SHAPE OPTIMISATION OF A WINGTIP FOR DRAG REDUCTION

ST7 High Performance Computing for footprint reduction

ONERA Integrated Project

BERRADA Tariq

GARIBALDI Alessia

PIERRE Marc

MERCAT Flora

CORNILLOT Antoine

ONERA
THE FRENCH AEROSPACE LAB

CentraleSupélec

Table des matières

I. Introduction

During the ST7 “High Performance Computing for footprint reduction”, we learnt about the differences between high-performance computers and normal ones and about the necessity to develop specific algorithms for these specific computers. Juliet Ryan, working at the ONERA, the French aerospace agency, suggested an application project in the field of aerodynamics. Our group of 5 students, composed of IDRISSI Tariq, GARIBALDI Alessia, PIERRE Marc, MERCAT Flora, and CORNILLOT Antoine, worked on this project of “Shape optimisation of a wingtip for the reduction of the drag”. In this project, we have focused ourselves on the resolution of Partial Derivative Equations using the python module FEniCS and using the supercomputer based on the campus of Metz “Machine du Grand-Est”.

This document is composed out six parts. The first one is a general presentation of the program we made. We will explicit there the physical, mathematical and programming problem we were trying to solve.

We will then focus on what is expected from our program, that is to say what kind of information we would like to have.

The third part is about the architecture of the solution we provide. We will describe there the functions we developed.

Finally, we will discuss about the tests we made during all the development of the program in order to be sure of our solution and will also show these final results.

To conclude with, we will end up with the possible improvements of the project and open up on its applications.

II. Presentation of the problem

Our problem is to minimize the drag that applies on a wingtip. We model the situation in a 2-dimensional plane, having the wingtip represented by a 2D obstacle. We study the 2-dimensional flow around this obstacle. A flow is described by its pressure and its speed.

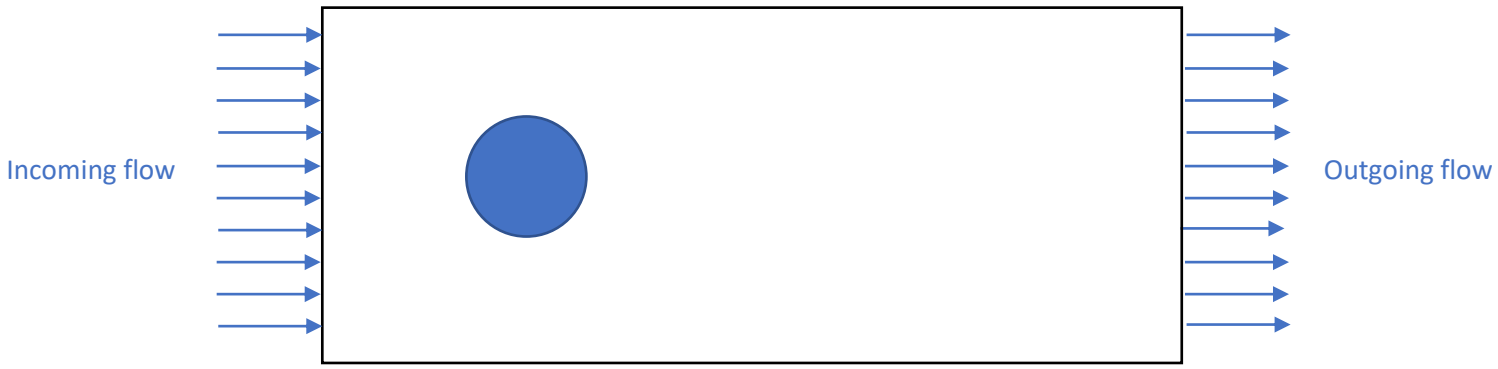


Figure 1: Schematic view of the problem

We decide to model the flow as a 2-dimensional Navier-Stockes flow. Navier-Stockes equations are known under the form:

$$\rho\left(\frac{du}{dt} + u.\nabla \mathrm{u}\right) = \nabla\sigma(\mathrm{u},\mathrm{p}) + \mathrm{f}$$

$$\nabla \mathrm{u} = 0$$

- As a volume force, in our problem we have f=0.
- $\sigma(\mathrm{u},\mathrm{p})$ is the stress tensor. $\sigma(\mathrm{u},\mathrm{p}) = 2\mu\varepsilon(u) - pI$
- $\varepsilon(u)$ is the stain rate tensor, $\varepsilon(u) = \frac{1}{2}(\nabla \mathrm{u} + (\nabla u)^T)$

We also study the stationary problem, so $\frac{du}{dt} = 0$. However, this formulation does not make it possible to solve the problem numerically. We are therefore interested in the following variational formulation:

$$\int[(\nabla \mathrm{u}, \nabla \mathrm{v}) + \mathrm{p} * \mathrm{div}\,\mathrm{v} - (\mathrm{div}\,\mathrm{u})\mathrm{q}]\mathrm{dx} = \int(f, v)dx = 0$$

This variational form can be solved using a python module called FEniCS.

The drag applied on the obstacle is proportional to the objective function $\mathrm{J}(\mathrm{u}) = \frac{1}{2}\int(\mathrm{grad}\,\mathrm{u}, \mathrm{grad}\,\mathrm{u})\mathrm{dx}$ (integration on the contours of the obstacles).

