Universitat Politècnica de Catalunya

Escola Superior d'Enginyeries Industrial, Aeroespacial
i Audiovisual de Terrassa

Numerical resolution of Potential Flows

Computational Engineering

220027

Professor

Carlos David Pérez Segarra

Francesc Xavier Trias Miquel

Author

Sergio Gutiérrez Sánchez

Saturday 22nd January, 2022

Abstract

In this report it is presented the numerical resolution of Potential Flows for different geometries and cases. It covers a profound explanation of this type of fluids study and its numerical implementation, from the mesh generation to the solving implementation and techniques used in it. Additionally are presented some studies related to the cases concerning the mesh influence and the paralelisation performance. Finally, the results obtained for each of the cases are presented as well as an explanation for each of them.

1 Introduction

In fluid dynamics, there are several methods to describe or simulate the behaviour of a determined fluid under certain conditions. Many of them require the presence of the viscosity of the fluid.

However, when simulating an aerodynamic object there can be clearly appreciated two different regions in the domain. First, a boundary layer near the solid walls, where viscosity plays a huge role in terms of fluid's behaviour. And the outer region, where the fluid can be considered as inviscid.

This phenomena is what allows to simulate a fluid using Potential Flow theory.

There are two different approaches for this method, Velocity Potential and Stream Function definition. Both of them are based on an irrotational velocity field. In this case, the simulations presented have been solved using Stream Function approach.

1.1 Theoretical approach

Stream function formulation is based on defining an scalar variable ψ, known as Stream Function. This parameter allows to calculate the velocity field of the domain by simply applying the following expressions.

$$v_x = \frac{\rho_{ref}}{\rho} \cdot \frac{\partial \psi}{\partial y} \tag{1}$$

$$v_y = -\frac{\rho_{ref}}{\rho} \cdot \frac{\partial \psi}{\partial x} \tag{2}$$

This velocity field should verify the Mass Conservation Equation $(\nabla \cdot (\rho \mathbf{u}))$ and the irrotational condition.

$$\frac{\partial}{\partial x}\left(\frac{\rho_{ref}}{\rho} \cdot \frac{\partial \psi}{\partial x}\right) + \frac{\partial}{\partial y}\left(\frac{\rho_{ref}}{\rho} \cdot \frac{\partial \psi}{\partial y}\right) = 0 \tag{3}$$

In order to calculate both the Stream Function scalar and velocity fields, equation 3 must be solved.

1.2 Simulation cases

For this report, there have been simulated 3 different cases using the same approach, Potential Flow.

1.2.1 Flow inside a channel

First of them, and the simplest one, is the simulation of the flow inside a channel.

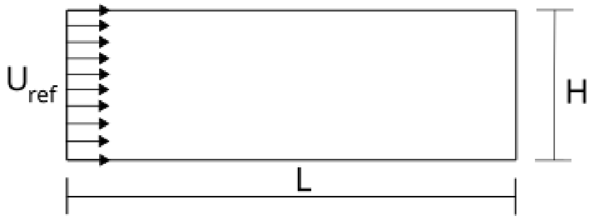

Figure 1: Flow inside a channel case scheme

1.2.2 Flow around a rotating cylinder

The second study case consists on the simulation of a rotating cylinder inside the potential flow.

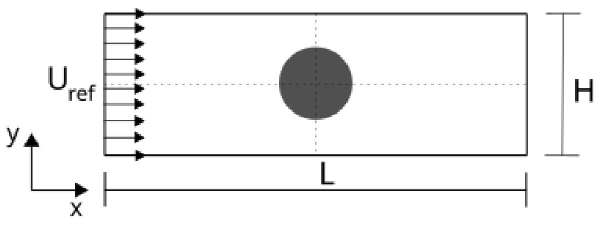

Figure 2: Flow around a cylinder case scheme

As there can be seen, in scheme 2, the flow domain is similar to previous case, and so the inlet, outlet and lower and upper walls boundary conditions.

1.2.3 Flow around NACA 0012 Airfoil

The last study case of the present report is the simulation of a NACA 0012 airfoil with different angles of attack.

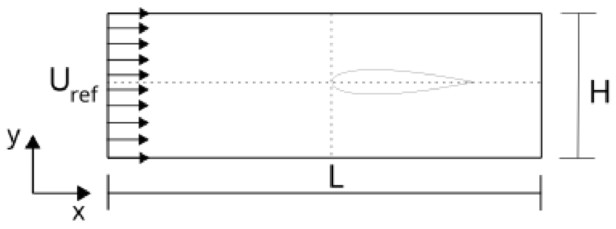

Figure 3: Flow around a NACA 0012 Airfoil case scheme

However, before showing the obtained results for the simulation of the airfoil, there must be mentioned that it may not give high accurate results in what comes to lift calculations. The sources used for the comparison use Navier-Stokes equations for the resolution of the flow and the rest of the parameters. Potential flow is a good and easy tool to simulate these kind of cases, but its accuracy can't compete with other Computational Fluid Dynamics methods.

The results of these 3 cases are presented in section 5.

2 Description of numerical methods

In this section, it is presented a brief description of the numerical methods and techniques used in the resolution of potential flows.

2.1 Spatial Discretization

The first step consists on the spatial discretization of the domain. For this task, Finite Volume Method approach has been selected. This consists on dividing the domain into a determine number of control volumes (figure 4).

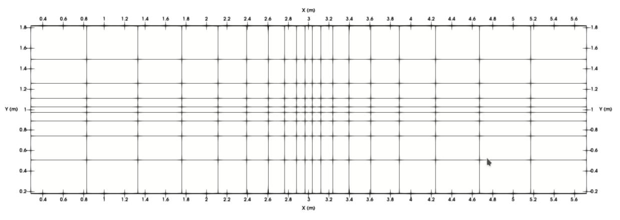

Figure 4: Spatial discretization scheme

As there can be seen in previous figure, the size of the control volumes decreases when approaching the centre of the channel. This is known as variable nodal distribution. In figure 4 has been used a hyperbolic tangent distribution, which results in a much higher nodal density at the centre of the channel.

The mathematical expression used for the node location is the following.

$$x_i = \frac{L}{2} \cdot \left[\frac{tanh(SF \cdot (\frac{2 \cdot i - NX}{NX}))}{tanh(SF)} + 1 \right] \qquad (4)$$

This type of nodal distribution aims to a reduction of the total amount of nodes in the domain while keeping the accuracy and resolution of the results. As it can be observed in section 1.2, the most important part of the Cylinder and Airfoil cases is the centre, and so, it is here where there must be the highest resolution possible.

It is possible to control the nodal density of the distribution thanks to a parameter known as stretch factor (SF) (equation (4)).

However, in order to show the usefulness and advantage of this type of distribution, the mesh influence study (section 5.1) has been done with a regular and hyperbolic tangent distributions.

2.2 Equations Discretization

In order to simulate Potential Flow, it is necessary to solve the Stream Function equation exposed in section 1.1. To do that, Stokes' Theorem is applied based on the irrotational flow condition previously mentioned.

$$\int_S (\nabla \times \mathbf{u}) \partial S = \oint_C \mathbf{u} \cdot \partial \vec{l} \tag{5}$$

Thanks to Stokes' Theorem, rotational surface integral can be transformed into the calculation of surface's circulation. Latter is considered to be zero due to the irrotational condition.

$$\Gamma = \oint_C \mathbf{u} \cdot \partial \vec{l} = 0 \tag{6}$$

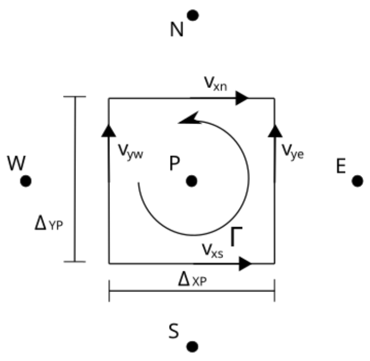

Figure 5: Node velocities scheme

Applying numerical methods to Γ calculation leads to the following expression.

$$\Gamma = v_{ye} \cdot \Delta y_P - v_{xn} \cdot \Delta x_P - v_{yw} \cdot \Delta y_P + v_{ys} \cdot \Delta x_P = 0 \tag{7}$$

And then, replacing velocity terms with equations 1 and 2.

$$\begin{aligned} \Gamma &= -\frac{\rho_{ref}}{\rho} \cdot \frac{\partial \psi}{\partial x} \cdot \Delta y_P \\ &\quad -\frac{\rho_{ref}}{\rho} \cdot \frac{\partial \psi}{\partial y} \cdot \Delta x_P \\ &\quad +\frac{\rho_{ref}}{\rho} \cdot \frac{\partial \psi}{\partial x} \cdot \Delta y_P \\ &\quad +\frac{\rho_{ref}}{\rho} \cdot \frac{\partial \psi}{\partial y} \cdot \Delta x_P \\ &= 0 \end{aligned} \tag{8}$$

And finally, approaching the differentials on each of the control volume faces leads to the following expression.

$$\begin{aligned} \Gamma &= -\frac{\rho_{ref}}{\rho_e} \cdot \frac{\psi_E - \psi_P}{\Delta_{PE}} \cdot \Delta y_P \\ &\quad -\frac{\rho_{ref}}{\rho_n} \cdot \frac{\psi_N - \psi_P}{\Delta_{PN}} \cdot \Delta x_P \\ &\quad +\frac{\rho_{ref}}{\rho_w} \cdot \frac{\psi_P - \psi_W}{\Delta_{PW}} \cdot \Delta y_P \\ &\quad +\frac{\rho_{ref}}{\rho_s} \cdot \frac{\psi_P - \psi_S}{\Delta_{PS}} \cdot \Delta x_P \\ &= 0 \end{aligned} \tag{9}$$

Equation 9 can be rearranged into the following iterative expression.

$$a_P \psi_P = a_W \psi_W + a_E \psi_E + a_S \psi_S + a_N \psi_N \tag{10}$$

Where:

- $a_W = \frac{\rho_{ref}}{\rho_w} \cdot \frac{\Delta y_P}{\Delta_{PW}}$
- $a_E = \frac{\rho_{ref}}{\rho_e} \cdot \frac{\Delta y_P}{\Delta_{PE}}$
- $a_S = \frac{\rho_{ref}}{\rho_s} \cdot \frac{\Delta x_P}{\Delta_{PS}}$
- $a_N = \frac{\rho_{ref}}{\rho_n} \cdot \frac{\Delta x_P}{\Delta_{PN}}$
- $a_P = a_W + a_E + a_S + a_N$

Solving equation 22 gives the Stream Function scalar field (ψ) for all the domain. Then, it is possible to compute other physical variables such as velocity, temperature, pressure and density.

Velocities.

$$v_{xn} = \frac{\rho_{ref}}{\rho_n} \cdot \frac{\psi_N - \psi_P}{\Delta_{PN}} \tag{11}$$

$$v_{xs} = \frac{\rho_{ref}}{\rho_s} \cdot \frac{\psi_P - \psi_S}{\Delta_{PS}} \tag{12}$$

$$v_{yw} = \frac{\rho_{ref}}{\rho_w} \cdot \frac{\psi_P - \psi_W}{\Delta_{PW}} \tag{13}$$

$$v_{ye} = \frac{\rho_{ref}}{\rho_e} \cdot \frac{\psi_E - \psi_P}{\Delta_{PE}} \tag{14}$$

$$v_{yP} = \frac{v_{yw} + v_{ye}}{2} \tag{15}$$

$$v_{xP} = \frac{v_{xs} + v_{xn}}{2} \tag{16}$$

Temperature.

$$T_P = T_{ref} + \frac{Uref^2 - (v_{yP}^2 + v_{xP}^2)}{2 \cdot Cp} \tag{17}$$

Pressure (considering an isentropic relationship).

$$p_P = p_{ref} \cdot (\frac{T_P}{T_{ref}})^{\frac{\gamma}{\gamma-1}} \tag{18}$$

And finally, density.

$$\rho_P = \frac{p_P}{R \cdot T_P} \tag{19}$$

2.3 Blocking Off

Another crucial aspect of the resolution of Potential Flows, and specially for the proposed cases, is the simulation of the solid objects such as the cylinder or the NACA Airfoil.

In order to do that, it is necessary to apply several techniques to the calculation.

First, it is necessary to know "where" are these solids. As it has been presented previously, the spatial domain must be discretized into multiple control volumes. Then, in order to determine which of them belongs to the Cylinder or to the Airfoil solid object Blocking Off method is used.

This consists on selecting if a control volume pertains to the solid or to the fluid by looking where is its centre (figure 6).

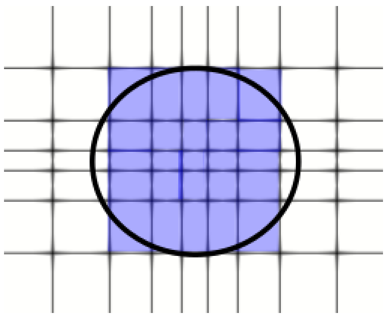

Figure 6: Blocking Off Cylinder scheme

As it can be though, the more nodes, the higher the accuracy of the results. This is the main reason for using a hyperbolic tangent nodal distribution in the mesh generation (section 2.1).

The other numerical techniques concerns the density calculation. To simulate the presence of these solid objects, density must be set to zero in the control volumes that pertain to the solids $\frac{\rho_{ref}}{\rho} = 0$. And, in addition to that, the computation of this parameter at the solid-fluid interface walls of these must be done using a harmonic mean.

$$\frac{\rho_{ref}}{\rho_e} = \frac{\Delta_{PE}}{\frac{\Delta_{Pe}}{\rho_{ref}/\rho_P} + \frac{\Delta_{Ee}}{\rho_{ref}/\rho_E}} \tag{20}$$

2.4 Boundary conditions

The boundary conditions of the simulation cases presented are similar.

In what comes to Stream Function, the inlet and lateral walls conditions are known. In case of these, the value of ψ is set to be $psi = U_{ref} \cdot y$. Therefore, for the inlet, this value is variable through all the entrance, while on the lower and upper parts are 0 and $U_{ref} \cdot H$ respectively.

On the other hand, the Stream Function value of the outlet is unknown, but not its gradient (Newmann condition), which is zero.

$$\frac{\partial \psi}{\partial x}|_{outlet} = 0 \tag{21}$$

Additionally, in case of simulating a solid, it is necessary to set the density of its control volumes to zero (previous section). However, in order to simulate the rotating cylinder, it is necessary to add another condition.

As it has been presented in equations discretization section, in order to solve the Stream Function field, it is necessary to apply equation (22). Nevertheless, in case of having a cylinder, and specially, a rotating cylinder, this equation can't be applied to the solid.

Instead of that, a certain value of ψ is set in all of the control volumes that pertain to the solid. And, depending on this value, the results obtained may be different.

In case of setting this ψ to exactly the mean value between the lower and upper parts of the domain ($\psi = \frac{\psi_{lower}+\psi_{upper}}{2}$), the cylinder doesn't rotate.

If this value of ψ set in the solid is higher than $(\frac{\psi_{lower}+\psi_{upper}}{2})$, the cylinder is rotating counterclockwise, and therefore generating negative lift force.

And finally, if ψ is lower than $(\frac{\psi_{lower}+\psi_{upper}}{2})$, the cylinder is rotating clockwise, and so generating positive lift.

Additionally, there must be considered as boundary conditions the set of reference values for velocity, temperature, pressure and density right at the inlet of the channel.

2.5 Resolution Algorithm

In order to solve each physical variable of the problem, it is necessary first to solve the Stream Function scalar field (equation 22). This consists on a set of equations that can be written in the form of a matrix with dimensions (N × N), where N is the total amount of control volumes.

