NTNU
Kunnskap for en bedre verden

Department of Marine Technology

TMR4345 - Marine Computer Science Lab

The Projection Method

Author:
Åsmund Aamodt Resell

June, 2020

Abstract

This project was developed as the grade giving assignment in the course Marin Computer Science Lab at the Norwegian University of Science and Technology. For this task, a simple Navier-Stokes solver was to be developed. The code was initially intended to be written in Fortran or C, however permission was given to write the project in C++. A Matlab script named sola.m, found in appendix D.1 was also handed out, along with a youtube video [1] and a PDF [2] explaining the code. The solving algorithm for the project is based upon this Matlab script.

In addition to creating the solving algorithm, the flow around a square, along with its forces was to be calculated. The student was also free to experiment with other geometries and parameters.

It was concluded that the final program generated desirable results that was validated with sola.m. However improvements could be made both with regards to compilation time, and to the accessibility.

Table of Contents

List of Figures

List of Tables

1 Introduction

In the introductory courses to fluid mechanics, we have been introduced to equations and principles that help us describe and understand the physical nature of fluids, and it's properties. We have acquired a set of tools that describe certain problems where we can obtain an analytical solution. However on most physical problems, it can become quite difficult to obtain solutions by only relying on these tools. This is especially true for engineering applications, where the situations often can become complex and impossible to solve with an analytical approach. Sometimes we also want to solve the system for transient problems, which adds another layer of complexity to the problem. Instead we can use a numerical approach to obtain an approximation of the exact solution. With a numerical solver combined with the power of modern computing, most physical problems can be described with a satisfying accuracy.

The project consists of building a simple Navier Stokes solver, and apply it to a variety of fluid flow problems. The first problem was to solve a flow in a cavity box, with a moving top lid. The other required task was to model a square in a uniform flow, and calculate acting forces. The student was also free to experiment with other geometries and parameters. In this thesis the Projection Method is explored, and applied on a MAC-grid (Marker and Cell), with a set of boundary conditions for each flow case.

The first and most important motivation for the project was to validate the solver as a useful tool to experiment with fluid dynamics. Another important motivation for the project was to learn more about compilation time. While algorithms are known for solving complex problems regarding fluids, computers are still not powerful enough to solve every problem within minutes. Calculations for complex problems can take hours or even days to solve, and its one of the most limiting factors within numerical simulations. This aspect has been central during the development and evaluation of the project.

2 Theoretical Background

A mathematical model of any physical problem needs a set of equations and boundary conditions, as well making certain assumptions to simplify and quantify it's properties. A numerical method is also required to actually solve the problem at hand. In the this chapter a description of the theory, assumptions, boundary conditions and method of solving is further explored. Following the suggestions from the excerpt from R. Kristoffersen, *A navier-stokes solver using the multigrid method*, 1994 [2] similar descriptions are used to represent this problem.

2.1 Assumptions

The following assumptions are made as suggested. [2]

- Incompressible flow
- Constant viscosity
- No body forces
- Single-component fluid
- Newtonian fluid
- Laminar flow
- 2-dimensional
- Cartesian coordinates

These assumptions will be important as they greatly simplify the model. The fact that a one component, incompressible flow is used, enables the use of a constant density for the fluid property. Since also a constant viscosity is used, the energy equation can be disregarded when assessing the continuity of mass and momentum.

2.2 Governing Equations

When describing a viscous flow the Navier-Stokes equations are used. As the density is assumed to be be constant, the incompressible version of these equations can be adopted.

$$\frac{\partial \vec{V}}{\partial t} + (\vec{V} \cdot \nabla)\vec{V} = -\frac{1}{\rho}\nabla p + \nu \nabla^2 \vec{V} \tag{1}$$

Notice that the gravitational term that is normally included have been neglected due to the assumption of no body forces. Also while the equation in general describes both laminar, transitional and turbulent flow, this solver will only be able to describe laminar flow. This comes from the assumption of 2-dimensional flow, which is not able to give an accurate description of turbulent flow as it requires a 3-dimensional space to develop the turbulent bodies. For further demonstration, a one component version of the Navier-Stokes equations is demonstrated using tensor notation.

$$\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial p}{\partial x_i} + \nu \frac{\partial^2 u_i}{\partial x_j \partial x_j} \tag{2}$$

The equation for conservation of mass becomes independent of the variation in density, since the density is constant. The incompressible version for the continuity equation can then be used

$$\nabla \cdot \vec{V} = 0 \tag{3}$$

Which in similar fashion as equation (2) can be written in a tensor notation the following way

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{4}$$

Note that the conservation of energy is usually also assessed in these types of problems. However it will not be important for this project as we are only interested in the resulting velocity and pressure.

2.3 Nondimensionalization

It can be beneficial to nondimensionalize the equations describing the problem at hand. By doing so, the number of free parameters decreases. Parameters such as length and velocity can be described implicit through nondimensional numbers. For this problem, the Reynolds number and the Strouhal number are used.

$$Re = \frac{UL}{\nu} \tag{5}$$

$$St = \frac{L}{TU} \tag{6}$$

The Reynolds number includes the reference velocity U, the length scale L and the viscosity ν. The Strouhal number also includes a reference timescale T, while not including the viscosity. By using these dimensionless numbers, the parameters mentioned does not need to be specified for each flow problem. Instead the only parameters for the flow properties that needs to be specified are these two dimensionless numbers. They can be incorporated in the incompressible Navier Stokes equations.

$$\frac{1}{St}\frac{\partial u_i}{\partial t} + u_j\frac{\partial u_i}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \frac{1}{Re}\frac{\partial^2 u_i}{\partial x_j \partial x_j} \tag{7}$$

While a reference velocity and a reference length scale is important for describing this problem, the characteristic time scale is not assessed, such that the problem is reduced to only using the Reynolds number as a parameter.

2.4 The Projection Method

The projection method was originally introduced in 1967 by Alexandre Joel Chorin [3], It is a potent method for solving the incompressible Navier-Stokes equations for transient problems. The method is formulated in such a way, that the pressure and the velocity is decoupled from one another. The method is demonstrated as follows, consider the dimensionless one component Navier-Stokes equation, with an explicit forward in time discretization.

$$\frac{1}{St}\frac{u_i^{n+1} - u_i^n}{\Delta t} + u_j^n\frac{\partial u_i^n}{\partial x_j} = -\frac{\partial p^n}{\partial x_i} + \frac{1}{Re}\frac{\partial^2 u_i^n}{\partial x_j \partial x_j} \tag{8}$$

In this formulation the only unknown variable to solve is u_i^{n+1}. All variables at time step $n\Delta t$ are known, and the formulation corresponds to an explicit first-order Euler scheme. The problem with this scheme is that the velocity field does not satisfy the incompressible continuity equation. Therefore the pressure field needs to be adjusted in such a way that the mass conservation (4) for

$t = (n+1)\Delta t$ is satisfied across the entire field. The adjusted pressure will therefore have to satisfy the pressure for the new timestep, and the equation becomes

$$\frac{1}{St}\frac{u_i^{n+1} - u_i^n}{\Delta t} + u_j^n \frac{\partial u_i^n}{\partial x_j} = -\frac{\partial p^{n+1}}{\partial x_i} + \frac{1}{Re}\frac{\partial^2 u_i^n}{\partial x_j \partial x_j} \tag{9}$$

The equation is no longer explicit, since both the velocity u_i^{n+1} and the pressure p^{n+1} are unknown. To solve the problem, first a tentative velocity field is formulated.

$$\frac{1}{St}\frac{u_i^{*} - u_i^n}{\Delta t} + u_j^n \frac{\partial u_i^n}{\partial x_j} = -\frac{\partial p^{n}}{\partial x_i} + \frac{1}{Re}\frac{\partial^2 u_i^n}{\partial x_j \partial x_j} \tag{10}$$

This tentative velocity can be solved explicit, since the pressure is known. Next the two equations are subtracted from one another (9)-(10):

$$\frac{1}{St}\frac{u_i^{n+1} - u_i^{*}}{\Delta t} = -\frac{\partial}{\partial x_i}(p^{n+1} - p^n) \tag{11}$$

While the tentative velocity can be calculated from (10) the new velocity and pressure are still unknown. To reduce the number of unknown variables the continuity constraint

$$\frac{\partial u_i^{n+1}}{\partial x_i} = 0 \tag{12}$$

can be applied by taking the divergence of equation (11). The new velocity u_i^{n+1} is thereby removed from the equation due to the continuity constraint, and the new equation is obtained

$$\frac{1}{St\Delta t}\frac{\partial u_i^{*}}{\partial x_i} = \frac{\partial^2}{\partial x_i^2}(p^{n+1} - p^n) \tag{13}$$

This equation takes the form of the Poisson equation. It needs to be solved in order to obtain the pressure p^{n+1}. This operation is not trivial and need to be solved iteratively. For each iteration a divergence is calculated and used to update the pressure until convergence is reached. The algorithm for the the system becomes the following:

1. Calculate the tentative velocity u_i^* from equation (10).
2. Find the pressure for the next timestep p^{n+1} using (11).
3. Update the values for the velocity field u_i^{n+1} at the next timestep.

2.5 Discretization of the Domain

Now that the method for solving is established, a discretization of the domain is required. For this the grid system called MAC (Marker And Cell) [4] is used. This grid is suggested for solving viscous, incompressible, transient fluid flow problems. The grid is a staggered system, which means that the scalar variables for each cell is stored in the center of the cell, while the velocities are stored at the cell boundaries. For this problem the scalar variable that is of interest is the pressure values. For a single cell the general formulation will correspond to figure 1.

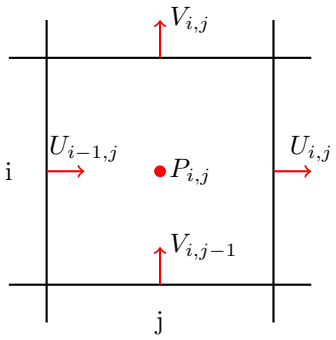


Figure 1: A general MAC cell

Here the components $U_{i,j}$ and $V_{i,j}$ corresponds to the velocity components at location $x = i\Delta x$, $y = j\Delta y$ in the x and y direction respectively.

A formulation for the whole domain is also needed. If the size of domain is of the reference length scale L in both the x and y direction, the grid can be discretized with N values in both directions such that $L = N\Delta x = N\Delta y$. However the MAC method also requires a ghost cell past each boundary, so the total number of cells in each direction becomes N+2. The complete grid is shown in figure 2 The total grid size will be (N+2)*(N+2). However only the white cells are used to actually update the velocities. The ghost cells are only used to establish boundary conditions.

