which solver will benefit more. Thus, the effect of the cells' shapes on the convergence and feasibility of the solvers would be an interesting subject for study.

The way in which a finer mesh affects the required computational effort remains somewhat ambiquous. It is seen that the required number of SIMPLE-loops increases with mesh size. However, with different tolerances within a SIMPLE-loop the results of this study would be different. One parameter might be emphasized more and another's effect might be less obvious. Thus a study on how much more SIMPLE-loops are required with different tolerances within a SIMPLE-loop would give a better insight into the problem and thus also a better starting point for a solver choise.

References

[1] Paterson Eric G., “Python Scripting for Gluing CFD Applications: A Case Study Demonstrating Automation of Grid Generation, Parameter Variation, Flow Simulation, Analysis, and Plotting” , The Pennsylvania State University, The Applied Research Laboratory Technical Report No. TR 09-001, 13 January 2009

[2] Issa R.I., “Solution of the Implicitly Discretised Fluid Flow Equations by Operator-Splitting”, *Journal of Computational Physics*, **62**(11), pp. 40-65, 1986

[3] Patankar S.V., “Numerical Heat Transfer and Fluid Flow”, Hemisphere, Washington, D.C, 1980

[4] OpenFOAM Website.
http://www.openfoam.com, January 2011

[5] OpenCFD Ltd., OpenFOAM Programmers Guide, Version 1.6, July 2009

[6] Jasak H., “Error Analysis and Estimation for the Finite Volume Method with Applications to Fluid Flows”, University of London, Imperial College, June, 1996

[7] Auvinen M.; Ala-Juusela J.; Pedersen N.; Siikonen T., "Time-Accurate Turbomachinery Simulations with Open-Source CFD; Flow Analysis of a Single-Channel Pump with OpenFOAM", "ECCOMAS CFD 2010", Fifth European Conference on Computational Fluid Dynamics, J.C.F Pereira and others, 2010

[8] B. Gscheider, PyFOAM Website. http://openfoamwiki.net/index.php/Contrib_PyFoam, January 2011

[9] B. Gscheider, OpenFOAM Utility: funkySetFields.
http://openfoamwiki.net/index.php/Contrib_funkySetFields, January 2011

[10] Siikonen Timo, “Virtaussimulointi”, Helsinki University of Technology, Department of Applied Mechanics, 2010

[11] Ferziger J.H.; Peric M., “Computational Methods for Fluid Dynamics”, Springer, 3rd edition, ISBN 3-540-42074-6, 2002

