Study for the computational resolution of potential flow and the incompressible Navier-stokes equations.

Contents:
REPORT

Bachelor's thesis

Lluís Montilla Rodríguez

Supervisor: Carlos David Pérez Segarra — Delivery date: 10/06/2018

GREVA - ESEIAAT
Universitat Politècnica de Catalunya (UPC)

Contents

List of Figures

List of Tables

ACKNOWLEDGEMENTS

First, I would like to thank my thesis supervisor Carlos David Pérez Segarra for all the help and support provided as well as the access he granted me to all the lectures and notes on the subject without all of which I could not have possibly finished this project.

I would also like to thank the CTTC group for the data provided to validate the code.

Last but not least, I would like to thank my family for all the support they gave me.

NOMENCLATURE

$\dot{m}$	Mass flow
$\dot{Q}$	Heat transfer rate
$\dot{q}$	Heat transfer rate per unit of mass
Γ	Circulation
λ	bulk viscosity coefficient
μ	Molecular viscosity coefficient
$\overline{c_p}$	Specific heat capacity at constant pressure
$\overrightarrow{a}$	Acceleration
$\overrightarrow{f}$	Body forces per unit of mass
$\overrightarrow{v}$	Velocity
ϕ	Velocity potential
Ψ	Stream function
ρ	Density
τ	Viscous stress
CFD	Computational fluid dynamics
DNS	Direct numerical simulation
e	Internal energy
k	Thermal conductivity
LES	Large eddy simulation
m	Mass
p	Pressure
$RANS$	Reynolds-averaged Navier-Stokes

Re	Reynolds number
S	Surface
T	Temperature
t	Time
$TDMA$	Tri-Diagonal Matrix Algorithm
V	Volume

Chapter 1

INTRODUCTION

Contents

1.1 Abstract

The aim of this project is to develop a CFD software using C++ which must be capable of solving the incompressible Navier-Stokes equations. The approach taken in its development is didactic, meaning that it is out of this project's scope to make any important innovation but rather to learn the inner workings of a CFD code in order to gain insight on the matter.

This project has been developed in steps, starting with the basic solving algorithm and continuing by adding complexity on top of that, hence allowing to solve the errors as they appear, while the program is still simple.

As the project's approach is didactic, every bit of the coding is original and has been tailored to work in conjunction with every subroutine included.

The end result will be compared to experimental results or results obtained from other codes which have been validated in order to assess the credibility of the results given by this software.

1.2 Scope

The contents of this project are the following:

- To develop a solving algorithm based on a line-by-line solver.
- To develop a blocking-off processor in order to introduce solids inside the mesh.
- To fully implement the incompressible Navier-Stokes equations in order to compute all fluid properties.

In this project, the following aspects are not included:

- Development of a graphic user interface (GUI).
- Development of a graphical processor to show the results. The software *GNUplot* will be used for that matter instead.

1.3 Requirements

Due to this project being didactic, all requirements are self-imposed and they are the following:

- Must give results akin to experimental results
- Must use original coding
- Must be done in C++

Chapter 2

STATE OF THE ART

Contents

2.1 Introduction

In this chapter, the current state of the CFD field development will be explained as well as the different approaches it offers. The CFD field can be split into three main methodologies: RANS, LES and DNS.

All methods are a way of computing the Navier-Stokes equations through numerical methods but they differ in both, precision and computing time.

DNS is the most precise amongst them but the computational power and time needed to perform an analysis is immense and thus is usually used only in small scale analysis of simple cases.

RANS is the polar opposite, it is the least precise but it allows for complex simulations that can be repeated due to its low computational time.

LES is a hybrid between them and it offers a compromise between precision and computational time, hence it is the field which is developing the most thanks to the current improvements in computer science.

All those methods and the current advancements will be explained with more detail in the following sections.

2.2 DNS

As its name implies, DNS directly solves the equation numerically. This allows for a high precision but at a high computing cost.

Although it might seem like a simple idea, the implementation is more complex than it appears. To begin with, the equations governing the flow, the Navier-Stokes equations, have never been solved analytically. Moreover, they are linear second order partial differential equations that must be turned into discrete equations in order for the computer to handle them. This leads to errors than can only be reduced by making a denser mesh, which increases computing time.

The main advantage to this approach is that it is only limited by how accurate the physical model used is and how powerful the computer being used is[1].

Keeping in mind that nowadays, the use of supercomputers is relatively common in research, it is not surprising that some results using DNS are considered valid as experimental results. This is possible thanks to the fact that DNS does not make any simplification in the physics model besides discretizing the equations, which can be solved with a finer mesh. This means that as long as the mesh is fine enough, turbulence can be perfectly computed without the use of any model[2].

2.2.1 Mathematical formulation

The equations used in DNS are the full Navier-Stokes equations, they are usually expressed in differential form as they need to be discretized in order to allow a numerical solution. The Navier-Stokes equations are based on three basic principles:

- Conservation of mass
- Newton's second law
- Conservation of energy

In order to find an adequate expression which describes the motion of the fluid, two approaches exist. One is to consider a finite control volume V defined by a control surface S. This volume will be fixed in space and thus the flow will be moving through it. This yields integral equations. The other option is to consider an infinitesimal fluid element which is big enough so that the inside is still a continuous medium. This fluid element will be moving through space and thus its characteristics will be changing in time and spaces as it moves through the domain.

Both approaches give the same result, however, the forms of the resulting expressions differ. While this might not have any effect for an analytical result, it does make a difference depending on the CFD application they are being used in.

Mass conservation

Also known as the continuity equation, the mass conservation equation states that mass in neither created nor destroyed. This implies that the mass entering a system must equal that which is exiting it. With that in mind, this equation is easier to understand using a finite control volume.

Inside an arbitrary control volume, the rate of decrease in the mass inside of it is equal to the net mass flow on its surfaces. the latter can be expressed as follows:

$$\dot{m} = \oiint_S \rho \overrightarrow{v} \overrightarrow{dS} \tag{2.1}$$

Keep in mind that due to convention, $\overrightarrow{dS}$ always points outwards making an inflow of mass a negative value.

Then, the mass change inside the control volume can be expressed in the following manner:

$$\frac{\partial m}{\partial t} = -\frac{\partial}{\partial t} \iiint_V \rho dV \tag{2.2}$$

Note that the rate of decrease in mass is being considered, hence the negative sign. This is done for coherence with the sign of the mass flow.

As mass cannot be created nor destroyed, the mass change inside the volume must be equal to the mass flow. Thus, equaling equations 2.1 and 2.2:

$$\frac{\partial}{\partial t} \iiint_V \rho dV + \oiint_S \rho \overrightarrow{v} \overrightarrow{dS} = 0 \tag{2.3}$$

Because the integration limits are constant, the time derivative can be entered into the integral; and using the divergence theorem, the surface integral can be expressed as a volume integral:

$$\iiint_V \left[\frac{\partial \rho}{\partial t} + \nabla \cdot \left(\rho \overrightarrow{v} \right) \right] dV \tag{2.4}$$

As the control volume has been chosen arbitrarily, for the integral to be zero, the terms inside of it must be zero at every point, thus:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \left(\rho \overrightarrow{v} \right) = 0 \tag{2.5}$$

At the beginning of this section, it is stated that equation 2.5 might take a different form if the derivation was to be done using an infinitesimal fluid element, it is possible to transform equation 2.5 into the other form using the following relation:

$$\nabla \cdot \left(\rho \overrightarrow{v} \right) \equiv \rho \nabla \cdot \overrightarrow{v} + \overrightarrow{v} \cdot \nabla \rho \tag{2.6}$$

The relation 2.6 can be substituted in equation 2.5:

$$\frac{\partial \rho}{\partial t} + \overrightarrow{v} \cdot \nabla \rho + \rho \nabla \cdot \overrightarrow{v} = 0 \tag{2.7}$$

Note that the first two terms in equation 2.7 correspond to the definition of the substantial derivative:

$$\frac{D\rho}{Dt} + \rho\nabla \cdot \overrightarrow{v} = 0 \tag{2.8}$$

Equations 2.5 and 2.8 are equivalent but depending on the specific CFD application one may be more useful than the other.

For a more detailed derivation of those equations, check [3].

Momentum equations

In order to obtain the momentum equations, Newton's second law applied to a moving infinitesimal fluid element is used. As it is a vector equation, it is more convenient to split it into its scalar components. For instance the x component:

$$F_x = ma_x \tag{2.9}$$

Where F_x represents the net force in the x direction. It is possible to split F_x into to sources of force, body forces and surface forces. The expression for body forces is the following:

$$F_x^{body} = \rho f_x dxdydz \tag{2.10}$$

As for surface forces, they correspond to pressure and viscous stresses. They are applied to all faces of the infinitesimal fluid element chosen and thus, the net forces corresponds to the difference between the two opposing faces of this element. Hence, the expression for the pressure net force, for instance, is as follows:

$$F_x^{Pressure} = \left[p - \left(p + \frac{\partial p}{\partial x}dx\right)\right]dydz \tag{2.11}$$

Analogous to equation 2.11, the net force for viscous stresses is obtained:

$$\begin{aligned} F_x^{Viscous} = &\left[\left(\tau_{xx} + \frac{\partial \tau_{xx}}{\partial x}dx\right) - \tau_{xx}\right]dydz \\ &+ \left[\left(\tau_{yx} + \frac{\partial \tau_{yx}}{\partial y}dy\right) - \tau_{xx}\right]dxdz \\ &+ \left[\left(\tau_{zx} + \frac{\partial \tau_{zx}}{\partial z}dz\right) - \tau_{xx}\right]dxdz \end{aligned} \tag{2.12}$$

Adding 2.10, 2.11 and 2.12, The sum of forces in direction x is obtained:

$$F_x = \left(-\frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z}\right)dxdydz + \rho f_x dxdydz \tag{2.13}$$

Expressing the mass as:

$$m = \rho dxdydz \tag{2.14}$$

And the acceleration of the fluid element as:

$$a_x = \frac{Du}{Dt} \tag{2.15}$$

Where u, v and w are the scalar components of $\overrightarrow{v}$ in the x, y and z directions respectively.

Taking equations 2.13, 2.14 and 2.15:

$$\rho\frac{Du}{Dt} = -\frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} + \rho f_x \tag{2.16}$$

Analogously, for the y and z directions:

$$\rho\frac{Dv}{Dt} = -\frac{\partial p}{\partial y} + \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \tau_{yy}}{\partial y} + \frac{\partial \tau_{zy}}{\partial z} + \rho f_y \tag{2.17}$$

$$\rho\frac{Dw}{Dt} = -\frac{\partial p}{\partial z} + \frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \tau_{zz}}{\partial z} + \rho f_z \tag{2.18}$$

Using a similar procedure than in 2.6, the other form of the momentum equations can be obtained:

$$\frac{\partial \rho u}{\partial t} + \nabla \cdot (\rho u \overrightarrow{v}) = -\frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} + \rho f_x \tag{2.19}$$

$$\frac{\partial \rho v}{\partial t} + \nabla \cdot (\rho v \overrightarrow{v}) = -\frac{\partial p}{\partial y} + \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \tau_{yy}}{\partial y} + \frac{\partial \tau_{zy}}{\partial z} + \rho f_y \tag{2.20}$$

$$\frac{\partial \rho w}{\partial t} + \nabla \cdot (\rho w \overrightarrow{v}) = -\frac{\partial p}{\partial z} + \frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \tau_{zz}}{\partial z} + \rho f_z \tag{2.21}$$

For the case of Newtonian fluids, Stokes obtained a relation for the viscous stresses:

$$\tau_{xx} = \lambda \nabla \cdot \overrightarrow{v} + 2\mu\frac{\partial u}{\partial x} \tag{2.22}$$

$$\tau_{yy} = \lambda \nabla \cdot \overrightarrow{v} + 2\mu\frac{\partial v}{\partial y} \tag{2.23}$$

$$\tau_{zz} = \lambda \nabla \cdot \overrightarrow{v} + 2\mu\frac{\partial w}{\partial z} \tag{2.24}$$

$$\tau_{xy} = \tau_{yx} = \mu\left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right) \tag{2.25}$$

$$\tau_{xz} = \tau_{zx} = \mu\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right) \tag{2.26}$$

$$\tau_{yz} = \tau_{zy} = \mu\left(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z}\right) \tag{2.27}$$

For a more detailed derivation of the momentum equations check [3].

Conservation of energy

For this equation, an infinitesimal fluid element moving with the flow will be taken in order to perform the derivation.

Due to the energy being conserved, the rate change of energy within a system is caused by two factors: The heat flux into the element and rate of work done to it by the body and surface forces. The latter can be defined as $F \cdot \overrightarrow{v}$, and thus, for the body forces:

$$\dot{W}^{Body} = \rho \overrightarrow{f} \cdot \overrightarrow{v} dxdydz \tag{2.28}$$

As for the surface forces, the procedure is the same than the one used for the momentum equation, only multiplying by the velocity. The resulting equation is the following one:

$$\begin{aligned} \dot{W}^{Surface} = \Bigg[& - \frac{\partial(up)}{\partial x} - \frac{\partial(vp)}{\partial y} - \frac{\partial(wp)}{\partial z} + \frac{\partial(u\tau_{xx})}{\partial x} + \frac{\partial(u\tau_{yx})}{\partial y} \\ & + \frac{\partial(u\tau_{zx})}{\partial z} + \frac{\partial(v\tau_{xy})}{\partial x} + \frac{\partial(v\tau_{yy})}{\partial y} + \frac{\partial(v\tau_{zy})}{\partial z} + \frac{\partial(w\tau_{xz})}{\partial x} \\ & + \frac{\partial(w\tau_{yz})}{\partial y} + \frac{\partial(w\tau_{zz})}{\partial z} \Bigg] dxdydz \end{aligned} \tag{2.29}$$

Where u, v and w are the components of the velocity vector in the x, y and z direction respectively.

Now, the heat flux into the element can be split in two different terms, one is the volumetric heating of the element, for instance due to radiation, and the other is thermal conduction across the boundaries of the fluid element. For the former, the resulting equation is the next one:

$$\dot{Q}_{vol} = \rho \dot{q}_{vol} dxdydz \tag{2.30}$$

For the thermal conduction, the strategy is also the same as before, this yields:

$$\dot{Q}_{cond} = -\left(\frac{\partial \dot{q}_{x\,cond}}{\partial x} + \frac{\partial \dot{q}_{y\,cond}}{\partial y} + \frac{\partial \dot{q}_{z\,cond}}{\partial z} \right) dxdydz \tag{2.31}$$

Also, heat conduction can be expressed as:

$$\dot{q}_i = -k \frac{\partial T}{\partial i} \tag{2.32}$$

The remaining term in the equation is the change in energy within the element, this consists in both the internal energy of the fluid element and its kinetic energy. The expression is obtained using the substantial derivative:

$$\rho \frac{D}{Dt} \left(e + \frac{V^2}{2} \right) dxdydz \tag{2.33}$$

Putting equations 2.28, 2.29, 2.30, 2.31, 2.32 and 2.33 together, the energy conservation equation is obtained:

$$\begin{aligned}\rho\frac{D}{Dt}\left(e+\frac{V^2}{2}\right) = \rho\dot{q}_{Body} &+ \frac{\partial}{\partial x}\left(k\frac{\partial T}{\partial x}\right)+\frac{\partial}{\partial y}\left(k\frac{\partial T}{\partial y}\right)+\frac{\partial}{\partial z}\left(k\frac{\partial T}{\partial z}\right)\\ &-\frac{\partial(up)}{\partial x}-\frac{\partial(vp)}{\partial y}-\frac{\partial(wp)}{\partial z}+\frac{\partial(u\tau_{xx})}{\partial x}+\frac{\partial(u\tau_{yx})}{\partial y}\\ &+\frac{\partial(u\tau_{zx})}{\partial z}+\frac{\partial(v\tau_{xy})}{\partial x}+\frac{\partial(v\tau_{yy})}{\partial y}+\frac{\partial(v\tau_{zy})}{\partial z}\\ &+\frac{\partial(w\tau_{xz})}{\partial x}+\frac{\partial(w\tau_{yz})}{\partial y}+\frac{\partial(w\tau_{zz})}{\partial z}+\rho\overrightarrow{f}\cdot\overrightarrow{v}\end{aligned} \tag{2.34}$$

For a more detailed derivation check [3].

2.3 RANS

The most used method in CFD is RANS (Reynolds Averaged Navier-Stokes), the reason why is computational speed. The basis of RANS computation is to split the time variation of the different variables into two: the mean value and the fluctuations.

Those fluctuations are modeled in order to eliminate the turbulence from the computation and only the mean values are calculated. This means that this method allows for a very fast computing time in comparison with DNS and that is why, despite its imprecision, it is the chosen method for practical applications[3].

2.3.1 Mathematical formulation

The base equations used in RANS are the same as in DNS, as the physical model is the same. But the mean value needs to be defined.

$$\overline{v_i}(t_0) = \frac{1}{T}\int_{t_0-T/2}^{t_0+T/2} v_i dt \tag{2.35}$$

Where $v_i = \overline{v_i} + v_i'$ being v_i' the fluctuating part of v_i which has to be modeled. Note that $\overline{v_i'} = 0$.

This generates extra terms in the equations. For instance, for incompressible flow:

$$\frac{\partial\overline{v_i}}{\partial\overline{x_i}} = 0 \tag{2.36}$$

$$\frac{\partial\overline{v_i}}{\partial t}+\frac{\partial}{\partial x_j}\overline{v_j}\cdot\overline{v_i}+\frac{1}{\rho}\frac{\partial\overline{p}}{\partial x_i} = v\frac{\partial^2}{\partial x_j^2}\overline{v_i}-\frac{\partial}{\partial x_j}\overline{v'_j v'_i} \tag{2.37}$$

The last term in equation 2.37 is called the turbulent stress tensor and the component $\overline{v'_j v'_i}$ is a Reynolds stress.

The equations for the Reynolds stresses can be obtained from the Navier-Stokes equations and the result is the following:

$$\begin{aligned}\frac{\partial}{\partial t}\overline{v`_j v`_i} + \overline{v_k}\frac{\partial}{\partial x_k}\overline{v`_j v`_i} = &-\overline{v`_j v`_i}\frac{\partial \overline{v_j}}{\partial x_k} - \overline{v`_j v`_k}\frac{\partial \overline{v_i}}{\partial x_k} + \frac{\partial}{\partial x_k}\overline{v`_i v`_j v`_k} \\ &+ \frac{1}{\rho}\overline{\frac{\partial p`}{\partial x_i}v`_j + \frac{\partial p'}{\partial x_j}v`_i} + v\frac{\partial^2}{\partial x_k^2}\overline{v`_j v`_i} - 2v\overline{\frac{\partial v`_i}{\partial x_k}\frac{\partial v`_j}{\partial x_k}}\end{aligned} \tag{2.38}$$

The left hand of the equation 2.38 corresponds to the substantial derivative. The first two terms on the right side are the production of the Reynolds stresses, the following two represent the turbulent diffusion and the lasts terms corresponds to the dissipation.

For more details consult[3].

2.4 LES

Lastly, the third method for CFD is known as LES (Large Eddy Simulation). It is a compromise between precision and computational resources.

The main idea behind LES is to model turbulence only in the smallest scales and calculate directly the bigger ones. This results on a simulation that is not as slow as DNS but it is much more precise than RANS. This approach is becoming viable as computer technology develops further as computing time in LES depends on the number of grid points and computer memory capacity is directly proportional to grid point capacity [2].

2.4.1 Mathematical formulation

The mathematical formulation in LES is a combination of DNS for the scales bigger than the grid scale, and RANS for the scales smaller than the grid scale.

This means that there is no extra formulation for LES other than what has already been discussed in the previous sections. It needs, however, some corrections in order to reduce the uncertainty due to modeling. In spite of that, for flows with no separation and small amounts of turbulence, no correction is required for engineering applications as the uncertainty generated is about 1% which is more than acceptable[2].

2.5 Current advancements

The field of CFD has advanced a lot in the last few decades as computer science allowed for faster computation. The possibilities are limitless going as far as one can imagine.

One very good example of how far the field has advanced is the design of the *Bloodhound SC*[4], a world record breaking supersonic car. In order to optimise the design in all aspects (performance,

noise, etc.), a specifically made CFD program has been used by the developers. Their code was capable of solving a very complex problem having supersonic flow, ground effect, rotating wheels and the exhaust from the jet and rocket engines present in the car. Solving this would have been a dream decades ago.

Another important advancement is the development of adaptive meshes. Those allow both faster and more precise computations. This kind of mesh is a boundary fitted unstructured mesh for transient problems which moves with time in the physical plane. By doing this, the denser part of the mesh stays around the sections of the domain of analysis which need greater accuracy. This type of mesh also compresses or stretches the mesh in different regions either to reduce the error or the computational time; this means that it can delete or create nodes as well.

This is done with different kinds of mesh transformations which must be recalculated after every iteration or time-step. To know more check [3] and [5].

References

[1] J. Graves, R. A., R. A., and Jr., "Computational fluid dynamics - The coming revolution," *Astronautics and Aeronautics, vol. 20, Mar. 1982, p. 20-28, 62.*, vol. 20, pp. 20–28, 1982. [Online]. Available: http://adsabs.harvard.edu/abs/1982AsAer..20...20G

[2] D. R. Chapman, "Computatisnal Aerodynamics Development and Outlook," *AIAA Journal*, vol. 17, no. 12, pp. 1293–1313, dec 1979. [Online]. Available: http://arc.aiaa.org/doi/abs/10.2514/3.61311

[3] J. F. Wendt, J. D. Anderson, and Von Karman Institute for Fluid Dynamics., *Computational fluid dynamics : an introduction.* Springer, 2008. [Online]. Available: http://mendeley.csuc.cat/fitxers/e9bc9947c308b34eda97afa922b53cc4

[4] B. Evans, T. Morton, L. Sheridan, O. Hassan, K. Morgan, J. W. Jones, M. Chapman, R. Ayers, and I. Niven, "Design optimisation using computational fluid dynamics applied to a land–based supersonic vehicle, the BLOODHOUND SSC," *Structural and Multidisciplinary Optimization*, vol. 47, no. 2, pp. 301–316, feb 2013. [Online]. Available: http://link.springer.com/10.1007/s00158-012-0826-0

[5] R. Löhner, J. R. Cebral, F. E. Camelli, S. Appanaboyina, J. D. Baum, E. L. Mestreau, and O. A. Soto, "Adaptive embedded and immersed unstructured grid techniques," *Computer Methods in Applied Mechanics and Engineering*, vol. 197, no. 25-28, pp. 2173–2197, apr 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045782507003714

Chapter 3

DEVELOPMENT METHOD

Contents

In order to face the challenges imposed by this project in a progressive manner, the code has been developed in small steps. In that way, errors can be detected while the code is still simple and short allowing an easier debugging.

The order followed during the development of the code is the following:

1. Potential flow
2. Blocking-off technique
3. Compressible flow
4. Convection-diffusion equation
5. Incompressible Navier-Stokes equations

In the first step, developing the base algorithm was the main objective. To do so, a simple case has been programmed to test the functionality of said algorithm.

In this manner, any errors in subsequent steps may not be attributed to the solver as it would have been tested in this step.

During the second step, solids would be introduced inside the mesh, wanting to test the capabilities of the code to compute fluid-solid interactions. Checking the result of a simple case with experimental or analytical result would validate the code's capabilities to calculate interactions.

The third step has as objective to allow the code to calculate the fluid's properties by analyzing a compressible flow case.

The fourth step aims to develop a code capable of computing convection-diffusion cases in order to get ready for the final step.

And lastly, the final step is the main objective of this project, the incompressible Navier-Stokes equations.

3.1 Process

In order to properly develop the code with ease, four basic steps have been followed:

1. Formulation of the problem
2. Algorithm design
3. Code development
4. Test and validation

Following this procedure, the task becomes more organized and thus, it is less prone to errors.

3.1.1 Formulation of the problem

To begin with the programming, first, it is necessary to define the problem and to linearize the governing equations. For that, the problem is first formulated in a paper sheet in a traditional manner, then, the equations are linearized and the resulting coefficients are determined. During this step, the boundary conditions are also defined as well as any other auxiliary variables or functions needed for the resolution of the given problem.

3.1.2 Algorithm design

Once the problem has been formulated, the next step is to design an algorithm capable of solving it. This is done first in a paper sheet as a diagram in order to facilitate programming.

In this step, all algorithm related variable and functions are also defined.

3.1.3 Code development

Once both the problem and the algorithm have been defined, the code itself needs to be written. So as to make troubleshooting easier and make the code easier to read and work with, the software has been designed using different functions.

The use of functions also allows the reuse of useful ones for other codes, which comes in handy for this project.

3.1.4 Test and validation

Once the code runs properly, it has to be tested with experimental or analytical results.

This comparison allows the detection of errors in the code which must be corrected to make the code perform properly.

Chapter 4

POTENTIAL FLOW

Contents

The development of a code capable of solving a potential flow case is the first step of the project. In this step the base algorithm will be developed and tested.

4.1 Formulation of the problem

In this case, as the flow is non-viscous, the problem simplifies and the governing equations used are called the Euler equations. Furthermore, as this case will be considered irrotational and 2D, the approach taken will use the stream functions instead, which simplify the problem even further.

The meshing chosen for this part has been a uniform mesh made of quadrilateral control volumes with centered nodes. Meshing is an art on itself and as the main objective of the project is to design the algorithms and functions needed to solve the problem, using a simple mesh like this one is ideal, as there will be less errors due to a poorly generated mesh.

$$v_x = \frac{\rho_0}{\rho}\frac{\partial \psi}{\partial y} \qquad\qquad v_y = -\frac{\rho_0}{\rho}\frac{\partial \psi}{\partial x} \tag{4.1}$$

Using Stoke's theorem, as the problem is irrotational, it is possible to evaluate the circulation as follows:

$$\Gamma = \oint_C \overrightarrow{v}\,\overrightarrow{dl} = 0 \tag{4.2}$$

Where, $\overrightarrow{dl}$ corresponds, in this case, to the corresponding differential length of the control volume.

Taking equations 4.1 and 4.2, assuming a quadrilateral control volume, the circulation of a control volume can be evaluated as:

$$-\frac{\rho_0}{\rho_e}\frac{\psi_E-\psi_P}{d_{PE}}\Delta y_P-\frac{\rho_0}{\rho_n}\frac{\psi_N-\psi_P}{d_{PN}}\Delta x_P+\frac{\rho_0}{\rho_w}\frac{\psi_P-\psi_W}{d_{PW}}\Delta y_P+\frac{\rho_0}{\rho_s}\frac{\psi_P-\psi_S}{d_{PS}}\Delta x_P=0 \tag{4.3}$$

Where the sub-indexes E, N, W, S and P correspond to the nodes on the right, top, left, bottom and the one evaluated respectively. The sub-indexes e, n, w and s are the boundaries between the node evaluated and the node on its right, top, left and bottom respectively. The variables d_{ij}, Δx_P and Δy_P are the distance between nodes, the differential length in the x direction and the differential length in the y direction respectively.

After obtaining equation 4.3, it is necessary to linearize it so as to be able to solve it with the program.

$$a_P\psi_P=a_E\psi_E+a_W\psi_W+a_N\psi_N+a_S\psi_S+b_P \tag{4.4}$$

The coefficients of equation 4.4 are defined as follows:

$$\begin{aligned}
a_E &= \frac{\rho_0}{\rho_e}\frac{\Delta y_P}{d_{PE}}\\
a_W &= \frac{\rho_0}{\rho_w}\frac{\Delta y_P}{d_{PW}}\\
a_N &= \frac{\rho_0}{\rho_n}\frac{\Delta x_P}{d_{PN}}\\
a_S &= \frac{\rho_0}{\rho_s}\frac{\Delta x_P}{d_{PS}}\\
a_P &= a_E+a_w+a_N+a_S\\
b_P &= 0
\end{aligned} \tag{4.5}$$

As we can see in 4.5, as the case is incompressible and the mesh will not change, the coeficients obtained are constant.