We then want to solve our physical problem by implementing a program that will be able to modify the mesh to optimize the shape of the obstacle, that is to say to minimize J. According to the objectives of our course, we will try to find a parallelizable program solving this optimisation problem.

III. Expectations over the solution

We are being asked to compute a program that optimises the form of the obstacle in order to minimise the drag $J(u)$ with $\partial\Omega$ being the obstacle frontier.

The program will iterate over a definite number of steps. At each step, the user wants to know what is the form of the wingtip and why it is an amelioration.

Formally speaking, this equivalent to finding $argmin_W J(u)$.

In order to do that, we implemented a simple gradient descent algorithm, which will move the vertices of the mesh in the direction of the gradient field while subject to the constraint that the surface of the wingtip must stay constant.

Let a be a certain vertex on the mesh, at each iteration it's coordinates are moved as follows:

$$a_x = a_x + \rho.(\nabla u.\nabla u).n_x$$

$$a_y = a_y + \rho.(\nabla u.\nabla u).n_y$$

Subsequently, each vertex of the mesh is moved in the opposite direction to its normal until the original surface area of the wingtip is recovered.

While surface(mesh) != surface(mesh_initial):

$$a_x = a_x - \varepsilon.n_x$$

$$a_y = a_y - \varepsilon.n_y$$

This is then the basic idea behind our optimization process, we simply repeat this process for as many iterations as needed in order to find an optimized solution.

For the variables ρ and ϵ, they were chosen as constants verifying $\varepsilon < 3.\rho$ in order to assure that the mesh surface will not diverge.

We tried experimenting with dynamic parameters that are proportional to the gradient of the loss function J(u), but the results obtained were not as good because of the amplitude of the variations of ∇J, which would vary from 0.9 to 10^{-8} in a few iterations, making the algorithm in this case converge before finding a local minima.

Our program will then provide information concerning the mesh, the field of speed u, the field or pressure p, and J(u) = $< \nabla u, \nabla u >$ at each step.

Our objective is also to compare the efficacity of High-Performance Computing and sequential codes. We should interest ourselves in the execution time of our codes.

IV. Conception

1. Global architecture

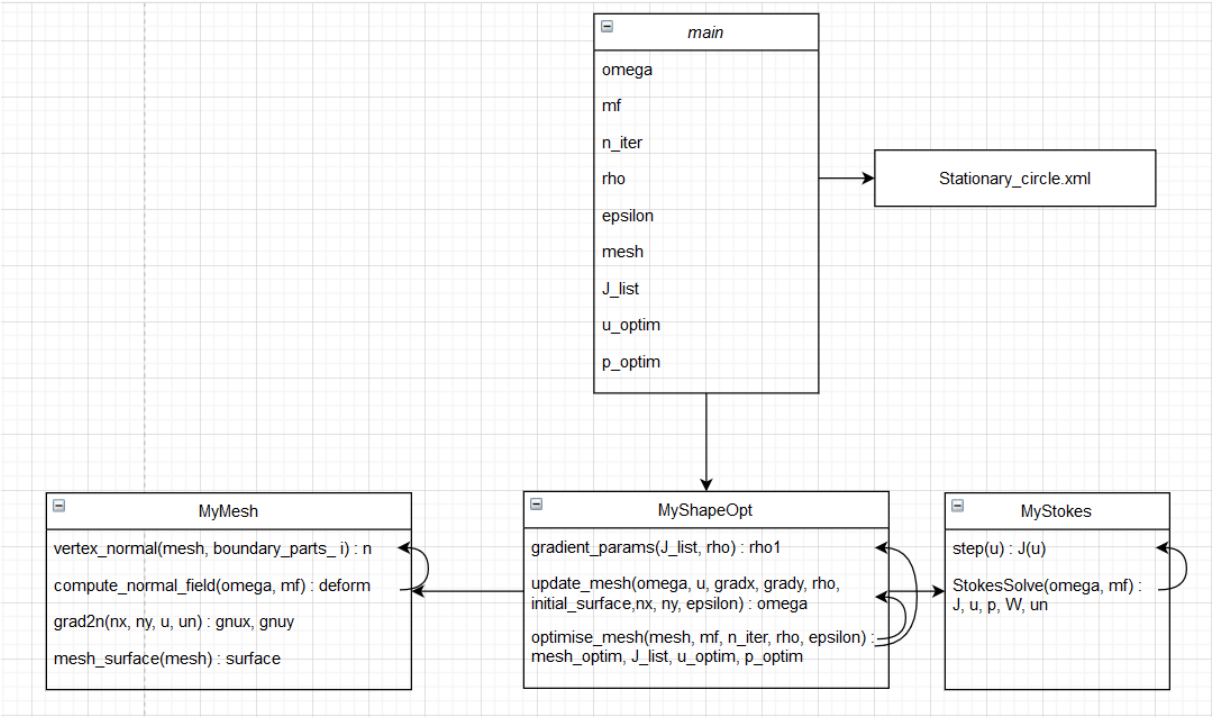


Figure 2 : Global Architecture of the program

Our code is composed of 4 python files:

- A main.py file which takes data from stationary_circle.xml and optimise the mesh using optimise_mesh from MyShapeOpt
- MyShapeOpt.py has all the functions in order to optimise the mesh after a certain number of iterations. These functions calls function from MyStokes and MyMesh.
- MyMesh.py contains all the functions that gives us information about a given mesh : its normal field, its gradient field, the surface of the mesh.
- MyStokes.py solve the Navier-Stokes equation over a given mesh.