Appendix 1

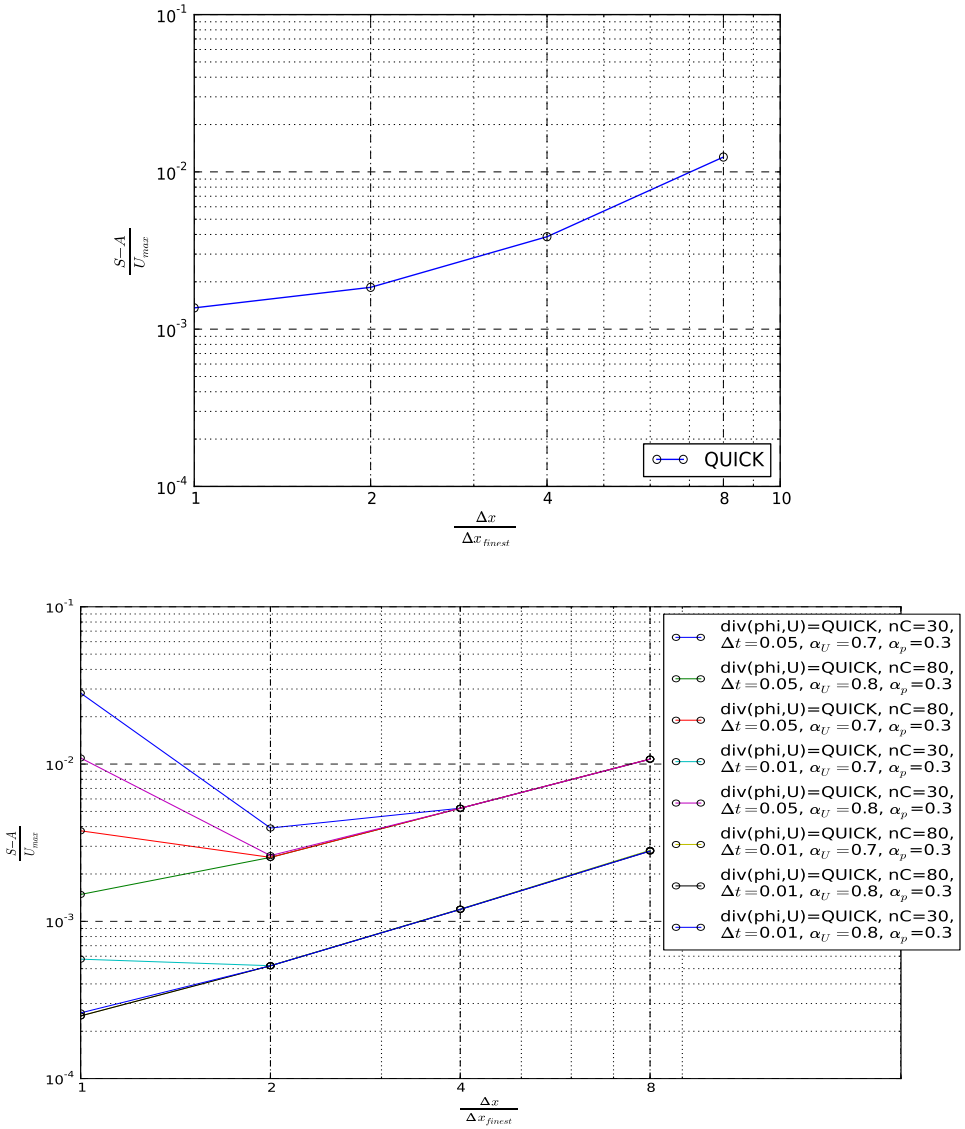

Figure 2 Convergence of Spatial and Temporal Errors with *transientSimpleFoam* and icoFoam. *icoFoam* in the upper and *transientSimpleFoam* in the lower picture