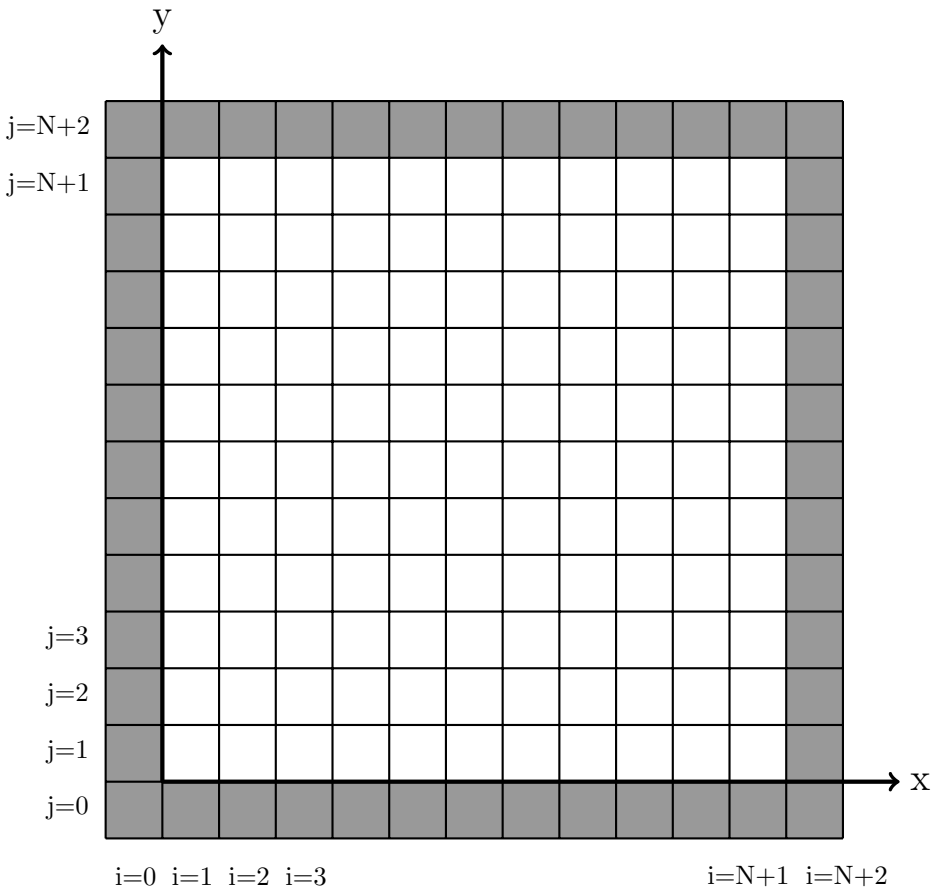


Figure 2: The grid for the entire domain, the ghost cells are colored grey

2.6 Numerical Scheme

To establish a numerical scheme, the driving equation for the tentative velocity (10) needs to be discretized, not only in the time domain, but also in the spatial domain. First the equation is rewritten. Consider the following derivative

$$\frac{\partial u_i u_j}{\partial x_j} = u_j \frac{\partial u_i}{\partial x_j} + u_i \frac{\partial u_j}{\partial x_j} \tag{14}$$

From the continuity equation (4) the last term is cancelled. This equation can now be substituted for the convective term into (10)

$$\frac{1}{St}\frac{u_i^* - u_i^n}{\Delta t} + \frac{\partial u_i^n u_j^n}{\partial x_j} = -\frac{\partial p^n}{\partial x_i} + \frac{1}{Re}\frac{\partial^2 u_i^n}{\partial x_j \partial x_j} \tag{15}$$

This way of writing the convective term is called the conservative form. The benefit of using this form is that the solution is more stable to instabilities such as shock waves. Now the terms needs to be discretized. Central differences are used for the discretization for all the terms. From now on the discrete velocity terms are used to establish the numerical scheme. The subscripts i and j is are no longer used as a dimensional subscript but as spatial subscript in the x and y direction. Consider the x-component of the Navier-Stokes equation for the tentative velocity

$$\frac{1}{St}\frac{U_{i,j}^* - U_{i,j}^n}{\Delta t} = -\frac{\partial (U_{i,j}^n)^2}{\partial x} - \frac{\partial U_{i,j}^n V_{i,j}^n}{\partial y} - \frac{\partial p_{i,j}^n}{\partial x} + \frac{1}{Re}\left(\frac{\partial^2 U_{i,j}^n}{\partial x^2} + \frac{\partial^2 U_{i,j}^n}{\partial y^2}\right) \tag{16}$$

Since the equation is solved for the tentative velocity component $U_{i,j}^*$ the central differences needs to be centered around this component. For the viscous terms this is straight forward

$$\frac{\partial^2 U_{i,j}^n}{\partial x^2} \approx \frac{U_{i-1,j}^n - 2U_{i,j}^n + U_{i+1,j}^n}{\Delta x^2} \tag{17}$$

$$\frac{\partial^2 U_{i,j}^n}{\partial y^2} \approx \frac{U_{i,j-1}^n - 2U_{i,j}^n + U_{i,j+1}^n}{\Delta y^2} \tag{18}$$

The convective (conservative) terms are not equally trivial because not all the terms are centered directly around the velocity component $U_{i,j}$. Consider figure 3, all the velocities that are used to calculate the tentative component $U_{i,j}^*$ are included in this figure. Notice the black dots that are marked on the figure. These dots represent the location at where the central difference components for the convective terms needs to be evaluated. The two black dots located horizontally are used to calculate the convective term $\frac{\partial (U_{i,j}^n)^2}{\partial x}$ from equation (16). A central difference around $U_{i,j}$ at these two points becomes

$$\frac{\partial (U_{i,j}^n)^2}{\partial x} \approx \frac{(\frac{1}{2}(U_{i+1,j}^n + U_{i,j}^n))^2 - (\frac{1}{2}(U_{i-1,j}^n + U_{i,j}^n))^2}{\Delta x} \tag{19}$$

For the second convective term $\frac{\partial U_{i,j}^n V_{i,j}^n}{\partial y}$ in equation (16), the central difference components are evaluated at the two black dots located on the vertical line across $U_{i,j}$. The scheme for this term now depends on both the U and V velocity terms

$$\frac{\partial U_{i,j}^n V_{i,j}^n}{\partial x} \approx \frac{\frac{1}{2}(U_{i,j+1}^n + U_{i,j}^n)\frac{1}{2}(V_{i,j}^n + V_{i+1,j}^n) - \frac{1}{2}(U_{i,j-1}^n + U_{i,j}^n \frac{1}{2}(V_{i+1,j-1}^n + V_{i,j-1}^n}{\Delta y} \tag{20}$$

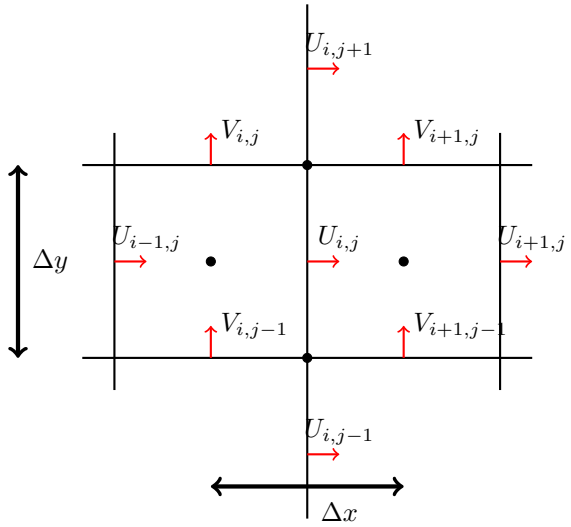


Figure 3: All velocities

To obtain the final explicit scheme for the tentative velocity, the viscous and convective schemes are now substituted into equation (16) along with the central difference for the pressure term. The pressure terms can be evaluated directly since they are located in the center of the MAC cell as shown in figure 1.

$$\begin{aligned}\frac{1}{St}\frac{U^*_{i,j}-U^n_{i,j}}{\Delta t} &= -\frac{(\frac{1}{2}(U^n_{i+1,j}+U^n_{i,j}))^2-(\frac{1}{2}(U^n_{i-1,j}+U^n_{i,j}))^2}{\Delta x}\\ &-\frac{\frac{1}{2}(U^n_{i,j+1}+U^n_{i,j})\frac{1}{2}(V^n_{i,j}+V^n_{i+1,j})-\frac{1}{2}(U^n_{i,j-1}+U^n_{i,j}\frac{1}{2}(V^n_{i+1,j-1}+V^n_{i,j-1}}{\Delta y}\\ &-\beta\frac{P^n_{i+1,j}-P^n_{i,j}}{\Delta x}+\frac{1}{Re}\left[\frac{U^n_{i-1,j}-2U^n_{i,j}+U^n_{i+1,j}}{\Delta x^2}+\frac{U^n_{i,j-1}-2U^n_{i,j}+U^n_{i,j+1}}{\Delta y^2}\right]\end{aligned} \tag{21}$$

Lastly an equation is needed to check for convergence of the Poisson equation. It is known that when mass conservation is preserved on the whole grid, the solution will have converged at that timestep. Therefore the continuity equation should be applied on every cell in the grid and the equation becomes the following

$$\nabla\cdot\vec{V}=\frac{U_{i,j}-U_{i-1,j}}{\Delta x}+\frac{V_{i,j}-V_{i,j-1}}{\Delta y} \tag{22}$$

Convergence will then be obtained when equation (22) is close to zero.

2.7 Successive Over-Relaxation (SOR)

For each timestep, the Poisson equation needs to be solved as a system of linear equations. A method for solving such a system is called successive over-relaxation which is a variant of the Gauss-Seidel method. The purpose of using this variant is that it will give faster convergence than pure Gauss-Seidel. The SOR method will converge with a number of iterations proportional to the unknowns $O(h)$, while Gauss-Seidel will converge with a number of iterations proportional to the square of unknowns $O(h^2)$.

The SOR method gives different weights to the Gauss-Seidel and the previous iterate based on value of omega. Omega has to be bigger than one to speed up the convergence, thus the name over-relaxation. The omega parameter can be found by trial and error, however for this task the omega values are linearly interpolated from previous results based on the grid size as suggested in [1].

2.8 Boundary Conditions

To solve the numerical model, a set of boundary condition is required. One might need different types of boundaries for different flow problems. The cavity box for instance, need to model a moving top lid as a boundary. Other problems might need internal geometries and external flows moving through the domain. The following boundary conditions might be applicable for this project.

- Solid walls/geometry which implies a no-slip condition
- Symmetry plane or surfaces with no adherence implying a free slip condition.
- Inlet and outlet conditions
- Periodic conditions

For setting the boundary conditions, two formulations can be used, the Dirichlet conditions and the Neumann conditions. The Dirichlet condition is categorised by setting a fixed value for the actual solution at the domain boundaries. The Neumann condition uses a derivative to set the conditions.

When setting the boundary condition one need to decide which discrete parameters to use for the boundary conditions. This is decided by which components has the highest derivative in the equation that is solved for. For the momentum equation the highest derivative is the viscous term and therefore the velocities are used as boundary conditions. However for the Poisson equation the pressure has the highest derivative, and therefore a boundary condition for the pressure is required.

For a solid surface with a no slip condition, the velocity in both the x and y direction is set to zero at the wall. Figure 4 demonstrates how this is done for a horizontal wall. They grey part of the figure represents solid geometry. Notice that all solid boundaries moves along the border of the MAC cells and never through the cell. For a horizontal solid boundary the x-velocity component $U_{i,j}$ is set to zero directly. For the y-velocity one can obtain zero velocity at the wall by setting the velocity inside the wall equal to the negative of the corresponding velocity outside the wall. This is demonstrated through the red lines in the figure. As mentioned in chapter 2.5 the ghost cells are used to establish boundary conditions at the edge. However this method can also be used to establish internal geometry within the domain.