Lastly, the boundary conditions need to be defined. The top and bottom boundaries have a condition of the Neumann type, which means that there is no flow going through them, thus, the velocity perpendicular to those boundaries needs to be 0. Even though this boundary condition is of Neumann type, the value of the stream function is known as it will correspond to the value at the inlet node of the same y position due to the fact that there is no y component of the velocity. This means it is actually possible to define this condition as a Dirichlet condition, where the value itself is known.

The other Dirichlet condition in this problem is found at the inlet, the values of the streamlines at the inlet nodes are $\psi = v_\infty y$. Where v_∞ is the free stream velocity and y is the vertical position of the node.

The remaining boundary condition is the outlet boundary. For the outlet, it is known that there is no vertical component of the velocity, that means that the value of the stream function will be the same as the node immediately to its left.

4.2 Algorithm design

Before designing the algorithm itself, the solver needs to be defined. For this problem, the solver chosen has been the line-by-line solver, as it is faster than a fully iterative Gauss-Seidel but still not too complex. Overall the complexity in relation to the computational efficiency makes this solver a good choice.

The line-by-line solver is based on the TDMA which is a direct solver for 1D cases. It is based on the fact that the equations of a 1D problem of this type, generate a tri-diagonal matrix of coefficients and this can be solved directly, without iteration. this means that the error in the solution is only related to the mesh size, and on top of that, it is very fast, as no iterations are needed.

The problem with the TDMA is that it can only be applied to 1D problems as 2D cases, for instance, no longer have a tri-diagonal matrix of coefficients. The line-by-line solver then transforms a higher dimension problem into a 1D problem before using a TDMA. It does so in the following way:

$$b_P^* = b_P + a_N \psi_N^0 + a_S \psi_S^0 \tag{4.6}$$

By adding the multiplication of the coefficients of the second dimension and the estimated value of the stream function the problem then transforms into a one dimensional case. This means a TDMA can be applied, but, the value of the stream function needs to be estimated which means the solver is going to need some iterations. However, unlike a Gauss-Seidel solver, the main way of calculation is not iteration but rather the TDMA, which means the number of iterations needed to solve the problem are less.

To make the solver even faster, it is possible to apply the line-by-line solver alternating between the two dimensions, meaning that first, the y direction coefficient are added to b_P and when the solver finishes one iteration, the x direction coefficients are added instead. In that way, the errors due to the estimation of the stream function do not spread as much.

For the TDMA, two coefficients need to be determined:

$$\begin{aligned} P[n] &= \frac{a_E[n]}{a_P[n] - a_W[n]P[n-1]} \\ Q[n] &= \frac{b_P^*[n] + a_W[n]Q[n-1]}{a_P[n] - a_W[n]P[n-1]} \end{aligned} \tag{4.7}$$

Where n is the node number in the direction in which the TDMA is being applied.

As seen in equation 4.7, both coefficients only depend on the preceding node's coefficients and given the fact that the first node's a_W coefficient is zero, it is possible to calculate directly all coefficients

using a for loop from the first node to the last.

Once those coefficients are calculated, the stream function can be obtained:

$$\psi[n] = P[n]\psi[n+1] + Q[n] \tag{4.8}$$

As seen in equation 4.8 each node's stream function only depends on the already calculated coefficients and the following node's stream function. As the last node's P coefficient will be zero, due to the fact that its a_E coefficient is always 0, the values of the stream function can be calculated directly from last to first node.

Once the solver is properly defined, the whole algorithm must be designed. The resulting algorithm has been the following:

1. input data
2. mesh generation
3. geometrical data calculation
4. initial ψ estimation
5. coefficient calculation
6. boundary conditions
7. solver
8. error check
    - if maxError > errorAccptance go to 7
    - else continue
9. final calculations

Inside point 7, the line-by-line solver can be found.

4.3 Code development

The code is structured in three main parts, a preprocessor, the solver and a postprocessor.

Inside the preprocessor, all necessary calculations related to initial values and mesh generation are found. In the solver, all necessary coefficients are calculated and then the solving process is initiated. Lastly, in the postprocessor, the data is stored in an external file to allow plotting using a program called *gnuplot*.

The code is structured in functions to facilitate troubleshooting and make it more visually pleasing, having a chunk of code with everything in it makes it harder to read. And on top of that, splitting

the code into functions allow re-usability for other programs as some of the functions used are very general and thus they can be recycled.

The whole code can be checked at attachment A.

4.4 Test and validation

Validating the code in this particular case is very easy as the expected result is just parallel lines across all the domain as there are no interaction with anything due to the problem being non-viscous and having no objects inside the domain.

The result obtained with the code is akin to reality as it can be seen in figure 4.1. It can be easily seen that the value of the stream function remains constant along the x direction.

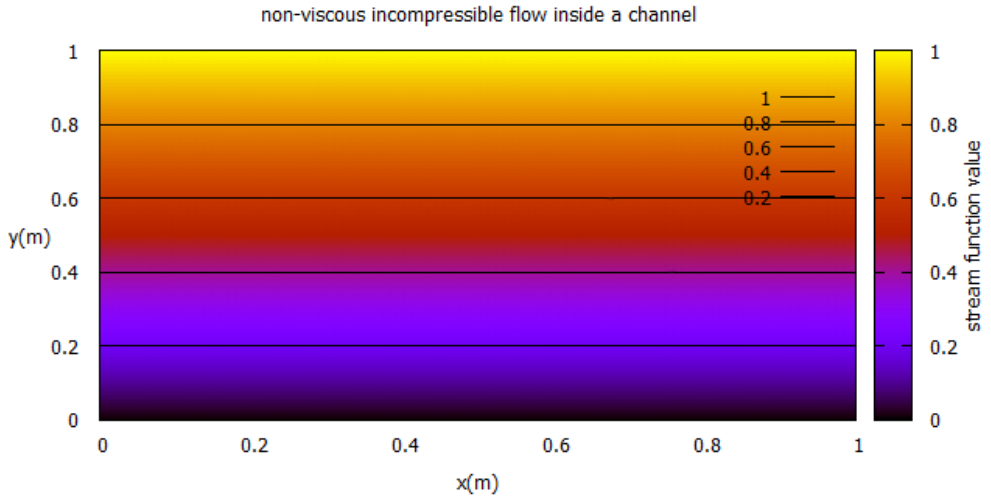


Figure 4.1: Result of the non-viscous incompressible problem

During the validation process, one of the objectives was to find the necessary precision needed for the code to run properly. As it can be seen in the code found in attachment A, the precision demanded is equal to $1/(100 \cdot nx \cdot ny)$ where the multiplication $nx \cdot ny$ is the number of nodes.

The actual minimal value for the code to yield the correct result is $1/(10 \cdot nx \cdot ny)$, but diminishing the order of magnitude by one did not increase the computation time too much for the amounts of nodes used.

In this particular case, a grid of 100x100 elements has been used as increasing it further did not show any improvement in the result and the computing time increased exponentially.

In figure 4.2 the results of the code using a precision of $1/(0.1 \cdot nx \cdot ny)$ can be seen. Note that the values deviate from the correct result.

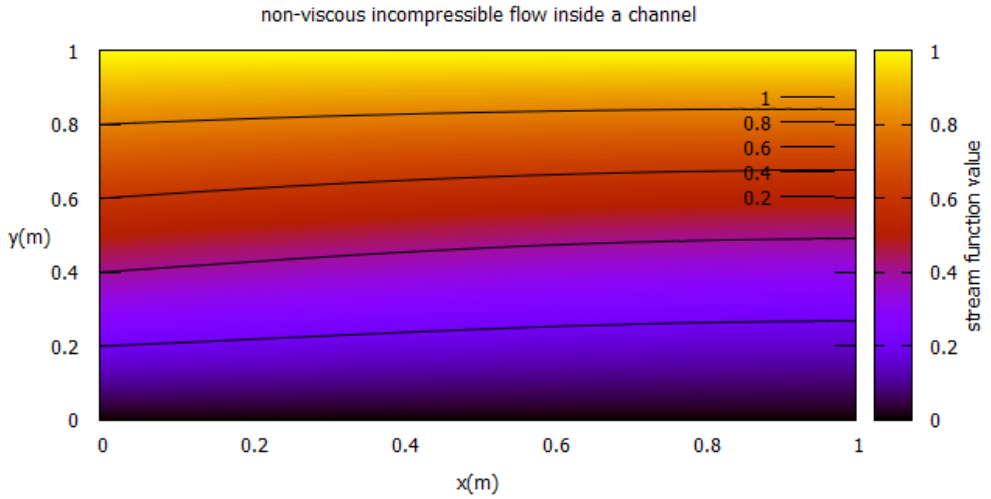


Figure 4.2: Result using only a precision value of $1/(0.1 \cdot nx \cdot ny)$

Chapter 5

BLOCKING-OFF

Contents

In this step, a blocking-off technique will be implemented as well as a method to find the appropriate value for the solid's stream function value.

5.1 Formulation of the problem

In this case, the problem obeys the same equations as in the potential flow case. The difference is that a solid will be immersed in the mesh. The solid chosen has been a cylinder as an analytical result exists making validation easy.

The best way to treat solids is to actually revolve the mesh around it, but as the objective of this project is not to develop a high quality mesh, a simpler method has been used. The blocking-off method assigns each node to either a fluid region or a solid region, then when calculating the coefficients, ρ_0/ρ_i is calculated as a harmonic average5.1.

$$\frac{\rho_0}{\rho_i} = \frac{dPI}{\frac{dPi}{\frac{\rho_0}{\rho_P}} + \frac{dIi}{\frac{\rho_0}{\rho_I}}} \tag{5.1}$$

That way, by assigning a density of 0 to solid nodes, the harmonic average between solids is 0, as this is an incompressible case, the average between liquids is ρ_0 and the only values that are affected are, then, the nodes that are in the boundary between solid and liquid.

Th second phase of this problem is to develop a subroutine that finds the appropriate value for the solid's stream function in order to meet the required circulation. The method used for that purpose

is the secant method.

The basis of that is to draw a line that passes through the points defined by the circulation and the value of the stream function in the solid of the current and previous iteration. Then, the next value of the stream function in the solid corresponds to the crossing of the line with the x axis.

The derivation of the equation to find that next value is the following:

$$\begin{aligned} \Gamma_{prev} &= m\Psi_{prev} + b \\ \Gamma_0 &= m\Psi_0 + b \end{aligned} \tag{5.2}$$

$$\Gamma_1 = \frac{\Gamma_{prev} - \Gamma_0}{\Psi_{prev} - \Psi_0}\Psi_1 + \Gamma_0 - \frac{\Gamma_{prev} - \Gamma_0}{\Psi_{prev}\Psi_0}\Psi_0 \tag{5.3}$$

Making $\Gamma_1 = 0$:

$$\Psi_1 = \frac{\frac{\Gamma_{prev}-\Gamma_0}{\Psi_{prev}\Psi_0}\Psi_0 - \Gamma_0}{\frac{\Gamma_{prev}-\Gamma_0}{\Psi_{prev}-\Psi_0}} = \frac{\Gamma_{prev}\Psi_0 - \Psi_{prev}\Gamma_0}{\Gamma_{prev} - \Gamma_0} \tag{5.4}$$

5.2 Algorithm design

The algorithm used for this case is very similar to the previous one in its core, but with some extra steps.

Mainly, this extra steps are required in order to find the correct stream function value of the solid and iterate its value until the desired circulation is met.

In order to calculate the circulation around the solid body, the next expression has been used:

$$\Gamma = -\frac{\rho_0}{\rho_e}\frac{\Psi_E - \Psi_P}{d_{EP}}\Delta y_P - \frac{\rho_0}{\rho_n}\frac{\Psi_N - \Psi_P}{d_{NP}}\Delta x_P + \frac{\rho_0}{\rho_w}\frac{\Psi_P - \Psi_W}{d_{WP}}\Delta y_P + \frac{\rho_0}{\rho_s}\frac{\Psi_P - \Psi_S}{d_{SP}}\Delta x_P \tag{5.5}$$

Equation 5.5 is evaluated at the end, when the solver has finished iterating. Then, using the secant method, a new value of Ψ_{solid} is found and the solver restarts the iteration using this newfound value for the solid.

The algorithm for this case has been designed as follows:

1. input data
2. mesh generation
3. geometrical data calculation
4. blocking-off routine
5. coefficient calculation

6. initial Ψ estimation
7. boundary conditions
8. solver
9. error check
   - if maxError > errorAcceptance go to 8
   - else continue
10. circulation evaluation
11. error check
    - if circMaxError > circErrorAcceptance do: secant method & go to 6
    - else continue
12. final calculations

Note that the boundary conditions are calculated inside the loop for the circulation, this is done because in the code, the coefficients of the nodes pertaining to the solid region are calculated there and they depend on the solid stream function value.

5.3 Code development

The code written for this task, uses as a base the code in attachment A and adds the necessary functions to allow a solid to fit inside the mesh and to calculate the circulation value.

All other functions have been taken from the previous code as the have already been tested and thus, errors are not likely to come from them.

The whole code for this case can be checked at attachment B.

5.4 Test and validation

The validation of this part has been done in two steps. First, the code is input the correct value for the solid's stream function so as to check the result as it is and lastly, the whole iterative process is tested by having the code start with an arbitrary stream function value.

In the first test, the results have been satisfactory, in figure 5.1 the result of the program is presented and in figure 5.2, the analytical solution to the problem is presented. This analytical solution has the following form in polar coordinates:

$$\Psi = U_\infty \sin\theta\left(r - \frac{R^2}{r}\right) + \frac{\Gamma}{2\pi}\ln\frac{r}{R} \tag{5.6}$$

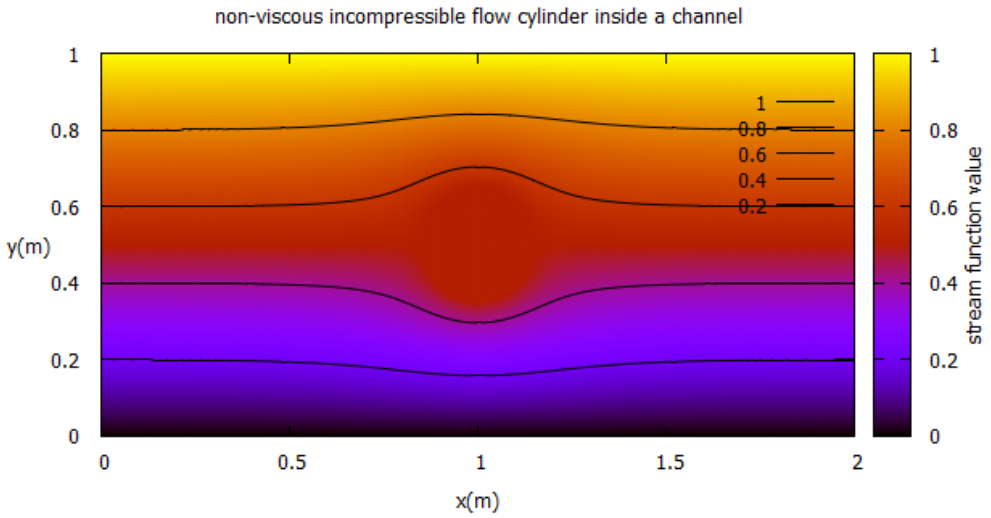


Figure 5.1: Program result of a cylinder in incompressible flow with 200x200 nodes

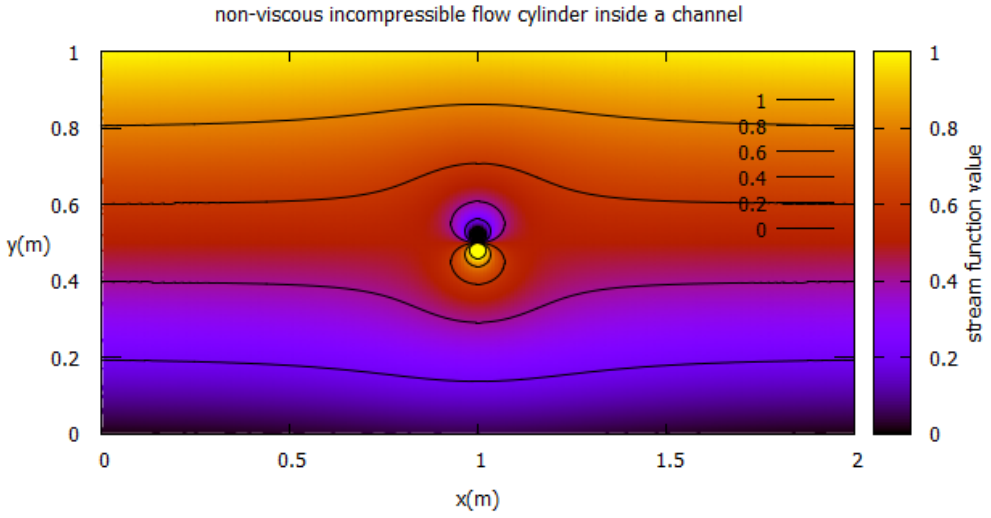


Figure 5.2: analytical result of a cylinder in incompressible flow

As we can see, the results are very similar but not exactly equal. A better result is obtained as the

mesh is made finer but the computational time increases exponentially. Hence, for fast calculations, using a grid made of 200x200 nodes is a good compromise as it gives a good precision in the result and the calculations are swift.

Note that while the program result considers only the domain in which the mesh is generated, the analytical solution does not, this means that, for instance, in the program the inlet conditions force a value of Ψ on every node which as we can see in both figure 5.1 and 5.2 does not match. This, however, can be solved by making the whole domain bigger.

This is made evident comparing figures 5.3 and 5.4.

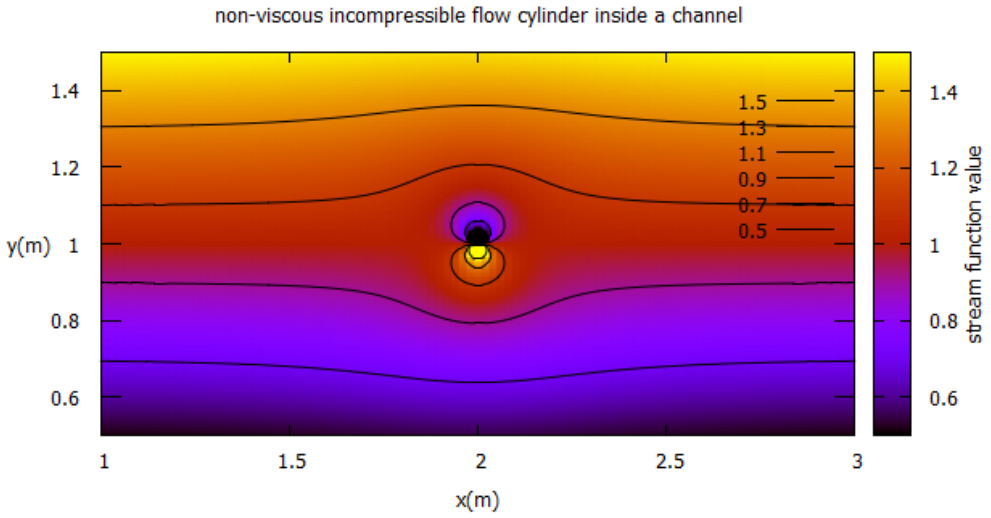


Figure 5.3: Analytical result of a cylinder in incompressible flow with a bigger domain of calculation

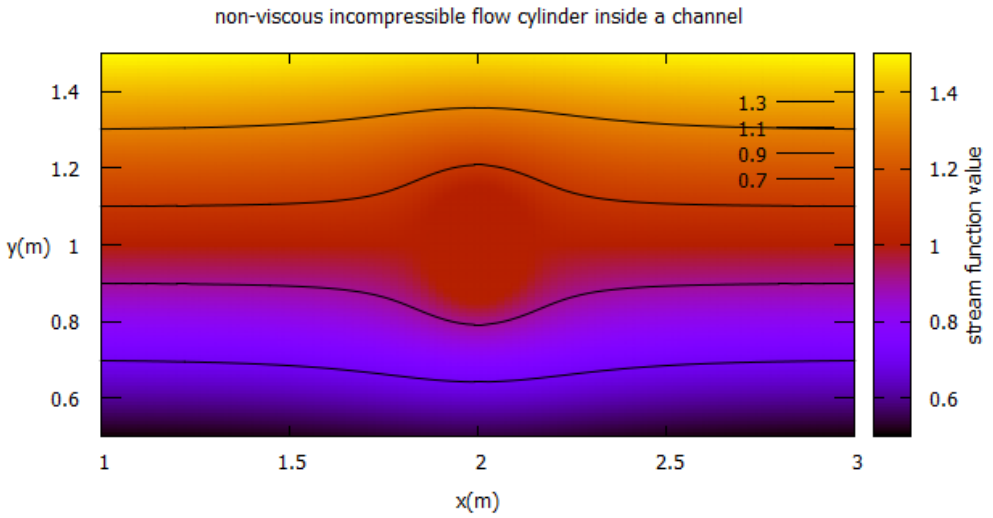


Figure 5.4: Program's result of a cylinder in incompressible flow with a bigger domain of calculation

Also, the different treatments of the solid in both the analytical and the program's result can be seen. In figure 5.1, the whole cylinder has the same stream function value while in figure 5.2, due to mathematics, the cylinder acts as a doublet.

Knowing that the program gives appropriate results, the second step in the validation is checking that the software can find the correct value of Ψ for a given Γ_{solid} on its own.

Unfortunately, this step in the validation has been unsuccessful as the software does not seem to iterate properly. The issue must lie in one of two steps: either the secant method or the circulation calculation.

With output from the software, the error has been traced back to the circulation calculation, which means the coefficients have to be incorrect, but this does not make sense given that the result obtained with the program is akin to the analytical result. Hence the issue remains unknown.

Chapter 6

COMPRESSIBLE FLOW

Contents

After validating the blocking off technique, The next step was to make the program capable of altering the physical properties of the fluid, particularly, a compressible case.

6.1 Formulation of the problem

In this case, the physical formulation is the same as in the previous chapter, the only difference being that the density changes, which means that the coefficients are altered on every iteration.

This introduces changes to the base algorithm, as the coefficients were previously calculated before starting the iterative process whereas now they must be calculated inside it.

In order to compute the physical properties of the fluid, the velocity in every node needs to be computed first. Taking the definition of the stream function:

$$\begin{aligned} v_x &= \frac{\rho_0}{\rho}\frac{\partial \Psi}{\partial y} \\ v_y &= -\frac{\rho_0}{\rho}\frac{\partial \Psi}{\partial x} \end{aligned} \tag{6.1}$$

Where the sub-index 0 corresponds to the free stream properties.

Using equation 6.1, both components of the velocity at the node can be calculated by averaging between the velocities at the faces; then, the modulus is computed.

Once the velocity is obtained, as the total energy of the fluid is conserved:

$$T_P = T_0 + \frac{\left(v_0^2 - v_P^2\right)}{2\overline{c_P}} \tag{6.2}$$

Then, using the isentropic relation, as the problem is non-viscous:

$$P_P = P_0 \left(\frac{T_P}{T_0}\right)^{\frac{\gamma}{\gamma-1}} \tag{6.3}$$

Finally, assuming ideal gas:

$$\rho_P = \frac{P_P}{RT_P} \tag{6.4}$$

Another point that must be taken into consideration is the fact that as density changes, the density at the surfaces of every node must be calculated using a harmonic average.

6.2 Algorithm design

As previously mentioned, the algorithm has changed for this case in order to allow the necessary recalculation of coefficients due to changes in the density of the fluid.

Besides that, the rest of the algorithm remains unchanged:

1. input data
2. mesh generation
3. geometrical data calculation
4. blocking-off routine
5. initial Ψ estimation
6. coefficient calculation
7. boundary conditions
8. solver
9. gas properties calculation
10. error check
    - if maxError > errorAcceptance go to 6
    - else continue
11. circulation evaluation
12. error check

- if circMaxError > circErrorAcceptance do: secant method & go to 5
- else continue

13. final calculations

6.3 Code development

Most of the functions used for this case have been reused from the previous case (incompressible flow), however, due to adding compressibility, some have been modified accordingly and some new functions have been written in order to calculate the physical properties of the fluid.

The complete code can be checked at attachment C.

Note that, in the code, when calculating the modulus of the velocity, the value is relaxed by a parameter. This is done to prevent overflow in the early iterations as initially the values calculated can be incredibly off. This, however, makes the code converge more slowly but it is a good solution nonetheless.

Another fix that has been done to the code, is to use the absolute value of the temperature when calculating the pressure and the density as even though the value of the velocity has been relaxed, it resulted in negative temperatures sometimes which caused the density to be negative, resulting in computation errors. This solution is a bit unstable and only allows for very low Mach number calculations, instead of that, another option would have been to further relax the velocity, but, this resulted in underflow or in wrong results as the relaxing factor needed to be too extreme.

6.4 Test and validation

This code suffers of the same issue as the incompressible potential flow code, it cannot find the proper Ψ value for a given Γ_{solid}. This was expected as the base program is the same and the solution of this problem still remains unknown.

Besides that, the program gives appropriate results for very low Mach numbers. As mentioned above, due to computational issues, this code is limited to very low Mach numbers.

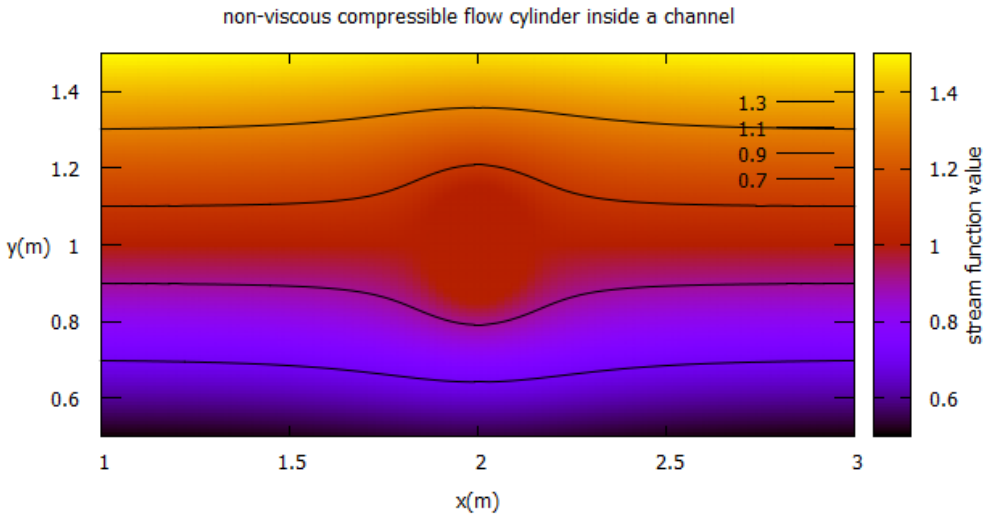


Figure 6.1: Program result of a cylinder in compressible flow with 200x200 nodes

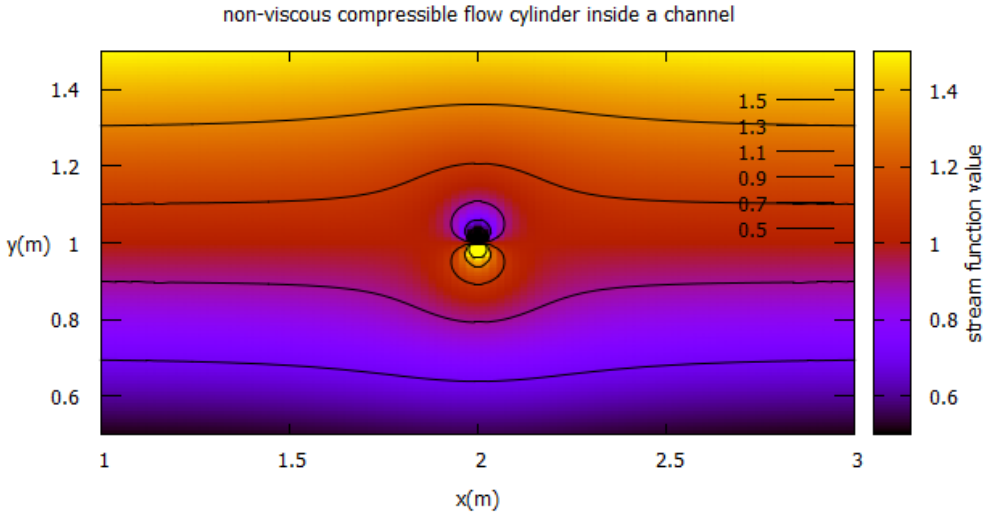


Figure 6.2: Analytical result of a cylinder in compressible flow with 200x200 nodes

As can be seen in figures 6.1 and 6.2, the results given by the program are akin to the analytical

solution. The analytical solution for this case is the following[1]:

$$\phi = U_\infty r\left(1+\frac{R^2}{r^2}\right)\cos\theta - \frac{M^2U_\infty r}{12}\left(\left(\frac{13R^2}{r^2}+\frac{R^6}{r^6}-\frac{6R^4}{r^4}\right)\cos\theta+\left(\frac{R^4}{r^4}-\frac{3R^2}{r^2}\right)\cos 3\theta\right) \quad (6.5)$$

Where, r and θ are the polar coordinates, M is the Mach number and R is the cylinder radius in meters.