This type of algebraic systems can be solved using numerical methods such as LU (Lower Upper Factorization) or iterative ones like TDMA (Tri Diagonal Matrix Algorightm). However, one of the main features of the code implementation is the paralelisation of the solver. This means that these two numerical solving methods can't be used. Therefore, a simple iterative Gauss-Seidel method has been selected.

This kind of solvers sweeps through all nodes of the domain computing the Stream Function at each of them.

$$\psi_P = \frac{a_W\psi_W + a_E\psi_E + a_S\psi_S + a_N\psi_N}{a_P} \tag{22}$$

It repeats the sweep through all nodes until the convergence condition (δ_{GS}) is meet for the Stream Function variable.

$$|\psi_{Computed} - \psi_{Supposed}| < \delta_{GS}? \tag{23}$$

In this case, this convergence condition has been set to $1 \cdot 10^{-6}$. It is a value that ensures a great accuracy on the results without being a complete wasteful of computational resources.

Then, once the Stream Function field has been calculates, it is necessary to compute all the other fluid parameters presented before, such as velocities, temperature, pressure and density.

Again, it is necessary to verify the convergence condition. However, in this case, this condition must be met for every of the physical variables calculated before. Its value has been set again to $1 \cdot 10^{-6}$ to ensure a good accuracy.

In case of success, the simulation has been completed and it is possible to compute final parameters such as aerodynamic coefficients, circulation, etc... If it doesn't met the condition, it is necessary to repeat the whole process from Gauss-Seidel iterative solver.

3 Code structure

In this section is presented the main features of the computational codes developed for the simulation of Potential Flow.

C++ has been selected as programming language for the coding. It is common and widely used language for numerical methods in engineering. In addition, object-oriented programming has been implemented too. Each important aspect of the code has its own C++ Class, from the data reading to the final solver. All these codes are presented in annex A.

However, clearly, the most important feature of the codes is the implementation of parallel computation. In CFD simulations, computers have to carry out millions or even billions of calculations. In case of sequential computing, these calculation have to be done by just 1 CPU core. Instead of that, parallel programming splits these into the number of CPU cores assigned to the simulation.

In this case, MPI (Message Passing Interface) parallelisation has been implemented (see Parallel Programming C++ Class in Annex A).

The basis of parallelisation consists on splitting the domain into several parts (figure 7).

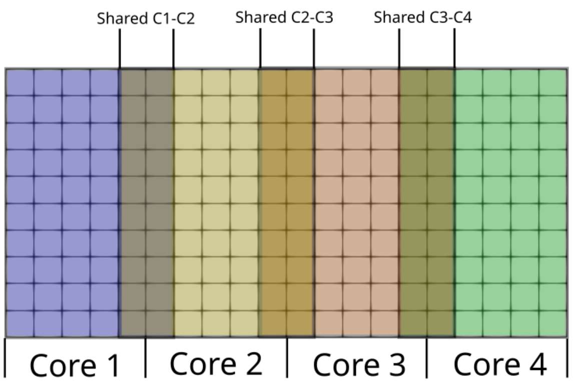

Figure 7: Domain split into cores scheme

As there can be seen in previous figure, each core is assigned a certain part of the spatial domain. The enormous reduction in computational time comes due to the fact that each core only has to solve the physical variables of its region.

To be able to do that, it is also necessary to maintain a continuous communication between all cores (or processes). As it has been shown in equation 22, in order to compute the value of the Stream Function in a node it is necessary to know the values of the surrounding nodes. But, what happens right at the interface between two cores. Here, it is necessary to define what's called as *Halo*. It consists on a shared region in the domain to which each core sends the information of the closest nodes to the core next to it.

Later, in section 5 is presented a brief study of the influence of parallel programming for this simulation.

Finally in this section, it is presented a summary of the codes structure used in the simulations.

1. Reading input data from text files.
2. Declaration and execution of parallel programming functions.
3. Mesh generation and geometrical entities calculation.
4. Calculation and setting of initial variables.
5. Computation until overall convergence.
   (a) Discretization coefficients calculation.
   (b) Stream Function field resolution.
   (c) Velocities, temperature, pressure and density calculation.
   (d) Checking of convergence criteria.
6. Calculation of final parameters.

7. Post - processing

4 Code Verification

Before presenting the numerical studies carried out and the results obtained in the simulations, it is crucial to verify the computational codes.

In order to do that, two different verifications have been made.

First of them concerns to the Flow inside a channel case. Luckily, there is analytical solution for this simulation. The Stream Function scalar field in this case consists on a linear increasing of it with the Y coordinate of the domain. At its lowest point, its value $\psi = 0$, and at its highest, $\psi = U_{ref} \cdot H$.

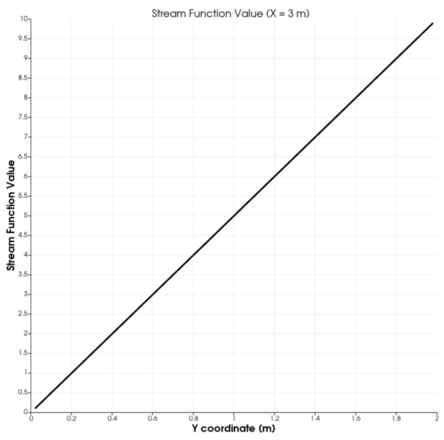

Figure 8: Stream Function value on the centre vertical line

The plot from figure 8 has been done using Paraview, extracting it right from the Stream Function field. And, as there can be seen, it presents a clear linear tendency of the parameter between 0 and its maximum value, 10 ($U_{ref} = 5m/s$, $H = 2.0m$).

In addition to that, the rest of the physical parameters such as velocities, temperature, pressure and density remain constant through all the domain with its reference value. This matches again with the analytical solution of the case.

The other verification done concerns the solid cylinder. The addition of certain numerical techniques such as the Blocking Off and the boundary conditions like the zero density or harmonic mean to simulate the presence of solid are quite important and not so easy to program, it is crucial to verify them.

To do that, there has been selected to compare the results obtained for the drag and lift force on the cylinder for different ψ values on the solid. In case of the lift, this are computed by two different methods. First, by looking at the resultant pressure field and computing the force contributions. And the other one, by calculating the circulation over the cylinder (adding the circulation of each control volume that pertains to it) and applying the Kutta–Joukowski theorem for the lift.

$$L = \Gamma \cdot U_{ref} \cdot \rho_{ref} \tag{24}$$

The results obtained are presented in table 1.

Table 1: Lift comparison results obtained

$\psi\|_{Solid}$	**Drag (P. Field) [N]**	**Lift (P. Field) [N]**	**Lift (Γ) [N]**	**Relative Error [%]**
$0.10 \cdot U_{ref} \cdot H$	-0.019	88.104	94.872	7.134
$0.30 \cdot U_{ref} \cdot H$	-0.005	44.066	47.438	7.106
$0.50 \cdot U_{ref} \cdot H$	0.000	-0.001	-0.000	-
$0.70 \cdot U_{ref} \cdot H$	-0.005	-44.065	-47.437	7.108
$0.90 \cdot U_{ref} \cdot H$	-0.019	-88.105	-94.875	7.136

There are two different conclusion that can be extracted from table 1. First is the Drag results obtained. Potential Flow theory basis doesn't consider any viscosity of the fluid, therefore, for a flow around a cylinder, the total drag force must be zero. And, as there can be seen in previous table, the values obtained can be considered as null from an engineering point of view.

Additionally to this, in what comes to lift calculation, there is a clear match between both computing methods. The relative errors obtained, 7 % approximately, are considered to be acceptable for CFD simulations.

Therefore, taking into account the results obtained for both verifications done, the programming codes results can be considered as more than acceptable to fulfil the accuracy scope of this report.

5 Results and Discussion

In this section are presented the numerical studies carried out for these cases, mesh influence and paralelisation performance. Additionally, the results obtained in the simulation of the rotating cylinder and NACA 0012 Airfoil are presented too.

5.1 Mesh Influence Study

Having a mesh with enough resolution is crucial for any CFD simulation. However, setting too much nodes can turn it into a complete wasteful of computational resources and time. Therefore, it is important to select the amount of nodes necessary depending on the desired accuracy of the results.

In addition to that, there are some numerical techniques that reduces the total number of nodes maintaining the results accuracy. One of these is the implementation of non-regular nodal distributions to the mesh.

As it has been shown in section 2.1, the code has been implemented with both nodal and hyperbolic tangent nodal distributions. And, although latter is harder to program, here in this section is going to be shown the superiority it has above regular one.

In order to do that, it has been chosen to use the relative error between lift calculation as accuracy parameter. This error has been computed using the values obtained from both methods explained in previous section, calculating the force due to pressure field, and calculating the circulation on the cylinder.

The results obtained are presented in figure 9.

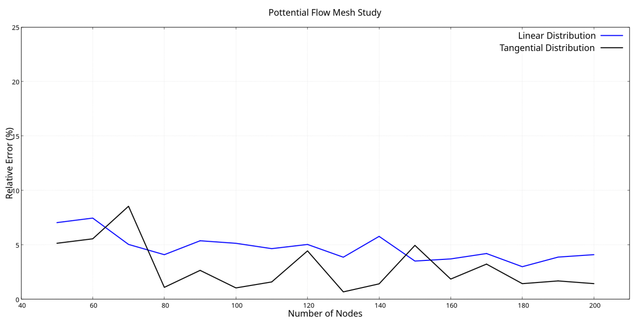

Figure 9: Mesh Influence Study results comparison

In figure 9 are presented the results of multiple simulations with different $N \times N$ nodes. Blue plot corresponds to a regular distribution and black to a hyperbolic tangent one (with a stretch factor of 1.2). The study meshes range from 50×50 to 200×200 total nodes.

As it has been mentioned, the performance parameter is the relative error of both lift calculation methods presented in previous section.

And, as there can be seen, the higher the number of nodes, the lower the relative error, this was obvious. However, another important feature that can be mentioned is the advantage of a hyperbolic tangential nodal distribution in front of a regular one.

Taking into account all the previous aspects mentioned, there was selected to carry out all the simulations of flow inside a channel and flow around a cylinder with a 200 × 200 mesh with a hyperbolic tangent distribution with stretch factor of 1.2.

For the NACA 0012 Arifoil simulation, the mesh has been modified slightly. These changes will be explained when presenting its results in section 5.4.

5.2 Paralelisation Performance

As it has been mentioned before, in section 3, one of, if not, the main feature and advantage of the codes developed is the paralelisation. Here is presented a brief study of this implementation performance.

It is true that a parallel CFD implementation takes much more time than sequential one, and so, much more programming knowledge is required. However, beyond a certain computational power needs, it becomes compulsory or extremely useful.

In this case, in order to quantify the reduction in computational time required for the simulations, there has been chosen to use the *Chrono* library from C++. This allows to measure the time between two steps of a code with amazing accuracy.

For the study, there has been selected to simulate the flow around a cylinder with previously set 200 × 200 mesh. The results obtained are presented in table 2

Table 2: Paralelisation performance results obtained

Number of cores	**Computational Time [s]**
2	80.34
3	56.81
4	43.36

In this case, it hasn't been possible to carry out a simulation with just 1 core. The codes are developed to work in parallel. And, in addition to that, it has only been possible to perform this study up to 4 cores, since all the simulations have been done in a personal computer, not a cluster.

In what comes to the results obtained, it was obvious that, the higher the number of cores, the lower the computational time required. However, this relationship isn't linear. Doubling the number of cores doesn't cut to a half the total time. There are many communication tasks and sequential analysis that each core has to perform or wait for the other ones that slows the whole process.

Nevertheless, for this tiny amount of cores, the computational time almost cut to half when doubling them. This doesn't happen with much higher number. In most cases, the more cores the simulation has, the lower the single-core performance due to the reasons mentioned before.

However, it is clear that the extra amount of time required in the programming tasks is worth. In this case, the reduction in time is huge, and so it would be for instance for 8 or 16 cores.

Obviously, all the simulations presented in the report have been carried out using as much cores as possible, 4.

Once all the simulation parameters and studies have been presented, it is possible to show the obtained results for the proposed cases.

For the case of the flow inside a channel, it has been considered that the result presented in the verification (section 4) is enough for it. This case doesn't allow to a much further analysis due to its simplicity.

5.3 Rotating Cylinder Results

First, before showing the obtained results, it is important to mention the simulation parameters (table 3).

Table 3: Rotating Cylinder Simulation Parameters

Simulation parameter	**Value**
Cylinder Diameter (D)	0.5 m
Channel Length (L)	6.0 m
Channel Height (H)	2.0 m
U_{Ref}	5 m/s
P_{Ref}	101325 Pa
T_{Ref}	293.15 K
ρ_{Ref}	1.225 Kg/m^3
Nodes	200 × 200
Nodal Distribution	Hyperbolic Tangent

In order to show the results obtained in a simple and clear way, it has been decided to present the lift numerical results from several simulations, as well as the streamlines for both a clock and counter wise cylinder rotations.

Next up is presented in figure 10, the cylinder lift dependence on the Stream Function value set on the solid.

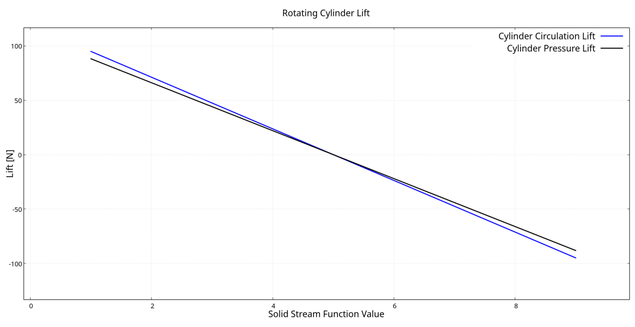

Figure 10: Cylinder Lift dependence on the Stream Function value

As it was shown before, in section 4, both lift calculation methods match in a great way. However, in this case, it is important to remark the linear dependence on the lift with the Stream Function Value selected. Taking into account the Kutta–Joukowski theorem, it means that the circulation value on the cylinder grows or decreases linearly with the Stream Function value [2].

Next up, in figure 11 is presented the velocity field and streamlines of a rotating cylinder in counter clockwise direction. In this case, the Stream Function value has been set to $\psi = 0.90 \cdot U_{ref} \cdot H$.

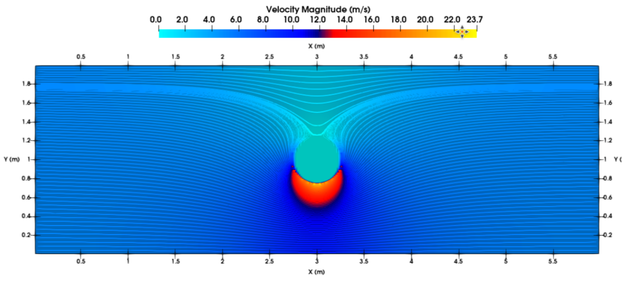

Figure 11: Counter Clockwise rotating cylinder

As there can be seen in previous figure, the rotation of the cylinder makes the flow of the upper part to decrease its velocity. This displaces the stagnation point (zero velocity point) from the left part of the cylinder to the upper one. And, again, due to the rotation, the flow accelerates on the lower part of the cylinder.

This decrease and increase in the velocities lead to a significant difference in the pressure. As Bernoulli's Principle states, the higher the velocity, the lower the pressure and vice versa. Therefore, in this case there is a much higher pressure on the upper part of the cylinder, which explains why the calculated lift is negative ($L = -94.875\ N$).

