Numerical Project

Planar Potential Flow

Cyril Tsilefski

Supervisor :
Vincent Ballenegger

Program made in Python - End of project 12-11-2020

Contents

Part I

Functional requirement of the program

1 The project

The goal of this project is to simulate the movement of a fluid through different geometries. The program creates a box of a choosen size, builds a geometry inside it and simulates the movement of a given fluid.

2 Files

In order to increase readability, the project is made of several files. I made the choice to work with Object Oriented Programming.

- main.py: This file calls for the needed functions/class
- matrices.py: This file contains the class "Matrices", it builds the geometry, the different matrices to plot and stores them
- plot.py: This file plots the matrices built in "matrices.py"
- parameters.py: This file contains all the variables that can be changed by the user
- data_check.py: This file checks the variables and makes sure that the program will run

3 Data

This project uses several piece of data set by the user to work.

- `nx: uint` and `ny: uint`, size of the domain
- `h: uint`, size of a cell
- `geometry: str`, choosen geometry
- `angle:` uint, angle of the widening/shrinkage geometry
- `vx: float`, initial fluid speed (Neuman's condition)
- `phi_ref: float`, reference velocity potential (Dirichlet's condition)
- `rho: float`, relative density of the fluid
- `pressure_init: float`, initial pressure of the fluid
- `recompute: bool`, if `True`, the program will check for existing dat files, if it finds them, it will not re-run the computations

Be careful in the case of a widening/shrinkage geometry! In order for the program to generate a domain from one end to another, there is a restriction on the angle, if the restriction is not met, the program will output a ValueError. The restriction is as follows:

$$|angle| < \arctan\left(\frac{0.5 \times N_y - 1}{N_x}\right)$$

The angle parameter should be set in degree, the program will convert it to radians for the computation.

4 Outputs

As of the alpha version, the program outputs 4 pdf files, one for each plot. The files are saved in a subfolder named **figures/** and the filenames are set with the following rule:

`<data>_<geometry>_Nx=<nx>_Ny=<ny>.pdf`

`data` stands for the plotted data (potential, velocity, streamlines, pressure).

As of the beta version, the program outputs more files depending on the choosen geometry in the same subfolder as the alpha version. They are named with the following rule:

`section_<geometry>_<nx>_<ny>(_<n>).pdf`

The `_<n>` is there when there are multiple sections plotted for the same geometry, it is not present for the widening and shrinkage geometries.

5 Concerning the running time

Due to the function `numpy.linalg.solve()` being slow for big matrices, the bigger the size of the domain, the higher the running time.

For a domain size of 3600 cells (60×60), it takes around 20 seconds to run, for a domain size of 14 400 cells (120×120), it increases to 23 minuts.

I searched for a faster method to solve the linear system in vain, thus I recommend to stay on relatively low values for N_x and N_y, the graphs are easily readable for a value of 60 each.

By default, python only uses one CPU core when executing a program, the solution to that problem is parallel programming, however I am not experienced enough in that field to develop an optimized program.

Part II

Internal structure of the program

6 Physical model

In order to build the model, the program uses a squared structured mesh model (matrix). The values are computed at each point of the matrix.

6.1 Velocity potential field

The velocity potential field is computed by solving a system of linear equations. The result is computed using the matrix method, by solving the equation $Ax = b$ we can find the velocity potential flow value in each cell.

The matrix A, of size $N_{\text{fluid}} \times N_{\text{fluid}}$ stores 2 information per cell, the number of neighboring cells and the label of each one of them. The number of neighboring cells is stored in the diagonal of the matrix. Let c be the number of a cell containing fluid and n_c the number of neighbors for the cell n, the diagonal elements are defined as follows:

$$A_{n,n} = -n_c$$

Let i be the set of fluid cells neighboring n, we then have:

$$A_{n,i} = 1$$

There are at most 4 neighbors for each fluid cell.

6.2 Velocity field

Knowing the potential in all cells of the domain, the velocity can be computed by calculating the gradient of the potential. The method used is the finite difference:

$$v_x(i,j) = -\frac{\phi_{i+1,j} - \phi_{i-1,j}}{2h}$$
$$v_y(i,j) = -\frac{\phi_{i,j+1} - \phi_{i,j-1}}{2h}$$

However, this method only works if the cell (i, j) is surrounded by fluid cells. For the cells at the edge of the domain, the method used is slightly different. As mentionned in the scope statement[1], if the cell $(i + 1, j)$ is not a fluid cell, we consider it as a fictitious cell with the value of the cell (i, j). The same rule applies for each of the neighboring cells that are not fluid cells.

6.3 Streamlines

In the alpha version, the streamlines are computed using matplotlib.

In the beta version, the method of Euler is used. However, I did not manage to complete that part of the version.

6.4 Pressure field

In order to compute the pressure, we need to know several things. Firstly, the velocity field must be know at each cell of the domain. Then we need the initial pressure of the domain. Moreover, there are certain conditions to meet regarding the system:

- the fluid is ideal
- the fluid is incompressible
- the fluid is irrotationnal

Knowing all that, we can use Bernoulli's Theorem:

$$\mathcal{H} = P + \frac{1}{2}\rho v^2 \quad [2]$$

$\mathcal{H}$ stays the same on each point of the domain. Using this constant, we can compute the pressure at each point of the domain.

7 Constitutive elements - alpha version

7.1 main.py

This file is executable (without parameters input in the command).

