Fast Octree pseudo-compressible solver for wind engineering application

Treball realitzat per:
Clément Lemardelé

Dirigit per:
Riccardo Rossi
Rubén Zorrilla Martínez

Màster en:
Enginyeria de Camins, Canals y Ports

Barcelona, 08/09/2019

Departament d'Enginyeria Civil i Ambiental

TREBALL FINAL DE MÀSTER

Abstract

Nowadays, the resolution of the Navier-Stokes equations in the case of quasi-incompressible fluids using standard methods requires powerful computers because of the too huge computational cost of the numerical simulation. The main objective of this work is to tackle this issue by developing a Proof of Concept of a new, faster and more optimised solver which will associate Q1P0 finite elements, the Shifted Boundary method, the Back and Forth Error Compensation and Correction method as well as a Fractional Step splitting.

Q1P0 finite elements have been developed in the late 1970's and have shown interesting results in structural mechanics, particularly in the case of incompressible material, but their utilisation in Computational Fluid Dynamics (CFD) problems is way-less documented. This type of finite element combines a good precision and a reasonable computational cost. However, they show very poor results when used in unstructured meshes and that is why the implementation of the Shifted Boundary method is needed to guarantee the consistence of the applied boundary conditions in a structured mesh. The equations one obtains after the Galerkin discretisation cannot be solved with a standard Forward Euler scheme because of the numerical instability induced by the high value of quasi-incompressible fluids bulk modulus. In this context, the Fractional Step splitting scheme makes the simulation stable for a larger range of time steps values, which considerably betters the computational cost of the overall simulation because bigger time steps (in comparison with the Forward Euler scheme) can be used. Finally, the Back and Forth Error Compensation and Correction method is a very robust and unconditionally stable way to deal with the convective term of the Navier-Stokes equations.

All these numerical methods are implemented in a Python application which enables the user to check the performance of this newly-designed solver when dealing with simple and well-documented problems. First of all, the Couette one is analysed in order to show the accuracy of the results when only considering viscous effects. The solver shows good convergence when refining the mesh. Then, the flow in a channel is simulated to highlight the behaviour in the incompressible limit. Finally, various simulations of the Von Karman problem, with Reynolds numbers between 10 and 2 000, have been done using different meshes. In spite of what one could expected, vortices could not be observed. Because of the limited capacity of the author's personal computer capacity, a quite coarse mesh had to be used and this may explain the failure because too big elements are not able to represent the competition between viscosity and the convective term that occurs in the boundary layer of the cylinder.

Abstract

Hoy en día, la resolución de las ecuaciones de Navier Stokes en el caso de fluidos casi incompresibles mediante métodos usuales requiere ordenadores muy potentes por culpa de un coste computacional demasiado importante. El objetivo principal de ese trabajo final de Master es resolver este problema haciendo la demostración conceptual de un solver nuevo, optimizado y más rápido que asocia elementos finitos Q1P0 y los método numéricos "Shifted Boundary conditions", "Back and Forth Error Compensation and Correction" y "Fractional Step Splitting".

Los elementos finitos Q1P0 fueron desarrollados en los años 1970 y permiten obtener resultados representativos en mecánica estructural, en particular en el caso de material incompresible, pero su utilización en problemas de Computational Fluid Dynamics (CFD) es mucho menos documentada. Este tipo de elemento combina una precisión aceptable con un coste computacional razonable. Sin embargo, presentan resultados poco representativos cuando se trabaja con mallas no estructuradas. Para garantizar la constitencia de las condiciones de contorno en una malla estructurada, es necesario implementar el método "Shifted Boundary Conditions". Las ecuaciones que uno obtiene después de la discretización no se pueden resolver con un esquema Forward Euler estandar por culpa del valor muy alto del módulo de compresibilidad. En ese contexto, un esquema tipo "Fractional Step splitting" estabiliza la simulación numérica y permite utilizar pasos de tiempo más largos. Finalmente, el método "Back and Forth Error Compensation and Correction" es una manera muy robusta e incondicionalmente estable de tratar el término convectivo de las ecuaciones de Navier Stokes.

Los métodos numéricos citados anteriormente fueron implementados en un código Python que permite al usuario comprobar el comportamiento de este solver nuevo al resolver casos de estudio sencillos y bien documentados. Primero, el problema de Couette está analizado para demostrar la precisión de los resultados cuando sólo se tiene en cuenta los efectos viscosos. Luego, la simulación de un flujo dentro de un canal nos permite estudiar la robustez del solver en el límite incompresible. Para terminar, varias simulaciones del problema de Von Karmann con un número de Reynolds, entre 10 y 2 000, fueron desarrolladas utilizando varias mallas con características distintas. A pesar de lo que se podía esperar, los vórtices de Von Karmann no aparecieron. Por culpa de la capacidad limitada del ordenador del autor, se tuvo que utilizar una malla con elementos bastante grandes que no pudieron captar la competición que existe entre los efectos viscosos y el término convectivo en la capa límite del obstáculo.

Contents

List of Figures

1 Solving very large engineering problems

Engineers, nowadays, almost systematically need to use numerical simulation in order to solve the problems they have to face. Numerical modelling is not only an extraordinary scientific tool, it enables company better their competitiveness by reducing the cost of development, above all prototyping.

Even if computational mechanics has made great improvements in the last decades (considering the number of applications and their complexity), some particular types of problems remain very difficult to solve with current techniques. Let us imagine one wants to solve a huge problem, for instance, modelling the wind field within a city like Barcelona for several days in order to study the pollution evolution depending on atmospheric conditions. Current technology is not able to provide engineers with a cheap solution: one may need to use a supercomputer, because personal computers lack computation power, which would cost lots of money. The final objective of this work is to solve this kind of problem, i.e. reduce the computational cost of the simulation and find a way to rapidly solve huge problems. They are nowadays solved using standard Computational Fluid Dynamics (CFD) techniques which are too slow and not optimised enough.

We will need to find a trade-off between quickness and precision. More precise numerical schemes generate longer simulations because more equations are to be solved by the computer and, of course, this process takes more time. If we are to find any possible means to make the simulation faster, this may imply using less precise technology.

However, this is not really a problem from an engineering point of view. Let us come back to the example of the wind distribution within a city. Having a very precise numerical simulation would not make any sense at all. In such a complicated analysis which would couple thermal, mechanical and maybe chemical problems (because of air contamination reactions), there are many sources of uncertainty in the mathematical model itself. Only for the geometry, many models are to be tackled. Should one model the trees which are in the streets of the city? Which precision does one have when considering the dimensions of the buildings? The boundary conditions are even more complicated to determine because it is impossible to precisely know the wind direction, its velocity or the turbulence index of the atmosphere. In this context, having a too precise numerical simulation does not make any sense because the inputs of the model themselves would have a too important variability. In a nutshell, it is reasonable to loose some precision in order to make the simulation faster.

1.1 The Finite Element Method

When dealing with complex multi-disciplinary engineering problems, most of the time it is impossible to find an analytical solution to the Partial Derivatives Equations (PDE) which govern the behaviour of the system. Nevertheless, many mathematical tools, such as the Finite Element Method (FEM), enable engineers to find a numerical approximation of the solution. Since some problems are too complex to be solved globally, they are divided into much smaller ones, where it is much easier to approximate the analytical solution. The FEM is nowadays a wide-spread technique which has many applications, from the numerical simulation of electromagnetic fields to the computation of the pressure field into some arteries of the human body.

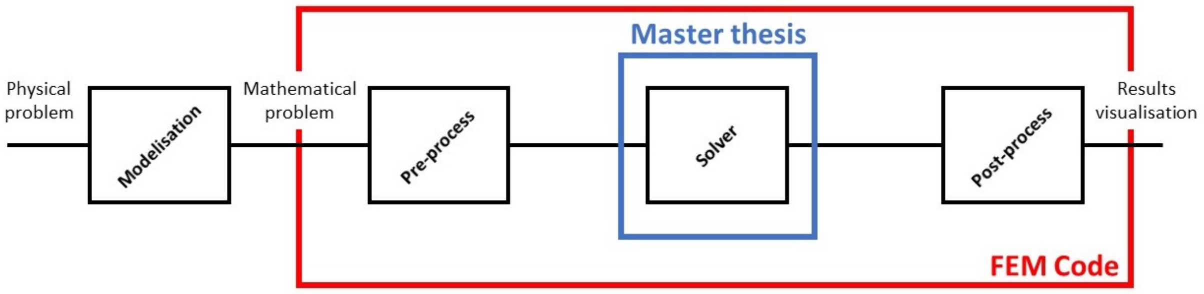

Figure 1: Modelling and resolution process of an engineering problem

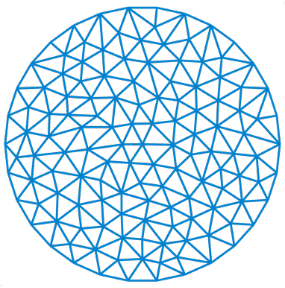

Figure 2: Example of a meshed round domain

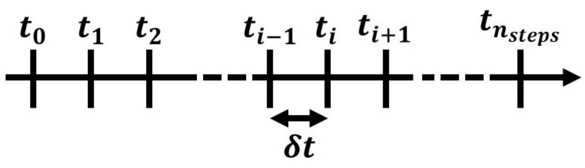

Figure 3: Example of time discretisation with constant time step

However, the FEM has some limitations. On the one hand, the precision of the numerical approximation depends on the mesh resolution. The more finite elements there are in the model, the more precise will be the numerical simulation. On the other hand, the time needed to carry out the numerical operations depends on the number of elements too. The more elements in the mesh, the longer will be the simulation. If one does not have access to a supercomputer, it is almost impossible to run huge simulations because the number of elements is too important and the computational cost to carry out the calculation is too high for a standard PC. In this context, we need to adapt and optimise the FEM in order to find a trade-off and achieve our particular goal.

1.2 The modelling process

Most of the time, engineers follow the process described in Figure 1 to solve the problems they are facing. The starting point is the physical problem itself (the wind field in a city). To use a FEM code, this problem has to be mathematically modelled, i.e. one has to write mathematical equations to represent the geometry of the problem (city geometry, ...), describe the behaviour of the system (constitutive equations of the fluid, Newton second law of mechanics, mass conservation, ...) and the boundary conditions (mostly the wind field far away from the city). Finally, these data are introduced in a FEM code which will solve the problem and, almost magically, display the results of the simulation on the screen.

Within the proper FEM code, if ones looks at it with more attention, three main operations are performed.

First, the mathematical problem is pre-processed. The geometrical domain is divided into

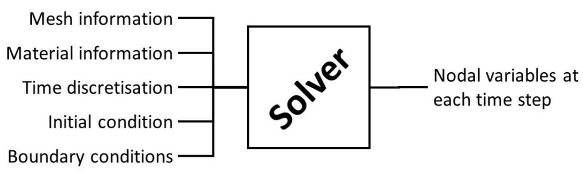

Figure 4: Inputs/Outputs of the solver

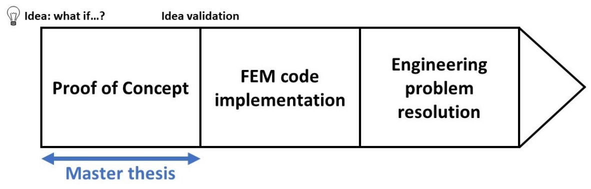

Figure 5: Project temporal development

smaller parts and meshed with finite elements (see Figure 2). The output of this process is a set of nodes and edges, which form the mesh of the domain. In addition, PDEs are discretised as well as time.

Once all the pre-process has been done, the solver can integrate the discrete problem over time to obtain the values of each nodal variables at each time step. At this stage, it is impossible to interpret the raw results given by the solver. So that they are useful to the user of the FEM code, they have to be post-processed, i.e. transformed into visual results, such as graph or contour lines.

The object of this master thesis is essentially the solver part. We will not focus on the domain meshing process for instance, even if it is one of the most important part of the modeling work, or the post-process of the solution. Schematically, we are to create a black box which takes into account some given inputs with a predefined structure and return the discrete solution of the problem. The inputs of FEM solvers are (see Figure 4):

- The mesh information. To entirely describe a mesh, one needs two different pieces of information: the nodes coordinates and the connectivity matrix, which contains information about the nodes which are associated to form a given element (the details of its implementation will be explained later on).

- The material characteristics. This part obviously depends on the type of engineering problem one is solving. For instance, in the case of an isotropic elastic problem, the behaviour of the material is completely defined by the Young modulus E and the Poisson ratio ν. For more complex materials, more parameters are needed.

- The time discretisation (see Figure 3). We are looking for the approximate solution for a discrete series of times t_i because it is impossible to solve the transient continuous problem. Most of the time, a constant time δt is adopted but considering a variable one could be possible.

- An initial condition which describes what the state of the system is before the simulation begins.

- Some boundary conditions which describe the interaction of the system with its environment.

1.3 The Proof of Concept (PoC)

The objective of the master thesis is to design a new solver which would be optimised for very large problems. At this stage of the project, it is impossible to say if this optimised solver will effectively give representative results in a practical engineering case. One could think about directly implementing it in a FEM code and see what happens. However, it would be a loss of time if the idea, in the end, is not good enough and does not work. To avoid this, what is generally done in the development of such projects is a Proof of Concept.

The main idea is to test the new solver for small and simplified problems for which there is bibliography and whose solutions (sometimes analytical ones) are well-known. By comparing the results of literature and the results of the new solver, we will know if it could potentially be efficient for more complex problems. To solve these simplified problems, we are to develop from scratch an application using the Python language. This application will not contain all the functionalities of a standard production-ready FEM code but this is not a problem because its only goal is to preliminary test the results ones gets from the new solver.

Once it is done and we get consistent results from the simple cases of study, we can assume that the idea we had is good enough, that this new solver will correctly work and we can pass to the next step, which would be the implementation directly in an alternative FEM code (see Figure 5).

To sum up, the objective of this master thesis is to do the Proof of Concept of a fast solver that could be used to deal with very large problems. To do that, we will use:

- Q1P0 finite elements technology to discretise PDEs
- A first-order Fractional Step Method to integrate PDEs over time
- A Shifted Boundary Conditions method to apply boundary conditions
- The Back and Forth Error Compensation and Correction Method to deal with the convective term of the Navier-Stokes equation

All these concepts as well as their numerical implementation will be detailed later on. These technologies have already been separately implemented and the objective of the thesis is to prove their robustness and efficiency when associated all together.

1.3.1 The Q1P0 finite element

In structural mechanics, low-order displacement-based elements are subject to locking when dealing with problems involving bending or incompressibility (see [CCCdS03] and [VdO05]), which are quite common restrictions. For instance, in geotechnical engineering, under certain conditions, the soil has to be modelled as an incompressible material. When locked, the finite element mesh happens to be much stiffer than it should be, which gives non-representative results.

Q1P0 elements were especially developed in the 1970's as a response to this problem. They are

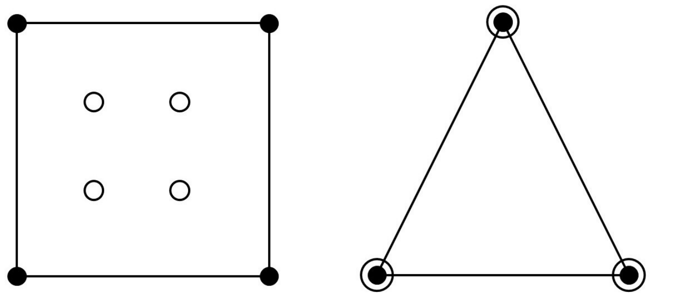
Figure 6: Examples of finite elements used for mixed formulation problems

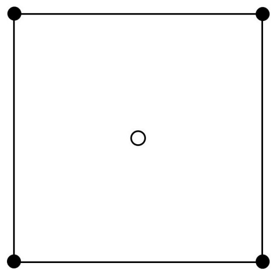
Figure 7: The Q1P0 finite element

based on a mixed formulation, i.e. the mechanical equations are expressed in terms of the displacement **u** and the isotropic pressure p, which is a new unknown of the problem. In the case of displacement-based elements, the whole formulation of equations is only done in terms of the displacement **u**. Displacement-based formulation are easier to discretise and their computational cost is lower but it is subject to locking.

	Displacement-based formulation	**Mixed u-p formulation**
Variables	Velocity **u**	Velocity **u**, pressure p
Locking	Yes	No
Computational cost	High	Low

Table 1: Comparison between displacement-based and mixed **u**-p formulations

Figure 6 presents examples of finite elements used to discretise mixed **u**-p formulations. The displacement and pressure nodes are respectively represented by $\bullet$ and $\circ$. Their number and positions[1] have a great impact on the element efficiency, as described in [VJP08]. The Q1P0 finite element is a quadrilateral one characterised by four displacement nodes (at the four corners of the element) and one pressure node in the centre of the element (see Figure 7).

Not all mixed elements provide representative results. One criterion to prove the mathematical convergence of mixed elements is the Ladyzhenska-Babuska-Brezzi (LBB) condition (see [CCCdS03] and [Hug12]). The idea is that, to work well, the element needs to have a "good balance" between the velocity unknowns and the pressure ones. In incompressible problems, the mass balance equation ($\boldsymbol{\nabla} \cdot \boldsymbol{u} = 0$) represents an additional constraint for the displacement field **u** which already has to satisfy the momentum balance equation. In the case of displacement-based formulations, the element does not manage to satisfy all the restrictions only with the displacement unknowns and that is the reason why it locks: there are too many constraints to be satisfied. Using a mixed **u**-p formulation makes the element less stiff because we introduce the nodal pressure values, which are new DOFs of the system. Schematically, the element has more variables to adapt itself to all the constraints.

However, if there are too many pressure DOFs, the elements looses too much stiffness because, in a sense, there are more variables than constraints. This is the main idea which is behind

[1] Let us remark that they are not necessarily located at the same place in the element

the LBB condition. Of course, the previous development cannot be considered as a rigorous mathematical demonstration. For more information, the reader can refer to [CCCdS03]. As for the Q1P0 element, it has been proven in literature that it satisfies the LBB condition and show good results[2].

In a nutshell, the Q1P0 element technology has been developed for structural mechanics to model incompressible problems. As a consequence, it makes sense to think that this kind of elements could work in Computational Fluid Dynamics (CFD) since most of the time we have to deal with quasi-incompressible material such as air or water. It is worth it to try to develop a CFD solver with this type of finite elements.

1.3.2 The Fractional Step Method

One has to distinguish between stationary problems and transient ones. In the first case, the variables (displacement $\mathbf{u}$ and pressure p for instance) do not depend on time whereas they do in transient ones. The objective of the new solver we want to develop is to deal with transient problems; we need to define a time integration strategy from the initial conditions of the problem.

The Fractional Step Mehtod (FSM) is widely-used in CFD to find the numerical solution of incompressible flows. It is based on a mixed $\mathbf{u}$-p formulation. At each time step, the main idea is to find a first approximation of the velocity field $\mathbf{u}$ which satisfies the momentum balance equation but which does not take into account the mass balance equation[3]. Then, the pressure field p is computed such that the mass balance equation is satisfied. The final operation is to update the first approximation of $\mathbf{u}$ we previously computed, taking into account this pressure field. For more details, the reader can refer to [Cod01a], [Cod01b], [BCH98], [FP02], or [MR18]. The main advantage of the FSM when compared to a standard monolithic scheme is that $\mathbf{u}$ and p are decoupled, which considerably lowers the computational cost.

[2]The Q1P0 element shows really good results in structured meshes. This is not always the case with unstructured ones

[3]This first approximation of $\mathbf{u}$ thus does not necessarily complies with the mass balance equation

2 Linear algebra preliminaries

Before beginning the formulation of the Navier-Stokes equations for a Q1P0 element, it is worth it to introduce some mathematical concepts that we will use later on in this work.

Analysing the computational cost of a simulation is one of the most important point in computational engineering. An interesting code can be completely useless in the practice if the time one needs to run it is too huge.

We are to determine the computational cost of some basic matrices operations, which are afterwards combined to create much more complicated functions. In our code, the most used matrices operations are basically the matrices multiplication and the resolution of linear systems of equations. Most of the time, the computational cost of some operations is expressed in terms of even more basic operations, such as scalar product, scalar addition or variable assignation. In the whole work, the capital letter C refers to the computational cost of a certain sequence of operations.

2.1 Matrix operations computational cost

2.1.1 Matrices multiplication computational cost

Let us consider two matrices $\mathbf{A}$ and $\mathbf{B}$ of size (n_A, m) and (m, n_B) respectively. The product of $\mathbf{A}$ and $\mathbf{B}$ is a (n_A, n_B) second-order tensor $\mathbf{D}$, defined as:

Definition 1. $\forall i \in \left[|1, n_A|\right], \forall j \in \left[|1, n_B|\right]$, $\mathbf{D}_{ij} = \sum_{k=1}^{m} \mathbf{A}_{ik}\mathbf{B}_{kj}$

where $\mathbf{D}_{ij}$ is the term of $\mathbf{D}$ at line i and column j.

Now let us count the number of scalar products $\mathbf{A}_{ik}\mathbf{B}_{kj}$ the computer has to do compute $\mathbf{D}$. For each $\mathbf{D}_{ij}$, we have to compute m scalar products (see the summation indices in Definition 1).$\mathbf{D}$ contains $n_A \cdot n_B$ terms. As a consequence, the standard matrices product has a computational cost of $m \cdot n_A \cdot n_B$ because the computer needs to do $n_A \cdot n_B$ times m scalar multiplications, which are unitary operations.