Next up, in figure 12, is presented the velocity field and streamlines of a rotating cylinder in clockwise direction. For this case, the Stream Function value has been set to $\psi = 0.10 \cdot U_{ref} \cdot H$.

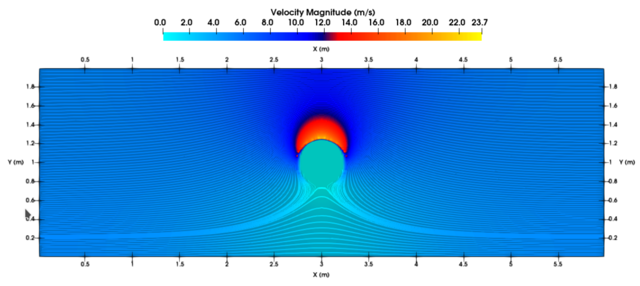

Figure 12: Clockwise rotating cylinder

In this case, on the other hand, the rotation of the cylinder leads to a decrease of the velocity of the flow on the lower part of the cylinder and an acceleration on the upper one. Following again Bernoulli's Principle explains why the cylinder lift is positive in this case. Low velocity on the lower part generates higher pressure and vice versa on the upper region, leading to a net lift force of 94.872 N.

5.4 NACA 0012 Airfoil Results

The simulation of a NACA 0012 Airfoil is clearly the most difficult study case presented in this report. And, obviously, there must be taken into account that Potential Flow is far from being as accurate as CFD aerodynamic simulations. Additionally, there must be taken into account that Blocking Off method is used with orthogonal meshes.

Nevertheless, the simulation of an Airfoil is a great way to complete a brief approach to the numerical simulation of Potential Flows.

NACA 0012 is a symmetric airfoil developed by NACA (National Advisory Committee for Aeronautics) from 4-series Airfoils. It isn't used nowadays, but it is a great and easy option for the simulation. In figure 13 is presented a scheme of it.

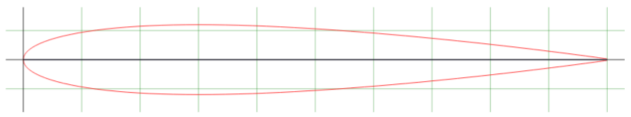

Figure 13: NACA 0012 Airfoil scheme. Extracted from [3]

The mathematical expression to define its geometry, and so, being able to perform a Blocking Off method can be found in [1]. In case of setting a non-null angle of attack, the geometry points have been rotated from the leading edge of the airfoil.

In order to have an acceptable accuracy, there has been decided to do some modification to the spatial domain and mesh generation. First of them consists on reducing the spatial domain to a 4m × 2m channel, in order to reduce the size of the nodes while having a fully developed flow before the leading edge of the airfoil. And, secondly, the mesh has been increased in total number of nodes up to 250 × 250, and so the stretch factor of the hyperbolic tangent distribution has been set to 1.5.

Next up, in table 4, are presented the lift coefficient (C_L) results for several angles of attack and the comparison with the ones extracted from [1].

Table 4: Lift coefficient results

Angle of attack [º]	Obtained results	NACA results	Relative Error [%]
7.5	0.340	0.750	54.667
5.0	0.430	0.550	21.81
2.5	0.259	0.250	3.60
0.0	0.000	0.000	0.00
-2.5	-0.259	-0.30	13.667

As there can be observed in previous table, the results obtained show a clear mismatch between obtained and NACA results when increasing the angle of attack. For low α, the viscosity and other aerodynamic phenomena doesn't play a big role in the lift force. However, it is clear that when α grows, the results obtained with Potential Flow theory are completely useless.

Additionally, it is presented, in figure 14, the velocity and streamlines field for the airfoil with an angle of attack of 5.0º.

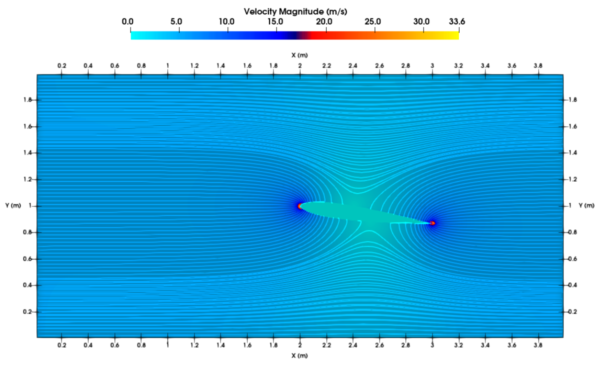

Figure 14: NACA 0012, $\alpha = 5.0º$ Velocity and Streamlines field

As there can be seen in figure 14, the velocity fields are not exactly the same on the upper and lower part of the airfoil due to its angle of attack. Therefore, lift force is not null for this α value.

Taking into account the results presented for this case, there can be clearly stated that Potential Flow can be only applied for the simulation of low α airfoils. There are many physical phenomena involved in the aerodynamic performance of an airfoil. And Potential Flow doesn't seem able to taking them into account. The only advantage this theory has over traditional Navier-Stokes in airfoils, is the lower computational costs. But, it restricts a lot the α range available for simulation.

6 Conclusions

In what comes to the results obtained, there are several conclusions that can be extracted.

First of them is the relatively good accuracy of them. As it has been shown, the theoretical and numerical results match in a great way, such as the first presented case of flow inside a channel. And so other like the numerical calculation of cylinder's lift under different Stream Function values.

Another important aspect is the implementation of parallel programming techniques to the codes, which has reduced enormously the computational times required.

And finally, there must be mentioned the simulation of the NACA 0012 Airfoil, which, although there are techniques that result in a much better accuracy, Potential Flow gives an insight of the order of the results while being much cheaper in computational resources.

Taking into account all aforementioned, there can be concluded that, the scopes for this report have been met, and that the resolution of Potential Flows may be considered as something relatively far away from a Navier Stokes CFD simulation. However, it has multiple common simulation basis that pave the path to more advanced studies.

References

[1] *2D NACA 0012 Airfoil Validation*. URL: https://turbmodels.larc.nasa.gov/naca0012_val.html.

[2] *Lift of a Rotating Cylinder*. URL: https://www.grc.nasa.gov/WWW/K-12/airplane/cyl.html.

[3] *NACA 0012 AIRFOILS (n0012-il)*. URL: http://airfoiltools.com/airfoil/details?airfoil=n0012-il.

A Programming Codes used

Main code

```
#include <fstream>
#include <cstdlib>
#include <iostream>
#include <cmath>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <mpi.h>

#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod
#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod
#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod
#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod
#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod

using namespace std;

#define DIRECTORIO "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/"

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

MPI_Init(&argc, &argv);
cout << endl;
int N = 11;
int i = 0;

Memory M1;

ReadData R1(M1, N);
R1.ReadInputs();

    ParPro MPI1(R1, i);
    MPI1.Execute();

    Mesher MESH(M1, R1, MPI1, i);
    MESH.ExecuteMesher(M1, MPI1);

    printf("Proceso %d, Initial X: %d, Final X: %d \n", MESH.Rank, MESH.Ix, MESH.Fx);

    Solver S1(M1, R1, MPI1, MESH, i);
    S1.ExecuteSolver(M1, R1, MPI1, MESH);

MPI_Finalize();
return 0;

}
```

Makefile code

```
#-----------------------------------------------------------------------------
# compile the UMFPACK demos (for GNU make and original make)
#-----------------------------------------------------------------------------
```

```
# UMFPACK Version 4.4, Copyright (c) 2005 by Timothy A. Davis.
# All Rights Reserved.  See ../Doc/License for License.

CODE = Main
DIRECTORIO = /home/sergiogus/Desktop/ComputationalEngineering/
 ConvectionDiffusion/Codes/
default: $(CODE)


C = mpic++

#------------------------------------------------------------------------------
# Create the demo programs, run them, and compare the output
#------------------------------------------------------------------------------

$(CODE): $(CODE).cpp $(INC)
         $(C) -o $(CODE) $(CODE).cpp $(DIRECTORIO)/CppCodes/Memory.cpp
         $(DIRECTORIO)/CppCodes/ReadData.cpp $(DIRECTORIO)/CppCodes/ParPro.cpp
         $(DIRECTORIO)/CppCodes/Mesher.cpp $(DIRECTORIO)/CppCodes/Solver.cpp
#------------------------------------------------------------------------------
# Remove all but the files in the original distribution
#------------------------------------------------------------------------------

clean:
         - $(RM) $(CLEAN) $(CODE)
```

Memory Class Header Code

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <cmath>
#include <mpi.h>

using namespace std;

#define PI 3.141592653589793

class Memory{

        public:
                //Constructor de la clase
                Memory();

                //Metodos de la clase

                //M todos de alojamiento de memoria
                int *AllocateInt(int, int, int);
                double *AllocateDouble(int, int, int);

};
```

Memory Class Cpp Code

```
#include <iostream>
```

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <cmath>
#include <mpi.h>

#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod
using namespace std;

#define DIRECTORIO "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/"

Memory::Memory(){

}

//Memoria din mica matriz (double)
double *Memory::AllocateDouble(int NX, int NY, int Dim){
double *M1;

        M1 = new double [NX*NY*Dim];
        return M1;
}

//Memoria din mica matriz (int)
int *Memory::AllocateInt(int NX, int NY, int Dim){
int *M1;

        M1 = new int [NX*NY*Dim];
        return M1;
}
```

Read Data Class Header Code

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <cmath>
#include <mpi.h>

using namespace std;

class ReadData{
        private:


        public:

                string Problema; //Problema Tobera/Tuberia

                double *GeometryData; //Datos de la geometr a del problema
                int *NumericalData; //Datos num ricos del problema
                double *ProblemData; //Datos del problema
                double *ProblemPhysicalData; //Datos f sicos sobre las condiciones del pr

                int *MeshStudyNX;
                int *MeshStudyNY;
                double *FactorRelax;

                double *AttackAngle;

                //Constructor de la clase
                ReadData(Memory, int);

                void ReadInputs(); //Lector datos en ficheros
                void ReadArrays(string, int , double*);
                void ReadMeshStudy(string, int);

};
```

Read Data Class Cpp Code

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <cmath>
#include <fstream>
#include <sstream>
#include <bits/stdc++.h>
#include <string>
#include <mpi.h>

#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod
#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod

using namespace std;

#define DIRECTORIO "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/InputI

//Constructor del lector de datos
ReadData::ReadData(Memory M1, int N){

        GeometryData = M1.AllocateDouble(3, 1, 1); //Datos de la geometr a del problema
        NumericalData = M1.AllocateInt(4, 1, 1); //Datos num ricos del problema
        ProblemData = M1.AllocateDouble(2, 1, 1); //Datos del problema
        ProblemPhysicalData = M1.AllocateDouble(8, 1, 1); //Datos f sicos sobre las condi

        MeshStudyNX = M1.AllocateInt(N, 1, 1);
        MeshStudyNY = M1.AllocateInt(N, 1, 1);

        AttackAngle = M1.AllocateDouble(14, 1, 1);
}

void ReadData::ReadArrays(string FileName, int TotalData, double *Array){
int i = 0;
stringstream InitialName;
string FinalName;

        InitialName<<DIRECTORIO<<FileName;
        FinalName=InitialName.str();
        ifstream Data(FinalName.c_str());

                if (Data){
                        string line;
                        while (getline(Data, line)){
                                istringstream iss(line);
                                        if(i < TotalData){
                                                if (iss >> Array[i]){ i++; }
                                        }
                         }
                }

        Data.close();

}

void ReadData::ReadInputs(){

        //Lectura datos en Arrays
        ReadArrays("GeometryData.txt", 3, GeometryData); //Input Datos Geometr a del prob
        ReadArrays("PhysicalData.txt", 8, ProblemPhysicalData);   //Input Datos f sicos d
        ReadArrays("AttackAngleData.txt", 14, AttackAngle);

int i = 0;
string FileName;
stringstream InitialName1;
string FinalName1;

        FileName = "ProblemData.txt";

        InitialName1<<DIRECTORIO<<FileName;
        FinalName1=InitialName1.str();

        ifstream DatosProblema(FinalName1.c_str());

                if (DatosProblema){
                        string line;
                        while (getline(DatosProblema, line)){
                                istringstream iss(line);
                                        if(i < 4){
                                                if (iss >> NumericalData[i]){ i++; }
                                        }
                                        else if(i >= 4 && i < 6){
                                                if (iss >> ProblemData[i-4]){ i++; }
                                        }
                                        else{
                                                if (iss >> Problema){ i++; }
                                        }

                         }
                }

        DatosProblema.close();

}
```

```
void ReadData::ReadMeshStudy(string FileName, int TotalData){
int a, b;

int i = 0;
stringstream InitialName;
string FinalName;

        InitialName<<DIRECTORIO<<FileName;
        FinalName=InitialName.str();

        ifstream Data(FinalName.c_str());

                if (Data){
                        string line;
                        while (getline(Data, line)){
                                istringstream iss(line);
                                                if (iss >> MeshStudyNX[i] >> MeshStudyNY[i

                         }
                }

        Data.close();

}
```

Parallel Programming Class Header Code

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <cmath>
#include <mpi.h>

using namespace std;

class ParPro{
        private:


        public:

                //Datos generales del problema
                int NX;
                int NY;

                //Datos y variables de computaci n paralela
                int Rank;
                int Procesos;
                int Ix;
                int Fx;
                int Halo;
                int pix, pfx;

                //Constructor de la clase
                ParPro(ReadData, int);
```

```
                    //M todos de la clase
                    void Rango();
                    void Processes();
                    void Initial_WorkSplit(int, int&, int&);
                    void Get_Worksplit(int, int, int, int&, int&);

                    void SendData(double*, int, int);
                    void ReceiveData(double*, int, int);

                    void SendDataInt(int*, int, int);
                    void ReceiveDataInt(int*, int, int);

                    void SendMatrixToZero(double*, double*, int, int, int, int, int);
                    void SendDataToZero(double, double*);
                    void SendDataToAll(double, double&);

                    void Execute();

                    int Get_Rank(){ return Rank; }
                    int Get_Processes(){ return Procesos; }
                    int Get_Halo(){ return Halo; }
                    int Get_Ix(){ return Ix; }
                    int Get_Fx(){ return Fx; }

};
```

Parallel Porgramming Class Cpp Code

```
#include <iostream>
#include <sstream>
#include <fstream>
#include <math.h>
#include <stdlib.h>
#include <cmath>
#include <stdio.h>
#include <cassert>
#include <time.h>
#include <mpi.h>

#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod
#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod
#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod

using namespace std;

#define DIRECTORIO "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/"

#define G(i,j,dim) (((j) + (i)*NY) + NX*NY*(dim)) //Global Index
#define GU(i,j,dim) (((j) + (i)*NY) + (NX+1)*NY*(dim)) //Global Index Matriz U
#define GR(i,j,dim) (((j) + (i)*(NY+1)) + NX*(NY+1)*(dim)) //Global Index Matriz R

#define MU(i,j,dim) ((j) + NY*(dim))
#define MR(i,j,dim) ((i) + (Fx - Ix)*(dim) + (Fx - Ix)*(4)*(j))

#define VR(i,j,dim) ((i) + (Fx - Ix)*(j))
```

```
#define LNH(i,j,dim) (((j) + ((i) - Ix + Halo)*NY)) //Local Index No Halo
#define LSH(i,j,dim) (((j) + ((i) - Ix)*NY) + NY*(Fx-Ix + 2*Halo)*dim) //Local Index Si Ha


ParPro::ParPro(ReadData R1, int i){

                NX = R1.NumericalData[0];
                NY = R1.NumericalData[1];

                Halo = 1;