After the imports, it will first call the file `check_data.py` to check the type of the input data.

Then it will call the file `matrices.py` to initiate the values of `G`, `M`, `cell_coords`, `A` and `b`.

It then checks for the `recompute` value, if `True` or if there are no existing **dat** files for the current parameters, it will call the function `domain_check()` to display a warning if the domain is large. It then calls the subroutine `make_data()` to compute `phi`, grad_x, grad_y, `pressure` and `pressure` and saves them in **dat** files.

Else, it will use the existing dat files to generate the plots.

Then it calls the function `load_data()` to read the **dat** files and stores them in a dictionary and it initiates the file `plot.py` which will be used to plot the different values.

The last lines call the subroutine `plot_graphs` with different arguments to plot and save all the wanted graphs.

7.2 data_check.py

This file has several subroutines and functions to rule the execution of the program.

7.2.1 directory_check() (beta version)

The subroutine `directory_check()` will check for the folders to store the data, if they don't exist, it creates them.

7.2.2 data_check()

The subroutine `data_check()` reads the value of each parameter and checks that the type and the value are correct and will not cause a crash of the program. If the value/type is incorrect, it will raise an error and stop the program.

7.2.3 existing_data()

The function `existing_data()` will check for specific files in the **dat/** subfolder. If at least one files is missing, it returns `True` and the program will recompute all the data. Else it does nothing.

7.2.4 domain_check()

The subroutine `domain_check()` is just a warning. It read the max value of the matrix `M` plus one which is the number of fluid cells to compute. If this number is higher than 5000, it will display a message warning the user that the program can take some time to run. Then it asks if the user wants to keep going. `"Yes"` will continue, `"No"` will stop the program and anything other than that will stop the program aswell.

7.3 matrices.py

This class is where everything is computed, from start to end.

7.3.1 __init__()

`__init__()` stores the value of `G`, `M`, `cell_coords`, `b` and `A` which will be used several times in the class.

7.3.2 make_data()

The subroutine `make_data{}` will call the functions that compute `phi`, `grax_x`, grad_y, `pressure` and `pressure`. Each function will write the data in a **dat** file.

7.3.3 load_data()

The function `load_data()` reads the **dat** files and return the values in a dictionary.

7.3.4 build_g()

The function build_g() takes 4 input arguments.

- Nx: int
- Ny: int
- geometry: str
- angle: int of size 8 bits, in radians

It acts as a selector. Given a geometry, it will create a matrix G filled with zeros and call the function build_geometry() with different input values. There is no real point in this function in the alpha version since the straight, widening and shrinkage geometries are generated from the same function, with only the angle changing. However, it will prove useful in the beta and gold version.

7.3.5 build_geometry()

The function build_geometry() takes 2 input arguments.

- G: np.array
- angle: int of size 8 bits, in radians

The function returns the matrix G.

The function starts by computing alpha as the tangent of the angle.

It then uses it in a for loop to compute an offset which will be used to build the geometry.

In case of a shrinkage geometry, the angle inputed is negative, the function will flip horizontally the matrix.

It then sets the inlet and outlet of the domain.

Since the program will always compute G no matter what and to avoid some problem, it will save the matrix only if recompute == True.

7.3.6 build_index_matrices()

The function build_index_matrices() takes 2 input arguments.

- Nx: int
- Ny: int

The function returns the matrix M and the array cell_coords.

It starts by creating the matrix M the shape of G, filled with zeros and initiate a counter count. It then parses M and for each fluid cell (G[c, r] != 0), it sets the value of M[r, c] to the value of the counter and increment the counter by 1. All the other values are set to -1.

Then it create an array cell_coords of size (M.max() + 1, 2) and sets the value of the counter back to zero. Then it parses the matrix M and for each cell with a value different from -1, it will store the coordinates of that cell in the index count of the array cell_coords and increment the counter.

7.3.7 build_a()

The function build_a() takes no input, it uses the imported data from parameters.py.

The function returns the matrix A.

It starts by creating an array A of size $(N_{\text{fluid}}, N_{\text{fluid}})$, filled with zeros. It then parses the array cell_coords and for each inlet and fluid cell (the oulet is not counted there), it counts the neighbors, stores their coordinates, set the value of the diagonal depending on the neighbors and sets each neighbor associated cell to 1. The diagonal cells corresponding to the outlet are all set to 1.

7.3.8 build_b()

The function build_b() takes no input, it uses the imported data from parameters.py.

It returns the array b.

It starts by creating an array b of size (cell_coords.shape[0], 1), filled with zeros.

It then parses the array. For each cell corresponding to the inlet, it sets the value to -vx * h and for each value corresponding to the outlet, it sets the value to phi_ref. The rest of the array is not modified.

7.3.9 build_phi()

The subroutine build_phi() takes no input, it uses the imported data from parameters.py.

This function is the slowest one since it compute the solution of $Ax = b$. For that it uses the function linalg.solve() from numpy. Then it creates an empty matrix phi and fills it with numpy.nan. It then parses x and sets the value of each phi cell corresponding to the x value. The other cells are not modified.

Finally, it saves the values in a **dat** file.

7.3.10 build_gradient()

The subroutine build_gradient() takes no input, it uses the imported data from parameters.py.

This function is rather big, but it is only a variation of the gradient function from numpy. The function numpy.gradient does not work the way I want with the numpy.nan values so I made my own.

