MASTER'S DEGREE IN AERONAUTICAL ENGINEERING

Potential flow solution to a channel flow using FVM

COMPUTATIONAL ENGINEERING

Authors: Ricard Arbat Carandell

Professor: Carlos David Perez Segarra

Escola Superior d'Enginyeria Industrial, Aeroespacial i Audiovisual de Terrassa (ESEIAAT)
Dated: November 21, 2024

Contents

Index of Figures

1 Problem definition

The problem to be solved is a case of a flow constricted within a channel, where a cylinder has been in such a way that the incoming fluid approaches radially towards it.

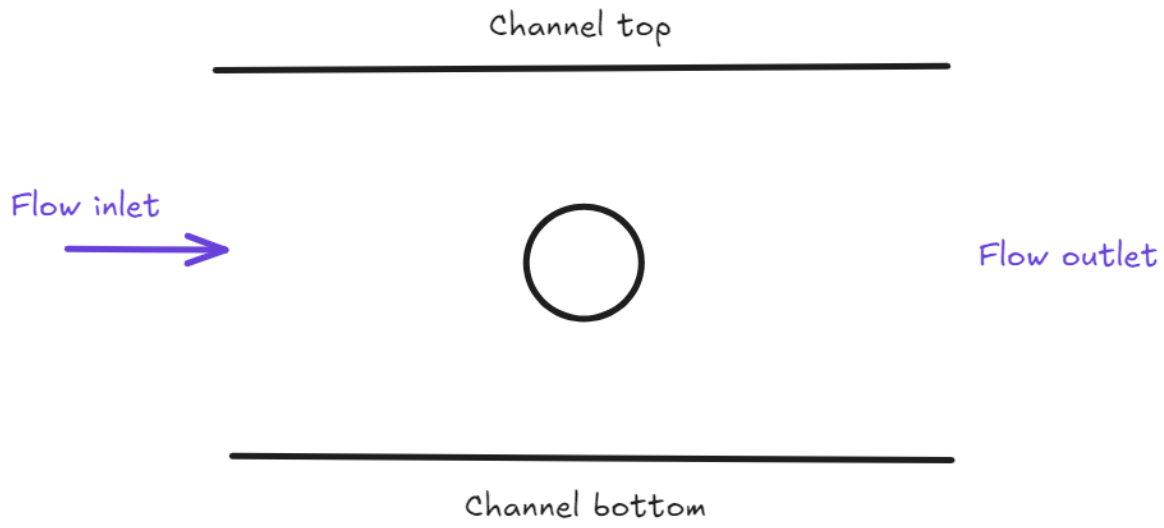


Figure 1. Diagram of the channel flow

1.1 Considerations

Some considerations need to be taken in order to simplify the case:

- **Potential flow**: As stated, the case will be computed using a numerical method based on the potential flow equations and the streamlines. In such cases, no friction or viscosity is considered in the fluid.
- **2D**: The problem will be reduced to a 2D simulation, as that is the only case that can be solved with the potential flow equations.
- **Steady flow**: The case will be also be steady, as it is a requirement from the potential flow method.
- **Incompressible**: The flow is also considered incompressible, so the density remains constant.

1.2 Boundary conditions

As for the boundary conditions, most of them are Dirichlet conditions, so they are fixed to a value. For the outflow, it can be assumed that the flow at that point is tangential to the channel

walls, and therefore, the streamline is equal to the west node.

- **Inlet**: $\psi_{in} = v_{in} \cdot y$
- **Outlet**: $\psi_{out} = \psi_W$
- **Bottom**: $\psi_{bot} = 0$
- **Top**: $\psi_{top} = v_{in} \cdot H$
- **Solid**: $\psi_{solid} = \frac{1}{2} v_{in} \cdot H$

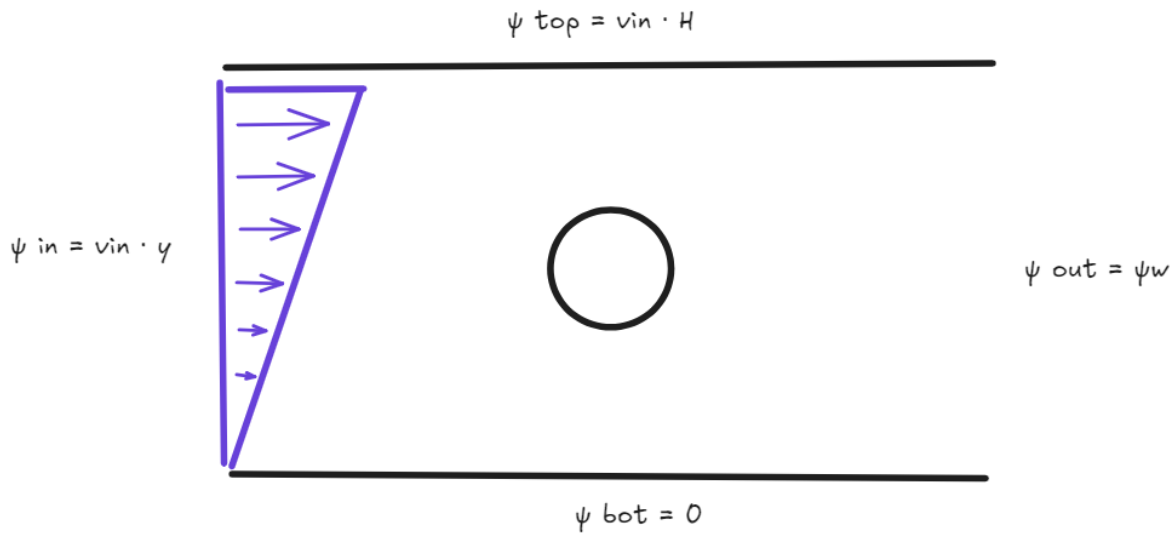


Figure 2. Boundary conditions of the domain

2 Numerical method

The method used to solve this case will be the Finite Volume Method (FVM). In this technique, the domain is discretized in small control volumes (or cells) centered around nodes. Then, by using the divergence theorem, the one can compute the flow of fluxes across the faces of each control volume, ensuring that properties such as mass, momentum, and energy are conserved across the domain.

2.1 Potential flow discretization

In the potential method, the main variable that is taken in account is the circulation (Γ) of fluid around a point.

Mathematically, the circulation can be described as the rotational of a closed surface, and using Green's theorem, it can be related to the integral of its sides.

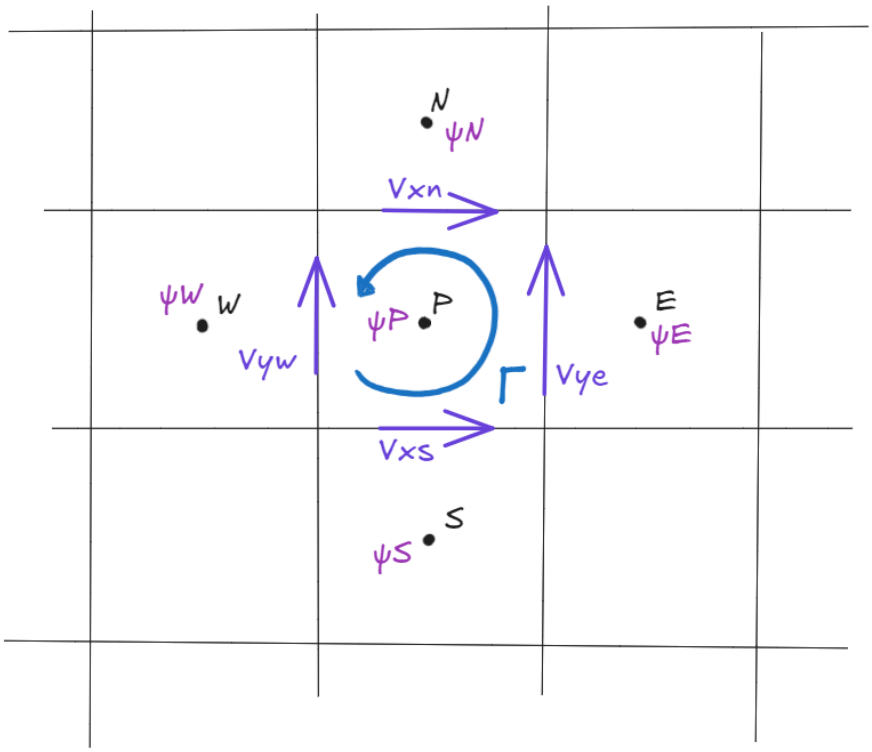


Figure 3. FVM diagram

$$\Gamma = \int_S (\nabla \times \mathbf{v}) dS = \int_C \mathbf{v} \cdot d\mathbf{l} \tag{1}$$