}

void ParPro::Rango(){
        int a;
        a = MPI_Comm_rank(MPI_COMM_WORLD, &Rank);
}

void ParPro::Processes(){
        int a;
        a = MPI_Comm_size(MPI_COMM_WORLD, &Procesos);
}

void ParPro::Initial_WorkSplit(int NX, int &Ix, int &Fx){
int Intervalo, Residuo;
int p;
        Intervalo = NX/Procesos;
        Residuo = NX%Procesos;

        if(Rank != Procesos-1){
                Ix = Rank*Intervalo;
                Fx = (Rank+1)*Intervalo;
        }
        else{
                Ix = Rank*Intervalo;
                Fx = (Rank+1)*Intervalo + Residuo;
        }
}

void ParPro::Get_Worksplit(int NX, int Procesos, int p, int &pix, int &pfx){
int Intervalo, Residuo;

        Intervalo = NX/Procesos;
        Residuo = NX%Procesos;

        if(p != Procesos-1){
                pix = p*Intervalo;
                pfx = (p+1)*Intervalo;
        }
        else{
                pix = p*Intervalo;
                pfx = (p+1)*Intervalo + Residuo;
        }
}
```

```
void ParPro::SendData(double *LocalSend, int Ix, int Fx){

        if(Rank != 0 && Rank != Procesos-1){
                MPI_Send(&LocalSend[LNH(Ix,0,0)], Halo*NY, MPI_DOUBLE, Rank-1, 0, MPI_COMM_
                MPI_Send(&LocalSend[LNH(Fx - Halo,0,0)], Halo*NY, MPI_DOUBLE, Rank+1, 0, M
        }
        else if(Rank == 0){
                MPI_Send(&LocalSend[LNH(Fx - Halo,0,0)], Halo*NY, MPI_DOUBLE, 1, 0, MPI_COM
        }
        else{
                MPI_Send(&LocalSend[LNH(Ix,0,0)], Halo*NY, MPI_DOUBLE, Procesos-2, 0, MPI_C
        }

}

void ParPro::ReceiveData(double *LocalReceive, int Ix, int Fx){
int p;
MPI_Status ST;

        if(Rank != 0 && Rank != Procesos-1){
                MPI_Recv(&LocalReceive[LNH(Ix - Halo,0,0)], Halo*NY, MPI_DOUBLE, Rank-1, 0
                MPI_Recv(&LocalReceive[LNH(Fx,0,0)], Halo*NY, MPI_DOUBLE, Rank+1, 0, MPI_CO
        }
        else if(Rank == 0){
                MPI_Recv(&LocalReceive[LNH(Fx,0,0)], Halo*NY, MPI_DOUBLE, 1, 0, MPI_COMM_W
        }
        else if(Rank == Procesos-1){
                MPI_Recv(&LocalReceive[LNH(Ix - Halo,0,0)], Halo*NY, MPI_DOUBLE, Procesos-
        }


}

void ParPro::SendDataInt(int *LocalSend, int Ix, int Fx){

        if(Rank != 0 && Rank != Procesos-1){
                MPI_Send(&LocalSend[LNH(Ix,0,0)], Halo*NY, MPI_INT, Rank-1, 0, MPI_COMM_WO
                MPI_Send(&LocalSend[LNH(Fx - Halo,0,0)], Halo*NY, MPI_INT, Rank+1, 0, MPI_C
        }
        else if(Rank == 0){
                MPI_Send(&LocalSend[LNH(Fx - Halo,0,0)], Halo*NY, MPI_INT, 1, 0, MPI_COMM_W
        }
        else{
                MPI_Send(&LocalSend[LNH(Ix,0,0)], Halo*NY, MPI_INT, Procesos-2, 0, MPI_COMM
        }

}

void ParPro::ReceiveDataInt(int *LocalReceive, int Ix, int Fx){
int p;
MPI_Status ST;

        if(Rank != 0 && Rank != Procesos-1){
                MPI_Recv(&LocalReceive[LNH(Ix - Halo,0,0)], Halo*NY, MPI_INT, Rank-1, 0, M
                MPI_Recv(&LocalReceive[LNH(Fx,0,0)], Halo*NY, MPI_INT, Rank+1, 0, MPI_COMM_
        }
        else if(Rank == 0){
                MPI_Recv(&LocalReceive[LNH(Fx,0,0)], Halo*NY, MPI_INT, 1, 0, MPI_COMM_WORL
        }
        else if(Rank == Procesos-1){
                MPI_Recv(&LocalReceive[LNH(Ix - Halo,0,0)], Halo*NY, MPI_INT, Procesos-2,
        }


}

void ParPro::SendMatrixToZero(double *LocalMatrix, double *GlobalMatrix, int NX, int NY, i
int i, j, p;
int pix, pfx;
MPI_Status ST;

        if(Rank != 0){
                MPI_Send(&LocalMatrix[LNH(Ix,0,0)], NY*(Fx-Ix), MPI_DOUBLE, 0, 0, MPI_COMM_
        }

        if(Rank == 0){
                for(i = Ix; i < Fx; i++){
                        for(j = 0; j < NY; j++){
                                GlobalMatrix[G(i,j,0)] = LocalMatrix[LNH(i,j,0)];
                        }
                }

                for(p = 1; p < Procesos; p++){
                        Get_Worksplit(NX, Procesos, p, pix, pfx);
                        MPI_Recv(&GlobalMatrix[G(pix,0,0)], NY*(pfx - pix), MPI_DOUBLE, p,
                }

        }

}

void ParPro::SendDataToZero(double DataSent, double *DataReceived){
int i, p;
MPI_Status ST;

        if(Rank != 0){
                MPI_Send(&DataSent, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        }

        if(Rank == 0){
                DataReceived[Rank] = DataSent;
                for(p = 1; p < Procesos; p++){
                        MPI_Recv(&DataReceived[p], 1, MPI_DOUBLE, p, 0, MPI_COMM_WORLD, &S
                }
        }
}

void ParPro::SendDataToAll(double DataSent, double &DataReceived){
int p;
MPI_Status ST;

        if(Rank == 0){
                for(p = 1; p < Procesos; p++){
                        MPI_Send(&DataSent, 1, MPI_DOUBLE, p, 0, MPI_COMM_WORLD);
                }
                DataReceived = DataSent;
        }

        if(Rank != 0){
                MPI_Recv(&DataReceived, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &ST);
        }

}

void ParPro::Execute(){

        Rango();
        Processes();

}
```

Mesher Class Header Code

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <cmath>
#include <stdio.h>
#include <time.h>
#include <mpi.h>

using namespace std;

class Mesher{
        private:
                //Datos sobre el problema
                        string Problema; //Problema (Canal/Canal con Cilindro/Perfil)

                //Datos sobre la geometr a del problema
                        double ChannelLength; //Longitud del canal
                        double ChannelHeight; //Altura del canal
                        double ChannelDepth; //Profundidad del canal

                //Datos del mallado
                        int OpcionR; //Tipo de discretizaci n en la direcci n radial
                        int OpcionA; //Tipo de discretizaci n en la direcci n axial
                        double StretFactorX; //Factor de estrechamiento de la discretizac
                        double StretFactorY; //Factor de estrechamiento de la discretizac

        public:
                //Constructor de la clase
                Mesher(Memory, ReadData, ParPro, int);

                //Datos para la computaci n en paralelo
                int Procesos;
                int Rank;
                int Ix;
                int Fx;
```

```
            //Densidad del mallado
            int NX; //Direcci n axial total
            int NY; //Direcci n radial

            //Matrices de coordenadas de discretizaci n
            double *AxialCoord; //Matriz coordenada axial de la distribuci n

            //Caso mallado tipo Staggered
            double *MP; //Coordenadas matriz de presi n/temperatura
            double *MU; //Coordenadas matriz velocidades axiales
            double *MR; //Coordenadas matriz velocidades radiales

            //Matrices de distancias de vol menes de control
            double *DeltasMP; //Deltas X y R de la matriz de Presi n/Temperatura
            double *DeltasMU; //Deltas X y R de la matriz de velocidades axiales (U)
            double *DeltasMR; //Deltas X y R de la matriz de velocidades radiales (V)

            double *SupMP;

            //M todos de la clase del mallador
            void AllocateMemory(Memory); //Alojamiento de memoria para cada matriz

            void Get_AxialCoordinates(); //C lculo del la posici n axial de los nodo
            void Get_Mesh(); //Creaci n de todas las mallas
            void Get_Deltas(); //C lculo de las distancias entre nodos en cada una de
            void Get_Surfaces();
            void PrintTxt();
            void MallaVTK2D(string, string, string, double*, int, int);

            void ExecuteMesher(Memory, ParPro); //Ejecutar todos los procesos del mall
};
```