It starts by loading the **dat** file containing the values of phi and creates 2 empty matrices grad_x and grad_y the same shape as G and fills them with numpy.nan. It the parse the cell_coords array and sets the values of grad_x and grad_y following the centered difference with step $2h$ mentioned in the scope statement [1]. I then found that the function numpy.gradient uses the same method. However, since the borders of the matrix phi is filled with numpy.nan values, the gradient is not computed at the outer edge of the domain.

To compute there values, I applied the impermeable wall condition specified in the section **5.1** of the scope statement. We consider the numpy.nan cell with coordinates [i - 1, j] as a fictitious cell with the value of the cell [i, j]. This rule applies to the cells to the left and to the right for grad_x and to the top and to the right for grad_y.

It then computes the norm of the gradient (used later to plot the normalized gradient).

Finally, it saves the 3 matrices in **dat** files.

7.3.11 build_pressure()

The subroutine build_pressure() takes no input, it uses the imported data from parameters.py.

The function starts by loading the **dat** files containing the values of the norm of the gradient.

It then creates an empty matrix pressure and fills it with numpy.nan. Then, it computes the Bernoulli's constant $\mathcal{H}$ using the initial values.

It then parses the array cell_coords, sets the inlet cells of the pressure matrix to pressure_init, computes and sets the values for each cell.

Finally, it saves the matrix in a **dat** file.

7.4 plot.py

The class is used to plot every piece of data.

7.4.1 plot_graphs()

The subroutine plot_graphs() takes 2 input arguments.

- display: str
- data: dict

This subroutine creates the figure, shows the matrix `G` (without interpolation) and disables the axes as they are not needed in that kind of plot. Then, depending on the `display` value, it will call other functions to plot the graphs and save them as **pdf** files.

7.4.2 Data specific plotting functions

Each one of the 4 next functions take 1 input argument.

- `data: dict`

They return the plotted graph.

The 4 functions work the same way, they set the name of the graph and plot the desired data.

8 Constitutive elements - beta version

In this section, I will detail the changes between the alpha and the beta version.

8.1 `matrices.py`

8.1.1 `build_elbow()`

The function `build_elbow()` takes 1 input argument.

- `G: np.array`

The function returns the matrix `G`.

In terms of geometry, an elbow is simply composed of 2 rectangles, it is easily drawed using slices. The functions computes the middle value of `nx` and `ny`, it takes an area around each value equal to $1/5$ of `nx` and `ny` respectively. It then draws 2 rectangles using slices from these values.

8.1.2 `build_obstacle()`

The function `build_elbow()` takes 1 input argument.

- `G: np.array`

The function returns the matrix `G`.

The choosen obstacle is a disc. The function starts by drawing a rectangle of width $4/5$ of `ny` and length `nx`. The radius of the disc is $1/5$ of the smallest side of the rectangle. It then checks for each cell of the domain and computes their distances to the center of the rectangle. If the distance is smaller than the radius, the cell is assigned the value 0, else the loop passes to the next cell.

8.2 `plot.py`

8.2.1 `interp_nan()`

The function `interp_nan()` takes 5 input arguments.

- `x: np.array`
- `y: np.array`
- `xnew: np.array`
- `xnew: np.array`
- `grad: np.array`

The function returns the matrix `grad_a_new`.

This function interpolates the `array grad` to a new size.

`x` and `y` are `linspaces` the size of the domain with the initial step. `xnew` and `ynew` are `linspaces` the size of the domain with the new step.

It calls the function `RectBivariateSpline` from `scipy` to interpolate the array. However, this function does not behave the way we want when the array contains `np.nan` values. I found a workaround[3] by saving the `np.nan` values using the same function.

Basically, it stores the `np.nan` values as 1 in a new zeros array to keep track of their positions. Then it changes the `np.nan` to zeros in the initial array and interpolates it. It then proceeds to interpolates the array used to save the `np.nan`.

On an array filled with zeros and ones, the scipy function does not modify a lot the values, but it still resizes the array correctly. We then checks for values above 0.5 which correspond to the 1 before the interpolation and replace them with `np.nan` in the interpolated version of the initial array.

8.2.2 `section`

The subroutine `section()` takes 2 input arguments.

- `data`: `dict`
- `geometry`: `str`

The subroutine calls different functions depending on the geometry used.

8.2.3 `section_wid_shrin()`

The subroutine `section_wid_shrin()` takes 2 input arguments.

- `grad_x`: np.array
- `grad_y`: np.array

The subroutine plots the velocity profile of a straight section (from left to right) at the center of the domain for the widening and shrinkage geometries.

We start by defining a linspace (the x value of the plot), we then define 2 zeros arrays, one for the x orientation and one for the y orientation. Then for each element of the arrays, we set the value to the corresponding ones in the grad_x and grad_y arrays.

We then plot the 2 lists of data with some information and save them.

8.2.4 `section_elbow()`

The subroutine `section_elbow()` takes 5 input arguments.

- `grad_x`: np.array
- `grad_y`: np.array
- `xpart`: int
- `ypart`: int
- `n`: int

The subroutine plots the velocity profile of a section section crossing the elbow geometry.

We start by defining 2 zeros arrays of size $xpart + ypart$ (see subsection 12.6 for examples) and a linspace. We then parse the first half of the domain and set each element of the arrays to the corresponding ones in the grad_x and grad_y arrays.