As the domain has been discretized in small cells, the circulation can be written as a simple sum of linear integrals

$$\Gamma = v_{ye}\Delta y_p - v_{xn}\Delta x_p - v_{yw}\Delta y_p + v_{xs}\Delta x_p \tag{2}$$

and as the streamline theory is going to be considered, the values of the velocities nearby volumes in Equation 4 can then be substituted,

$$v_x = \frac{\rho_{\text{ref}}}{\rho}\frac{\partial \psi}{\partial y} \quad v_y = \frac{\rho_{\text{ref}}}{\rho}\frac{\partial \psi}{\partial x} \tag{3}$$

resulting in the following differential equation:

$$\Gamma = -\frac{\rho_{\text{ref}}}{\rho_e}\frac{\partial \psi}{\partial x}\bigg|_e \Delta y_p - \frac{\rho_{\text{ref}}}{\rho_n}\frac{\partial \psi}{\partial y}\bigg|_n \Delta x_p + \frac{\rho_{\text{ref}}}{\rho_w}\frac{\partial \psi}{\partial x}\bigg|_w \Delta y_p + \frac{\rho_{\text{ref}}}{\rho_s}\frac{\partial \psi}{\partial y}\bigg|_s \Delta x_p \tag{4}$$

Then the derivatives can be simplified using second order approximations, finally yielding the equation that will need to be resolved in an iterative process:

$$\Gamma = -\frac{\rho_{\text{ref}}}{\rho_e}\frac{\psi_E - \psi_P}{d_{PE}}\Delta y_P - \frac{\rho_{\text{ref}}}{\rho_n}\frac{\psi_N - \psi_P}{d_{PN}}\Delta x_P + \frac{\rho_{\text{ref}}}{\rho_w}\frac{\psi_P - \psi_W}{d_{PW}}\Delta y_P + \frac{\rho_{\text{ref}}}{\rho_s}\frac{\psi_P - \psi_S}{d_{PS}}\Delta x_P \tag{5}$$

The expression is simplified using some coefficients,

$$a_P\psi_P = a_E\psi_E + a_W\psi_W + a_N\psi_N + a_S\psi_S + b_p \tag{6}$$

and is rewritten in a more useful way:

$$\psi_P = \frac{a_E\psi_E + a_W\psi_W + a_N\psi_N + a_S\psi_S + b_p}{a_P} \tag{7}$$

Where the coefficients are:

$$a_E = \frac{\rho_{\text{ref}}}{\rho_e}\frac{\Delta y_P}{d_{PE}}, \quad a_W = \frac{\rho_{\text{ref}}}{\rho_w}\frac{\Delta y_P}{d_{PW}}, \quad a_N = \frac{\rho_{\text{ref}}}{\rho_n}\frac{\Delta x_P}{d_{PN}}, \quad a_S = \frac{\rho_{\text{ref}}}{\rho_s}\frac{\Delta x_P}{d_{PS}} \tag{8}$$

2.2 Blocking off method

The blocking-off method is used because the domain includes both fluid and solid regions. In this method, each control volume is classified based on whether its center lies in the fluid or solid region (See Figure 4). If the center is within the solid, the entire volume is treated as solid; otherwise, it is considered fluid.

At the solid-fluid interface, while the velocity continuity is maintained, the stream function's derivative becomes discontinuous. This affects density calculations, which are handled using a harmonic mean to evaluate densities across the control volume faces at the interface.

$$\frac{\rho_{\text{ref}}}{\rho_e} = \frac{d_{PE}}{\frac{\rho_P}{d_{Pe}\,\rho_{ref}} + \frac{\rho_E}{d_{Ee}\,\rho_{ref}}} \tag{9}$$

2.3 Final calculations

2.3.1 Pressure Coefficient Calculation

The pressure coefficient C_p at each node is calculated using the following expression:

$$C_p = 1 - \left(\frac{v}{v_{\text{in}}}\right)^2 \tag{10}$$

Where v_{in} is the inlet velocity and $v = \sqrt{v_x^2 + v_y^2}$ is the velocity magnitude at a node.

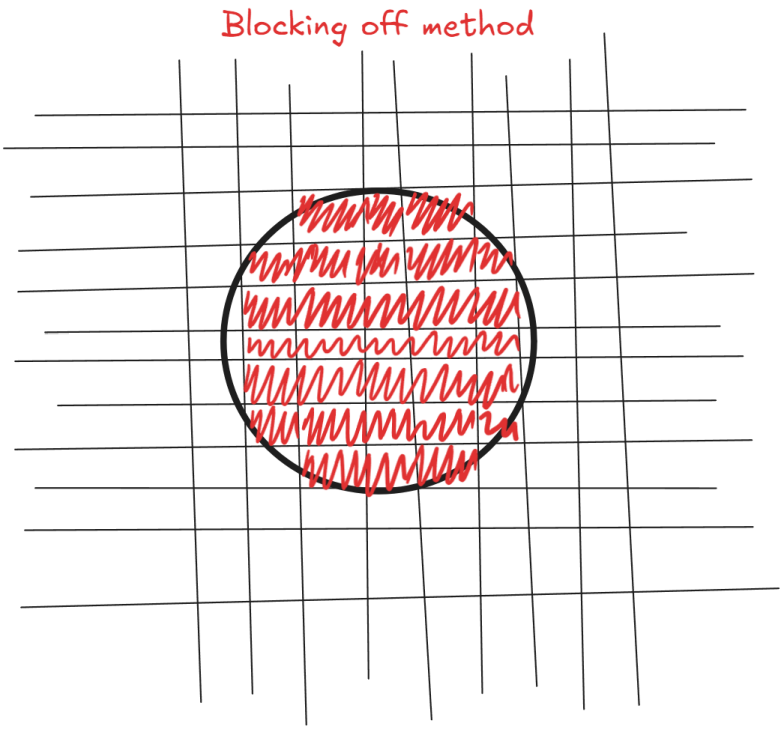


Figure 4. Blocking off method discretization

2.3.2 Pressure Calculation

The pressure at each node can be determined using the following relation, which accounts for changes in temperature:

$$p = p_{\text{in}} \left(\frac{T}{T_{\text{in}}} \right)^{\frac{\gamma}{\gamma - 1}} \tag{11}$$

Where p_{in} and T_{in} are the inlet pressure and temperature, and γ is the specific heat ratio.

2.3.3 Temperature Calculation

The temperature at each node is calculated as:

$$T = T_{\text{in}} + \frac{v_{\text{in}}^2 - v^2}{2c_p} \tag{12}$$

Where c_p is the specific heat at constant pressure, T_{in} is the inlet temperature, and v_{in} is the inlet velocity.

3 Code structure

The code has been developed in C++, while Matlab has been used to plot the different graphs.

3.1 Code structuring

The code has been structured in various C++ files:

- **parameters.cpp**: Structure including the parameters of the simulation
- **mesh.cpp**: Package with functions related to the mesh structure, and it's creation. Also allows mesh data export.
- **compute potential.cpp**: Package with functions related to the computation of the potential flow with numerical and analytical methods.
- **main.cpp**: Executes the simulation using the other packages.

A few Matlab files have also been used in order to plot the data obtained. However, these are not relevant for this report, so they will not be commented. And finally, all the results of the C++ calculations are stored in the "output" folder as CSV files, so that the Matlab files can open and process them.

3.2 Numerical resolution

To solve this problem, a Gauss-Seidel resolution scheme is followed. Overrelaxation schemes using the SOR method have been tested to speed up the process, but the resolutions diverged, so the basic method was used in the end. Up next, one can find the step-by-step guide of the process followed for the resolution.

1. Input of physical and numerical data in the **parameters.cpp** file.
2. Mesh generation and setup of the initial conditions using the **mesh.cpp** file.
3. Evaluation of new streamline values using the discretized coefficients using *computeStream* from **compute potential.cpp**. Then the values are compared to the previous ones, and if the error is low enough, the iteration is finished. This process is a Gauss-Seidel resolution with a SOR relaxation scheme.
4. Calculation of the velocities at the main nodes, and also the pressure coefficients.
5. Calculation of other properties for the fluid, such as temperature and pressure.
6. Calculation of the final coefficients and circulation.
7. Output of the final files in a .csv format for further processing.

4 Code verification

Before proceeding to using the code to obtain and check physical phenomena, it is important to check the validity of the code. For this reason, there are two things that can be done: compare to the analytical solution and check the mesh for different sizes and cases.