But, as the program works with stream functions, this solution needs to be transformed to its stream function counterpart.

As the Mach numbers with which the program works are very low, the whole expansion can be neglected as $M^2 << 1$ and everything inside the parentheses has an order of magnitude of 1 or less (what happens inside the cylinder is of no concern as it is purely mathematical and does not represent a solid). Then, using the definition of the velocity potential:

$$\begin{aligned} v_r &= \frac{\partial\phi}{\partial r} \\ v_\theta &= \frac{1}{r}\frac{\partial\phi}{\partial\theta} \end{aligned} \quad (6.6)$$

And then the definition of the stream function:

$$\begin{aligned} v_r &= \frac{1}{r}\frac{\partial\Psi}{\partial\theta} \\ v_\theta &= -\frac{\partial\Psi}{\partial r} \end{aligned} \quad (6.7)$$

The following expression is obtained:

$$\Psi = U_\infty \sin\theta\left(r-\frac{R^2}{r}\right) \quad (6.8)$$

Which is identical to the solution for incompressible flow because the Mach number is very low.

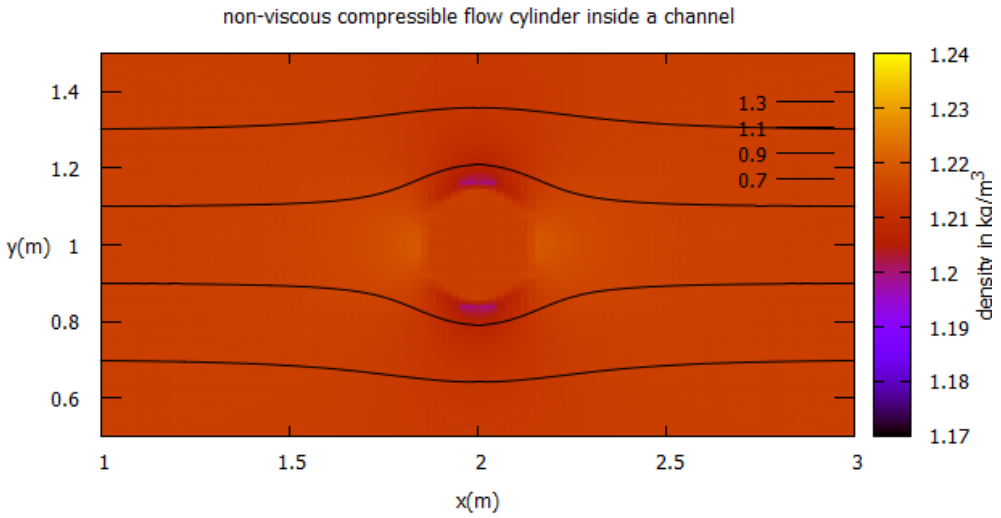


Figure 6.3: Density of the fluid around a cylinder in compressible flow with 200x200 nodes

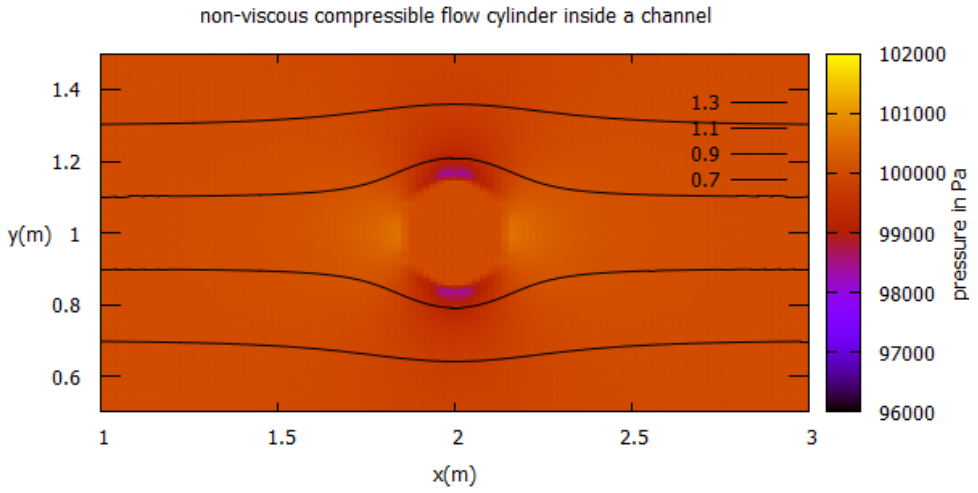


Figure 6.4: Pressure of the fluid around a cylinder in compressible flow with 200x200 nodes

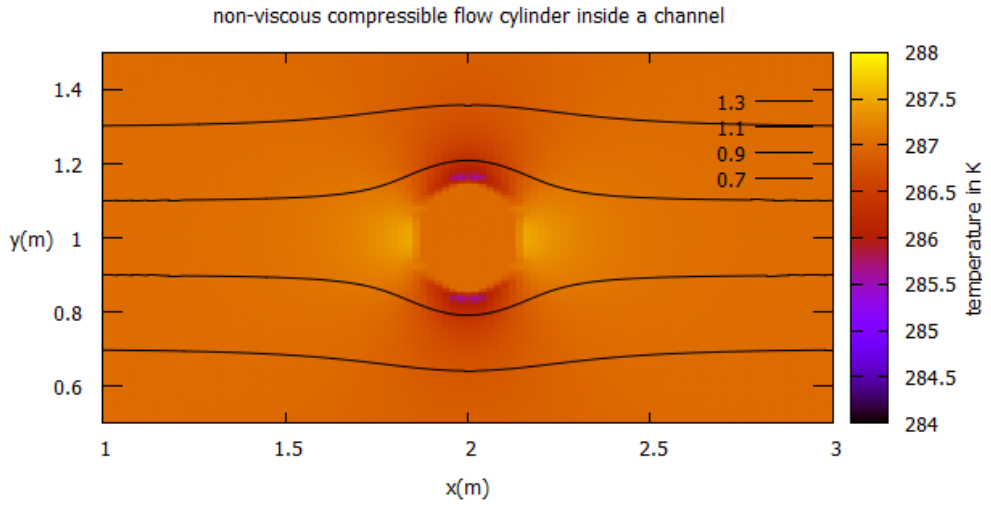


Figure 6.5: Temperature of the fluid around a cylinder in compressible flow with 200x200 nodes

In figures 6.3, 6.4 and 6.5, the variation of the physical properties of the fluid are represented. The pattern observed is very similar in the three properties. The properties increase in the front and back of the cylinder while they decrease in the top and bottom.

References

[1] L. Rayleigh, "I. On the flow of compressible fluid past an obstacle," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 32, no. 187, pp. 1–6, jul 1916. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/14786441608635539

Chapter 7

CONVECTION-DIFFUSION

Contents

In this part, the convection-diffusion equation is solved in order to develop techniques for the treatment of convective terms. For this purpose, an *Upwind-difference scheme* has been used.

7.1 Formulation of the problem

Starting from the Navier-Stokes equations for perfect gases, the generic convection-diffusion equation can be obtained:

$$\frac{\partial(\rho\phi)}{\partial t} + \nabla \cdot \left(\rho \overrightarrow{v} \phi\right) = \nabla \cdot \left(\Gamma_\phi \nabla \phi\right) + S_\phi \tag{7.1}$$

Where ϕ is a generic variable, Γ_ϕ is the diffusion coefficient and S_ϕ are the source/sink terms.

In equation 7.1, the following terms can be identified: the leftmost term is the unsteady term, immediately to its right the convective terms are located; on the right hand side, from right to left, the diffusion terms and source/sink terms are found.

In order to solve equation 7.1, it is necessary to linearise it. This is done in the following manner using an implicit scheme:

$$\int_{t^n}^{t^{n+1}} \int_{V_P} \frac{\partial(\rho\phi)}{\partial t} dV dt \approx V_P\left(\rho_P \phi_P - \rho_P^0 \phi_P^0\right) \tag{7.2}$$

$$\int_{t^n}^{t^{n+1}} \int_{V_P} \nabla \cdot \big(\rho \overrightarrow{v} \phi\big) dV dt \approx \big(\dot{m}_e \phi_e - \dot{m}_w \phi_w + \dot{m}_n \phi_n - \dot{m}_s \phi_s\big) \Delta t \tag{7.3}$$

$$\int_{t^n}^{t^{n+1}} \int_{V_P} \nabla \cdot \big(\Gamma_\phi \nabla \phi\big) dV dt \approx \left(\Gamma_e \frac{\phi_E - \phi_P}{d_{PE}} S_e - \Gamma_w \frac{\phi_P - \phi_W}{d_{PW}} S_w + \Gamma_n \frac{\phi_N - \phi_P}{d_{PN}} S_n - \Gamma_s \frac{\phi_P - \phi_S}{d_{PS}}\right) \Delta t \tag{7.4}$$

$$\int_{t^n}^{t^{n+1}} \int_{V_P} S_\phi dV dt \approx \big(S_C^\phi + S_P^\phi \phi_P\big) V_P \Delta t \tag{7.5}$$

Introducing equations 7.2, 7.3, 7.4 and 7.5 into 7.1:

$$\begin{aligned} &\frac{\rho_P \phi_P - \rho_P^0 \phi_P^0}{\Delta t} V_P + \dot{m}_e \phi_e - \dot{m}_w \phi_w + \dot{m}_n \phi_n - \dot{m}_s \phi_s \\ &= D_e\big(\phi_E - \phi_P\big) - D_w\big(\phi_P - \phi_W\big) + D_n\big(\phi_N - \phi_P\big) - D_s\big(\phi_P - \phi_S\big) + \big(S_C^\phi + S_P^\phi \phi_P\big) V_P \end{aligned} \tag{7.6}$$

Where $D_i = \Gamma_i / d_{PI}$.

In order to evaluate the convective terms, an *upwind-difference scheme* has been used:

$$\phi_e - \phi_P = f_e\big(\phi_E - \phi_P\big) \tag{7.7}$$

With $f_e = 0$ if $\dot{m}_e > 0$ and $f_e = 1$ if $\dot{m}_e < 0$.

7.2 Algorithm design

The algorithm used is very similar to the previous ones, but incorporates the *Upwind-difference scheme*.

In order to check the mass flow for the scheme, the velocity at the node has been used as an approximation.

Finally, the algorithm has been designed as:

1. input data
2. mesh generation
3. geometrical data calculation
4. initial map
5. boundary conditions
6. ϕ estimation

7. upwind scheme
8. coefficients evaluation
9. solver
10. error calculation
    - if maxError > errorAcceptance $\phi_{est} = \phi$ go to 7
    - else continue
11. $\phi_{prev} = \phi$
    - if t<tmax go to 6
    - else continue
12. final calculations

Note that instead of checking whether the code has arrived at a steady state, it keeps computing until a set time has passed. If this time is set high enough, it will arrive at a steady state and computing past the steady state adds very little computing time based on experience. Nevertheless, this will be improved later on for the Navier-Stokes equations code.

7.3 Code development

The main difference between this code and any of the other codes is that it is a transient problem, thus, it has an extra loop for computing different time steps.

Besides that, the only significant difference is the appearance of the upwind scheme function. Other differences are just related to the coefficients, boundary conditions and such as the governing equations change.

The complete code can be checked at attachment D.

7.4 Test and validation

In order to test this code, two different cases have been tested. The former is the parallel flow case, very simple but easy to check results with as an analytical solution exists. The latter is the Smith-Hutton case, characterised by a vortex-like velocity field centered on the lower face's center. In both cases, density and diffusion coefficients have been considered constant.

Starting with the parallel flow case, the results have been highly accurate with respect to the analytical solution. The analytical solution to this case is the following one:

$$\frac{\phi - \phi_{In}}{\phi_{Out} - \phi_{In}} = \frac{e^{xPe/L} - 1}{e^{Pe} - 1} \tag{7.8}$$

Where Pe is the Peclect number and $Pe = \rho v_0 L/\Gamma$.

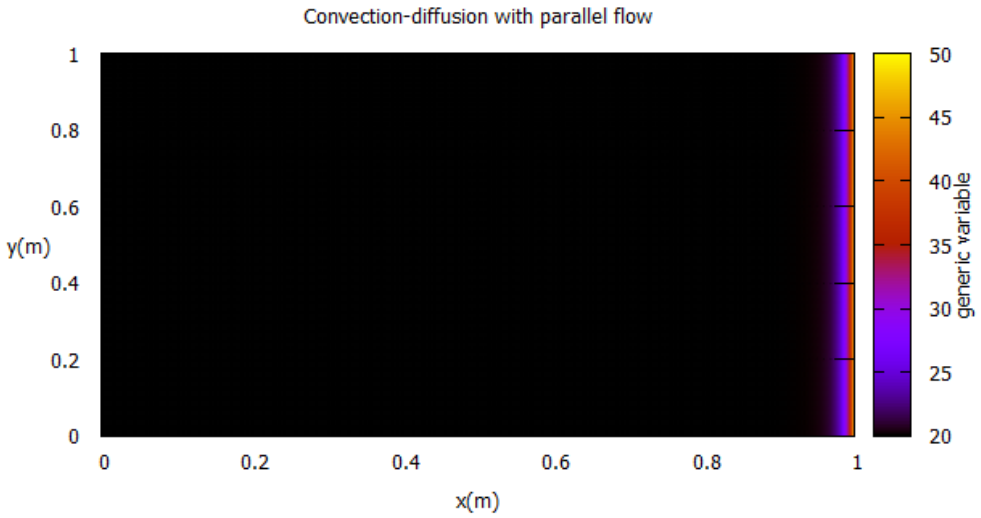


Figure 7.1: Analytical solution of the parallel flow case with Pe=100

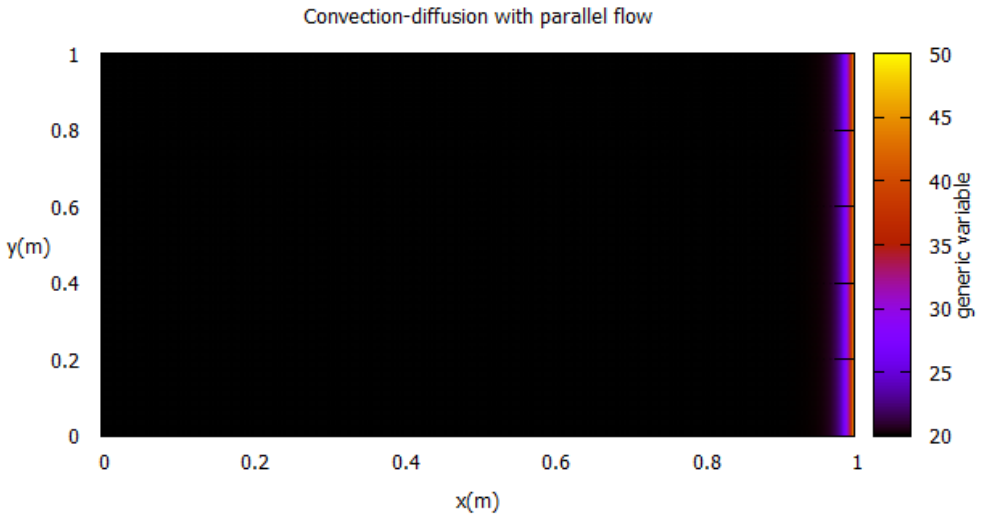


Figure 7.2: Program result of the parallel flow case with Pe=100 and 400x200 nodes

Comparing figures 7.1 and 7.2, it is clear that the results are very similar. The amount of nodes used has been determined by comparing the value of ϕ at y=0.5 m for both the analytical and the program result with different mesh densities. The amount of nodes in the y direction has been halved as this is mostly a one-dimensional case. The comparison between analytical and program results can be checked in figures 7.3, 7.4, 7.5 and 7.6.

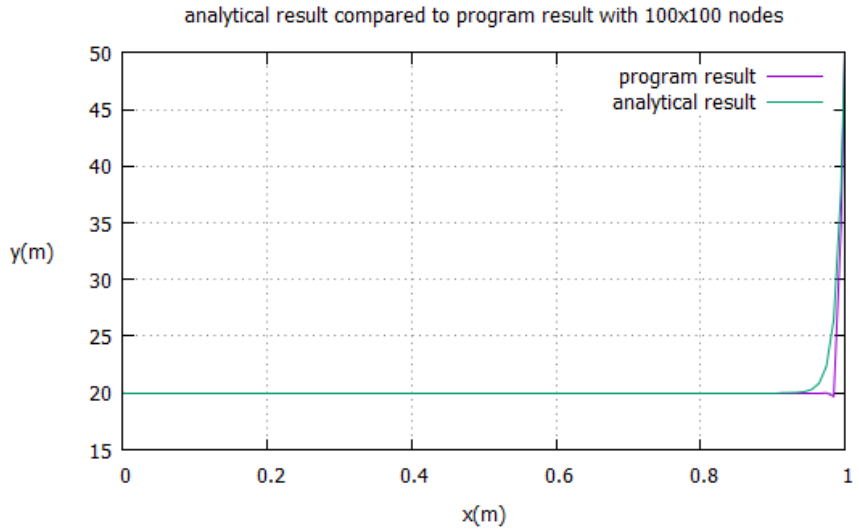


Figure 7.3: Comparison between analytical and program result for 100x100 nodes

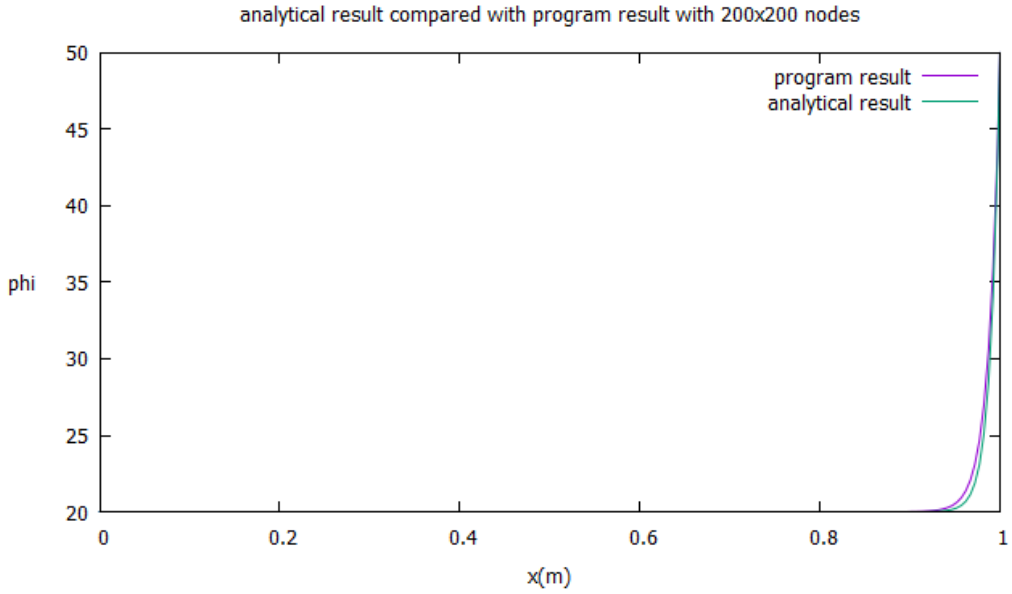


Figure 7.4: Comparison between analytical and program result for 200x200 nodes

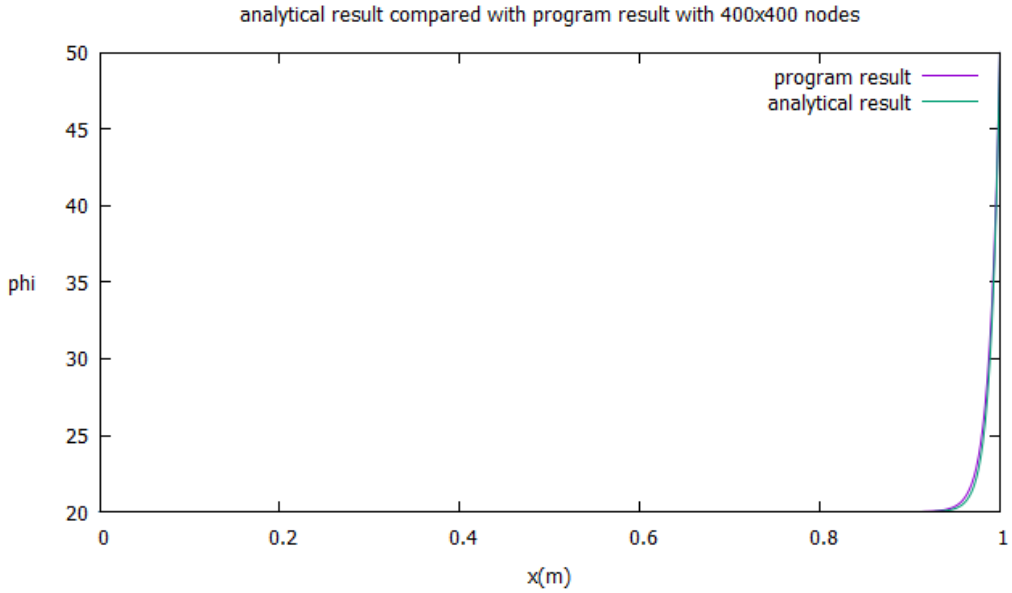


Figure 7.5: Comparison between analytical and program result for 400x400 nodes

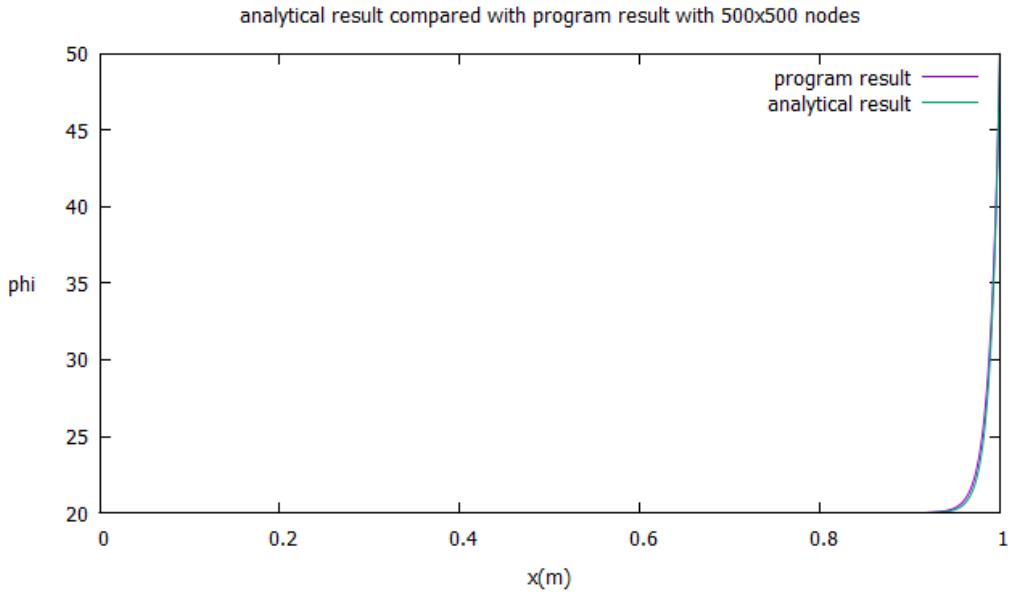


Figure 7.6: Comparison between analytical and program result for 500x500 nodes

As the difference between using 400x400 nodes and 500x500 nodes is insignificant, the chosen den-

sity has been 400x400 as the increase in precision is not worth the increase in computation time.

The second case studied was the Smith-Hutton case. For this particular problem, no analytical solution exists but the CTTC group from ESEIAAT has provided a table with results extracted from their code for different ρ/Γ values. The values from CTTC can be consulted at table 7.1

x-position (m)	ϕ value for $\rho/\Gamma = 10$
0.0	1.989
0.1	1.402
0.2	1.146
0.3	0.946
0.4	0.775
0.5	0.621
0.6	0.480
0.7	0.349
0.8	0.227
0.9	0.111
1.0	0.000

Table 7.1: CTTC results for the Smith-Hutton case

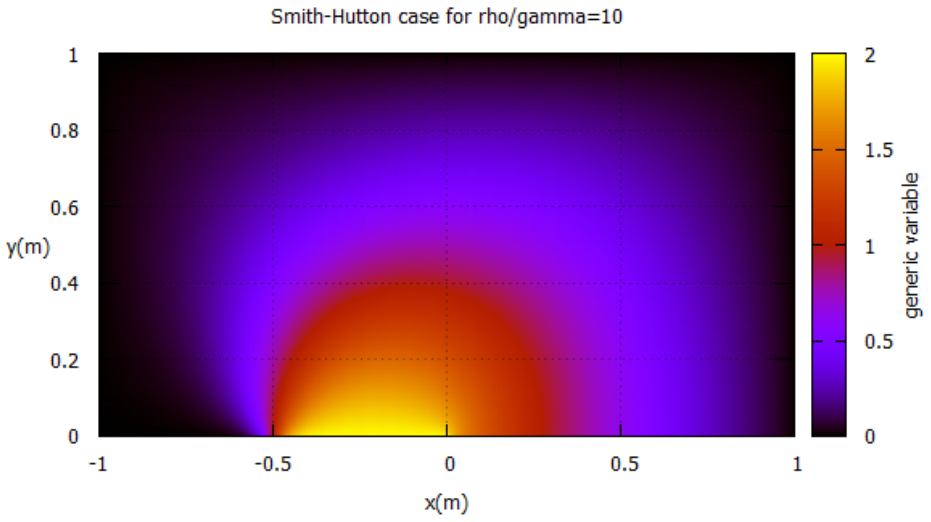


Figure 7.7: program result of the Smith-Hutton case with 400x200 nodes and $\rho/\Gamma = 10$

The result from the developed code of this particular case can be checked in figure 7.7 and the com-

parison of the values at the outlet between the CTTC results and the code results can be checked in figure 7.8.