We then parse the second half of the domain, it works the same a the first half but the value of the arrays is set with an offset (we don't want to erase the values of the first half) and the indices in grad_x and grad_y are inverted.

We then plot the 2 lists of data with some information and save them.

8.2.5 `section_obstacle()`

The subroutine `section_obstacle()` takes 4 input arguments.

- `grad_x`: np.array
- `grad_y`: np.array
- `xpart`: int
- `n`: int

The subroutine plots the velocity profile of a section section crossing the obstacle geometry.

We start by defining 2 zeros arrays of size nx (see subsection 12.6 for examples) and a linspace. We then parse the domain until we reach a `np.nan` value. The program will start a loop and each time it encounters a `np.nan` cell, it will check the cell above until it finds a cell with a `float` value.

We then plot the 2 lists of data with some information and save them.

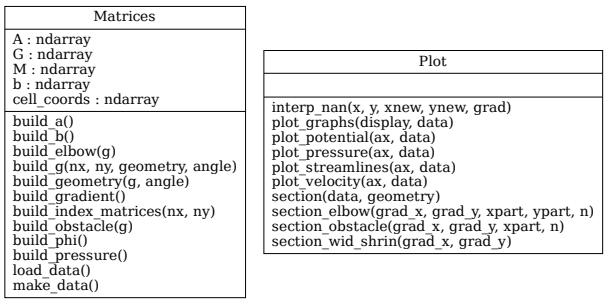
Part III

Optimization

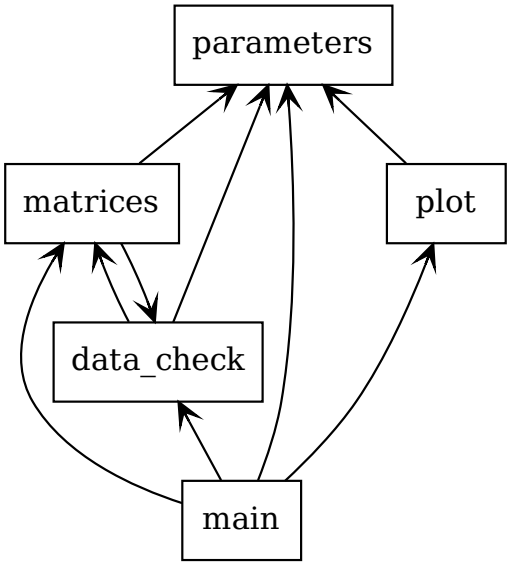
In order to reduce the memory usage of the program, I tried to limit as much as possible the size allocated to each array by setting the datatype at declaration.

At runtime, the memory usage stays relatively low since the program does not have to store anything big. However, if you set a domain ludicrously big, the program will crash telling you there is not enough memory to store the data since the matrix A increases exponentially with the size of the domain.
Concerning the CPU, I tried running the program on a really big domain to see how it reacts. The CPU usage rises to the maximum on one core (since python natively uses only one core) until the end of the `linalg.solve` computation. Aside from this function, the CPU usage stays low at runtime.

9 Informational diagrams



(a) Classes diagram



(b) Diagram linking the imports

Part IV

Outcome examples

10 Input data

The program needs some data to run. For the example, I found that a box of size 60×60 is good enough. Obviously we can use a bigger box, but the computing time increases exponentially as we increase the box size. A smaller box can also produce acceptable results. I also choosed to use cells of size 3 ($h = 3$) which plots better velocity fields for a domain of this size.

On the velocity potential field, the colors show the differences thoughout the domain.

On the velocity plots, the colors correspond to the norm of the velocity vector.

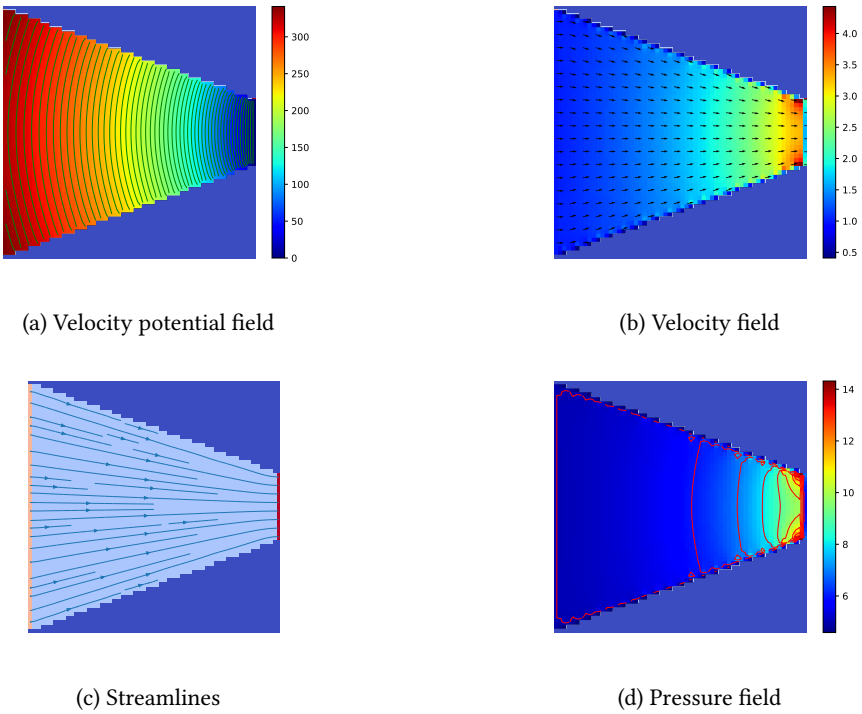
On the velocity field, the colors correspond to the pressure inside the domain.