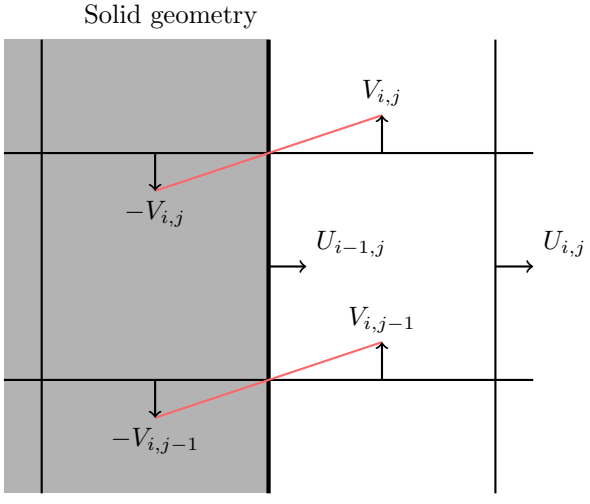


Figure 4: No-slip boundary conditions

Inlet and outlet boundary conditions is also needed for some of the flow problems. The ingoing flow is simply set equal to one, as the inlet velocity is represented trough the reference velocity as

explained in chapter 2.3. For the outgoing flow the ghost cell velocity-components are set equal to the corresponding velocity inside of the domain. Because the velocities on both sides of the domain border are equal, the gradient at the border is zero, and the outlet boundary condition corresponds to a Neumann condition.

Note that for transient problems an initial condition is also required for the velocity and pressure. For all the problems in this project the initial conditions are set to zero on the entire domain.

2.9 The Stability Criteria

It will be important to establish a stability criteria in order for the numerical model to converge. With a stable criteria, the model also becomes more user friendly and flexible. An important parameter that needs to be set for the stability is the timestep Δt. If the timestep is chosen to big, then the solution will become unstable. On the other hand, choosing a small timestep certainly will help the solution to become stable, but it can be also be impractical. While this would help to the convergence of the model, one need to consider the the total iteration time when dealing with numerical models. As such, a compromise is needed to establish a practical timestep.

For this numerical solver, three different stability criteria are used, they are the following

1. $\Delta t \leq Min\left[\frac{\Delta x}{|U_{i,j}^n|}, \frac{\Delta y}{|V_{i,j}^n|}\right]$
2. $\Delta t \leq \frac{Re}{2}\left(\frac{\Delta x^2 \Delta y^2}{\Delta x^2 + \Delta y^2}\right)$
3. $\Delta t \leq Min\left[\frac{2}{Re(U_{i,j}^n)^2}, \frac{2}{Re(V_{i,j}^n)^2}\right]$

Criteria 1 basically states that the fluid should not move through more than one cell in one timestep, because the numerical scheme assumes that flux only takes place between adjacent cells. The second criteria states that the viscous term in the momentum equation must be limited to one cell per time step. This comes from the fact that the viscous term depends on velocities one cell apart, but not further than that. If momentum was diffused further than that, the grid would display nonphysical behaviour as the diffusion taking place would depend on the fluid in that cell. The last criteria ensures that a numerical unstable advection equation becomes stabilized by positive diffusion.

3 Building the Program

In this chapter a brief description on the code, as well as an exploration on the different flow cases is given. While the driving algorithm was mostly based on the Matlab script sola.m in appendix D.1 other additional components was implemented. These includes flow visualization with GLview, support for transient visualization, calculations for pressure, streamlines and forces. Also boundary conditions for the additional flow cases needed to be implemented.

3.1 Flow Cases and Boundary Conditions

For the project three different flow cases was made. The cavity box which is also what the matlab script sola.m uses as example, flow around a square and flow around a vertical plate.

3.1.1 Cavity Box

The cavity box is a common test to use for benchmark tests in computational fluid dynamics. In this test, there are no fluids streaming into or out of the domain. The flow starts with a the velocity field being equal to zero. Then the top lid of the cavity box start moving to the right, illustrated in figure 5 . The speed of the top lid is governed by the reference velocity U, which is set through the Reynolds number. This can be done because the lid is the only externally moving part on the domain i.e. no inlet or outlet flow with its own velocity.

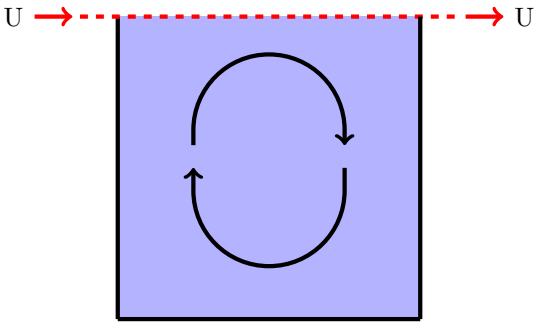


Figure 5: An illustration of the cavity box

Applying boundary conditions on this case is straight forward for the three stationary walls. A no-slip condition with zero velocity is applied on the walls as explained in chapter 2.8. For the moving top lid, the y-velocity V can be set to zero as no fluid leave the domain. The x-velocity U needs to be equal to one at the boundary. However this can not be done directly since the U component is not directly on the wall boundary. To achieve the correct velocity, the ghost cell velocity is first set equal to the negative of the corresponding border velocity such that the velocity is zero. Then the value of two is added such that the linearly interpolated velocity at the border becomes one.

3.1.2 Flow around a square

For the flow around a square, inlet and outlet conditions need to be specified. These are implemented just as explained in chapter 2.8. The flow comes in from the left side and goes out at the right side, as illustrated in figure 6. The internal geometry also needs to be specified as boundary conditions i.e. for every iteration of the Poisson equation the velocity is forced to be zero at the square boundary.

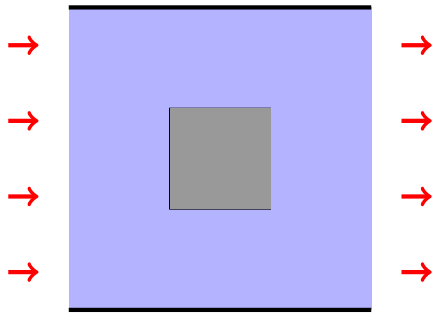


Figure 6: A square in a channel flow

3.1.3 Vertical Plate in a Channel Flow

This flow is quite similar to the square in a channel flow. The square is switched to a thin and longer vertical plate. The thickness of the plate is artificial and in reality it is infinitely thin, as the plate is set on the border between two MAC cells. This flow case will be used as comparison material for the flow around a square to check for characteristic differences. See figure 7.

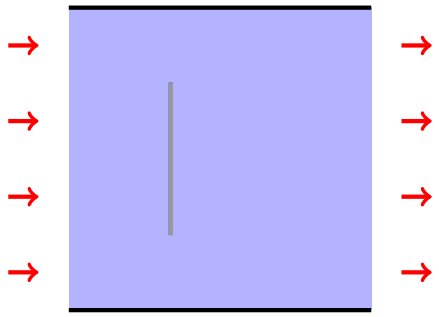


Figure 7: Vertical plate in a channel flow

3.2 Description of the Code

As mentioned, the driving algorithm for the project is mostly based on the Matlab script sola.m. The code was written in a procedural oriented programming style, in c++. The main algorithm from the Matlab script had to be translated to c++, and was also restructured into different subroutines. Since the program required a lot of additional code for visualization etc. it was more orderly to split the code into different files. Two flow charts for the code are given in Appendix A and Appendix B. A describes the "main.cpp" file where the setup and stability test is executed. The B chart is for the "calculateFlowField.cpp" file, where the actual calculations are executed.

3.2.1 Visualization

Unlike dedicated math programs such as Matlab, the c++ standard library does include a standard tool for graphing geometries and making plots. These has to be custom made for the application. There are many ways to do this. An elegant way could be to use a graphics API such as OpenGL and make a custom plotting tool tailored for this specific program. However that would be out of the scope for this project and require a lot more work to implement. Instead the proprietary software GLview inova has been used for this project, as the students have licensing for the program. GLview inova is a software for CAE (Computer Aided Engineering) that is used to display result for computational fluid dynamics and finite element analysis.

To use GLview for visualization, the code needs to produce its own files that GLview can read from and display results to the user. For this project the file format .vtf has been used. The

VTF format is GLview own file format and while it is an older format its more easy to use than the newer GLview format VFTx. For each flow case with a set of parameters, one VTF file was produced. The file contains a scalar plot for both the pressure field and the streamlines, as well as a vector field for the velocities. An example of visualization through GLview for the flow around a vertical plate is shown in figure 8. In this figure the pressure and velocity fields are displayed.

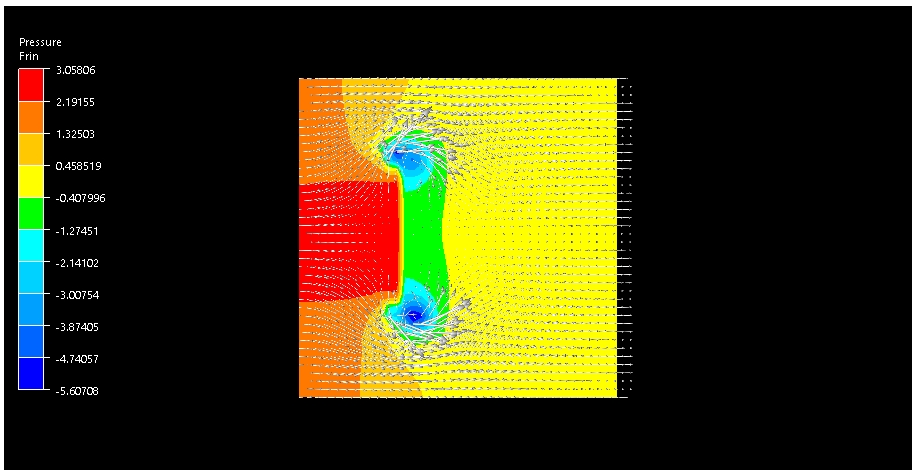


Figure 8: An illustrative example on a flow case using GLview

3.2.2 Transient Analysis

Another implementation that was done for the program was the ability to do a transient analysis on the development of the flow field. Being able to not only see the stationary result on the field, but actually seeing the development during the calculations can be a quite helpful tool to understand how the fluid reacts. It can also be useful to check for nonphysical behaviour during the transient development of the field.

The implementation was quite straight forward as the VTF file format had its own "step" convention that was supported in GLview to make animations. For a given timestep, one could simply append the results at that timestep to the VTF file, before moving on to the next timestep. However appending a result for each timestep was a bit excessive, and resulted in very large VTF files so the steps were often appended for every tenth or hundred timestep or so. A transient visualization for the velocity and pressure field is demonstrated below in figure 9 for the vertical plate. The images are taken from individual timesteps, and in this case illustrates how the vortexes behind the plate develops over time. The illustration is sampled with a timestep of $\Delta t = 4 \cdot 10^{-3}$ between each image.