Mesher Class Cpp Code

```
#include <iostream>
#include <sstream>
#include <fstream>
#include <math.h>
#include <stdlib.h>
#include <cmath>
#include <stdio.h>
#include <cassert>
#include <time.h>
#include <bits/stdc++.h>
#include <string>
#include <time.h>
#include <mpi.h>

using namespace std;

#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod
#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod
#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod
#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod
```

```
#define DIRECTORIO "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/"

#define PI 3.141592653589793

#define sind(x) sin(x * (PI/180.0)) //C lculo seno en grados
#define cosd(x) cos(x * (PI/180.0)) //C lculo coseno en grados
#define tand(x) tan(x * (PI/180.0)) //C lculo tangente en grados

#define Hyp(x1, x2, y1, y2) sqrt(pow(x2-x1,2.0) + pow(y2-y1,2.0)) //C lculo hipotenusa

#define G(i,j,dim) (((j) + (i)*NY) + NX*NY*(dim)) //Global Index
#define GU(i,j,dim) (((j) + (i)*NY) + (NX+1)*NY*(dim)) //Global Index Matriz U
#define GR(i,j,dim) (((j) + (i)*(NY+1)) + NX*(NY+1)*(dim)) //Global Index Matriz R

#define MU(i,j,dim) ((j) + NY*(dim))
#define MR(i,j,dim) ((i) + (Fx - Ix)*(dim) + (Fx - Ix)*(4)*(j))

#define VR(i,j,dim) ((i) + (Fx - Ix)*(j))

#define LNH(i,j,dim) (((j) + ((i) - Ix + Halo)*NY)) //Local Index No Halo
#define LSH(i,j,dim) (((j) + ((i) - Ix)*NY) + NY*(Fx-Ix + 2*Halo)*dim) //Local Index Si Ha

//Constructor del mallador
Mesher::Mesher(Memory M1, ReadData R1, ParPro MPI1, int i){

        //Datos sobre el problema
                Problema = R1.Problema; //Problema (Canal/Canal con Cilindro/Perfil)

        //Datos para la computaci n en paralelo
                Procesos = MPI1.Procesos;
                Rank = MPI1.Rank;

        //Datos sobre la geometr a del problema

                //Geometr a del caNAl
                ChannelLength = R1.GeometryData[0]; //Longitud del canal
                ChannelHeight = R1.GeometryData[1]; //Altura del canal
                ChannelDepth = R1.GeometryData[2]; //Profundidad del canal

                //Densidad del mallado
                NX = R1.NumericalData[0];
                NY = R1.NumericalData[1];
                // //N mero de nodos direcci n X
                //NY = //N mero de nodos direcci n Y

                //Opciones del mallado
                OpcionR = R1.NumericalData[2]; //Tipo de discretizaci n en la direcci n
                OpcionA = R1.NumericalData[3]; //Tipo de discretizaci n en la direcci n
                StretFactorX = R1.ProblemData[0]; //Factor de estrechamiento de la discret
                StretFactorY = R1.ProblemData[1]; //Factor de estrechamiento de la discret

}

//C lculo del la posici n axial de los nodos de velocidad axial
void Mesher::Get_AxialCoordinates(){
int i;
```

```
double I;
double nx1 = NX/2;
int NX2 = NX - NX/2;
double nx2 = NX2;

        if(OpcionA == 1){ //Regular
                for(i = 0; i < NX + 1; i++){
                        I = i;
                        AxialCoord[i] = I*(ChannelLength/NX);
                }
        }
        else if(OpcionA == 2){ //Tangencial hiperb lica

                for(i = 0; i < NX/2 + 1; i++){
                        I = i;
                        AxialCoord[i] = (0.50*ChannelLength)*(tanh(StretFactorX*(I/nx1))/ta
                }
                for(i = 1; i < NX2 + 1; i++){
                        I = i;
                        AxialCoord[i+NX2] = (0.50*ChannelLength) + (0.50*ChannelLength)*((
                }
        }


}

//Alojamiento de memoria para cada matriz
void Mesher::AllocateMemory(Memory M1){

        AxialCoord = M1.AllocateDouble(NX + 1, 1, 1); //Matriz coordenada axial de la dist

        //Matrices de coordeNXdas de discretizaci n
        MP = M1.AllocateDouble(NX, NY, 2); //CoordeNXdas matriz de presi n/temperatura
        MU = M1.AllocateDouble(NX + 1, NY, 2); //CoordeNXdas matriz velocidades axiales
        MR = M1.AllocateDouble(NX, NY + 1, 2); //CoordeNXdas matriz velocidades radiales

        //Matrices de distancias de vol menes de control
        DeltasMP = M1.AllocateDouble(NX, NY, 2); //Deltas X y R de la matriz de Presi n/T
        DeltasMU = M1.AllocateDouble(NX + 1, NY, 2); //Deltas X y R de la matriz de velocid
        DeltasMR = M1.AllocateDouble(NX, NY + 1, 2); //Deltas X y R de la matriz de velocid

        //Matrices de superficies de los vol menes de control
        SupMP = M1.AllocateDouble(NX, NY, 4);

}

//Creaci n de todas las mallas de tipo Staggered
void Mesher::Get_Mesh(){
int i, j;
double ny1 = NY/2;
int NY2 = NY - NY/2;
double ny2 = NY2;
double J;

        //Coordenadas Axiales
        //Coordenadas axiales matriz de velocidades axiales (U)
        for(i = 0; i < NX + 1; i++){
                for(j = 0; j < NY; j++){
                        MU[GU(i,j,0)] = AxialCoord[i];
                }
        }

        //CoordeNXdas axiales matriz de velocidades radial (V)
        for(i = 0; i < NX; i++){
                for(j = 0; j < NY + 1; j++){
                        MR[GR(i,j,0)] = 0.5*(AxialCoord[i] + AxialCoord[i+1]);
                }
        }

        //CoordeNXdas axiales matriz de Presi n/Temperatura
        for(i = 0; i < NX; i++){
                for(j = 0; j < NY; j++){
                        MP[G(i,j,0)] = 0.5*(AxialCoord[i] + AxialCoord[i+1]);
                }
        }

        //CoordeNXdas Radiales

        //CoordeNXdas radiales de la matriz de velocidades radial (V)
        if(OpcionR == 1){ //Regular
                for(i = 0; i < NX; i++){
                        for(j = 0; j < NY + 1; j++){
                                J = j;
                                MR[GR(i,j,1)] = J*(ChannelHeight/NY);
                        }
                }
        }
        else if(OpcionR == 2){ //Tangencial hiperb lica
                for(i = 0; i < NX; i++){
                        for(j = 0; j < NY/2 + 1; j++){
                                J = j;
                                MR[GR(i,j,1)] = (0.50*ChannelHeight)*(tanh(StretFactorY*(J
                        }
                        for(j = 1; j < NY2 + 1; j++){
                                J = j;
                                MR[GR(i,j+NY/2,1)] = (0.50*ChannelHeight) + (0.50*ChannelH
                        }
                }
        }

        //CoordeNXdas radiales de la matriz de Presi n/Temperatura
        for(i = 0; i < NX; i++){
                for(j = 0; j < NY; j++){
                        MP[G(i,j,1)] = 0.5*(MR[GR(i,j,1)] + MR[GR(i,j + 1,1)]);
                }
        }

        //Coordenadas radiales de la matriz de velocidades axial (U)
        for(j = 0; j < NY; j++){
                //Parte Izquierda
                MU[GU(0,j,1)] = MP[G(0,j,1)];
                //Parte Derecha
                MU[GU(NX,j,1)] = MP[G(NX - 1,j,1)];

                for(i = 1; i < NX; i++){
                        MU[GU(i,j,1)] = 0.5*(MP[G(i-1,j,1)] + MP[G(i,j,1)]);
                }
        }

}

//Pasar los resultados de las mallas a un txt
void Mesher::PrintTxt(){
int i, j;
string Carpeta = "GnuPlotResults/Meshes/";
ofstream file;
string FileName;
stringstream InitialNameMP;
string FinalNameMP;

        FileName = "MallaMP.txt";

        InitialNameMP<<DIRECTORIO<<Carpeta<<FileName;

        FinalNameMP = InitialNameMP.str();
        file.open(FinalNameMP.c_str());

                for(i = 0; i < NX; i++){
                for(j = 0; j < NY; j++){
                        file<<MP[G(i,j,0)]<<"\t"<<MP[G(i,j,1)]<<"\t"<<endl;
                        }
                file<<endl;
        }

        file.close();

stringstream InitialNameMU;
string FinalNameMU;

        FileName = "MallaMU.txt";

        InitialNameMU<<DIRECTORIO<<Carpeta<<FileName;

        FinalNameMU = InitialNameMU.str();
        file.open(FinalNameMU.c_str());

                for(i = 0; i < NX + 1; i++){
                for(j = 0; j < NY; j++){
                        file<<MU[GU(i,j,0)]<<"\t"<<MU[GU(i,j,1)]<<"\t"<<endl;
                        }
                file<<endl;
        }

        file.close();

stringstream InitialNameMR;
```

```
string FinalNameMR;

        FileName = "MallaMR.txt";

        InitialNameMR<<DIRECTORIO<<Carpeta<<FileName;

        FinalNameMR = InitialNameMR.str();
        file.open(FinalNameMR.c_str());

                for(i = 0; i < NX; i++){
                for(j = 0; j < NY + 1; j++){
                        file<<MR[GR(i,j,0)]<<"\t"<<MR[GR(i,j,1)]<<"\t"<<endl;
                        }
                file<<endl;
        }

        file.close();

}

//C lculo de las distancias entre nodos en cada uNX de las matrices
void Mesher::Get_Deltas(){
int i, j;

        //Deltas X e Y de la matriz de Presi n/Temperatura
        for(i = 0; i < NX; i++){
                for(j = 0; j < NY; j++){
                        DeltasMP[G(i,j,0)] = MU[GU(i + 1,j,0)] - MU[GU(i,j,0)]; //Deltas
                        DeltasMP[G(i,j,1)] = MR[GR(i,j + 1,1)] - MR[GR(i,j,1)]; //Deltas
                }
        }

        //Deltas X e Y de la matriz de velocidades axiales (U)
        for(j = 0; j < NY; j++){
                //Parte Izquierda
                DeltasMU[GU(0,j,0)] = MP[G(0,j,0)] - MU[GU(0,j,0)]; //Deltas X
                DeltasMU[GU(0,j,1)] = MR[GR(0,j + 1,1)] - MR[GR(0,j,1)]; //Deltas Y

                //Parte Derecha
                DeltasMU[GU(NX,j,0)] = MU[GU(NX,j,0)] - MP[G(NX - 1,j,0)]; //Deltas X
                DeltasMU[GU(NX,j,1)] = MR[GR(NX - 1,j + 1,1)] - MR[GR(NX - 1,j,1)]; //Delt

                for(i = 1; i < NX; i++){
                        DeltasMU[GU(i,j,0)] = MP[G(i,j,0)] - MP[G(i - 1,j,0)]; //Deltas X
                        DeltasMU[GU(i,j,1)] = MR[GR(i,j + 1,1)] - MR[GR(i,j,1)]; //Deltas
                }
        }

        //Deltas X e Y de la matriz de velocidades Verticales (V)
        for(i = 0; i < NX; i++){
                //Parte abajo
                DeltasMR[GR(i,0,0)] = MU[GU(i + 1,0,0)] - MU[GU(i,0,0)]; //Deltas X
                DeltasMR[GR(i,0,1)] = MP[G(i,0,1)] - MR[GR(i,0,1)]; //Deltas Y

                //Parte arriba
                DeltasMR[GR(i,NY,0)] = MU[GU(i + 1,NY - 1,0)] - MU[GU(i,NY - 1,0)]; //Delt
                DeltasMR[GR(i,NY,1)] = MR[GR(i,NY,1)] - MP[G(i,NY - 1,1)]; //Deltas Y

                for(j = 1; j < NY; j++){
                        DeltasMR[GR(i,j,0)] = MU[GU(i + 1,j,0)] - MU[GU(i,j,0)]; //Deltas
                        DeltasMR[GR(i,j,1)] = MP[G(i,j,1)] - MP[G(i,j - 1,1)]; //Deltas Y
                }
        }

}

//C lculo de las superficies de los vol menes de control
void Mesher::Get_Surfaces(){
int i, j;

        for(i = 0; i < NX; i++){
                for(j = 0; j < NY; j++){
                        SupMP[G(i,j,0)] = DeltasMP[G(i,j,0)]*ChannelDepth;
                        SupMP[G(i,j,1)] = DeltasMP[G(i,j,0)]*ChannelDepth;
                        SupMP[G(i,j,2)] = DeltasMP[G(i,j,1)]*ChannelDepth;
                        SupMP[G(i,j,3)] = DeltasMP[G(i,j,1)]*ChannelDepth;
                }
        }

}

//Pasar los resultados a un archivo VTK en 2D
void Mesher::MallaVTK2D(string Carpeta, string Variable, string NombreFile, double *MC, in
int i, j;

        ofstream file;
    stringstream InitialName;
    string FinalName;

        InitialName<<DIRECTORIO<<Carpeta<<NombreFile<<".vtk";

                FinalName = InitialName.str();
        file.open(FinalName.c_str());

        file<<"#_vtk_DataFile_Version_2.0"<<endl;
        file<<Variable<<endl;
        file<<"ASCII"<<endl;
        file<<endl;
        file<<"DATASET_STRUCTURED_GRID"<<endl;
        file<<"DIMENSIONS"<<"___"<<NX<<"___"<<NY<<"___"<<1<<endl;
        file<<endl;
        file<<"POINTS"<<"___"<<NX*NY<<"___"<<"double"<<endl;

                for(j = 0; j < NY; j++){
                        for(i = 0; i < NX; i++){
                                file<<MC[G(i,j,0)]<<"___"<<MC[G(i,j,1)]<<"___"<<0.0<<endl;
                        }
                }

        file<<endl;
                file<<"POINT_DATA"<<"___"<<NX*NY<<endl;
        file<<"SCALARS_"<<Variable<<"_double"<<endl;
        file<<"LOOKUP_TABLE"<<"␣␣␣"<<Variable<<endl;
        file<<endl;
        for(j = 0; j < NY; j++){
                        for(i = 0; i < NX; i++){
                                file<<0.0<<"␣";
                        }
                }

        file.close();
}

//Ejecutar todos los procesos del mallador
void Mesher::ExecuteMesher(Memory M1, ParPro MPI1){

        MPI1.Initial_WorkSplit(NX, Ix, Fx);
        AllocateMemory(M1); //Alojamiento de memoria para cada matriz
        Get_AxialCoordinates(); //C lculo del la posici n axial de los nodos de velocida
        Get_Mesh(); //Creaci n de todas las mallas
        Get_Deltas(); //C lculo de las distancias entre nodos en cada uNX de las matrices
        Get_Surfaces();

        if(Rank == 0){
                PrintTxt();
                MallaVTK2D("ParaviewResults/MeshResults/", "MallaBase", "MallaMP", MP, NX,
        }

        cout<<"Mesh␣created."<<endl;

}
```

Solver Class Header Code

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <cmath>
#include <mpi.h>

using namespace std;

class Solver{
        private:

                //Problema a simular
                string Problema; //(Canal/Canal con Cilindro/Perfil)

                //Densidad del mallado
                int NX; //Direcci n horizontal
                int NY; //Direcci n vertical

                //Datos para la computaci n en paralelo
                int Ix;
                int Fx;
                int Procesos;
                int Halo;
                int Rank;

                //Datos de la geometr a
                double ChannelLength; //Longitud del canal
                double ChannelHeight; //Altura del canal
                double ChannelDepth; //Profundidad del canal

                double CylinderRadius;

                //Datos F sicos del problema
                double Uref;
                double Vref;
                double Tref;
                double RhoRef;
                double Pref;
                double Cp;
                double Rideal;
                double Gamma;

                double Circulation;
                double PhiCilindro;

                //Datos sobre el solver iterativo
                double ConvergenciaGlobal;
                double ConvergenciaGS;

                double MaxDiffGS;
                double MaxDiffGlobal;

                double Cd;
                double Cl;

                double AnguloAtaque;

                double NumericalCirculation;

                double *PDiff;

        public:
                Solver(Memory, ReadData, ParPro, Mesher, int);

                //Matrices de los mapas de propiedades del problema

                //Matrices globales de propiedades
                double *PCirc;

                //Step Presente
                double *PhiGlobalPres;
                double *UglobalPres;
                double *VglobalPres;
                double *TglobalPres;
                double *RhoGlobalPres;
                double *PglobalPres;

                //Step Futuro
                double *PhiGlobalFut;
        double *UglobalFut;
        double *VglobalFut;
        double *TglobalFut;
        double *RhoGlobalFut;
        double *PglobalFut;

        //Matrices locales de propiedades

        int *DensityIndicator;
        int *GlobalDensityIndicator;

        //Streamline
        double *PhiLocalPres;
        double *PhiLocalFut;

        double *PhiLocalSup;

        //Velocidad Horizontal
        double *UlocalPres;
        double *UlocalFut;

        //Velocidad Vertical
        double *VlocalPres;
        double *VlocalFut;

        //Temperatura
        double *TlocalPres;
        double *TlocalFut;

        //Densidad
        double *RhoLocalPres;
        double *RhoLocalFut;

        //Presi n
        double *PlocalPres;
        double *PlocalFut;

        //Matrices de valores de las propiedades en las paredes de los vol menes
        double *UwallsMR;
        double *VwallsMU;

        double *UwallsMR_Global;
        double *VwallsMU_Global;

        double *RhoWallsMU;
        double *RhoWallsMR;

        //Matrices de c lculo de contribuciones a las propiedades

        double *aw;
        double *ae;
        double *as;
        double *an;
        double *ap;

        double *awGlobal;
        double *aeGlobal;
        double *asGlobal;
        double *anGlobal;
        double *apGlobal;

        //Matrices y arrays de condiciones de contorno

        //Condiciones de contorno del mapa de valor de streamlines
        double *PhiLeft; //Presi n pared izquierda
        double *PhiRight; //Presi n pared derecha

        double *PhiUp; //Presi n pared superior
        double *PhiDown; //Presi n pared inferior

        //Condiciones de contorno del mapa de densidades
        double *RhoLeft; //Presi n pared izquierda
        double *RhoRight; //Presi n pared derecha

        double *RhoUp; //Presi n pared superior
        double *RhoDown; //Presi n pared inferior

        //Condiciones de contorno del mapa de velocidades axiales (U)
        double *Uleft; //Velocidad axial pared izquierda
        double *Uright; //Velocidad axial pared derecha

        double *Uup; //Velocidad axial pared superior
        double *Udown; //Velocidad axial pared inferior

        //Condiciones de contorno del mapa de velocidades radiales (V)
        double *Vleft; //Velocidad radial pared izquierda
        double *Vright; //Velocidad radial pared derecha

        double *Vup; //Velocidad radial pared superior
        double *Vdown; //Velocidad radial pared inferior

        //Condiciones de contorno del mapa de temperaturas (T)
        double *Tleft; //Temperaturas pared izquierda
        double *Tright; //Temperaturas pared derecha

        double *Tup; //Temperaturas pared superior
        double *Tdown; //Temperaturas pared inferior

        //Condiciones de contorno del mapa de viscosidades din micas
        double *Pleft; //Presi n pared izquierda
        double *Pright; //Presi n pared derecha

        double *Pup; //Presi n pared superior
        double *Pdown; //Presi n pared inferior

        //M todos de la clase Solver
        void AllocateMatrix(Memory);

        void InitializeFields(Mesher); //Pasar todos los coeficientes termoq mico
        void UpdateBoundaryConditions(Mesher);

        void Get_DensityIndicatorCylinder(Mesher);
```

```
                void Get_DensityIndicatorAirfoil(Mesher);

                void Get_DensityWalls(Mesher, ParPro);
                void Get_Coeficients(Mesher); //C lculo de los coeficientes de discretiza

                void Get_Streamlines(ParPro, Mesher);
                void Get_MaxDifGS(ParPro);
                void Get_Velocities(Mesher, ParPro); //C lculo del mapa de velocidades fu
                void Get_Temperatures(); //C lculo del mapa de temperaturas futuro
                void Get_Density();
                void Get_Pressure();

                void Get_Stop(ParPro);

                void UpdatePropertiesFields();
                void Get_Circulation(Mesher, ParPro);
                void Get_AeroCoefficients(Mesher);

                void Send_Coefficients(Mesher, ParPro);

                void EscalarVTK2D(Mesher, string, string, string, double*, double*, int, i
                void EscalarVTK2DInt(Mesher, string, string, string, double*, int*, int, i
                void VectorialVTK2D(Mesher, string, string, string, double*, double*, doub

                void ExecuteSolver(Memory, ReadData, ParPro, Mesher);

};
```

Solver Class Cpp Code

```
#include <iostream>
#include <sstream>
#include <fstream>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <cmath>
#include <chrono>
#include <mpi.h>

#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod
#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod
#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod
#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod
#include "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Codes/HeaderCod

using namespace std;

#define PI 3.141592653589793

#define G(i,j,dim) (((j) + (i)*NY) + NX*NY*(dim)) //Global Index
#define GU(i,j,dim) (((j) + (i)*NY) + (NX+1)*NY*(dim)) //Global Index Matriz U
#define GR(i,j,dim) (((j) + (i)*(NY+1)) + NX*(NY+1)*(dim)) //Global Index Matriz R

#define MU(i,j,dim) ((j) + NY*(dim))
#define MR(i,j,dim) ((i) + (Fx - Ix)*(dim) + (Fx - Ix)*(4)*(j))
```

```
#define VR(i,j,dim) ((i) + (Fx - Ix)*(j))

#define LU(i, j, dim) (((j) + ((i)-Ix) * NY)) //Local Index Axial Velocity Nodes
#define LR(i, j, dim) (((j) + ((i)-Ix) * (NY + 1))) //Local Index Radial Velocity Nodes

#define LNH(i,j,dim) (((j) + ((i) - Ix + Halo)*NY)) //Local Index No Halo
#define LSH(i,j,dim) (((j) + ((i) - Ix)*NY) + NY*(Fx-Ix + 2*Halo)*dim) //Local Index Si Ha

#define DIRECTORIO "/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/"

#define sind(x) sin((PI/180)*x)
#define cosd(x) cos((PI/180)*x)

//Constructor del mallador
Solver::Solver(Memory M1, ReadData R1, ParPro MPI1, Mesher MESH, int i){