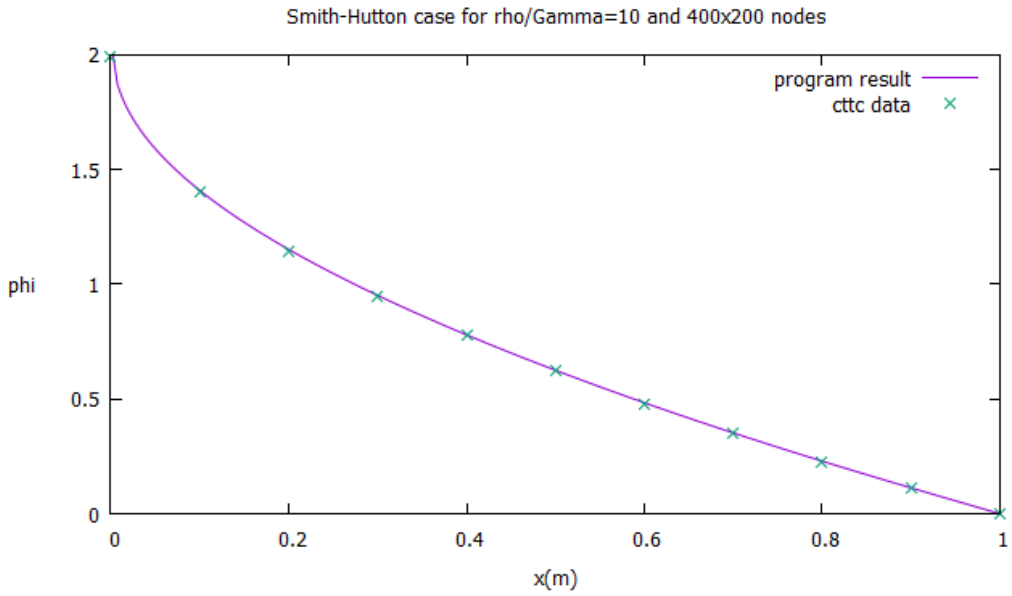


Figure 7.8: comparison between the program results and the CTTC data of the Smith-Hutton case for $\rho/\Gamma = 10$

It is clear that the developed code gives appropriate results.

Chapter 8

NAVIER-STOKES

Contents

Finally, the incompressible Navier-Stokes equations are solved using the *Fractional Step Method*. The case studied for the validation is the lid-driven cavity problem, which consists in a square cavity whose upper boundary has a prescribed horizontal velocity.

8.1 Formulation of the problem

Starting with the Navier-Stokes for incompressible fluids with constant viscosity:

$$\begin{aligned} \nabla \cdot v &= 0 \\ \rho \frac{\partial v}{\partial t} &= R(v) - \nabla p \end{aligned} \tag{8.1}$$

Where:

$$R(v) = -(\rho v \cdot \nabla)v + \mu \Delta v \tag{8.2}$$

Taking equation 8.1, its time integration becomes:

$$\begin{aligned} \nabla \cdot v^{n+1} &= 0 \\ \rho \frac{v^{n+1} - v^n}{\Delta t} &= \frac{3}{2} R(v^n) - \frac{1}{2} R(v^{n-1}) - \nabla p^{n+1} \end{aligned} \tag{8.3}$$

Note that, in equation 8.3, the continuity equation is integrated in an implicit manner whilst the momentum equation is integrated at $n^{n+1/2}$.

Using the Helmholtz-Hodge theorem, it is possible to decompose the velocity vector into a pure gradient field and a vector with divergence equal to zero and a vector parallel to the boundary:

$$v^p = v^{n+1} + \frac{\Delta t}{\rho}\nabla p^{n+1} \tag{8.4}$$

Merging equation 8.4 with the momentum equation from 8.3:

$$\rho\frac{v^P - v^n}{\Delta t} = \frac{3}{2}R(v^n) - \frac{1}{2}R(v^{n-1}) \tag{8.5}$$

Then, by applying a divergence operator to equation 8.4, keeping in mind that from equation 8.3 $\nabla \cdot v^{n+1} = 0$:

$$\Delta p^{n+1} = \frac{\rho}{\Delta t}\nabla \cdot v^P \tag{8.6}$$

Finally, using equation 8.4 with the results obtained from the Poisson equation 8.6:

$$v^{n+1} = v^P - \frac{\Delta t}{\rho}\nabla p^{n+1} \tag{8.7}$$

While the method seems straight forward, because equation 8.7 in its discretised form does not take into consideration the pressure at the current node, the velocity field obtained might be convergent while using physically impossible pressure distributions such as having a checkerboard pattern in the pressure field values.

The way to solve this issue is to use a staggered mesh setup, which is not too complex for structured meshes. With this method, the velocities are placed into different meshes whose nodes are offset to the node's boundaries of the main mesh. Doing so, the pressures used for the discretization cannot converge with physically impossible distributions.

The main mesh is, then, only used for the pressure field calculation.

With the staggered mesh setup, the x component of the velocity is placed in a staggered mesh offset to the sides whereas the y component is placed in a mesh offset to the top/bottom.

In order to introduce the equations into the program, it is necessary to discretise them. Starting from 8.5, for both components of the velocity:

$$\begin{aligned} u^P &= u^n + \frac{\Delta t}{\rho}\Big(\frac{3}{2}R(u^n) - \frac{1}{2}R(u^{n-1})\Big) \\ v^P &= u^n + \frac{\Delta t}{\rho}\Big(\frac{3}{2}R(v^n) - \frac{1}{2}R(v^{n-1})\Big) \end{aligned} \tag{8.8}$$

Where u and v are the x and y components of the velocity respectively.

Next, R needs to be discretised, this is done by integrating R(u) over the staggered-x control volume and R(v) over the staggered-y control volume:

$$R(u)V_{staggX} = -\Big((\rho u)_e u_e A_e - (\rho u)_w u_w A_w + (\rho v)_n u_n A_n - (\rho v)_s u_s A_s\Big) + \Big(\mu_e \frac{u_E - u_P}{d_{EP}} A_e - \mu_w \frac{u_P - u_W}{d_{WP}} A_w + \mu_n \frac{u_N - u_P}{d_{NP}} A_n - \mu_s \frac{u_P - u_S}{d_{SP}} A_s\Big) \tag{8.9}$$

$$R(v)V_{staggY} = -\Big((\rho u)_e v_e A_e - (\rho u)_w v_w A_w + (\rho v)_n v_n A_n - (\rho v)_s v_s A_s\Big) + \Big(\mu_e \frac{v_E - v_P}{d_{EP}} A_e - \mu_w \frac{v_P - v_W}{d_{WP}} A_w + \mu_n \frac{v_N - v_P}{d_{NP}} A_n - \mu_s \frac{v_P - v_S}{d_{SP}} A_s\Big) \tag{8.10}$$

Where E, W, N, S and P subindexes correspond to the values on the values on the nodes on the right, left, top, bottom and itself respectively. Subindexes in lower case correspond to the boundaries.

In order to evaluate the velocities and the mass fluxes in the faces of the nodes, the following schemes are used:

- For velocities perpendicular to the boundary, an *upwind-difference scheme* has been used
- For velocities parallel to the boundaries, a simple arithmetic average has been taken between two nodes' velocities.
- For mass fluxes, an average has been taken between two nodes.

Then, the Poisson equation is discretised for the main mesh:

$$\frac{p_E^{n+1} - p_P^{n+1}}{d_{EP}} A_e - \frac{p_P^{n+1} - p_W^{n+1}}{d_{WP}} A_w + \frac{p_N^{n+1} - p_P^{n+1}}{d_{NP}} A_n - \frac{p_P^{n+1} - p_S^{n+1}}{d_{SP}} A_s = \frac{1}{\Delta t}\big((\rho u^P)_e A_e - (\rho u^P)_w A_w + (\rho v^P)_n A_n - (\rho v^P)_s A_s\big) \tag{8.11}$$

From equation 8.11, the following coefficients are obtained:

$$\begin{aligned} a_E &= \frac{A_e}{d_{EP}} \\ a_W &= \frac{A_w}{d_{WP}} \\ a_N &= \frac{A_n}{d_{NP}} \\ a_S &= \frac{A_s}{d_{SP}} \\ a_P &= a_E + a_W + a_N + a_S \\ b_P &= -\frac{1}{\Delta t}\big((\rho u^P)_e A_e - (\rho u^P)_w A_w + (\rho v^P)_n A_n - (\rho v^P)_s A_s\big) \end{aligned} \tag{8.12}$$

Once the coefficients are obtained, any linear solver can be used to obtain the pressure field. In this case, the line-by-line solver has been used.

In order to solve equation 8.11, boundary conditions must be set. for both the walls and the lid-driven boundary, the condition is $\frac{\partial p}{\partial n} = 0$. However, it is necessary to fix the pressure of one node on the inside to an arbitrary value (the value is not relevant but the gradient), otherwise the solver might not be able to converge.

For the boundary conditions applied to the velocities, in the walls they are zero while in the lid-driven boundary they are fixed at the reference value.

The last step is to discretise equation 8.7 in the staggered meshes:

$$\begin{aligned} u_P^{n+1} &= u_P^P - \frac{\Delta t}{\rho} \frac{P_B^{n+1} - P_A^{n+1}}{d_{BA}} \\ v_P^{n+1} &= v_P^P - \frac{\Delta t}{\rho} \frac{P_B^{n+1} - P_A^{n+1}}{d_{BA}} \end{aligned} \tag{8.13}$$

Where the subindexes B and A correspond to the nodes in the main mesh immediately upstream and downstream from the node considered in the staggered mesh.

8.2 Algorithm design

For this part, some parts of the code have been improved in an attempt to make the code faster as computing the Navier-Stokes equations is more intensive than the previous cases considered.

The solver has remained the same though, as it has been proven to work well.

The preprocessor has been slightly modified in order to also generate the staggered meshes.

One important step that was not present in the other codes is the addition of a time step selection routine. For convergence reasons, according to the *Courant-Friedrich-Levy condition*, the time step must be adaptive following the next criteria:

$$\Delta t = min(\Delta tc, \Delta t_d) \tag{8.14}$$

Where:

$$\begin{aligned} \Delta t_c &= min\left(0.35 \frac{\Delta x}{|v|}\right) \\ \Delta t_d &= min\left(0.2 \frac{\rho \Delta x^2}{\mu}\right) \end{aligned} \tag{8.15}$$

If condition 8.14 is not met, a steady state solution is not guaranteed.

With all that in mind, the algorithm designed is the following one:

1. input data
2. mesh generation
3. initial map
4. velocity boundary conditions
5. time step selection
6. R evaluation
7. v^P evaluation
8. pressure initial estimation
9. coefficient calculation
10. pressure boundary conditions
11. line-by-line solver
12. error calculation
    - if maxError > errorAcceptance $P_{est} = P$ go to 9
    - else continue
13. v^{n+1} calculation
14. steady state check
    - if maxErrorTime > errorAcceptance and t < tmax $P_{prev} = P$, $v^{n-1} = v^n$, $v^n = v^{n+1}$, $R^{n-1} = R^n$ go to 5
    - else continue
15. final calculations

Note that on the time loop a second condition has been imposed. This has been done in order to prevent the program to loop forever in case there is no steady state solution. The value tmax must be set high enough to allow sufficient time for the solution to converge.

8.3 Code development

For the development of this code, every piece of knowledge obtained during the making of this project has been used to make the code more efficient than the previous codes while still maintaining the base as it has been proven to work.

One small but significant difference between this code and the convection-diffusion code is that the convective scheme is outside the iterative part of the solver as the velocities do not need to be iterated.

The whole code can be checked at attachment E.

8.4 Test and validation

The program has been validated using the data provided by the CTTC group. Three different Reynolds numbers have been tested with different mesh densities. Those are Re=100, Re=400 and Re=1000. The data from the CTTC can be consulted in tables 8.1 and 8.2.

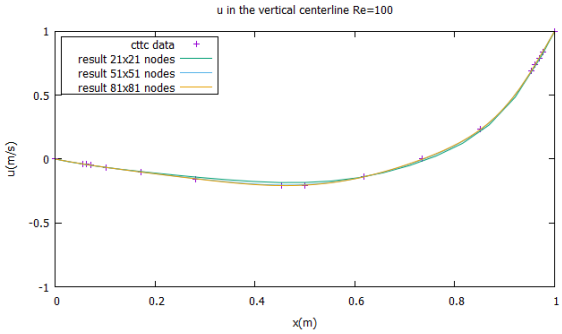
y-position (m)	Re=100	Re=400	Re=1000
1.0000	1.00000	1.00000	1.00000
0.9766	0.84123	0.75837	0.65928
0.9688	0.78871	0.68439	0.57492
0.9609	0.73722	0.61756	0.51117
0.9531	0.68717	0.55892	0.46604
0.8516	0.23151	0.29093	0.33304
0.7344	0.00332	0.16256	0.18719
0.6172	-0.13641	0.02135	0.05702
0.5000	-0.20581	-0.11477	-0.06080
0.4531	-0.21090	-0.17119	-0.10648
0.2813	-0.15662	-0.32726	-0.27805
0.1719	-0.10150	-0.24299	-0.38289
0.1016	-0.06434	-0.14612	-0.29730
0.0703	-0.04775	-0.10338	-0.22220
0.0625	-0.04192	-0.09266	-0.20196
0.0547	-0.03717	0.08186	-0.18109
0.0000	0.00000	0.00000	0.00000

Table 8.1: CTTC results for u in the vertical center line for the lid-driven cavity

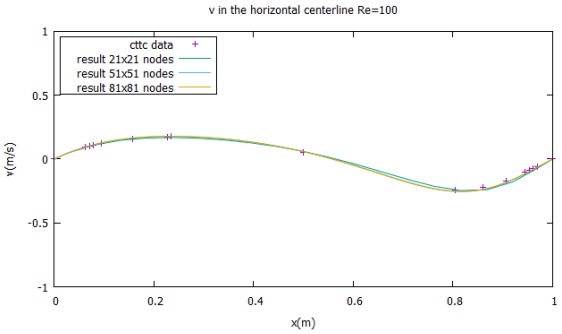
y-position (m)	Re=100	Re=400	Re=1000
1.0000	1.00000	1.00000	1.00000
0.9766	0.84123	0.75837	0.65928
0.9688	0.78871	0.68439	0.57492
0.9609	0.73722	0.61756	0.51117
0.9531	0.68717	0.55892	0.46604
0.8516	0.23151	0.29093	0.33304
0.7344	0.00332	0.16256	0.18719
0.6172	-0.13641	0.02135	0.05702
0.5000	-0.20581	-0.11477	-0.06080
0.4531	-0.21090	-0.17119	-0.10648
0.2813	-0.15662	-0.32726	-0.27805
0.1719	-0.10150	-0.24299	-0.38289
0.1016	-0.06434	-0.14612	-0.29730
0.0703	-0.04775	-0.10338	-0.22220
0.0625	-0.04192	-0.09266	-0.20196
0.0547	-0.03717	0.08186	-0.18109
0.0000	0.00000	0.00000	0.00000

Table 8.2: CTTC results for u in the vertical center line for the lid-driven cavity

As it can be seen in figures 8.1a, 8.1b, 8.2a, 8.2b, 8.3a and 8.3b the solution converges to the CTTC's data as the grid becomes denser, which was expected.

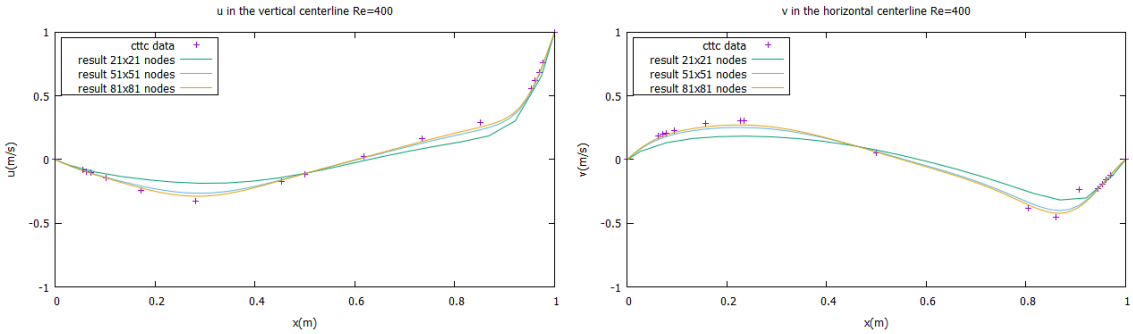


(a) u in the horizontal centerline



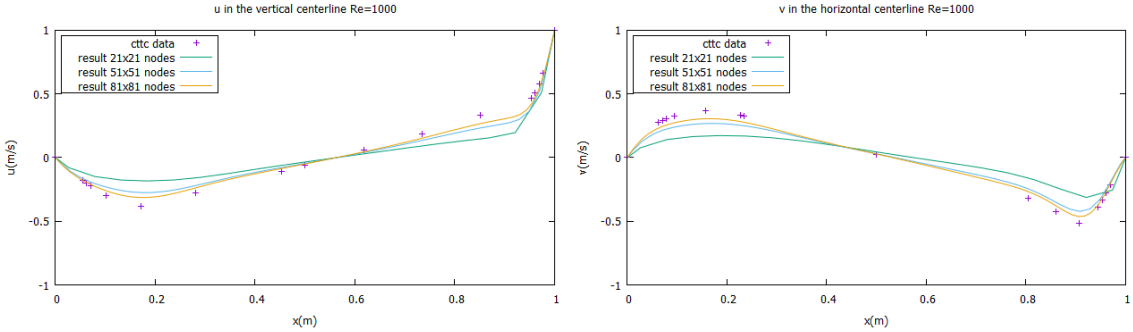
(b) v in the vertical centerline

Figure 8.1: Comparison between the program's result and the cttc data for Re=100



(a) u in the horizontal centerline (b) v in the vertical centerline

Figure 8.2: Comparison between the program's result and the cttc data for Re=400



(a) u in the horizontal centerline (b) v in the vertical centerline

Figure 8.3: Comparison between the program's result and the cttc data for Re=1000

Note that for Re=100, less nodes are required in order to achieve the correct result whereas for Re=1000 the solution has the biggest error of the three. This is due to the convection scheme used. *Upwind-difference schemes* are not good approximations for meshes that are not very dense. This problem gets accentuated as the Re increases due the larger velocity gradients present.

The results obtained can be checked at figures 8.4, 8.5 and 8.6.

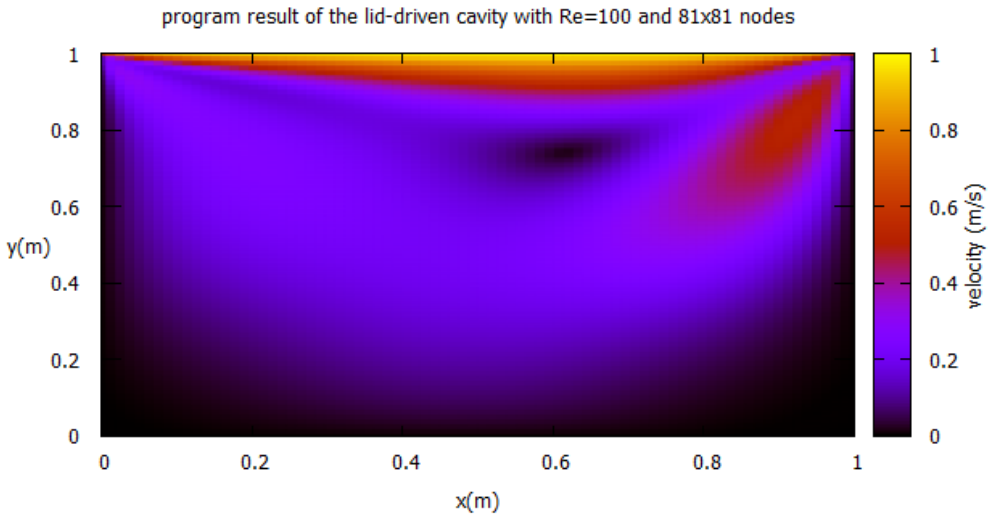


Figure 8.4: program result of the lid-driven cavity with Re=100 and 81x81 nodes

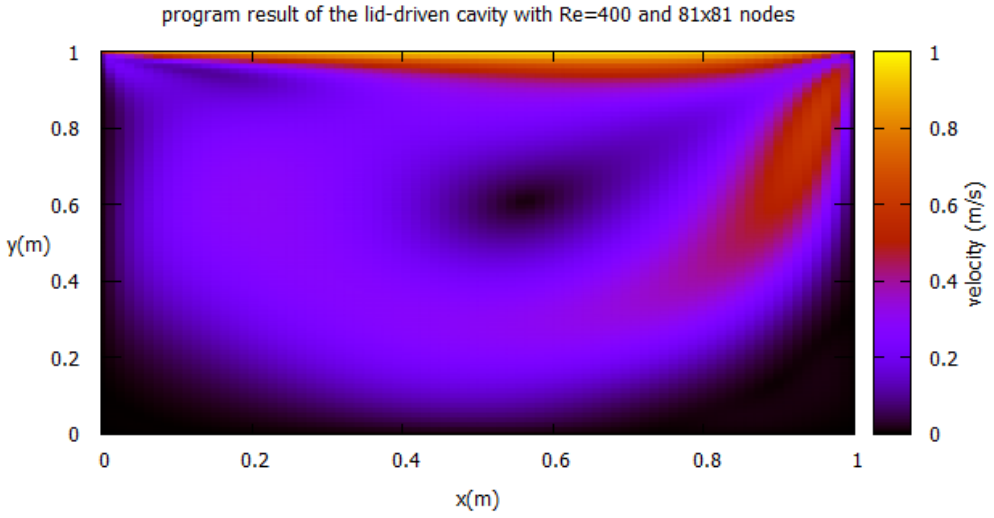


Figure 8.5: program result of the lid-driven cavity with Re=400 and 81x81 nodes

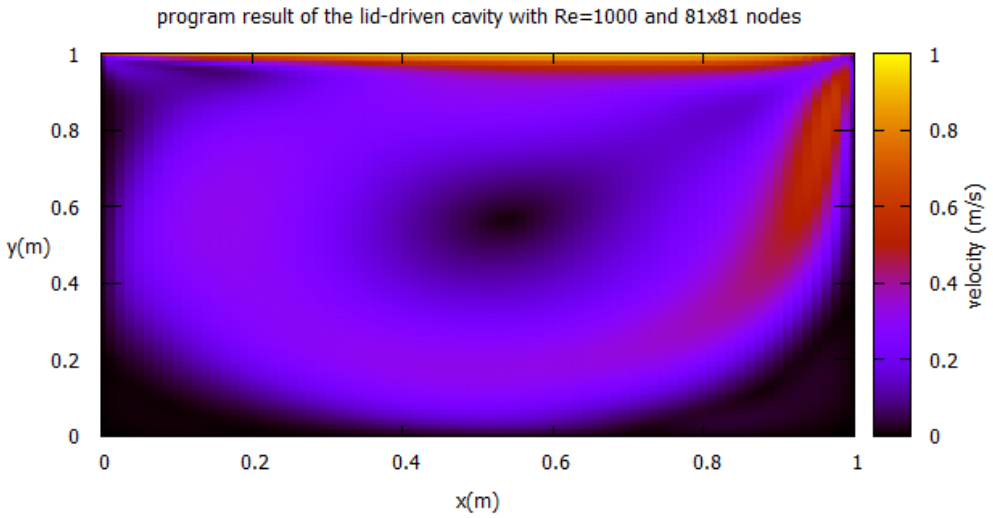
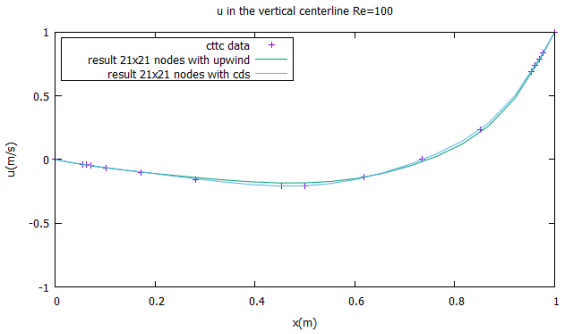
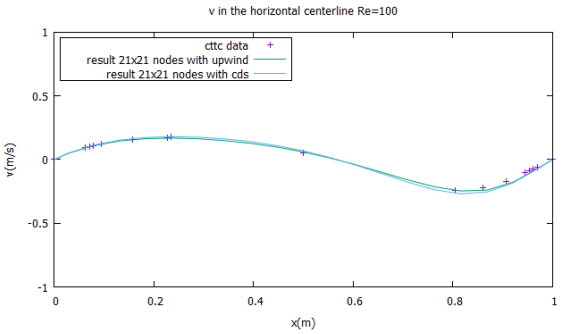


Figure 8.6: program result of the lid-driven cavity with Re=1000 and 81x81 nodes

All those result have been obtained using an *upwind-difference scheme*, which is not very precise. In figures 8.7a, 8.7b, 8.8a, 8.8b, 8.9a and 8.9b a comparison between *upwind-difference scheme* and *central-difference scheme* can be seen. The comparison has been done with only 21x21 nodes in order to make the differences more apparent. Note that the *central-difference scheme* is more precise overall. The reason why *upwind-difference* has been chosen over *central-difference* is a matter of stability. With a *central-difference scheme* convergence is not guaranteed while an *upwind-difference scheme* is perfectly stable.

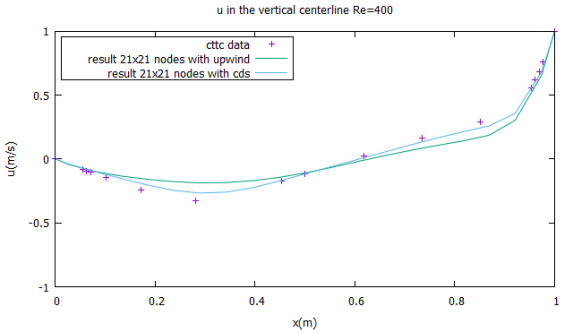


(a) u in the horizontal centerline

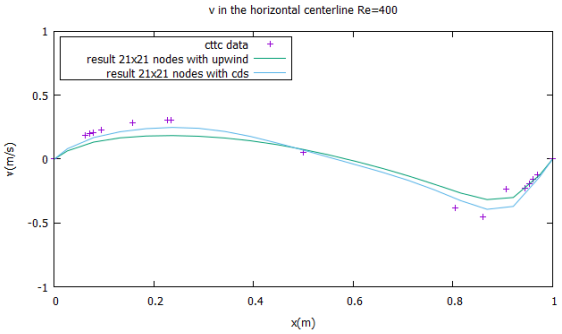


(b) v in the vertical centerline

Figure 8.7: Comparison between *upwind-difference* and *central-difference* with Re=100 and 21x21 nodes

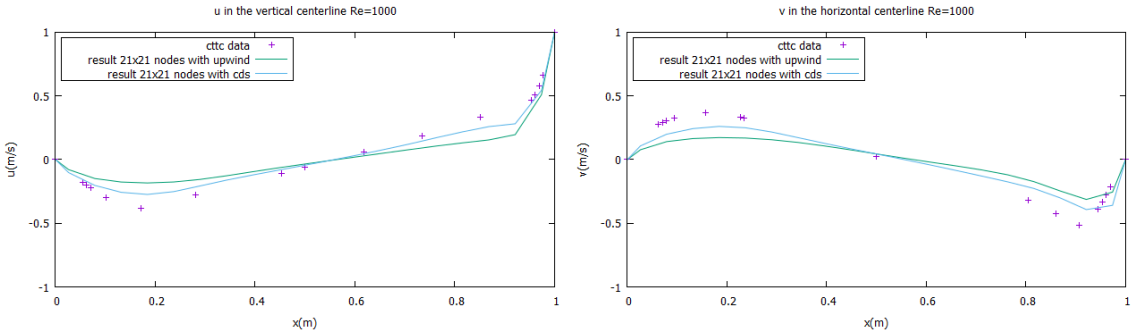


(a) u in the horizontal centerline



(b) v in the vertical centerline

Figure 8.8: Comparison between *upwind-difference* and *central-difference* with Re=400 and 21x21 nodes



(a) u in the horizontal centerline

(b) v in the vertical centerline

Figure 8.9: Comparison between *upwind-difference* and *central-difference* with Re=1000 and 21x21 nodes

Chapter 9

BUDGET AND ENVIRONMENTAL IMPACT

Contents

In this section, the total cost and environmental impact of the project will be discussed.