4.1 Analytical solution

The potential flow of the cylinder can be easily solved by applying its analytical equation

$$\psi_{cylinder} = U_{in}\left(r - \frac{R^2}{r}\right) sin(\theta) \tag{13}$$

But considering the gradient flow proportional to the distance from the bottom of the channel, one needs to consider its contribution:

$$\psi_{in} = U_{in} \cdot y = U_{in} \cdot r sin(\theta) \tag{14}$$

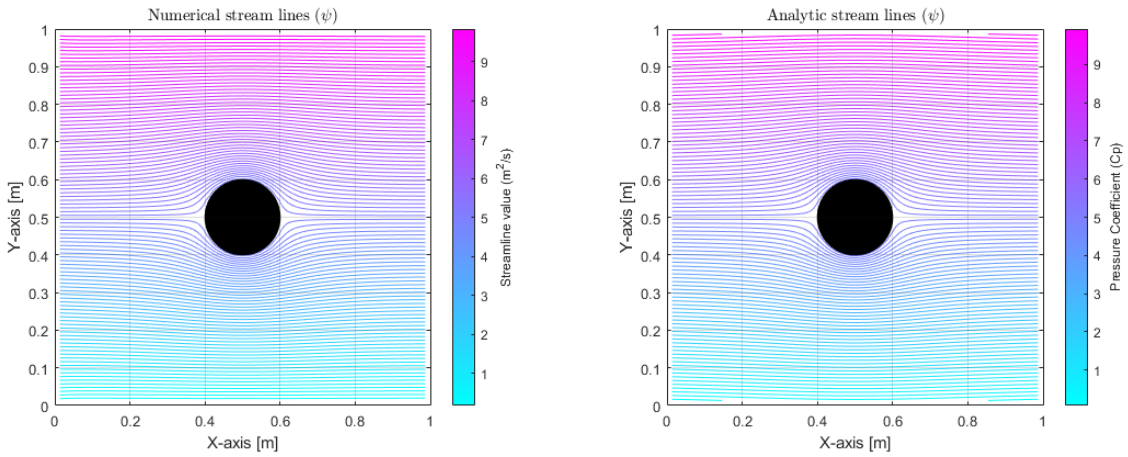
Yielding the final equation:

$$\psi_{in} = U_{in}\left(r sin(\theta) + \left(r - \frac{R^2}{r}\right) sin(\theta)\right) \tag{15}$$

In the case of the used mesh, one needs to correct the location of the origin of coordinates, as the analytical solution is centered around the cylinder, while the mesh origin is at the bottom left of the channel.

4.2 Analytical verification

In order to verify the results of the code, the numerical results have been compared to the analytical results.



(a) Numerical solution streamlines

(b) Analytical solution streamlines

Figure 5. Comparison of numerical and analytical cases in a mesh of N = M = 150, and L = H = 1m

The difference between both cases is quite small, but the relative error between both solutions can be also obtained. In Figure 6 we can see a peak of error in the center of the domain, where the center of the cylinder is. As this is out of the boundaries of the fluid, this high error can be ignored.

If the average value of the error is computed, this yields a result of 1.0949%, what is quite a good value considering the boundary conditions.

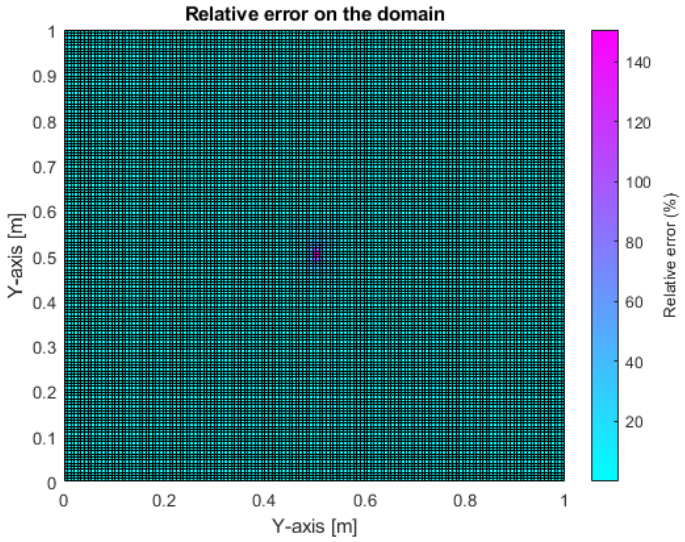


Figure 6. Relative error between the numerical and analytical solutions of the cylinder flow. Average error in the mesh of 1.0949%

If the error along a vertical line is plot 7, one can see how the error is maximum at near the boundary conditions, and it minimizes as we get closer to the center of the channel.

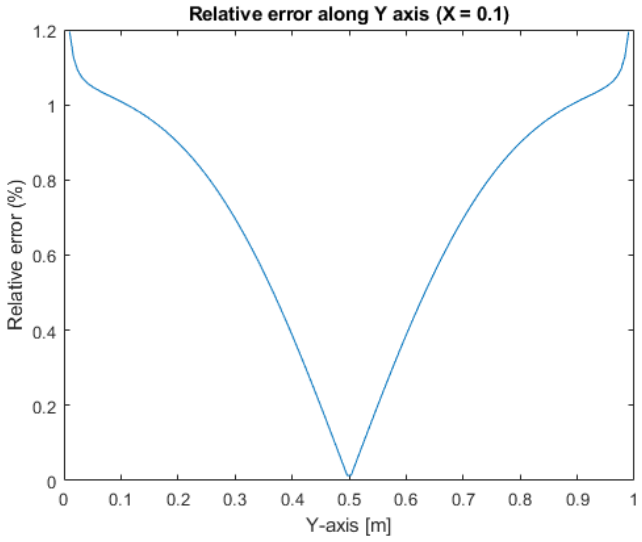


Figure 7. Error along the Y axis at x = 0.1m

4.3 Boundary conditions effect

It has also been tested how the boundary conditions affect both cases, as the analytical case does not include them. The previous analysis has been performed now with a bigger domain of L = H = 3m.

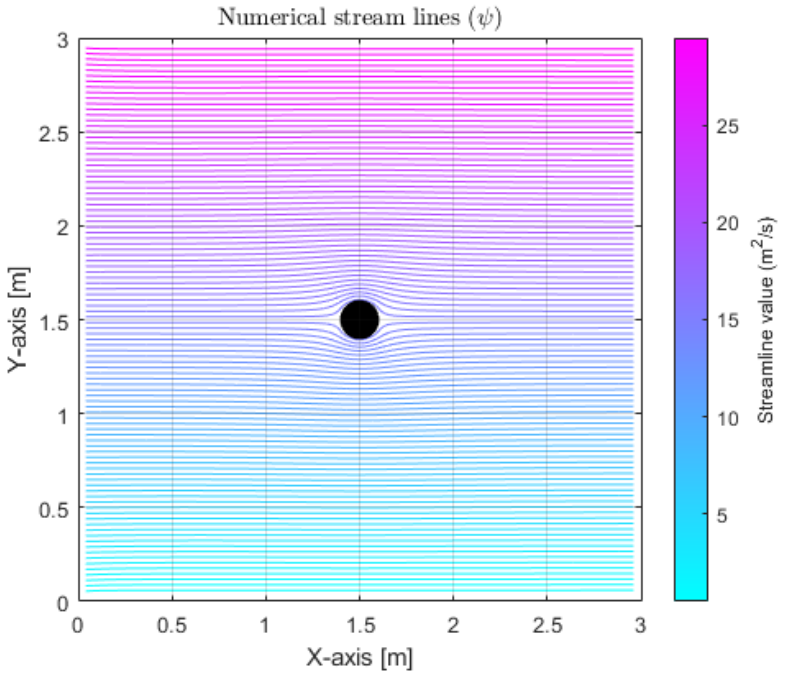


Figure 8. Numerical result in a mesh of N = M = 150, and L = H = 3m. Average relative error of 0.231%

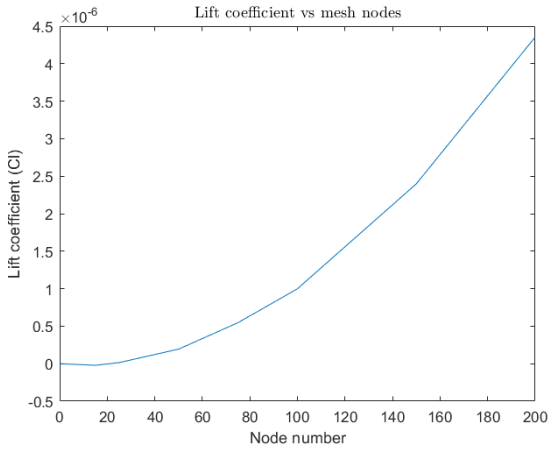
This study has resulted in a lower average error between numerical and analytical results of 0.231%. Therefore, the boundary effect has been considered to be present in the analysis, but is not very significant when comparing the cases.