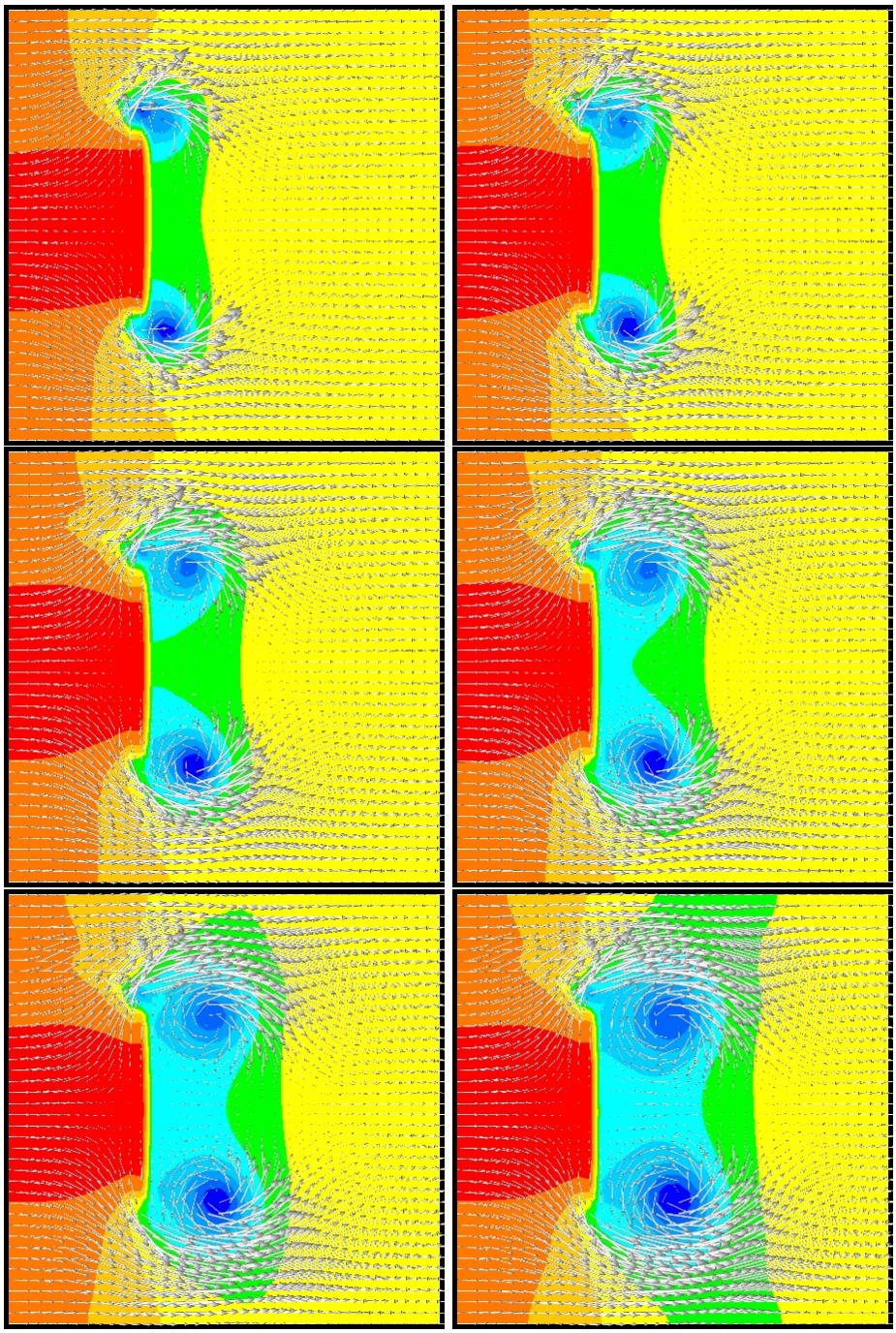


Figure 9: An illustrative example on the development of vortexes behind a vertical plate

3.2.3 Program Versions

Execution time has been one of the leading motivations during the development of the project. In the same way that one can use different parameters to experiment with flow situations and accuracy, different versions of the program has been developed to test the execution time of the program code. The versions has been restricted to two main version. One that uses multidimensional arrays, and one that uses single arrays. Note that the code in Appendix ... is the version that uses single arrays.

The program was initially developed with the use of multidimensional arrays. In C or C++ for a 2-dimensional array these are defined as an "array of arrays". What this means in practice is that the

array that contains the arrays, actually contains memory addresses to the different single arrays somewhere in the memory. The single arrays will correspond to the rows in the 2-dimensional matrix. Multidimensional arrays is often easier to deal with, as they have built in functionality to access the specific elements in the matrix. For example, if one wishes to access the element (i,j) in the multidimensional array "multi" one can just write "multi[i][j]". However because the different rows are located at different places in memory, a small penalty to the execution time is added when accessing an element because the compiler needs to locate the memory at different locations.

The other version that was built uses single arrays. A single array is essentially a vector that contains all its data sequentially in memory. These will be faster to access for the compiler since all the data is contained sequentially at the same place in memory. However since the array is only defined as a single vector, functionality has to be added to use the array as a matrix. The functionality was added through the use of defines in header.cpp which can be seen in Appendix C.1 in line 12 and 13. As an example if a matrix of size $N \cdot N$ is needed, one can simply define the single array as a vector of size $N \cdot N$ and use the functionality to access a specific element.

4 Results

For this section results from the project is displayed. As the project was based of the Matlab script "sola.m" in appendix D.1, the results is substantial for validating the code that was built.

4.1 Code Verification

If the code is to be used as a CFD solver, the code needs to be verified. Even though the code can display graphical results that may look authentic, a small error in the code can generate huge errors for the displayed results. However since the solving algorithm is directly based on sola.m, the script can also be used for validation purposes.

For the validation the default parameters was used as the test case. These can be seen in table 1.

Parameters	Value
Re	100
Δt	0.01
tmax	10
n	30
itmax	300
ϵ	10^{-6}
β	0.0542

Table 1: Test parameters for the default test case on the cavity box

The first indication of validation was that the Matlab script and the C++ code used exactly the same numbers of iterations, ending on 14 iterations for the last timestep. Also the graphical results seems to correspond well with each other. A comparison between the Matlab and the GLview plots for the velocity, stream function and pressure are given in the figures 10, 11 and12. All the comparisons are sampled at the last timestep.

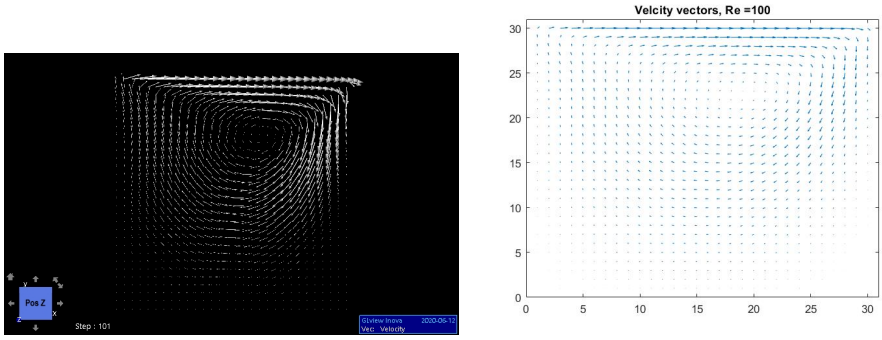


Figure 10: Comparison between the velocity vector field for the matlab script and the project code

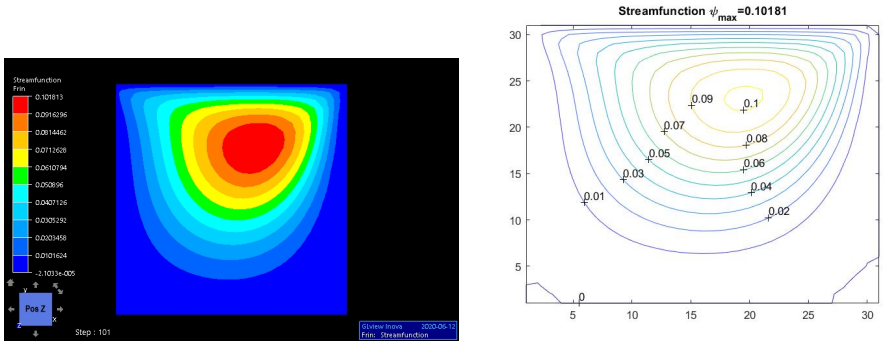


Figure 11: Comparison between the stream scalar field for the matlab script and the project code

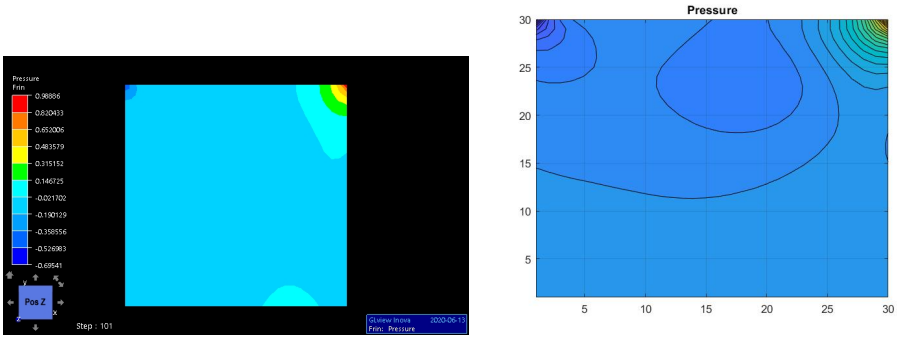


Figure 12: Comparison between the pressure scalar field for the matlab script and the project code

These comparison figures demonstrates that the program can generate and display results that correlates well with the algorithm that is used for sola.m. Now that the code has been validated, it can be used to generate results for other parameters or/and flow cases.

4.2 Test Case: The Cavity Box

Different parameters was used to experiment with the cavity box. The original parameters from table 1 was used as a reference. The test case with a lower Reynolds number can be seen in table 2. Also an experiment case with a much higher Reynolds number was conducted.

Parameters	Value
Re	40
Δt	0.001
tmax	10
n	50
itmax	2000
ϵ	10^{-6}
β	0.0542

Parameters	Value
Re	800
Δt	0.001
tmax	10
n	50
itmax	2000
ϵ	10^{-6}
β	0.0542

Table 2: Test parameters for the cavity box with a lower Reynolds number in the left table, and a higher in the right table

A transient analysis through GLview as demonstrated in section 3.2.2 was conducted on these two cases. For the case with a lower Reynolds number it was observed that already at time t=2, the flow had almost reached a steady state on the domain. This is shown in figure 13. The left image is for t=2, notice that the field is almost exactly the same as for t=10 in the right image.