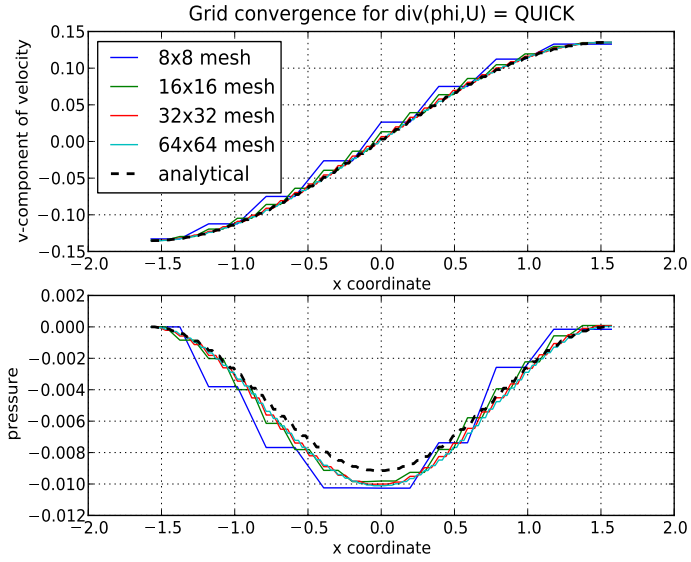

Figure 3 div(phi,U)=QUICK, mesh (32x32), *icoFoam*

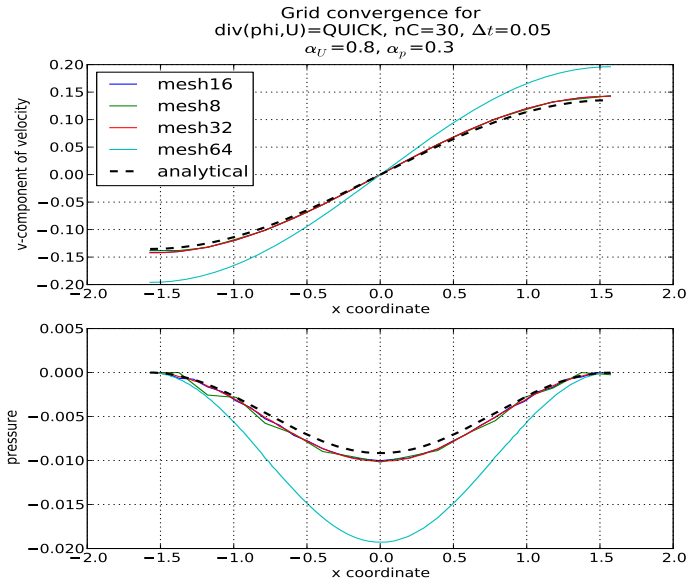

Figure 4 div(phi,U)=QUICK, mesh=(32x32), nCorrectors=30, Δt=0.05 , α_U=0.7, α_p=0.3

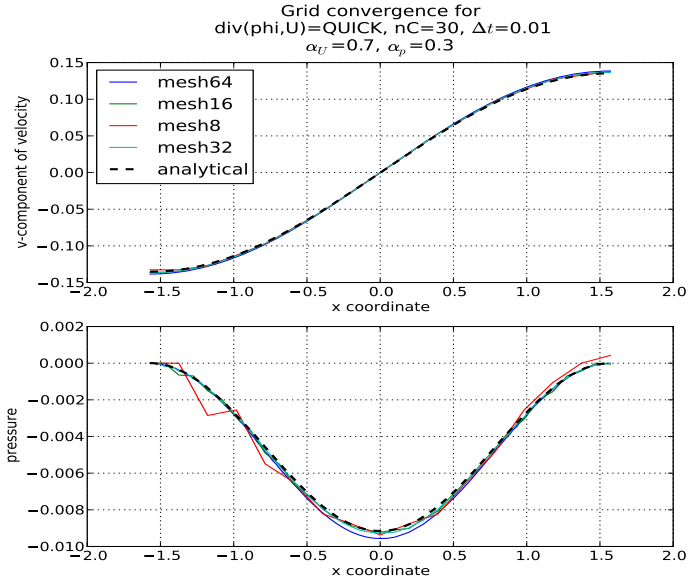

Figure 5 div(phi,U)=QUICK, mesh=(32x32), nCorrectors=80, Δt=0.01 , α_U=0.7, α_p=0.3

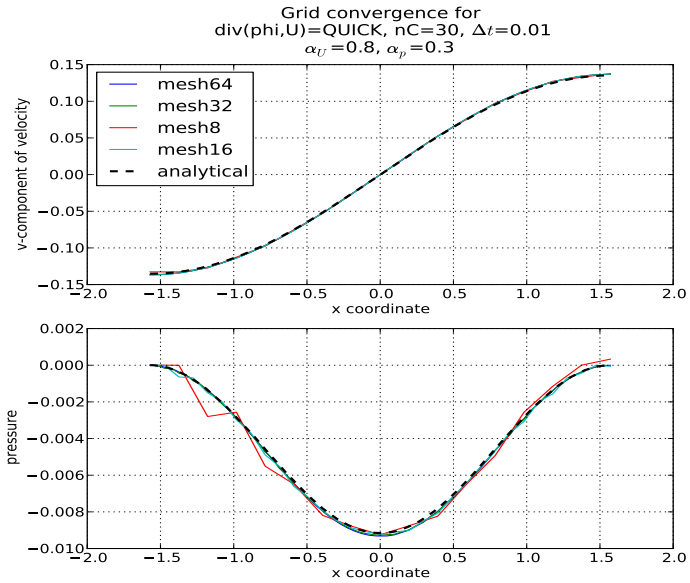

Figure 6 div(phi,U)=QUICK, mesh=(32x32), nCorrectors=80, Δt=0.05 , α_U=0.8, α_p=0.3