9.1 Budget

Computing the cost for a project based on programming is pretty straight forward as the only two variables are the salary of the programmer and the cost of electricity.

For the salary, it has been assumed that the qualification of the programmer is that of a scientific professional which according to [1], has an average salary for a male of about 23.86 €/h.

As for the electricity cost, the computer used for the project is an *HP Pavilion Notebook*, which according to the manufacturer [2], has a power supply of 45W. The cost of the KWh has been determined with [3] and it is about 0.12 €/KWh as a rough average. Thus the cost per hour is 0.0054 €/h.

The amount of hours used for this project are 600 hours in all of them the computer has been working.

The project's budget can be consulted at table 9.1.

Concept	Unit cost (€/h)	Hours	Total cost (€)
Salary	23.86	600	14316
Electricity	0.0054	600	3.24
Total	-	-	14319.24

Table 9.1: Project's budget

9.2 Environmental impact

The environmental impact of this project is very low as the only factor is the electricity used. According to [4], the CO_2 emissions can be taken as 0.16 t/MWh as a rough average.

As for what radioactive waste is concerned, according to [5], 0.00213 cm^3/KWh of nuclear waste with low/medium activity and 0.26 mg/KWh of nuclear waste with high activity are produced. Those values are from 2016, but as the amount of nuclear power generated has remained the same, it is possible to extrapolate these values.

In table 9.2,the environmental impact of the project can be consulted.

Pollutant	Emissions (per KWh)	Energy expense (KWh)	Total
CO_2	0.00016 t	27	0.00432 t
Low/medium activity nuclear waste	0.00213 cm^3	27	0.05751 cm^3
High activity nuclear waste	0.26 mg	27	7.02 mg

Table 9.2: Project's environmental impact

References

[1] Institut d'Estadística de Catalunya, "Idescat. Indicadors anuals. Salari brut anual i guany per hora. Per sexe i tipus d'ocupació." 2016. [Online]. Available: https://www.idescat.cat/indicadors/?id=anuals{&}n=10403

[2] HP, "HP Pavilion 15 — HP® Official Store." [Online]. Available: https://store.hp.com/us/en/mdp/pavilion-15-344522--1{#}!{&}tab=vao

[3] Red Eléctrica de España, "PVPC — ESIOS electricidad · datos · transparencia." [Online]. Available: https://www.esios.ree.es/es/pvpc

[4] ——, "Demanda de energía eléctrica en tiempo real, estructura de generación y emisiones de CO2." [Online]. Available: https://demanda.ree.es/visiona/peninsula/demanda/total/2019-06-04

[5] WWF, "Observatorio de la Electricidad mayo 2016," Tech. Rep. [Online]. Available: https://d80g3k8vowjyp.cloudfront.net/downloads/oe{_}mayo{_}2016.pdf

Chapter 10

CONCLUSIONS

Contents

10.1 Conclusions

The fractional-step method is an effective way of computing the incompressible Navier-Stokes equations, however, the current state of the art is very far away from the work done for this project.

This is not an issue as the objective of this thesis was not to innovate in the field, but rather to learn the basics of CFD so that in the future a more effective, efficient and complete program can be developed.

The results obtained meet the expectations, the only issue is the circulation calculation in the blocking-off and comrpessible codes but that is a minor problem given the fact that it did not affect results or any of the other codes.

As for the convective schemes used, the *upwind-difference scheme* is the most basic of them all but it allows for an easy implementation and is stable under any situation. The only issue with that scheme is that it requires a denser mesh to give the same results than the others. The *central-difference scheme* is also an easy scheme to implement but is not as stable, hence the choice was *upwind*.

10.2 Recommendations and future studies

As said before, the complexity of this thesis is not up to date with the current state of the art. Thus, the subject can be more thoroughly studied and improved.

For instance, a possible upgrade would be the use of unstructured adaptive meshes which would increase both the precision and efficiency of the program at the cost of complexity.

Other possible expansions of the code might include but are not limited to: 3D implementation, compressibility for the Navier-Stokes equations or the capacity to compute supersonic flow and shockwaves.

As for what convective schemes are concerned, the ones used for this project are the most basic of them all and a possible code enhancement would be the addition of other, most efficient and precise, convective schemes such as *QUICK*.

Attachment A

POTENTIAL FLOW CODE

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <stdlib.h>
#include <math.h>

using namespace std;

//number of nodes
const int nx=100, ny=100;
//relative error acceptance
const double delta=1.0/(100*(nx*ny));
//geometric definition in meters (m)
const double h=1, l=1;
//free stream velocity in meters per second (m/s)
const double vinf=1;

//differential lengths in meters (m)
double dx, dy;
//nodes positions [dimension][nx][ny] for dimension 0) x coord. 1) y coord.
double pos[2][nx][ny];
//nodes geometrical data [parameter][nx][ny] for parameters 0) Sx 1) Sy
double geo[2][nx][ny];
//streamlines (+2 to include "ghost" nodes around the geometry
              //for computational reasons)
double phi[nx+2][ny+2];
//TDMA coefficients 0) ap 1) ae 2) aw 3) an 4) as 5)bp
double coef[6][nx][ny];
//error matrix
double errMat[nx][ny];
double phiprev[nx+2][ny+2];
```

```
double bstarx[nx], bstary[ny];
double Prow[nx+1], Qrow[nx+1];
double Pcolumn[ny+1], Qcolumn[ny+1];

void preprocessor(double& dx, double& dy, double pos[2][nx][ny],
                  double geo[2][nx][ny])
{
    dx=l/(nx-2);
    dy=h/(ny-2);
    pos[0][0][0]=0; //corner nodes positions and geometry
    pos[1][0][0]=0;
    geo[0][0][0]=0;
    geo[1][0][0]=0;
    pos[0][nx-1][0]=l;
    pos[1][nx-1][0]=0;
    geo[0][nx-1][0]=0;
    geo[1][nx-1][0]=0;
    pos[0][nx-1][ny-1]=l;
    pos[1][nx-1][ny-1]=h;
    geo[0][nx-1][ny-1]=0;
    geo[1][nx-1][ny-1]=0;
    pos[0][0][ny-1]=0;
    pos[1][0][ny-1]=h;
    geo[0][0][ny-1]=0;
    geo[1][0][ny-1]=0;
    for (int i=1; i<=(nx-2); i++) //top and bottom boundaries
    {
        pos[0][i][0]=0.5*dx+(i-1)*dx;
        pos[1][i][0]=0;
        geo[0][i][0]=dx;
        geo[1][i][0]=0;
        pos[0][i][ny-1]=0.5*dx+(i-1)*dx;
        pos[1][i][ny-1]=h;
        geo[0][i][ny-1]=dx;
        geo[1][i][ny-1]=0;
    }
    for (int j=1; j<=(ny-2); j++) //side boundaries
    {
        pos[0][0][j]=0;
        pos[1][0][j]=0.5*dy+(j-1)*dy;
        geo[0][0][j]=0;
        geo[1][0][j]=dy;
        pos[0][nx-1][j]=l;
        pos[1][nx-1][j]=0.5*dy+(j-1)*dy;
        geo[0][nx-1][j]=0;
        geo[1][nx-1][j]=dy;
```

```
    }
    for (int i=1; i<=(ny-2); i++) //inner nodes
    {
        for (int j=1; j<=(nx-2); j++)
        {
            pos[0][i][j]=pos[0][i][0];
            pos[1][i][j]=pos[1][0][j];
            geo[0][i][j]=geo[0][i][0];
            geo[1][i][j]=geo[1][0][j];
        }
    }

}

void initialEstimation(double phi[nx+2][ny+2])
{
    for (int i=0; i<=(nx+1); i++)
    {
        for (int j=0; j<=(ny+1); j++)
        {
            phi[i][j]=0;
        }
    }
}

void coefCalc(double coef[6][nx][ny],double geo[2][nx][ny],
              double pos[2][nx][ny])
{
    for (int i=1; i<=(nx-2); i++) //TDMA coefficients calculation inner nodes
    {
        for (int j=1; j<=(ny-2); j++)
        {
            coef[1][i][j]=geo[1][i][j]/(pos[0][i+1][j]-pos[0][i][j]);
            coef[2][i][j]=geo[1][i][j]/(pos[0][i][j]-pos[0][i-1][j]);
            coef[3][i][j]=geo[0][i][j]/(pos[1][i][j+1]-pos[1][i][j]);
            coef[4][i][j]=geo[0][i][j]/(pos[1][i][j]-pos[1][i][j-1]);
            coef[0][i][j]=coef[1][i][j]+coef[2][i][j]+coef[3][i][j]
            +coef[4][i][j];
            coef[5][i][j]=0;
        }
    }

    coef[0][0][0]=1;
    coef[1][0][0]=0;
    coef[2][0][0]=0;
    coef[3][0][0]=0;
```

```
    coef[4][0][0]=0;
    coef[5][0][0]=0;
    coef[0][nx-1][0]=1;
    coef[1][nx-1][0]=0;
    coef[2][nx-1][0]=0;
    coef[3][nx-1][0]=0;
    coef[4][nx-1][0]=0;
    coef[5][nx-1][0]=0;
    coef[0][nx-1][ny-1]=1;
    coef[1][nx-1][ny-1]=0;
    coef[2][nx-1][ny-1]=0;
    coef[3][nx-1][ny-1]=0;
    coef[4][nx-1][ny-1]=0;
    coef[5][nx-1][ny-1]=0;
    coef[0][0][ny-1]=1;
    coef[1][0][ny-1]=0;
    coef[2][0][ny-1]=0;
    coef[3][0][ny-1]=0;
    coef[4][0][ny-1]=0;
    coef[5][0][ny-1]=0;

    coef[3][1][0]=1; //recalculation due to an unknown glitch
}

void conditions(double phi[nx+2][ny+2], double coef[6][nx][ny])
{
    for (int i=1; i<=(nx-1); i++) //top and bottom conditions
    {
        coef[5][i][0]=0;
        coef[4][i][0]=0;
        coef[3][i][0]=0;
        coef[2][i][0]=0;
        coef[1][i][0]=0;
        coef[0][i][0]=1;
        coef[5][i][ny-1]=vinf*h;
        coef[1][i][ny-1]=0;
        coef[2][i][ny-1]=0;
        coef[3][i][ny-1]=0;
        coef[4][i][ny-1]=0;
        coef[0][i][ny-1]=1;
    }
    for (int j=0; j<=(ny-1); j++) //inlet conditions
    {
        coef[5][0][j]=vinf*pos[1][0][j];
        coef[0][0][j]=1;
        coef[1][0][j]=0;
```

```
        coef[2][0][j]=0;
        coef[3][0][j]=0;
        coef[4][0][j]=0;
    }
    for (int j=1; j<=(ny-2); j++) //outlet conditions
    {
        coef[1][nx-1][j]=0;
        coef[2][nx-1][j]=1;
        coef[3][nx-1][j]=0;
        coef[4][nx-1][j]=0;
        coef[0][nx-1][j]=1;
        coef[5][nx-1][j]=0;
    }
    coef[0][0][ny-1]=1; //for computational reasons
    coef[0][nx-1][ny-1]=1;
    coef[0][0][0]=1;
    coef[0][nx-1][0]=1;


}

void lineByLineRow(double bstarx[nx], double coef[6][nx][ny],
                   double phi[nx+2][ny+2], int j)
{
    for (int i=0; i<=(nx-1); i++)
    {
        bstarx[i]=coef[5][i][j]+coef[3][i][j]*phi[i+1][j+2]
        +coef[4][i][j]*phi[i+1][j];
    }
}

void TDMArow(double phi[nx+2][ny+2], double coef[6][nx][ny], double bstarx[nx],
             int j,double Prow[nx+1],double Qrow[nx+1])
{
    Prow[0]=0;
    Qrow[0]=0;
    for (int a=1; a<=nx; a++)
    {
        Prow[a]=coef[1][a-1][j]/(coef[0][a-1][j]-coef[2][a-1][j]*Prow[a-1]);
        Qrow[a]=(bstarx[a-1]+coef[2][a-1][j]*Qrow[a-1])/(coef[0][a-1][j]
                                                -coef[2][a-1][j]*Prow[a-1]);
    }
    for (int a=(nx); a>=1; a--)
    {
        phi[a][j+1]=Prow[a]*phi[a+1][j+1]+Qrow[a];
    }
```

```
}

void lineByLineColumn(double bstary[ny], double coef[6][nx][ny],
                      double phi[nx+2][ny+2], int i)
{
    for (int j=0; j<=(ny-1); j++)
    {
        bstary[j]=coef[5][i][j]+coef[2][i][j]*phi[i][j+1]
        +coef[1][i][j]*phi[i+2][j+1];
    }

}

void TDMAcolumn(double phi[nx+2][ny+2], double coef[6][nx][ny],
                double bstary[ny], int i,double Pcolumn[ny+1],
                 double Qcolumn[ny+1])
{
    Pcolumn[0]=0;
    Qcolumn[0]=0;
    for (int a=1; a<=ny; a++)
    {
        Pcolumn[a]=coef[3][i][a-1]/(coef[0][i][a-1]
                                    -coef[4][i][a-1]*Pcolumn[a-1]);
        Qcolumn[a]=(bstary[a-1]+coef[4][i][a-1]*Qcolumn[a-1])/(coef[0][i][a-1]
                                                    -coef[4][i][a-1]*Pcolumn[a-1]);
    }
    for (int a=(ny); a>=1; a--)
    {
        phi[i+1][a]=Pcolumn[a]*phi[i+1][a+1]+Qcolumn[a];
    }
}

double abs(double value)
{
    if(value<0)
        value=-value;
    return value;
}

void errCalc(double phi[nx+2][ny+2],double phiprev[nx+2][ny+2],
             double errMat[nx][ny])
{
    double diff;
    for (int i=1; i<=(nx); i++)
    {
        for (int j=1; j<=(ny); j++)
```

```
        {
            diff=phi[i][j]-phiprev[i][j];
            errMat[i-1][j-1]=(abs(diff))/phiprev[i][j];
        }
    }

}

double findMax(double errMat[nx][ny])
{
    double err=0;
    for(int i=0; i<=(nx-1); i++)
    {
        for(int j=0; j<=(ny-1); j++)
        {
            if(errMat[i][j]>err)
                err=errMat[i][j];
        }
    }
    return err;
}

void solver(double pos[2][nx][ny],double geo[2][nx][ny],
            double phi[nx+2][ny+2],double coef[6][nx][ny],double errMat[nx][ny],
            double bstarx[nx],double bstary[ny],double Prow[nx+1],
            double Qrow[nx+1],double Pcolumn[ny+1],double Qcolumn[ny+1])
{
    double maxErr=10;
    coefCalc(coef,geo,pos);
    conditions(phi,coef);


    while (maxErr>delta)
    {
        //saving previous phi for later error calculation
        for (int i=0; i<=(nx+1); i++)
        {
            for (int j=0; j<=(ny+1); j++)
            {
                phiprev[i][j]=phi[i][j];
            }
        }

        for (int j=0; j<=(ny-1); j++) //by rows
        {
            //turns a 2D problem into a 1D problem
```

```
            lineByLineRow(bstarx,coef,phi,j);
            TDMArow(phi,coef,bstarx,j,Prow,Qrow);
        }
        for (int i=0; i<=(nx-1); i++) //by columns
        {
            //turns a 2D problem into a 1D problem
            lineByLineColumn(bstary,coef,phi,i);
            TDMAcolumn(phi,coef,bstary,i,Pcolumn,Qcolumn);
        }
        errCalc(phi,phiprev,errMat);
        maxErr=findMax(errMat);

        cout << "err=" << maxErr << endl;
    }
}

void postprocessor(double pos[2][nx][ny],double phi[nx+2][ny+2])
{
    string filename="heatmap.dat";
    ofstream datafile (filename.c_str());
    for(int i=0; i<=(nx-1); i++)
    {
        for(int j=0; j<=(ny-1); j++)
        {
            datafile << pos[0][i][j] << " " << pos[1][i][j] << " "
            << phi[i+1][j+1] << endl;
        }
        datafile << endl;
    }
    datafile.close();
}

int main()
{
    preprocessor(dx,dy,pos,geo);
    initialEstimation(phi);
    solver(pos,geo,phi,coef,errMat,bstarx,bstary,Prow,Qrow,Pcolumn,Qcolumn);
    postprocessor(pos,phi);
}
```

Attachment B

BLOCKING-OFF CODE

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <stdlib.h>
#include <math.h>

using namespace std;

//number of nodes
const int nx=200, ny=200;
 //relative error acceptance
const double delta=1.0/(100*(nx*ny));
//geometric definition in meters (m)
const double h=1.0, l=2.0;
//free stream velocity in meters per second (m/s)
const double vinf=1.0;
//air density at inlet in kilograms per meter cubed (Kg/m^3)
const double rho0=1.225;
//cylinder radius in meters (m);
const double R=0.15;
//absolute error allowed in solid circulation
const double deltaCirc=0.000001;
//requested circulation around the solid
const double circCondition=0.0;

//differential lengths in meters (m)
double dx, dy;
//nodes positions [dimension][nx][ny] for dimension 0) x coord. 1) y coord.
double pos[2][nx][ny];
//nodes geometrical data [parameter][nx][ny] for parameters 0) Sx 1) Sy
double geo[2][nx][ny];
```

```
//streamlines (+2 to include "ghost" nodes around the geometry for computational
                // reasons)
double phi[nx+2][ny+2];
//TDMA coefficients 0) ap 1) ae 2) aw 3) an 4) as 5)bp
double coef[6][nx][ny];
//error matrix
double errMat[nx][ny];
double phiprev[nx+2][ny+2];
double bstarx[nx], bstary[ny];
double Prow[nx+1], Qrow[nx+1];
double Pcolumn[ny+1], Qcolumn[ny+1];
//nodes densities
double rho[nx][ny];
//blocking-off matrix (1=solid;0=fluid)
double blocking[nx][ny];
//solid stream function value
double circPsi=0.5;
//previous solid stream function value
double circPsiPrev=10.0;
//previous circulation around the solid
double circPrev=1.0;
//circulation around the solid
double circ=0.0;
double errCirc=10;
double maxErr=10;
double aux; //for circulation calculation
bool check=0;
double analytical[nx][ny];
double r[nx][ny];
double theta[nx][ny];

void blockingGen(double pos[2][nx][ny], double blocking[nx][ny]){
    for (int i=0;i<=(nx-1);i++){
        for (int j=0;j<=(ny-1);j++){
            if(pos[0][i][j]>=(l/2-sqrt(pow(R,2.0)-pow(pos[1][i][j]-h/2,2.0)))
               &&pos[0][i][j]<=(l/2+sqrt(pow(R,2.0)-pow(pos[1][i][j]-h/2,2.0)))
               &&pos[1][i][j]>=(h/2-sqrt(pow(R,2.0)-pow(pos[0][i][j]-l/2,2.0)))
               &&pos[1][i][j]>=(h/2-sqrt(pow(R,2.0)-pow(pos[0][i][j]-l/2,2.0))))
                 {
                 blocking[i][j]=1;
                 rho[i][j]=0;
            }else{
                 blocking[i][j]=0;
                 rho[i][j]=rho0;
            }
        }
```

```
    }
}

void preprocessor(double& dx, double& dy, double pos[2][nx][ny],
                  double geo[2][nx][ny], double blocking[nx][ny]){
    dx=l/(nx-2); dy=h/(ny-2);
    pos[0][0][0]=0; //corner nodes positions and geometry
    pos[1][0][0]=0;
    geo[0][0][0]=0;
    geo[1][0][0]=0;
    pos[0][nx-1][0]=l;
    pos[1][nx-1][0]=0;
    geo[0][nx-1][0]=0;
    geo[1][nx-1][0]=0;
    pos[0][nx-1][ny-1]=l;
    pos[1][nx-1][ny-1]=h;
    geo[0][nx-1][ny-1]=0;
    geo[1][nx-1][ny-1]=0;
    pos[0][0][ny-1]=0;
    pos[1][0][ny-1]=h;
    geo[0][0][ny-1]=0;
    geo[1][0][ny-1]=0;
    for (int i=1;i<=(nx-2);i++){ //top and bottom boundaries
        pos[0][i][0]=0.5*dx+(i-1)*dx;
        pos[1][i][0]=0;
        geo[0][i][0]=dx;
        geo[1][i][0]=0;
        pos[0][i][ny-1]=0.5*dx+(i-1)*dx;
        pos[1][i][ny-1]=h;
        geo[0][i][ny-1]=dx;
        geo[1][i][ny-1]=0;
    }
    for (int j=1;j<=(ny-2);j++){ //side boundaries
        pos[0][0][j]=0;
        pos[1][0][j]=0.5*dy+(j-1)*dy;
        geo[0][0][j]=0;
        geo[1][0][j]=dy;
        pos[0][nx-1][j]=l;
        pos[1][nx-1][j]=0.5*dy+(j-1)*dy;
        geo[0][nx-1][j]=0;
        geo[1][nx-1][j]=dy;
    }
    for (int i=1;i<=(ny-2);i++){ //inner nodes
        for (int j=1;j<=(nx-2);j++){
            pos[0][i][j]=pos[0][i][0];
            pos[1][i][j]=pos[1][0][j];
```

```
            geo[0][i][j]=geo[0][i][0];
            geo[1][i][j]=geo[1][0][j];
        }
    }
    blockingGen(pos,blocking);

}

void initialEstimation(double phi[nx+2][ny+2],double blocking[nx][ny]){
    for (int i=0;i<=(nx+1);i++){
        for (int j=0;j<=(ny+1);j++){
            phi[i][j]=0;
        }
    }
}

double harmonicAvg(double pos2,double pos1,double surface2,double surface1,
                   double rho2,double rho1){
    double rho,dnodes,d1,d2;
    dnodes=pos2-pos1;
    d1=surface1/2;
    d2=surface2/2;
    if (rho1==0&&rho2==0){
        rho=0;
    }else{
        rho=(dnodes/(((rho1*d1)/rho0)+((rho2*d2)/rho0)));
    }
    return rho;
}

void coefCalc(double coef[6][nx][ny],double geo[2][nx][ny],
              double pos[2][nx][ny]){
    for (int i=1;i<=(nx-2);i++){ //TDMA coefficients calculation inner nodes
        for (int j=1;j<=(ny-2);j++){
            coef[1][i][j]=harmonicAvg(pos[0][i+1][j],pos[0][i][j],
                                      geo[0][i+1][j],geo[0][i][j],rho[i+1][j],
                                      rho[i][j])*(geo[1][i][j]/(pos[0][i+1][j]
                                                                -pos[0][i][j]));
            coef[2][i][j]=harmonicAvg(pos[0][i][j],pos[0][i-1][j],geo[0][i][j],
                                      geo[0][i-1][j],rho[i][j],rho[i-1][j])*
                                      (geo[1][i][j]/(pos[0][i][j]
                                                     -pos[0][i-1][j]));
            coef[3][i][j]=harmonicAvg(pos[1][i][j+1],pos[1][i][j],
                                      geo[1][i][j+1],geo[1][i][j],rho[i][j+1],
                                      rho[i][j])*(geo[0][i][j]/(pos[1][i][j+1]
                                                                -pos[1][i][j]));
```

```
            coef[4][i][j]=harmonicAvg(pos[1][i][j],pos[1][i][j-1],geo[1][i][j],
                                      geo[1][i][j-1],rho[i][j],rho[i][j-1])*
                                      (geo[0][i][j]/(pos[1][i][j]
                                                     -pos[1][i][j-1]));
            coef[0][i][j]=coef[1][i][j]+coef[2][i][j]+coef[3][i][j]
            +coef[4][i][j];
            coef[5][i][j]=0;
            if(coef[0][i][j]==0){
                coef[0][i][j]=1; //for computational reasons
            }
        }
    }


    coef[0][0][0]=1;
    coef[1][0][0]=0;
    coef[2][0][0]=0;
    coef[3][0][0]=0;
    coef[4][0][0]=0;
    coef[5][0][0]=0;
    coef[0][nx-1][0]=1;
    coef[1][nx-1][0]=0;
    coef[2][nx-1][0]=0;
    coef[3][nx-1][0]=0;
    coef[4][nx-1][0]=0;
    coef[5][nx-1][0]=0;
    coef[0][nx-1][ny-1]=1;
    coef[1][nx-1][ny-1]=0;
    coef[2][nx-1][ny-1]=0;
    coef[3][nx-1][ny-1]=0;
    coef[4][nx-1][ny-1]=0;
    coef[5][nx-1][ny-1]=0;
    coef[0][0][ny-1]=1;
    coef[1][0][ny-1]=0;
    coef[2][0][ny-1]=0;
    coef[3][0][ny-1]=0;
    coef[4][0][ny-1]=0;
    coef[5][0][ny-1]=0;


}

void conditions(double phi[nx+2][ny+2], double coef[6][nx][ny],
                double blocking[nx][ny], double circPsi){
    for (int i=1;i<=(nx-1);i++){ //top and bottom conditions
        coef[5][i][0]=0;
```

```
            coef[4][i][0]=0;
            coef[3][i][0]=0;
            coef[2][i][0]=0;
            coef[1][i][0]=0;
            coef[0][i][0]=1;
            coef[5][i][ny-1]=vinf*h;
            coef[1][i][ny-1]=0;
            coef[2][i][ny-1]=0;
            coef[3][i][ny-1]=0;
            coef[4][i][ny-1]=0;
            coef[0][i][ny-1]=1;
        }
        for (int j=0;j<=(ny-1);j++){ //inlet conditions
            coef[5][0][j]=vinf*pos[1][0][j];
            coef[0][0][j]=1;
            coef[1][0][j]=0;
            coef[2][0][j]=0;
            coef[3][0][j]=0;
            coef[4][0][j]=0;
        }
        for (int j=1;j<=(ny-2);j++){ //outlet conditions
            coef[1][nx-1][j]=0;
            coef[2][nx-1][j]=1;
            coef[3][nx-1][j]=0;
            coef[4][nx-1][j]=0;
            coef[0][nx-1][j]=1;
            coef[5][nx-1][j]=0;
        }
        for (int i=0;i<=(nx-1);i++){
            for (int j=0;j<=(ny-1);j++){
                if (blocking[i][j]==1){
                    coef[5][i][j]=circPsi;
                    coef[0][i][j]=1;
                    coef[1][i][j]=0;
                    coef[2][i][j]=0;
                    coef[3][i][j]=0;
                    coef[4][i][j]=0;
                }
            }
        }

        coef[0][nx-1][ny-1]=1;

        coef[0][nx-1][0]=1;
}
```

```
void lineByLineRow(double bstarx[nx], double coef[6][nx][ny],
                   double phi[nx+2][ny+2], int j){
    for (int i=0;i<=(nx-1);i++){
        bstarx[i]=coef[5][i][j]+coef[3][i][j]*phi[i+1][j+2]
        +coef[4][i][j]*phi[i+1][j];
    }
}

void TDMArow(double phi[nx+2][ny+2], double coef[6][nx][ny],
             double bstarx[nx], int j,double Prow[nx+1],double Qrow[nx+1],
             double blocking[nx][ny], double circPsi){
    Prow[0]=0; Qrow[0]=0;
    for (int a=1;a<=nx;a++){
        Prow[a]=coef[1][a-1][j]/(coef[0][a-1][j]-coef[2][a-1][j]*Prow[a-1]);
        Qrow[a]=(bstarx[a-1]+coef[2][a-1][j]*Qrow[a-1])/(coef[0][a-1][j]
                                                        -coef[2][a-1][j]
                                                        *Prow[a-1]);
    }
    for (int a=(nx);a>=1;a--){
        phi[a][j+1]=Prow[a]*phi[a+1][j+1]+Qrow[a];
    }
}

void lineByLineColumn(double bstary[ny], double coef[6][nx][ny],
                      double phi[nx+2][ny+2], int i){
    for (int j=0;j<=(ny-1);j++){
        bstary[j]=coef[5][i][j]+coef[2][i][j]*phi[i][j+1]+coef[1][i][j]*
        phi[i+2][j+1];
    }
}

void TDMAcolumn(double phi[nx+2][ny+2], double coef[6][nx][ny],
                double bstary[ny], int i,double Pcolumn[ny+1],
                double Qcolumn[ny+1],double blocking[nx][ny],double circPsi){
    Pcolumn[0]=0; Qcolumn[0]=0;
    for (int a=1;a<=(ny);a++){
        Pcolumn[a]=coef[3][i][a-1]/(coef[0][i][a-1]-coef[4][i][a-1]
                                   *Pcolumn[a-1]);
        Qcolumn[a]=(bstary[a-1]+coef[4][i][a-1]*Qcolumn[a-1])/(coef[0][i][a-1]
                                                              -coef[4][i][a-1]
                                                              *Pcolumn[a-1]);
    }
    for (int a=(ny);a>=1;a--){
        phi[i+1][a]=Pcolumn[a]*phi[i+1][a+1]+Qcolumn[a];
    }
}
```

```
double abs(double value){
    if(value<0) value=-value;
    return value;
}

void errCalc(double phi[nx+2][ny+2],double phiprev[nx+2][ny+2],
             double errMat[nx][ny]){
    double diff;
    for (int i=1; i<=(nx); i++){
        for (int j=1; j<=(ny); j++){
            diff=phi[i][j]-phiprev[i][j];
            errMat[i-1][j-1]=(abs(diff))/phiprev[i][j];
        }
    }
}

double findMax(double errMat[nx][ny]){
    double err=0;
    for(int i=0;i<=(nx-1);i++){
        for(int j=0;j<=(ny-1);j++){
            if(errMat[i][j]>err) err=errMat[i][j];
        }
    }
    return err;
}

double circEval(double blocking[nx][ny], double coef[6][nx][ny],
                double phi[nx+2][ny+2]){
    double sum=0;
    for (int i=1;i<=(nx-2);i++){
        for (int j=1;j<=(ny-2);j++){
            if (blocking[i][j]==0&&(blocking[i+1][j]==1||blocking[i-1][j]==1
                                    ||blocking[i][j+1]==1
                                    ||blocking[i][j-1]==1)){
                sum=sum+coef[1][i][j]*(phi[i+1][j+1]-
                    phi[i+2][j+1])-coef[2][i][j]*(phi[i+1][j+1]-phi[i][j+1])
                    -coef[3][i][j]*
                    (phi[i+1][j+2]-phi[i+1][j+1])+coef[4][i][j]*(phi[i+1][j+1]-
                                                                 phi[i+1][j]);
            }
        }
    }
    return sum;
}
```

```
double secant(double a, double b, double c, double d, bool& check){
    double newPsi;
    newPsi=(a*d-c*b)/(a-b);

    return newPsi;
}

void solver(double pos[2][nx][ny],double geo[2][nx][ny],double phi[nx+2][ny+2],
            double coef[6][nx][ny],double errMat[nx][ny],double bstarx[nx],
            double bstary[ny],double Prow[nx+1],double Qrow[nx+1],
            double Pcolumn[ny+1],double Qcolumn[ny+1],double blocking[nx][ny],
            double circPsi,double circPsiPrev,double circ,double circPrev,
            double errCirc,double maxErr,double aux,bool check){


    coefCalc(coef,geo,pos);

    while (errCirc>deltaCirc){

        initialEstimation(phi,blocking);

        conditions(phi,coef,blocking,circPsi);

        while (maxErr>delta){
            //saving previous phi for later error calculation
            for (int i=0;i<=(nx+1);i++){
                for (int j=0;j<=(ny+1);j++){
                    phiprev[i][j]=phi[i][j];
                }
            }
            for (int i=0;i<=(nx-1);i++){ //by columns
                //turns a 2D problem into a 1D problem
                lineByLineColumn(bstary,coef,phi,i);
                TDMAcolumn(phi,coef,bstary,i,Pcolumn,Qcolumn,blocking,circPsi);
            }

            for (int j=0;j<=(ny-1);j++){ //by rows
                //turns a 2D problem into a 1D problem
                lineByLineRow(bstarx,coef,phi,j);
                TDMArow(phi,coef,bstarx,j,Prow,Qrow,blocking,circPsi);
            }

            errCalc(phi,phiprev,errMat);
            maxErr=findMax(errMat);
            cout << "err=" << maxErr << endl;
        }
```

```
        maxErr=10;

        circ=circEval(blocking,coef,phi);

        errCirc=abs(circ-circCondition);
        aux=secant(circPrev,circ,circPsiPrev,circPsi,check);
        circPrev=circ;
        circPsiPrev=circPsi;
        circPsi=aux;
        cout << errCirc << endl;



    }

}

void postprocessor(double pos[2][nx][ny],double phi[nx+2][ny+2],
                   double analytical[nx][ny],double r[nx][ny],
                   double theta[nx][ny]){
    string filename="stream.dat";
    ofstream datafileC (filename.c_str());
    for(int i=0;i<=(nx-1);i++){
        for(int j=0;j<=(ny-1);j++){
            datafileC << pos[0][i][j] << " " << pos[1][i][j]
            << " " << blocking[i][j] << " " << phi[i+1][j+1] << endl;
        }
        datafileC << endl;
    }
    datafileC.close();
    for (int i=0;i<=(nx-1);i++){
        for (int j=0;j<=(ny-1);j++){
            r[i][j]=sqrt(pow((pos[0][i][j]-l/2),2.0)+
                         pow((pos[1][i][j]-h/2),2.0));
            theta[i][j]=atan2((pos[1][i][j]-h/2),(pos[0][i][j]-l/2));
            analytical[i][j]=vinf*sin(theta[i][j])*(r[i][j]-(pow(R,2)/r[i][j]))
            +circCondition/(2*3.1416)*log(r[i][j]/R);
        }
    }
    filename="analytical.dat";
    ofstream datafileA (filename.c_str());
    for(int i=0;i<=(nx-1);i++){
        for(int j=0;j<=(ny-1);j++){
            datafileA << r[i][j]*cos(theta[i][j])+l/2 << " "
            << r[i][j]*sin(theta[i][j])+h/2 << " " << analytical[i][j]
            << " " << analytical[i][j]+0.5 << endl;
```

```
            }
            datafileA << endl;
        }
        datafileA.close();
}

int main(){
        preprocessor(dx,dy,pos,geo,blocking);
        solver(pos,geo,phi,coef,errMat,bstarx,bstary,Prow,Qrow,Pcolumn,
                Qcolumn,blocking,circPsi,circPsiPrev,circ,circPrev,errCirc,
                maxErr,aux,check);
        postprocessor(pos,phi,analytical,r,theta);

}
```