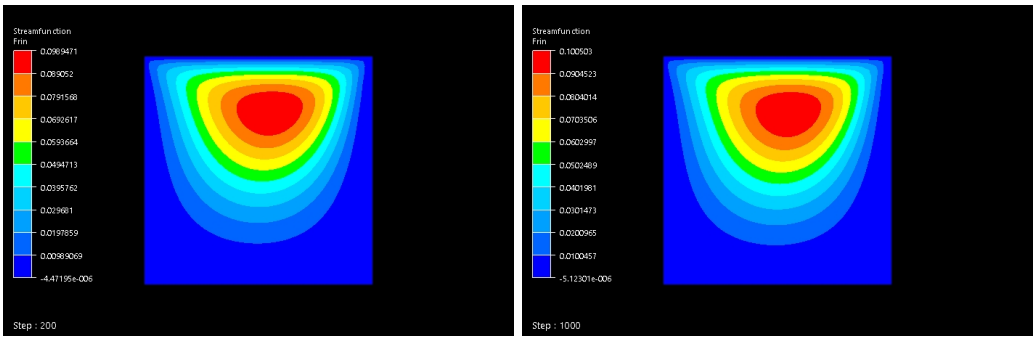


Figure 13: A comparison for the cavity box with a low Reynolds number. The left image for t = 2, and the right image for t = 10

For the flow with a higher Reynolds number, the development was quite different. At the timestep t = 2, the flow was far from finished developing. A similar comparison as for the low Reynolds number can be seen in figure ... With a transient analysis it was observed that even at the last timestep t = 10, the flow had still not reached steady state as it was still developing. Ideally tmax should be set higher for the flow case to ensure a solution closer to steady state.

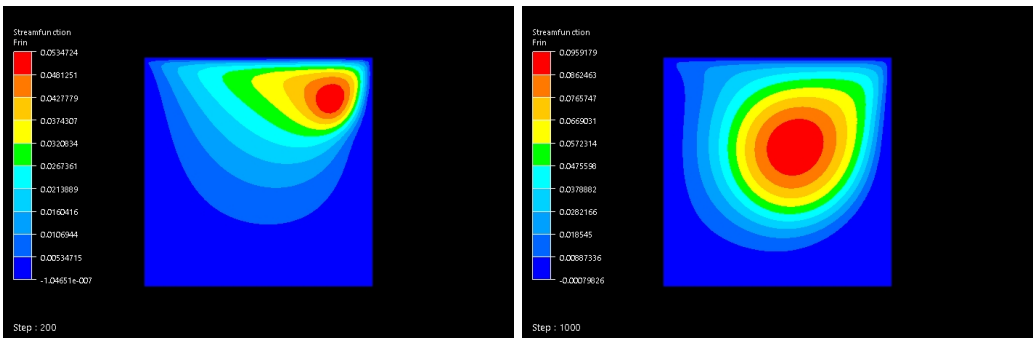


Figure 14: A comparison for the cavity box with a higher Reynolds number. The left image for t = 2, and the right image for t = 10

These results are as expected. The Reynolds number describes the ratio of inertial to viscous forces. When the Reynolds number is low, the viscous forces are large compared to the inertial forces. The large viscous forces will "slow down" the flow , such that the acceleration on the domain will disappear quicker.

4.3 Test Case: Flow around a Square

For the test case on the box in a flow, the parameters are seen in table 3. In figure 16 a comparison with these values at timesteps t = 0.1 and t = 3 are displayed.

Parameters	Value
Re	100
Δt	0.001
tmax	3
n	50
itmax	1000
ϵ	10^{-6}
β	0.0542

Table 3: Test parameters for the box in a flow

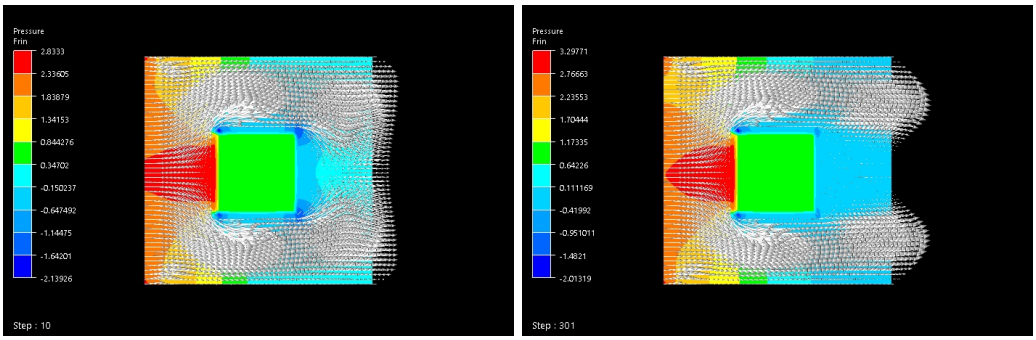


Figure 15: Pressure and velocity plot at t = 0.1 in the left figure and at t = 3 for the right figure

In figure 15 a plot for the drag and lift forces acting on the box are presented. The force plots was generated by using a simple Matlab script that can be seen in appendix D.2 Notice the drops in the beginning of the two. These drops are due to vortexes being present in the beginning of the analysis as shown in the left image of figure 16. These vortexes eventually disappear and the forces gain a stable value. Also notice the forces are in general much higher for the drag forces than for the lift forces, even though they act on the same amount of area. This is due to the pressure difference being much higher for the front and back of the box, than for the upper and lower part of the box.

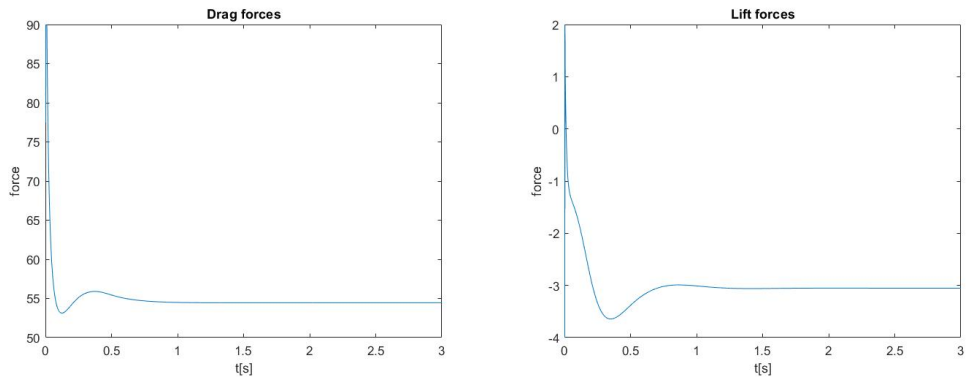


Figure 16: A transient force plot for the square in a flow. The left image displays the drag forces, while the right one displays the lift forces.

4.4 Test Case: Flow Around a Vertical Plate

Since the plate is defined as being infinitely thin, the lift force is zero because the pressure has zero area to work on. A plot for the force can be seen in figure 17. The values from table 3 that was used for the force plot on the square was also used in this analysis. Notice that the drop in drag forces are much bigger in the beginning for this flow case, than it was for the box. This is due to the vortexes being more dominant in the beginning of this case. A comparison at different timesteps for this flow case is shown in figure 18 for four different timesteps. Notice how the vortexes start at the plate and then moves forward and eventually fades away.

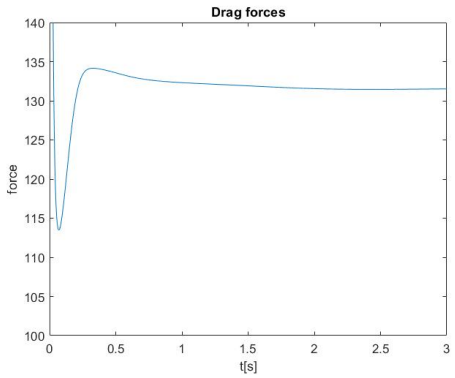


Figure 17: Force plot for the drag acting on the vertical plate.

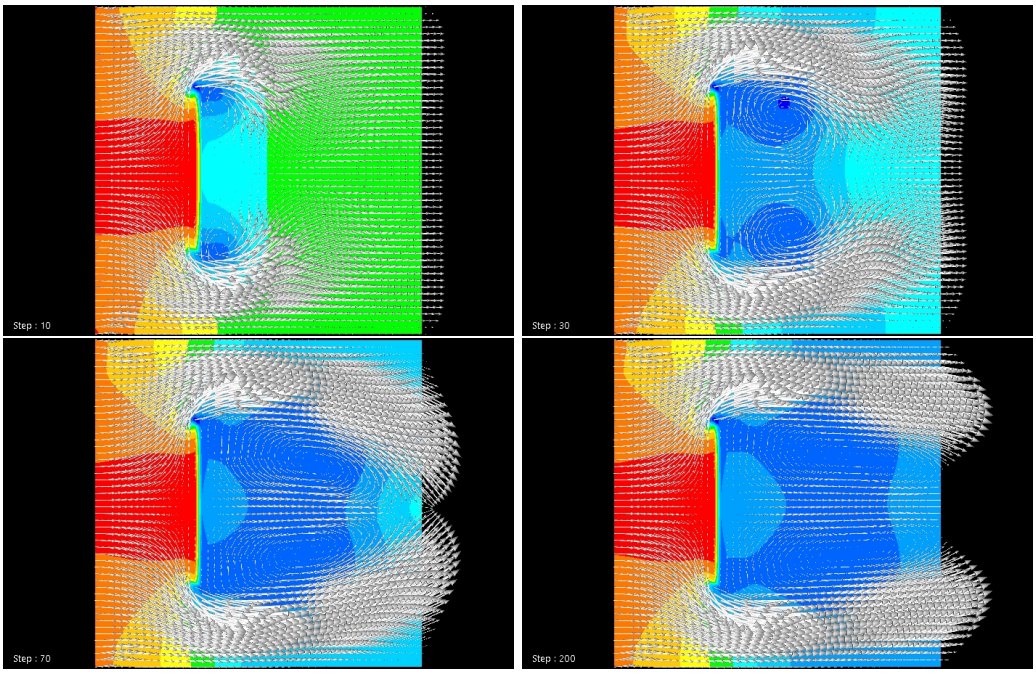


Figure 18: Vortexes acting on the backside of the vertical plate. Upper left: t = 0.1, Upper right: t = 0.3, Lower left: t = 0.7, Lower right: t = 2

4.5 Execution Time

In this section the results for the execution times of the different builds of the program is presented. The execution times are compared, both for the different builds, as well as for the Matlab script. The Matlab script and the C++ program are not directly comparable because of how different the two languages run. However it can still be interesting to compare the to languages. It gives a sense of how large the gain in execution time is when switching from Matlab which runs on a virtual machine, to C++ which generates platform specific machine code.

As comparison cases for the different execution time, the standard values from table 1 is used on the cavity box. Note that build times are excluded from these comparison cases because the comparisons are intended for a finished build of the project code. Only the compilation time is evaluated. Also to make the results as comparable as possible, all subroutines regarding writing to files and producing plots was excluded during the tests. The execution times was sampled directly before and after the main time loop. The results can be seen in table 4.

Versions	Single Arrays		2-Dimensional Arrays		Matlab
Builds	Debug	Release	Debug	Release	Standard
Test 1	1.95	1.37	2.08	1.49	2.84
Test 2	1.97	1.33	2.08	1.48	2.46
Test 3	1.98	1.33	2.20	1.33	2.46
Test 4	2.05	1.30	2.06	1.41	2.45
Test 5	1.98	1.29	2.05	1.40	2.54
Test 6	1.99	1.29	1.99	1.39	2.38
Average	1.99	1.32	2.08	1.42	2.52

Table 4: Execution times for the different builds of the project

The different build versions "Debug" and "Release" are the names of the default builds for Visual Studio. The release version is the regular build of the code that could be used for the finished program. The debug version have the purpose of being used while building the code. It has less optimization than the release version, and also includes debugging information. Of course these two builds are not standard and can be customised in whichever way the coder needs. The release version is certainly the most important for direct use of the program. However for the one building the code, it is important to have a short execution time. One might need to execute the code continuously while in debug mode to fix errors in the code, or add functionality.