        //Datos del problema
        NX = MESH.NX;
        NY = MESH.NY;

        //Datos para la computacion en paralelo
        Procesos = MPI1.Procesos;
        Rank = MPI1.Rank;
        Ix = MESH.Ix;
        Fx = MESH.Fx;
        Halo = MPI1.Get_Halo();

        Problema = R1.Problema;  //Problema (Canal/Canal con Cilindro/Perfil)
        ConvergenciaGlobal = 1e-6;
        ConvergenciaGS = 1e-6;

        //Geometr a del canal
        ChannelLength = R1.GeometryData[0]; //Longitud del canal
        ChannelHeight = R1.GeometryData[1]; //Altura del canal
        ChannelDepth = R1.GeometryData[2]; //Profundidad del canal

        CylinderRadius = 0.25; //Radio del cilindro (m)

        //Datos f sicos del problema
        Uref = R1.ProblemPhysicalData[0];
        Vref = R1.ProblemPhysicalData[1];
        RhoRef = R1.ProblemPhysicalData[2];
        Tref = R1.ProblemPhysicalData[3];


        Cp = R1.ProblemPhysicalData[5];
        Gamma = R1.ProblemPhysicalData[6];
        Rideal = R1.ProblemPhysicalData[7];
        Pref = R1.ProblemPhysicalData[4];

        NumericalCirculation = 0.0;

        AnguloAtaque = R1.AttackAngle[i];
        PhiCilindro = 0.10*Uref*ChannelHeight;
```

```
}

//Alojamiento de memoria para las matrices necesarias
void Solver::AllocateMatrix(Memory M1){

        //Matrices de los mapas de propiedades del problema

                //Matrices globales de propiedades

                if(Rank == 0){

                        //Step Presente
                        PhiGlobalPres = M1.AllocateDouble(NX, NY, 1);
                        UglobalPres = M1.AllocateDouble(NX, NY, 1);
                        VglobalPres = M1.AllocateDouble(NX, NY, 1);
                        TglobalPres = M1.AllocateDouble(NX, NY, 1);
                        PglobalPres = M1.AllocateDouble(NX, NY, 1);

                        //Step Futuro
                        PhiGlobalFut = M1.AllocateDouble(NX, NY, 1);
                        UglobalFut = M1.AllocateDouble(NX, NY, 1);
                        VglobalFut = M1.AllocateDouble(NX, NY, 1);
                        TglobalFut = M1.AllocateDouble(NX, NY, 1);
                        RhoGlobalFut = M1.AllocateDouble(NX, NY, 1);
                        PglobalFut = M1.AllocateDouble(NX, NY, 1);

                        UwallsMR_Global = M1.AllocateDouble(NX, NY + 1, 1);
                        VwallsMU_Global = M1.AllocateDouble(NX + 1, NY, 1);

                        PDiff = M1.AllocateDouble(Procesos, 1, 1);
                        PCirc = M1.AllocateDouble(Procesos, 1, 1);

                        GlobalDensityIndicator = M1.AllocateInt(NX, NY, 1);

                        awGlobal = M1.AllocateDouble(NX, NY, 1);
                        aeGlobal = M1.AllocateDouble(NX, NY, 1);
                        asGlobal = M1.AllocateDouble(NX, NY, 1);
                        anGlobal = M1.AllocateDouble(NX, NY, 1);
                        apGlobal  = M1.AllocateDouble(NX, NY, 1);
                }


                //Matrices locales de propiedades

                DensityIndicator = M1.AllocateInt(Fx - Ix + 2 * Halo, NY, 1);

                //Streamline
                PhiLocalPres = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
                PhiLocalFut = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);

                PhiLocalSup = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);

                //Velocidad Horizontal
                UlocalPres = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
                UlocalFut = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
        //Velocidad Vertical
        VlocalPres = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
        VlocalFut = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);

        //Temperatura
        TlocalPres = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
        TlocalFut = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);

        //Densidad
        RhoLocalFut = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);

        //Presi n
        PlocalPres = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
        PlocalFut = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);

        //Matrices de valores de las propiedades en las paredes de los vol menes
        UwallsMR = M1.AllocateDouble(Fx - Ix, NY + 1, 1);
        VwallsMU = M1.AllocateDouble(Fx - Ix + 1, NY, 1);

        RhoWallsMU = M1.AllocateDouble(Fx - Ix + 1, NY, 1);
        RhoWallsMR = M1.AllocateDouble(Fx - Ix, NY + 1, 1);

        //Matrices de c lculo de contribuciones a las propiedades

        aw = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
        ae = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
        as = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
        an = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);
        ap = M1.AllocateDouble(Fx - Ix + 2 * Halo, NY, 1);

        //Matrices y arrays de condiciones de contorno

        if(Rank == 0){
                PhiLeft = M1.AllocateDouble(1, NY, 1); //Presi n pared izquierda
                RhoLeft = M1.AllocateDouble(1, NY, 1); //Presi n pared izquierda
                Uleft = M1.AllocateDouble(1, NY, 1); //Velocidad axial pared izqui
                Vleft = M1.AllocateDouble(1, NY, 1); //Velocidad radial pared izqu
                Tleft = M1.AllocateDouble(1, NY, 1); //Temperaturas pared izquierd
                Pleft = M1.AllocateDouble(1, NY, 1); //Presi n pared izquierda
        }
        else if(Rank == Procesos - 1){
                PhiRight = M1.AllocateDouble(1, NY, 1); //Presi n pared derecha
                RhoRight = M1.AllocateDouble(1, NY, 1); //Presi n pared derecha
                Uright = M1.AllocateDouble(1, NY, 1); //Velocidad axial pared dere
                Vright = M1.AllocateDouble(1, NY, 1); //Velocidad radial pared der
                Tright = M1.AllocateDouble(1, NY, 1); //Temperaturas pared derecha
                Pright = M1.AllocateDouble(1, NY, 1); //Presi n pared derecha
        }

        PhiUp = M1.AllocateDouble(Fx - Ix, 1, 1);
        PhiDown = M1.AllocateDouble(Fx - Ix, 1, 1);

        RhoUp = M1.AllocateDouble(Fx - Ix, 1, 1);
        RhoDown = M1.AllocateDouble(Fx - Ix, 1, 1);

        Uup = M1.AllocateDouble(Fx - Ix, 1, 1);
```

```
                Udown = M1.AllocateDouble(Fx - Ix, 1, 1);

                Vup = M1.AllocateDouble(Fx - Ix, 1, 1);
                Vdown = M1.AllocateDouble(Fx - Ix, 1, 1);

                Tup = M1.AllocateDouble(Fx - Ix, 1, 1);
                Tdown = M1.AllocateDouble(Fx - Ix, 1, 1);

                Pup = M1.AllocateDouble(Fx - Ix, 1, 1);
                Pdown = M1.AllocateDouble(Fx - Ix, 1, 1);

}

//Inicializaci n de los campos de temperaturas
void Solver::InitializeFields(Mesher MESH){
int i, j;
double J;
double ny = NY;
double Xcentro = 0.50*ChannelLength;
double Ycentro = 0.50*ChannelHeight;

        for(i = Ix; i < Fx; i++){
                for(j = 0; j < NY; j++){
                        J = j;
                        PhiLocalPres[LNH(i,j,0)] = 0.5*Uref*ChannelHeight;
                        PhiLocalFut[LNH(i,j,0)] =  0.5*Uref*ChannelHeight;

                        PhiLocalSup[LNH(i,j,0)] =  0.5*Uref*ChannelHeight;

                        UlocalPres[LNH(i,j,0)] = Uref;
                        UlocalFut[LNH(i,j,0)] = Uref;

                        VlocalPres[LNH(i,j,0)] = 0.0;
                        VlocalFut[LNH(i,j,0)] = 0.0;

                        TlocalPres[LNH(i,j,0)] = Tref;
                        TlocalFut[LNH(i,j,0)] = Tref;

                        PlocalFut[LNH(i,j,0)] = Pref;


                                if(DensityIndicator[LNH(i,j,0)] == 1){
                                        RhoLocalFut[LNH(i,j,0)] = 0.0;
                                }
                                else{
                                        RhoLocalFut[LNH(i,j,0)] = RhoRef;
                                }

                }
        }

}


//Asignaci n de temperaturas a las condiciones de contorno
void Solver::UpdateBoundaryConditions(Mesher MESH){
```

```
int i, j;

        if(Rank == 0){

                for(j = 0; j < NY; j++){
                        RhoLeft[j] = RhoRef;
                        Pleft[j] = Pref;
                        Vleft[j] = Vref;
                        Uleft[j] = Uref;
                        Tleft[j] = Tref;
                        PhiLeft[j] = Uleft[j]*MESH.MP[G(0,j,1)];
                }

        }
        else if(Rank == Procesos - 1){

                for(j = 0; j < NY; j++){
                        RhoRight[j] = RhoLocalFut[LNH(NX-1,j,0)];
                        Pright[j] = PlocalFut[LNH(NX - 1,j,0)];
                        Vright[j] = 0.0;
                        Uright[j] = UlocalFut[LNH(NX - 1,j,0)];
                        Tright[j] = TlocalFut[LNH(NX - 1,j,0)];
                        PhiRight[j] = PhiLocalFut[LNH(NX - 1, j, 0)];
                }

        }

        for(i = Ix; i < Fx; i++){

                RhoUp[i - Ix] = RhoLocalFut[LNH(i,NY-1,0)];
                Pup[i - Ix] = 0.0;
                Vup[i - Ix] = Vref;
                Uup[i - Ix] = Uref;
                Tup[i - Ix] = 0.0;
                PhiUp[i - Ix] = MESH.MR[GR(i, NY, 1)]*Uref;

                RhoDown[i - Ix] = RhoLocalFut[LNH(i,0,0)];
                Pdown[i - Ix] = 0.0;
                Vdown[i - Ix] = Vref;
                Udown[i - Ix] = Uref;
                Tdown[i - Ix] = 0.0;
                PhiDown[i - Ix] = 0.0;
        }

}

//Seteo matriz de 0 y 1 para el c lculo de la densidad en los s lidos
void Solver::Get_DensityIndicatorCylinder(Mesher MESH){
int i,j;
double Xcentro = 0.50*ChannelLength;
double Ycentro = 0.50*ChannelHeight;

                for(i = Ix; i < Fx; i++){
                        for(j = 0; j < NY; j++){
                                if(pow(MESH.MP[G(i,j,0)] - Xcentro,2.0) + pow(MESH.MP[G(i,
                                        DensityIndicator[LNH(i,j,0)] = 1;
                                        }
                                        else{
                                                DensityIndicator[LNH(i,j,0)] = 0;
                                        }
                                }
                        }
}

//Seteo de matriz de 0 para el caso del perfil aerodin mico
void Solver::Get_DensityIndicatorAirfoil(Mesher MESH){
int i, j;
double Xcentro = 0.50*ChannelLength;
double Ycentro = 0.50*ChannelHeight;
double m = 0.04; //Curvatura m xima en (%)
double p = 0.4; //Posici n de la curvatura m xima (%)
double t = 0.12; //Espesor relativo m ximo
double Cuerda = 1.0; //Cuerda (c) (m)
double Alpha = AnguloAtaque; // ngulo  de ataque
double UpLimit;
double DownLimit;


                for(i = Ix; i < Fx; i++){
                        for(j = 0; j < NY; j++){
                                if(MESH.MP[G(i,j,0)] >= Xcentro && MESH.MP[G(i,j,0)] <= Xc
                                                UpLimit = (MESH.MP[G(i,j,0)] - Xcentro)*sin
                                                DownLimit = (MESH.MP[G(i,j,0)] - Xcentro)*
                                                if(MESH.MP[G(i,j,1)] <= UpLimit && MESH.MP
                                                        DensityIndicator[LNH(i,j,0)] = 1;
                                                }
                                                else{
                                                        DensityIndicator[LNH(i,j,0)] = 0;
                                                }

                                }
                                else{
                                        DensityIndicator[LNH(i,j,0)] = 0;
                                }

                        }
                }
}

//C lculo de las densidades en las paredes de lo vol menes de control
void Solver::Get_DensityWalls(Mesher MESH, ParPro MPI1){
int i, j;

        //Comunicaci n de densidades entre los procesos
        MPI1.SendData(RhoLocalFut, Ix, Fx);
        MPI1.ReceiveData(RhoLocalFut, Ix, Fx);

        MPI1.SendDataInt(DensityIndicator, Ix, Fx);
        MPI1.ReceiveDataInt(DensityIndicator, Ix, Fx);

        //Nodos U
        if(Rank != 0 && Rank != Procesos - 1){
                for(i = Ix; i < Fx + 1; i++){
                        for(j = 0; j < NY; j++){
                                if(DensityIndicator[LNH(i,j,0)] == 1 && DensityIndicator[LN
                                        RhoWallsMU[LU(i,j,0)] = 0.0;
                                }
                                if(DensityIndicator[LNH(i,j,0)] == 1 && DensityIndicator[LN
                                        RhoWallsMU[LU(i,j,0)] = (RhoRef/RhoLocalFut[LNH(i -
                                }
                                if(DensityIndicator[LNH(i,j,0)] == 0 && DensityIndicator[LN
                                        RhoWallsMU[LU(i,j,0)] = (RhoRef/RhoLocalFut[LNH(i,
                                }
                                if(DensityIndicator[LNH(i,j,0)] == 0 && DensityIndicator[LN
                                        RhoWallsMU[LU(i,j,0)] = MESH.DeltasMU[GU(i,j,0)]/(
(MESH.MP[G(i,j,0)] - MESH.MU[GU(i,j,0)])/(RhoRef/RhoLocalFut[LNH(i,j,0)]) +
(MESH.MU[GU(i,j,0)] - MESH.MP[G(i-1,j,0)])/(RhoRef/RhoLocalFut[LNH(i-1,j,0)]) );
                                }
                        }
                }

        }
        else if(Rank == 0){

                for(j = 0; j < NY; j++){

                        //Parte izquierda
                        RhoWallsMU[LU(0,j,0)] = RhoRef/RhoLeft[j];

                        for(i = Ix + 1; i < Fx + 1; i++){
                                if(DensityIndicator[LNH(i,j,0)] == 1 && DensityIndicator[LN
                                                RhoWallsMU[LU(i,j,0)] = 0.0;
                                }
                                if(DensityIndicator[LNH(i,j,0)] == 1 && DensityIndicator[LN
                                        RhoWallsMU[LU(i,j,0)] = (RhoRef/RhoLocalFut[LNH(i -
                                }
                                if(DensityIndicator[LNH(i,j,0)] == 0 && DensityIndicator[LN
                                        RhoWallsMU[LU(i,j,0)] = (RhoRef/RhoLocalFut[LNH(i,
                                }
                                if(DensityIndicator[LNH(i,j,0)] == 0 && DensityIndicator[LN
                                        RhoWallsMU[LU(i,j,0)] = MESH.DeltasMU[GU(i,j,0)]/(
(MESH.MP[G(i,j,0)] - MESH.MU[GU(i,j,0)])/(RhoRef/RhoLocalFut[LNH(i,j,0)]) +
(MESH.MU[GU(i,j,0)] - MESH.MP[G(i-1,j,0)])/(RhoRef/RhoLocalFut[LNH(i-1,j,0)]) );
                                }
                        }
                }
        }
        else if(Rank == Procesos - 1){

                for(j = 0; j < NY; j++){

                        //Parte derecha
                        RhoWallsMU[LU(NX,j,0)] = RhoRef/RhoLocalFut[LNH(NX-1,j,0)];