As the cylinder gets bigger or the channel height lower, the boundary effects will be increased quite significantly, and that needs to be taken in account.

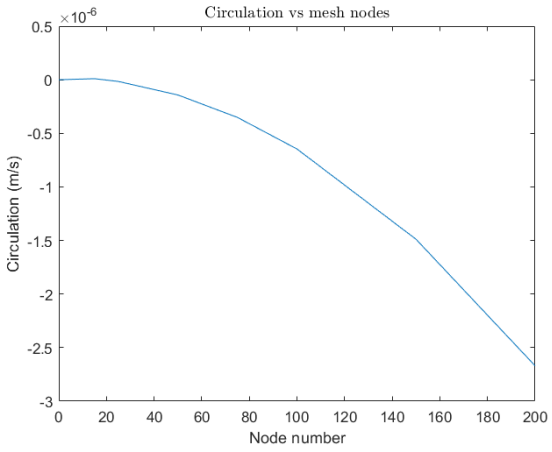
4.4 Mesh verification

The mesh has been tested for different levels of mesh refinement to see if the result converges to a value as the mesh gets finer. The study has been done for different mesh sizes, ranging from 15 nodes for both M and N, up to 200 nodes, and the value of the error (δ) has been set to $10e^{-10}$

Firstly, it has been seen how the values of the lift and the circulation (See Figure 9) don't converge to an exact value, but it diverges for these mesh densities. Nonetheless, the calculated values are both tiny, in the order of $10e^{-6}$.



(a) Lift coefficient variation with mesh refinement



(b) Circulation variation with mesh refinement

Figure 9. Comparison of numerical and analytical cases in a mesh of N = M = 150, and L = H = 1m

However, in the case of the drag, its value seems to converge to a value slightly negative value of around $10e^{-3}$. This small effect could be due to the boundary conditions having some effect on the drag. But overall, the results seem to be acceptable, as they are minimal.

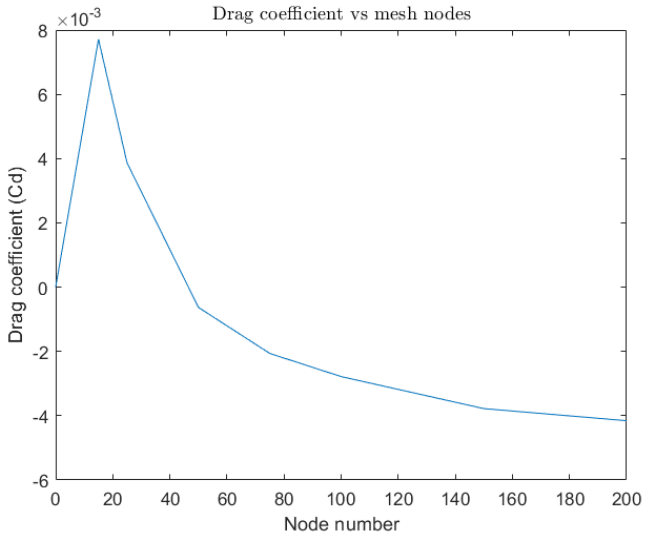


Figure 10. Drag coefficient variation with mesh refinement

Finally, the computation times have also been studied, and they have been measured to increase exponentially, making them much more challenging to calculate.

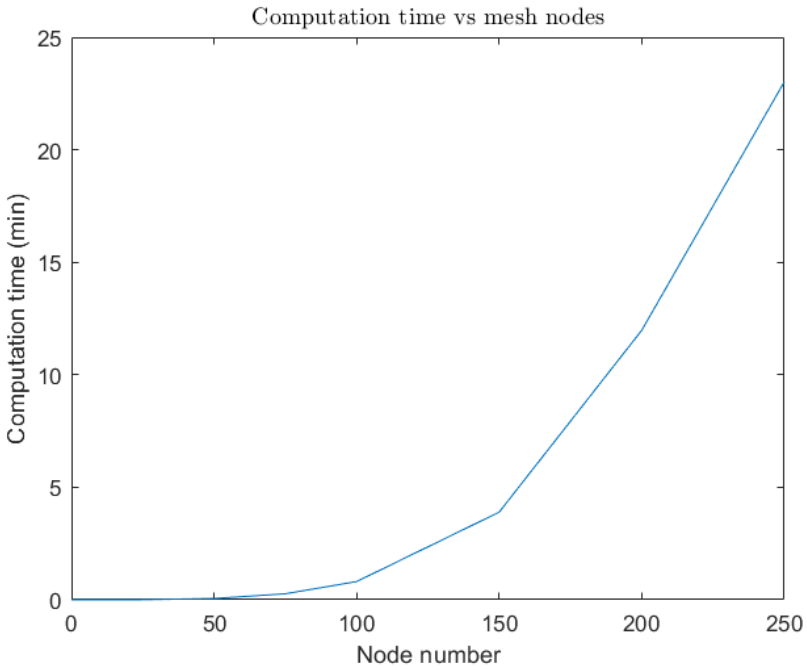


Figure 11. Computation time versus node number (N = M)

Tests with bigger meshes should also be done, but the time they take to compute is quite high, so they will not be computed and tested for this project.

5 Results and discussion

For both rotating and static analysis, the selected parameters have been mostly the same, except for the mesh refinement: The selected conditions have been the following:

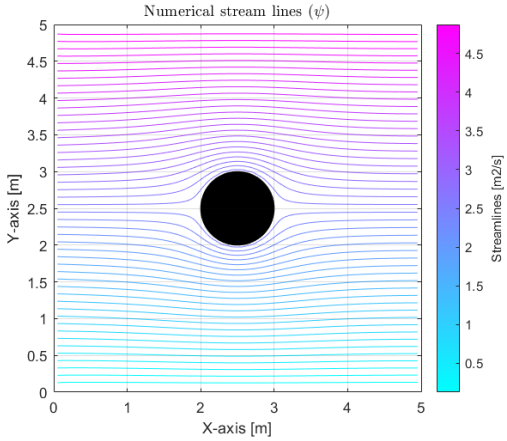
- $N = M = 200$
- $L = H = 5\ m$
- $R = 0.5\ m$
- $P_{in} = 1\ bar$
- $T_{in} = 298\ K$
- $v_{in} = 1\ m/s$
- $\rho_{in} = 1.225\ kg/m^3$
- $C_p = 1005\ J/kgK$
- $R = 287\ J/kgK$
- $\gamma = 1.4$
- $\psi_{solid} = 2.5m^2/s$

5.1 Static cylinder

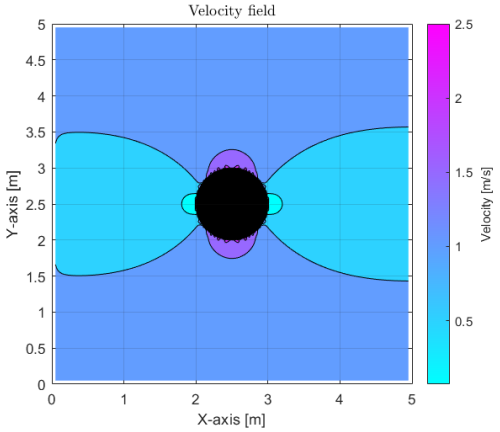
5.1.1 Static cylinder channel flow

Simulations have been carried out for the case of a static cylinder in a channel flow, and the results behave as expected.

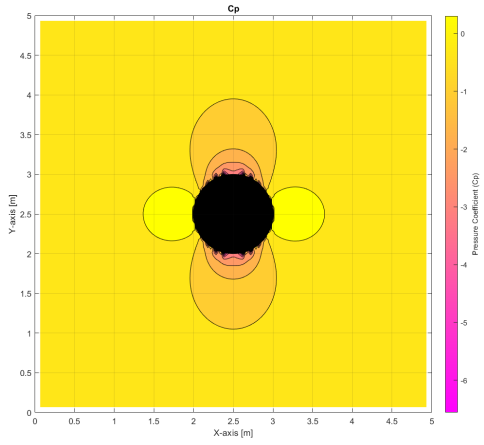
- **Streamlines**: Flow behaves as expected, diverting around the object, and in the boundary conditions it goes completely straight.
- **Velocity**: Velocities also behave as expected, with slower speeds at the front and back of the cylinder, while at the top and bottom the fluid speeds up to around 1.5m/s.
- **Pressure coefficient**: Pressure coefficient increases at the stagnation points in front and behind the cylinder, while it decreases substantially at the top and bottom, where the flow speeds up.
- **Pressure**: Behaves quite like the pressure coefficient, but as the speeds are quite low, the pressure increase or decrease is minimal.
- **Temperature**: At the parts where the flow speeds up, the temperature sees a slight decrease, but the change is minimal.