Meshing the situation

We have decided to use Finite Element Method in order to solve the variational problem. To implement this method, we need to define a mesh of the space.

A description of the space has first been implemented on a .geo file. A circle of radius 1 and center (0,0) is drawn inside a rectangle of size [-4,30]x[-6,6].

From this description of the situation, we implement a mesh using the program gmesh. We convert this .msh file into a .xml file more compatible with FEniCS named "stationary_circle.xml".

Solving the Poisson equation over the mesh

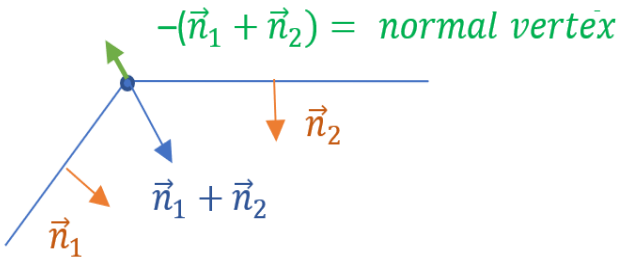
Having defined the mesh of the space, we are now interested in solving the variational prolem over it.

To do this, we created the StokesSolve function, which takes as an argument the omega mesh and the regions belonging to the obstacle, and which outputs the fields of speed and pressure, as well as the objective function J.

To do this, we first define the set of functions to which the functions u, p, v and q defined above belong, that is to say the polynomials of degree 2 by pieces for u and v, of degree 1 for p and q. v and q are test functions, so they are equal to zero on the edges. We impose the Dirichlet conditions on the edges for u : a positive flux according to the first component on the left face, a zero flux on the obstacle and zero on the lateral edges. Thanks to this and the variational equation, FEniCS is able to do the resolution.

Creating a normal field over the mesh

The optimisation problem is solved using a gradient descent. In order to compute the drag and to know for each of the vertices of the obstacles in which direction it has to be displaced, we need the normal field over the mesh.



3. Figure: direction of normal_vertex

Let's first define the normal of a vertex on the boundary. As it is naturally easy to understand the normal of a facet (in 2D : a link between two vertices), the FEniCS-object "facet" has the method normal(), which returns its normal. We then understand the normal of a vertex on the boundary as the sum of the normal of the facet adjacent to the vertex and being on the boundary. In the function vertex_normal, we added a normalization constant for a more refined modification of the mesh.

Then the function **compute_normal_field** computes the normal field over the mesh. A normal field in FEniCS is a function of the mesh. However the upper definition of the normal of a vertex applies only on vertex on the boundary. The goal of this function is to generalize the definition of a normal field from over the boundary to over the mesh by interpolation, having a smooth but rapid cancellation of the normal field when moving away from the obstacle. The function is then composed out of two parts. In a first place, a normal field over the boundary of the obstacle is computed, and then, out of this, a normal field over the whole mesh.

The solve function is used in order to develop such functions. In the first part we decide to solve a vector function defined on the boundary called vector_field. We define constant functions normal_x and normal_y being the components of the normal at each vertex with the **vertex_normal** function. We define x, y as being trialfunction of the boundary space and we solve the variational problem :

<normal_field[0], x> + <normal_field[1], y> = <normal_x, x> + <normal_y, y>

In the second part, we want to solve the deform function, a vectorial function of the mesh. We define a test function v and we solve the variational problem :

$<\nabla deform, \nabla v> + <deform, v> = <(0.0, 0.0), v>$

Having the boundary conditions:

Deform = 0 on the left and right boundaries

Deform = normal_fieldV on the obstacle boundary

Normal_fieldV is a vectorial function of the mesh which is null everywhere except on the obstacle boundary where it is equal to the previously defined normal_field.

Optimising the form of the mesh

The function **optimize_mesh** is the only one accessed by the user in the main file. It is therefore a synthesis of all functions explained before.

In this last function, we aim at computing an optimal form of the obstacle in order to minimize the function

$J(u) = \frac{1}{2} . <\nabla u, \nabla u>$

, after n_iter iterations and under a volume constraint: we allow little variation over the surface of the mesh (otherwise the surface would naturally drive towards 0). At each iteration, we have to plot the speed u, the pressure p and J(u) in order for the user to see a drive towards an optimized form.

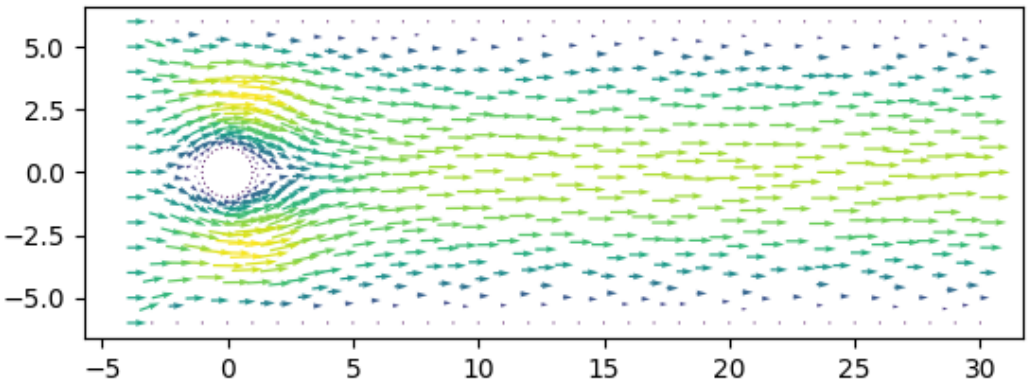
It has been decided to implement a gradient-based descent minimization. Taking an original mesh as entry, we modify it at each iteration with **update_mesh**. We then make a call to **StokesSolve** and see if J(u) then obtained is lower than the previous best value we had obtained. If so, the new form is accounted as the new optimised form and we move forward.