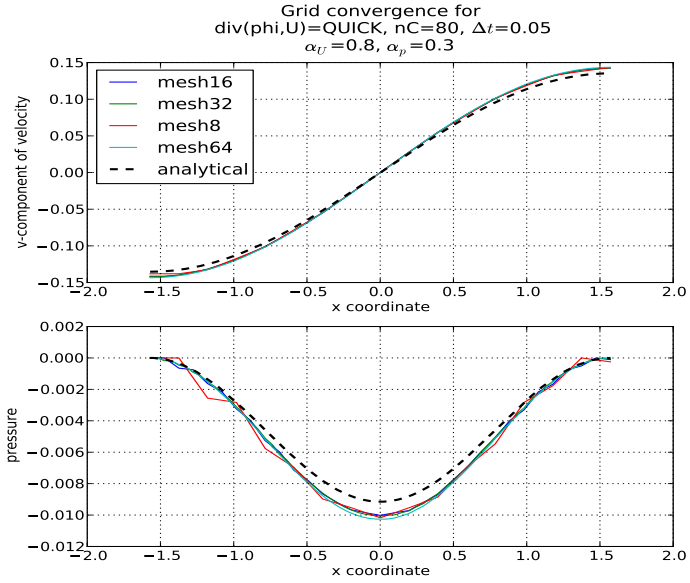

Figure 7 div(phi,U)=QUICK, mesh=(32x32), nCorrectors=80, Δt=0.05 , α_U=0.7, α_p=0.3

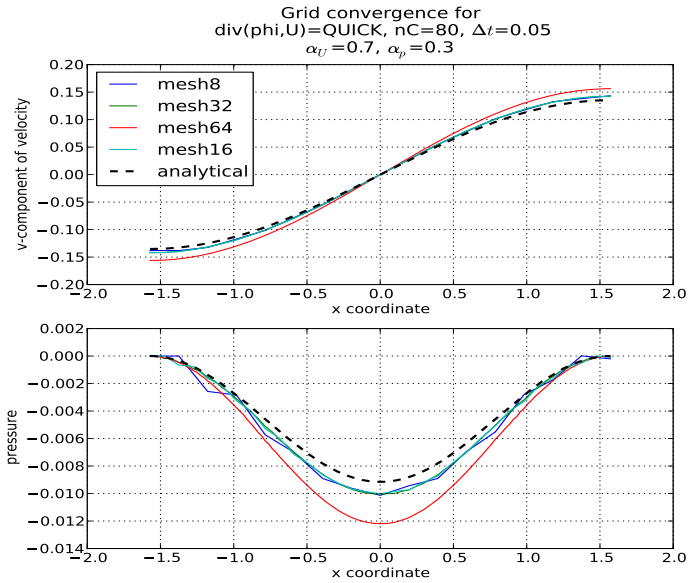

Figure 8 div(phi,U)=QUICK, mesh=(32x32), nCorrectors=30, Δt=0.01 , α_U=0.8, α_p=0.3