Concerning the fluid, here are the data I choosed :

- `rho` = 1, relative density of water
- `vx` = $1\,\mathrm{m\,s^{-1}}$, typical speed in a watering pipe
- `phi_ref` = 0
- `pressure_init` = 5 bar, typical pressure in a watering pipe

The geometry in the output section is the shrinkage geometry, the others are in the appendices Part VI.

11 Output



(a) Velocity potential field

(b) Velocity field

(c) Streamlines

(d) Pressure field

Figure 2: Results for `Nx` = `Ny` = 60 in a shrinkage geometry

12 Interpretation

12.1 Expectations

The expected behavior for this kind of geometry (bottleneck) is that the fluid (water in this case) will bump into the boundaries of the domain and converge toward the center as it crosses the domain. Regarding the pressure of the fluid, since the number of fluid particles is constant throughout the simulation, we should observe a rise of pressure as the domain narrows.

12.2 Observations

Firstly, looking at Figure 2b and Figure 2c, the water crosses the domain correctly and converges toward the center of the domain. Despite some accuracy problems (the velocity on the edges seems off), I can say that the simulation meets the expected behavior. It is mandatory to normalize these vectors, else the output graph is not readable. However, the bigger the domain, the bigger the number of vectors. As the size of the domain is 60×60, there are quite a lot of vectors to show. With the default display, there are too much arrows on the graph and they are too small to analyze anything, I added a small algorithm using interpolation to reduce the number of arrows while keeping the data as accurate as possible. In the alpha version, the streamlines are computed using the velocity, there should not be any problem with them, they show the path of the fluid and it seems to be the right one.

On the Figure 2d, it is clear that the pressure increases as the fluid travels the domain. It is once again the expected behavior.

12.3 Elbow

On the elbow geometry, the flow of water is following the angle in the domain.

Looking at the velocity field plot Figure 8b, the fluid is faster near the inner angle and slower near the outer angle. It is a satisfying result since around the inner angle, there are more particles, increasing the flow and around the outer angle, there are less particles, decreasing the flow.

On the pressure field Figure 8d, the observations of the velocity field are reinforced. There is more pressure at the inner angle, where there are more particles, explaining the increase in speed. On the other hand, the pressure at the outer angle is lower because there are less particles.

12.4 Obstacle

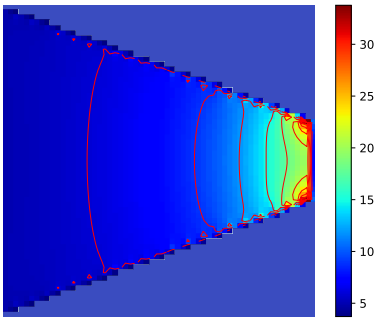
The choosen obstacle for this project is the disc. The flow of water is correctly going around the obstacle.

Looking at the velocity field Figure 9b, the water is slowing down at the lateral edges of the disc and speeding up at the vertical edges of the disc.

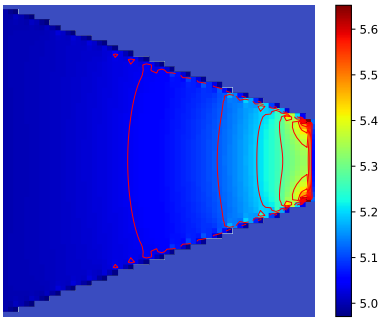
On the pressure field Figure 9d, we see an increase in pressure where there are more particles (ie: when the path narrows down) and a decrease in pressure where there are less particles.

12.5 Pressure verifications

I ran some tests regarding the pressure field when changing the relative density of the fluid and the results are satisfying.



(a) Bromine at 0 °C, $\rho_{\text{Br}} = 3.087$



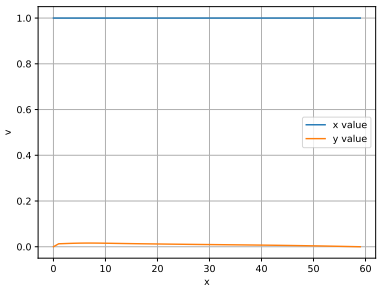
(b) Liquid hydrogen at -252 °C, $\rho_H = 0.07$

Figure 3: Comparison of the pressure field for bromine and liquid hydrogen

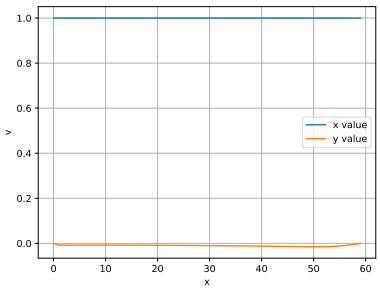
According to the Bernoulli's Theorem, when you have 2 different fluids in the same conditions, the heavier the fluid, the higher the pressure. In this example, the bromine is heavier than the liquid hydrogen, and the graphs are showing that property really well. With the same initial pressure and the same conditions, the pressure of the bromine rises higher than the one of the liquid hydrogen. The isobaric lines are also slightly different.