The function **update_mesh** is at the core of the process. It is called n_iter times, and at each time, begins by modifying each vertex of the mesh in the direction of $<\nabla u, \nabla u> . \vec{n}$. $\vec{n}$ is a null vector function

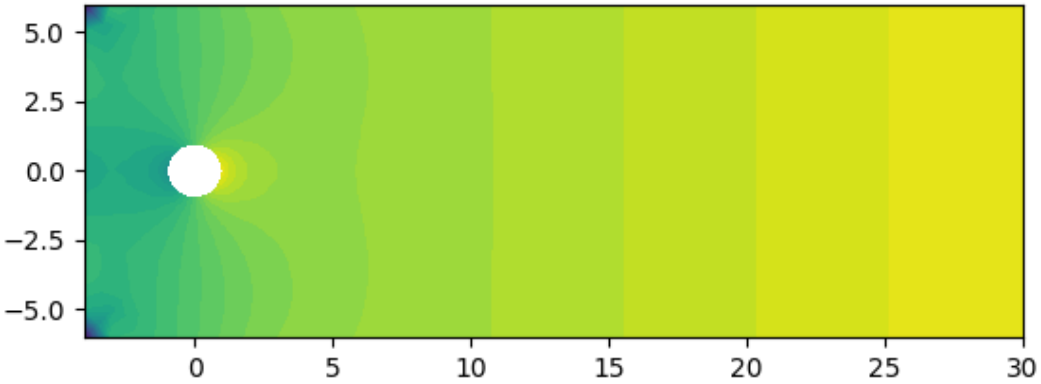
everywhere except close to the boundary of the obstacle this implies that only the vertex close to obstacles are being modified. The constant rho characterizes this modification. However, the modification is sometimes too important. When this occurs, we decide to slightly modify the vertices in a direction $\varepsilon<\nabla u, \nabla u>.\vec{n}$, until the surface of the mesh is close satisfies the volume constraint.