2.1.2 Matrices summation computational cost

Let it be $\mathbf{A}$ and $\mathbf{B}$ two (n, m) second-order tensors and $\mathbf{E} = \mathbf{A} + \mathbf{B}$.

Definition 2. $\forall i \in \left[|1, n|\right], \forall j \in \left[|1, m|\right]$, $\mathbf{E}_{ij} = \mathbf{A}_{ij} + \mathbf{B}_{ij}$

For each term $\mathbf{E}_{ij}$, it is necessary to do one basic scalar addition. There are $n \cdot m$ terms in matrix $\mathbf{E}$. The total computational cost of a matrix summation is $n \cdot m$ because the computer needs to compute $n \cdot m$ times one basic scalar addition. To have a fast simulation, one should limit the number of matrices multiplication because this operation as a much bigger computational cost.

2.2 Solving linear systems of equations

2.2.1 Symmetric positive definite matrix

Symmetric positive definite matrices are fundamental when dealing with FEM codes because they naturally appear in the formulation of discretised PDEs. Moreover, their main features make it possible to introduce some optimisations in the code.

Let us introduce some useful definitions.

Definition 3. $\mathbf{A}$ is symmetric $\iff \mathbf{A}^{\mathbf{T}} = \mathbf{A}$

where $\mathbf{A}^{\mathbf{T}}$ refers to the standard transposed matrix of $\mathbf{A}$.

The positive definite concept is a bit more complicated to define.

Definition 4. $\mathbf{A}$ is definite positive $\iff \forall \mathbf{x} \neq \mathbf{0},\ \mathbf{x}^{\mathbf{T}}\mathbf{A}\mathbf{x} > 0$

If a matrix is positive definite, it does not necessarily mean that all the terms of the matrix are positive. Moreover, a matrix containing only positive terms can be positive definite, or not. Let us study a numerical example.

Example 1. $\mathbf{N} = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} \qquad \mathbf{x} = \begin{pmatrix} -3 \\ 1 \end{pmatrix} \neq \mathbf{0}$

All the terms of $\mathbf{N}$ are positive. However, $\mathbf{x}^{\mathbf{T}}\mathbf{N}\mathbf{x} = -2$, which means that $\mathbf{N}$ is not positive definite even if all its terms are positive. Even though the definition of a positive definite matrix is not really intuitive, later on we will see that it can be geometrically interpreted.

2.2.2 Iterative methods for the resolution of linear systems of equations

Solving a system of linear equations is one of the most basic problem of linear algebra. However, it is one of the most time-consuming one too. The main objective here is to find a way to speed up the resolution of a linear system of equations, based on the Conjugate Gradient method.

We will not explain the entire theory behind this algorithm but just give some clue concepts. Deeply understand what it is at stake with the Conjugate Gradient algorithm is not the object of the work we are doing and we just consider it as a useful tool. For more information, the reader can refer to [S+94].

Let $\mathbf{A}$ be a symmetric positive definite second-order tensor of size (n, n). We are to find the solution of the linear system of equations

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

where $\mathbf{b}$ is a known vector and $\mathbf{x}$ the unknown of the problem.

All the iterative methods one can use to solve this kind of linear systems of equations rely on the following lemma.

Lemma 1. *Find* $\mathbf{x}^*$ *such that* $\mathbf{A}\mathbf{x}^* = \mathbf{b} \iff \mathbf{x}^* = \arg\left(\min f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\mathbf{T}\mathbf{A}\mathbf{x} - \mathbf{b}^\mathbf{T}\mathbf{x} + c\right)$

Finding the solution of the system $\mathbf{A}\mathbf{x} = \mathbf{b}$ and finding the value of $\mathbf{x}$ which minimises the quadratic form $f(\mathbf{x}) = 0.5\,\mathbf{x}^\mathbf{T}\mathbf{A}\mathbf{x} - \mathbf{b}^\mathbf{T}\mathbf{x} + c$ are strictly equivalent problems. In the following, the proof of this statement is developed.

Let it be $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ the solution of the linear system and $\mathbf{p}$ a given point of the domain.

Proof.

$$\begin{aligned}
&f(\mathbf{x}) + \frac{1}{2}(\mathbf{p}-\mathbf{x})^T\mathbf{A}(\mathbf{p}-\mathbf{x})\\
&= \frac{1}{2}\mathbf{x}^\mathbf{T}\mathbf{A}\mathbf{x} - \mathbf{b}^\mathbf{T}\mathbf{x} + c + \frac{1}{2}(\mathbf{p}-\mathbf{x})^T\mathbf{A}(\mathbf{p}-\mathbf{x}) \text{ by definition of } f(\mathbf{x})\\
&= \frac{1}{2}\mathbf{x}^\mathbf{T}\mathbf{A}\mathbf{x} - \mathbf{b}^\mathbf{T}\mathbf{x} + c + \frac{1}{2}\mathbf{p}^\mathbf{T}\mathbf{A}\mathbf{p} - \frac{1}{2}\mathbf{p}^\mathbf{T}\mathbf{A}\mathbf{x} - \frac{1}{2}\mathbf{x}^\mathbf{T}\mathbf{A}\mathbf{p} + \frac{1}{2}\mathbf{x}^\mathbf{T}\mathbf{A}\mathbf{x}\\
&= \mathbf{x}^\mathbf{T}\mathbf{A}\mathbf{x} - \mathbf{b}^\mathbf{T}\mathbf{x} + c + \frac{1}{2}\mathbf{p}^\mathbf{T}\mathbf{A}\mathbf{p} - \frac{1}{2}\mathbf{p}^\mathbf{T}(\mathbf{A}\mathbf{x}) - \frac{1}{2}(\mathbf{A}\mathbf{x})^T\mathbf{p}\,\text{, because } \mathbf{A}^\mathbf{T} = \mathbf{A}\\
&= \mathbf{x}^\mathbf{T}\mathbf{b} - \mathbf{b}^\mathbf{T}\mathbf{x} + c + \frac{1}{2}\mathbf{p}^\mathbf{T}\mathbf{A}\mathbf{p} - \frac{1}{2}\mathbf{p}^\mathbf{T}\mathbf{b} - \frac{1}{2}\mathbf{b}^\mathbf{T}\mathbf{p}\,\text{, because } \mathbf{A}\mathbf{x} = \mathbf{b}\\
&= \frac{1}{2}\mathbf{p}^\mathbf{T}\mathbf{A}\mathbf{p} - \mathbf{b}^\mathbf{T}\mathbf{p} + c\,\text{, because } \mathbf{x}^\mathbf{T}\mathbf{b} = \mathbf{b}\mathbf{x}^\mathbf{T} \text{ and } \mathbf{p}^\mathbf{T}\mathbf{b} = \mathbf{b}\mathbf{p}^\mathbf{T}\\
&= f(\mathbf{p})\,\text{, by definition of } f
\end{aligned}$$

□

Finally, the result of this development is that

$$f(\mathbf{p}) = f(\mathbf{x}) + \frac{1}{2}(\mathbf{p}-\mathbf{x})^T\mathbf{A}(\mathbf{p}-\mathbf{x}) \tag{1}$$

Since $\mathbf{A}$ is definite positive,

$$\forall\,\mathbf{p} \neq \mathbf{x},\ (\mathbf{p}-\mathbf{x})^T\mathbf{A}(\mathbf{p}-\mathbf{x}) > 0 \tag{2}$$

Equations 1 and 2 prove that $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ is a global minimum of f because for all $\mathbf{p} \neq \mathbf{x}$, $f(\mathbf{p}) > f(\mathbf{x})$, which is the definition itself of a global minimum.

We just proved that, if $\mathbf{x}$ is a solution of the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$, then it is a global minimum of the quadratic form f. To completely prove Lemma 1, we need to show that the reciprocal is true. In other words, if $\mathbf{x}$ is a global minimum of the quadratic form f, then it is a solution of the linear system $\mathbf{A}\mathbf{x} = b$.

The main idea is to compute the gradient of f:

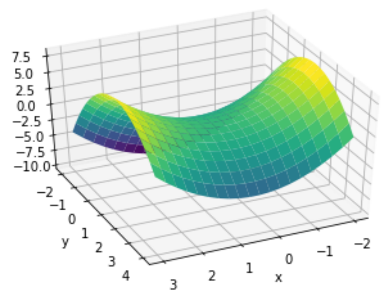

Figure 8: Quadratic form associated with non-positive matrix

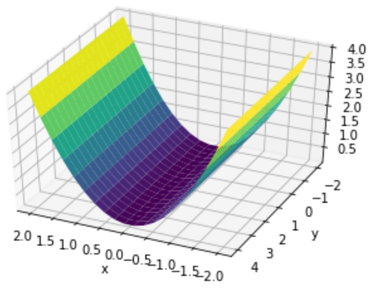

Figure 9: Quadratic form associated with singular matrix

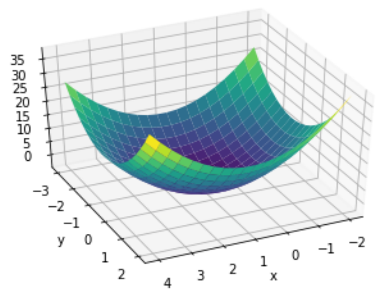

Figure 10: Quadratic form associated with positive definite matrix

$$
\begin{aligned}
f'(\mathbf{x}) &= \frac{1}{2}\mathbf{A}^{\mathbf{T}}\mathbf{x} + \frac{1}{2}\mathbf{A}\mathbf{x} - \mathbf{b} \\
&= \mathbf{A}\mathbf{x} - \mathbf{b} \text{ since } \mathbf{A} \text{ is symmetric}
\end{aligned}
$$

Then, if $\mathbf{x}$ is a global minimum of f, the theory tells us that the gradient is null in $\mathbf{x}$

$$
f'(\mathbf{x}) = \mathbf{0} \iff \mathbf{A}\mathbf{x} = \mathbf{b} \tag{3}
$$

and $\mathbf{x}$ is a solution of the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$, which proves the reciprocal of Lemma 1. For more details and a more rigorous demonstration, the reader can refer to [S+94].

Actually, the features of matrix $\mathbf{A}$ have a direct impact on the shape of f. Figures 8, 9 and 10 present the shapes of the quadratic form considering different types of matrices $\mathbf{A}$. When $\mathbf{A}$ is positive and definite (see Figure 10), there is clearly a unique global minimum. For a singular matrix $\mathbf{A}$, the set of global minima is infinite. Ultimately, if $\mathbf{A}$ is definite but not positive (see Figure 8), the presence of a global minimum is not ensured. A good understanding of the features of matrix $\mathbf{A}$ is mandatory.

2.2.3 The Steepest Descent method

Now that we know that solving a linear system of equations is, in some cases, equivalent to finding the minimum of a quadratic function, we should look for an efficient way to do it.

The first idea one can have when dealing with the minimisation of a certain function is the Steepest Descent method. At each iteration, we compute the gradient of f and we look for the minimum of the function in the direction given by this gradient[4]. The process is repeated until a convergence criterion is satisfied. The equations of the Steepest Descent method will not be detailed here because this is not the object of this work. For more details about the algorithm, one can refer to [S+94].

[4]The gradient of a scalar function f is a first-order tensor which points in the direction of the mayor values of f

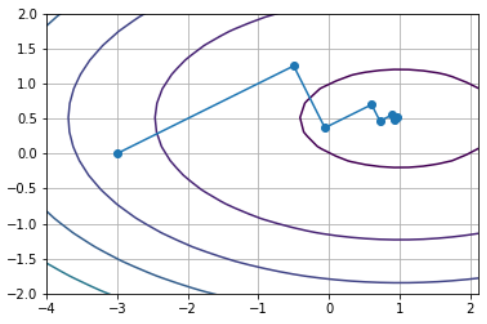

Figure 11: Iterations using the Steepest Descent method and $\mathbf{x_0} = [-3\,,\,0]$

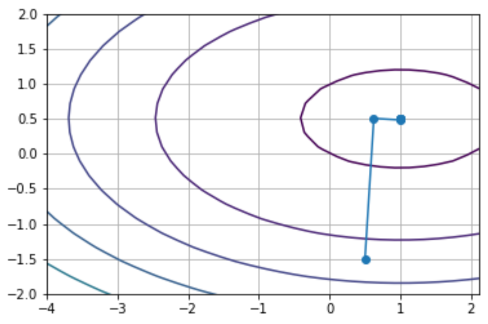

Figure 12: Iterations using the Steepest Descent method and $\mathbf{x_0} = [0.5\,,\,-1.5]$

Figure 11 presents the results of the iteration with:

Example 2. $\mathbf{A} = \begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix} \qquad \mathbf{b} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \qquad \mathbf{x_0} = \begin{pmatrix} -3 \\ 0 \end{pmatrix}$

The blue points represent the approximations $\mathbf{x_i}$ of the solution that are computed at each iteration. For the sake of clarity, we plotted the contours of the quadratic form f. The convergence up to the minimum of f is quite clear.

Nevertheless, the Steepest Descent algorithm is a really inefficient way to look for the minimum of a function because the different search directions at each iteration are orthogonal (see Figure 11), which means we search several times in the same direction. Moreover, the convergence depends a lot on the matrix $\mathbf{A}$ and the initial point $\mathbf{x_0}$. Figure 12 presents the results we get when taking as initial point $\mathbf{x_0} = [0.5\,,\,-1.5]$. In this case, the convergence is much faster than previously.

The Steepest Descent method does not take into account all the information we have about matrix $\mathbf{A}$ (we know it is symmetric, definite and positive). More generally, this algorithm can be used to look for the minimum of any function $f(\mathbf{x})$, and not only quadratic forms. In a nutshell, the Steepest Descent method is not optimal and it can be optimised.

2.2.4 The Conjugate Gradient method

The idea behind the Conjugate Gradient method is to look for the minimum of the quadratic form f using independent $\mathbf{A}$-orthogonal directions.

Definition 5. $\mathbf{x}$ and $\mathbf{y}$ are $\mathbf{A}$-orthogonal $\iff \mathbf{x^T A y} = 0$

At each iteration of the Conjugate Gradient algorithm, we look for a new search direction which has not been explored yet[5]. This way, the search directions are not repeated such as in the Steep-

[5]This is the key point of this numerical scheme

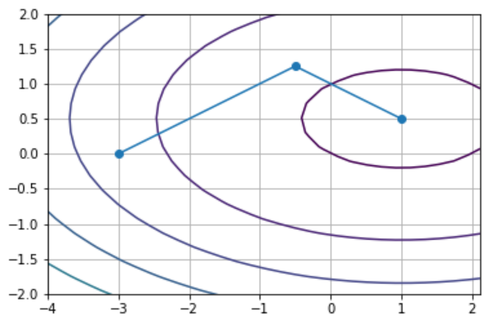

Figure 13: Iterations using the Conjugate Gradient method and $\mathbf{x_0} = [-3\,,\,0]$

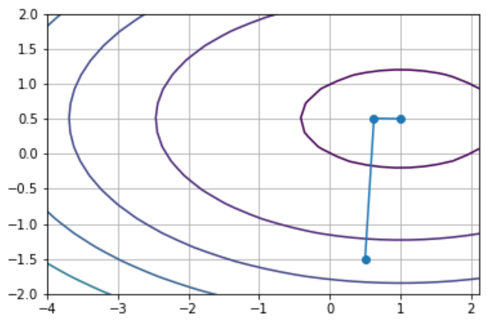

Figure 14: Iterations using the Conjugate Gradient method and $\mathbf{x_0} = [0.5\,,\,-1.5]$

est Descent method (see Figure 11). $\mathbf{A}$ being positive definitive, there are at most n independent $\mathbf{A}$-orthogonal directions[6]. The Conjugate Gradient algorithm thus converges to the exact solution with at most n iterations. Figures 13 and 14 prove that the Conjugate Gradient method exactly converges in 2 iterations whatever the iteration starting point is. This highlights the robustness of this minimisation algorithm.

Algorithm 1 Conjugate Gradient iterative method

```
Require: $\mathbf{A}$, $\mathbf{b}$, $\mathbf{x_0}$, $maxiter$, $tol$
  $\mathbf{d_0} = \mathbf{r_0} = \mathbf{b} - \mathbf{Ax_0}$
  $count = 0$
  while $count < maxiter$ and $||r_i|| > tol$ do
    $\alpha_i = \frac{\mathbf{r_i^T r_i}}{\mathbf{d_i^T A d_i}}$
    $\mathbf{x_{i+1}} = \mathbf{x_i} + \alpha_i \mathbf{d_i}$
    $\mathbf{r_{i+1}} - \alpha_i \mathbf{A d_i}$
    $\beta_{i+1} = \frac{\mathbf{r_{i+1}^T r_{i+1}}}{\mathbf{r_i^T r_i}}$
    $\mathbf{d_{i+1}} = \mathbf{r_{i+1}} + \beta_{i+1} \mathbf{d_i}$
    $count = count + 1$
  end while
  if $count = maxiter$ then
    print "The algorithm has not converged"
  end if
  return $\mathbf{x_i}$
```

Algorithm 1 presents the different operations one has to do within each iteration. For the sake of clarity, the details and interpretation of each operation are not described here. The output of the algorithm is the approximation $\mathbf{x}_n$ of the coordinates global minimum position.

We know that it would converge to the exact solution in n iterations. However, this would be too long if matrix $\mathbf{A}$ is huge (let us remember we are looking for the fastest way to solve a linear system of equations). As a consequence, we impose a maximum number of iterations

[6]This is a direct consequence of the spectral theorem

$maxiter$. Moreover, if the norm of the residual r_i is close enough to zero, one can assume that the approximation has reached a certain level of precision (defined through the variable tol) and stops the algorithm. The variable $count$ contains information about the number of iterations. If this number of iterations is equal to $maxiter$, this means the algorithm has not converged because we did not reach the level of precision imposed by tol. We went out off the loop because we had reached the maximum allowed number of iterations.

Let us say that $\mathbf{A}$ is a matrix of size (n, n). The most time-consuming operation of Algorithm 1 is the computation of $\mathbf{Ad_i}$. It has a computational cost of n^2. For instance, computing the scalar product $\mathbf{r_i^T r_i}$ only requires n scalar operations, which is negligible when considering very large problems ($n >> 1$). As we are imposing a maximum number of iterations (we do not let the algorithm go up to the exact solution), the computational cost C_{CG} of the Conjugate Gradient method is $O(n^2)$.

Proposition 2. $C_{CG} = O\left(n^2\right)$

Taking into account that the computational cost of the Gauss-Jordan elimination[7] is $O(n^3)$, this is a great improvement. The resolution of a linear system of equations where $\mathbf{A}$ is definite positive is much faster using the Conjugate Gradient method.

2.3 Norm of a solution

To prove the convergence of the solver we are to design, we will need to define a quality criterion for the numerical simulation. To do that, we will use the L^2-norm.

Let it be a scalar function $f\left(\mathbf{x}, t\right)$ which depends on the space variable $\mathbf{x} \in \Omega$ and time $t \in [0, t_{max}]$. We can define the L^2-norm of f by

$$||f||_2 = \sqrt{\int_\Omega \int_0^{t_{max}} \left|f\left(\mathbf{x}, t\right)\right|^2 \, dt \, d\Omega} \tag{4}$$

This norm is very useful for continuous (in time and space) functions. However, in the FEM, one deals with the nodal values of the function (space discretisation) at discrete times t_i. As a consequence, it is necessary to change and adapt the previous definition.

Let us consider a set of discrete times $\mathcal{T} = \left\{t_i \in [0, t_{max}]\right\}$ as well as a set of points $X = \{\mathbf{x}_i \in \Omega\}$. We are now able to define a discrete version of the L^2-norm as

$$||f||_2 = \sqrt{\sum_{\mathbf{x}_i \in X} \sum_{t_i \in \mathcal{T}} \left|f\left(\mathbf{x}_i, t_i\right)\right|^2} \tag{5}$$

In a sense, this is an approximation of the continuous integral. From our point of view, it is much more useful because we are only interested in the nodal values of the function at some

[7]Other numerical technique for solving linear system of equations

predefined times.

3 Formulation

The first part of this section is dedicated to the formulation of the Navier-Stokes problem. Then, the weak form, which will be solved afterwards using FEM, is presented.