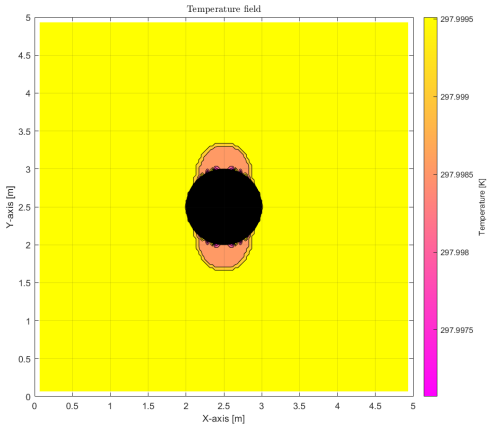
(a) Streamlines



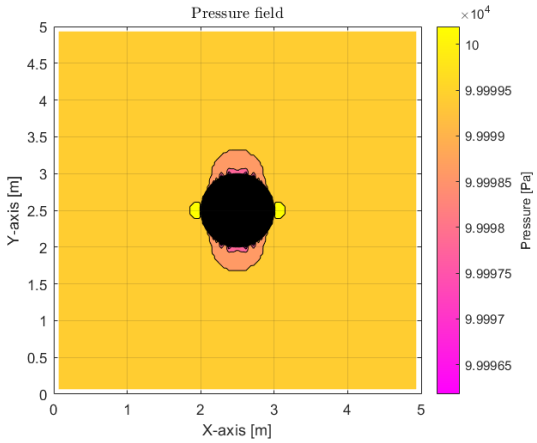
(b) Velocities field



(c) Pressure coefficient distribution



(d) Temperature distribution



(e) Pressure distribution

Figure 12. Results for the static case with N = M = 200

5.1.2 Static cylinder physical results

As for the overall results of the simulation, the values of circulation (Γ), lift coefficient(C_l) and drag coefficient (C_d) are very close to zero. In the case of the drag, the value is a bit higher than expected. This could be maybe due to the boundary conditions being closer to the cylinder.

- $C_l = 4.35 \cdot 10^{-6}$
- $C_d = -4.15 \cdot 10^{-3}$
- $\Gamma = -2.67 \cdot 10^{-6}$

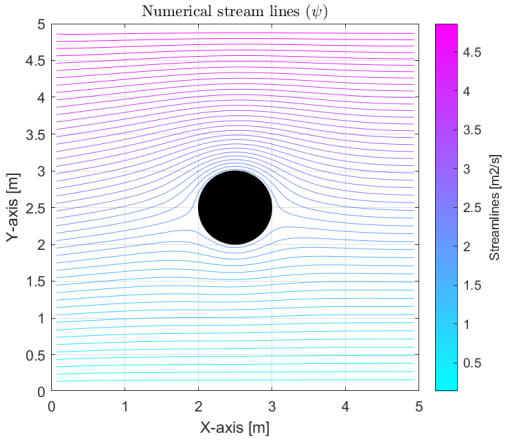
But overall, the results are the expected and no unexpected effects appear.

5.2 Rotating cylinder

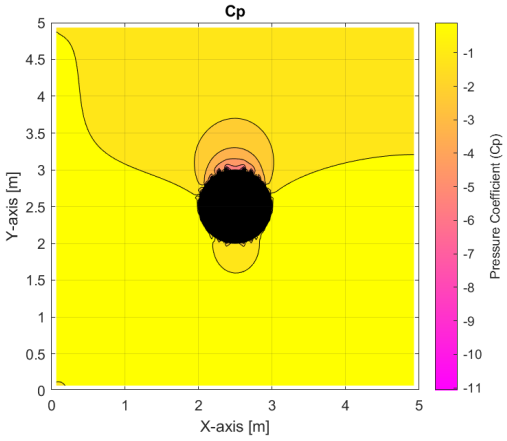
5.2.1 Rotating cylinder channel flow

Simulations have been carried out for the case of a rotating cylinder. All the parameters have been the same, except for the mesh refinement, where now it is coarser, at N = M = 150 and the value of the streamlines at the solid is. Also, in this case, the circulation is 80% the value of the circulation ($\psi_{solid} = 2m^2/s$) in the static case, so the spin is clockwise, and the lift is expected to be upwards.

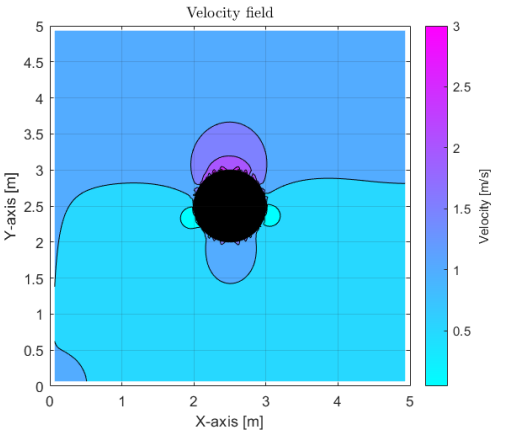
- **Streamlines**: Flow behaves as expected, with streamlines getting closer at the top, where the fluid speeds up. At the rest of the field, the lines do not change much.
- **Velocity**: Yet again, velocities also behave as expected and are coherent with the streamlines, speeding up significantly at the top.
- **Pressure coefficient**: Pressure coefficient decreases a quite a lot at the top of the cylinder, and it seems to get a bit swept back bi the flow,
- **Pressure**: Behaves quite like the pressure coefficient, but as the speeds are quite low, the pressure increase or decrease is minimal.
- **Temperature**: Now the temperature almost only decreases at the bottom, while the rest of the fluid does not change



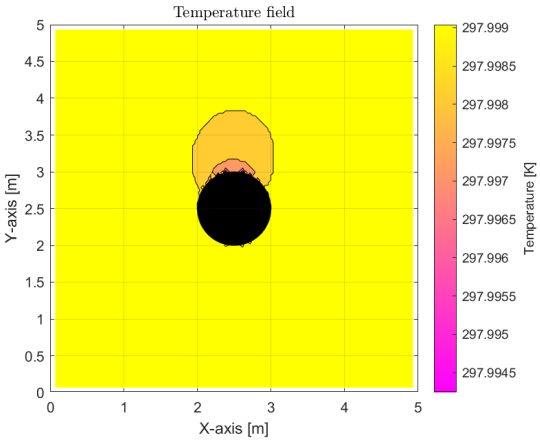
(a) Streamlines rotating



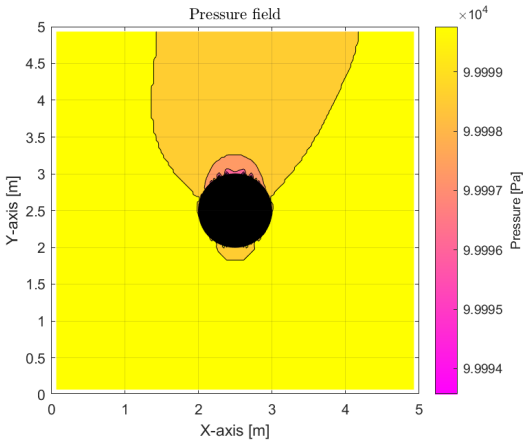
(b) Pressure coefficient distribution rotating



(c) Velocities field



(d) Temperature distribution rotating



(e) Pressure distribution rotating

Figure 13. Results for the rotating case with N = M = 150

5.2.2 Rotating cylinder physical results

The results for this case have also been as expected. The lift is positive, while the circulation negative and about half the value of the lift coefficient, as it should be. And again, the drag is close to zero.

- $C_l = 3.226$
- $C_d = -9.61 \cdot 10^{-2}$
- $\Gamma = -1.72$

The differences with the exact solutions can be due to the mesh refinement, the boundary conditions and the error in the resolution scheme. If a smaller mesh was used at the boundary conditions of the cylinder, a more accurate solution should be found, while not increasing the computational power in such a critical way.

References

[1] CTTC. (n.d.). *Course on numerical methods in heat transfer and fluid dynamics: Non-viscous flows*. (accessed: 23-09-2024).

[2] W3Schools. (n.d.). *Learn c++*. https://www.w3schools.com/cpp/ (accessed: 25-09-2024).

[3] Wikipedia. (n.d.[a]). *Potential flow around a circular cylinder*. https://en.wikipedia.org/wiki/Potential_flow_around_a_circular_cylinder (accessed: 27-09-2024).

[4] Wikipedia. (n.d.[b]). *Successive over-relaxation*. https://en.wikipedia.org/wiki/Successive_over-relaxation (accessed: 01-10-2024).