Going from a multidimensional array to a single array gave an average increase of 7.04% in execution time for the release version. For the debug version the increase in execution time was at 4.33%. While the differences in execution time are not that significant since the total execution time is so short, it can give differences in minutes or even hours when the total execution time grows.

Also comparing with the Matlab execution time the difference is quite significant and displays some of the limitation to only using Matlab for numerical simulation. While Matlab certainly is useful for prototyping and testing, a solver based on low level programming will have significant benefits when it comes to the execution time.

5 Discussion

The code was validated in section 4.1 and in section 4.2, 4.3 and 4.3 it demonstrated that it could provide realistic result for different flow and parameters as well. That is not to say that the solver is perfect in its current state. While the cavity box handled different parameters well, the other flow cases required small timesteps with a large number of iterations to converge. It was certainly possible to acquire satisfying results but with small increases in Reynolds number or number of grids, the execution time became significantly larger.

The change from multidimensional arrays to single arrays proved to be an effective enchantment to the code. However other improvements could be made to the code to decrease the execution time. A simple improvement could be to define the time interval based on the size of the Reynolds number. As was seen in section 4.2, the time interval needed to reach steady state depended on the Reynolds number. While this would not decrease the computational time for higher Reynolds numbers, it could remove some unnecessary computations for cases with a lower Reynolds number.

Other improvements could also be done to optimize the algorithm even more. An effective enhancement could be to experiment with parallelization. For instance the tentative velocities can be calculated independent of each other as Reidar Kristoffersen states in [1]. Therefore all the operations calculating these velocities could be done simultaneously. The same goes for applying the boundary conditions, as these conditions are often dependant on values from a previous iteration. This could be achieved by for example using Cuda or similar software that enables use of the GPU to perform parallelizable numerical tasks.

Also several improvements could have been made to the user-friendliness of the program. For example, a simple graphical display could have been implemented into the program as stated in section 3.2.1. Being able to display the results directly in one application could certainly improve the work flow of the program. Other functionality could also been added such as a user interface to set the parameters. Having to change the parameters within the code, and then build the code for each small change adds a lot of unnecessary time when using the program.

6 Conclusion

The program displayed satisfying results and proved to be a good basis for a numerical solver of incompressible, laminar flow. It demonstrated that it was useful for different analysis on different flow cases and parameters. The code was validated which was the main goal of the project. Also insight and knowledge was acquired on how compiled program works and what affects their run time. It was concluded that while using single arrays for the calculation added a bit more complexity, the gain in execution time was ultimately worth it.

References

[1] R. Kristoffersen, *Demo lecture: Navier-stokes solver, sola.* DOI: https://www.youtube.com/watch?v=RSncu1io0VA&feature=youtu.be.

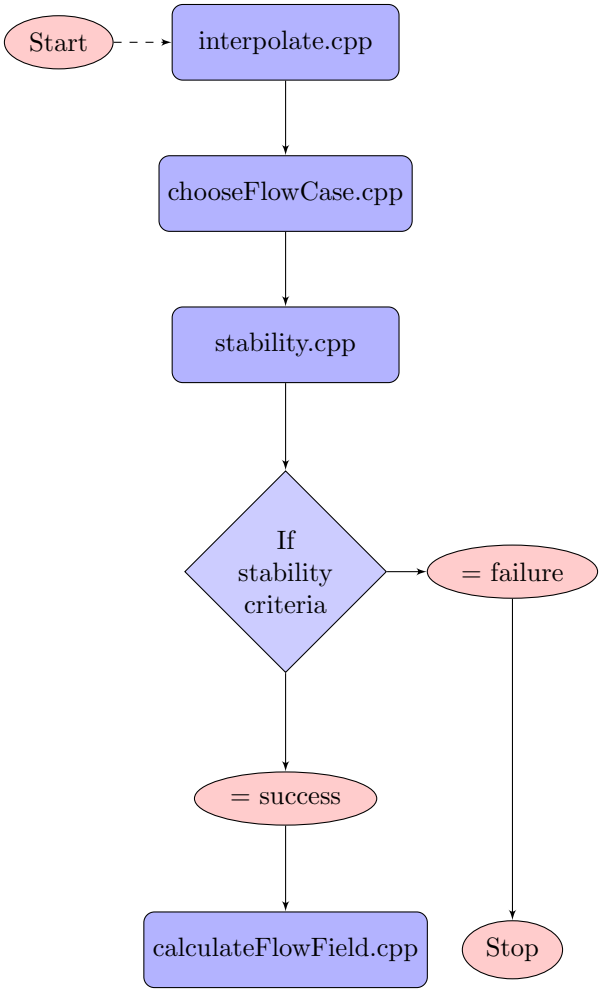
[2] R. Kristoffersen, *A navier-stokes solver using the multigrid method*, 1994.

[3] A. J. Chorin, "A numerical method for solving incompressible viscous flow problems," *Journal of Computational Physics 135*, vol. 135, pp. 118–125, 1997.

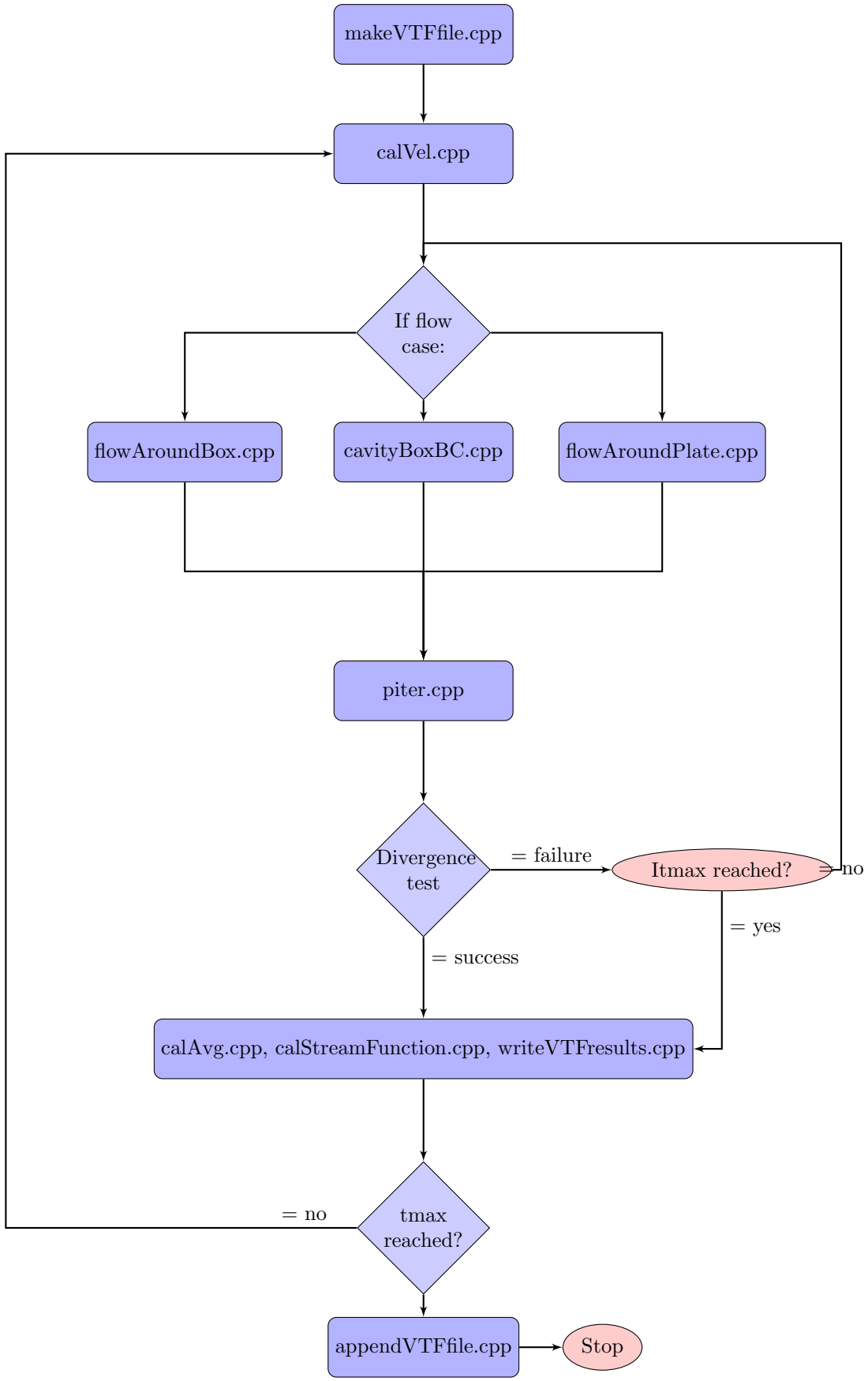
[4] J. Welch, F. H. Harlow, J. Shannon, and B. J. Daly, "The mac method a computing technique for solving viscous incompressible, transient fluid-flow problems involving free surfaces," 1965.

Appendix

A Flow Chart for main.cpp



B Flow Chart for calculateFlowField.cpp



C C++ Code

C.1 header.h

```
#pragma once
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <iomanip>
#include <time.h>
#include <algorithm>
#include <vector>

#define n 50

#define IX(x, y) ((x) + (y)*(n+2))
#define JX(x, y) ((x) + (y)*(n))

struct Field
{
    double u[(n+2)*(n+2)] = {0};
    double v[(n+2)*(n+2)] = {0};
    double p[(n+2)*(n+2)] = {0};

};

struct Field_Val
{
//#define IX(x, y) ((x) + (y)*(n))
    double u[n*n] = {0};
    double v[n*n] = {0};
    double p[n*n] = {0};
    double psi[(n+1)*(n+1)] = {0}; // stream function values
};

double interpolate();

bool stability(double h, double Re,double dt);

void calculateFlowField(double Re,double h, int tmax, double dt, Field& V,Field_Val
    &  U, int itmax, double beta, double epsilon, int choice);

void calVel(double h, double Re, double dt, Field &V);

void cavityBoxBC(Field &V);

void flowAroundSquareBC(Field &V);

void flowAroundVPlateBC(Field &V);

void piter(double h, double dt, double beta, double epsi, Field &V,int &iflag,
    double* div);

void calculateAverage(const Field& V, Field_Val& U);

void streamFunction(const Field& V, Field_Val& U, double h);

int choice();

void makeVTFfiles(std::ofstream &file, std::ofstream &pFile, std::ofstream &psiFile
    , std::ofstream &vFile);

void writeResults(Field_Val F, std::ofstream &file, std::ofstream &pFile, std::
    ofstream &psiFile, std::ofstream &vFile, int step);

void appendVTFsteps(std::ofstream &file,  std::ofstream &pFile, std::ofstream &
    psiFile, std::ofstream &vFile);

void calculateForce(int situation, const Field& V, std::ofstream &dFile, std::
    ofstream& lFile);
```