3.1 Navier-Stokes problem

First of all, we need to define some useful notations. The velocity field in the fluid domain $\mathbf{u}\left[m.s^{-1}\right]$ is described by a first-order tensor which a priori depends on all the space variables as well as time.

$$\mathbf{u} = \begin{pmatrix} u(x,y,z,t) & v(x,y,z,t) & w(x,y,z,t) \end{pmatrix}^T$$

Then, we introduce the strain-rate tensor $\boldsymbol{\nabla}^S\mathbf{u}\left[s^{-1}\right]$, also known as symmetric gradient of $\mathbf{u}$

$$\boldsymbol{\nabla}^{\mathbf{S}}\mathbf{u} = \frac{1}{2}\left(\boldsymbol{\nabla}\mathbf{u} + \boldsymbol{\nabla}\mathbf{u}^T\right)$$

and express the constitutive equation of Newtonian fluids.

$$\boldsymbol{\sigma} = \begin{pmatrix} \sigma_x & \tau_{xy} \\ \tau_{yx} & \sigma_y \end{pmatrix} = -p\mathbf{I} + 2\mu\boldsymbol{\nabla}^{\mathbf{S}}\mathbf{u} \tag{6}$$

where $\boldsymbol{\sigma}\,[Pa]$ is the stress tensor, $p[Pa]$ the pressure in the fluid, $\mu\,[Pa.s]$ the fluid viscosity and $\mathbf{I}$ the identity matrix. Equation 6 relates the rate of change of velocity, expressed through $\boldsymbol{\nabla}^{\mathbf{S}}\mathbf{u}$, to the internal stresses in the fluid. Assuming that we are dealing with a Newtonian fluid, μ is assumed to be constant. Using all previous notations, it is possible to write the Navier-Stokes equations for a quasi-incompressible fluid

$$\boldsymbol{\nabla}\cdot\boldsymbol{\sigma} + \rho\,\mathbf{b} = \rho\frac{\partial\mathbf{u}}{\partial t} + \rho\,\mathbf{u}\cdot\boldsymbol{\nabla}\mathbf{u} \tag{7a}$$

$$\frac{\partial\rho}{\partial t} + \boldsymbol{\nabla}\cdot(\rho\mathbf{u}) = 0 \tag{7b}$$

where $\rho\left[kg.m^{-3}\right]$ refers to the fluid density. The formulation of the Navier-Stokes equations relies on two very simple physical concepts: the momentum balance equation 7a and the mass conservation equation 7b. The momentum balance equation is valid for every type of fluid; this is not the case for the mass balance equation. For an incompressible fluid, since $\partial_t\rho = 0$ and $\boldsymbol{\nabla}\rho = 0$, we directly get $\boldsymbol{\nabla}\cdot\mathbf{u} = 0$.

The mayor difficulty in Equation 7a is given by the convective term $\mathbf{u}\cdot\boldsymbol{\nabla}\mathbf{u}$. It introduces non-linearity in the Partial Differential Equation (PDE), which complicates the resolution of the problem, makes the numerical simulation unstable and greatly increases its computational cost. One way to efficiently deal with this problem is to split the PDE.

$$\rho\frac{\partial \mathbf{u} - \tilde{\mathbf{u}}}{\partial t} = \boldsymbol{\nabla}\cdot\boldsymbol{\sigma} + \rho\mathbf{b} \tag{8a}$$

$$\rho\frac{\partial \tilde{\mathbf{u}}}{\partial t} + \rho\mathbf{u}\cdot\boldsymbol{\nabla}\mathbf{u} = 0 \tag{8b}$$

The sum of Equations 8a and 8b gives back Equation 7a. This way, the convective term has been insulated into Equation 8b. In addition, we make the assumption that $\mathbf{u}\cdot\boldsymbol{\nabla}\mathbf{u} \approx \tilde{\mathbf{u}}\cdot\boldsymbol{\nabla}\tilde{\mathbf{u}}$, so that the equations to be solved are:

$$\rho\frac{\partial \mathbf{u}}{\partial t} = \boldsymbol{\nabla}\cdot\boldsymbol{\sigma} + \rho\mathbf{b} + \rho\frac{\partial \tilde{\mathbf{u}}}{\partial t} \tag{9a}$$

$$\frac{\partial \tilde{\mathbf{u}}}{\partial t} + \tilde{\mathbf{u}}\cdot\boldsymbol{\nabla}\tilde{\mathbf{u}} = 0 \tag{9b}$$

The idea is to compute the term $\partial\tilde{\mathbf{u}}/\partial t$ using Equation 9b and then inject it in Equation 9a. In this formulation, the increment of $\tilde{\mathbf{u}}$ can be interpreted as a body force such as the gravity, which is very comfortable. Some very efficient techniques can be used to solve Equation 9b. In this work, we decided to implement the "Back and Forth error compensation and correction" method; it will be detailed later on.

As for the mass balance, Equation 7b has to be modified to be expressed in terms of the pressure p. The objective being to develop a $\mathbf{u}$-p formulation to use Q1P0 finite elements, we need to eliminate the density ρ. For that, we introduce the bulk modulus $K\ [Pa]$, a new parameter of the fluid, whom expression is:

$$K = \rho\frac{\partial p}{\partial \rho} \tag{10}$$

This is a very useful parameter to quantify the variation of the density fluid $\partial\rho$ when there is a change of the isotropic pressure ∂p. We make the assumption that the bulk modulus is constant. $\partial_t\rho$ is expressed as a function of $\partial_t p$ using Equation 10.

$$\partial_t\rho = \frac{\partial\rho}{\partial p}\frac{\partial p}{\partial t} = \frac{\rho}{K}\partial_t p \tag{11}$$

Considering Equations 10 and 11, we can derive the mass conservation equation in terms of the pressure p.

$$\rho\frac{\partial p}{\partial t} + K\boldsymbol{\nabla}\cdot(\rho\mathbf{u}) = 0 \tag{12}$$

There we make the assumption that the density ρ uniform, which means that $\boldsymbol{\nabla}\cdot(\rho\mathbf{u}) = \rho\boldsymbol{\nabla}\cdot\mathbf{u}$.

To determine the range of value of the bulk modulus K for which this assumption is valid, let us rearrange Equation 10 as

$$\frac{K}{\partial p} = \frac{\rho}{\partial \rho} \tag{13}$$

If we assume that the variation of density $\partial\rho$ is negligible compared to the density ρ, it means that

$$\frac{\rho}{\partial \rho} >> 1 \iff \frac{K}{\partial p} >> 1 \tag{14}$$

To sum up, the assumption $\partial\rho \approx 0$ is valid if and only if $K >> \partial p$. The pressure variations need to be much smaller than the bulk modulus. For instance, water has a bulk modulus of 2.2 GPa. If we want the assumption of quasi-incompressible fluid to be valid, we cannot simulate a pressure field superior, in absolute value, than approximately $2.2 \cdot 10^7$ Pa $\approx$ 200 bar. In the practice, it is important to check, for each simulation we are to do, that the pressure variations respect this assumption because, for higher values of the pressure, the fluid cannot be considered as incompressible and the formulation we are doing is not representative of the fluid behaviour.

In a nutshell, the whole set of equations which defines the Navier-Stokes problem we have to solve is the following.

$$\begin{cases} \boldsymbol{\nabla}^{\mathbf{S}}\mathbf{u} = \frac{1}{2}\left(\boldsymbol{\nabla}\mathbf{u} + \boldsymbol{\nabla}\mathbf{u}^T\right) \\ \boldsymbol{\sigma} = -p\mathbf{I} + 2\mu\boldsymbol{\nabla}^{\mathbf{S}}\mathbf{u} \\ \boldsymbol{\nabla}\cdot\boldsymbol{\sigma} + \rho\mathbf{b} + \rho\frac{\partial\tilde{\mathbf{u}}}{\partial t} = \rho\frac{\partial\mathbf{u}}{\partial t} \\ \frac{\partial\tilde{\mathbf{u}}}{\partial t} + \tilde{\mathbf{u}}\cdot\boldsymbol{\nabla}\tilde{\mathbf{u}} = 0 \\ \frac{\partial p}{\partial t} + K\,\boldsymbol{\nabla}\cdot\mathbf{u} = 0 \\ \forall\mathbf{x}\in\Gamma_{\mathbf{u}}, \mathbf{u} = \mathbf{u_0} \end{cases} \tag{15}$$

$\Gamma_{\mathbf{u}}$ refers to the part of the domain boundary where Dirichlet boundary conditions are defined. The unknowns of the problem are the pressure p (scalar function), the velocity field $\mathbf{u}$ (first-order tensor) and the convective velocity field $\tilde{\mathbf{u}}$ (first-order tensor) for a total of 7 unknowns. As for the number of equations, we have 3 from the momentum balance equation, 3 from the convective equations and 1 from the mass balance equation. There are as many unknowns as equations, so we can assume the problem is well-posed and has a solution.

3.2 Weak form

We now need to derive the weak form of the problem in order to implement it in a FEM code. Let us assume that $\boldsymbol{\delta u}$ is the test function for velocities and δp for pressure. $\boldsymbol{\delta u}$ is null over the part of the boundary where velocity Dirichlet boundary conditions are defined. To obtain the weak form of the problem, we multiply each term by the corresponding test function and integrate by part. The idea behind deriving the weak form of the problem is to relax the constraints we have on the velocity field $\mathbf{u}$. If we only consider the Navier-Stokes Problem 15, the velocity field $\mathbf{u}(x, y, z, t)$ needs to be C^2 with respect to space variables since we need to compute its second derivative. First, we compute $\boldsymbol{\nabla}^{\mathbf{S}}\mathbf{u}$ (first spatial derivation) and then $\boldsymbol{\sigma}$ (second spatial derivation). By integrating by parts over the fluid domain, we will relax this constraint.

Let us consider Equation 9a

$$\boldsymbol{\nabla} \cdot \boldsymbol{\sigma} + \rho \mathbf{b} + \rho \frac{\partial \tilde{\mathbf{u}}}{\partial t} = \rho \frac{\partial \mathbf{u}}{\partial t}$$

It is to be multiplied by a test function $\boldsymbol{\delta u}$ and integrated over the domain Ω.

$$\int_\Omega (\boldsymbol{\nabla} \cdot \boldsymbol{\sigma}) \cdot \boldsymbol{\delta u} \, d\Omega + \int_\Omega \rho \, \mathbf{b} \cdot \boldsymbol{\delta u} \, d\Omega + \int_\Omega \rho \frac{\partial \tilde{\mathbf{u}}}{\partial t} \cdot \boldsymbol{\delta u} \, d\Omega = \int_\Omega \rho \frac{\partial \mathbf{u}}{\partial t} \cdot \boldsymbol{\delta u} \, d\Omega$$

Now, to relax the C^2 constraint we have on $\mathbf{u}$, we integrate by parts. For a 2D domain Ω, it reads

$$\int_\Omega (\boldsymbol{\nabla} \cdot \boldsymbol{\sigma}) \cdot \boldsymbol{\delta u} \, d\Omega = \int_\Omega \boldsymbol{\nabla} \cdot (\boldsymbol{\sigma} \cdot \boldsymbol{\delta u}) \, d\Omega - \int_\Omega \boldsymbol{\sigma} : \boldsymbol{\nabla}^{\boldsymbol{S}} \boldsymbol{\delta u} \, d\Omega \tag{16}$$

Then, we apply the Divergence Theorem to pass from an integral over the domain to an integral over the border of the domain, which would enable us to include potential Neumann boundary conditions in the weak form formulation.

$$\int_\Omega \boldsymbol{\nabla} \cdot (\boldsymbol{\sigma} \cdot \boldsymbol{\delta u}) \, d\Omega = \int_{\partial\Omega} \boldsymbol{\sigma} \cdot \boldsymbol{\delta u} \cdot \mathbf{n} \, d\Gamma \tag{17}$$

where $\mathbf{n}$ stands for the normal of the domain border.

By definition, we know that $\boldsymbol{\delta u}_{|\Gamma_u} = \mathbf{0}$, which means that:

$$\int_{\partial\Omega} \boldsymbol{\sigma} \cdot \boldsymbol{\delta u} \cdot \mathbf{n} \, d\Gamma = \int_{\partial\Omega\setminus\Gamma_u} \boldsymbol{\sigma} \cdot \boldsymbol{\delta u} \cdot \mathbf{n} \, d\Gamma$$

It is not necessary any more to integrate over the part of the boundary where Dirichlet boundary conditions are applied.

Combining all the equations, we finally get the weak form of the momentum balance equation.

$$\int_{\partial\Omega\setminus\Gamma_u} \boldsymbol{\sigma} \cdot \boldsymbol{\delta u} \cdot \mathbf{n} \, d\Gamma - \int_\Omega \boldsymbol{\sigma} : \boldsymbol{\nabla}^{\boldsymbol{S}} \boldsymbol{u} \, d\Omega + \int_\Omega \rho \, \mathbf{b} \cdot \boldsymbol{\delta u} \, d\Omega + \int_\Omega \rho \frac{\partial \tilde{\mathbf{u}}}{\partial t} \cdot \boldsymbol{\delta u} \, d\Omega = \int_\Omega \rho \frac{\partial \mathbf{u}}{\partial t} \cdot \boldsymbol{\delta u} \, d\Omega$$

The weak form of the mass balance equation is much easier to obtain since we just need to multiply by the pressure test function and integrate over the whole domain Ω. Finally, the weak formulation of the Navier-Stokes problem is stated as:

Find $\mathbf{u}$ and p, such that for all $\boldsymbol{\delta u}$ and δp with $\boldsymbol{\delta u}_{|\Gamma_u} = \mathbf{0}$,

$$\int_\Omega \boldsymbol{\sigma} : \boldsymbol{\nabla^S} \boldsymbol{\delta u} \, d\Omega + \int_\Omega \rho \frac{\partial \mathbf{u}}{\partial t} \cdot \boldsymbol{\delta u} \, d\Omega = \int_\Omega \rho \mathbf{b} \cdot \boldsymbol{\delta u} \, d\Omega + \int_{\partial\Omega \setminus \Gamma_u} \boldsymbol{\sigma} \cdot \boldsymbol{\delta u} \cdot \mathbf{n} \, d\Gamma + \int_\Omega \rho \frac{\partial \tilde{\mathbf{u}}}{\partial t} \cdot \boldsymbol{\delta u} \, d\Omega \tag{18a}$$

$$\int_\Omega \delta p \frac{\partial p}{\partial t} \, d\Omega + \int_\Omega K \delta p \nabla \cdot \mathbf{u} \, d\Omega = 0 \tag{18b}$$

$$\mathbf{u}_{|\Gamma_u} = \mathbf{u_0} \tag{18c}$$

Only the first spatial derivative of the velocity field appears in this weak form, which means the constraints on the function $\mathbf{u}(x, y, z, t)$ we look for are not as strong as before. We do not need to derive a weak form for the convective equation 9b since it will be solved using another method.

3.3 Voigt notation

We introduce the "engineering" Voigt notation and rewrite the 2 previous equations using these new variables. This formulation is much more comfortable because it only deals with vectors, which are much easier to implement in a FEM code.

From now on, we make the assumption of a 2D problem, the main idea being to make the numerical implementation easier later on. As we will need to perform many test cases with the solver (main objective of this Proof of Concept), it is really worth it to have a fast code and, working in 2D will considerably lower the computational cost of the simulation. However, all what we set out in this section can be easily extended to the 3D case.

$$\boldsymbol{\nabla^S u} = \begin{pmatrix} \frac{\partial u}{\partial x} & \frac{\partial v}{\partial y} & \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \end{pmatrix}^T$$

$$\boldsymbol{\sigma} = \begin{pmatrix} \sigma_x & \sigma_y & \gamma_{xy} \end{pmatrix}^T$$

It is necessary to rewrite the momentum balance equation in terms of these new variables. We do not need to compute any more the double dot product as in $\boldsymbol{\sigma} : \boldsymbol{\nabla^S \delta u}$. This notation only requires vector multiplication, which has a lower computational cost too.

The Weak Form of the Navier-Stokes problem in Voigt notation is:

Find $\mathbf{u}$ and p, such that for all $\boldsymbol{\delta u}$ and δp with $\boldsymbol{\delta u}_{|\Gamma_u} = \mathbf{0}$,

$$\int_\Omega \left(\boldsymbol{\nabla^S \delta u}\right)^{\mathbf{T}} \boldsymbol{\sigma} \, d\Omega + \int_\Omega \rho \boldsymbol{\delta u}^T \frac{\partial \mathbf{u}}{\partial t} \, d\Omega = \int_\Omega \rho \boldsymbol{\delta u}^{\mathbf{T}} \mathbf{b} \, d\Omega + \int_{\partial\Omega \setminus \Gamma_u} \boldsymbol{\delta u}^{\mathbf{T}} \mathbf{t} \, d\Gamma + \int_\Omega \rho \boldsymbol{\delta u}^T \frac{\partial \tilde{\mathbf{u}}}{\partial t} \, d\Omega \tag{19a}$$

$$\int_\Omega \delta p \frac{\partial p}{\partial t} \, d\Omega + \int_\Omega K \delta p \boldsymbol{\nabla^T} \mathbf{u} \, d\Omega = 0 \tag{19b}$$

$$\mathbf{u}_{|\Gamma_u} = \mathbf{u_0} \tag{19c}$$

4 Q1P0 Galerkin discretisation

In common engineering problems, analytical solutions for the Navier-Stokes problem can be hardly found. This difficulty is likely overcome by simplifying the problem to find an approximation of the analytical solution. Hence, we are going to interpolate the analytical solution using a certain number of points in the domain. This implies to pass from a continuous problem (find the expressions of continuous functions), which is really hard, to a discrete problem (find the value of the analytical solution at certain points), much easier to solve. Furthermore, one needs to define the expression of the test functions $\boldsymbol{\delta u}$ and $\boldsymbol{\delta p}$. The main idea behind Galerkin discretisation is to use the interpolation functions of $\mathbf{u}$ and p as test functions.

4.1 Global discretisation

Let us consider a set of N_{node} points, characterised by their coordinates $\left\{\mathbf{x}_i | \, i \in \left[|1; N_{node}|\right]\right\}$, and a set of N_{node} interpolation functions $\left\{N_i | \, i \in \left[|1; N_{node}|\right]\right\}$, each one associated to one particular node (there are as many interpolation functions as points). We approximate the analytical solution of the Navier-Stokes weak form problem using the interpolation functions N_i.

$$\mathbf{u}(\mathbf{x}, t) \approx \mathbf{u}^h(\mathbf{x}, t) = \sum_{i=1}^{N_{node}} \mathbf{u_i}(t) N_i(\mathbf{x})$$

$$p(\mathbf{x}, t) \approx p^h(\mathbf{x}, t) = \sum_{i=1}^{N_{node}} p_i(t) N_i(\mathbf{x})$$

where $\mathbf{u}^h$, p^h are the approximation of the continuous solution and $\mathbf{u}_i$, p_i the nodal values of the approximated solution at point $\mathbf{x}_i$.

Figures 15 and 16 present examples of interpolation functions in 1D and 2D meshes. They are piece-wise linear functions (for 2D triangular meshes) with $N_i(\mathbf{x_i}) = 1$. For instance, in Figure 15, $N_1(x_0) = N_1(x_2) = 0$ and $N_1(x_1) = 1$.

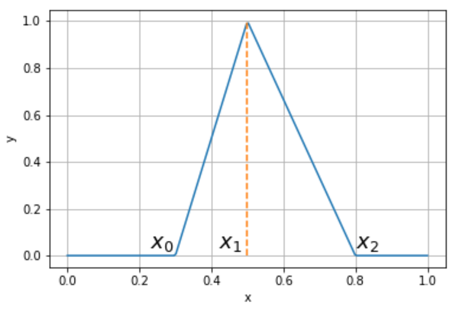

Figure 15: Example of interpolation function in a 1D mesh

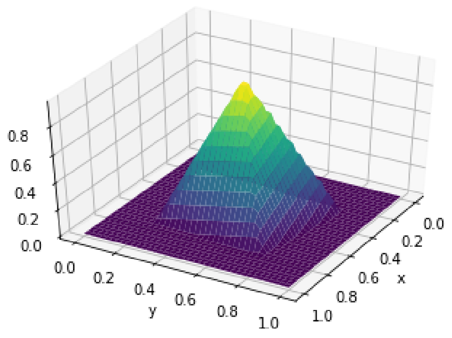

Figure 16: Example of interpolation function in a 2D mesh

The next step is to distinguish between the points which are on the surface of the domain where a Dirichlet boundary condition is applied and those which are not. Let us define a set $\mathcal{D}$ such as:

$$\mathcal{D} = \{i, \mathbf{x}_i \in \Gamma_u\}$$

The set $\mathcal{D}$ contains the indices of the points which are on a boundary where a Dirichlet boundary condition is applied. As a consequence, we have:

$$\mathbf{u}^h = \sum_{i \in \mathcal{D}} \mathbf{u}_i(t) N_i(\mathbf{x}) + \sum_{i \notin \mathcal{D}} \mathbf{u}_i(t) N_i(\mathbf{x})$$

To finally discretise Equations 19a and 19b, we use the interpolation functions N_i as test functions (Galerkin approach) and we substitute the values of $\mathbf{u}^h$ and p^h into the Navier-Stokes problem weak form. This would lead to a linear system of equations, which we would have to solve. These are the basics of the Galerkin discretisation method.

However this point-based vision is not very easy to manipulate because, within this framework, it is hard to define the expression of the interpolation function. For instance, if we consider Figures 15 and 16, we have to deal with piece-wise linear functions, which have different expressions depending on the area of the mesh we are working in. For instance, in Figure 15, let us say that, for a given x, we want to compute $N_1(x)$. We have to distinguish between $x < x_0$, $x_0 < x < x_1$, $x_1 < x < x_2$ and $x > x_2$, i.e. we have four different cases and four different expressions for N_1. Moreover, to find the expression of such a hat function, we need to determine the neighbours of the associated point. For an unstructured mesh, this can be very difficult.

For all these reasons, to make the discretisation easier, most of the time, we adopt an element-oriented vision, also known as local discretisation. In other words, we divide the original domains into sub-domains (the so called elements of the mesh), we derive the linear system of equations for one element and, through an assembly process we will detail afterwards, we determine the linear system of equations corresponding to the entire discrete domain.

The two points of view (point-oriented and element-oriented) finally would give the same results. However, the element-oriented point of view provides us with a much simpler framework. From now on, we adopt this element-centred point of view.