A C++ Code

A.1 Parameters

The parameters file includes a struct used to contain all the main parameters of the simulation. This way, the number of variables that are passed to the functions is reduced.

```cpp
// Last update: 2024/10/06
// Author: Ricard Arbat Carandell

// Master in Aerospace Engineering - Computational Engineering
// Universitat Politècnica de Catalunya (UPC) - BarcelonaTech
// Overview: Potential flow solution in a channel around a cilinder using the stream function formulation.

#include <iostream>
#include <cmath>
#include <fstream>
#include <array>

const int N = 250; // number of rows
const int M = N;   // number of columns

/**
 * Struct containing the parameters of the simulation. These values are used
 * by other functions to calculate the results of the simulation.
 *
 */

struct Parameters
{
    double L = 5;                                   // length of the channel
    double H = 5;                                   // height of the channel
    double cylinder_x = L / 2;                      // x position of the cylinder
    double cylinder_y = H / 2;                      // y position of the cylinder
    double cylinder_r = 0.5;                        // radius of the cylinder
    double p_in = 100000;                           // inlet pressure
    double t_in = 298;                              // inlet temperature
    double v_in = 1;                                // inlet velocity
    double rho_in = 1.225;                          // inlet density
    double initial_density = 1.225;                 // initial density value
    double solid_density = pow(10, -10);            // solid density (very small value at solid nodes)
    double spining_factor = 0.8;                    // spinning factor
```

```
    double solid_stream = H * 0.5 * v_in;          // stream value at solid
    nodes
    double start_stream = 10;                      // initial stream value
    double delta = 0.000000001;                    // maximum delta for the
    error
    double relaxation_factor = 1;                  // relaxation factor
    double initial_error = 1.4;                    // initial error
    double dx = L / N;                             // x step
    double dy = L / M;                             // ystep
    double R = 287;                                // gas constant
    double gamma = 1.4;                            // specific heat ratio
    double specific_heat = 1005;                   // specific heat
    std::string folder = "output_final/";          // output file name
    std::string mesh_number = std::to_string(N); // mesh number
};
```

A.2 Meshing

The mesh file contains functions used to build the mesh, set its values and export the data stored in it.

```
// Last update: 2024/10/06
// Author: Ricard Arbat Carandell

// Master in Aerospace Engineering - Computational Engineering
// Universitat Politècnica de Catalunya (UPC) - BarcelonaTech
// Overview: Mesh definition functions

#include <iostream>
#include <vector>
#include <cmath>
#include <fstream>
#include "parameters.cpp"
using namespace std;

// node struct
struct node
{
    double x, y, u, v, cp, stream, rho, T, p;
    bool is_solid;
};

/**
 * Fills the mesh with nodes as it defines their positions.
 * Also defines the cylinder solid nodes by considering if they are inside a
    circle
 *
```

```
26  * @param mesh Mesh matrix (vector of vectors) to be filled with Node
        structs
27  * @param p Parameters of the simulation
28  */
29 void buildMesh(vector<vector<node>> &mesh, Parameters p)
30 {
31
32     for (int i = 0; i < N; i++)
33     {
34         for (int j = 0; j < M; j++)
35         {
36             mesh[i][j].x = (i * p.dx) + 0.5 * p.dx;
37             mesh[i][j].y = (j * p.dy) + 0.5 * p.dy;
38
39             double dist = sqrt(pow(mesh[i][j].x - p.cylinder_x, 2) + pow(
       mesh[i][j].y - p.cylinder_y, 2));
40             if (dist < p.cylinder_r)
41             {
42                 mesh[i][j].is_solid = true;
43             }
44             else
45             {
46                 mesh[i][j].is_solid = false;
47             }
48         }
49     }
50 }
51
52 /**
53  * Sets the initial density of the mesh nodes.
54  * Also checks if the node is solid and sets the solid density
55  *
56  * @param mesh Mesh matrix
57  * @param p Parameters of the simulation
58  */
59 void setRho(vector<vector<node>> &mesh, Parameters p)
60 {
61     for (int i = 0; i < N; i++)
62     {
63         for (int j = 0; j < M; j++)
64         {
65             if (mesh[i][j].is_solid == true)
66             {
67                 mesh[i][j].rho = p.solid_density;
68             }
69             else
70             {
```

```
71                  mesh[i][j].rho = p.initial_density;
72              }
73          }
74      }
75  }
76
77  /**
78   * Sets the initial stream value of the mesh nodes.
79   * Also checks if the node is solid and sets the solid density.
80   *
81   * @param mesh Mesh matrix
82   * @param p Parameters of the simulation
83   */
84  void setStream(vector<vector<node>> &mesh, Parameters p)
85  {
86      for (int i = 0; i < N; i++)
87      {
88          for (int j = 0; j < M; j++)
89          {
90              if (mesh[i][j].is_solid)
91              {
92                  mesh[i][j].stream = p.solid_stream;
93              }
94              else
95              {
96                  mesh[i][j].stream = p.start_stream;
97              }
98          }
99      }
100 }
101
102 /**
103  * Exports the mesh data to a CSV file
104  *
105  * @param mesh Mesh matrix
106  * @param filename Name of the file to be exported
107  */
108 void exportData(vector<vector<node>> &mesh, string filename = "output/output
        .csv")
109 {
110     ofstream outfile(filename);
111     outfile << "X,Y,U,V,Stream,Density,Cp,Solid,P,T" << endl;
112     for (int i = 1; i < N - 1; i += 1)
113     {
114         for (int j = 1; j < M - 1; j += 1)
115         {
```

```
116             outfile << mesh[i][j].x << "," << mesh[i][j].y << "," << mesh[i
    ][j].u << "," << mesh[i][j].v << "," << mesh[i][j].stream << "," << mesh[
    i][j].rho << "," << mesh[i][j].cp << "," << mesh[i][j].is_solid << "," <<
     mesh[i][j].p << "," << mesh[i][j].T << endl;
117         }
118     }
119     outfile.close();
120 }
```