V. Réalisation

2. Results for MyStokes.py



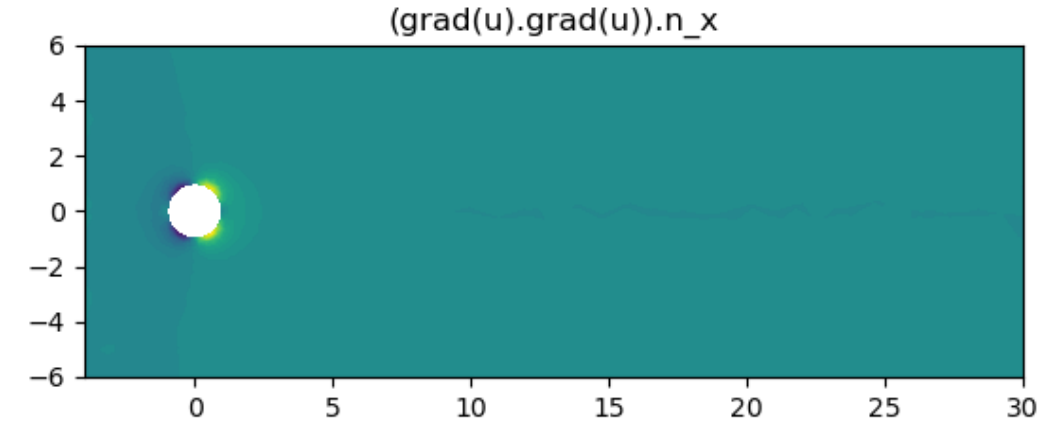
Velocity field

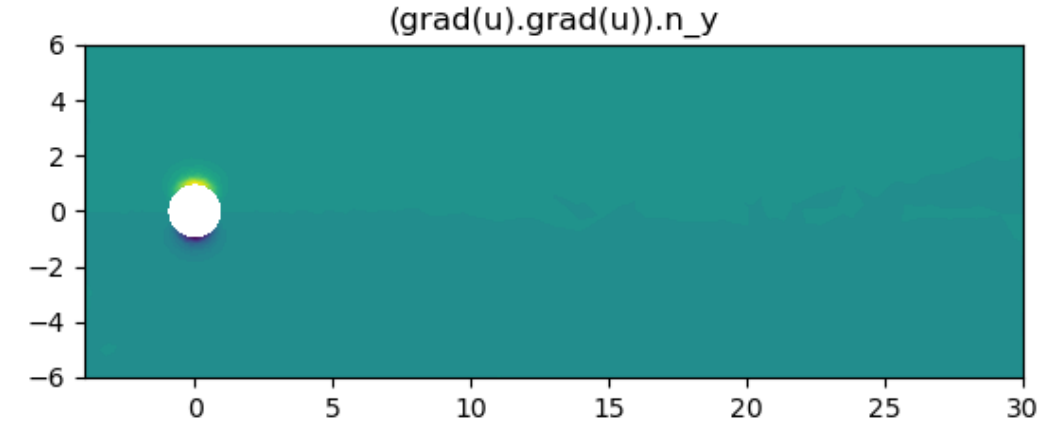


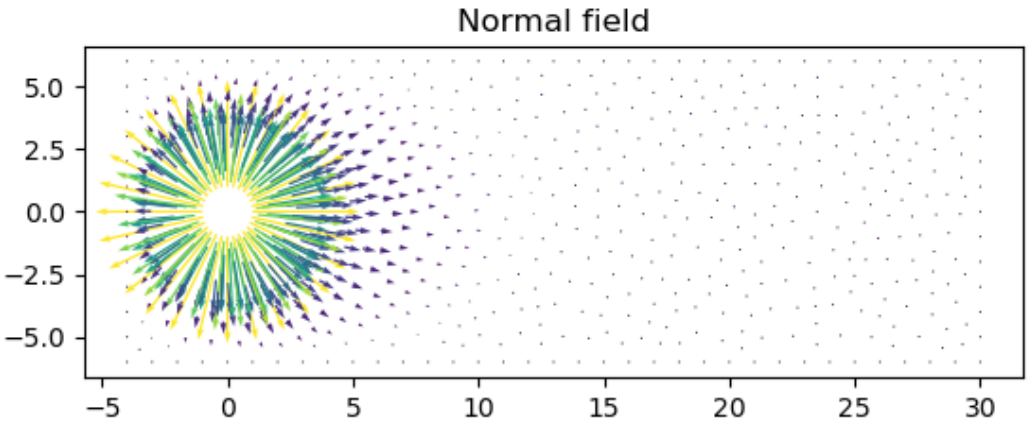
Pressure field

3. Compute_normal_field results

Here we can see what our normal field and gradient field initially looked like:



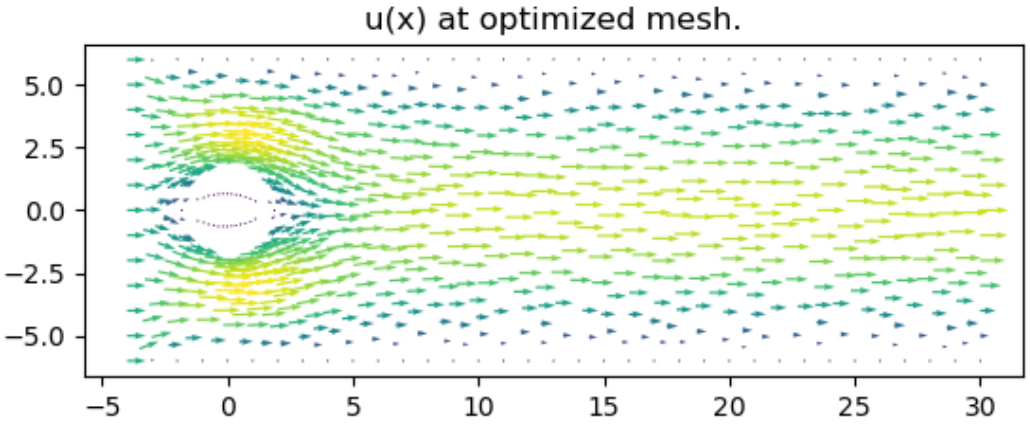


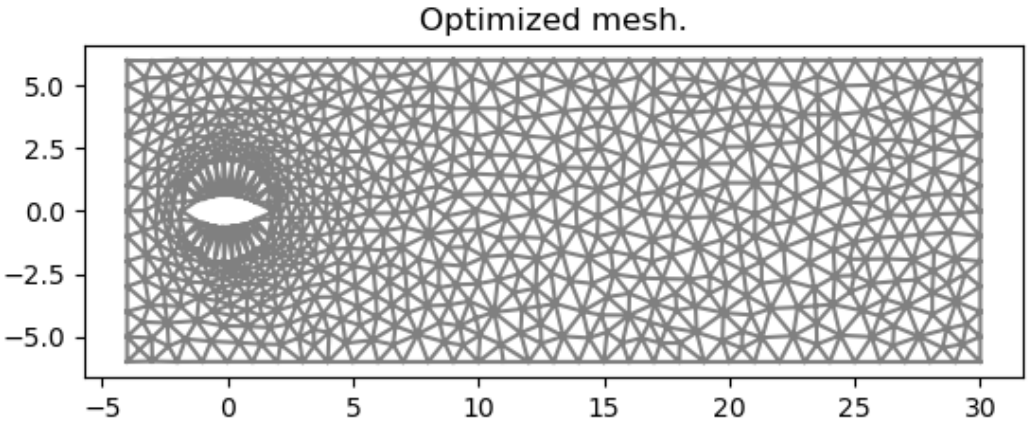


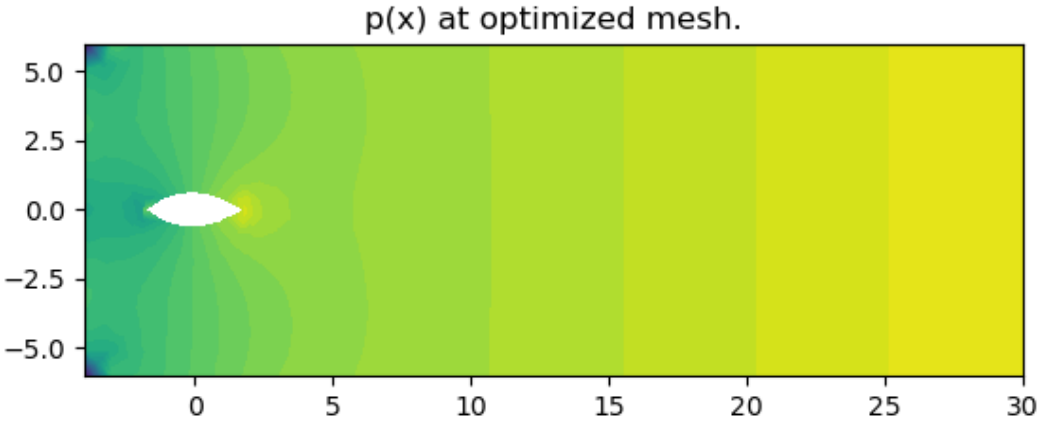
4. Final results for the sequential code

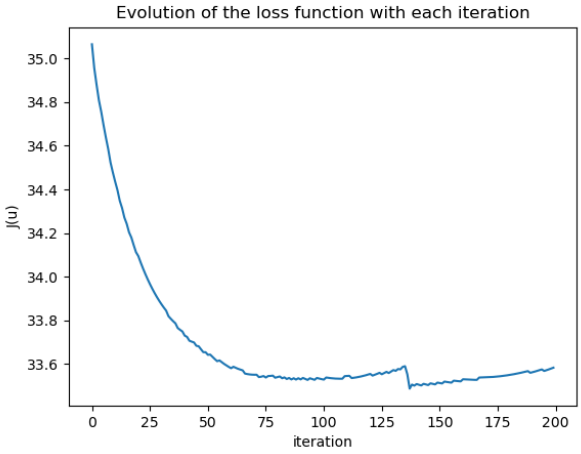
Here the results that we obtained while running the algorithm with the following parameters:

$$\rho = 0.003, \varepsilon = 0.001\,, n_{iter} = 200$$

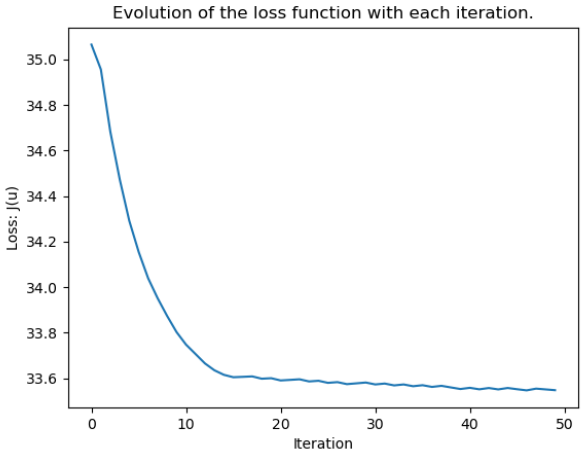

u(x) at optimized mesh.
5.0
2.5
0.0
−2.5
−5.0
−5
0
5
10
15
20
25
30


Optimized mesh.
5.0
2.5
0.0
−2.5
−5.0
−5
0
5
10
15
20
25
30


p(x) at optimized mesh.
5.0
2.5
0.0
−2.5
−5.0
0
5
10
15
20
25
30


Evolution of the loss function with each iteration
J(u)
35.0
34.8
34.6
34.4
34.2
34.0
33.8
33.6
0
25
50
75
100
125
150
175
200
iteration

Additionally, here is the evolution of the loss obtained with variable step sizes for the gradient descent:



We can see that the algorithm converges indefinitely to a point that is not a local minimum for the loss function.

VI. Parallelization of the algorithm

5. Description

Up until this point the project discussed has been developed to run sequentially. The further goal for us has been to translate our work into a parallelized version, in order to do that, we need to:

- In the first place, be able to distribute efficiently the workload required to solve the optimization problem.
- In second place to run our project on the Machine Du Grand Est.

Therefore we proceeded by dividing our domain in four overlapping subdomains through the use of the gmsh software, on which the wing tip and the related mesh were originally plotted, and after that we applied the *Parallel Schwartz Method* as a means to parallelize the code, applying MyStokes on each subdomain separately.

The Parallel Schwarz Method is an iterative method that comes from the alternating Schwarz method, which was created to solve a Dirichlet problem in the case of two overlapping domains (specifically, it was firstly studied in the case of a circle and a square interlaced). The relative algorithm goal is to converge on a solution for the value at the boundary, taking into account that, since the domains are jointed, it should reach an according value on it. The Parallel improvement was later introduced in the method in order to be able to solve the problem of the two different domains on different processors, so to effectively parallelize the algorithm:

$$Lu_i^n = f, in\ \Omega_i\ \ with\ u_i^n = u_j^{n-1}, on\ x = l_{i,j}$$
$$Lu_j^n = f, in\ \Omega_j\ \ with\ u_j^n = u_i^{n-1}, on\ x = 0_{i,j}$$