4.2 Element-based discretisation

The degrees of freedom (DOF) of the Q1P0 finite element are the velocity components at the 4 nodes and the pressure in the centre of the element, which is assumed to be constant in all the interior, leading to discontinuous pressure between elements. However, this is not a problem because, to derive the weak form of equations, we only require the pressure field $p(x, y, z, t)$ to be square-integrable on the domain. Thus, there is no restriction as for its first derivative.

The vector $\mathbf{U}$ contains the 8 degrees of freedom (DOF) of one 2D element. This could be perfectly generalised to 3D but, for the sake of clarity, we only consider the 2D case.

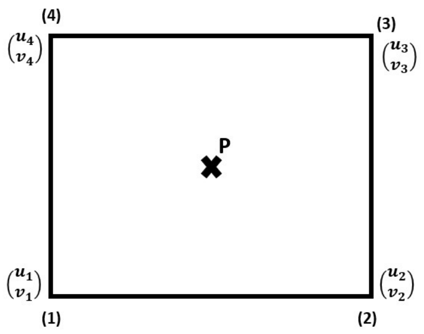

Figure 17: Q1P0 element DOF

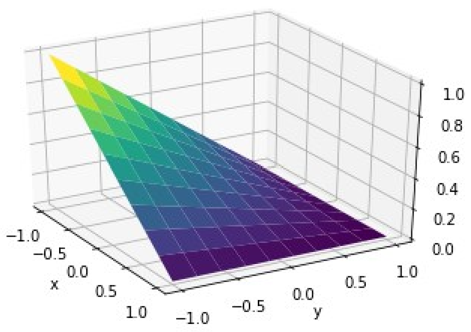

Figure 18: Reference Q1P0 element first shape function

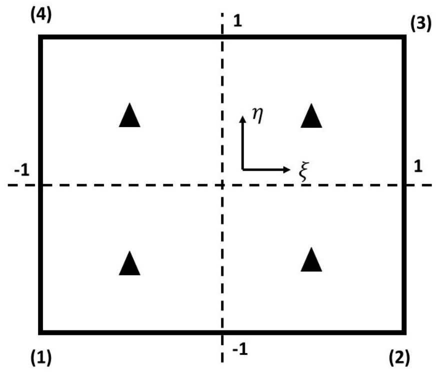

Figure 19: Reference Q1P0 element and Gauss integration points

Coordinates of the Gauss point	Weight
$-\frac{1}{\sqrt{3}};\frac{1}{\sqrt{3}}$	1
$\frac{1}{\sqrt{3}};\frac{1}{\sqrt{3}}$	1
$\frac{1}{\sqrt{3}};-\frac{1}{\sqrt{3}}$	1
$-\frac{1}{\sqrt{3}};-\frac{1}{\sqrt{3}}$	1

Figure 20: Gauss integration points and weights

$$\mathbf{U} = \begin{pmatrix} u_1 & v_1 & u_2 & v_2 & u_3 & v_3 & u_4 & v_4 \end{pmatrix}^T$$

The velocity field within the element is to be approximated using the value of the velocity at the 4 nodes of the element and the associated shape function. The value of the shape function is 1 at the associated node and 0 at the other 3 nodes of the element. For instance, N_1 is equal to 1 at $(x_1; y_1)$ and is null at $(x_2; y_2)$, $(x_3; y_3)$ and $(x_4; y_4)$ (see Figure 18 for a more representative illustration[8]).

We interpolate the velocity field $\mathbf{u}$ using the DOFs of the elements and the shape functions.

$$\begin{cases} u = u_1N_1 + u_2N_2 + u_3N_3 + u_4N_4 \\ v = v_1N_1 + v_2N_2 + v_3N_3 + v_4N_4 \end{cases}$$

The expressions above comply with the restriction $\mathbf{u}(x_1, y_1) = \mathbf{u_1}$, $\mathbf{u}(x_2, y_2) = \mathbf{u_2}$, $\mathbf{u}(x_3, y_3) = \mathbf{u_3}$ and $\mathbf{u}(x_4, y_4) = \mathbf{u_4}$. This leads to the following matrix equation, which is a much more compact way to write the interpolation.

[8]In Figure 18 is only plotted the first shape functions. N_2, N_3 and N_4 are totally symmetric.

$$\mathbf{u} = \mathbf{NU} = \begin{pmatrix} N_1 & 0 & N_2 & 0 & N_3 & 0 & N_4 & 0 \\ 0 & N_1 & 0 & N_2 & 0 & N_3 & 0 & N_4 \end{pmatrix} \mathbf{U}$$

Using this expression, the temporal derivative of $\mathbf{u}$ can be obtained as $\partial_t \mathbf{u} = \mathbf{N}(x,y)\,\partial_t \mathbf{U}$[9].

Then, we compute the derivatives of $\mathbf{u}$

$$\begin{cases} \frac{\partial u}{\partial x} = u_1 \frac{\partial N_1}{\partial x} + u_2 \frac{\partial N_2}{\partial x} + u_3 \frac{\partial N_3}{\partial x} + u_4 \frac{\partial N_4}{\partial x} \\ \frac{\partial u}{\partial y} = u_1 \frac{\partial N_1}{\partial y} + u_2 \frac{\partial N_2}{\partial y} + u_3 \frac{\partial N_3}{\partial y} + u_4 \frac{\partial N_4}{\partial y} \\ \frac{\partial v}{\partial x} = v_1 \frac{\partial N_1}{\partial x} + v_2 \frac{\partial N_2}{\partial x} + v_3 \frac{\partial N_3}{\partial x} + v_4 \frac{\partial N_4}{\partial x} \\ \frac{\partial v}{\partial y} = v_1 \frac{\partial N_1}{\partial y} + v_2 \frac{\partial N_2}{\partial y} + v_3 \frac{\partial N_3}{\partial y} + v_4 \frac{\partial N_4}{\partial y} \end{cases}$$

which can be expressed in matrix form

$$\boldsymbol{\nabla^S u} = \mathbf{BU} = \begin{pmatrix} \frac{\partial N_1}{\partial x} & 0 & \frac{\partial N_2}{\partial x} & 0 & \frac{\partial N_3}{\partial x} & 0 & \frac{\partial N_4}{\partial x} & 0 \\ 0 & \frac{\partial N_1}{\partial y} & 0 & \frac{\partial N_2}{\partial y} & 0 & \frac{\partial N_3}{\partial y} & 0 & \frac{\partial N_4}{\partial y} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial y} & \frac{\partial N_2}{\partial x} & \frac{\partial N_3}{\partial y} & \frac{\partial N_3}{\partial x} & \frac{\partial N_4}{\partial y} & \frac{\partial N_4}{\partial x} \end{pmatrix} \mathbf{U}$$

To have a complete formulation, we need to express the stress tensor in terms of the Voigt notation.

$$\boldsymbol{\sigma} = -P_c \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} + \mathbb{C}\,\boldsymbol{\nabla^S u}\,, \text{ with } \mathbb{C} = \begin{pmatrix} 2\mu & 0 & 0 \\ 0 & 2\mu & 0 \\ 0 & 0 & \mu \end{pmatrix}$$

where P_c is the value of the pressure in the centre of the element.

We introduce all these expressions in Equation 19a but, first of all, we have to express the test functions $\boldsymbol{\delta u}$ and $\boldsymbol{\nabla^S \delta u}$. In the Galerkin approach, the velocity test function and its gradient are $\boldsymbol{\delta u} = \mathbf{N}\,\boldsymbol{\delta U}$ and $\boldsymbol{\nabla^S \delta u} = \mathbf{B}\,\boldsymbol{\delta U}$ because we use the same interpolation for the test functions. We insert these expressions into Equation 19a to obtain:

[9]The shape functions are time-independent

$$\forall\,\boldsymbol{\delta U} \neq \mathbf{0},\ \int_\Omega \boldsymbol{\delta U^T}\mathbf{B^T}\left(-P_c\begin{pmatrix}1\\1\\0\end{pmatrix} + \mathbb{C}\,\boldsymbol{\nabla^S u}\right) + \int_\Omega \rho\,\boldsymbol{\delta U^T}\mathbf{N^T}\mathbf{N}\,\partial_t\mathbf{U} = \int_\Omega \rho\,\boldsymbol{\delta U^T}\mathbf{N^T}\mathbf{N}\,\partial_t\tilde{\mathbf{U}} + \int_\Omega \rho\,\boldsymbol{\delta U^T}\mathbf{N^T}\mathbf{b} + \int_{\partial\Omega\backslash\Gamma_u} \boldsymbol{\delta U^T}\mathbf{N^T}\mathbf{t}\,d\Gamma$$

Moreover, the previous equation is true for every $\boldsymbol{\delta U}$ which complies with the Dirichlet boundary conditions, which means that:

$$\int_\Omega \mathbf{B^T}\left(-P_c\begin{pmatrix}1\\1\\0\end{pmatrix} + \mathbb{C}\mathbf{B}\mathbf{U}\right) + \int_\Omega \rho\,\mathbf{N^T}\mathbf{N}\,\partial_t\mathbf{U} = \int_\Omega \rho\,\mathbf{N^T}\mathbf{N}\,\partial_t\tilde{\mathbf{U}} + \int_\Omega \rho\,\mathbf{N^T}\mathbf{b} + \int_{\partial\Omega\backslash\Gamma_u} \mathbf{N^T}\mathbf{t} \quad (21)$$

As $\mathbf{U}$ and P_c do not depend on space variables, Equation 22 is a direct consequence of previous development.

$$-\left(\int_\Omega \mathbf{B^T}\begin{pmatrix}1\\1\\0\end{pmatrix}\right)P_c + \left(\int_\Omega \mathbf{B^T}\mathbb{C}\mathbf{B}\right)\mathbf{U} + \left(\int_\Omega \rho\mathbf{N^T}\mathbf{N}\right)\partial_t\mathbf{U} = \left(\int_\Omega \rho\mathbf{N^T}\mathbf{N}\right)\partial_t\tilde{\mathbf{U}} + \left(\int_\Omega \rho\mathbf{N^T}\mathbf{b}\right) + \left(\int_{\partial\Omega} \mathbf{N^T}\mathbf{t}\right) + \mathbf{q^T}\mathbf{U} \quad (22)$$

$\mathbf{q}$ and $\mathbf{t}$ are first-order tensors containing respectively the nodal and boundary forces which are applied on the element.

We define a discrete gradient operator.

$$\mathbf{G} = \begin{pmatrix}\frac{\partial N_1}{\partial x} & \frac{\partial N_1}{\partial y} & \frac{\partial N_2}{\partial x} & \frac{\partial N_2}{\partial y} & \frac{\partial N_3}{\partial x} & \frac{\partial N_3}{\partial y} & \frac{\partial N_4}{\partial x} & \frac{\partial N_4}{\partial y}\end{pmatrix}^T$$

Considering the matrices product, we can easily prove that:

$$\mathbf{B^T}\begin{pmatrix}1\\1\\0\end{pmatrix} = \mathbf{G}$$

Remains Equation 18b to be discretised. The pressure test function is $\delta p = 1$ as we only consider the value of the pressure P_c in the centre of the Q1P0 finite element.

First of all, we need to express the divergence of $\mathbf{u}$.

$$\boldsymbol{\nabla} \cdot \mathbf{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = \mathbf{G^T U}$$

Then, we substitute this expression of the divergence in Equation 19b.

$$\left(\int_\Omega d\Omega \right) \dot{P_c} + \left(\int_\Omega K \mathbf{G^T} \, d\Omega \right) \mathbf{U} = 0$$

Let us define some useful matrices.

$$\mathbf{K_{uu,loc}} = \int_\Omega \mathbf{B^T} \mathbb{C} \mathbf{B} \, d\Omega \qquad \mathbf{M_{uu,loc}} = \int_\Omega \rho \mathbf{N^T N} \, d\Omega \qquad \mathbf{D_{loc}} = \int_\Omega \mathbf{G} \, d\Omega$$

$$\mathbf{F} = \int_\Omega \rho \mathbf{N^T b} \, d\Omega + \int_{\partial\Omega} \mathbf{N^T t} \, d\Gamma + \mathbf{q^T U}$$

Expressing the previous equations using the notation above, we get:

$$\boxed{\begin{array}{c} \mathbf{K_{uu,loc}} \, \mathbf{U} + \mathbf{M_{uu,loc}} \, \dot{\mathbf{U}} - \mathbf{D_{loc}} \, P = \mathbf{F} + \mathbf{M_{uu,loc}} \dot{\tilde{\mathbf{U}}} \\ \frac{A_e}{K} \dot{P_c} + \mathbf{D^T_{loc}} \mathbf{U} = 0 \end{array}} \tag{23}$$

The matrices $\mathbf{K_{uu,loc}}$, $\mathbf{M_{uu,loc}}$, $\mathbf{D_{loc}}$ are computed using Gauss integration techniques over the reference element depicted in Figure 19. The $\blacktriangle$ symbols represent the four Gauss-quadrature points. $\mathbf{K_{uu,loc}}$, $\mathbf{M_{uu,loc}}$ and $\mathbf{D_{loc}}$ are respectively the local stiffness matrix, the local mass matrix and the local gradient operator. $\mathbf{D^T_{loc}}$ can be considered as the local divergence operator.

4.3 Isoparametric element integration

To efficiently compute the element integrals, an isoparametric reference element is to be used. This way, we do not have to express the shape functions for each element of the mesh. We work using this reference element and, with a certain mapping, we are able to come back to the "physical element" and compute the stiffness, mass and gradient matrices. This a key point in the FEM implementation because an error in this part of the process has huge consequences.

Let us call ξ and η the space variables of the reference element and Ω_{ref} the domain of the reference element. For the sake of simplicity, let us focus on the computation of the stiffness matrix. Once the computation process of this particular integral is understood, the computation of the other ones is immediate.

First of all, it is necessary to build a mapping between the reference element and the "physical" one, i.e., find a transformation $x = x(\xi, \eta)$ and $y = y(\xi, \eta)$ to pass from the reference coordinates to x and y which are associated to our original element. This mapping depends on the type of element we are implementing. The Q1P0 element is an isoparametric one, which means that we will reuse the exact same shape functions we previously defined for the DOF.

$$x = x_1 N_1(\xi, \eta) + x_2 N_2(\xi, \eta) + x_3 N_3(\xi, \eta) + x_4 N_4(\xi, \eta)$$
$$y = y_1 N_1(\xi, \eta) + y_2 N_2(\xi, \eta) + y_3 N_3(\xi, \eta) + y_4 N_4(\xi, \eta)$$

It can be easily checked that the point $(-1, -1)$ of the reference element (see Figure 19), which is labelled as number 1, is "sent" to (x_1, y_1), which is labelled as number 1 of the physical element because $N_1(-1, -1) = 1$, $N_1(1, -1) = 0$, $N_1(1, 1) = 0$ and $N_1(-1, 1) = 0$ (we defined the shape functions this way). This demonstration can be repeated for all the points of the reference element.

Once we defined this transformation, the Jacobian matrix $\mathbf{J}$ is to be introduced. It is possible to directly relate $d\Omega$ and $d\Omega_{ref}$ using its determinant.

$$\mathbf{J} = \begin{pmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{pmatrix} \qquad d\Omega = |\mathbf{J}| \, d\Omega_{ref} \tag{24}$$

where $|\mathbf{J}|$ is the determinant of $\mathbf{J}$. Now we are able to transfer the computation of the integral from the physical elements to the reference one.

$$\int_{\Omega} \mathbf{B^T} \mathbb{C} \mathbf{B} \, d\Omega = \int_{\Omega_{ref}} \mathbf{B^T}(x(\xi, \eta), y(\xi, \eta)) \, \mathbb{C} \, \mathbf{B}(x(\xi, \eta), y(\xi, \eta)) \, |\mathbf{J}(\xi, \eta)| \, d\Omega_{ref}$$

Up to now, we do not know the expression of matrix $\mathbf{B}$ because the information we have is the derivative of the shape function with respect to ξ and η and, let us remember that the entries of $\mathbf{B}$ are the derivatives of the shape functions with respect to x and y. The following step is to express $\mathbf{B}(x(\xi, \eta), y(\xi, \eta))$. To do this, we need to apply the chain rule.

$$\forall i \in [|1, 4|], \; N_i(x, y) = N_i(x(\xi, \eta), y(\xi, \eta))$$

$$\forall i \in [|1, 4|], \; \frac{\partial N_i}{\partial x} = \frac{\partial N_i}{\partial \xi} \frac{\partial \xi}{\partial x} + \frac{\partial N_i}{\partial \eta} \frac{\partial \eta}{\partial x}$$

$$\forall i \in [|1, 4|], \; \frac{\partial N_i}{\partial y} = \frac{\partial N_i}{\partial \xi} \frac{\partial \xi}{\partial y} + \frac{\partial N_i}{\partial \eta} \frac{\partial \eta}{\partial y}$$

The terms $\frac{\partial \xi}{\partial x}$, $\frac{\partial \eta}{\partial x}$, $\frac{\partial \xi}{\partial y}$ and $\frac{\partial \eta}{\partial y}$ are directly related to the inverse of the Jacobian matrix. Since the transformation from the reference element to the original one is a mapping (bijective transformation), the inverse of the Jacobian exists and can be computed easily and accurately if the original quadrilateral is not too deformed.

To finish the computation of the integral, a 4-point Gauss quadrature is implemented.

$$\int_{\Omega_{ref}} \mathbf{B^T}(x(\xi,\eta),y(\xi,\eta))\,\mathbb{C}\,\mathbf{B}(x(\xi,\eta),y(\xi,\eta))\,|\mathbf{J}(\xi,\eta)|\,d\Omega_{ref} = \sum_{\boldsymbol{\xi_g}} \boldsymbol{B^T}(\boldsymbol{\xi_g})\mathbb{C}\mathbf{B}(\boldsymbol{\xi_g})|\mathbf{J}(\boldsymbol{\xi_g})|\omega_g \quad (25)$$

The integral over Ω_{ref} is approximated by the sum over the four Gauss integration points whose coordinates are given by $\boldsymbol{\xi_g}$.

4.4 Numerical expression

In order to implement this formulation within a FEM code, it is necessary to numerically express the matrices **N**, **B** and **D**.

We are to compute the shape functions associated to each node of the quadrilatere and the subsequent partial derivatives.

$$N_1(\xi,\eta) = \tfrac{1}{4}(\xi-1)(\eta-1) \quad \frac{\partial N_1}{\partial \xi} = \tfrac{1}{4}(\eta-1) \quad \frac{\partial N_1}{\partial \eta} = \tfrac{1}{4}(\xi-1)$$

$$N_2(\xi,\eta) = -\tfrac{1}{4}(\xi+1)(\eta-1) \quad \frac{\partial N_2}{\partial \xi} = -\tfrac{1}{4}(\eta-1) \quad \frac{\partial N_2}{\partial \eta} = -\tfrac{1}{4}(\xi+1)$$

$$N_3(\xi,\eta) = \tfrac{1}{4}(\xi+1)(\eta+1) \quad \frac{\partial N_3}{\partial \xi} = \tfrac{1}{4}(\eta+1) \quad \frac{\partial N_3}{\partial \eta} = \tfrac{1}{4}(\xi+1)$$

$$N_4(\xi,\eta) = -\tfrac{1}{4}(\xi-1)(\eta+1) \quad \frac{\partial N_4}{\partial \xi} = -\tfrac{1}{4}(\eta+1) \quad \frac{\partial N_4}{\partial \eta} = -\tfrac{1}{4}(\xi-1)$$

We choose a 4-point Gauss integration to perform the numerical calculation of the different integrals. Using isoparametric element, we can compute the Jacobian of the geometric transformation between the physical configuration and the reference one.

$$\frac{\partial x}{\partial \xi} = \tfrac{1}{4}\big((x_1-x_2+x_3-x_4)\eta - x_1+x_2+x_3-x_4\big)$$
$$\frac{\partial x}{\partial \eta} = \tfrac{1}{4}\big((x_1-x_2+x_3-x_4)\xi - x_1-x_2+x_4+x_3\big)$$

$$\frac{\partial y}{\partial \xi} = \tfrac{1}{4}\big((y_1-y_2+y_3-y_4)\eta - y_1+y_2+y_3-y_4\big)$$
$$\frac{\partial x}{\partial \eta} = \tfrac{1}{4}\big((y_1-y_2+y_3-y_4)\xi - y_1-y_2+y_4+y_3\big)$$

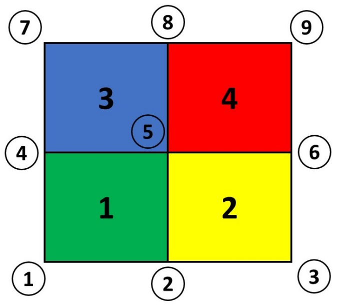

Figure 21: Simple mesh example

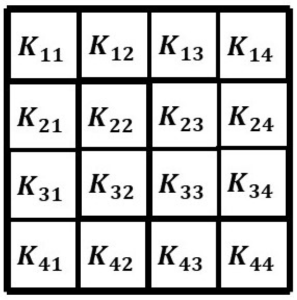
K_{11} K_{12} K_{13} K_{14}
K_{21} K_{22} K_{23} K_{24}
K_{31} K_{32} K_{33} K_{34}
K_{41} K_{42} K_{43} K_{44}

Figure 22: Block decomposition of local stiffness matrix

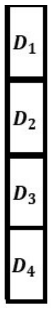

Figure 23: Block decomposition of local gradient matrix

4.5 Assembly of a B-bar elements mesh

Equations 23 are the equations for 1 element. Of course, we want to study a more complex system with several elements in a structured mesh. To do that, we need to assemble the elemental matrices $\mathbf{K_{uu,loc}}$, $\mathbf{M_{uu,loc}}$ and $\mathbf{D}$ to get the following global linear system of equations:

$$\mathbf{M_{UU}\dot{U}} + \mathbf{K_{UU}U} - \mathbf{DP} = \mathbf{F} + \mathbf{M_{UU}\dot{\tilde{U}}} \tag{26a}$$

$$\mathbf{M_{PP}\dot{P}} + \mathbf{D^T U} = \mathbf{0} \tag{26b}$$

For the sake of clarity, we are going to explain the assembly process on a simple example as the one presented in Figure 21. The mesh is composed by 4 Q1P0 elements. Each element is labelled from 1 to 4. There are 9 nodes, labelled from 1 to 9. It is very important to distinguish between the local numbering (in each element) and the global numbering of the nodes. For instance, node 5 (global numbering) is the node 3 of element 1 but the node 4 of element 2.