A.3 Potential method computation

The compute potential file contains all the algorithms used to compute the simulation. It also includes functions to compute the analytical resolution and compare the values of the numerical and analytical results.

```
1  // Last update: 2024/10/06
2  // Author: Ricard Arbat Carandell
3
4  // Master in Aerospace Engineering - Computational Engineering
5  // Universitat Politècnica de Catalunya (UPC)
6  // Overview: Stream computing functions
7
8  #define pass (void)0
9
10 #include <iostream>
11 #include <vector>
12 #include <cmath>
13 #include <fstream>
14 #include "mesh.cpp"
15 using namespace std;
16
17 struct Coefficients
18 {
19     double C_L, C_D;
20 };
21
22 /**
23  * Compares two matrices and returns the maximum error between them.
24  *
25  * @param v1 Matrix (Vector of vectors) to compare
26  * @param v2 Matrix (Vector of vectors) to compare
27  */
28 double streamsError(vector<vector<double>> v1, vector<vector<double>> v2)
29 {
30     double maxError = 0.0;
31     for (int i = 0; i < v1.size(); i++)
32     {
```

```
33          for (int j = 0; j < v1[i].size(); j++)
34          {
35              double diff = abs(v1[i][j] - v2[i][j]);
36              if (diff > maxError)
37              {
38                  maxError = diff;
39              }
40          }
41      }
42      return maxError;
43  }
44
45  /**
46   * Fills the initial stream values of the mesh nodes.
47   *
48   * @param initial_stream_value Matrix (Vector of vectors) containing the
        next stream values
49   * @param mesh Mesh matrix (vector of vectors) to be filled with Node
        structs
50   * @param p Parameters of the simulation
51   */
52  void fillStream(vector<vector<double>> &initial_stream_value, vector<vector<
        node>> &mesh, Parameters p)
53  {
54      for (int i = 0; i < N; i++)
55      {
56          for (int j = 0; j < M; j++)
57          {
58              if (mesh[i][j].is_solid == true)
59              {
60                  initial_stream_value[i][j] = p.solid_stream;
61              }
62              else
63              {
64                  initial_stream_value[i][j] = p.start_stream;
65              }
66          }
67      }
68  }
69
70  /**
71   * Computes the stream function of the mesh nodes. It uses the discretized
        stream function
72   * equation to solve the stream values of the nodes. The function iterates
        until the error
73   * is below a certain threshold.
74   *
```

```
75  * @param mesh Mesh matrix (vector of vectors) to be filled with Node
        structs
76  * @param p Parameters of the simulation
77  */
78 void computeStream(vector<vector<node>> &mesh, Parameters p)
79 {
80     double an, ae, as, aw, ap, b_p;
81     double dPE, dPe, dEe, dPS, dPs, dSs, dPW, dPw, dWw, dPN, dPn, dNn;
82     double error = p.initial_error;
83     double gauss_seidel;
84 
85     vector<vector<double>> next_stream_value(N, vector<double>(M));
86     vector<vector<double>> last_stream_value(N, vector<double>(M));
87 
88     fillStream(next_stream_value, mesh, p);
89     int cont = 0;
90     while (error > p.delta)
91     {
92         last_stream_value = next_stream_value;
93         for (int i = 0; i < N; i++)
94         {
95             for (int j = 0; j < M; j++)
96             {
97 
98                 if (j == 0) // Bottom channel nodes
99                 {
100                     next_stream_value[i][j] = 0;
101                 }
102                 else if (j == M - 1) // Top channel nodes
103                 {
104                     next_stream_value[i][j] = p.v_in * p.H;
105                 }
106                 else if (i == 0) // Inlet nodes
107                 {
108                     next_stream_value[i][j] = p.v_in * mesh[i][j].y;
109                 }
110                 else if (i == N - 1) // Outlet nodes
111                 {
112 
113                     next_stream_value[i][j] = last_stream_value[i - 1][j];
114                 }
115                 else // Internal nodes
116                 {
117                     // East node
118                     dPE = p.dx;
119                     dPe = p.dx / 2;
120                     dEe = p.dx / 2;
```

```
                    ae = (p.dx / ((dPe * mesh[i][j].rho + dEe * mesh[i + 1][
    j].rho) / p.rho_in)) * p.dy / dPE;

                    // South node
                    dPS = p.dy;
                    dPs = p.dy / 2;
                    dSs = p.dy / 2;
                    as = (dPS / ((dPs * mesh[i][j].rho + dSs * mesh[i][j -
    1].rho) / p.rho_in)) * p.dx / dPS;

                    // West node
                    dPW = p.dx;
                    dPw = p.dx / 2;
                    dWw = p.dx / 2;
                    aw = (dPW / ((dPw * mesh[i][j].rho + dWw * mesh[i - 1][j
    ].rho) / p.rho_in)) * p.dy / dPW;

                    // North node
                    dPN = p.dy;
                    dPn = p.dy / 2;
                    dNn = p.dy / 2;
                    an = (dPN / ((dPn * mesh[i][j].rho + dNn * mesh[i][j +
    1].rho) / p.rho_in)) * p.dx / dPN;

                    // Discretization of the stream function equation
                    ap = an + ae + as + aw;
                    gauss_seidel = (last_stream_value[i + 1][j] * ae +
    last_stream_value[i - 1][j] * aw + last_stream_value[i][j + 1] * an +
    last_stream_value[i][j - 1] * as + b_p) / ap;
                    next_stream_value[i][j] = last_stream_value[i][j] + p.
    relaxation_factor * (gauss_seidel - last_stream_value[i][j]);
                }
            }
        }
        error = streamsError(next_stream_value, last_stream_value);
        cont++;
        if (cont == 100)
        {
            printf("Error = %f\n", error);
            cont = 0;
        }
    }

    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < M; j++)
        {
```

```
161             mesh[i][j].stream = next_stream_value[i][j];
162         }
163     }
164 }
165
166 /**
167  * Calculates the velocity of the mesh nodes. It uses the stream function values
168  * to calculate the velocity components of the nodes.
169  *
170  * @param mesh Mesh matrix (vector of vectors) to be filled with Node structs
171  * @param p Parameters of the simulation
172  */
173 void calculateVelocity(vector<vector<node>> &mesh, Parameters p)
174 {
175     double vxn, vye, vxs, vyw, dPN, dPn, dNn, dPE, dPe, dEe, dPS, dPs, dSs, dPW, dPw, dWw;
176
177     for (int i = 1; i < N - 1; i++)
178     {
179         for (int j = 1; j < M - 1; j++)
180         {
181
182             dPE = p.dy;
183             dPe = p.dy / 2;
184             dEe = p.dy / 2;
185             vye = -(dPE / ((dPe * mesh[i][j].rho + dEe * mesh[i + 1][j].rho) / p.rho_in)) * ((mesh[i + 1][j].stream - mesh[i][j].stream) / dPE);
186
187             dPS = p.dx;
188             dPs = p.dx / 2;
189             dSs = p.dx / 2;
190             vxs = -(dPS / ((dPs * mesh[i][j].rho + dSs * mesh[i][j - 1].rho) / p.rho_in)) * ((mesh[i][j - 1].stream - mesh[i][j].stream) / dPS);
191
192             dPW = p.dy;
193             dPw = p.dy / 2;
194             dWw = p.dy / 2;
195             vyw = (dPW / ((dPw * mesh[i][j].rho + dWw * mesh[i - 1][j].rho) / p.rho_in)) * ((mesh[i - 1][j].stream - mesh[i][j].stream) / dPW);
196
197             // North node
198             dPN = p.dx;
199             dPn = p.dx / 2;
200             dNn = p.dx / 2;
```

```
201             vxn = (dPN / ((dPn * mesh[i][j].rho + dNn * mesh[i][j + 1].rho)
    / p.rho_in)) * ((mesh[i][j + 1].stream - mesh[i][j].stream) / dPN);
202
203             mesh[i][j].u = (vxn + vxs) / 2;
204             mesh[i][j].v = (vye + vyw) / 2;
205         }
206     }
207 }
208
209 /**
210  * Calculates the circulation around the cylinder. It uses the velocity
        values
211  * to calculate the circulation of each cell around the cylinder, adding all
        of them up.
212  *
213  * @param mesh Mesh matrix (vector of vectors) to be filled with Node
        structs
214  * @param p Parameters of the simulation
215  */
216 double calculateCylinderCirculation(vector<vector<node>> &mesh, Parameters p
    )
217 {
218     double vxn, vye, vxs, vyw, dPN, dPn, dNn, dPE, dPe, dEe, dPS, dPs, dSs,
    dPW, dPw, dWw, circ = 0;
219
220     for (int i = 1; i < N - 1; i++)
221     {
222         for (int j = 1; j < M - 1; j++)
223         {
224             dPN = p.dy;
225             dPn = p.dy / 2;
226             dNn = p.dy / 2;
227             vxn = (dPN / ((dPn * mesh[i][j].rho + dNn * mesh[i][j + 1].rho /
    p.rho_in))) * ((mesh[i][j + 1].stream - mesh[i][j].stream) / dPN);
228
229             dPE = p.dx;
230             dPe = p.dx / 2;
231             dEe = p.dx / 2;
232             vye = -(dPE / ((dPe * mesh[i][j].rho + dEe * mesh[i + 1][j].rho
    / p.rho_in))) * ((mesh[i + 1][j].stream - mesh[i][j].stream) / dPE);
233
234             dPS = p.dy;
235             dPs = p.dy / 2;
236             dSs = p.dy / 2;
237             vxs = -(dPS / ((dPs * mesh[i][j].rho + dSs * mesh[i][j - 1].rho
    / p.rho_in))) * ((mesh[i][j - 1].stream - mesh[i][j].stream) / dPS);
238
```

```
239             dPW = p.dx;
240             dPw = p.dx / 2;
241             dWw = p.dx / 2;
242             vyw = (dPW / ((dPw * mesh[i][j].rho + dWw * mesh[i - 1][j].rho /
    p.rho_in))) * ((mesh[i - 1][j].stream - mesh[i][j].stream) / dPW);
243
244             if (mesh[i][j].is_solid == true)
245             {
246                 if (mesh[i][j - 1].is_solid == false)
247                 {
248                     circ += vxs * p.dx;
249                 }
250                 if (mesh[i][j + 1].is_solid == false)
251                 {
252                     circ += -vxn * p.dx;
253                 }
254                 if (mesh[i - 1][j].is_solid == false)
255                 {
256                     circ += -vyw * p.dy;
257                 }
258                 if (mesh[i + 1][j].is_solid == false)
259                 {
260                     circ += vye * p.dy;
261                 }
262             }
263         }
264     }
265     return circ;
266 }
267
268 /**
269  * Calculates the pressure coefficient of the mesh nodes. It uses the
    velocity values
270  * to calculate the pressure coefficient of the nodes.
271  *
272  * @param mesh Mesh matrix (vector of vectors) to be filled with Node
    structs
273  * @param p Parameters of the simulation
274  */
275 void calculateCp(vector<vector<node>> &mesh, Parameters p)
276 {
277     double v = 0;
278     for (int i = 1; i < N - 1; i++)
279     {
280         for (int j = 1; j < M - 1; j++)
281         {
```

```
282             v = sqrt(mesh[i][j].u * mesh[i][j].u + mesh[i][j].v * mesh[i][j
    ].v);
283             mesh[i][j].cp = 1 - (v / p.v_in) * (v / p.v_in);
284         }
285     }
286 }
287
288 /**
289  * Calculates pressure and temperature of the mesh nodes.
290  *
291  * @param mesh Mesh matrix (vector of vectors) to be filled with Node
        structs
292  * @param p Parameters of the simulation
293  */
294 void calculatePressureTemperature(vector<vector<node>> &mesh, Parameters p)
295 {
296     double v = 0;
297     for (int i = 1; i < N - 1; i++)
298     {
299         for (int j = 1; j < M - 1; j++)
300         {
301             v = sqrt((mesh[i][j].u * mesh[i][j].u) + (mesh[i][j].v * mesh[i
    ][j].v));
302             mesh[i][j].T = p.t_in + ((p.v_in * p.v_in) - (v * v)) / (2 * p.
    specific_heat);
303             mesh[i][j].p = p.p_in * pow(mesh[i][j].T / p.t_in, p.gamma / (p.
    gamma - 1));
304             mesh[i][j].cp = 1 - (v / p.v_in) * (v / p.v_in);
305         }
306     }
307 }
308
309 /**
310  * Calculates the forces around the cylinder. It uses the pressure
        coefficient values
311  * to calculate the lift and drag coefficients of the cylinder.
312  *
313  * @param mesh Mesh matrix (vector of vectors) to be filled with Node
        structs
314  * @param p Parameters of the simulation
315  */
316 struct Coefficients cylinderForces(vector<vector<node>> &mesh, Parameters p)
317 {
318     Coefficients c;
319
320     double q = 0.5 * p.rho_in * p.p_in * p.v_in;
321     for (int i = 1; i < N - 1; i++)
```

```
322     {
323         for (int j = 1; j < M - 1; j++)
324         {
325             if (mesh[i][j].is_solid == true)
326             {
327                 if (mesh[i][j - 1].is_solid == false)
328                 {
329                     c.C_L += (mesh[i][j - 1].cp * q + p.p_in) * p.dx;
330                 }
331 
332                 if (mesh[i][j + 1].is_solid == false)
333                 {
334                     c.C_L += -(mesh[i][j + 1].cp * q + p.p_in) * p.dx;
335                 }
336 
337                 if (mesh[i - 1][j].is_solid == false)
338                 {
339                     c.C_D += (mesh[i - 1][j].cp * q + p.p_in) * p.dy;
340                 }
341 
342                 if (mesh[i + 1][j].is_solid == false)
343                 {
344                     c.C_D += -(mesh[i + 1][j].cp * q + p.p_in) * p.dy;
345                 }
346             }
347         }
348     }
349 
350     c.C_L = c.C_L / (q * 2 * p.cylinder_r);
351     c.C_D = c.C_D / (q * 2 * p.cylinder_r);
352 
353     return c;
354 }
355 
356 /**
357  * Converts Cartesian coordinates to polar coordinates.
358  *
359  * @param x The x-coordinate in Cartesian coordinates.
360  * @param y The y-coordinate in Cartesian coordinates.
361  * @return A pair where the first element is the radius (r) and the second
       element is the angle (theta) in radians.
362  */
363 pair<double, double> cartesianToPolar(double x, double y)
364 {
365     double r = sqrt(x * x + y * y);
366     double theta = atan2(y, x);
367     return make_pair(r, theta);
```

```
368 }
369
370 /**
371  * Computes the analytic stream function of the mesh nodes.
372  *
373  * @param mesh Mesh matrix (vector of vectors) to be filled with Node
        structs
374  * @param p Parameters of the simulation
375  */
376 void computeAnalyticStream(vector<vector<node>> &mesh, Parameters p)
377 {
378     for (int i = 0; i < N; i++)
379     {
380         for (int j = 0; j < M; j++)
381         {
382             pair<double, double> polar;
383             pair<double, double> cartesian;
384
385             polar = cartesianToPolar(mesh[i][j].x - p.H / 2, mesh[i][j].y -
        p.H / 2);
386
387             double input_flow = p.v_in * p.H / 2;
388             double cylinder = p.v_in * (polar.first - (p.cylinder_r * p.
        cylinder_r) / polar.first) * sin(polar.second);
389             // double circulation = p.cylinder_r * p.cylinder_r * log(polar.
        first) * p.v_in; // <missing rotation speed
390
391             mesh[i][j].stream = input_flow + cylinder; // + circulation;
392         }
393     }
394 }
395
396 /**
397  * Compares two matrices and returns the maximum error between them.
398  *
399  * @param v1 Matrix (Vector of vectors) to compare
400  * @param v2 Matrix (Vector of vectors) to compare
401  */
402 vector<vector<node>> analyticError(vector<vector<node>> numerical, vector<
        vector<node>> analytical)
403 {
404     vector<vector<node>> error_matrix(N, vector<node>(M));
405     for (int i = 0; i < N; i++)
406     {
407         for (int j = 0; j < M; j++)
408         {
```

```
409             error_matrix[i][j].stream = numerical[i][j].stream - analytical[
    i][j].stream;
410         }
411     }
412     return error_matrix;
413 }
```