12.6 Sections

The velocity profiles show in this section are all plotted with $Nx = Ny = 60$ and $h = 3$. The value used to plot the sections are not interpolated, although a quick workaround using the `interp_nan` function from `plot.py` is possible to increase the number of points. All the computations are made with normalized vectors, I found the plots are easier to read that way.

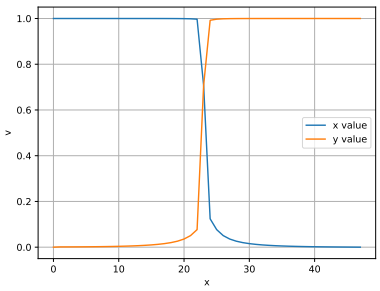


(a) Velocity profile for the widening geometry

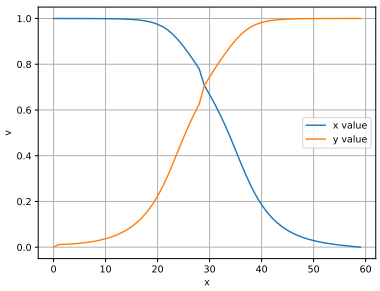


(b) Velocity profile for the shrinkage geometry

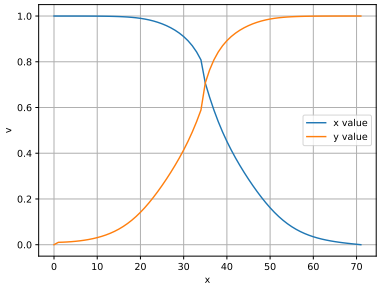
There are no really interesting velocity profiles for the widening and shrinkage geometries since the velocity basically stays the same. It quickly changes at the start of the domain and then keeps going with the same vectors.



(a) Section 1, close to the inner angle



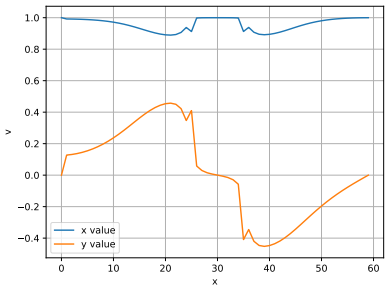
(b) Section 2, close to the center



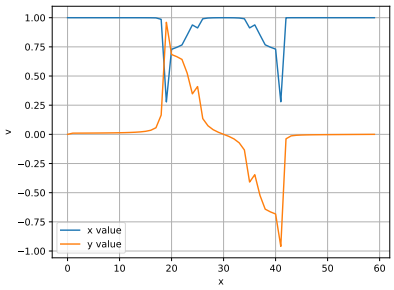
(c) Section 3, close to the outer angle

Figure 5: Velocity profile for the elbow geometry

Concerning the elbow geometry, the 3 sections are alike. On the first section (around the inner angle), the velocity quickly changes, however on the 2 other sections, the changes in velocity are a bit smaller, it corresponds to the observation that we made on the velocity plot.



(a) Section 1, starting on a line parallel to the upper edge of the obstacle



(b) Section 2, starting on the line in the middle of the domain

Figure 6: Velocity profile for the obstacle geometry

I find the obstacle geometry to be the most interesting of all. On both sections of the velocity plot, we can clearly see the variation on the x and y orientations. despite some spikes that I failed to remove, the plots are really accurate and satisfying.

Part V

Conclusion

The simulation of fluid mechanics is an interesting domain of numerical simulations. Overhaul I am satisfied with the simulations, in my opinion, they are a relatively accurate reflexion of real cases of fluid mechanics. This project was really interesting, however, there are several addition that would make the project even more accurate. Adding the fluid friction and the viscosity of the fluid would make it more ”real”. Plus with fluid frictions, the straight geometry becomes interesting to study.

References

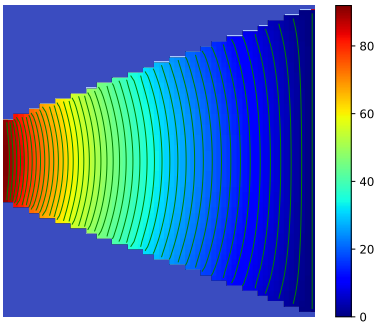
[1] Vincent Ballenegger. *Planar potential flows*. 2020.

[2] Wikipedia. *Bernoulli's Principle*. Apr. 11, 2020. URL: https://en.wikipedia.org/wiki/Bernoulli%27s_principle.

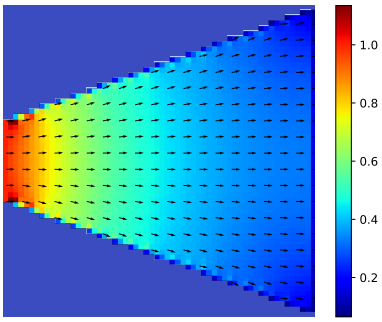
[3] StackOverflow - xdze2. *2d interpolation with NaN values in python*. July 24, 2020. URL: https://stackoverflow.com/questions/51474792/2d-interpolation-with-nan-values-in-python.