In the whole process of assembling, we do not take into account the boundary conditions of the problem. This is a subsequent step.

4.5.1 Assembly of the stiffness and mass matrices

The stiffness matrix $\mathbf{K_{uu,loc}}$ and mass matrix $\mathbf{M_{uu,loc}}$ will be assembled in an usual way. Their size is $(8;8)$ and they can be divided into 16 different blocks (see Figure 22). In our simple example, all the elements are equal so the local stiffness matrices are equal but it is not always the case. The size of the global stiffness and mass matrices is $(2N_{node};2N_{node})$ where N_{node} is the number of nodes of the mesh.

To assemble the local stiffness matrices into the global one, we need to loop over the 4 elements of the mesh. For each element, we place the 16 blocks we previously described taking into account the local numbering of nodes in the element and the global one. Figure 24 presents the first step of the loop. It is mandatory to carefully implement it because, for instance, the node 5 (in global numbering) is the third node of element 1. As a consequence, the block $\mathbf{K_{loc,13}}$ goes to the block $\mathbf{K_{15}}$ of the global stiffness matrix. We repeat the same set of operations for the 4 elements (see Figures 24, 25, 26 and 27).

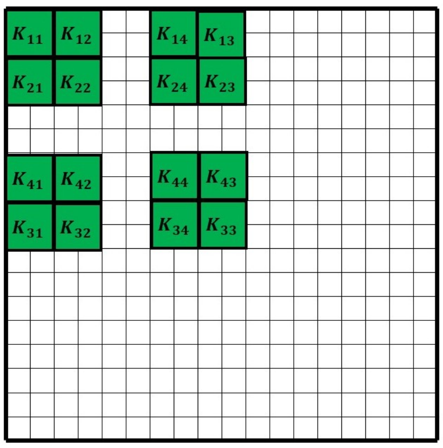

Figure 24: First step of the stiffness matrix assembly process. Loop over the first element.

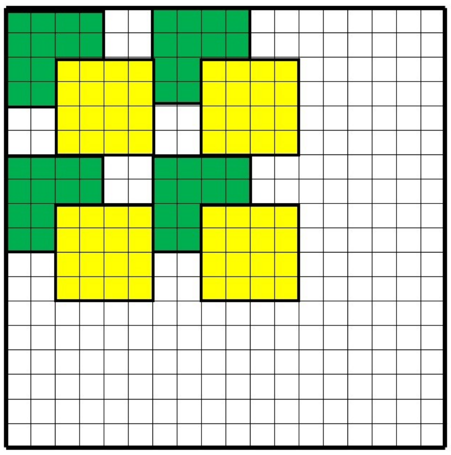

Figure 25: Second step of the stiffness matrix assembly process. Loop over the second element.

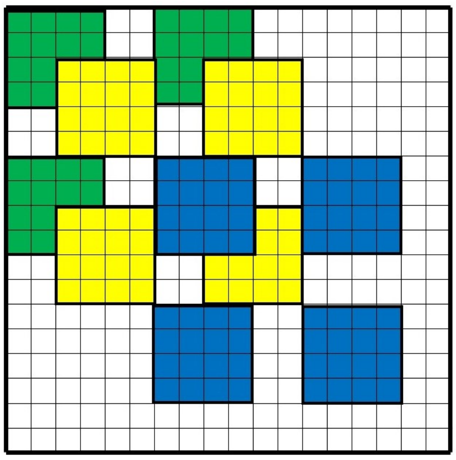

Figure 26: Third step of the stiffness matrix assembly process. Loop over the third element.

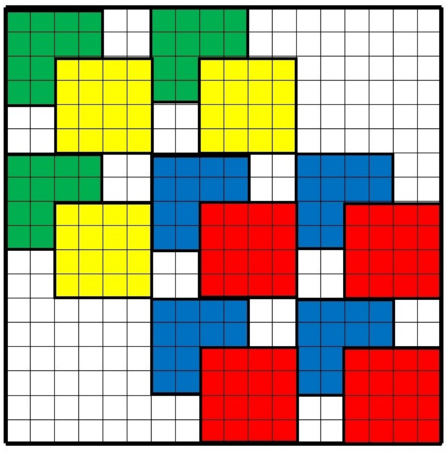

Figure 27: Fourth step of the stiffness matrix assembly process. Loop over the fourth element.

In the end, the stiffness and mass matrices are quite sparse and have a tridiagonal structure, which will lead to coding and memory optimisations. More generally, the structure of the stiffness matrix and mass matrices depends on the global and local numbering. If we had labelled the nodes another way, the structure of the stiffness and mass matrices would have been different. This highlights the importance of the FEM code mesher. If the numbering of the nodes is efficient, the optimisation of the stiffness and mass matrices can lead to huge reduction of the computational cost of the simulation.

Reasoning in terms of blocks makes the implementation easier because, for each node of the element, it is important to remember there are 2 DOF. If we had to implement the same element in 3D, the blocks would be of size $(3;3)$ because there will be 3 DOF.

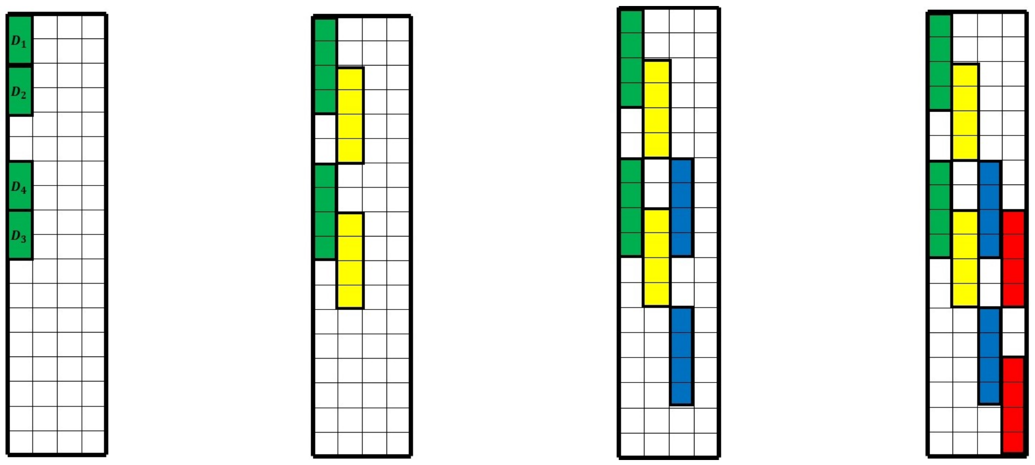

Figure 28: The 4 steps of the discrete gradient operator assembly process

4.5.2 Discrete gradient operator assembly

The process is quite different to get $\mathbf{D_{global}}$ from the $\mathbf{D_{local}}$ matrices. Let us say the number of elements is $N_{element}$ and the number of nodes of the mesh is N_{node}. The size of $\mathbf{D_{global}}$ will be $(2N_{node}, N_{element})$, to be consistent with the matrices product. In our particular example, the size of D is $(18, 4)$ (see Figure 28) because there are 9 nodes and 4 elements in the mesh.

To assemble $\mathbf{D}$, it is necessary to compute the column vectors $\mathbf{D_{local}}$, which, in our case, are equal because the elements are the same, and put its different components in the column of $\mathbf{D_{global}}$ which corresponds to the number of the element we are looping on. For instance, let us study the first step of the loop (see Figure 28). As it corresponds to the first element, we will work in the first column of matrix $\mathbf{D_{global}}$. The node 5 (in global numbering) is the third node of the element (in local numbering). As a consequence, the block $\mathbf{D_{local,3}}$ goes to the block $\mathbf{D_5}$ of the global matrix $\mathbf{D}$ and so on for the rest of the nodes and elements. We change of columns each time we change of elements.

4.5.3 Pressure mass matrix assembly

Because we use Q1P0 elements, the global pressure "mass matrix" $\mathbf{M_{PP}}$ will be very easy to compute. It is a diagonal matrix of size $(N_{element}, N_{element})$ and the term i on the diagonal will be ${}^{A_{e,i}}/_{K_i}$ where $A_{e,i}$ is the area of the element i and K_i the bulk modulus of the element i. $\mathbf{M_{PP,\,global}}$ being diagonal will be a great advantage for time integration; this is a key feature when using Q1P0 elements.

4.5.4 Global assembly

It is possible to go a step further in the assembly process. The total number of DOF (taking into account pressure DOFs and velocity nodal values) is $2N_{node} + N_{element}$. We can define a vector $\mathbb{Y}$ containing all the DOF of the problem (first the velocity nodal values, then the pressure ones) and define from Equations 26a and 26b the following Ordinary Differential Equation (ODE):

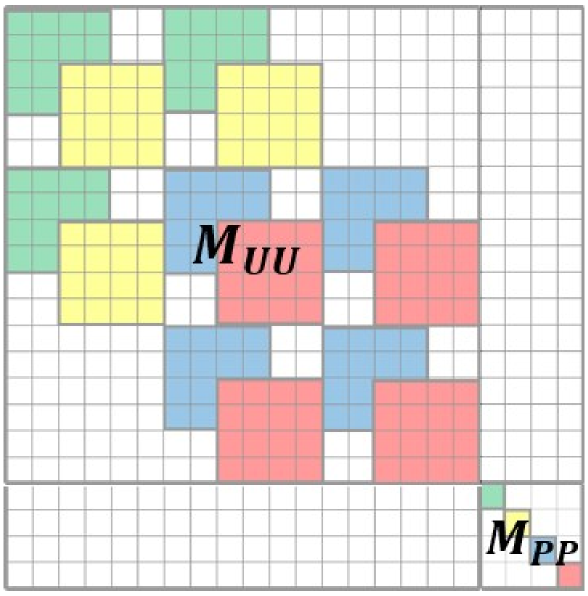

Figure 29: The globally assembled mass matrices

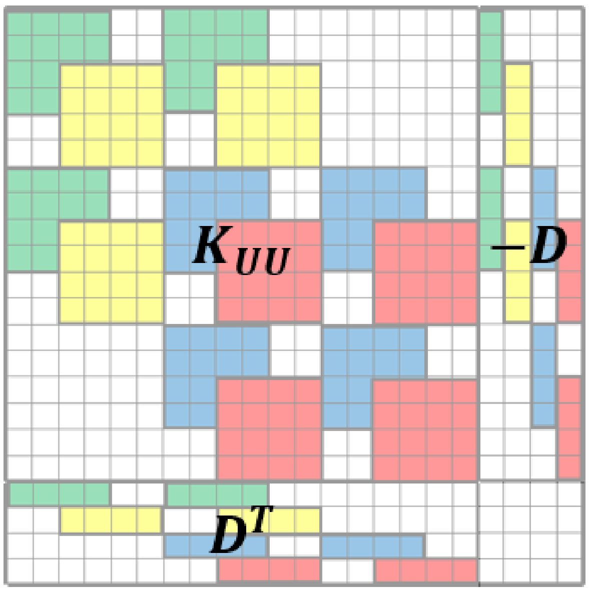

Figure 30: The globally assembled stiffness, gradient and divergence matrices

$$\mathbb{Y} = \begin{pmatrix} \mathbf{U} \\ \mathbf{P} \end{pmatrix} = \begin{pmatrix} u_1 \\ v_1 \\ \dots \\ u_9 \\ v_9 \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{pmatrix} \qquad \begin{pmatrix} \mathbf{M_{UU}} & \mathbf{0} \\ \mathbf{0} & \mathbf{M_{PP}} \end{pmatrix} \dot{\mathbb{Y}} + \begin{pmatrix} \mathbf{K_{UU}} & -\mathbf{D} \\ \mathbf{D^T} & \mathbf{0} \end{pmatrix} \mathbb{Y} = \begin{pmatrix} \mathbf{F} \\ \mathbf{0} \end{pmatrix} \tag{27}$$

The globally assembled stiffness, mass, gradient and divergence matrices are presented in Figures 29 and 30. The advantage of presenting the problem as in ODE 27 is that one only needs to manipulate the vector $\mathbb{Y}$ which considerably simplifies the writing of the linear system of equations. Previously we had to deal with two different variables ($\mathbf{U}$ and $\mathbf{P}$) and two ODEs. It enables us to gather all the unknowns of the problem in the same vector.

4.6 Implementation optimisation

Thanks to some of the $\mathbf{M_{UU}}$, $\mathbf{M_{PP}}$, $\mathbf{K_{UU}}$ and $\mathbf{D}$ specific frames, it is possible to optimise their implementation to make the simulation faster and less memory-consuming. To do that, it is possible to use the Python library "scipy.sparse" which allows us to store the matrices in an optimised way. Moreover, this library supports the basic operations of the Numpy library, in particular the dot product, which is very comfortable.

As we previously highlighted, $\mathbf{M_{PP}}$ is a $(N_{element}, N_{element})$ diagonal matrix, where $N_{element}$ is the number of elements in the mesh. The number of terms it contains is thus $N_{element}^2$, i.e, one needs to allocate $N_{element}^2$ memory spaces to the matrix storage. Entirely storing $\mathbf{M_{PP}}$ (without any kind of memory optimisation) is completely inefficient because it is practically equivalent to storing a matrix of zeros, which does not contain any kind of useful information. The non-zero entries of the matrix are the $N_{element}$ terms of the diagonal. As a consequence the ratio of "useful" information is $^{N_{element}}/_{N_{element}^2} = {}^1/_{N_{element}}$. For instance, considering a mesh of 1000 elements (this is a really small one), we already have a ratio of "useful" information of 0,1%. Storing the entire matrix without taking into account its structure would be completely inefficient because 99,9% of the stored information would be useless. As a consequence, it is much more efficient to store it as a first-order tensor (vector) which contains the entries of the diagonal of the original matrix $\mathbf{M_{PP}}$. This is done with the variable type "dia_matrix" from the "scipy.sparse" library.

Using the same idea, the stiffness matrix $\mathbf{K_{UU}}$, and gradient matrix $\mathbf{D}$ are stored using the variable type "coo_matrix" of the "scipy.sparse" library. Rather than storing the entire matrix, the computer stores the line and column of the non-zero entries of the matrix as well as their value.

Let us have a look at a numerical example to make things clearer. We want to optimise the storage of matrix $\mathbf{A}$ defined as

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 3 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{28}$$

The "coo_matrix" associated to $\mathbf{A}$ would be

$$\mathbf{A}_{Coo} = \begin{pmatrix} 1 & 3 & 3 \\ 2 & 1 & 1 \end{pmatrix} \tag{29}$$

The first column of $\mathbf{A_{Coo}}$ represent the lines of the non-zero terms of $\mathbf{A}$, the second column of $\mathbf{A_{Coo}}$ the columns of the non-zero terms of $\mathbf{A}$ and finally the third column of $\mathbf{A_{Coo}}$ the non-zero entries of $\mathbf{A}$. For instance, the first line of $\mathbf{A_{Coo}}$ is 1, 3, 3 because the term of $\mathbf{A}$ at line 1 and column 3 is 3. The second line of $\mathbf{A_{Coo}}$ is 2, 1, 1 because the term of $\mathbf{A}$ at line 2 and column 1 is 1. This is how works the variable type "coo_matrix".

One important thing we need to know is that the variable type "coo_matrix" supports all the operations of the Numpy library, which greatly facilitates the solver implementation.

Figure 31 shows the memory space required to store the stiffness matrix $\mathbf{K_{UU}}$ in both standard and sparse matrix implementations. The horizontal axis represents the number of elements in

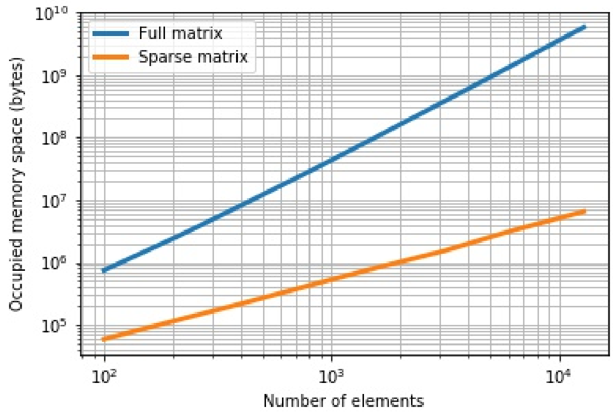

Figure 31: Memory space for standard and sparse matrices implementations

the mesh whereas the vertical one stands for the RAM memory space, in bytes. The two axis are presented in logarithmic scale. This graph is the result of a test I run myself. I simply build finer and finer meshes (thus increasing the number of elements) and compare them all using the Python method "object.nbytes" which returns the memory space (in bytes) occupied by the given "object".

As previously explained, the storage of sparse matrices is much more efficient. For instance, if we consider a mesh of 13 000 elements, the storage of the full stiffness matrix $\mathbf{K_{UU}}$ requires almost $6 \cdot 10^9$ bytes whereas it is of only $6 \cdot 10^6$ bytes for the sparse matrix, which is 1 000 times less. In addition, Figure 31 clearly highlights that the memory space for the full matrix is $O\left(N_{element}^2\right)$, because the slope of the blue line (which corresponds to the full matrix implementation) is 2[10], whereas it is only $O\left(N_{element}\right)$ for the sparse matrix implementation, an order of magnitude lower; this means that we will be able to use larger meshes with sparse matrices before reaching the RAM limit of the computer.

[10]Let us remember that the axis are presented in logarithmic scale.

5 Back and Forth Error Compensation and Correction method

To obtain the equations of Problem 15, we split the Navier-Stokes equations into two with, on the one hand, the PDE which describes the effects of compressibility, inertia and viscosity [11] and, on the other hand, a purely convective one, we will solve using the Back and Forth Error Compensation and Correction (BFECC) method

$$\frac{\partial \tilde{\mathbf{u}}}{\partial t} + \tilde{\mathbf{u}} \cdot \boldsymbol{\nabla} \tilde{\mathbf{u}} = 0 \tag{30}$$

where $\tilde{\mathbf{u}}$ is, in our case, the velocity field.

In the phenomena which are governed by PDEs which have the same shape as Equation 30, the transport of information only occurs thanks to the convective term. In other words, the particles of material transport with themselves all their physical characteristics and that is why there are changes in the velocity field throughout time. There is no viscosity neither compressibility.

The objective of this part is to present the main features and outcomes of the BFECC method without entering its details since it is not the core of the project. For more information, the reader can refer to [DL03].

5.1 First step forward

The BFECC is a combination of two steps forward and one step backward.

Let us imagine we are given a velocity field $\tilde{\mathbf{u}}^n$ at a given time t^n and we want to compute the velocity field $\tilde{\mathbf{u}}^{n+1}$ at time t^{n+1} at each point of the mesh. Since the PDE is only convective we only need to compute the velocity of the fluid particle which coincides with the point of the mesh $\mathbf{x_I}$ at time t^{n+1}. Figure 32 presents the follow-up of the particle in its trajectory. At time t^n, the fluid particle is in $\mathbf{y}$, then in $\mathbf{x_I}$ at t^{n+1} and in $\mathbf{z}$ at t^{n+2}. Let us recall that $\mathbf{x_I}$ is a node of the mesh. To know the velocity of the particle at t^{n+1} in $\mathbf{x_I}$, we just need to know where the particle was at the previous time step t^n and what its velocity at t^n was. To sum up, we have that

$$\overline{\mathbf{u}}^{n+1}\left(\mathbf{x_I}\right) = \tilde{\mathbf{u}}^n(\mathbf{y}) \tag{31}$$

where $\mathbf{y}$ is the position of the fluid particle at time t^n.

To compute $\mathbf{y}$, we use an approximation of the velocity

$$\tilde{\mathbf{u}}^n\left(\mathbf{x_I}\right) = \frac{\mathbf{x_I} - \mathbf{y}}{\delta t} \iff \mathbf{y} = \mathbf{x_I} - \delta t\, \tilde{\mathbf{u}}^n\left(\mathbf{x_I}\right) \tag{32}$$

If we gather Equation 31 and Equation 32 together, we get that

[11] We discretised it in the previous section, using Q1P0 finite elements.