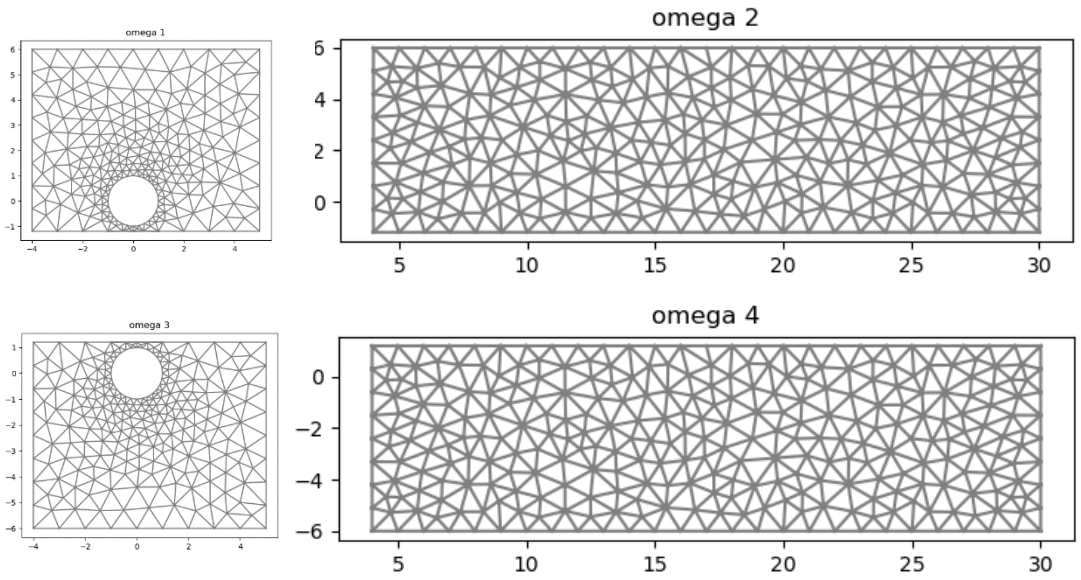
In this way the computation for Ω_i and Ω_j is still not independent but can be done in parallel.

In our case it allowes us to apply our modified Navier-Stokes equation code to the four overlapping subdomains we generated simultaneously, granted the return for each step of the boundary conditions.

The rest of the code (MyStokes, MyShapeOpt and MyMesh) stays unchanged with respect to the sequential codification.

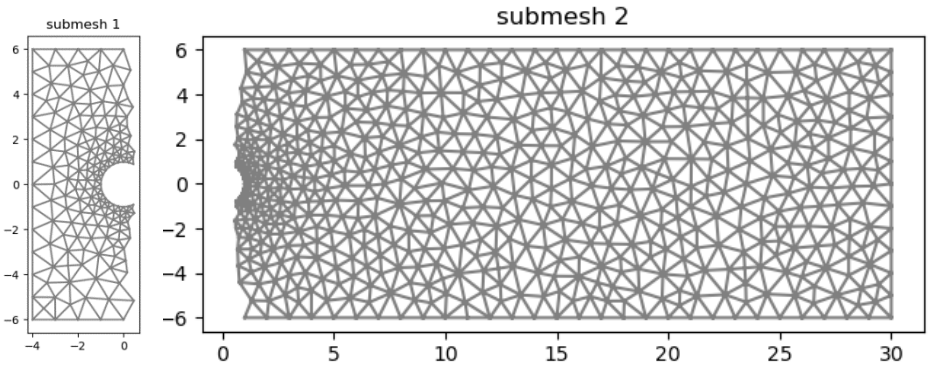
6. Mesh Subdivision

As explained previously, the mesh subdivision was done manually on gmsh in order to obtain four meshes that share a similar workload.



However, since our algorithm is intended to be scalable, we also implemented a function that automatically segments a given mesh into a specified number of sub-meshes (see Parallel Improvement part).

The output of the given function can be visualized below:

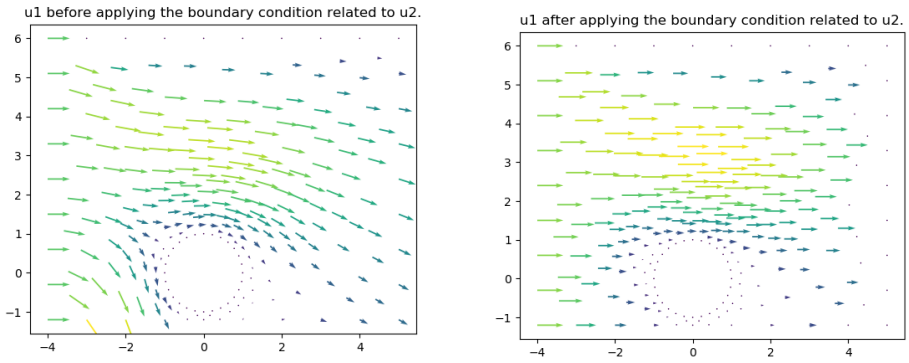


7. First Version of the code: single computer use

To properly implement the Schwarz method, we followed these steps:

- First, we calculate the velocities and normal fields using StokesSolve on each of the meshes.
- For each Submesh, the corresponding velocity function is modified by equalizing it to the other velocities on each of the intersections of their meshes, which can be formulated as:

$$\forall i, j : u_{i|W_i \cap W_j} = u_j$$



- To update the shape of the mesh is a bit trickier, since the coordinates of the vertices in the overlapping domain must be the same for the each of the subdomains.
- We do a similar loop that updates the mesh using the updated velocities then equalizes the coordinates of the mesh on each of the vertices that are in some intersection:

$$\forall i \neq j \; and \; v \in W_i \cap W_j : c_i^u(v) = c_j^u(v) = \frac{c_i(v) + c_j(v)}{2}$$

Where i, j refers to the meshes, c is the corresponding coordinate of the vertex in the mesh and u means after update.