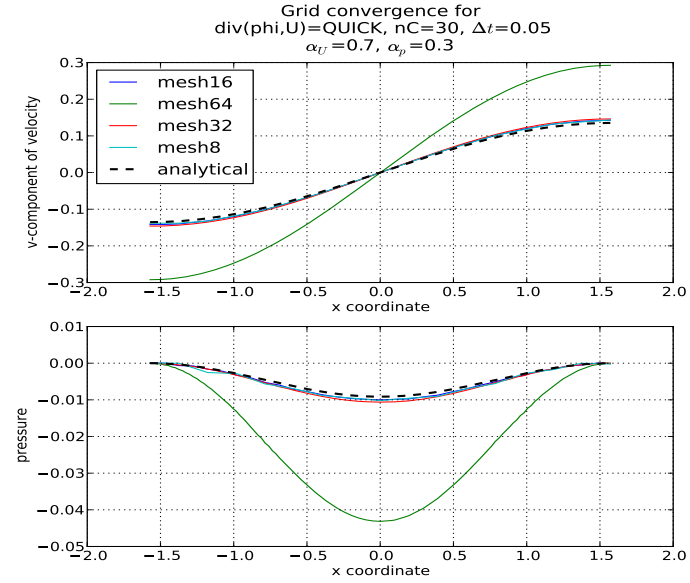

Figure 9 div(phi,U)=QUICK, mesh=(32x32), nCorrectors=30, Δt=0.05 , α_U=0.8, α_p=0.3

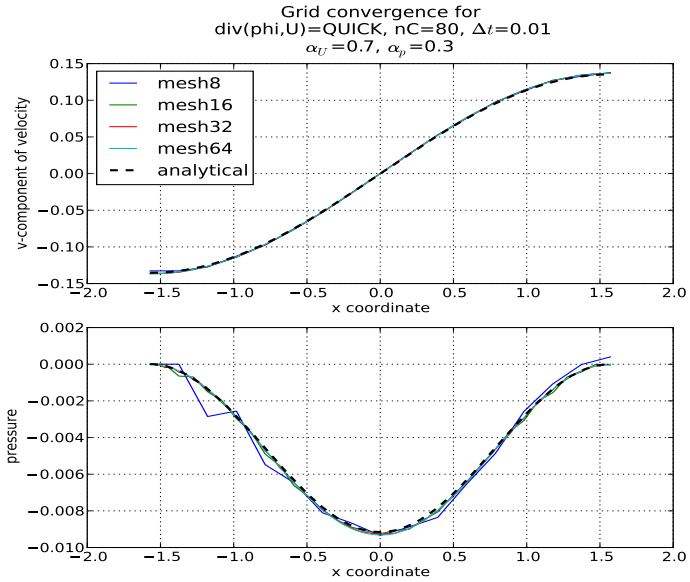

Figure 10 div(phi,U)=QUICK, mesh=(32x32), nCorrectors=80, Δt=0.01 , α_U=0.8, α_p=0.3

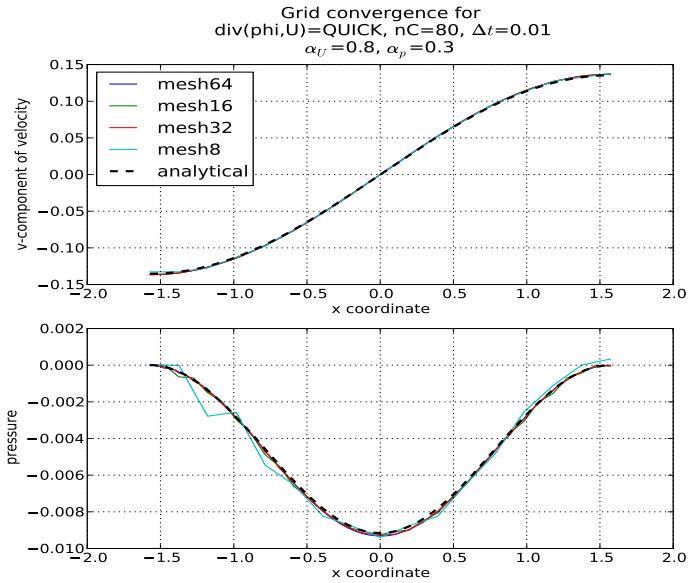

Figure 11 div(phi,U)=QUICK, mesh=(32x32), nCorrectors=30, Δt=0.01 , α_U=0.7, α_p=0.3