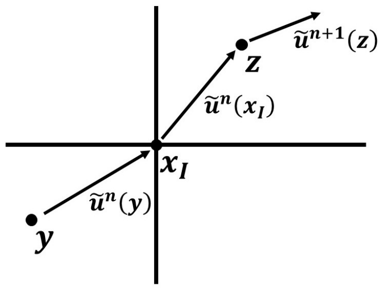

Figure 32: Follow-up of a fluid particle

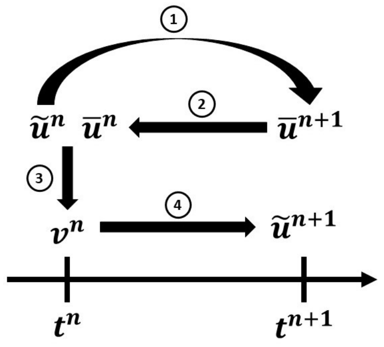

Figure 33: BFECC method summary

$$\overline{\mathbf{u}}^{n+1}\left(\mathbf{x_I}\right) = \tilde{\mathbf{u}}^{n}\left(\mathbf{x_I} - \delta t\,\tilde{\mathbf{u}}^{n}\left(\mathbf{x_I}\right)\right) \tag{33}$$

Since the velocity field $\tilde{\mathbf{u}}^n$ is known, we can interpolate it thanks to our mesh to find the value of $\tilde{\mathbf{u}}^n\left(\mathbf{x_I} - \delta t \tilde{\mathbf{u}}(\mathbf{x_I})\right)$. We have done the first step of the BFECC method (see Figure 33).

5.2 Backward step

Now, let us imagine we compute the velocity at time t^n knowing its value at time t^{n+1}, i.e. going back to the past[12]. As before, we need to track the particle which coincides with the position $\mathbf{x_I}$ at time t^n, i.e. determine where it will be and what its velocity will be at time t^{n+1}. The fluid particle which is at $\mathbf{x_I}$ at time t^n will follow its trajectory and will be at $\mathbf{z}$ at time t^{n+1} (see Figure 32). As a consequence:

$$\overline{\mathbf{u}}^{n+1}(\mathbf{z}) = \overline{\mathbf{u}}^{n}\left(\mathbf{x_I}\right) \tag{34}$$

since the particle transports with itself the information (in our case the velocity) it contains.

The same way we did before:

$$\overline{\mathbf{u}}^{n+1}(\mathbf{x_I}) = \frac{\mathbf{z} - \mathbf{x_I}}{\delta t} \iff \mathbf{z} = \mathbf{x_I} + \delta t\,\overline{\mathbf{u}}^{n+1}(\mathbf{x_I}) \tag{35}$$

And finally:

$$\overline{\mathbf{u}}^{n}\left(\mathbf{x_I}\right) = \overline{\mathbf{u}}^{n+1}\left(\mathbf{x_I} + \delta t\,\overline{\mathbf{u}}^{n+1}(\mathbf{x_I})\right) \tag{36}$$

[12]This is one of the key features of the BFECC method

5.3 Error compensation

Until now, we did a step forward in the future to get the nodal values of $\overline{\mathbf{u}}^{n+1}$ and a step backward in the past to compute the nodal values of $\overline{\mathbf{u}}^{n}$ (see the second step in Figure 33). If the process were perfect and reversible, the velocity field $\tilde{\mathbf{u}}^{n}$ and $\overline{\mathbf{u}}^{n}$ should be equal. However, we are doing some approximation and that is why there is an error $\mathbf{e}^{n} = \tilde{\mathbf{u}}^{n} - \overline{\mathbf{u}}^{n}$. This overall error is the sum of the error we do in the forward step and the one of the backward step.

$$\mathbf{e}^{n} = \mathbf{e}^{n}_{forward} + \mathbf{e}^{n}_{backward} \tag{37}$$

Let us assume that the error we do when going backward in the past is the same as the one we do when going forward in the future.

$$\mathbf{e}^{n}_{forward} = \mathbf{e}^{n}_{backward} \tag{38}$$

The expression of the forward error is thus:

$$\mathbf{e}^{n}_{forward} = \frac{1}{2}\mathbf{e}^{n} = \frac{1}{2}\left(\tilde{\mathbf{u}}^{n} - \overline{\mathbf{u}}^{n}\right) \tag{39}$$

As we know the nodal values of $\tilde{\mathbf{u}}^{n}$ and $\overline{\mathbf{u}}^{n}$, we can compensate it (as the name of the method indicates) so that it cancels itself when going forward in time once more.

5.4 Second forward step

We define:

$$\mathbf{v}^{n} = \tilde{\mathbf{u}}^{n} + \frac{1}{2}\left(\tilde{\mathbf{u}}^{n} - \overline{\mathbf{u}}^{n}\right) \tag{40}$$

and do one more step in the future to compute the value of $\tilde{\mathbf{u}}^{n+1}$.

$$\tilde{\mathbf{u}}^{n+1}\left(\mathbf{x_I}\right) = \mathbf{v^n}\left(\mathbf{x_I} - \delta t\, \mathbf{v}^{n}\left(\mathbf{x_I}\right)\right) \tag{41}$$

The idea behind this last step is that the forward error should cancel itself. In the end, we can compute the nodal variables at time step t^{n+1}.

Algorithm 2 is a sum up of what has to be done in the BFECC method. One disadvantage is that before going forward or backward, we need to compute all the nodal values, which implies the same operations to be done in three different loops.

Algorithm 2 The BFECC method algorithm

```
Require: $\tilde{\mathbf{u}}^n$, $\delta t$
  for node $\mathbf{x_I}$ in $\Gamma_{mesh}$ do
    Compute $\overline{\mathbf{u}}^{n+1}(\mathbf{x_I}) = \tilde{\mathbf{u}}^n\left(\mathbf{x_I} - \delta t\,\tilde{\mathbf{u}}^n(\mathbf{x_I})\right)$
  end for
  for node $\mathbf{x_I}$ in $\Gamma_{mesh}$ do
    Compute $\overline{\mathbf{u}}^n(\mathbf{x_I}) = \overline{\mathbf{u}}^{n+1}\left(\mathbf{x_I} + \delta t\,\overline{\mathbf{u}}^{n+1}(\mathbf{x_I})\right)$
  end for
  Compute $\mathbf{v}^n = \tilde{\mathbf{u}}^n + \frac{1}{2}\left(\tilde{\mathbf{u}}^n - \overline{\mathbf{u}}^n\right)$
  for node $\mathbf{x_I}$ in $\Gamma_{mesh}$ do
    Compute $\tilde{\mathbf{u}}^{n+1}(\mathbf{x_I}) = \mathbf{v^n}\left(\mathbf{x_I} - \delta t\,\mathbf{v}^n(\mathbf{x_I})\right)$
  end for
```

6 Time integration of equations

Let us recall the equations we got after having done the assembly of the stiffness, mass and gradient matrices.

$$\mathbf{M_{UU}}\dot{\mathbf{U}} + \mathbf{K_{UU}}\mathbf{U} - \mathbf{DP} = \mathbf{F_U} + \mathbf{M_{UU}}\dot{\tilde{\mathbf{U}}} \tag{42a}$$

$$\mathbf{M_{PP}}\dot{\mathbf{P}} + \mathbf{D^T U} = 0 \tag{42b}$$

We can adopt different strategies to integrate our system of equations in time, i.e. pass from time step t^n to time step t^{n+1}. The objective of this part of the work is to implement different ones, compare them and, in the end, validate the robustness of the fractional time step method.

Before writing the different time integration methods, let us sum up the situation taking into account each matrix and its "order of magnitude", i.e. the order of magnitude of the terms it contains. This has a huge importance when dealing with the stability of the numerical scheme.

Let us say that h is the characteristic size of the elements in the mesh, N_{node} the number of nodes and $N_{element}$ the number of elements.

To understand how to determine the order of magnitude of a matrix term, let us take the example of matrix $\mathbf{M_{PP}}$, which is the easiest one. We know that $\mathbf{M_{PP}}$ is a diagonal matrix and its diagonal terms are equal to the area of the corresponding element divided by the bulk modulus (constant in the element).

$$(\mathbf{M_{PP}})_i = \frac{A_i}{K}$$

Where $(\mathbf{M_{PP}})_i$ is the i-th element of $\mathbf{M_{PP}}$ diagonal and A_i the area of element i.

If we consider a structured mesh with square Q1P0 finite elements, $A_i = h^2$. Consequently, in this case, the diagonal terms of matrix $\mathbf{M_{PP}}$ are equal to h^2/K. In this case, we have more than an order of magnitude since it is possible to compute the exact value. Because the computation of $\mathbf{K_{UU}}$, $\mathbf{M_{UU}}$ and $\mathbf{D}$ requires assembling, it is harder to give the exact value of their entries, that is why we limit ourselves to an order of magnitude.

Matrix	Order of magnitude	Size
$\mathbf{M_{UU}}$	ρh^2	$(2N_{node}, 2N_{node})$
$\mathbf{K_{UU}}$	μ	$(2N_{node}, 2N_{node})$
$\mathbf{D}$	h	$(2N_{node}, N_{element})$
$\mathbf{M_{PP}}$	h^2/K	$(N_{element}, N_{element})$

Table 2: Matrices main features

6.1 Forward Euler (FE)

6.1.1 Equations of Forward Euler

This is the most basic scheme. It consists in writing the equilibrium at t_n (after discretisation of time) and approximate the time derivatives as:

$$\partial_t \mathbf{U} = \frac{1}{\delta t}\left(\mathbf{U^{n+1}} - \mathbf{U^n}\right) \qquad \partial_t \mathbf{P} = \frac{1}{\delta t}\left(\mathbf{P^{n+1}} - \mathbf{P^n}\right) \qquad \partial_t \tilde{\mathbf{U}} = \frac{1}{\delta t}\left(\tilde{\mathbf{U}}^{n+1} - \mathbf{U}^n\right) \tag{43}$$

Which gives:

$$\frac{1}{\delta t}\mathbf{M_{UU}}(\mathbf{U^{n+1}} - \mathbf{U^n}) + \mathbf{K_{UU}U^n} - \mathbf{DP^n} = \mathbf{F^n} + \frac{1}{\delta t}\mathbf{M_{UU}}\left(\tilde{\mathbf{U}}^{n+1} - \mathbf{U}^n\right) \tag{44a}$$

$$\frac{1}{\delta t}\mathbf{M_{PP}}(\mathbf{P^{n+1}} - \mathbf{P^n}) + \mathbf{D^T U^n} = \mathbf{0} \tag{44b}$$

Then we just need to find the expression of $\mathbf{P^{n+1}}$ and $\mathbf{U^{n+1}}$ as a function of $\mathbf{P^n}$ and $\mathbf{U^n}$.

$$\mathbf{P^{n+1}} = \mathbf{P^n} - \delta t\, \mathbf{M_{PP}^{-1} D^T U^n} \tag{45a}$$

$$\mathbf{U^{n+1}} = \delta t\, \mathbf{M_{UU}^{-1}}\left(\mathbf{F^n} + \mathbf{DP^n} - \mathbf{K_{UU}U^n}\right) + \tilde{\mathbf{U}}^{n+1} \tag{45b}$$

To avoid computing the solution of the linear system in the first equation, the mass matrix $\mathbf{M_{UU}}$ is lumped so that it is diagonal and its inverse is straightforwardly computed. To lump a matrix, we compute the sum of each matrix line i and store this value in the diagonal term i (see Figure 34). As a consequence, the lumped matrix is diagonal.

The Forward Euler scheme is an explicit one; the computational cost of the simulation is considerably reduced since no linear system of equations is solved.

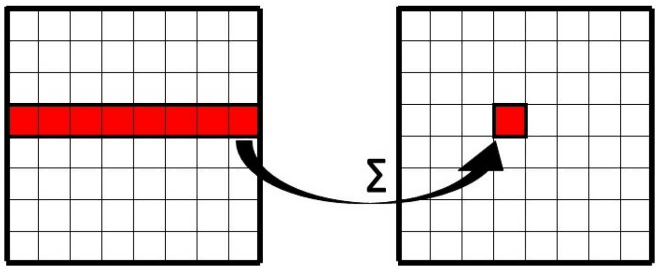

Figure 34: Lumping process

Operation	Operation type	Computational cost
$\delta t\, \mathbf{M_{PP}^{-1} D^T U^n}$	Matrix multiplication	$(2N_{node})N_{element}$
$\mathbf{P^{n+1}} = \mathbf{P^n} - \delta t\, \mathbf{M_{PP}^{-1} D^T U^n}$	Vector addition	$N_{element}$

Table 3: Computational cost of Equation 45a

6.1.2 Computational cost of the Forward Euler scheme

Let us study now one time integration step. The goal is to compute the number of operations the computer needs to do to pass from $\mathbf{U^n}$ and $\mathbf{P^n}$ to $\mathbf{U^{n+1}}$ and $\mathbf{P^{n+1}}$. We study each equation of the numerical scheme and estimate the number of elementary operations (addition, matrix/matrix multiplication, linear system resolution,...) the computer needs to do to go to the following one.

First, let us consider Equation 45a. Two operations are required: one matrix/vector multiplication and one vector addition (see Table 3). We consider that the matrix $\mathbf{M_{PP}^{-1} D^T}$ of size $(N_{element}, 2N_{node})$ is computed once and for all at the beginning of the process. It is not necessary to compute it at each time step.

Globally, Equation 45a has a computational cost of $(2N_{node}N_{element}) + N_{element} = N_{element}(2N_{node} + 1)$. As we want to study the asymptotic behaviour of the model, we make the assumptions that $N_{element} >> 1$ and $N_{node} >> 1$. This basically means that we are studying very large meshes, which is the main purpose of the work we are doing.

As a consequence:

$$C_{45a} = O(2N_{node}N_{element}) \tag{46}$$

where C_{45a} is the computational cost of Equation 45a.

Equation 45b is the second one of the Forward Euler scheme. There are three matrix/vector products and two vector additions (see Table 4).

Finally, we have $C_{45b} = 2N_{node}(4N_{node} + N_{element} + 2)$ (see Table 4).

$$C_{45b} = O(2N_{node}(4N_{node} + N_{element})) \tag{47}$$

Operation	Operation type	Computational cost
$\mathbf{K_{UU}U^n}$	Matrix/vector product	$(2N_{node})^2$
$\mathbf{DP^n}$	Matrix/vector product	$(2N_{node})N_{element}$
$\mathbf{F^n} + \mathbf{DP^n} - \mathbf{K_{UU}U^n}$	Vector addition	$2N_{node}$
$\delta t \mathbf{M_{UU}^{-1}} (\mathbf{F^n} + \mathbf{DP^n} - \mathbf{K_{UU}U^n})$	Matrix/vector multiplication	$(2N_{node})^2$
$\mathbf{U^{n+1}} = \mathbf{\tilde{U}^{n+1}} + ...$	Vector addition	$2N_{node}$

Table 4: Computational cost of Equation 45b

To obtain the computational cost of the whole Forward Euler scheme, we add the complexity of Equations 45a and 45b, which gives:

$$C_{FE} = O(4N_{node}(2N_{node} + N_{element})) \tag{48}$$

6.1.3 Stability of the Forward Euler scheme

The Forward Euler method is explicit, which means it is conditionally stable. First of all, let us study the stability of Equation 44a. To make it stable, $^1/_{\delta t}\,\mathbf{M_{UU}}$ must "dominate" $\mathbf{K_{UU}}$ and $\mathbf{D}$. We know the value of the entries of these matrices. The order of magnitude of the $^1/_{\delta t}\,\mathbf{M_{UU}}$ terms is $^1/_{\delta t}\,\rho h^2$. The order of magnitude of the $\mathbf{K_{UU}}$ terms is μ. Finally, the order of magnitude of the $\mathbf{D}$ terms is h, which leads to 2 restrictions on the time step δt.

$$\delta t << \rho \frac{h^2}{\mu} \tag{49a}$$

$$\delta t << \rho h \tag{49b}$$

Let us study now the stability of Equation 44b. The process is the same as before. $\mathbf{M_{PP}}$ must dominate $\mathbf{D^T}$. The order of magnitude of the $^1/_{\delta t}\,\mathbf{M_{PP}}$ terms is $^1/_{\delta t} \cdot {}^{h^2}/_{K}$, which leads to the following restriction on δt.

$$\delta t << \frac{h}{K} \tag{50}$$

Globally, the most restrictive condition we obtain is $\delta t << {}^h/_K$. For instance, if we consider a problem involving water, the bulk modulus would be of 2,2 GPa and 0,1 for h. Of course the value of h depends on the required resolution. This numerical example is meant to get a first insight of the problems we would face. The restriction we found only gives us an order of magnitude for the time step we should use. Indeed, the reality is much more complex because the frontier of the stability domain of the Forward Euler scheme is not so clear. It depends on many parameters such as the truncation error of the computer for instance or the quality of the mesh we use.

As a consequence, in this particular example, the restriction becomes $\delta t << 4.5.10^{-11}$, which is incompatible with reasonable computation time. A Forward Euler scheme is really easy to implement and understand but its conditional stability, in the case of quasi-incompressible fluids, is a weak point. This is the reason why we should look for an alternative time integration strategy.

One of the direct consequence of Equation 50 is that a smaller time step is to be used for smaller elements. A priori, it appears a bit strange, the main thought being that for smaller elements, a bigger time step should be used. However, Equation 50 is totally coherent with CFD theory, particularly with the use of the Courant number, defined for each element as $Co = {}^{v\delta t}/_{h}$, where v, is a characteristic velocity of the element, δt, the time step and h the size of the element. The Courant–Friedrichs–Lewy condition states that the Courant number should inferior to 1 so that the numerical simulation is stable. If we use smaller elements (smaller h), we should decrease the value of δt too, which is consistent with our previous analysis.

6.2 Backward Euler (BE)

The main outcome from the previous section is that we cannot use Q1P0 elements mesh coupled with the Forward Euler scheme because the condition on the time step is too restrictive. To overcome this problem, a completely implicit numerical scheme could be implemented. The most basic one we can think of is the Backward Euler one. To obtain the equations, we need to write the equilibrium at t_{n+1}.

$$\frac{1}{\delta t}\mathbf{M_{UU}}(\mathbf{U^{n+1}} - \mathbf{U^{n}}) + \mathbf{K_{UU}}\mathbf{U^{n+1}} - \mathbf{D}\mathbf{P^{n+1}} = \mathbf{F^{n+1}} + \frac{1}{\delta t}\left(\tilde{\mathbf{U}}^{n+1} - \mathbf{U}^{n}\right)$$
$$\frac{1}{\delta t}\mathbf{M_{PP}}(\mathbf{P^{n+1}} - \mathbf{P^{n}}) + \mathbf{D^{T}}\mathbf{U^{n+1}} = \mathbf{0}$$

It is necessary to operate over the equations to finally get the expressions of $\mathbf{P^{n+1}}$ and $\mathbf{U^{n+1}}$.

$$(\frac{1}{\delta t}\mathbf{M_{UU}} + \mathbf{K_{UU}} + \delta t\,\mathbf{D}\mathbf{M_{PP}^{-1}}\mathbf{D^{T}})\mathbf{U^{n+1}} = \mathbf{F^{n+1}} + \mathbf{D}\mathbf{P^{n}} + \frac{1}{\delta t}\mathbf{M_{UU}}\tilde{\mathbf{U}}^{\mathbf{n+1}} \tag{52a}$$
$$\mathbf{P^{n+1}} = \mathbf{P^{n}} - \delta t\mathbf{M_{PP}^{-1}}\mathbf{D^{T}}\mathbf{U^{n+1}} \tag{52b}$$

The Backward Euler scheme is completely implicit, which means that it is unconditionally stable: there is no restriction on the time step we can use. This is a great advantage but the price we have to pay for this is the computation of several matrices dot product (with a high computational cost) and the resolution of a linear system of equations, which can be very large depending on the mesh we use (see Equation 52a). Using the Python library Numpy, the cost of solving the linear system of equations would approximately be $O((2N_{node})^{3})$, which is an order of magnitude bigger than the Forward Euler scheme.

As a consequence, to make the BE numerical scheme competitive, we need to find a way to speed up the resolution of the linear system of equations. To do that, we will use the Conjugate Gradient method, such as it was presented in Section 2. To be able to use this method to solve the linear system of equations, we need to prove that the matrix $\mathbf{Q} = {}^{1}/_{\delta t}\mathbf{M_{UU}} + \mathbf{K_{UU}} + \delta t\mathbf{D}\mathbf{M_{PP}^{-1}}\mathbf{D^T}$ is symmetric positive definite. Otherwise, the Conjugate Gradient algorithm would not work.

First of all, it appears quite clearly that this matrix is symmetric.

$$\begin{aligned}\mathbf{Q^T} &= \frac{1}{\delta t}\mathbf{M_{UU}^T} + \mathbf{K_{UU}^T} + \delta t\left(\mathbf{D}\mathbf{M_{PP}^{-1}}\mathbf{D}^T\right)^T \\ &= \frac{1}{\delta t}\mathbf{M_{UU}} + \mathbf{K_{UU}} + \delta t\mathbf{D}\mathbf{M_{PP}^{-T}}\mathbf{D^T}\end{aligned}$$

since $\mathbf{M_{UU}} = \mathbf{M_{UU}^T}$ (diagonal matrix) and $(\mathbf{AB})^T = \mathbf{B^T}\mathbf{A^T}$

$$= \frac{1}{\delta t}\mathbf{M_{UU}} + \mathbf{K_{UU}} + \delta t\mathbf{D}\mathbf{M_{PP}^{-1}}\mathbf{D^T}$$

because $\mathbf{M_{PP}}$ and $\mathbf{M_{PP}^{-1}}$ are diagonal matrices

In the end, we have $\mathbf{Q^T} = \mathbf{Q}$, which proves that $\mathbf{Q}$ is symmetric.