                        for(i = Ix-1; i < NX; i++){
                                if(DensityIndicator[LNH(i,j,0)] == 1 && DensityIndicator[LN
                                                RhoWallsMU[LU(i,j,0)] = 0.0;
                                }
                                if(DensityIndicator[LNH(i,j,0)] == 1 && DensityIndicator[LN
                                        RhoWallsMU[LU(i,j,0)] = (RhoRef/RhoLocalFut[LNH(i-
                                }
                                if(DensityIndicator[LNH(i,j,0)] == 0 && DensityIndicator[LN
                                        RhoWallsMU[LU(i,j,0)] = (RhoRef/RhoLocalFut[LNH(i,
                                }
                                if(DensityIndicator[LNH(i,j,0)] == 0 && DensityIndicator[LN
                                        RhoWallsMU[LU(i,j,0)] = MESH.DeltasMU[GU(i,j,0)]/(
(MESH.MP[G(i,j,0)] - MESH.MU[GU(i,j,0)])/(RhoRef/RhoLocalFut[LNH(i,j,0)])    +
(MESH.MU[GU(i,j,0)] - MESH.MP[G(i-1,j,0)])/(RhoRef/RhoLocalFut[LNH(i-1,j,0)])     );
                                }
                        }
                }

        }

        //Nodos R

        for(i = Ix; i < Fx; i++){

                //Parte abajo
                RhoWallsMR[LR(i,0,0)] = RhoRef/RhoDown[i - Ix];

                //Parte arriba
                RhoWallsMR[LR(i,NY,0)] = RhoRef/RhoUp[i - Ix];

                for(j = 1; j < NY; j++){
                        if(DensityIndicator[LNH(i,j,0)] == 1 && DensityIndicator[LNH(i,j-1
                                RhoWallsMR[LR(i,j,0)] = 0.0;
                        }
                        if(DensityIndicator[LNH(i,j,0)] == 1 && DensityIndicator[LNH(i,j-1
                                RhoWallsMR[LR(i,j,0)] = (RhoRef/RhoLocalFut[LNH(i,j-1,0)])
                        }
                        if(DensityIndicator[LNH(i,j,0)] == 0 && DensityIndicator[LNH(i,j-1
                                RhoWallsMR[LR(i,j,0)] = (RhoRef/RhoLocalFut[LNH(i,j,0)])*(
                        }
                        if(DensityIndicator[LNH(i,j,0)] == 0 && DensityIndicator[LNH(i,j-1
                                RhoWallsMR[LR(i,j,0)] = MESH.DeltasMR[GR(i,j,1)]/(
(MESH.MP[G(i,j,1)] - MESH.MR[GR(i,j,1)])/(RhoRef/RhoLocalFut[LNH(i,j,0)])    +
(MESH.MR[GR(i,j,1)] - MESH.MP[G(i,j-1,1)])/(RhoRef/RhoLocalFut[LNH(i,j-1,0)])     );
                        }

                }
        }
}

//C lculo de los coeficientes de discretizaci n
void Solver::Get_Coeficients(Mesher MESH){
int i, j;

        for(i = Ix; i < Fx; i++){
                for(j = 0; j < NY; j++){

                        aw[LNH(i,j,0)] = RhoWallsMU[LU(i,j,0)]*(MESH.DeltasMU[GU(i,j,1)]/M
                        ae[LNH(i,j,0)] = RhoWallsMU[LU(i+1,j,0)]*(MESH.DeltasMU[GU(i+1,j,1
                        as[LNH(i,j,0)] = RhoWallsMR[LR(i,j,0)]*(MESH.DeltasMR[GR(i,j,0)]/M
                        an[LNH(i,j,0)] = RhoWallsMR[LR(i,j+1,0)]*(MESH.DeltasMR[GR(i,j+1,0

                }
        }

        for(i = Ix; i < Fx; i++){
                for(j = 0; j < NY; j++){
                        ap[LNH(i,j,0)] = aw[LNH(i,j,0)] + ae[LNH(i,j,0)] + as[LNH(i,j,0)] +
                }
        }




}

void Solver::Send_Coefficients(Mesher MESH, ParPro MPI1){
int i, j;
char FileName[300];

        MPI1.SendMatrixToZero(aw, awGlobal, NX, NY, Procesos, Ix, Fx);
        MPI1.SendMatrixToZero(ae, aeGlobal, NX, NY, Procesos, Ix, Fx);
        MPI1.SendMatrixToZero(as, asGlobal, NX, NY, Procesos, Ix, Fx);
        MPI1.SendMatrixToZero(an, anGlobal, NX, NY, Procesos, Ix, Fx);
        MPI1.SendMatrixToZero(ap, apGlobal, NX, NY, Procesos, Ix, Fx);

        if(Rank == 0){


                sprintf(FileName, "MapaAw_Step_%d", 10);
                EscalarVTK2D(MESH, "ParaviewResults/PropertiesResults/", "aw", FileName, M

                        sprintf(FileName, "MapaAe_Step_%d", 10);
                EscalarVTK2D(MESH, "ParaviewResults/PropertiesResults/", "ae", FileName, M

                sprintf(FileName, "MapaAs_Step_%d", 10);
                EscalarVTK2D(MESH, "ParaviewResults/PropertiesResults/", "as", FileName, M

                sprintf(FileName, "MapaAn_Step_%d", 10);
                EscalarVTK2D(MESH, "ParaviewResults/PropertiesResults/", "an", FileName, M

                        sprintf(FileName, "MapaAp_Step_%d", 10);
                EscalarVTK2D(MESH, "ParaviewResults/PropertiesResults/", "ap", FileName, M
        }

}
void Solver::Get_MaxDifGS(ParPro MPI1){
int i, j;
MaxDiffGS = 0.0;
MPI_Status ST;

        for(i = Ix; i < Fx; i++){
                for(j = 0; j < NY; j++){
                        if(abs((PhiLocalFut[LNH(i,j,0)] - PhiLocalSup[LNH(i,j,0)])) >= Max
                                MaxDiffGS = abs((PhiLocalFut[LNH(i,j,0)] - PhiLocalSup[LNH
                        }
                }
        }

        MPI1.SendDataToZero(MaxDiffGS, PDiff);

        double Diff;
        if(Rank == 0){
                Diff = PDiff[0];
                for(i = 1; i < Procesos; i++){
                        if(PDiff[i] >= Diff){
                                Diff = PDiff[i];
                        }
                }
        }

        MPI1.SendDataToAll(Diff, MaxDiffGS);
}

//Resoluci n de las ecuaciones con Gauss-Seidel
void Solver::Get_Streamlines(ParPro MPI1, Mesher MESH){
int i, j;
MaxDiffGS = 2.0*ConvergenciaGS;

        while(MaxDiffGS >= ConvergenciaGS){

                if(Rank != 0 && Rank != Procesos - 1){

                        for(i = Ix; i < Fx; i++){
                                //Parte abajo
                                PhiLocalFut[LNH(i,0,0)] = (aw[LNH(i,0,0)]*PhiLocalFut[LNH(
                                //Parte arriba
                                PhiLocalFut[LNH(i,NY-1,0)] = (aw[LNH(i,NY-1,0)]*PhiLocalFu

                                for(j = 1; j < NY - 1; j++){
                                //      if(DensityIndicator[LNH(i,j,0)] == 1){
                                        //      PhiLocalFut[LNH(i,j,0)] = PhiCilindro;
                                //      }
                                //      else{
                                                PhiLocalFut[LNH(i,j,0)] = (aw[LNH(i,j,0)]*

                                //      }

                                }
                        }

                }
                else if(Rank == 0){

                                for(i = Ix + 1; i < Fx; i++){
                                        //Parte abajo
                                        PhiLocalFut[LNH(i,0,0)] = (aw[LNH(i,0,0)]*PhiLocalF

                                        //Parte arriba
                                        PhiLocalFut[LNH(i,NY-1,0)] = (aw[LNH(i,NY-1,0)]*Ph

                                        for(j = 1; j < NY - 1; j++){
```

```
                                                        //if(DensityIndicator[LNH(i,j,0)] == 1){
                                        //                      PhiLocalFut[LNH(i,j,0)] = PhiCilin
                                        //              }
                                        //              else{
                                                                PhiLocalFut[LNH(i,j,0)] = (aw[LNH(
                                        //              }
                                        }
                                }

                                //Parte izquierda
                                for(j = 1; j < NY - 1; j++){
                                        PhiLocalFut[LNH(0,j,0)] = (aw[LNH(0,j,0)]*PhiLeft[
                                }

                                //Esquina abajo izquierda
                                PhiLocalFut[LNH(0,0,0)] = (aw[LNH(0,0,0)]*PhiLeft[0] + ae[
                                
                                //Esquina arriba izquierda
                                PhiLocalFut[LNH(0,NY-1,0)] = (aw[LNH(0,NY-1,0)]*PhiLeft[NY

                }
                else if(Rank == Procesos - 1){

                        for(i = Ix; i < Fx - 1; i++){
                                //Parte abajo
                                PhiLocalFut[LNH(i,0,0)] = (aw[LNH(i,0,0)]*PhiLocalFut[LNH(

                                //Parte arriba
                                PhiLocalFut[LNH(i,NY-1,0)] = (aw[LNH(i,NY-1,0)]*PhiLocalFu

                                for(j = 1; j < NY - 1; j++){
                                //      if(DensityIndicator[LNH(i,j,0)] == 1){
                                        //      PhiLocalFut[LNH(i,j,0)] = PhiCilindro;
                                //      }
                                        //else{
                                                PhiLocalFut[LNH(i,j,0)] = (aw[LNH(i,j,0)]*
                                //      }
                                }
                        }

                        //Parte derecha
                        for(j = 1; j < NY - 1; j++){
                                PhiLocalFut[LNH(NX-1,j,0)] = (aw[LNH(NX-1,j,0)]*PhiLocalFu
                        }

                        //Esquina abajo derecha
                        PhiLocalFut[LNH(NX-1,0,0)] = (aw[LNH(NX-1,0,0)]*PhiLocalFut[LNH(NX

                        //Esquina arriba derecha
                        PhiLocalFut[LNH(NX-1,NY-1,0)] = (aw[LNH(NX-1,NY-1,0)]*PhiLocalFut[L

                }


                Get_MaxDifGS(MPI1);
                for(i = Ix; i < Fx; i++){
                        for(j = 0; j < NY; j++){
                                PhiLocalSup[LNH(i,j,0)] = PhiLocalFut[LNH(i,j,0)];
                        }
                }

                MPI1.SendData(PhiLocalFut, Ix, Fx);
                MPI1.ReceiveData(PhiLocalFut, Ix, Fx);

        }

}

//C lculo de los campos de velocidades
void Solver::Get_Velocities(Mesher MESH, ParPro MPI1){
int i, j;

        MPI1.SendData(PhiLocalFut, Ix, Fx);
        MPI1.ReceiveData(PhiLocalFut, Ix, Fx);

        //Velocidades Horizontales (Nodos R)
        for(i = Ix; i < Fx; i++){

                //Parte arriba
                UwallsMR[LR(i,NY,0)] = Uref;

                //Parte abajo
                UwallsMR[LR(i,0,0)] = Uref;

                for(j = 1; j < NY; j++){
                        UwallsMR[LR(i,j,0)] = (RhoWallsMR[LR(i,j,0)]*(PhiLocalFut[LNH(i,j,
                }
        }


        //Velocidades Verticales (Nodos U)

        if(Rank != 0 && Rank != Procesos - 1){
                for(j = 0; j < NY; j++){
                        for(i = Ix; i < Fx + 1; i++){
                                VwallsMU[LU(i,j,0)] = - (RhoWallsMU[LU(i,j,0)]*(PhiLocalFu
                        }
                }
        }
        else if(Rank == 0){
                for(j = 0; j < NY; j++){

                        //Parte izquierda
                        VwallsMU[LU(0,j,0)] = Vleft[j];

                        for(i = Ix + 1; i < Fx + 1; i++){
                                VwallsMU[LU(i,j,0)] = - (RhoWallsMU[LU(i,j,0)]*(PhiLocalFu
                        }
                }
        }
        else if(Rank == Procesos - 1){
                for(j = 0; j < NY; j++){

                        //Parte derecha
                        VwallsMU[LU(NX,j,0)] = Vright[j];

                        for(i = Ix; i < Fx; i++){
                                VwallsMU[LU(i,j,0)] = - (RhoWallsMU[LU(i,j,0)]*(PhiLocalFu
                        }
                }
        }

        for(i = Ix; i < Fx; i++){
                for(j = 0; j < NY; j++){

                                UlocalFut[LNH(i,j,0)] = 0.50*(UwallsMR[LR(i,j,0)] + UwallsM
                                VlocalFut[LNH(i,j,0)] = 0.50*(VwallsMU[LU(i,j,0)] + VwallsM

                }
        }
}

//C lculo del mapa de temperaturas futuro
void Solver::Get_Temperatures(){
int i, j;

        for(i = Ix; i < Fx; i++){
                for(j = 0; j < NY; j++){
                        TlocalFut[LNH(i,j,0)] = Tref + ((pow(Uref,2.0) + pow(Vref,2.0)) -
                }
        }

}

//C lculo del mapa de presiones futuro
void Solver::Get_Pressure(){
int i, j;

        for(i = Ix; i < Fx; i++){
                for(j = 0; j < NY; j++){
                        PlocalFut[LNH(i,j,0)] = Pref*pow(TlocalFut[LNH(i,j,0)]/Tref,Gamma/
                        //Pref + (1 - (pow(UlocalFut[LNH(i,j,0)],2.0) + pow(VlocalFut[LNH(
                }
        }

}

//C lculo del mapa de densidades futuro
void Solver::Get_Density(){
int i, j;

        for(i = Ix; i < Fx; i++){
                for(j = 0; j < NY; j++){
                        if(DensityIndicator[LNH(i,j,0)] == 1){
                                RhoLocalFut[LNH(i,j,0)] = 0.0;
                        }
                        else{
```

```
                                        RhoLocalFut[LNH(i,j,0)] = PlocalFut[LNH(i,j,0)]/(Rideal*Tl
                              }
                    }
          }

}

//C lculo de la diferencia de resultados entre Steps
void Solver::Get_Stop(ParPro MPI1){
int i, j;
MaxDiffGlobal = 0.0;

          if(Rank == 0){
                    for(i = 0; i < NX; i++){
                              for(j = 0; j < NY; j++){
                                        if(abs((TglobalFut[G(i,j,0)] - TglobalPres[G(i,j,0)])) >=
                                                  MaxDiffGlobal = abs((TglobalFut[G(i,j,0)] - Tgloba
                                        }
                                        if(abs((PglobalFut[G(i,j,0)] - PglobalPres[G(i,j,0)])) >=
                                                  MaxDiffGlobal = abs((PglobalFut[G(i,j,0)] - Pgloba
                                        }

                                        if(abs((PhiGlobalFut[G(i,j,0)] - PhiGlobalPres[G(i,j,0)]))
                                                  MaxDiffGlobal = abs((PhiGlobalFut[G(i,j,0)] - PhiG
                                        }
                                        if(abs((UglobalFut[G(i,j,0)] - UglobalPres[G(i,j,0)])) >=
                                                  MaxDiffGlobal = abs((UglobalFut[G(i,j,0)] - Ugloba
                                        }
                                        if(abs((VglobalFut[G(i,j,0)] - VglobalPres[G(i,j,0)])) >=
                                                  MaxDiffGlobal = abs((VglobalFut[G(i,j,0)] - Vgloba
                                        }
                              }
                    }
          }

          MPI1.SendDataToAll(MaxDiffGlobal, MaxDiffGlobal);