8. Second Version: MPI deployment for the cloud of Machine Du Grand Est

Now we want to implement the previous algorithm by adding the parallel part.

First of all, we need to allocate using mpirun command one process per sub-mesh. To do so, we use the command: "`mpirun -np nbProcesses -map-by ppr:1:core -rank-by core -bind-to core python3 parallel.py`". Here, we bind each process to one core. Then each process computes the velocity function and normal fields using StokesSolve on his sub-mesh.

As the mesh is divided in 4 sub-meshes overlapping each other, each process needs to send the value of the velocity on the overlap to each other process. The even processes start sending data using `Ssend` function while the odd ones are waiting to receive data (`Recv` function). Attention must be paid to which process the information is sent and from which process the information is expected. If not, we create a dead point where every process waits something that will never arrive.

When the information has been shared, each process can compute, as in sequential, the new velocity function and the new normal fields.

9. Results

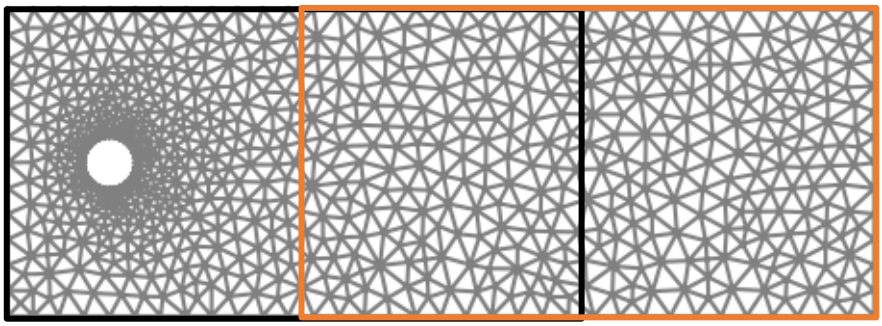
Unfortunately, the cluster of Machine du Grand Est wasn't working when we wanted to launch our code. And we didn't manage to run it on our own computer. But we can point some difficulties: if the latest version of FEniCS is running is parallel, he refuses to open ".xml" files that code for a MeshFunction. To create one, we must convert ".xml" files into ".h5" files using FEniCS HDF5File function.

VII. Conclusion and improvement

10. Parallel improvement

We divided the mesh by hand using gmsh. This allowed us to have 4 sub-meshes to run the code in parallel. With this method, if we want to use more processes, we must write again the gmsh code by hand and create more sub-meshes. We can improve this by creating a function that take the initial mesh and divides it into overlapping sub-meshes.

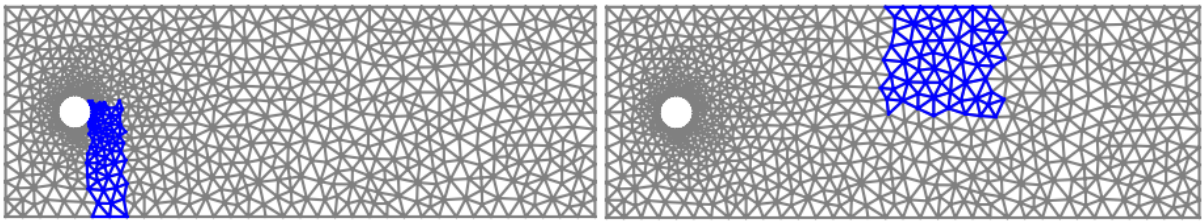
The idea is to create rectangles: (Here two overlapping rectangles)



Each triangle is considered in a sub-mesh if and only if it is in the corresponding rectangle. It isn't too complicated to create these sub-meshes with FEniCS:

A rectangle can be describe by "`min_x - tol <= x[0] && x[0] <= max_x + tol`" (the same thing with x[1] which correspond to the vertical axis). "`tol`" is a little tolerance (=1e-14 for example) to handle with calculation approximation. Then we call the CompiledSubDomain FEniCS function to create a subdomain, and we transform it into a mesh using SubMesh FEniCS function.

But one can notice that in the example, where the two rectangles have the same shape, that there are much more vertices in the black rectangle than in the orange one. So, the workload will be unbalanced. To manage this problem, we can look at the different abscissa of the point and take the median (if we want to cut in half) without forgetting to add an overlap constant. Same thing with y if we also want to divide the mesh following the vertical axis. At the end, we can obtain the following result (2 different rectangles among 2 lines of 8 rectangles):



It is important that the workload is well balanced. As all the processes will run in parallel, to optimize the time consumption we must be sure that all processes will end each iteration almost at the same time.

There is still some work to do with this method. If the number of points in the mesh is not sufficient, or the number of processes too high and the overlap constant too low, the sub-meshes might not overlap at some places. So, we have to be sure that there is no problem by looking the result and changing the overlap constant by hand. We can imagine some more complex function that can adapt to this kind of problem and even define more astute sub-meshes (than simple rectangles).

11. Conclusion

FEniCS is a very powerful tool. But we first struggle to use it. We had to understand how to deal with the notation and the different python objects created by FEniCS. Now our understanding of FEniCS is much better and we can focus on the physical meaning of the equations and the results rather than on computing the results correctly.