To prove that it is definite positive, let us consider a vector $\mathbf{x} \neq \mathbf{0}$. We have to show that $\mathbf{x^T}\mathbf{Q}\mathbf{x} > 0$. By definition:

$$\mathbf{x^T}\mathbf{Q}\mathbf{x} = \frac{1}{\delta t}\mathbf{x^T}\mathbf{M_{UU}}\,\mathbf{x} + \mathbf{x^T}\mathbf{K_{UU}}\,\mathbf{x} + \delta t\,\mathbf{x^T}\mathbf{D}\mathbf{M_{PP}^{-1}}\mathbf{D^T}\,\mathbf{x}$$

Let us consider the first term $\mathbf{x^T}\mathbf{M_{UU}}\,\mathbf{x}$ and say that $\mathbf{x} = \begin{pmatrix} x_0 & x_1 & ... & x_n \end{pmatrix}^T$. After some computation, we get that

$$\mathbf{x^T}\mathbf{M_{UU}}\,\mathbf{x} = \sum_{i=1}^{n}(\mathbf{M_{UU}})_i\,x_i^2$$

Since $(\mathbf{M_{UU}})_i > 0$, we can conclude that $\mathbf{x^T}\mathbf{M_{UU}}\,\mathbf{x} > \mathbf{0}$ because it is a sum of strictly positive terms.

Secondly, let us consider the term $\mathbf{x^T}\mathbf{K_{UU}}\mathbf{x}$. To prove that it is positive, it is important to recall the definition of $\mathbf{K_{UU}}$

$$\mathbf{K_{UU}} = \int_{\Omega} \mathbf{B^T}\mathbb{C}\mathbf{B}\,d\Omega \tag{53}$$

which means that

$$\mathbf{x^T K_{UU} x} = \mathbf{x^T}\left(\int_\Omega \mathbf{B^T}\mathbb{C}\mathbf{B}\, d\Omega\right)\mathbf{x}$$
$$= \int_\Omega \mathbf{x^T B^T}\mathbb{C}\mathbf{B x}\, d\Omega$$

since $\mathbf{x}$ is a constant vector

$$= \int_\Omega (\mathbf{Bx})^T\, \mathbb{C}\, (\mathbf{Bx})\ d\Omega$$

Let us introduce the vector $\mathbf{z} = \begin{pmatrix} z_0 & z_1 & z_2 \end{pmatrix}^T = \mathbf{Bx}$. We have that

$$\mathbf{x^T K_{UU} x} = \int_\Omega \mathbf{z^T}\mathbb{C}\mathbf{z}\, d\Omega \tag{54}$$

It can be almost immediately seen that the function $\mathbf{z^T}\mathbb{C}\mathbf{z} = 2\mu z_0^2 + 2\mu z_1^2 + \mu z_2^2$ is always positive, which means that $\mathbf{x^T K_{UU} x} > 0$.

Lastly, we have to study $\mathbf{x^T D M_{PP}^{-1} D^T x}$. Let us introduce $\mathbf{y} = \mathbf{D^T x}$. It is possible to interpret the previous expression as:

$$\mathbf{x^T D M_{PP}^{-1} D^T x} = (\mathbf{D^T x})^T \mathbf{M_{PP}^{-1}} (\mathbf{D^T x}) = \mathbf{y^T M_{PP}^{-1} y}$$

As previously, let us say $\mathbf{y} = \begin{pmatrix} y_0 & y_1 & \dots & y_m \end{pmatrix}^T$

$$\mathbf{y^T M_{PP}^{-1} y} = \sum_{i=1}^{m} (\mathbf{M_{PP}^{-1}})_i\, y_i^2$$

Let us remember that all the terms of matrix $\mathbf{M_{PP}}$ are positive. Therefore, $(\mathbf{M_{PP}^{-1}})_i > 0$ and $\mathbf{y^T M_{PP}^{-1} y} > 0$.

From this development, we can deduce that for all $\mathbf{x} \neq 0$, $\mathbf{x^T Q x} > 0$ and $\mathbf{Q}$ is symmetric definite positive. Due to this, we can use the Conjugate gradient iterative method to find the solution of the linear system, which considerably reduces the computational cost of the Backward Euler scheme since it is only limited by the resolution of the linear system of equations using the Conjugate Gradient method.

Let us compute the overall computational cost of the overall BE method. As previously, we need to find the computational cost of each equation and make the sum.

The computational cost of Equations 52b and 52a are given by Table 5 and 5 respectively.

$$C_{52b} = 2 N_{node} N_{element} \tag{55a}$$

Operation	Operation type	Computational cost
$\delta t \mathbf{M_{PP}^{-1} D^T U^{n+1}}$	Matrix/vector product	$2N_{node}N_{element}$
$\mathbf{P^n} - \delta t \mathbf{M_{PP}^{-1} D^T U^{n+1}}$	Vector addition	$N_{element}$

Table 5: Computational cost of Equation 52b

Operation	Operation type	Computational cost
$\mathbf{DP^n}$	Matrix/vector product	$2N_{node}N_{element}$
$\frac{1}{\delta t}\mathbf{M_{UU}\tilde{U}^{n+1}}$	Matrix/vector product	$(2N_{node})^2$
$\mathbf{F^{n+1}} + \mathbf{DP^n} + \frac{1}{\delta t}\mathbf{M_{UU}\tilde{U}^{n+1}}$	Vector addition	$2N_{node}$
$\mathbf{QU^{n+1}} = \mathbf{F^{n+1}} + ...$	Linear system resolution	$(2N_{node})^2$

Table 6: Computational cost of Equation 52a

$$C_{52a} = 2N_{node}N_{element} + 8N_{node}^2 \tag{55b}$$

Finally, the computational cost of the Backward Euler scheme is

$$C_{BE} = O(8N_{node}^2 + 4N_{node}N_{element}) \tag{56}$$

The computational cost of the BE method is relatively high. In order to find a compromise, it is worth it to study a third numerical scheme, known as Fractional Step splitting and widely used in Computational Fluid Dynamics. In theory, this "hybrid" scheme is conditionally stable, which is a disadvantage in comparison with the BE numerical scheme, but it has a lower computational cost.

6.3 Fractional step (FS) splitting

In order to overcome the problem induced by the mass balance equation (where the high value of the bulk modulus K does not allow to use a large time step), we need to write this equation at time step t_{n+1}, like in the Backward Euler scheme so that the resolution of this equation is implicit and does not depend on the time step we use.

6.3.1 Fractional step equations

The first equation approximately remains equal (see Equation 44a). The unique difference is that we use the term $\mathbf{P^{n+1}}$.

$$\frac{1}{\delta t}\mathbf{M_{UU}}(\mathbf{U^{n+1}} - \mathbf{U^n}) + \mathbf{K_{UU}U^n} - \mathbf{DP^{n+1}} = \mathbf{F^n} + \frac{1}{\delta t}\mathbf{M_{UU}}\left(\mathbf{\tilde{U}}^{n+1} - \mathbf{U}^n\right) \tag{57a}$$

$$\frac{1}{\delta t}\mathbf{M_{PP}}(\mathbf{P^{n+1}} - \mathbf{P^{n}}) + \mathbf{D^{T}U^{n+1}} = \mathbf{0} \tag{57b}$$

Up to this stage, we have to solve 2 implicit linear equations. This would have a huge computational cost, that is why we decide to add an intermediate step. We introduce an "intermediate" velocity variable $\hat{\mathbf{U}}$ which helps us solving Equation 57a. $\hat{\mathbf{U}}$ can be seen as a first approximation of $\mathbf{U}$.

$$\frac{1}{\delta t}\mathbf{M_{UU}}(\hat{\mathbf{U}}^{\mathbf{n+1}} - \mathbf{U^{n}}) + \mathbf{K_{UU}}\hat{\mathbf{U}}^{n} - \mathbf{DP^{n}} = \mathbf{F^{n}} + \frac{1}{\delta t}\mathbf{M_{UU}}\left(\tilde{\mathbf{U}}^{n+1} - \mathbf{U}^{n}\right) \tag{58a}$$

$$\frac{1}{\delta t}\mathbf{M_{UU}}(\mathbf{U^{n+1}} - \hat{\mathbf{U}}^{\mathbf{n+1}}) - \mathbf{D}(\mathbf{P^{n+1}} - \mathbf{P^{n}}) = 0 \tag{58b}$$

$$\frac{1}{\delta t}\mathbf{M_{PP}}(\mathbf{P^{n+1}} - \mathbf{P^{n}}) + \mathbf{D^{T}U^{n+1}} = \mathbf{0} \tag{58c}$$

We can remark that the sum of Equations 58a and 58b gives back Equation 57a. The two systems are strictly equivalent, except we assume that $\mathbf{K_{UU}U^{n}} \approx \mathbf{K_{UU}}\hat{\mathbf{U}}^{n}$. Equation 57b remains equal.

After some manipulations of equations, we finally get Equations 59a, 59b, 59c and 59d.

$$\hat{\mathbf{U}}^{\mathbf{n+1}} = \tilde{\mathbf{U}}^{\mathbf{n+1}} + \delta t\,\mathbf{M_{UU}^{-1}}(\mathbf{F^{n}} - \mathbf{K_{UU}}\hat{\mathbf{U}}^{\mathbf{n}} + \mathbf{DP^{n}}) \tag{59a}$$

$$(\frac{1}{\delta t}\mathbf{M_{PP}} + \delta t\,\mathbf{D^{T}M_{UU}^{-1}D})\,\mathbf{dP^{n+1}} = -\mathbf{D^{T}}\hat{\mathbf{U}}^{\mathbf{n+1}} \tag{59b}$$

$$\mathbf{P^{n+1}} = \mathbf{P^{n}} + \mathbf{dP^{n+1}} \tag{59c}$$

$$\mathbf{U^{n+1}} = \hat{\mathbf{U}}^{\mathbf{n+1}} + \delta t\,\mathbf{M_{UU}^{-1}D}\,\mathbf{dP^{n+1}} \tag{59d}$$

We have to solve two explicit equations (59a and 59d) and an implicit one (see Equation 59b).

6.3.2 Fractional step computational cost

The computational cost of this Fractional Step scheme is higher than for a Forward Euler scheme. We still consider that the number of velocity DOFs is $2N_{node}$ and the number of pressure DOFs is $N_{element}$.

First, let us study Equation 59a, which involves 3 vector/matrix multiplication and two vector additions. Its computational cost is thus $C_{59a} = O(2N_{node}(4N_{node} + N_{element}))$, by adding the contribution of each step presented in Table 7.

The same process is repeated for Equation 59b.

We do not consider, in this analysis of the computational cost, the computation of the matrix $1/\delta t\,\mathbf{M_{PP}} + \delta t\,\mathbf{D^{T}M_{UU}^{-1}D}$ because it is done once and for all at the beginning of the time integration loop, i.e. we do not need to compute it for each time iteration and we do not consider relevant to include it in this analysis since it is a punctual operation.

Operation	Operation type	Computational cost
$\mathbf{DP^n}$	Vector/matrix multiplication	$(2N_{node})N_{element}$
$\mathbf{K_{UU}\hat{U}^n}$	Vector/matrix multiplication	$(2N_{node})^2$
$\mathbf{F^n} - \mathbf{K_{UU}\hat{U}^n} + \mathbf{DP^n}$	Vector addition	$2N_{node}$
$\delta t\, \mathbf{M_{UU}^{-1}}(\mathbf{F^n} - \mathbf{K_{UU}\hat{U}^n} + \mathbf{DP^n})$	Vector/matrix product	$(2N_{node})^2$
$\mathbf{\hat{U}^{n+1}} = \mathbf{\tilde{U}^{n+1}} + \ldots$	Vector addition	$2N_{node}$

Table 7: Computational cost of Equation 59a

Operation	Operation type	Computational cost
$\mathbf{D^T\hat{U}^{n+1}}$	Vector/matrix multiplication	$(2N_{node})N_{element}$
Computation of $\mathbf{dP^{n+1}}$	Linear system resolution	$N^2_{element}$

Table 8: Equation 59b computational cost

Let us focus on the resolution of the linear system to find the value of $\mathbf{dP^{n+1}}$. Obviously, it is possible to use the Python library "Numpy" and its method "linalg.solve". However, this would be completely useless. Indeed, despite we do not have a clear insight of what this function really does to solve the linear system, the user manual of the Numpy library estimates that the computational cost of solving a linear system with this function is $O(N^3_{element})$.

The linear system to be solved in the Fractional Step time integration is:

$$\left(\frac{1}{\delta t}\mathbf{M_{PP}} + \delta t \mathbf{D^T M_{UU}^{-1} D}\right) \mathbf{dP^{n+1}} = -\mathbf{D^T \hat{U}^{n+1}}$$

We need to prove that $\mathbf{L} = {}^{1}/_{\delta t}\mathbf{M_{PP}} + \delta t\, \mathbf{D^T M_{UU}^{-1} D}$ is symmetric, definite and positive. It appears quite clearly that this matrix is symmetric.

$$\begin{aligned} \mathbf{L^T} &= \frac{1}{\delta t}\mathbf{M_{PP}^T} + \delta t \left(\mathbf{D^T M_{UU}^{-1} D}\right)^T \\ &= \frac{1}{\delta t}\mathbf{M_{PP}} + \delta t \mathbf{D^T M_{UU}^{-T} D} \end{aligned}$$

since $\mathbf{M_{PP}} = \mathbf{M_{PP}^T}$ (diagonal matrix) and $(\mathbf{AB})^T = \mathbf{B^T A^T}$

$$= \frac{1}{\delta t}\mathbf{M_{PP}} + \delta t \mathbf{D^T M_{UU}^{-1} D}$$

because $\mathbf{M_{UU}}$ has been previously lumped and $\mathbf{M_{UU}^{-1}}$ is a diagonal matrix

Operation	Operation type	Computational cost
$\delta t\, \mathbf{M_{UU}^{-1} D\, dP^{n+1}}$	Vector/matrix multiplication	$(2N_{node})N_{element}$
$\mathbf{U^{n+1}} = \mathbf{\hat{U}^{n+1}} + ...$	Vector addition	$N_{element}$

Table 9: Equation 59d computational cost

In the end, we have $\mathbf{L^T} = \mathbf{L}$, which proves that $\mathbf{L}$ is symmetric. To prove that is definite and positive, let us consider a vector $\mathbf{x} \neq \mathbf{0}$. We have to prove that $\mathbf{x^T L x} > 0$. By definition:

$$\mathbf{x^T L x} = \frac{1}{\delta t}\mathbf{x^T M_{PP}\, x} + \delta t\, \mathbf{x^T D^T M_{UU}^{-1} D\, x}$$

Let us consider the first term $\mathbf{x^T M_{PP}\, x}$ and say that $\mathbf{x} = \begin{pmatrix} x_0 & x_1 & ... & x_n \end{pmatrix}^T$.

$$\mathbf{x^T M_{PP}\, x} = \sum_{i=1}^{n} (\mathbf{M_{PP}})_i\, x_i^2$$

Since $(\mathbf{M_{PP}})_i > 0$ because it contains the area of the mesh element i divided by the bulk modulus K, we can conclude that $\mathbf{x^T M_{PP}\, x} > \mathbf{0}$ (it is a sum of strictly positive terms).

On the other hand, we have to study $\mathbf{x^T D^T M_{UU}^{-1} D\, x}$. Let us introduce $\mathbf{y} = \mathbf{Dx}$. It is possible to interpret the previous expression as:

$$\mathbf{x^T D^T M_{UU}^{-1} D x} = (\mathbf{Dx})^T \mathbf{M_{UU}^{-1}} (\mathbf{Dx}) = \mathbf{y^T M_{UU}^{-1} y}$$

As previously, let us say $\mathbf{y} = \begin{pmatrix} y_0 & y_1 & ... & y_m \end{pmatrix}^T$

$$\mathbf{y^T M_{UU}^{-1} y} = \sum_{i=1}^{m} (\mathbf{M_{UU}^{-1}})_i\, y_i^2$$

Let us remember that all the terms of matrix $\mathbf{M_{UU}}$ are positive and that the matrix has been lumped. Therefore, $(\mathbf{M_{UU}})_i > 0$. As a consequence, $\mathbf{y^T M_{UU}^{-1} y} > 0$.

From this development, we can deduce that for all $\mathbf{x} \neq 0$, $\mathbf{x^T L x} > 0$ and $\mathbf{L}$ is a symmetric, definite and positive. Due to this, we can use the Conjugate gradient iterative method to find the solution of the linear system $\mathbf{L\, dP^{n+1}} = \mathbf{F_P}^{\mathbf{n+1}} - \mathbf{D^T \hat{U}^{n+1}}$, which will considerably speed up the resolution of the problem.

The computational cost of solving the system of linear equations is $N_{element}^2$, which means that $C_{59b} = O(N_{element}(2N_{node} + N_{element}))$

Finally, one can compute the computational cost of Equation 59d (see Table 9):

$$C_{59d} = O(2N_{node}N_{element}) \tag{60}$$

Let us sum up the computational cost of the whole Fractional Step method. We have to add every contribution of each resolution step, which gives:

$$C_{FS} = O(8N_{node}^2 + N_{element}^2 + 6N_{node}N_{element}) \tag{61}$$

6.3.3 Fractional step stability

There are similarities between the Forward Euler method and the Fractional Step splitting method. As before, to study the stability of this numerical scheme, we have to take into account each equation of the process and compare the order of magnitude of the different matrices.

Equation 59b is implicit; this means it is unconditionally stable. To determine the critical time step for which the numerical simulation converges, we only have to consider Equations 59a and 59d.

On the one hand, for Equation 59a, we want $\mathbf{M_{UU}}$ to dominate $\mathbf{K_{UU}}$ and $\mathbf{D}$, which leads to the restrictions $\delta t << {}^{\rho h^2}/_{\mu}$ and $\delta t << \rho h$. On the other hand, in Equation 59d, we only want $\mathbf{M_{UU}}$ to dominate $\mathbf{D}$, which leads to $\delta t << \rho h$

$$\delta t << \frac{\rho h^2}{\mu}$$
$$\delta t << \rho h$$

In many applications, $\rho h << {}^{\rho h^2}/_{\mu}$, since μ is quite low. As a consequence, the most restrictive condition is $\delta t << \rho h$. Let us study the same numerical example as in the Forward Euler part in order to compare the two methods. In this example, we had $h = 0.1$ and, for a real case application with water, the density ρ would be equal to $1000\,kg/m^3$, which means the condition we get on the time step is $\delta t << 100$. Fractional step is much more stable than the Forward Euler scheme so we can use larger time step for the temporal integration.

6.4 Summary

To finish this section, let us sum up the different time integration methods and the order of magnitude of their critical time step as well as their computational cost.

We will express the computational cost of each numerical scheme with the variables of the problem. Let us say that number of elements in the horizontal and vertical directions are respectively N_x and N_y in a structured mesh. The total number of elements is $N_{element} = N_xN_y$. The total number of points in the mesh is $N_{node} = (N_x + 1)(N_y + 1) \approx N_xN_y \approx N_{element}$ because we assume $N_x, N_y >> 1$.

Numerical scheme	**Critical time step**	**Computational cost**
Forward Euler	h/K	$8N_{node}^2 + 4N_{node}N_{element} = O(N_{element}^2)$
Fractional step	ρh	$8N_{node}^2 + N_{element}^2 + 6N_{node}N_{element} = O(N_{element}^2)$
Backward Euler	Unconditionally stable	$8N_{node}^2 + 4N_{node}N_{element} = O(N_{element}^2)$

Table 10: Time integration method stability / Computational cost

The main conclusion we can draw from Table 10 is that there is no significant difference in terms of computational cost between the three numerical scheme we studied in this section because we use the Conjugate Gradient method to solve the linear system of equations. Otherwise, the computational cost of the Fractional step splitting and the Backward Euler scheme would be $O(N_{element}^3)$ because, in both cases, one has to solve a linear system of equations.

The Forward Euler scheme is to be discarded because its critical time step is too small and the numerical simulation would be too long. Using the BE or the FS method is quite equivalent since the FS critical time step is quite big and it could be suitable for practical cases.

7 The Shifted Boundary method

As we previously explained, because of the Q1P0 special features, we are able to make the numerical simulation faster, especially because the mass matrix $\mathbf{M_{PP}}$ is diagonal. Nevertheless, the behaviour of Q1P0 elements strongly depends on the type of the mesh we use, Q1P0 elements presenting really poor results when assembled in unstructured meshes. For instance, Q1P0 elements would work correctly when disposed like in Figure 35 but would give really bad results with the mesh of Figure 36.

However, building a structured mesh over a complex geometry as well as imposing boundary conditions in such situation are not so easy and that is why we need to use the Shifted Boundary method (see [MS18] for the reference article).

7.1 Complex geometries and structured meshes. Implicit geometrey representation

Normally, the type of mesh one has to use depends on the problem geometry. To represent rectangular domain as in Figure 35, it is strongly recommended to use a structured mesh because the geometry presents two dominant directions and the structured mesh can adapt to this particular configuration. However, everything becomes much more difficult with complex geometry. For example, standard meshers will not be able to provide a structured mesh of the geometry presented in Figure 36. Structured meshes are not as adaptative as unstructured ones, which makes the meshing complicated.

In order to make the Q1P0 elements work properly, it is necessary to find a way to create structured meshes of complex geometry.

7.1.1 Detect inner nodes

Let us say we want to build a structured mesh taking into account a hole in the geometry (see Figure 37). The first thing to do is to spot the inner points, i.e. the points of the mesh which are situated within the circle (see Figure 38).

Every given geometry Ω can be completely characterised by its level set function f_Ω. A given point $\mathbf{x}$ is an inner point of the geometry Ω if and only if $f_\Omega(\mathbf{x}) < 0$.