Attachment C

COMPRESSIBLE FLOW CODE

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <stdlib.h>
#include <math.h>

using namespace std;


const int nx=200, ny=200; //number of nodes
const double delta=1.0/(100*(nx*ny)); //relative error acceptance
const double h=2.0, l=4.0; //geometric definition in meters (m)
const double vinf=1; //free stream velocity in meters per second (m/s)
const double R=0.15; //cylinder radius in meters (m);
const double deltaCirc=0.000001; //absolute error allowed in solid circulation
const double circCondition=0.0;
const double Rg=287;
const double cp=1.012;
const double gamma=1.4;
const double P0=100000;
const double T0=287;
//air density at inlet in kilograms per meter cubed (Kg/m^3)
const double rho0=P0/(Rg*T0);

double dx, dy; //differential lengths in meters (m)
//nodes positions [dimension][nx][ny] for dimension 0) x coord. 1) y coord.
double pos[2][nx][ny];
//nodes geometrical data [parameter][nx][ny] for parameters 0) Sx 1) Sy
double geo[2][nx][ny];
//streamlines (+2 to include "ghost" nodes around the geometry
//for computational reasons)
```

```
double phi[nx+2][ny+2];
//TDMA coefficients 0) ap 1) ae 2) aw 3) an 4) as 5)bp
double coef[6][nx][ny];
double errMat[nx][ny]; //error matrix
double phiprev[nx+2][ny+2];
double bstarx[nx], bstary[ny];
double Prow[nx+1], Qrow[nx+1];
double Pcolumn[ny+1], Qcolumn[ny+1];
double rho[nx][ny]; //nodes densities
double blocking[nx][ny]; //blocking-off matrix (1=solid;0=fluid)
double circPsi=vinf*h/2.0;
double circPsiPrev=10.0;
double circPrev=1.0;
double circ=0.0;
double errCirc=10;
double maxErr=10;
double P[nx][ny];
double T[nx][ny];
double v[3][nx][ny]; //0=v^2 1=vx 2=vy
double aux;
double r[nx][ny];
double theta[nx][ny];
double analytical[nx][ny];
double M[nx][ny];

void blockingGen(double pos[2][nx][ny], double blocking[nx][ny],
                 double rho[nx][ny],double P[nx][ny], double T[nx][ny]){
    for (int i=0;i<=(nx-1);i++){
        for (int j=0;j<=(ny-1);j++){
            if(pos[0][i][j]>=(l/2-sqrt(pow(R,2.0)-pow(pos[1][i][j]-h/2,2.0)))
               &&pos[0][i][j]<=(l/2+sqrt(pow(R,2.0)-pow(pos[1][i][j]-h/2,2.0)))
               &&pos[1][i][j]>=(h/2-sqrt(pow(R,2.0)-pow(pos[0][i][j]-l/2,2.0)))
               &&pos[1][i][j]>=(h/2-sqrt(pow(R,2.0)-pow(pos[0][i][j]-l/2,2.0))))
                 {
                 blocking[i][j]=1;
                 rho[i][j]=0;
                 P[i][j]=0;
                 T[i][j]=0;
            }else{
                 blocking[i][j]=0;
                 rho[i][j]=rho0;
                 P[i][j]=P0;
                 T[i][j]=T0;
            }
        }
    }
```

```
}

void preprocessor(double& dx, double& dy, double pos[2][nx][ny],
                  double geo[2][nx][ny], double blocking[nx][ny],
                  double rho[nx][ny],double P[nx][ny], double T[nx][ny]){
    dx=l/(nx-2); dy=h/(ny-2);
    pos[0][0][0]=0; //corner nodes positions and geometry
    pos[1][0][0]=0;
    geo[0][0][0]=0;
    geo[1][0][0]=0;
    pos[0][nx-1][0]=l;
    pos[1][nx-1][0]=0;
    geo[0][nx-1][0]=0;
    geo[1][nx-1][0]=0;
    pos[0][nx-1][ny-1]=l;
    pos[1][nx-1][ny-1]=h;
    geo[0][nx-1][ny-1]=0;
    geo[1][nx-1][ny-1]=0;
    pos[0][0][ny-1]=0;
    pos[1][0][ny-1]=h;
    geo[0][0][ny-1]=0;
    geo[1][0][ny-1]=0;
    for (int i=1;i<=(nx-2);i++){ //top and bottom boundaries
        pos[0][i][0]=0.5*dx+(i-1)*dx;
        pos[1][i][0]=0;
        geo[0][i][0]=dx;
        geo[1][i][0]=0;
        pos[0][i][ny-1]=0.5*dx+(i-1)*dx;
        pos[1][i][ny-1]=h;
        geo[0][i][ny-1]=dx;
        geo[1][i][ny-1]=0;
    }
    for (int j=1;j<=(ny-2);j++){ //side boundaries
        pos[0][0][j]=0;
        pos[1][0][j]=0.5*dy+(j-1)*dy;
        geo[0][0][j]=0;
        geo[1][0][j]=dy;
        pos[0][nx-1][j]=l;
        pos[1][nx-1][j]=0.5*dy+(j-1)*dy;
        geo[0][nx-1][j]=0;
        geo[1][nx-1][j]=dy;
    }
    for (int i=1;i<=(ny-2);i++){ //inner nodes
        for (int j=1;j<=(nx-2);j++){
            pos[0][i][j]=pos[0][i][0];
            pos[1][i][j]=pos[1][0][j];
```

```
            geo[0][i][j]=geo[0][i][0];
            geo[1][i][j]=geo[1][0][j];
        }
    }
    blockingGen(pos,blocking,rho,P,T);

}

void initialEstimation(double phi[nx+2][ny+2],double blocking[nx][ny]){
    for (int i=0;i<=(nx+1);i++){
        for (int j=0;j<=(ny+1);j++){
            phi[i][j]=0;
        }
    }
}

double harmonicAvg(double pos2,double pos1,double surface2,double surface1,
                   double rho2,double rho1){
    double rho,dnodes,d1,d2;
    dnodes=pos2-pos1;
    d1=surface1/2;
    d2=surface2/2;
    if (rho1==0&&rho2==0){
        rho=0;
    }else{
        rho=(dnodes/(((rho1*d1)/rho0)+((rho2*d2)/rho0)));
    }
    return rho;
}

void coefCalc(double coef[6][nx][ny],double geo[2][nx][ny],
              double pos[2][nx][ny]){
    for (int i=1;i<=(nx-2);i++){ //TDMA coefficients calculation inner nodes
        for (int j=1;j<=(ny-2);j++){
            coef[1][i][j]=harmonicAvg(pos[0][i+1][j],pos[0][i][j],
                                      geo[0][i+1][j],geo[0][i][j],
                                      rho[i+1][j],rho[i][j])*(geo[1][i][j]
                                                              /(pos[0][i+1][j]
                                                                -pos[0][i][j]));
            coef[2][i][j]=harmonicAvg(pos[0][i][j],pos[0][i-1][j],
                                      geo[0][i][j],geo[0][i-1][j],
                                      rho[i][j],rho[i-1][j])
                                      *(geo[1][i][j]/(pos[0][i][j]
                                                      -pos[0][i-1][j]));
            coef[3][i][j]=harmonicAvg(pos[1][i][j+1],pos[1][i][j],
                                      geo[1][i][j+1],geo[1][i][j],rho[i][j+1],
```

```
                                        rho[i][j])*(geo[0][i][j]
                                                    /(pos[1][i][j+1]
                                                      -pos[1][i][j]));
            coef[4][i][j]=harmonicAvg(pos[1][i][j],pos[1][i][j-1],
                                        geo[1][i][j],geo[1][i][j-1],rho[i][j],
                                        rho[i][j-1])*(geo[0][i][j]
                                                      /(pos[1][i][j]
                                                        -pos[1][i][j-1]));
            coef[0][i][j]=coef[1][i][j]+coef[2][i][j]+coef[3][i][j]
            +coef[4][i][j];
            coef[5][i][j]=0;
            if(coef[0][i][j]==0){
                coef[0][i][j]=1; //for computational reasons
            }
        }
    }



}

void conditions(double phi[nx+2][ny+2], double coef[6][nx][ny],
                double blocking[nx][ny], double circPsi){
    for (int i=1;i<=(nx-1);i++){ //top and bottom conditions
        coef[5][i][0]=0;
        coef[4][i][0]=0;
        coef[3][i][0]=0;
        coef[2][i][0]=0;
        coef[1][i][0]=0;
        coef[0][i][0]=1;
        coef[5][i][ny-1]=vinf*h;
        coef[1][i][ny-1]=0;
        coef[2][i][ny-1]=0;
        coef[3][i][ny-1]=0;
        coef[4][i][ny-1]=0;
        coef[0][i][ny-1]=1;
    }
    for (int j=0;j<=(ny-1);j++){ //inlet conditions
        coef[5][0][j]=vinf*pos[1][0][j];
        coef[0][0][j]=1;
        coef[1][0][j]=0;
        coef[2][0][j]=0;
        coef[3][0][j]=0;
        coef[4][0][j]=0;

    }
```

```
    for (int j=1;j<=(ny-2);j++){ //outlet conditions
        coef[1][nx-1][j]=0;
        coef[2][nx-1][j]=1;
        coef[3][nx-1][j]=0;
        coef[4][nx-1][j]=0;
        coef[0][nx-1][j]=1;
        coef[5][nx-1][j]=0;
    }
    for (int i=0;i<=(nx-1);i++){
        for (int j=0;j<=(ny-1);j++){
            if (blocking[i][j]==1){
                coef[5][i][j]=circPsi;
                coef[0][i][j]=1;
                coef[1][i][j]=0;
                coef[2][i][j]=0;
                coef[3][i][j]=0;
                coef[4][i][j]=0;
            }
        }
    }

    coef[0][nx-1][ny-1]=1;

    coef[0][nx-1][0]=1;
}

void lineByLineRow(double bstarx[nx], double coef[6][nx][ny],
                   double phi[nx+2][ny+2], int j){
    for (int i=0;i<=(nx-1);i++){
        bstarx[i]=coef[5][i][j]+coef[3][i][j]*phi[i+1][j+2]
        +coef[4][i][j]*phi[i+1][j];
    }
}

void TDMArow(double phi[nx+2][ny+2], double coef[6][nx][ny],
             double bstarx[nx], int j,double Prow[nx+1],double Qrow[nx+1],
             double blocking[nx][ny], double circPsi){
    Prow[0]=0; Qrow[0]=0;
    for (int a=1;a<=nx;a++){
        Prow[a]=coef[1][a-1][j]/(coef[0][a-1][j]-coef[2][a-1][j]*Prow[a-1]);
        Qrow[a]=(bstarx[a-1]+coef[2][a-1][j]*Qrow[a-1])
        /(coef[0][a-1][j]-coef[2][a-1][j]*Prow[a-1]);
    }
    for (int a=(nx);a>=1;a--){
        phi[a][j+1]=Prow[a]*phi[a+1][j+1]+Qrow[a];
    }
```

```
}

void lineByLineColumn(double bstary[ny], double coef[6][nx][ny],
                      double phi[nx+2][ny+2], int i){
    for (int j=0;j<=(ny-1);j++){
        bstary[j]=coef[5][i][j]+coef[2][i][j]*phi[i][j+1]+
        coef[1][i][j]*phi[i+2][j+1];
    }
}

void TDMAcolumn(double phi[nx+2][ny+2], double coef[6][nx][ny],
                double bstary[ny], int i,double Pcolumn[ny+1],
                double Qcolumn[ny+1],double blocking[nx][ny],double circPsi){
    Pcolumn[0]=0; Qcolumn[0]=0;
    for (int a=1;a<=(ny);a++){
        Pcolumn[a]=coef[3][i][a-1]/(coef[0][i][a-1]-coef[4][i][a-1]
                                   *Pcolumn[a-1]);
        Qcolumn[a]=(bstary[a-1]+coef[4][i][a-1]*Qcolumn[a-1])
        /(coef[0][i][a-1]-coef[4][i][a-1]*Pcolumn[a-1]);
    }
    for (int a=(ny);a>=1;a--){
        phi[i+1][a]=Pcolumn[a]*phi[i+1][a+1]+Qcolumn[a];
    }
}

double abs(double value){
    if(value<0) value=-value;
    return value;
}

void velCalc(int i, int j,double v[3][nx][ny], double coef[6][nx][ny],
             double geo[2][nx][ny], double phi[nx+2][ny+2]){
    if((i==0&&j==0)||(i==0&&j==(ny-1))||(i==(nx-1)&&j==0)
       ||(i==(nx-1)&&j==(ny-1))){
        v[1][i][j]=0;
        v[2][i][j]=0;
        v[0][i][j]=0;
    }else if(i==0&&(j!=0||j!=(ny-1))){
        v[1][i][j]=0;
        v[2][i][j]=0.5*(coef[1][i][j]/geo[1][i][j]*(phi[i+2][j+1]
                                                    -phi[i+1][j+1]));
        v[0][i][j]=v[0][i][j]+(pow(v[2][i][j],2.0)-v[0][i][j])*0.01;
    }else if(i==(nx-1)&&(j!=0||j!=(ny-1))){
        v[1][i][j]=0;
        v[2][i][j]=0.5*(coef[2][i][j]/geo[1][i][j]*(phi[i+1][j+1]
                                                    -phi[i][j+1]));
```

```
        v[0][i][j]=v[0][i][j]+(pow(v[2][i][j],2.0)-v[0][i][j])*0.01;
    }else if (j==0&&(i!=0||i!=(nx-1))){
        v[1][i][j]=0.5*(coef[3][i][j]/geo[0][i][j]*(phi[i+1][j+2]
                                                    -phi[i+1][j+1]));
        v[2][i][j]=0;
        v[0][i][j]=v[0][i][j]+(pow(v[1][i][j],2.0)-v[0][i][j])*0.01;
    }else if (j==(ny-1)&&(i!=0||i!=(nx-1))){
        v[1][i][j]=0.5*(coef[4][i][j]/geo[0][i][j]*(phi[i+1][j+1]
                                                    -phi[i+1][j]));
        v[2][i][j]=0;
        v[0][i][j]=v[0][i][j]+(pow(v[1][i][j],2.0)-v[0][i][j])*0.01;
    }else{
        v[1][i][j]=0.5*(coef[3][i][j]/geo[0][i][j]*(phi[i+1][j+2]
                                                    -phi[i+1][j+1])
                        +coef[4][i][j]/geo[0][i][j]*(phi[i+1][j+1]
                                                     -phi[i+1][j]));
        v[2][i][j]=0.5*(coef[1][i][j]/geo[1][i][j]*(phi[i+2][j+1]
                                                    -phi[i+1][j+1])
                        +coef[2][i][j]/geo[1][i][j]*(phi[i+1][j+1]
                                                     -phi[i][j+1]));
        v[0][i][j]=v[0][i][j]+(pow(v[1][i][j],2.0)+pow(v[2][i][j],2.0)
                               -v[0][i][j])*0.01;
    }


}

void gasProp(double v[3][nx][ny],double coef[6][nx][ny],double geo[2][nx][ny],
             double phi[nx+2][ny+2],double T[nx][ny],double P[nx][ny],
             double rho[nx][ny]){
    for (int i=0; i<=(nx-1);i++){
        for (int j=0; j<=(ny-1);j++){
            velCalc(i,j,v,coef,geo,phi);
            T[i][j]=T0+(pow(vinf,2.0)-v[0][i][j])/(2*cp);
            if (T[i][j]<=0){
                T[i][j]=(T[i+1][j]+T[i-1][j]+T[i][j+1]+T[i][j-1])/4.0;
            }
            P[i][j]=P0*pow(abs(T[i][j]/T0),(gamma/(gamma-1)));
            rho[i][j]=abs(P[i][j]/(Rg*T[i][j]))*(1-blocking[i][j]);
        }
    }
}

void errCalc(double phi[nx+2][ny+2],double phiprev[nx+2][ny+2],
             double errMat[nx][ny]){
    double diff;
```

```
    for (int i=1; i<=(nx); i++){
        for (int j=1; j<=(ny); j++){
            diff=phi[i][j]-phiprev[i][j];
            errMat[i-1][j-1]=(abs(diff))/phiprev[i][j];
        }
    }
}

double findMax(double errMat[nx][ny]){
    double err=0;
    for(int i=0;i<=(nx-1);i++){
        for(int j=0;j<=(ny-1);j++){
            if(errMat[i][j]>err) err=errMat[i][j];
        }
    }
    return err;
}

double circEval(double blocking[nx][ny], double coef[6][nx][ny],
                double phi[nx+2][ny+2]){
    double sum=0;
    for (int i=1;i<=(nx-2);i++){
        for (int j=1;j<=(ny-2);j++){
            if (blocking[i][j]==0&&(blocking[i+1][j]==1||blocking[i-1][j]==1
                                    ||blocking[i][j+1]==1
                                    ||blocking[i][j-1]==1)){
                sum=sum+coef[1][i][j]*(phi[i+1][j+1]-
                    phi[i+2][j+1])-coef[2][i][j]*(phi[i+1][j+1]-phi[i][j+1])
                    -coef[3][i][j]*
                    (phi[i+1][j+2]-phi[i+1][j+1])+coef[4][i][j]*(phi[i+1][j+1]-
                                                                 phi[i+1][j]);
            }
        }
    }
    return sum;
}

double secant(double a, double b, double c, double d){
    double newPsi;
    newPsi=(a*d-c*b)/(a-b);
    return newPsi;
}

void solver(double pos[2][nx][ny],double geo[2][nx][ny],double phi[nx+2][ny+2],
            double coef[6][nx][ny],double errMat[nx][ny],double bstarx[nx],
            double bstary[ny],double Prow[nx+1],double Qrow[nx+1],
```

```
        double Pcolumn[ny+1],double Qcolumn[ny+1],double blocking[nx][ny],
        double circPsi,double circPsiPrev,double circ,double circPrev,
        double errCirc,double maxErr,double v[3][nx][ny],double T[nx][ny],
        double P[nx][ny],double rho[nx][ny],double aux){


coefCalc(coef,geo,pos);

while (errCirc>deltaCirc){


    initialEstimation(phi,blocking);


    while (maxErr>delta){
        coefCalc(coef,geo,pos);
        conditions(phi,coef,blocking,circPsi);
        //saving previous phi for later error calculation
        for (int i=0;i<=(nx+1);i++){
            for (int j=0;j<=(ny+1);j++){
                phiprev[i][j]=phi[i][j];
            }
        }

        for (int i=0;i<=(nx-1);i++){ //by columns
            //turns a 2D problem into a 1D problem
            lineByLineColumn(bstary,coef,phi,i);
            TDMAcolumn(phi,coef,bstary,i,Pcolumn,Qcolumn,blocking,circPsi);
        }
        for (int j=0;j<=(ny-1);j++){ //by rows
            //turns a 2D problem into a 1D problem
            lineByLineRow(bstarx,coef,phi,j);
            TDMArow(phi,coef,bstarx,j,Prow,Qrow,blocking,circPsi);
        }

        gasProp(v,coef,geo,phi,T,P,rho);
        errCalc(phi,phiprev,errMat);
        maxErr=findMax(errMat);
        cout << "err=" << maxErr << endl;
    }
    maxErr=10;
    circ=circEval(blocking,coef,phi);
    errCirc=abs(circ-circCondition);
    aux=secant(circPrev,circ,circPsiPrev,circPsi);
    circPrev=circ;
    circPsiPrev=circPsi;
```

```
        circPsi=aux;
        cout << circ << endl;
    }

}

void postprocessor(double pos[2][nx][ny],double phi[nx+2][ny+2],
                   double rho[nx][ny],double P[nx][ny], double T[nx][ny],
                   double r[nx][ny],double theta[nx][ny],
                   double analytical[nx][ny],double v[0][nx][ny]){
    for(int i=0;i<=(nx-1);i++){
        for(int j=0;j<=(ny-1);j++){
            if(blocking[i][j]==1){
                P[i][j]=P0;
                T[i][j]=T0;
                rho[i][j]=rho0;
            }
        }
    }
    string filename="stream.dat";
    ofstream datafileS (filename.c_str());
    for(int i=0;i<=(nx-1);i++){
        for(int j=0;j<=(ny-1);j++){
            datafileS << pos[0][i][j] << " " << pos[1][i][j]
            << " " << phi[i+1][j+1] << " " << phi[i+1][j+1] << endl;
        }
        datafileS << endl;
    }
    datafileS.close();
    filename="temperature.dat";
    ofstream datafileT (filename.c_str());
    for(int i=0;i<=(nx-1);i++){
        for(int j=0;j<=(ny-1);j++){
            datafileT << pos[0][i][j] << " " << pos[1][i][j]
            << " " << phi[i+1][j+1] << " " << T[i][j] << endl;
        }
        datafileT << endl;
    }
    datafileT.close();
    filename="pressure.dat";
    ofstream datafileP (filename.c_str());
    for(int i=0;i<=(nx-1);i++){
        for(int j=0;j<=(ny-1);j++){
            datafileP << pos[0][i][j] << " " << pos[1][i][j]
             << " " << phi[i+1][j+1] << " " << P[i][j] << endl;
        }
```