C.2 main.cpp

```
#include "header.h"

int main(int argc, char **argv)
{
    //constant variables
    const double epsilon = 1e-6;
    const double Re = 100;
    const int tmax = 3;
    const double dt = 0.001;
    const int itmax = 1000;
    const double h = 1.0/n;
    const double omega = interpolate();
    const double beta = omega*h*h/(4*dt);

    //chooses flow situation
    int ch = choice();

  Field F;

  Field_Val E;

    if (stability(h, Re, dt))
    {
      std::cout << "stability: success" << std::endl;
    calculateFlowField(Re, h, tmax, dt, F, E, itmax, beta, epsilon, ch);
    }
    else
    {
      std::cout << "stability: failure" << std::endl;
    }

    return 0;

}
```

C.3 calculateFlowField.cpp

```
#include "header.h"

using namespace std;

void calculateFlowField(double Re,double h, int tmax, double dt, Field& V,
    Field_Val& U,
 int itmax, double beta, double epsilon, int choice)
{
    int t, iflag, iter;

    int step = 1;
    double* div = new double(0.0);

    //VTF-Files
    std::ofstream file;
    std::ofstream fileResP;
    std::ofstream fileResPSI;
    std::ofstream fileResV;

    //Force-File
    std::ofstream fileDragForce;
    std::ofstream fileLiftForce;
    fileDragForce.open("drag.txt");
    fileLiftForce.open("lift.txt");
    if (!fileLiftForce || !fileDragForce)
        std::cout << "file for drag or lift not found";

    makeVTFfiles(file, fileResP, fileResPSI, fileResV);

    clock_t tStart = clock();

    for (t = 0; t <= tmax/dt; t++) //main time loop
    {
        calVel(h, Re, dt, V);

        for (iter = 1; iter <= itmax; iter++) //Applying boundary conditions for
    the velocities
        {
            if (choice == 1)
            {
                cavityBoxBC(V);
            }
            if (choice == 2)
            {
                flowAroundSquareBC(V);
            }
            if (choice == 3)
            {
                flowAroundVPlateBC(V);
            }

            piter(h, dt, beta, epsilon, V, iflag, div); //calculating new pressure

            if (iflag == 0) { break; }

        }
        if (iter >= itmax)
        {
            printf("Warning! Time t = %f, iter = %d ,div = %f\n" ,t*dt ,iter, *div)
     ;
        }
        else
        {
            printf("Time t = %f iter = %d \n",t*dt, iter);
        }


        if( t % 10 == 0)
        {
            calculateAverage(V, U);

```

```
            streamFunction(V, U, h);

            writeResults(U, file, fileResP, fileResPSI, fileResV, step);

            step++;
        }

        calculateForce(choice,V,fileDragForce, fileLiftForce);

    }

    printf("Time taken: %.2fs\n", (float)(clock() - tStart) / CLOCKS_PER_SEC);

    delete(div);

    appendVTFsteps(file, fileResP, fileResPSI, fileResV);

    fileDragForce.close();
    fileLiftForce.close();


}
```

C.4 calVel.cpp

```
#include "header.h"

void calVel( double h, double Re, double dt, Field& V)
{
  double fux, fuy, fvx, fvy, visu, visv;
    //Calculation of tentative velocities for the known pressure
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= n; j++)
        {
      fux = 1 / h * 0.25 * ((V.u[IX(i + 1,j)] + V.u[IX(i,j)]) * (V.u[IX(i + 1,j)] + V.u[IX(i,j)]) - (V.u[IX(i,j)] + V.u[IX(i - 1,j)]) * (V.u[IX(i,j)] + V.u[IX(i - 1,j)]));
      fuy = 1 / h * 0.25 * ((V.v[IX(i + 1,j)] + V.v[IX(i,j)]) * (V.u[IX(i,j + 1)] + V.u[IX(i,j)]) - (V.v[IX(i,j - 1)] + V.v[IX(i + 1,j - 1)]) * (V.u[IX(i,j - 1)] + V.u[IX(i,j)]));
      fvx = 1 / h * 0.25 * ((V.u[IX(i,j + 1)] + V.u[IX(i,j)]) * (V.v[IX(i + 1,j)] + V.v[IX(i,j)]) - (V.u[IX(i - 1,j)] + V.u[IX(i - 1,j + 1)]) * (V.v[IX(i,j)] + V.v[IX(i - 1,j)]));
      fvy = 1 / h * 0.25 * ((V.v[IX(i,j + 1)] + V.v[IX(i,j)]) * (V.v[IX(i,j + 1)] + V.v[IX(i,j)]) - (V.v[IX(i,j)] + V.v[IX(i,j - 1)]) * (V.v[IX(i,j)] + V.v[IX(i,j - 1)]));

      visu = (V.u[IX(i + 1,j)] + V.u[IX(i - 1,j)] + V.u[IX(i,j + 1)] + V.u[IX(i,j - 1)] - 4.0 * V.u[IX(i,j)]) / (Re * h * h);
      visv = (V.v[IX(i + 1,j)] + V.v[IX(i - 1,j)] + V.v[IX(i,j + 1)] + V.v[IX(i,j - 1)] - 4.0 * V.v[IX(i,j)]) / (Re * h * h);

      V.u[IX(i,j)] = V.u[IX(i,j)] + dt * ((V.p[IX(i,j)] - V.p[IX(i + 1,j)]) / h - fux - fuy + visu);
      V.v[IX(i,j)] = V.v[IX(i,j)] + dt * ((V.p[IX(i,j)] - V.p[IX(i,j + 1)]) / h - fvx - fvy + visv);

    }
    }
}
```

C.5 piter.cpp

```
#include "header.h"

void piter(double h, double dt,double beta,
   double epsi, Field &V, int &iflag, double* div)
{
    double delp;
    iflag = 0;

    for (int j = 1; j <= n; j++)
    {
        for (int i = 1; i <= n; i++)
        {
            *div = (V.u[IX(i,j)]-V.u[IX(i-1,j)])/h + (V.v[IX(i,j)]-V.v[IX(i,j-1)])/
    h;

            if (std::fabs(*div) >= epsi)
            {
                iflag = 1;
            }
            delp = -beta*(*div);
            V.p[IX(i,j)] = V.p[IX(i,j)] + delp;
            V.u[IX(i,j)] = V.u[IX(i,j)] + delp*dt/h;
            V.u[IX(i-1,j)] = V.u[IX(i-1,j)] - delp*dt/h;
            V.v[IX(i,j)] = V.v[IX(i,j)] + delp*dt/h;
            V.v[IX(i,j-1)] = V.v[IX(i,j-1)] - delp*dt/h;

        }
    }
}
```

C.6 calAvg.cpp

```
#include "header.h"

void calculateAverage(const Field& V, Field_Val &U)
{
for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
        U.u[JX(j,i)] = (V.u[IX(i,j+1)]+V.u[IX(i+1,j+1)])/2;
        U.v[JX(j,i)] = (V.v[IX(i+1,j)]+V.v[IX(i+1,j+1)])/2;
        U.p[JX(j,i)] = V.p[IX(i+1,j+1)];
        }
    }
}
```

C.7 calStreamFunction.cpp

```
#include "header.h"

void streamFunction(const Field& V, Field_Val &U, double h)
{
    double psi [(n+1)*(n+1)] = {0};

    for(int i = 1;i < n+1; i++)
    {
        psi[JX(i,0)] = psi[JX(i-1,0)] - V.v[IX(i,0)]*h;
    }

    for(int i = 0; i < n+1; i++)
    {
        for(int j= 1; j < n+1; j++)
        {
            psi[JX(i,j)] = psi[JX(i,j-1)]+V.u[IX(i,j)]*h;
        }
    }

    for (int i = 0; i <= n ; i++)
    {
        for (int j = 0 ; j <= n; j++ )
        {
            U.psi [JX(i,j)] = -psi[JX(j,i)] ;
        }
    }
}
```

C.8 calForce.cpp

```
#include "header.h"

void calculateForce(int choice,const Field& V, std::ofstream &dFile, std::ofstream&
     lFile)
{
    double forceX = 0;
    double forceY = 0;
    if (choice == 2)
    {
        //box
        for(int i = n/3; i <= 2*n/3; i++)
        {
            forceX = forceX + V.p[IX(n/3-1,i)] - V.p[IX(n*2/3+1,i)];
            forceY = forceY + V.p[IX(i, 2 * n / 3 + 1)] - V.p[IX(i, n / 3 - 1)];
        }
    }

    if (choice == 3)
    {
       //plate
       for (int i = n/4; i <= n*3/4; i++)
       {
           forceX = forceX + V.p[IX(n/3, i)] - V.p[IX(n/3+1, i)];
       }
    }


    dFile << forceX << "\n";
    lFile << forceY << "\n";

}
```

C.9 cavityBoxBC.cpp

```
#include "header.h"

void cavityBoxBC(Field &V)
{
    for (int j = 0;j <= n+1;j++) //Reduced to one loop to reduce computational time
    {
        //Applying boundaries    to the vertical walls
        V.u[IX(0,j)] = 0.0;
        V.v[IX(0,j)] = -V.v[IX(1,j)];
        V.u[IX(n,j)] = 0.0;
        V.v[IX(n+1,j)] = -V.v[IX(n,j)];
    }
    for (int i = 0; i <= n+1; i++)
    {
        //Applying boundaries to the horizontal walls
        V.v[IX(i,n)] = 0.0;
        V.v[IX(i,0)] = 0.0;
        V.u[IX(i,n+1)] = -V.u[IX(i,n)]+2.0;
        V.u[IX(i,0)] = -V.u[IX(i,1)];
    }
}
```

C.10 flowAroundSquareBC.cpp

```
#include "header.h"

void flowAroundSquareBC(Field &V)
{
    for (int j = 0; j <= n+1; j++)
    {
        //Ingoing flow
        V.v[IX(0,j)] = 0.0;
        V.u[IX(0,j)] = 1.0;

        //Outgoing flow
        V.u[IX(n+1,j)] = V.u[IX(n,j)];
        V.v[IX(n+1,j)] = -V.v[IX(n,j)];

        if (j >= n/3 && j <= n*2/3) //Set all boundaries for the box to zero
        {
            V.u[IX(n/3,j)] = 0.0;
            V.v[IX(n/3,j)] = -V.v[IX(n/3+1,j)];
            V.u[IX(n*2/3,j)] = 0.0;
            V.v[IX(n*2/3+1,j)] = -V.v[IX(n*2/3,j)];
            V.u[IX(j,n/3)] = -V.u[IX(j,n/3+1)];
            V.v[IX(j,n/3)] = 0.0;
            V.u[IX(j,n*2/3+1)] = -V.u[IX(j,n/3)];
            V.v[IX(j,n*2/3)] = 0.0;
        }

   //     Upper & lower part
        V.u[IX(j,0)] = -V.u[IX(j,1)];
        V.v[IX(j,0)] = 0.0;
        V.u[IX(j,n+1)] = -V.u[IX(j,n)];
        V.v[IX(j,n)] = 0.0;
    }
}
```