A.4 Main

Finally, the main file uses all of the defined functions in order to define the parameters, build the mesh, and compute the numerical resolution. It is the entry point of the program, and the one to be executed.

```
1  // Last update: 2024/10/06
2  // Author: Ricard Arbat Carandell
3
4  // Master in Aerospace Engineering - Computational Engineering
5  // Universitat Politècnica de Catalunya (UPC) - BarcelonaTech
6  // Overview: Potential flow solution in a channel around a cilinder using
       the stream function formulation.
7
8  // Libraries
9  #include <iostream>
10 #include <vector>
11 #include <cmath>
12 #include <fstream>
13 #include "compute_potential.cpp"
14 #include <chrono>
15 using namespace std;
16 using namespace std::chrono;
17
18 /**
19  * Main function of the program. It initializes the mesh, computes the
       stream
20  * function and calculates the forces around the cilinder.
21  */
22 int main(void)
23 {
24
25     // Main mesh
26     Parameters p;
27     vector<vector<node>> mesh(N, vector<node>(M));
28     buildMesh(mesh, p); // creating the mesh
29     setStream(mesh, p); // setting initial values
30     setRho(mesh, p);
31
32     // Compute stream function
```

```
33      auto start = high_resolution_clock::now();
34      computeStream(mesh, p); // stream solver
35      auto stop = high_resolution_clock::now();
36      auto duration = duration_cast<microseconds>(stop - start);
37      calculateVelocity(mesh, p);                                    //
        calculating velocity
38      calculateCp(mesh, p);                                          //
        calculating pressure coeficient distribution
39      calculatePressureTemperature(mesh, p);                         //
        calculating pressure and temperature
40      double circulation = calculateCylinderCirculation(mesh, p); // calculate
         circulation around cilinder
41      struct Coefficients c = cylinderForces(mesh, p);               // calculate
         forces around cilinder
42      string name = p.folder + p.mesh_number + "_output.csv";
43      exportData(mesh, name); // export data to file output.dat
44
45      name = p.folder + p.mesh_number + "_results.csv";
46      ofstream outfile_r(name);
47      outfile_r << "L,H,R,N,M,Cl,Cd,Circ,t" << endl;
48      outfile_r << p.L << "," << p.H << "," << p.cylinder_r << "," << N << ","
         << M << "," << c.C_L << "," << c.C_D << "," << circulation << "," <<
        duration.count() / 1000 << endl;
49
50      // Compute analytic stream function
51      vector<vector<node>> analytic_mesh(N, vector<node>(M));
52      buildMesh(analytic_mesh, p);                // creating the mesh
53      computeAnalyticStream(analytic_mesh, p); // stream solver
54      calculateVelocity(mesh, p);                 // calculating velocity
55      calculateCp(mesh, p);                       // calculating pressure
        coeficient distribution
56      calculatePressureTemperature(mesh, p);    // calculating pressure and
        temperature
57      name = p.folder + p.mesh_number + "_analytic_output.csv";
58      exportData(analytic_mesh, name); // export data to file output.dat
59
60      vector<vector<node>> error_mesh = analyticError(mesh, analytic_mesh); //
         calculating error
61      name = p.folder + p.mesh_number + "_error_output.csv";
62      exportData(error_mesh, name);
63
64      // Print final results
65      cout << "### POTENTIAL FLOW RESULTS ###" << endl;
66      cout << "C_L = " << c.C_L << endl;
67      cout << "C_D = " << c.C_D << endl;
68      cout << "Cylinder circulation = " << circulation << endl;
```

```
69      cout << "Equivalent rotation speed = " << circulation / (2 * M_PI * p.
        cylinder_r * p.cylinder_r) << endl;
70      cout << "Computation time = " << duration.count() / 1000 << "ms" << endl
        ;
71      return 0;
72  }
```