Part VI

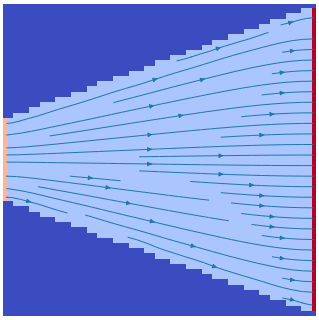
Appendices



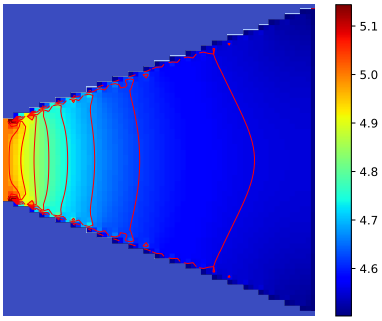
(a) Velocity potential field



(b) Velocity field

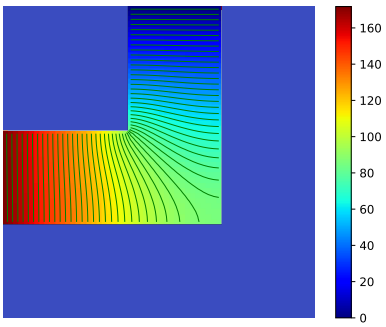


(c) Streamlines

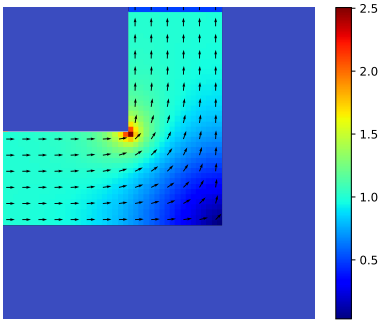


(d) Pressure field

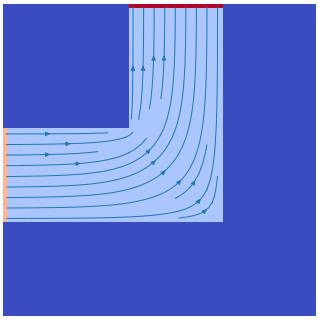
Figure 7: Results for `Nx` = `Ny` = 60 in a widening geometry



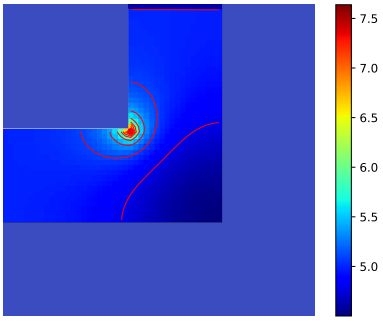
(a) Velocity potential field



(b) Velocity field

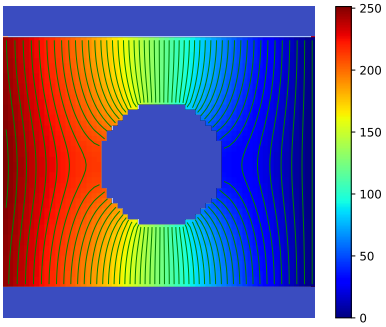


(c) Streamlines

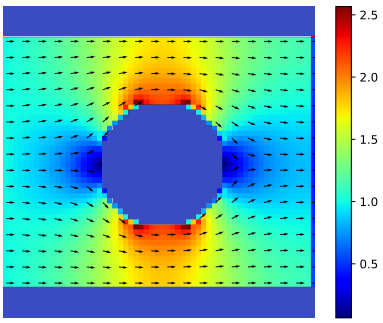


(d) Pressure field

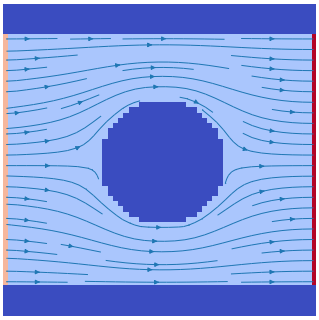
Figure 8: Results for `Nx` = `Ny` = 60 in an elbow geometry



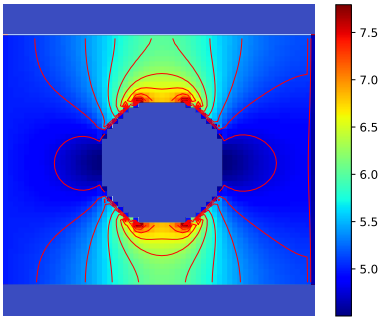
(a) Velocity potential field



(b) Velocity field

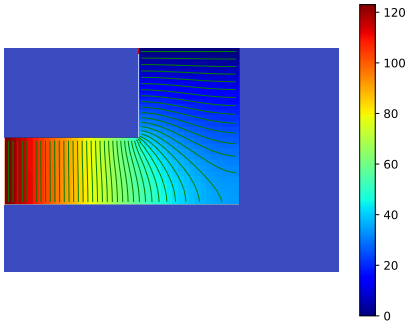


(c) Streamlines

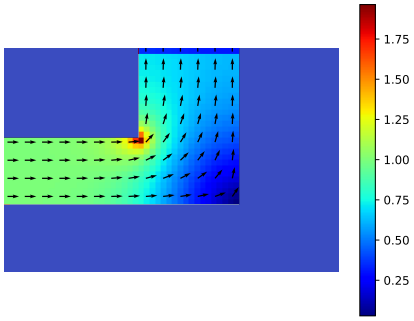


(d) Pressure field

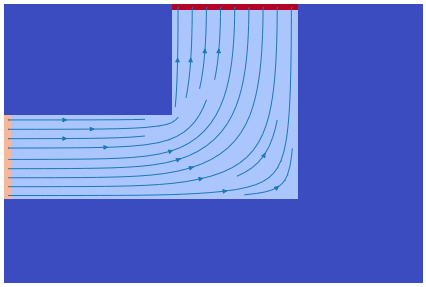
Figure 9: Results for `Nx` = `Ny` = 60 in an obstacle geometry