```
            datafileP << endl;
        }
        datafileP.close();
        filename="density.dat";
        ofstream datafileD (filename.c_str());
        for(int i=0;i<=(nx-1);i++){
            for(int j=0;j<=(ny-1);j++){
                datafileD << pos[0][i][j] << " " << pos[1][i][j]
                << " " << phi[i+1][j+1] << " " << rho[i][j] << endl;
            }
            datafileD << endl;
        }
        datafileD.close();
        for (int i=0;i<=(nx-1);i++){
            for (int j=0;j<=(ny-1);j++){
                r[i][j]=sqrt(pow((pos[0][i][j]-l/2),2.0)
                            +pow((pos[1][i][j]-h/2),2.0));
                theta[i][j]=atan2((pos[1][i][j]-h/2),(pos[0][i][j]-l/2));
                M[i][j]=sqrt(v[0][i][j])/sqrt(gamma*Rg*T[i][j]);
                analytical[i][j]=vinf*sin(theta[i][j])*(r[i][j]
                                                        -(pow(R,2)/r[i][j]));

            }
        }
        filename="analytical.dat";
        ofstream datafileA (filename.c_str());
        for(int i=0;i<=(nx-1);i++){
            for(int j=0;j<=(ny-1);j++){
                datafileA << r[i][j]*cos(theta[i][j])+l/2 << " "
                << r[i][j]*sin(theta[i][j])+h/2 << " " << analytical[i][j]+h/2
                << " " << analytical[i][j]+h/2 << endl;
            }
            datafileA << endl;
        }
}

int main(){
    preprocessor(dx,dy,pos,geo,blocking,rho,P,T);
    solver(pos,geo,phi,coef,errMat,bstarx,bstary,Prow,Qrow,Pcolumn,Qcolumn,
           blocking,circPsi,circPsiPrev,circ,circPrev,errCirc,maxErr,v,T,P,
           rho,aux);
    postprocessor(pos,phi,rho,P,T,r,theta,analytical,v);
}
```

Attachment D

CONVECTION-DIFFUSION CODE

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <stdlib.h>
#include <math.h>

using namespace std;

const int nx=500, ny=100; //number of nodes in each dimension
const double delta=1.0/(100*(nx*ny)); //relative error acceptance
const double deltat=0.1; //time step (s)
const double tmax=100; //simulation time (s)
const double h=1.0, l=1.0; //geometrical definitions (m)
const double rho0=1; //density in (kg/m^3)
const double gamma=0.1; //diffusion coefficient
const double Sc=0, Sp=0; //source terms
const double phiIn=20; //inlet boundary condition (Dirichlet)
const double phiOut=50; //outlet boundary condition (Dirichlet)
const double flux=0; //top/bottom boundary condition (Neumann)
const double v0=10; //free stream velocity

double dx,dy; //nodal lengths
double facePosX[nx-1], facePosY[ny-1]; //face positions
double nodePos[2][nx][ny]; //node positions | 0-> x | 1-> y |
double geo[3][nx][ny]; //node geometrical properties | 0-> Sx | 1-> Sy | 2-> V |
double rhoMat[nx][ny]; //density matrix
double vMat[2][nx][ny]; //velocity field | 0-> vx | 1-> vy |
double gammaMat[nx][ny]; //gamma matrix
```

```
double SMat[2][nx][ny]; //S matrix | 0-> Sc | 1-> Sp |
double phi[nx][ny]; //variable matrix
//coefficients | 0-> ap | 1-> ae | 2-> aw | 3-> an | 4-> as | 5-> b |
double coef[6][nx][ny];
double errMat[nx][ny]; //error matrix
double phiPrev[nx][ny]; //previous phi
double phiEst[nx][ny]; //estimated phi
double bstarx[nx], bstary[ny]; //line-by-line b coefficients
double Prow[nx+1], Qrow[nx+1]; //TDMA coefficients by row
double Pcolumn[ny+1], Qcolumn[ny+1]; //TDMA coefficients by column
//max error in errMat (initialized at 10 for computational reasons
double maxErr=10;
double t=0; //time step
double fVec[4]; //Upwind-scheme f values | 0-> fe | 1-> fw | 2-> fn | 3-> fs |
double analyticalPhi[nx][ny];//analytical solution

//Utility functions

double abs(double val){
if (val<0) val=-val;
return val;
}

double findMax(double errMat[nx][ny]){
double err=0;
for (int i=0;i<=(nx-1);i++){
    for (int j=0;j<=(ny-1);j++){
        if (errMat[i][j]>err) err=errMat[i][j];
    }
}
return err;
}

//Main functions

double velocityField(double x, double y, int dim){
double v;
switch(dim){
    case 0: //vx field
    v=v0;
    break;
    case 1: //vy field
    v=0;
    break;
    }
return v;
```

```
}

void preprocessor(double& dx,double& dy,double facePosX[nx-1],
                  double facePosY[ny-1],double nodePos[2][nx][ny],
                  double geo[3][nx][ny],double rhoMat[nx][ny],
                  double vMat[2][nx][ny],double gammaMat[nx][ny],
                  double SMat[2][nx][ny],double phiPrev[nx][ny],
                  double coef[6][nx][ny]){
//nodal lengths
dx=l/(nx-2);
dy=h/(ny-2);
//faces positions
for (int i=0;i<=(nx-2);i++){
    facePosX[i]=i*dx;
}
for (int j=0;j<=(ny-2);j++){
    facePosY[j]=j*dy;
}
//corner nodes positions
nodePos[0][0][0]=0;
nodePos[1][0][0]=0;
nodePos[0][nx-1][0]=l;
nodePos[1][nx-1][0]=0;
nodePos[0][0][ny-1]=0;
nodePos[1][0][ny-1]=h;
nodePos[0][nx-1][ny-1]=l;
nodePos[0][nx-1][ny-1]=h;
//top/bottom boundary nodes positions
for (int i=1;i<=(nx-2);i++){
    nodePos[0][i][0]=(facePosX[i]+facePosX[i-1])/2;
    nodePos[1][i][0]=0;
    nodePos[0][i][ny-1]=(facePosX[i]+facePosX[i-1])/2;
    nodePos[1][i][ny-1]=h;
}
//left/right boundary nodes positions
for (int j=1;j<=(ny-2);j++){
    nodePos[0][0][j]=0;
    nodePos[1][0][j]=(facePosY[j]+facePosY[j-1])/2;
    nodePos[0][nx-1][j]=l;
    nodePos[1][nx-1][j]=(facePosY[j]+facePosY[j-1])/2;
}
//inner nodes positions
for (int i=1;i<=(nx-2);i++){
    for (int j=1;j<=(ny-2);j++){
        nodePos[0][i][j]=nodePos[0][i][0];
        nodePos[1][i][j]=nodePos[1][0][j];
```

```
    }
}
//corner nodes geometrical properties
geo[0][0][0]=0;
geo[1][0][0]=0;
geo[2][0][0]=0;
geo[0][nx-1][0]=0;
geo[1][nx-1][0]=0;
geo[2][nx-1][0]=0;
geo[0][0][ny-1]=0;
geo[1][0][ny-1]=0;
geo[2][0][ny-1]=0;
geo[0][nx-1][ny-1]=0;
geo[1][nx-1][ny-1]=0;
geo[2][nx-1][ny-1]=0;
//top/bottom boundary nodes geometrical properties
for (int i=1;i<=(nx-2);i++){
    geo[0][i][0]=facePosX[i]-facePosX[i-1];
    geo[1][i][0]=0;
    geo[2][i][0]=0;
    geo[0][i][ny-1]=facePosX[i]-facePosX[i-1];
    geo[1][i][ny-1]=0;
    geo[2][i][ny-1]=0;
}
//left/right boundary nodes geometrical properties
for (int j=1;j<=(ny-2);j++){
    geo[0][0][j]=0;
    geo[1][0][j]=facePosY[j]-facePosY[j-1];
    geo[2][0][j]=0;
    geo[0][nx-1][j]=0;
    geo[1][nx-1][j]=facePosY[j]-facePosY[j-1];
    geo[2][nx-1][j]=0;
}
//inner nodes geometrical properties
for (int i=1;i<=(nx-2);i++){
    for (int j=1;j<=(ny-2);j++){
        geo[0][i][j]=geo[0][i][0];
        geo[1][i][j]=geo[1][0][j];
        geo[2][i][j]=geo[0][i][j]*geo[1][i][j];
    }
}

//density matrix
for (int i=0;i<=(nx-1);i++){
    for (int j=0;j<=(ny-1);j++){
        rhoMat[i][j]=rho0;
```

```
    }
}

//velocity field
for (int i=0;i<=(nx-1);i++){
    for (int j=0;j<=(ny-1);j++){
        vMat[0][i][j]=velocityField(nodePos[0][i][j],nodePos[1][i][j],0);
        vMat[1][i][j]=velocityField(nodePos[0][i][j],nodePos[1][i][j],1);
    }
}

//convection properties matrixes
for (int i=0;i<=(nx-1);i++){
    for (int j=0;j<=(ny-1);j++){
        gammaMat[i][j]=gamma;
        SMat[0][i][j]=Sc;
        SMat[1][i][j]=Sp;
    }
}
//initial phi values
for (int i=0;i<=(nx-1);i++){
    for (int j=0;j<=(ny-1);j++){
        phiPrev[i][j]=phiIn;
    }
}

//coefficients initialization
for (int i=0;i<=(nx-1);i++){
    for (int j=0;j<=(ny-1);j++){
        coef[0][i][j]=1;
        coef[1][i][j]=0;
        coef[2][i][j]=0;
        coef[3][i][j]=0;
        coef[4][i][j]=0;
        coef[5][i][j]=0;
    }
}

}

void boundaryConditions(double coef[6][nx][ny],double nodePos[2][nx][ny],
                        double gammaMat[nx][ny]){
//inlet&outlet
for (int j=0;j<=(ny-1);j++){
    coef[5][0][j]=phiIn;
    coef[5][nx-1][j]=phiOut;
```

```
}
//top&bottom
for (int i=1;i<=(nx-2);i++){
    coef[4][i][ny-1]=1;
    coef[5][i][ny-1]=flux*(nodePos[1][i][ny-1]-nodePos[1][i][ny-2])
    /gammaMat[i][ny-1];
    coef[3][i][0]=1;
    coef[5][i][0]=flux*(nodePos[1][i][1]-nodePos[1][i][0])
    /gammaMat[i][0];
}

}

void upwindScheme(int i,int j,double rhoMat[nx][ny],double vMat[2][nx][ny],
                  double geo[3][nx][ny],double fVec[4]){
//fe
if (rhoMat[i][j]*vMat[0][i][j]*geo[1][i][j]>=0){
    fVec[0]=0;
}else{
    fVec[0]=1;
}
//fw
if (rhoMat[i][j]*vMat[0][i][j]*geo[1][i][j]<=0){
    fVec[1]=0;
}else{
    fVec[1]=1;
}
//fn
if (rhoMat[i][j]*vMat[1][i][j]*geo[0][i][j]>=0){
    fVec[2]=0;
}else{
    fVec[2]=1;
}
//fs
if (rhoMat[i][j]*vMat[1][i][j]*geo[0][i][j]<=0){
    fVec[3]=0;
}else{
    fVec[3]=1;
}
}

void coefEval(double rhoMat[nx][ny],double vMat[2][nx][ny],
              double geo[3][nx][ny],double fVec[4],double coef[6][nx][ny],
              double gammaMat[nx][ny],double nodePos[2][nx][ny],
              double SMat[2][nx][ny],double phiPrev[nx][ny]){
for (int i=1;i<=(nx-2);i++){
```

```
    for (int j=1;j<=(ny-2);j++){
        //upwind coefficient f evaluation
        upwindScheme(i,j,rhoMat,vMat,geo,fVec);
        //ae
        coef[1][i][j]=-1*rhoMat[i][j]*vMat[0][i][j]*geo[1][i][j]*fVec[0]
        +(gammaMat[i][j]*geo[1][i][j])/(nodePos[0][i+1][j]-nodePos[0][i][j]);
        //aw
        coef[2][i][j]=rhoMat[i][j]*vMat[0][i][j]*geo[1][i][j]*fVec[1]
        +(gammaMat[i][j]*geo[1][i][j])/(nodePos[0][i][j]-nodePos[0][i-1][j]);
        //an
        coef[3][i][j]=-1*rhoMat[i][j]*vMat[1][i][j]*geo[0][i][j]*fVec[2]
        +(gammaMat[i][j]*geo[0][i][j])/(nodePos[1][i][j+1]-nodePos[1][i][j]);
        //as
        coef[4][i][j]=rhoMat[i][j]*vMat[1][i][j]*geo[0][i][j]*fVec[3]
        +(gammaMat[i][j]*geo[0][i][j])/(nodePos[1][i][j]-nodePos[1][i][j-1]);
        //ap
        coef[0][i][j]=(geo[2][i][j]*rhoMat[i][j])/deltat+coef[1][i][j]
        +rhoMat[i][j]*vMat[0][i][j]*geo[1][i][j]+coef[2][i][j]-rhoMat[i][j]
        *vMat[0][i][j]*geo[1][i][j]+coef[3][i][j]+rhoMat[i][j]*vMat[1][i][j]
        *geo[0][i][j]+coef[4][i][j]-rhoMat[i][j]*vMat[1][i][j]*geo[0][i][j]
        +geo[2][i][j]*SMat[1][i][j];
        //b
        coef[5][i][j]=(geo[2][i][j]*rhoMat[i][j]*phiPrev[i][j])/deltat
        +SMat[0][i][j]*geo[2][i][j];
    }
}
}

void bstarCoefCalcRow(int j,double bstarx[nx],double coef[6][nx][ny],
                      double phiEst[nx][ny]){
//inner rows
if (j!=0&&j!=(ny-1)){
    for (int i=0;i<=(nx-1);i++){
        bstarx[i]=coef[5][i][j]+coef[3][i][j]*phiEst[i][j+1]+coef[4][i][j]
        *phiEst[i][j-1];
    }
}else if(j==0){ //bottom row
    for (int i=0;i<=(nx-1);i++){
        bstarx[i]=coef[5][i][j]+coef[3][i][j]*phiEst[i][j+1];
    }
}else if (j==(ny-1)){ //top row
    for (int i=0;i<=(nx-1);i++){
        bstarx[i]=coef[5][i][j]+coef[4][i][j]*phiEst[i][j-1];
    }
}
}
```

```
void TDMArow(double Prow[nx+1],double Qrow[nx+1],double coef[6][nx][ny],
             double phi[nx][ny],double bstarx[nx],int j){
//for computational reasons
Prow[0]=0; Qrow[0]=0;
//coefficient calculation
for (int i=1;i<=nx;i++){
    Prow[i]=coef[1][i-1][j]/(coef[0][i-1][j]-coef[2][i-1][j]*Prow[i-1]);
    Qrow[i]=(bstarx[i-1]+coef[2][i-1][j]*Qrow[i-1])/(coef[0][i-1][j]
                                                    -coef[2][i-1][j]
                                                    *Prow[i-1]);
}
//phi calculation
phi[nx-1][j]=Qrow[nx];
for (int i=(nx-2);i>=0;i--){
    phi[i][j]=Prow[i+1]*phi[i+1][j]+Qrow[i+1];
}

}

void bstarCoefCalcColumn(int i,double bstary[ny],double coef[6][nx][ny],
                         double phi[nx][ny]){
//inner columns
if (i!=0&&i!=(nx-1)){
    for (int j=0;j<=(ny-1);j++){
        bstary[j]=coef[5][i][j]+coef[1][i][j]*phi[i+1][j]+coef[2][i][j]
        *phi[i-1][j];
    }
}else if(i==0){ //leftmost column
    for (int j=0;j<=(ny-1);j++){
        bstary[j]=coef[5][i][j]+coef[1][i][j]*phi[i+1][j];
    }
}else if (i==(nx-1)){ //rightmost column
    for (int j=0;j<=(ny-1);j++){
        bstary[j]=coef[5][i][j]+coef[2][i][j]*phi[i-1][j];
    }

}

}

void TDMAcolumn(double Pcolumn[ny+1],double Qcolumn[ny+1],
                double coef[6][nx][ny],double phi[nx][ny],
                double bstary[ny],int i){
//for computational reasons
Pcolumn[0]=0; Qcolumn[0]=0;
```

```
//coefficient calculation
for (int j=1;j<=ny;j++){
    Pcolumn[j]=coef[3][i][j-1]/(coef[0][i][j-1]-coef[4][i][j-1]*Pcolumn[j-1]);
    Qcolumn[j]=(bstary[j-1]+coef[4][i][j-1]*Qcolumn[j-1])/(coef[0][i][j-1]
                                                          -coef[4][i][j-1]
                                                          *Pcolumn[j-1]);
}
//phi calculation
phi[i][ny-1]=Qcolumn[ny];
for (int j=(ny-2);j>=0;j--){
    phi[i][j]=Pcolumn[j+1]*phi[i][j+1]+Qcolumn[j+1];
}
}

void errorCalc(double phi[nx][ny],double phiPrev[nx][ny],double errMat[nx][ny]){
double diff;
for (int i=0;i<=(nx-1);i++){
    for (int j=0;j<=(ny-1);j++){
        diff=phi[i][j]-phiPrev[i][j];
        errMat[i][j]=(abs(diff))/phiPrev[i][j];
    }
}

}

void solver(double coef[6][nx][ny],double nodePos[2][nx][ny],
            double gammaMat[nx][ny],double& t,double phiEst[nx][ny],
            double phiPrev[nx][ny],double& maxErr,double rhoMat[nx][ny],
            double vMat[2][nx][ny],double geo[3][nx][ny],double fVec[4],
            double SMat[2][nx][ny],double bstarx[nx],double bstary[ny],
            double Prow[nx+1],double Qrow[nx+1],double Pcolumn[ny+1],
            double Qcolumn[ny+1],double phi[nx][ny],double errMat[nx][ny]){
//boundary conditions evaluation
boundaryConditions(coef,nodePos,gammaMat);
//time step evaluation
while (t<=tmax){
    t=t+deltat;
    //phi estimation
    for (int i=0;i<=(nx-1);i++){
        for (int j=0;j<=(ny-1);j++){
            phiEst[i][j]=phiPrev[i][j];
        }
    }
    //iteration
    while (maxErr>delta){
        //coefficient evaluation
```

```
        coefEval(rhoMat,vMat,geo,fVec,coef,gammaMat,nodePos,SMat,phiPrev);
        //line-by-line by rows
        for (int j=0;j<=(ny-1);j++){
            bstarCoefCalcRow(j,bstarx,coef,phiEst);
            TDMArow(Prow,Qrow,coef,phi,bstarx,j);
        }

        //line-by-line by columns
        for (int i=0;i<=(nx-1);i++){
            bstarCoefCalcColumn(i,bstary,coef,phi);
            TDMAcolumn(Pcolumn,Qcolumn,coef,phi,bstary,i);
        }

        //error calculation and check
        errorCalc(phi,phiEst,errMat);
        maxErr=findMax(errMat);
        //new phiEst
        for (int i=0;i<=(nx-1);i++){
            for (int j=0;j<=(ny-1);j++){
                phiEst[i][j]=phi[i][j];
            }
        }
        cout << "time=" << t << " err=" << maxErr << endl;
    }
    maxErr=10;
    //new phiPrev
    for (int i=0; i<=(nx-1);i++){
        for (int j=0; j<=(ny-1);j++){
            phiPrev[i][j]=phi[i][j];
        }
    }
}
}

void postprocessor(double nodePos[2][nx][ny],double phi[nx][ny],
                   double analyticalPhi[nx][ny],double rhoMat[nx][ny]){
//saving results into .dat file
string filename="result.dat";
ofstream datafileR (filename.c_str());
for (int i=0;i<=(nx-1);i++){
    for (int j=0;j<=(ny-1);j++){
        datafileR << nodePos[0][i][j] << " " << nodePos[1][i][j] << " "
       << phi[i][j] << " " << phi[i][j] << endl;
    }
    datafileR << endl;
}
```

```
datafileR.close();
//obtaining analytical solution (parallel flow)
for (int i=0;i<=(nx-1);i++){
    for (int j=0;j<=(ny-1);j++){
        analyticalPhi[i][j]=(exp((nodePos[0][i][j]*(rhoMat[i][j]*l*v0/gamma))
                                 /l)-1)/(exp(rhoMat[i][j]*l*v0/gamma)-1)*
                                 (phiOut-phiIn)+phiIn;
    }
}
//saving analytical results int .dat file
filename="analytical.dat";
ofstream datafileA (filename.c_str());
for (int i=0;i<=(nx-1);i++){
    for (int j=0;j<=(ny-1);j++){
        datafileA << nodePos[0][i][j] << " " << nodePos[1][i][j]
        << " " << analyticalPhi[i][j] << " " << analyticalPhi[i][j] << endl;
    }
    datafileA << endl;
}
datafileA.close();

filename="comparison.dat";
ofstream datafileC (filename.c_str());
for (int i=0;i<=(nx-1);i++){
    datafileC << nodePos[0][i][(int)(ny/2)] << " " << phi[i][(int)(ny/2)]
    << " " << nodePos[0][i][(int)(ny/2)] << " " << analyticalPhi[i][(int)(ny/2)]
     << endl;
}
datafileC.close();
}

int main()
{
    preprocessor(dx,dy,facePosX,facePosY,nodePos,geo,rhoMat,vMat,
                 gammaMat,SMat,phiPrev,coef);

    solver(coef,nodePos,gammaMat,t,phiEst,phiPrev,maxErr,rhoMat,vMat,geo,fVec,
           SMat,bstarx,bstary,Prow,Qrow,Pcolumn,Qcolumn,phi,errMat);
    postprocessor(nodePos,phi,analyticalPhi,rhoMat);
}
```

Attachment E

NAVIER-STOKES CODE

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <stdlib.h>
#include <math.h>

using namespace std;

const int nx=81, ny=81; //number of nodes in each dimension
const double delta=0.0001; //relative error acceptance
const double deltat=0.0001; //relative error between time-steps
const double tmax=200; //max simulation time (s)
const double h=1.0, l=1.0; //geometrical definitions (m)
const double uref=1.0;
const double Re=1000;
const double mu=0.01; //viscosity in Pa*s
const double rho=mu*Re/(l*uref); //density (kg/m^3)
const double P0=0; //ref pressure in Pa

double dx, dy; //nodal lengths
//face positions
double facePosX[nx-1];
double facePosY[ny-1];
//node positions 0-> x 1-> y
double nodePos[2][nx][ny], nodePosStagX[2][nx-1][ny], nodePosStagY[2][nx][ny-1];
//geometric properties 0-> Sx 1-> Sy 2-> V
double geo[3][nx][ny];
//physical properties
double rhoMat[nx][ny], rhoMatStagX[nx-1][ny], rhoMatStagY[nx][ny-1];
double muMat[nx][ny];
```

```
//v matrixes for staggered meshes
double uprev[nx-1][ny], u[nx-1][ny], unext[nx-1][ny];
double vprev[nx][ny-1], v[nx][ny-1], vnext[nx][ny-1];
double unode[nx][ny], vnode[nx][ny], unodeprev[nx][ny], vnodeprev[nx][ny];

//Pressure matrixes
double P[nx][ny], Pprev[nx][ny], PEst[nx][ny];

//mass flows
double muxPrev[nx][ny], mux[nx][ny], muy[nx][ny], muyPrev[nx][ny];
double mvxPrev[nx][ny], mvx[nx][ny], mvyPrev[nx][ny], mvy[nx][ny];
//
double coef[6][nx][ny]; //coefficients matrix
double bstarx[nx], bstary[ny]; //line-by-line coeffs
double Prow[nx], Qrow[nx], Pcol[ny], Qcol[ny]; //TDMA coeffs
double errMat[nx][ny], errU[nx-1][ny],errV[nx][ny-1]; // error matrixes
double maxErr=10, maxErrTime=10; // maximum errors
double timeStep; //time-step
double t=0; //current time

//R matrixes and v^P matrixes
double Ru[nx-1][ny], RuPrev[nx-1][ny];
double Rv[nx][ny-1], RvPrev[nx][ny-1];
double vpu[nx-1][ny], vpv[nx][ny-1];

double vresult[2][nx][ny];


double abs(double a)
{
    if (a<0) a=-a;
    return a;
}

double findMin(double a,double b)
{
    double aux;
    if (a<=b) aux=a;
    else aux=b;
    return aux;
}

double findMax(double errMat[nx][ny])
{
    double err=0;
    for (int i=0;i<=(nx-1);i++){
```

```
        for (int j=0;j<=(ny-1);j++){
            if (errMat[i][j]>err) err=errMat[i][j];
        }
    }
    return err;
}

double findMaxU(double errMat[nx-1][ny])
{
    double err=0;
    for (int i=0;i<=(nx-2);i++){
        for (int j=0;j<=(ny-1);j++){
            if (errMat[i][j]>err) err=errMat[i][j];
        }
    }
    return err;
}

double findMaxV(double errMat[nx][ny-1])
{
    double err=0;
    for (int i=0;i<=(nx-1);i++){
        for (int j=0;j<=(ny-2);j++){
            if (errMat[i][j]>err) err=errMat[i][j];
        }
    }
    return err;
}

void preprocessor(double& dx,double& dy,double facePosX[nx-1],
                  double facePosY[ny-1],double nodePos[2][nx][ny],
                  double nodePosStagX[2][nx-1][ny],
                  double nodePosStagY[2][nx][ny-1],double geo[3][nx][ny],
                  double rhoMat[nx][ny],double muMat[nx][ny],
                  double rhomatStagX[nx-1][ny],double rhoMatStagY[nx][ny-1],
                  double Pprev[nx][ny],double coef[6][nx][ny],
                  double u[nx-1][ny],double uprev[nx-1][ny],
                  double v[nx][ny-1],double vprev[nx][ny-1],
                  double RuPrev[nx-1][ny],double RvPrev[nx][ny-1])
{
    //nodal lengths
    dx=l/(nx-2);
    dy=h/(ny-2);
    //faces positions
    for (int i=0; i<=(nx-2);i++){
        facePosX[i]=i*dx;
```

```
}

for (int j=0;j<=(ny-2);j++){
    facePosY[j]=j*dy;
}

//corner node positions
nodePos[0][0][0]=0;
nodePos[1][0][0]=0;
nodePos[0][nx-1][0]=l;
nodePos[1][nx-1][0]=0;
nodePos[0][0][ny-1]=0;
nodePos[1][0][ny-1]=h;
nodePos[0][nx-1][ny-1]=l;
nodePos[1][nx-1][ny-1]=h;
//top/bottom boundary nodes positions
for (int i=1;i<=(nx-2);i++){
    nodePos[0][i][0]=(facePosX[i]+facePosX[i-1])/2;
    nodePos[1][i][0]=0;
    nodePos[0][i][ny-1]=(facePosX[i]+facePosX[i-1])/2;
    nodePos[1][i][ny-1]=h;
}
//left/right boundary nodes positions
for (int j=1;j<=(ny-2);j++){
    nodePos[0][0][j]=0;
    nodePos[1][0][j]=(facePosY[j]+facePosY[j-1])/2;
    nodePos[0][nx-1][j]=l;
    nodePos[1][nx-1][j]=(facePosY[j]+facePosY[j-1])/2;
}
//inner nodes positions
for (int i=1;i<=(nx-2);i++){
    for (int j=1;j<=(ny-2);j++){
        nodePos[0][i][j]=nodePos[0][i][0];
        nodePos[1][i][j]=nodePos[1][0][j];
    }
}

//staggX nodePos
for (int i=0;i<=(nx-2);i++){
    for (int j=0;j<=(ny-1);j++){
        nodePosStagX[0][i][j]=facePosX[i];
        nodePosStagX[1][i][j]=nodePos[1][i][j];
    }
}

//staggY nodePos
```

```
for (int i=0;i<=(nx-1);i++){
    for (int j=0;j<=(ny-2);j++){
        nodePosStagY[0][i][j]=nodePos[0][i][j];
        nodePosStagY[1][i][j]=facePosY[j];
    }
}