C.11 flowAroundVplateBC.cpp

```
#include "header.h"

void flowAroundVPlateBC(Field &V)
{
    for(int j = 0; j <= n+1; j++)
    {
        //Ingoing flow
        V.v[IX(0,j)] = 0.0;
        V.u[IX(0,j)] = 1.0;

        //Outgoing flow
        V.u[IX(n+1,j)] = V.u[IX(n,j)];
        V.v[IX(n+1,j)] = V.v[IX(n+1,j)];

   //     Setting the value zero at plate
        if(j > n*1/4 && j <= n*3/4 )
        {
        V.u[IX(n/3,j)] = 0.0;
        V.v[IX(n/3,j)] = -V.v[IX(n/3+1,j)];
        }

   //     Upper & lower part
        V.u[IX(j,0)] = -V.u[IX(j,1)];
        V.v[IX(j,0)] = 0.0;
        V.u[IX(j,n+1)] = -V.u[IX(j,n)];
        V.v[IX(j,n)] = 0.0;
    }
}
```

C.12 chooseFlowCase.cpp

```
#include "header.h"

using namespace std;

int choice()
{
    int choice = 0;
    cout << "Choose one of the following situations: " << endl;
    cout << "Type 1 for Cavity box" << endl;
    cout << "Type 2 for flow over box" << endl;
    cout << "Type 3 for flow over plate" << endl;
    cin >> choice;
    return choice;
}
```

C.13 interpolate.cpp

```
#include "header.h"

double interpolate()
{
    //Linear interpolation to find the omega parameter

    const int size = 9;
    int nn[9] = {0, 5, 10, 20, 30, 40, 60, 100, 500};
    double oo[9] = {1.7, 1.78, 1.86, 1.92, 1.95, 1.96, 1.97, 1.98, 1.99};

    double omega;
    //returns the closest previous position's
    int pos = static_cast<int>(std::upper_bound(nn, nn + size, n) - nn-1);
    std::cout << "position: " << pos  <<std::endl;
      //linear interpolation to obtain omega

    omega = oo[pos+1] +  static_cast<int>((double)n-nn[pos+1])*(oo[pos]-oo[pos+1])
    /((double)nn[pos]-(double)nn[pos+1]);
    std::cout << "Omega: " << omega  <<std::endl;

    return omega;
}
```

C.14 stability.cpp

```cpp
#include "header.h"

using namespace std;

bool stability(double h, double Re, double dt)
{
    vector<double> a = { h, Re * h * h / 4, 2 / Re };
    double min = a[0];
    if (min > a[1])
        min = a[1];
    if (min > a[2])
        min = a[2];

    if(dt > min)
    {
        cout << "Warning! dt should be less than "<< min << '/n';
        return 0;
    }
    return 1;

}
```

C.15 makeVTFfile.cpp

```
#include "header.h"

void makeVTFfiles(std::ofstream &file, std::ofstream &pFile, std::ofstream &psiFile
    , std::ofstream &vFile)
{
    file.open("glview.vtf");
    pFile.open("results_pressure.txt");
    psiFile.open("results_streamfunction.txt");
    vFile.open("results_velocity.txt");

      if (!file.is_open() ||!pFile.is_open() ||!psiFile.is_open() ||!vFile.is_open
    ())
    {
        std::cout << "file not found";
    }

    //adding the acii header
      file << "*VTF-1.00\n\n!Setup of nodes \n*NODES 1\n%NO_ID\n";

    //setting up all the nodes, without specifying the nodes ID becomes
    sequentially numbered
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= n; j++)
        {
        file  << j<< ". "<< i << ". " << 0.0 << ".\n";
        }
    }

   file << "\n*ELEMENTS 1\n%NODES #1\n%QUADS\n";
   for (int i = 0; i < n-1; i++)
   {
        for(int j = 1; j < n; j++)
        file << i*n + j << " "<< i*n + (j+1) << " " << (i+1)*(n) + (j+1) << " " <<
    (i+1)*n + j << "\n";
   }

    //assemble the geometry<< " " << i*(n+1)+j
    file << "\n\n*GLVIEWGEOMETRY 1\n%ELEMENTS\n1\n";

    pFile << "\n*GLVIEWSCALAR 1\n%NAME \"Pressure\"";

    psiFile << "\n*GLVIEWSCALAR 2\n%NAME \"Streamfunction\"";

    vFile << "\n*GLVIEWVECTOR 1\n%NAME \"Velocity\"";

}
```

C.16 writeVTFresults.cpp

```
#include "header.h"

void writeResults(Field_Val F, std::ofstream &file, std::ofstream &pFile, std::
    ofstream &psiFile, std::ofstream &vFile, int step)
{
    file << "\n*RESULTS "<< step*3-2 <<"\n%DIMENSION 1\n%PER_NODE  #1\n";
    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < n; j++)
        {
            file << F.p[JX(i,j)] << "\n";
        }
    }

    file << "\n*RESULTS "<< step*3-1 <<"\n%DIMENSION 1\n%PER_NODE  #1\n";
    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < n; j++)
        {
            file << F.psi[JX(i+1,j+1)] << "\n";
        }
    }

    file << "\n*RESULTS "<< step*3 <<"\n%DIMENSION 3\n%PER_NODE #1\n";
    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < n; j++)
        {
            file << F.u[JX(i,j)] << " " << F.v[JX(i,j)] <<" 0.0\n";
        }
    }

    pFile << "\n%STEP "<< step << "\n" << step*3-2;

    psiFile << "\n%STEP "<< step<<"\n"<< step*3-1;

    vFile << "\n%STEP "<< step<<"\n"<< step*3;
}
```

C.17 appendVTFfile.cpp

```
#include "header.h"

void appendVTFsteps(std::ofstream& file, std::ofstream& pFile, std::ofstream&
    psiFile, std::ofstream& vFile)
{
    //Purpose: Append the step blocks at the end of the VTF file
    pFile.close();
    psiFile.close();
    vFile.close();

    std::ifstream iP("results_pressure.txt");
    std::ifstream iPSI("results_streamfunction.txt");
    std::ifstream iV("results_velocity.txt");

    if (!iP.is_open() || !iPSI.is_open() || !iV.is_open())
    {
        std::cout << "file not found";
    }
    else
    {
        file << iP.rdbuf();
        file << iPSI.rdbuf();
        file << iV.rdbuf();
    }
    iP.close();
    iPSI.close();
    iV.close();
    file.close();
}
```

D Matlab Code

D.1 sola.m

```
% SOLA - MATLAB

clear all
close all
clc
n=30;
epsi=1e-6;
nn=[0   5    10   20   30   40   60   100  500];
oo=[1.7 1.78 1.86 1.92 1.95 1.96 1.97 1.98 1.99];
omega=interp1(nn,oo,n);               % Interpolating a reasonable value
Re=100;
tmax=10;
dt=0.01;
itmax=300;
h=1/n;
beta=omega*h^2/(4*dt);
u=zeros(n+2,n+2);v=u;p=u;

if dt>min([h,Re*h^2/4,2/Re])
    disp(['Warning! dt should be less than ',num2str(min([h,Re*h^2/4,2/Re]))])
    pause
end

for t=0:dt:tmax                 % Main loop
    for i=2:n+1                 % CalVel, calculation of velocities
        for j=2:n+1
            fux=((u(i,j)+u(i+1,j))^2-(u(i-1,j)+u(i,j))^2)*0.25/h;
            fuy=((v(i,j)+v(i+1,j))*(u(i,j)+u(i,j+1))-(v(i,j-1)+v(i+1,j-1))*(u(i,j
    -1)+u(i,j)))*0.25/h;
            fvx=((u(i,j)+u(i,j+1))*(v(i,j)+v(i+1,j))-(u(i-1,j)+u(i-1,j+1))*(v(i-1,j
    )+v(i,j)))*0.25/h;
            fvy=((v(i,j)+v(i,j+1))^2-(v(i,j-1)+v(i,j))^2)*0.25/h;
            visu=(u(i+1,j)+u(i-1,j)+u(i,j+1)+u(i,j-1)-4.0*u(i,j))/(Re*h^2);
            visv=(v(i+1,j)+v(i-1,j)+v(i,j+1)+v(i,j-1)-4.0*v(i,j))/(Re*h^2);
            u(i,j)=u(i,j)+dt*((p(i,j)-p(i+1,j))/h-fux-fuy+visu);
            v(i,j)=v(i,j)+dt*((p(i,j)-p(i,j+1))/h-fvx-fvy+visv);
        end
    end
    for iter=1:itmax            % BcVel, Boundary conditions for the velocities
        for j=1:n+2
            u(1,j)=0.0;
            v(1,j)=-v(2,j);
            u(n+1,j)=0.0;
            v(n+2,j)=-v(n+1,j);
        end
        for i=1:n+2
            v(i,n+1)=0.0;
            v(i,1)=0.0;
            u(i,n+2)=-u(i,n+1)+2.0;
            u(i,1)=-u(i,2);
        end
        iflag=0;                % Piter, Pressure iterations
        for j=2:n+1
            for i=2:n+1
                div=(u(i,j)-u(i-1,j))/h+(v(i,j)-v(i,j-1))/h;
                if (abs(div)>=epsi),iflag=1;end
                delp=-beta*div;
                p(i,j)  =p(i,j)  +delp;
                u(i,j)  =u(i,j)  +delp*dt/h;
                u(i-1,j)=u(i-1,j)-delp*dt/h;
                v(i,j)  =v(i,j)  +delp*dt/h;
                v(i,j-1)=v(i,j-1)-delp*dt/h;
            end
        end
        if(iflag==0)break,end
    end
    if iter>=itmax
       disp(['Warning! Time t= ',num2str(t),' iter= ',int2str(iter),' div= ',
```

```
    num2str(div)])
    else
        disp(['Time t= ',num2str(t),' iter= ',int2str(iter)])
    end
end

% Graphic display:

U=zeros(n);
V=U;
P=U;
for i=1:n
    for j=1:n
        U(j,i)=(u(i,j+1)+u(i+1,j+1))/2;
        V(j,i)=(v(i+1,j)+v(i+1,j+1))/2;
        P(j,i)=p(i+1,j+1);
    end
end
figure(1)
quiver(U,V);
axis([0 n+1 0 n+1]);
title(['Velcity vectors, Re =',num2str(Re)])
figure(2)
contourf(P,30)
title('Pressure')
grid on

figure(3)
psi=zeros(n+1);
for i=2:n+1
    psi(i,1)=psi(i-1,1)-v(i,1)*h;
end
for i=1:n+1
    for j=2:n+1
        psi(i,j)=psi(i,j-1)+u(i,j)*h;
    end
end
psi=-psi';
[C,H]=contour(psi);
clabel(C)
title(['Streamfunction \psi_{max}=',num2str(max(max(abs(psi))))])
```

D.2 plotLiftAndDrag.m

```
clear all
close all
clc

drag = readmatrix('drag.txt');
lift = readmatrix('lift.txt');

td = linspace(0,3,length(drag));
tl = linspace(0,3,length(lift));

figure(1)
plot(td,drag);
ylim([100,140]);
xlabel('t[s]');
ylabel('force');
title('Drag forces');

figure(2)
plot(tl,lift);
ylim([-4,2]);

xlabel('t[s]');
ylabel('force');
title('Lift forces');
```