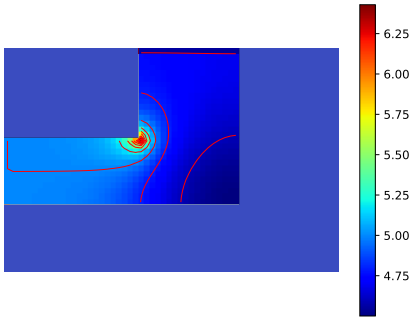
(a) Velocity potential field



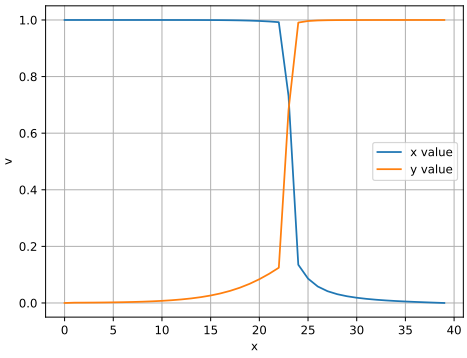
(b) Velocity field



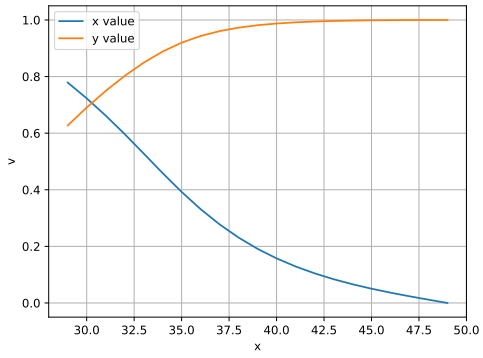
(c) Streamlines



(d) Pressure field

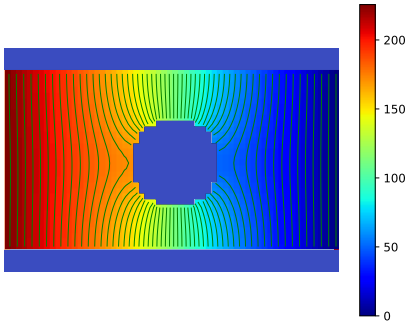


(e) Section 1

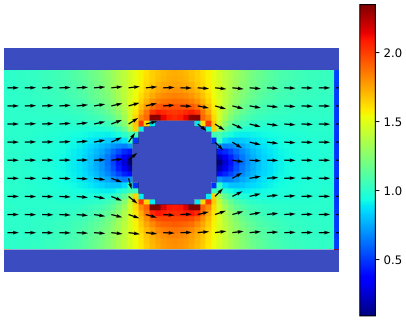


(f) Section 2

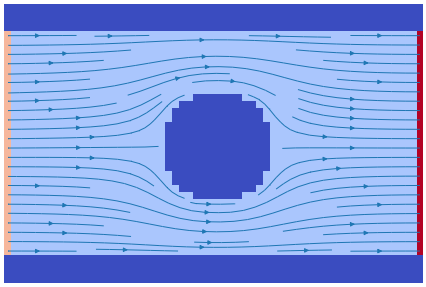
Figure 10: Results for `Nx` = 60 and `Ny` = 40 in an elbow geometry



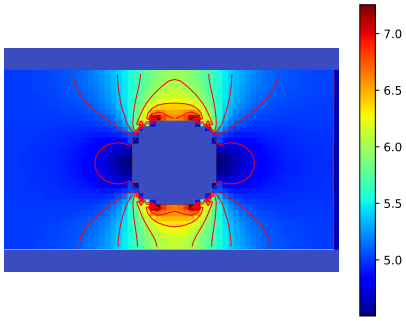
(a) Velocity potential field



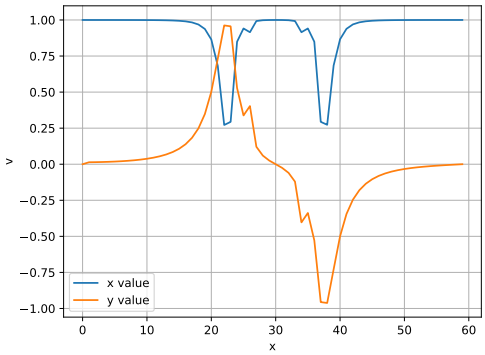
(b) Velocity field



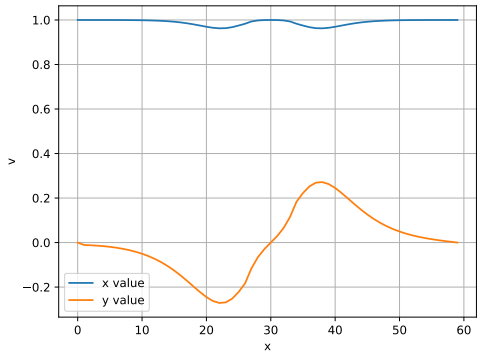
(c) Streamlines



(d) Pressure field



(e) Section 1



(f) Section 2

Figure 11: Results for `Nx` = 60 and `Ny` = 40 in an obstacle geometry