}

void Solver::UpdatePropertiesFields(){
int i, j;

          for(i = Ix; i < Fx; i++){
                    for(j = 0; j < NY; j++){

                              TlocalPres[LNH(i,j,0)] = TlocalFut[LNH(i,j,0)];
                              PlocalPres[LNH(i,j,0)] = PlocalFut[LNH(i,j,0)];
                              PhiLocalPres[LNH(i,j,0)] = PhiLocalFut[LNH(i,j,0)];
                              UlocalPres[LNH(i,j,0)] = UlocalFut[LNH(i,j,0)];
                              VlocalPres[LNH(i,j,0)] = VlocalFut[LNH(i,j,0)];

                    }
          }
}
```

```
//Pasar los resultados a un archivo VTK en 2D
void Solver::EscalarVTK2D(Mesher MESH, string Carpeta, string Variable, string NombreFile,
int i, j;

        ofstream file;
    stringstream InitialName;
    string FinalName;

        InitialName<<DIRECTORIO<<Carpeta<<NombreFile<<".vtk";

        FinalName = InitialName.str();
    file.open(FinalName.c_str());

        file<<"#␣vtk␣DataFile␣Version␣2.0"<<endl;
        file<<Variable<<endl;
        file<<"ASCII"<<endl;
        file<<endl;
        file<<"DATASET␣STRUCTURED_GRID"<<endl;
        file<<"DIMENSIONS"<<"␣␣␣"<<Na<<"␣␣␣"<<Nr<<"␣␣␣"<<1<<endl;
        file<<endl;
        file<<"POINTS"<<"␣␣␣"<<Na*Nr<<"␣␣␣"<<"double"<<endl;

                for(j = 0; j < Nr; j++){
                        for(i = 0; i < Na; i++){
                                file<<MC[G(i,j,0)]<<"␣␣␣"<<MC[G(i,j,1)]<<"␣␣␣"<<0.0<<endl;
                        }
                }

        file<<endl;
                file<<"POINT_DATA"<<"␣␣␣"<<Na*Nr<<endl;
        file<<"SCALARS␣"<<Variable<<"␣double"<<endl;
        file<<"LOOKUP_TABLE"<<"␣␣␣"<<Variable<<endl;
        file<<endl;

       for(j = 0; j < Nr; j++){
                        for(i = 0; i < Na; i++){
                                file<<GlobalField[G(i,j,0)]<<"␣";
                    }
            }

    file.close();

}

//Pasar los resultados a un archivo VTK en 2D
void Solver::EscalarVTK2DInt(Mesher MESH, string Carpeta, string Variable, string NombreFi
int i, j;

        ofstream file;
    stringstream InitialName;
    string FinalName;

        InitialName<<DIRECTORIO<<Carpeta<<NombreFile<<".vtk";

        FinalName = InitialName.str();
    file.open(FinalName.c_str());
```

```
          file<<"# vtk DataFile Version 2.0"<<endl;
          file<<Variable<<endl;
          file<<"ASCII"<<endl;
          file<<endl;
          file<<"DATASET STRUCTURED_GRID"<<endl;
          file<<"DIMENSIONS"<<"   "<<Na<<"   "<<Nr<<"   "<<1<<endl;
          file<<endl;
          file<<"POINTS"<<"   "<<Na*Nr<<"   "<<"double"<<endl;

                    for(j = 0; j < Nr; j++){
                              for(i = 0; i < Na; i++){
                                        file<<MC[G(i,j,0)]<<"   "<<MC[G(i,j,1)]<<"   "<<0.0<<endl;
                              }
                    }

          file<<endl;
                    file<<"POINT_DATA"<<"   "<<Na*Nr<<endl;
          file<<"SCALARS "<<Variable<<" double"<<endl;
          file<<"LOOKUP_TABLE"<<"   "<<Variable<<endl;
          file<<endl;

        for(j = 0; j < Nr; j++){
                              for(i = 0; i < Na; i++){
                                        file<<GlobalField[G(i,j,0)]<<" ";
                         }
               }

     file.close();

}

//Pasar a un .vtk los resultados de campos vectoriales
void Solver::VectorialVTK2D(Mesher MESH, string Carpeta, string Variable, string NombreFil
int i, j;

     ofstream file;
     stringstream InitialName;
     string FinalName;

          InitialName<<DIRECTORIO<<Carpeta<<NombreFile<<".vtk";

          FinalName = InitialName.str();
     file.open(FinalName.c_str());

          file<<"# vtk DataFile Version 2.0"<<endl;
          file<<Variable<<endl;
          file<<"ASCII"<<endl;
          file<<endl;
          file<<"DATASET STRUCTURED_GRID"<<endl;
          file<<"DIMENSIONS"<<"   "<<Na<<"   "<<Nr<<"   "<<1<<endl;
          file<<endl;
          file<<"POINTS"<<"   "<<Na*Nr<<"   "<<"double"<<endl;

                    for(j = 0; j < Nr; j++){
                              for(i = 0; i < Na; i++){
```

```
                                        file <<MC[G(i,j,0)]<<" "<<MC[G(i,j,1)]<<" "<<0.0<<endl;
                                }
                        }

            file <<endl;
            file <<"POINT_DATA"<<" "<<Na*Nr<<endl;
            file <<"VECTORS "<<Variable<<" double"<<endl;
            file <<endl;

            for(j = 0; j < Nr; j++){
                                for(i = 0; i < Na; i++){
                                file <<GlobalField1[G(i,j,0)]<<" "<<GlobalField2[G(i,j,0)]<<" "<
                    }
            }

        file.close();

}

//C lculo de la circulaci n total en el cilindro
void Solver::Get_Circulation(Mesher MESH, ParPro MPI1){
int i,j;
double Circu = 0.0;

            MPI1.SendData(RhoLocalFut, Ix, Fx);
            MPI1.ReceiveData(RhoLocalFut, Ix, Fx);

            for(i = Ix; i < Fx; i++){
                        for(j = 0; j < NY; j++){
                                if(DensityIndicator[LNH(i,j,0)] == 1){
                                        Circu += MESH.DeltasMU[GU(i+1,j,1)]*VwallsMU[LU(i+1,j,0)] 
                                }
                        }
            }

            MPI1.SendDataToZero(Circu, PCirc);


            if(Rank == 0){
                        for(i = 0; i < Procesos; i++){
                                NumericalCirculation += PCirc[i];
                        }

                        //cout<<"Numerical Circulation: "<<CirculationNumerical<<endl;
            }

}

//C lculo del Cd y del Cl del cilindro
void Solver::Get_AeroCoefficients(Mesher MESH){
int i, j;
double DragForce = 0.0;
double LiftForce = 0.0;
double Xcentro = 0.50*ChannelLength;
double Ycentro = 0.50*ChannelHeight;
double m = 0.04; //Curvatura m xima en (%)
double p = 0.4; //Posici n de la curvatura m xima (%)
double t = 0.12; //Espesor relativo m ximo
double Cuerda = 1.0; //Cuerda (c) (m)
double Alpha = AnguloAtaque; // ngulo  de ataque
double UpLimit;
double DownLimit;


                for(i = 0; i < NX; i++){
                        for(j = 0; j < NY; j++){
                                if(MESH.MP[G(i,j,0)] >= Xcentro && MESH.MP[G(i,j,0)] <= Xc
                                                UpLimit = (MESH.MP[G(i,j,0)] - Xcentro)*si
                                                DownLimit = (MESH.MP[G(i,j,0)] - Xcentro)*
                                                if(MESH.MP[G(i,j,1)] <= UpLimit && MESH.MP
                                                        GlobalDensityIndicator[G(i,j,0)] =
                                                }
                                                else{
                                                        GlobalDensityIndicator[G(i,j,0)] =
                                                }

                                }
                                else{
                                        GlobalDensityIndicator[G(i,j,0)] = 0;
                                }

                        }
                }
/*
        for(i = 0; i < NX; i++){
                for(j = 0; j < NY; j++){
                                if(pow(MESH.MP[G(i,j,0)] - Xcentro,2.0) + pow(MESH.MP[G(i,j
                                        GlobalDensityIndicator[G(i,j,0)] = 1;
                                        PglobalFut[G(i,j,0)] = 0.0;
                                }
                                else{
                                        GlobalDensityIndicator[G(i,j,0)] = 0;
                                }
                }
        }
*/
        FILE *fp5;
        fp5 = fopen("/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Num

        FILE *fp9;
        fp9 = fopen("/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFlows/Num
        for(i = 1; i < NX-1; i++){
                for(j = 1; j < NY-1; j++){
                        if(GlobalDensityIndicator[G(i,j,0)] == 1){
                                if(GlobalDensityIndicator[G(i+1,j,0)] == 0){

                                        DragForce += - MESH.DeltasMP[G(i,j,1)]*PglobalFut[G
                                }
                                if(GlobalDensityIndicator[G(i-1,j,0)] == 0){
                                        DragForce += MESH.DeltasMP[G(i,j,1)]*PglobalFut[G(
                                }
                                if(GlobalDensityIndicator[G(i,j-1,0)] == 0){
                                        fprintf(fp5,"%f \t %f \n",MESH.MP[G(i,j,0)] - 0.50
                                        LiftForce += MESH.DeltasMP[G(i,j,0)]*PglobalFut[G(
                                }
                                if(GlobalDensityIndicator[G(i,j+1,0)] == 0){
                                        fprintf(fp9,"%f \t %f \n",MESH.MP[G(i,j,0)] - 0.50
                                        LiftForce += - MESH.DeltasMP[G(i,j,0)]*PglobalFut[
                                }
                        }
                }
        }

        fclose(fp5);
        fclose(fp9);

        Cd = DragForce/(0.50*RhoRef*pow(Uref,2)*1.0);
        Cl = LiftForce/(0.50*RhoRef*pow(Uref,2)*1.0);

        FILE *fp1;
                fp1 = fopen("/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFl
                        fprintf(fp1,"%f \t %f \t %f \t %f \t %f \t %f \n", DragForce, LiftF

        fclose(fp1);

        FILE *fp2;
                fp2 = fopen("/home/sergiogus/Desktop/ComputationalEngineering/NonViscousFl
                        fprintf(fp2,"%f \t %f \t %f \t %f \n",-AnguloAtaque, Cl, LiftForce

        fclose(fp2);

}

//Ejecuci n de todos los procesos del solver
void Solver::ExecuteSolver(Memory M1, ReadData R1, ParPro MPI1, Mesher MESH){
int i, j;
int Step = 0;
char FileName[300];
MaxDiffGlobal = 2.0*ConvergenciaGlobal;

        AllocateMatrix(M1);
        Get_DensityIndicatorAirfoil(MESH);
        //Get_DensityIndicatorCylinder(MESH);
        InitializeFields(MESH);

        //Pasar todas las matrices al ZERO
        //Matrices
        MPI1.SendMatrixToZero(TlocalFut, TglobalFut, NX, NY, Procesos, Ix, Fx);
        MPI1.SendMatrixToZero(RhoLocalFut, RhoGlobalFut, NX, NY, Procesos, Ix, Fx);
        MPI1.SendMatrixToZero(PlocalFut, PglobalFut, NX, NY, Procesos, Ix, Fx);
        MPI1.SendMatrixToZero(UlocalFut, UglobalFut, NX, NY, Procesos, Ix, Fx);
        MPI1.SendMatrixToZero(PhiLocalFut, PhiGlobalFut, NX, NY, Procesos, Ix, Fx);
        MPI1.SendMatrixToZero(VlocalFut, VglobalFut, NX, NY, Procesos, Ix, Fx);

        if(Rank == 0){
```

```
			sprintf(FileName, "MapaStreamFunction_Step_%d", Step);
			EscalarVTK2D(MESH, "ParaviewResults/PropertiesResults/", "StreamFunctions"

			sprintf(FileName, "MapaPresiones_Step_%d", Step);
			EscalarVTK2D(MESH, "ParaviewResults/PropertiesResults/", "Presiones", FileN

			sprintf(FileName, "MapaTemperaturas_Step_%d", Step);
			EscalarVTK2D(MESH, "ParaviewResults/PropertiesResults/", "Temperaturas", F

			sprintf(FileName, "MapaDensidades_Step_%d", Step);
			EscalarVTK2D(MESH, "ParaviewResults/PropertiesResults/", "Densidades", File

			sprintf(FileName, "MapaVelocidades_Step_%d", Step);
			VectorialVTK2D(MESH, "ParaviewResults/PropertiesResults/", "Velocidades", 
		}

		auto start = std::chrono::high_resolution_clock::now();
		while(MaxDiffGlobal >= ConvergenciaGlobal){

			Step++;
			UpdateBoundaryConditions(MESH);

			Get_DensityWalls(MESH, MPI1);
			Get_Coeficients(MESH);

			Get_Streamlines(MPI1, MESH);

			Get_Velocities(MESH, MPI1);
			Get_Temperatures();
			Get_Pressure();

			Get_Density();

			if(Step%500 == 0){

				//Pasar todas las matrices al ZERO

				//Matrices Step Presente
				MPI1.SendMatrixToZero(TlocalPres, TglobalPres, NX, NY, Procesos, I
				MPI1.SendMatrixToZero(PlocalPres, PglobalPres, NX, NY, Procesos, I
				MPI1.SendMatrixToZero(PhiLocalPres, PhiGlobalPres, NX, NY, Procesos
				MPI1.SendMatrixToZero(UlocalPres, UglobalPres, NX, NY, Procesos, I
				MPI1.SendMatrixToZero(VlocalPres, VglobalPres, NX, NY, Procesos, I

				//Matrices Step Futuro
				MPI1.SendMatrixToZero(TlocalFut, TglobalFut, NX, NY, Procesos, Ix,
				MPI1.SendMatrixToZero(RhoLocalFut, RhoGlobalFut, NX, NY, Procesos,
				MPI1.SendMatrixToZero(PlocalFut, PglobalFut, NX, NY, Procesos, Ix,

				MPI1.SendMatrixToZero(UlocalFut, UglobalFut, NX, NY, Procesos, Ix,
				MPI1.SendMatrixToZero(PhiLocalFut, PhiGlobalFut, NX, NY, Procesos,
				MPI1.SendMatrixToZero(VlocalFut, VglobalFut, NX, NY, Procesos, Ix,


				Get_Stop(MPI1);
```

```
                        if(Rank == 0){

                                cout<<"Step: "<<Step<<", MaxDif: "<<MaxDiffGlobal<<endl;


                        }
                }
                UpdatePropertiesFields();

        }

        // Record end time
        auto finish = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> elapsed = finish - start;
        std::cout << "Elapsed time: " << elapsed.count() << " s\n";

        if(Rank == 0){

                                sprintf(FileName, "MapaStreamFunction_Step_%d", 2);
                                EscalarVTK2D(MESH, "ParaviewResults/PropertiesResults/", "

                                sprintf(FileName, "MapaPresiones_Step_%d", 2);
                                EscalarVTK2D(MESH, "ParaviewResults/PropertiesResults/", "

                                sprintf(FileName, "MapaTemperaturas_Step_%d", 2);
                                EscalarVTK2D(MESH, "ParaviewResults/PropertiesResults/", "

                                sprintf(FileName, "MapaDensidades_Step_%d", 2);
                                EscalarVTK2D(MESH, "ParaviewResults/PropertiesResults/", "

                                sprintf(FileName, "MapaVelocidades_Step_%d", 2);
                                VectorialVTK2D(MESH, "ParaviewResults/PropertiesResults/",

                        }

        Get_Circulation(MESH, MPI1);
        if(Rank == 0){
                Get_AeroCoefficients(MESH);
        }


}
```