//corner nodes geometrical properties
geo[0][0][0]=0;
geo[1][0][0]=0;
geo[2][0][0]=0;
geo[0][nx-1][0]=0;
geo[1][nx-1][0]=0;
geo[2][nx-1][0]=0;
geo[0][0][ny-1]=0;
geo[1][0][ny-1]=0;
geo[2][0][ny-1]=0;
geo[0][nx-1][ny-1]=0;
geo[1][nx-1][ny-1]=0;
geo[2][nx-1][ny-1]=0;
//top/bottom boundary nodes geometrical properties
for (int i=1;i<=(nx-2);i++){
    geo[0][i][0]=facePosX[i]-facePosX[i-1];
    geo[1][i][0]=0;
    geo[2][i][0]=0;
    geo[0][i][ny-1]=facePosX[i]-facePosX[i-1];
    geo[1][i][ny-1]=0;
    geo[2][i][ny-1]=0;
}
//left/right boundary nodes geometrical properties
for (int j=1;j<=(ny-2);j++){
    geo[0][0][j]=0;
    geo[1][0][j]=facePosY[j]-facePosY[j-1];
    geo[2][0][j]=0;
    geo[0][nx-1][j]=0;
    geo[1][nx-1][j]=facePosY[j]-facePosY[j-1];
    geo[2][nx-1][j]=0;
}
//inner nodes geometrical properties
for (int i=1;i<=(nx-2);i++){
    for (int j=1;j<=(ny-2);j++){
        geo[0][i][j]=geo[0][i][0];
        geo[1][i][j]=geo[1][0][j];
        geo[2][i][j]=geo[0][i][j]*geo[1][i][j];
    }
}
```

```
//phys props
for (int i=0;i<=(nx-1);i++){
    for (int j=0;j<=(ny-1);j++){
        rhoMat[i][j]=rho;
        muMat[i][j]=mu;
    }
}

for (int i=0;i<=(nx-2);i++){
    for (int j=0;j<=(ny-1);j++){
        rhoMatStagX[i][j]=rho;
    }
}

for (int i=0;i<=(nx-1);i++){
    for (int j=0;j<=(ny-2);j++){
        rhoMatStagY[i][j]=rho;
    }
}
//initial map

for (int i=0;i<=(nx-2);i++){
    for (int j=0;j>=(ny-1);j++){
        u[i][j]=0;
        uprev[i][j]=0;
        RuPrev[i][j]=0;
    }
}
for (int i=0;i<=(nx-1);i++){
    for (int j=0;j>=(ny-2);j++){
        v[i][j]=0;
        vprev[i][j]=0;
        RvPrev[i][j]=0;
    }
}

//coefficients initialization
for (int i=0;i<=(nx-1);i++){
    for (int j=0;j<=(ny-1);j++){
        coef[0][i][j]=1;
        coef[1][i][j]=0;
        coef[2][i][j]=0;
        coef[3][i][j]=0;
        coef[4][i][j]=0;
        coef[5][i][j]=0;
```

```
        }
    }
}

double stepSel(double dx,double dy,double rhoMat[nx][ny],double muMat[nx][ny],
               double u[nx-1][ny],double v[nx][ny-1])
{
    double tc=100,td=100;
    double vMod;
    double aux;

    for (int i=0;i<=(nx-1);i++){
        for (int j=0;j<=(ny-1);j++){
            vMod=sqrt(pow(v[i][j],2.0)+pow(u[i][j],2.0));

            if(i==0||i==nx-1){
                aux=0.35*(findMin(dx/2,dy))/abs(vMod);
                if (aux<tc) tc=aux;
                aux=0.2*(rhoMat[i][j]*pow(findMin(dx/2,dy),2.0)/muMat[i][j]);
                if (aux<td) td=aux;
            }else if (j==0||j==ny-1){
                aux=0.35*(findMin(dx,dy/2))/abs(vMod);
                if (aux<tc) tc=aux;
                aux=0.2*(rhoMat[i][j]*pow(findMin(dx,dy/2),2.0)/muMat[i][j]);
                if (aux<td) td=aux;
            }else{
                aux=0.35*(findMin(dx,dy))/abs(vMod);
                if (aux<tc) tc=aux;
                aux=0.2*(rhoMat[i][j]*pow(findMin(dx,dy),2.0)/muMat[i][j]);
                if (aux<td) td=aux;
            }

        }

    }

    aux=findMin(tc,td);

    return aux;
}

void vConditions(double u[nx-1][ny],double v[nx][ny-1])
{
    for(int i=0;i<=nx-2;i++){
        u[i][ny-1]=uref;
        v[i][ny-2]=0;
```

```
    }
    v[nx-1][ny-2]=0;


}

void Reval(double muxPrev[nx][ny],double mux[nx][ny],double mvxPrev[nx][ny],
           double mvx[nx][ny],double muyPrev[nx][ny],double muy[nx][ny],
           double mvyPrev[nx][ny],double mvy[nx][ny],double RuPrev[nx-1][ny],
           double Ru[nx-1][ny],double RvPrev[nx][ny-1],double Rv[nx][ny-1],
           double nodePosStagX[2][nx-1][ny],double nodePosStagY[2][nx][ny-1],
           double u[nx-1][ny],double uprev[nx-1][ny],double v[nx][ny-1],
           double vprev[nx][ny-1],double unodeprev[nx][ny],double unode[nx][ny],
           double vnodeprev[nx][ny],double vnode[nx][ny])
{
    //massflow vx staggX
    for (int j=0;j<=(ny-1);j++){
        for (int i=0;i<=(nx-3);i++){
            mux[i][j]=0.5*(u[i][j]+u[i+1][j])*rho;
        }
        mux[nx-2][j]=rho*u[nx-2][j];
    }
    //massflow vy staggY
    for (int i=0;i<=(nx-1);i++){
        for (int j=0;j<=(ny-3);j++){
            mvy[i][j]=0.5*(v[i][j]+v[i][j+1])*rho;
        }
        mvy[i][ny-2]=rho*v[i][ny-2];
    }

    //vx upwind scheme
    for (int j=0;j<=(ny-1);j++){
        for (int i=0;i<=(nx-3);i++){
            if (mux[i][j]>=0) unode[i+1][j]=u[i][j];
            else unode[i+1][j]=u[i+1][j];
        }
        unode[0][j]=u[0][j];
        unode[nx-1][j]=u[nx-2][j];
    }

    //vy upwind scheme
    for (int i=0;i<=(nx-1);i++){
        for (int j=0;j<=(ny-3);j++){
            if (mvy[i][j]>=0) vnode[i][j+1]=v[i][j];
            else vnode[i][j+1]=v[i][j+1];
        }
```

```
        vnode[i][0]=v[i][0];
        vnode[i][ny-1]=v[i][ny-2];
    }
    //massflow vy staggX
    for (int i=1;i<=(nx-2);i++){
        for (int j=0;j<=(ny-2);j++){
            mvx[i][j]=0.5*rho*(v[i][j]+v[i+1][j]);
        }
        mvx[i][ny-1]=rho*v[i][ny-2];
    }
    for (int j=0;j<=ny-2;j++){
        mvx[0][j]=rho*v[0][j];
        mvx[nx-1][j]=rho*v[nx-1][j];
    }
    mvx[0][ny-1]=rho*v[0][ny-2];
    //massflow vx staggY
    for (int j=1;j<=(ny-2);j++){
        for (int i=0;i<=(nx-2);i++){
            muy[i][j]=0.5*rho*(u[i][j]+u[i][j+1]);
        }
        muy[nx-1][j]=rho*u[nx-2][j];
    }
    for (int i=0;i<=nx-2;i++){;
        muy[i][0]=rho*u[i][ny-1];
        muy[i][ny-1]=rho*u[i][ny-1];
    }
    muy[nx-1][0]=rho*u[nx-2][0];

    //Ru calc
    for (int i=1;i<=(nx-3);i++){
        for (int j=1;j<=(ny-2);j++){
            Ru[i][j]=(-(mux[i][j]*unode[i+1][j]*geo[1][i][j]-mux[i-1][j]*
                        unode[i][j]*geo[1][i][j]+(u[i][j]+u[i][j+1])*0.5
                        *geo[0][i][j]*mvx[i][j]-(u[i][j-1]+u[i][j])*0.5
                        *geo[1][i][j]*mvx[i][j-1])+(mu*geo[1][i][j]*(u[i+1][j]
                                                                    -u[i][j])
                                                   /(nodePosStagX[0][i+1][j]
                                                     -nodePosStagX[0][i][j])
                                                   -mu*geo[1][i][j]*(u[i][j]
                                                        -u[i-1][j])
                                                   /(nodePosStagX[0][i][j]
                                                     -nodePosStagX[0][i-1][j])
                                                   +mu*geo[0][i][j]*(u[i][j+1]
                                                                    -u[i][j])
                                                   /(nodePosStagX[1][i][j+1]
                                                     -nodePosStagX[1][i][j])
```

```
                                                    -mu*geo[0][i][j]*(u[i][j]
                                                            -u[i][j-1])
                                                    /(nodePosStagX[1][i][j]
                                                -nodePosStagX[1][i][j-1])))
                                                /geo[2][i][j];
        }
    }
    for (int j=1;j<=(ny-2);j++){
        Ru[0][j]=0;
        Ru[nx-2][j]=0;
    }
    for (int i=0;i<=(nx-2);i++){
        Ru[i][0]=0;
        Ru[i][ny-1]=0;
    }
    //Rv Calc
    for (int i=1;i<=(nx-2);i++){
        for (int j=1;j<=(ny-3);j++){
            Rv[i][j]=(-(mvy[i][j]*vnode[i][j+1]*geo[0][i][j]-mvy[i][j-1]
                        *vnode[i][j]*geo[0][i][j]+(v[i][j]+v[i+1][j])*0.5
                        *geo[1][i][j]*muy[i][j]-(v[i-1][j]+v[i][j])
                        *0.5*geo[0][i][j]*muy[i-1][j])+(mu*geo[1][i][j]
                                                        *(v[i+1][j]-v[i][j])
                                                    /(nodePosStagY[0][i+1][j]
                                                      -nodePosStagY[0][i][j])
                                                        -mu*geo[1][i][j]
                                                        *(v[i][j]-v[i-1][j])
                                                        /(nodePosStagY[0][i][j]
                                                    -nodePosStagY[0][i-1][j])
                                                        +mu*geo[0][i][j]
                                                        *(v[i][j+1]-v[i][j])
                                                    /(nodePosStagY[1][i][j+1]
                                                      -nodePosStagY[1][i][j])
                                                    -mu*geo[0][i][j]*(v[i][j]
                                                                -v[i][j-1])
                                                        /(nodePosStagY[1][i][j]
                                                -nodePosStagY[1][i][j-1])))
                                                /geo[2][i][j];
        }
    }
    for (int i=1;i<=(nx-2);i++){
        Rv[i][0]=0;
        Rv[i][ny-2]=0;
    }
    for (int j=0;j<=(ny-1);j++){
        Rv[0][j]=0;
```

```
        Rv[nx-1][j]=0;
    }

}

void vPeval(double timeStep,double vpu[nx-1][ny],double vpv[nx][ny-1],
            double Ru[nx-1][ny],double RuPrev[nx-1][ny],double RvPrev[nx][ny-1],
            double Rv[nx][ny-1],double u[nx-1][ny],double v[nx][ny-1])
{
    for (int i=0;i<=(nx-2);i++){
        for (int j=0;j<=(ny-1);j++){
            vpu[i][j]=u[i][j]+(timeStep/rho)*((3.0/2.0)*Ru[i][j]
                                              -(1.0/2.0)*RuPrev[i][j]);
        }
    }
    for (int i=0;i<=nx-1;i++){
        for (int j=0;j<=ny-2;j++){
            vpv[i][j]=v[i][j]+(timeStep/rho)*((3.0/2.0)*Rv[i][j]
                                              -(1.0/2.0)*RvPrev[i][j]);
        }
    }

}

void coefEval(double coef[6][nx][ny],double geo[3][nx][ny],
              double nodePos[2][nx][ny],double rhoMat[nx][ny],
              double timeStep,double vpu[nx-1][ny],double vpv[nx][ny-1])
{
    for(int i=1;i<=(nx-2);i++){
        for(int j=1;j<=(ny-2);j++){
            coef[1][i][j]=geo[1][i][j]/(nodePos[0][i+1][j]-nodePos[0][i][j]);
            coef[2][i][j]=geo[1][i][j]/(nodePos[0][i][j]-nodePos[0][i-1][j]);
            coef[3][i][j]=geo[0][i][j]/(nodePos[1][i][j+1]-nodePos[1][i][j]);
            coef[4][i][j]=geo[0][i][j]/(nodePos[1][i][j]-nodePos[1][i][j-1]);
            coef[0][i][j]=coef[1][i][j]+coef[2][i][j]+coef[3][i][j]
            +coef[4][i][j];
            coef[5][i][j]=-(1/timeStep)*(rhoMat[i][j]*vpu[i][j]*geo[1][i][j]
                                         -rhoMat[i][j]*vpu[i-1][j]*geo[1][i][j]
                                         +rhoMat[i][j]*vpv[i][j]*geo[0][i][j]
                                         -rhoMat[i][j]*vpv[i][j-1]
                                         *geo[0][i][j]);
        }
    }
}

void boundaryCoef(double coef[6][nx][ny])
```

```
{
    for(int i=0;i<=(nx-1);i++){
        coef[3][i][0]=1;
        coef[4][i][ny-1]=1;
    }
    for (int j=1;j<=(ny-2);j++){
        coef[1][0][j]=1;
        coef[2][nx-1][j]=1;
    }
    coef[0][1][1]=1;
    coef[1][1][1]=0;
    coef[2][1][1]=0;
    coef[3][1][1]=0;
    coef[4][1][1]=0;
    coef[5][1][1]=P0;
}

void bstarCalcRow(double bstarx[nx],double coef[6][nx][ny],
                  double PEst[nx][ny],int j)
{
    if (j!=0&&j!=(ny-1)){
        for (int i=0;i<=(nx-1);i++){
            bstarx[i]=coef[5][i][j]+coef[3][i][j]*PEst[i][j+1]
            +coef[4][i][j]*PEst[i][j-1];
        }
    }
    else if (j==0){ //bottom row
        for (int i=0;i<=(nx-1);i++){
            bstarx[i]=coef[5][i][j]+coef[3][i][j]*PEst[i][j+1];
        }
    }
    else if (j==ny-1){ //top row
        for (int i=0;i<=(nx-1);i++){
            bstarx[i]=coef[5][i][j]+coef[4][i][j]*PEst[i][j-1];
        }
    }

}

void TDMArow(double Prow[nx],double Qrow[nx],double bstarx[nx],
             double coef[6][nx][ny],double P[nx][ny],int j)
{
    //coef calc
    Prow[0]=coef[1][0][j]/coef[0][0][j];
    Qrow[0]=bstarx[0]/coef[0][0][j];
    for (int i=1;i<=(nx-1);i++){
```

```
        Prow[i]=coef[1][i][j]/(coef[0][i][j]-coef[2][i][j]*Prow[i-1]);
        Qrow[i]=(bstarx[i]+coef[2][i][j]*Qrow[i-1])/(coef[0][i][j]
                                                    -coef[2][i][j]*Prow[i-1]);
    }
    //P calc
    P[nx-1][j]=Qrow[nx-1];
    for (int i=(nx-2);i>=0;i--){
        P[i][j]=Prow[i]*P[i+1][j]+Qrow[i];
    }


}

void bstarCalcCol(double bstary[ny],double coef[6][nx][ny],
                  double P[nx][ny],int i)
{
    if (i!=0&&i!=(nx-1)){
        for (int j=0;j<=(ny-1);j++){
            bstary[j]=coef[5][i][j]+coef[1][i][j]*P[i+1][j]+coef[2][i][j]
            *P[i-1][j];
        }
    }
    else if (i==0){ //left col
        for (int j=0;j<=(ny-1);j++){
            bstary[j]=coef[5][i][j]+coef[1][i][j]*P[i+1][j];
        }
    }
    else if (i==nx-1){ //right col
        for (int j=0;j<=(ny-1);j++){
            bstary[j]=coef[5][i][j]+coef[2][i][j]*P[i-1][j];
        }
    }

}

void TDMAcol(double Pcol[ny],double Qcol[ny],double bstary[ny],
             double coef[6][nx][ny],double P[nx][ny],int i)
{
    //coef calc
    Pcol[0]=coef[3][i][0]/coef[0][i][0];
    Qcol[0]=bstary[0]/coef[0][i][0];
    for (int j=1;j<=(ny-1);j++){
        Pcol[j]=coef[3][i][j]/(coef[0][i][j]-coef[4][i][j]*Pcol[j-1]);
        Qcol[j]=(bstary[j]+coef[4][i][j]*Qcol[j-1])/(coef[0][i][j]
                                                    -coef[4][i][j]*Pcol[j-1]);
    }
```

```
    //P calc
    P[i][ny-1]=Qcol[ny-1];
    for (int j=(ny-2);j>=0;j--){
        P[i][j]=Pcol[j]*P[i][j+1]+Qcol[j];
    }

}

void errorCalc(double P[nx][ny],double PEst[nx][ny],double errMat[nx][ny])
{
    double diff;
    for (int i=0;i<=(nx-1);i++){
        for (int j=0;j<=(ny-1);j++){
            diff=P[i][j]-PEst[i][j];
            errMat[i][j]=abs(diff)/PEst[i][j];
        }
    }
}

void vNextCalc(double P[nx][ny],double geo[3][nx][ny],double timeStep,
               double vpu[nx-1][ny],double vpv[nx][ny-1],double unext[nx-1][ny],
               double vnext[nx][ny-1])
{
    //vx next
    for (int j=1;j<=ny-2;j++){
        for (int i=1;i<=nx-3;i++){
            unext[i][j]=vpu[i][j]-timeStep/rho*(P[i+1][j]-P[i][j])
            /(nodePos[0][i+1][j]-nodePos[0][i][j]);
        }
        unext[0][j]=vpu[0][j];
        unext[nx-2][j]=vpu[nx-2][j];
    }
    for (int i=0;i<=nx-2;i++){
        unext[i][0]=vpu[i][0];
        unext[i][ny-1]=vpu[i][ny-1];
    }
    //vy next
    for (int i=1;i<=nx-2;i++){
        for (int j=1;j<=ny-3;j++){
            vnext[i][j]=vpv[i][j]-timeStep/rho*(P[i][j+1]-P[i][j])
            /(nodePos[1][i][j+1]-nodePos[1][i][j]);
        }
        vnext[i][0]=vpv[i][0];
        vnext[i][ny-2]=vpv[i][ny-2];
    }
```

```
    for (int j=0;j<=ny-2;j++){
        vnext[0][j]=vpv[0][j];
        vnext[nx-1][j]=vpv[nx-1][j];
    }

}

void errorTime(double unext[nx-1][ny],double u[nx-1][ny],double errU[nx-1][ny],
               double vnext[nx][ny-1],double v[nx][ny-1],double errV[nx][ny-1])
{
    double diff1,diff2;
    for(int i=0;i<=nx-2;i++){
        for (int j=0;j<=ny-1;j++){
            diff1=unext[i][j]-u[i][j];
            errU[i][j]=abs(diff1)/(u[i][j]);
        }
    }
    for (int j=0;j<=ny-2;j++){
        for (int i=0;i<=nx-1;i++){
            diff2=vnext[i][j]-v[i][j];
            errV[i][j]=abs(diff2)/(v[i][j]);
        }
    }

}

void solver(double t,double timeStep,double dx,double dy,double Pprev[nx][ny],
            double PEst[nx][ny],double maxErr,double maxErrTime,
            double coef[6][nx][ny],double geo[3][nx][ny],
            double nodePos[2][nx][ny],double bstarx[nx],double bstary[ny],
            double Prow[nx],double Qrow[nx],double Pcol[ny],double Qcol[ny],
            double P[nx][ny],double errMat[nx][ny],double mux[nx][ny],
            double muxPrev[nx][ny],double muy[nx][ny],double muyPrev[nx][ny],
            double mvx[nx][ny],double mvxPrev[nx][ny],double mvy[nx][ny],
            double mvyPrev[nx][ny],double Ru[nx-1][ny],double RuPrev[nx-1][ny],
            double Rv[nx][ny-1],double RvPrev[nx][ny-1],double vpu[nx-1][ny],
            double vpv[nx][ny-1],double uprev[nx-1][ny],double u[nx-1][ny],
            double unext[nx-1][ny],double errU[nx-1][ny],double vprev[nx][ny-1],
            double v[nx][ny-1],double vnext[nx][ny-1],double errV[nx][ny-1],
            double vresult[2][nx][ny])
{
    while(t<tmax&&maxErrTime>=deltat){
        vConditions(u,v);
        timeStep=stepSel(dx,dy,rhoMat,muMat,u,v);
        t=t+timeStep;
        Reval(muxPrev,mux,mvxPrev,mvx,muyPrev,muy,mvyPrev,mvy,RuPrev,Ru,RvPrev,
```

```
        Rv, nodePosStagX, nodePosStagY, u, uprev, v, vprev, unodeprev, unode,
        vnodeprev, vnode);
vPeval(timeStep, vpu, vpv, Ru, RuPrev, RvPrev, Rv, u, v);
//P estimated
for (int i=0;i<=(nx-1);i++){
    for (int j=0;j<=(ny-1);j++){
        PEst[i][j]=Pprev[i][j];
    }
}
while (maxErr>delta){
    coefEval(coef, geo, nodePos, rhoMat, timeStep, vpu, vpv);
    boundaryCoef(coef);

    //line-by-line row
    for (int j=0;j<=(ny-1);j++){
        bstarCalcRow(bstarx, coef, PEst, j);
        TDMArow(Prow, Qrow, bstarx, coef, P, j);
    }
    for (int i=0;i<=(nx-1);i++){
        for (int j=0;j<=(ny-1);j++){
            PEst[i][j]=P[i][j];
        }
    }
    //line-by-line column
    for (int i=0;i<=(nx-1);i++){
        bstarCalcCol(bstary, coef, PEst, i);
        TDMAcol(Pcol, Qcol, bstary, coef, P, i);
    }

    errorCalc(P, PEst, errMat);
    maxErr=findMax(errMat);
    //new PEst
    for (int i=0;i<=(nx-1);i++){
        for (int j=0;j<=(ny-1);j++){
            PEst[i][j]=P[i][j];
        }
    }

}
maxErr=10;
vNextCalc(P, geo, timeStep, vpu, vpv, unext, vnext);

errorTime(unext, u, errU, vnext, v, errV);
if(findMaxU(errU)>=findMaxV(errV)) maxErrTime=findMaxU(errU);
else maxErrTime=findMaxV(errV);
```

```
        for (int i=0;i<=nx-1;i++){
            for (int j=0;j<=ny-1;j++){
                Pprev[i][j]=P[i][j];

            }

        }


        for (int i=0;i<=nx-2;i++){
            for (int j=0;j<=ny-1;j++){
                uprev[i][j]=u[i][j];
                u[i][j]=unext[i][j];
                RuPrev[i][j]=Ru[i][j];

            }

        }
        for (int i=0;i<=nx-1;i++){
            for (int j=0;j<=ny-2;j++){
                vprev[i][j]=v[i][j];
                v[i][j]=vnext[i][j];
                RvPrev[i][j]=Rv[i][j];

            }

        }
        cout << "time=" << t << " time step=" << timeStep<< " err=" << maxErr
        << " errTime=" << maxErrTime <<  endl;

    }
    for (int i=1;i<=nx-2;i++){
        for (int j=1;j<=ny-2;j++){
            vresult[0][i][j]=unext[i][j];
            vresult[1][i][j]=vnext[i][j];
        }
    }
    for (int i=1;i<=nx-2;i++){
        vresult[0][i][0]=unext[i][0];
        vresult[0][i][ny-1]=unext[i][ny-1];
    }
    for (int j=0;j<=ny-1;j++){
        vresult[0][0][j]=unext[0][j];
        vresult[0][nx-1][j]=unext[nx-2][j];
    }
    for (int j=1;j<=ny-2;j++){
```

```
        vresult[1][0][j]=vnext[0][j];
        vresult[1][nx-1][j]=vnext[nx-1][j];
    }
    for (int i=0;i<=nx-1;i++){
        vresult[1][i][0]=vnext[i][0];
        vresult [1][i][ny-1]=vnext[i][ny-2];
    }
}

void postprocessor(double P[nx][ny],double vNext[2][nx][ny],
                   double nodePos[2][nx][ny])
{
    string filename="pressure.dat";
    ofstream datafileP (filename.c_str());
    for (int i=0;i<=(nx-1);i++){
        for (int j=0;j<=(ny-1);j++){
            datafileP << nodePos[0][i][j] << " " << nodePos[1][i][j] << " "
            << P[i][j] << " " << P[i][j] << endl;
        }
        datafileP << endl;
    }
    datafileP.close();
    filename="vx.dat";
    ofstream datafileVX (filename.c_str());
    for (int i=0;i<=(nx-1);i++){
        for (int j=0;j<=(ny-1);j++){
            datafileVX << nodePos[0][i][j] << " " << nodePos[1][i][j]
            << " " << vNext[0][i][j] << " " << vNext[0][i][j] << endl;
        }
        datafileVX << endl;
    }
    datafileVX.close();
    filename="vy.dat";
    ofstream datafileVY (filename.c_str());
    for (int i=0;i<=(nx-1);i++){
        for (int j=0;j<=(ny-1);j++){
            datafileVY << nodePos[0][i][j] << " " << nodePos[1][i][j]
            << " " << vNext[1][i][j] << " " << vNext[1][i][j] << endl;
        }
        datafileVY << endl;
    }
    datafileVY.close();
    filename="vmod.dat";
    ofstream datafileV (filename.c_str());
    for (int i=0;i<=(nx-1);i++){
        for (int j=0;j<=(ny-1);j++){
```

```
            datafileV << nodePos[0][i][j] << " " << nodePos[1][i][j] << " "
            << sqrt(pow(vNext[0][i][j],2.0)+pow(vNext[1][i][j],2.0)) << " "
            << sqrt(pow(vNext[0][i][j],2.0)+pow(vNext[1][i][j],2.0)) << endl;
        }
        datafileV << endl;
    }
    datafileV.close();
    filename="vvec.dat";
    ofstream datafileVe (filename.c_str());
    for (int i=0;i<=(nx-1);i++){
        for (int j=0;j<=(ny-1);j++){
            datafileVe << nodePos[0][i][j] << " " << nodePos[1][i][j] << " "
            << vNext[0][i][j] << " " << vNext[1][i][j] << endl;
        }
        datafileVe << endl;
    }
    datafileVe.close();
    filename="comparisonHor.dat";
    ofstream datafileC1 (filename.c_str());
    for (int i=0;i<=nx-1;i++){
        datafileC1 << nodePos[0][i][(int)(ny/2)] << " "
        << vNext[1][i][(int)(ny/2)] << endl;
    }
    datafileC1.close();
    filename="comparisonVer.dat";
    ofstream datafileC2 (filename.c_str());
    for (int j=0;j<=ny-1;j++){
        datafileC2 << nodePos[1][(int)(nx/2)][j] << " "
        << vNext[0][(int)(nx/2)][j] << endl;
    }
    datafileC2.close();
}

int main()
{
    preprocessor(dx,dy,facePosX,facePosY,nodePos,nodePosStagX,nodePosStagY,geo,
                 rhoMat,muMat,rhoMatStagX,rhoMatStagY,Pprev,coef,u,uprev,v,
                 vprev,RuPrev,RvPrev);
    solver(t,timeStep,dx,dy,Pprev,PEst,maxErr,maxErrTime,coef,geo,nodePos,
           bstarx,bstary,Prow,Qrow,Pcol,Qcol,P,errMat,mux,muxPrev,muy,muyPrev,
           mvx,mvxPrev,mvy,mvyPrev,Ru,RuPrev,Rv,RvPrev,vpu,vpv,uprev,u,unext,
           errU,vprev,v,vnext,errV,vresult);
    postprocessor(P,vresult,nodePos);
}
```