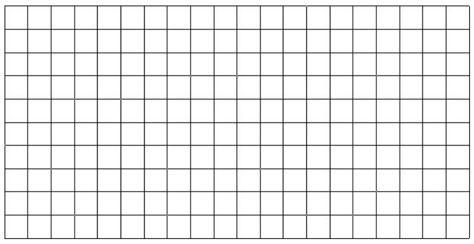
Figure 35: Example of a structured mesh using quadrilateral elements

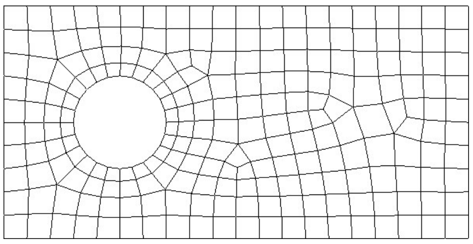
Figure 36: Example of an unstructured mesh using quadrilateral elements

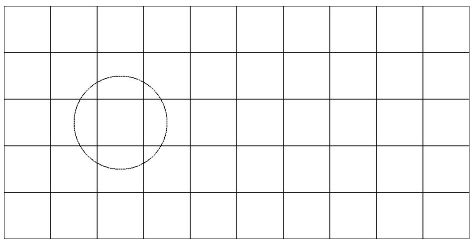
Figure 37: Geometry to be meshed

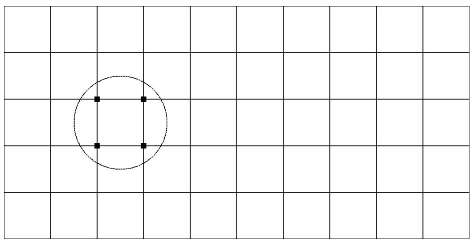
Figure 38: Detection of inner points

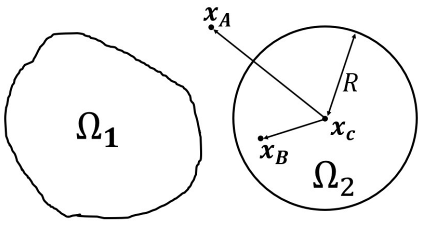

Figure 39: Different hole geometries

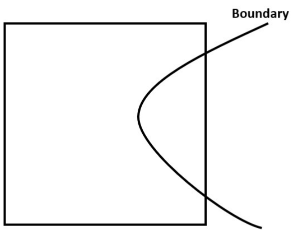

Figure 40: A particular configuration

Definition 6. $\forall \mathbf{x}$, $\mathbf{x}$ is an interior point of $\Omega \iff f_\Omega(\mathbf{x}) < 0$

For a given geometry with unknown characteristics (see Ω_1 in Figure 39), it is not always possible to find an analytical expression of the level set function f_Ω. However, in the case of less complex geometries such as circles (see Ω_2 in Figure 39), the analytical expression of the level set function is known. Let us take the example of a circle whose centre is $\mathbf{x_C}$ and radius R as depicted in Figure 39. The expression of the level set function is:

$$\forall \mathbf{x}, f_{\Omega_2}(\mathbf{x}) = (\mathbf{x} - \mathbf{x_C})^T (\mathbf{x} - \mathbf{x_C}) - R^2 = ||\mathbf{x} - \mathbf{x_C}||^2 - R^2 \tag{63}$$

Point B with coordinates $\mathbf{x_B}$ is within the circle because $f_{\Omega_2}(\mathbf{x_B}) < 0$. The norm of the vector $\mathbf{x_B} - \mathbf{x_C}$ is inferior to the radius of the circle, which means, by definition, that point B is an interior point. On the contrary, point A is out of the circle because $f_{\Omega_2}(\mathbf{x_A}) > 0$. Using the level set function of a given shape is really useful to rapidly determine if a point $\mathbf{x}$ is an inner or an outer one. For each point of the structured mesh (see Figure 37) with coordinates $\mathbf{x}_I$, we compute the value of the level set function at point $\mathbf{x}$ and determine if the point is an inner or an outer one. An example of this process is given in Figure 38, where four different inner points (represented by ■) have been detected. Algorithm 3 sums up the different steps to be executed to detect all the inner nodes.

7.1.2 Detect inner elements

The second step is to define the inner elements of the mesh. An inner elements is compound by one or more inner nodes.

For instance, in Figure 41 (where inner elements are coloured in red), element 12 is an inner one because its up right node is within the circle. Element 22 has 2 nodes within the circle and element 23 is completely within the circle, that is why they are labelled as inner elements. We

Algorithm 3 Detecting inner nodes

```
Require: coordinates_matrix, $f_\Omega$, $n_{node}$
  for $node = 1 \dots n_{node}$ do
    Get node coordinates $\mathbf{x_{node}}$
    Compute $f_\Omega(\mathbf{x_{node}})$
    if $f_\Omega(\mathbf{x_{node}}) < 0$ then
      Add $node$ to the list of inner node
    end if
  end for
  Return the list of inner nodes
```

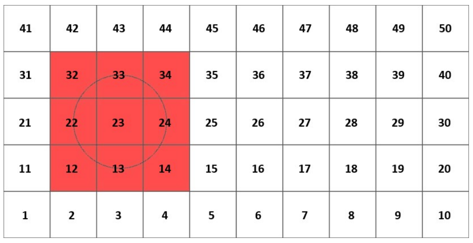

Figure 41: Original structured mesh and detected inner elements

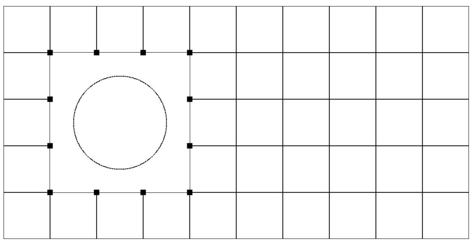

Figure 42: Final structured mesh

can imagine some configurations where some parts of the element are situated within the hole and the element is not referenced as an inner one, for example in the configuration of Figure 40. The element is not considered to be in the boundary shape because none of its node is an interior one, even if a part of its right side is "in the boundary".

Algorithm 4 Detecting inner elements

```
Require: $inner\_nodes$, $connectivity\_matrix$, $N_{element}$
  for $element = 1 \dots N_{element}$ do
    From $connectivity\_matrix$ get the labels $n_1$, $n_2$, $n_3$ and $n_4$ of the elements nodes
    if $n_1$ in $inner\_nodes$ or $n_2$ in $inner\_nodes$ or $n_3$ in $inner\_nodes$ or $n_4$ in $inner\_nodes$ then
      Add $element$ to the list of inner elements
    end if
  end for
  Return the list of inner elements
```

The final mesh we get (see Figure 42) has the great advantage of being structured, which means that we will be able to use Q1P0 finite elements. The inner elements elements (coloured in red in Figure 41), will not be taken into account in the process of assembly we depicted, as if they did not exist at all in the mesh.

Sometimes, obstacles in the fluid domain are not taken into account in the numerical simulation. In the example of Figure 43, there is no inner elements[13] (they would be coloured in red). If we run the simulation with this particular mesh, everything will happen as if the hole had been eliminated because the mesher was not able to detect it. Depending on the application

[13]This is because of how inner elements have been defined

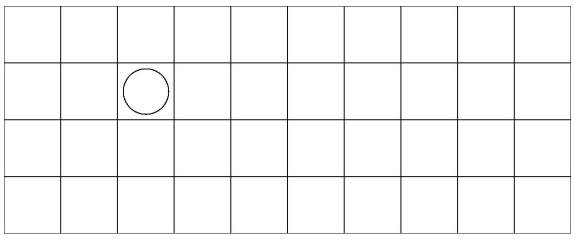
Figure 43: Example of geometry simplification

of the simulation, this can be seen as a drawback or an advantage. If one wants to model a flow with lots of precision, of course this geometry simplification is a huge problem because the approximation of the problem is not representative at all. However, if one is dealing with enormous geometry[14], this is not a problem; it can even be seen as an advantage because the geometry is considerably simplified and useless details are removed from the simulation, which can maybe make it more stable too.

To sum up, what we did in these previous steps was an approximation of the original domain geometry with a structured mesh. Now we have to tackle the problem of the boundary conditions. In the unstructured mesh of Figure 36, it is reasonable to consider the circle as the inner boundary of the mesh because the geometrical approximation of an unstructured mesh is really good[15]. However, in the case of the structured one, the boundary has been shifted from the circle (which corresponds to the original problem we want to solve) to the points represented by ■. These points are the new boundary of the domain where we are going to solve the Navier equations and the points where we are going to apply some given boundary conditions.

We cannot apply the exact same type of boundary conditions as the ones associated to the circle in the original problem because the geometrical approximation of the hole is really bad (we are trying to approximate a circle using square elements). The obtained results considering this very simple solution would not be representative at all. Obviously, if we reduce the size of the elements, the mesh will be a better approximation of the geometry and we could apply the same type of boundary conditions. For instance, if we impose the velocity to be null on the circle in the original problem, if the mesh geometrical approximation is good enough, it could be possible to consider imposing a null velocity at the ■ points too. However, let us remember the main purpose of this work. We want to design a really fast solver to do simulations of huge domains. Creating meshes with really small elements (because we want to do a representative approximation of the original geometry) is not a really good idea to make the simulation faster. More elements in the mesh necessarily means a higher computational cost.

To overcome this problem, we need to find a way to transpose the original boundary conditions (in our case the boundary conditions on the circle) to the new boundary points.

[14]Let us remember we are designing a special solver for this kind of geometry.
[15]This is one reason for which this type of mesh is widely used

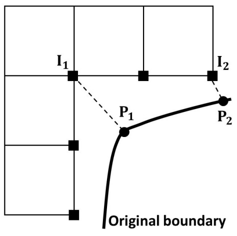

Figure 44: The closest point search

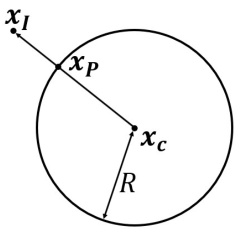

Figure 45: Closest point on a circle

7.2 Shifting the boundary conditions imposition

7.2.1 Closest point search

First of all, to transpose the boundary conditions from the original boundary to the actual mesh boundary, it is necessary to look for the points of the hole which are the closest to the structured mesh boundary.

Let us consider the situation of Figure 44, where the mesh boundary points are still represented by ■. I_1 and I_2 are two mesh boundary points. For each of them, we are to look for the points P_1 and P_2 of the hole which are closest to I_1 and I_2 respectively. In most cases, this search is complicated, especially when the shape of the hole is very complicated. However, in the case of simpler ones, the analytical expression of points P coordinates can be computed.

Let us study the example of Figure 45 where we are to find the coordinates $\mathbf{x_P} = (x_P, y_P)$ of point P, defined as the point of the circle which is the closest to I, knowing the coordinates $\mathbf{x_I} = (x_I, y_I)$ of point I, the coordinates $\mathbf{x_C} = (x_C, y_C)$ of the circle centre and the radius R of the circle.

We know that P is on the circle so, by definition:

$$(x_P - x_C)^2 + (y_P - y_C)^2 = ||\mathbf{x_P} - \mathbf{x_C}||^2 = R^2 \tag{64}$$

Moreover, the two vectors $\mathbf{x_I} - \mathbf{x_C}$ and $\mathbf{x_P} - \mathbf{x_C}$ are colinear, which means that:

$$\mathbf{x_P} - \mathbf{x_C} = \alpha\left(\mathbf{x_I} - \mathbf{x_C}\right),\ \alpha > 0 \tag{65}$$

We need to find the value of α. Let us compute the norm of $\mathbf{x_P} - \mathbf{x_C}$, as a function of α:

$$||\mathbf{x_P} - \mathbf{x_C}|| = \alpha||\mathbf{x_I} - \mathbf{x_C}|| \tag{66}$$

It is possible to combine Equations 66 and 64 to find the value of the parameter α:

$$\alpha = \frac{R}{||\mathbf{x_I} - \mathbf{x_C}||} \tag{67}$$

Finally, we compute the value of $\mathbf{x_P}$:

$$\mathbf{x_P} = \mathbf{x_C} + \alpha \left(\mathbf{x_I} - \mathbf{x_C}\right) = \mathbf{x_C} + R\,\frac{\mathbf{x_I} - \mathbf{x_C}}{||\mathbf{x_I} - \mathbf{x_C}||} \tag{68}$$

Thanks to Equation 68, it is possible to rapidly compute the coordinates of point P knowing all the geometrical parameters of the model.

Analytically knowing the position of the closest point depends on the complexity of the original geometry. Are currently implemented in the code two particular kinds of obstacles: circles and rectangles. This part of the application is object-oriented, i.e. it is possible to define the obstacles as new class instantiations with their own methods and attributes, which makes the code more flexible.

7.2.2 Velocity gradient approximation

By definition, point P is situated on the original boundary. As a consequence, the velocity $\mathbf{u_P}$ at this point is known because a boundary condition was defined. Nevertheless, what interests us is to know the value of velocity $\mathbf{u_I}$ at point I because this is the new boundary of the mesh (point P is not on the mesh boundary any more). To pass from $\mathbf{u_P}$ (whose value is known) to $\mathbf{u_I}$ (whose value we want to determine), we use the first-order Taylor approximation:

$$\mathbf{u_P} = \mathbf{u_I} + \boldsymbol{\nabla}\mathbf{u}\left(\mathbf{x_I}\right)\left(\mathbf{x_P} - \mathbf{x_I}\right) \tag{69a}$$

$$\mathbf{u_I} = \mathbf{u_P} - \boldsymbol{\nabla}\mathbf{u}\left(\mathbf{x_I}\right)\left(\mathbf{x_P} - \mathbf{x_I}\right) \tag{69b}$$

where $\mathbf{x_I}$ and $\mathbf{x_P}$ are the coordinates of points I and P respectively and $\boldsymbol{\nabla}\mathbf{u}\left(\mathbf{x_I}\right)$ the value of the gradient of $\mathbf{u}$ evaluated in $\mathbf{x_I}$. In most cases, the analytical expression of $\boldsymbol{\nabla}\mathbf{u}$ is unknown and that is why we have to approximate it by a function $\boldsymbol{\Pi}$. Since we are just interested in the nodal values of $\boldsymbol{\nabla}\mathbf{u}$, we interpolate $\boldsymbol{\Pi}$ using shape functions.

$$\boldsymbol{\Pi}\left(\mathbf{x}\right) = \sum_{k=1}^{n_{node}} N_k\left(\mathbf{x}\right)\boldsymbol{\Pi_k} \text{ with } \boldsymbol{\Pi}\left(\mathbf{x}\right) = \begin{pmatrix} \Pi_{11}(\mathbf{x}) & \Pi_{12}(\mathbf{x}) \\ \Pi_{21}(\mathbf{x}) & \Pi_{22}(\mathbf{x}) \end{pmatrix} \tag{70}$$

Let us just recall that, if $\mathbf{x_k}$ are the coordinates of node k, $N_k(\mathbf{x_k}) = 1$ and $N_k(\mathbf{x_i}) = 0$ for $i \neq k$.

Because $\boldsymbol{\Pi}(\mathbf{x})$ is a matrix function, we need to work term by term:

$$\boldsymbol{\Pi}_{ij}\left(\mathbf{x}\right) = \sum_{k=1}^{n_{node}} N_k\left(\mathbf{x}\right)\left(\boldsymbol{\Pi_k}\right)_{ij} \tag{71}$$

There, different strategies can be followed. We chose to approximate $\boldsymbol{\nabla u}$ in the sense of least square minimisation. The problem to be solved is thus:

$$\text{Find } \mathbf{\Pi}_{ij}(\mathbf{x}) \text{ s.t.} \Psi(\mathbf{\Pi}_{ij}) = \int_\Omega \left(\mathbf{\Pi}_{ij}(\mathbf{x}) - (\boldsymbol{\nabla}\mathbf{u})_{ij}\right)^2 d\Omega \text{ is minimum} \tag{72}$$

We substitute the expression of $\mathbf{\Pi}_{ij}(\mathbf{x})$ in the function $\Psi(\mathbf{\Pi}_{\boldsymbol{ij}})$, which gives:

$$\Psi\left((\mathbf{\Pi}_0)_{ij}, ..., (\mathbf{\Pi}_{n_{node}})_{ij}\right) = \int_\Omega \left(\left(\sum_{k=1}^{n_{node}} N_k(\mathbf{x}) \left(\mathbf{\Pi}_k\right)_{ij}\right) - (\boldsymbol{\nabla}\mathbf{u})_{ij}\right)^2 d\Omega \tag{73}$$

The variables of this minimisation problem are the nodal values of $\mathbf{\Pi}_{\boldsymbol{ij}}$ and we want to minimise Ψ with respect to each one of them. Let us compute each partial derivative.

$$\begin{aligned}\frac{\partial\Psi}{\partial\left(\mathbf{\Pi}_l\right)_{ij}} &= \frac{\partial}{\partial\left(\mathbf{\Pi}_l\right)_{ij}} \int_\Omega \left(\left(\sum_{k=1}^{n_{node}} N_k(\mathbf{x}) \left(\mathbf{\Pi}_k\right)_{ij}\right) - (\boldsymbol{\nabla}\mathbf{u})_{ij}\right)^2 d\Omega, \ \forall l \in \left[|1, n_{node}|\right] \\ &= \int_\Omega \frac{\partial}{\partial\left(\mathbf{\Pi}_l\right)_{ij}} \left(\left(\sum_{k=1}^{n_{node}} N_k(\mathbf{x}) \left(\mathbf{\Pi}_k\right)_{ij}\right) - (\boldsymbol{\nabla}\mathbf{u})_{ij}\right)^2 d\Omega \\ &= \int_\Omega 2N_l(\mathbf{x}) \left(\sum_{k=1}^{n_{node}} \left(N_k(\mathbf{x}) \left(\mathbf{\Pi}_k\right)_{ij}\right) - (\boldsymbol{\nabla}\mathbf{u})_{ij}\right) d\Omega\end{aligned}$$

Since we are looking for the values of $\left(\mathbf{\Pi}_l\right)_{ij}$ which minimise the function $\Psi(\mathbf{x})$, we equal the derivatives to zero:

$$\frac{1}{2}\frac{\partial\Psi}{\partial\left(\mathbf{\Pi}_l\right)_{ij}} = \int_\Omega N_l(\mathbf{x}) \left(\sum_{k=1}^{n_{node}} \left(N_k(\mathbf{x}) \left(\mathbf{\Pi}_k\right)_{ij}\right) - (\boldsymbol{\nabla}\mathbf{u})_{ij}\right) d\Omega = 0 \tag{74}$$

$$\int_\Omega N_l(\mathbf{x}) \left(\sum_{k=1}^{n_{node}} N_k(\mathbf{x}) \left(\mathbf{\Pi}_k\right)_{ij}\right) d\Omega = \int_\Omega N_l(\mathbf{x}) \left(\boldsymbol{\nabla}\mathbf{u}\right)_{ij} d\Omega \tag{75}$$

And finally:

$$\forall l \in \left[|1, n_{node}|\right], \sum_{k=1}^{n_{node}} \left(\int_\Omega N_k(\mathbf{x}) N_l(\mathbf{x}) d\Omega\right) \left(\mathbf{\Pi}_k\right)_{ij} = \int_\Omega N_l(\mathbf{x}) \left(\boldsymbol{\nabla}\boldsymbol{u}\right)_{ij} d\Omega \tag{76}$$

We now need to deal with the term $(\boldsymbol{\nabla}\mathbf{u})_{ij}$ because its analytical expression is unknown. We also need to approximate its value, interpolating the nodes values of $\mathbf{u}$.

$$
\begin{aligned}
(\boldsymbol{\nabla}\mathbf{u})_{ij} &= \frac{\partial u_i}{\partial x_j} \text{ by definition} \\
&= \frac{\partial}{\partial x_j}\sum_{k=1}^{n_{node}} (\mathbf{u_k})_i \, N_k(\mathbf{x}) \\
&= \sum_{k=1}^{n_{node}} (\mathbf{u_k})_i \, \frac{\partial N_k}{\partial x_j}
\end{aligned}
$$

Let us gather all the elements we have obtained so far. We get that:

$$
\forall l \in \left[|1, n_{node}|\right], \; \sum_{k=1}^{n_{node}} \left(\int_\Omega N_k(\mathbf{x}) N_l(\mathbf{x}) d\Omega\right) (\boldsymbol{\Pi}_k)_{ij} = \sum_{k=1}^{n_{node}} \left(\int_\Omega N_l(\mathbf{x}) \frac{\partial N_k}{\partial x_j} d\Omega\right) (\mathbf{u_k})_i \tag{77}
$$

Let us remember that the objective is to compute the nodal values $(\boldsymbol{\Pi}_k)_{ij}$ so that we can compute the approximation of the gradient of $\mathbf{u}$ at each node of the mesh. Equation 77 actually describes a linear system of equations in which the nodal values $(\boldsymbol{\Pi}_k)_{ij}$ are the unknowns. Let us write Equation 77 using matrices notations:

$$
\begin{pmatrix}
\int_\Omega N_1 N_1 & \int_\Omega N_1 N_2 & \dots & \int_\Omega N_1 N_{n_{node}} \\
\int_\Omega N_2 N_1 & \int_\Omega N_2 N_2 & \dots & \int_\Omega N_2 N_{n_{node}} \\
\dots & \dots & \dots & \dots \\
\int_\Omega N_{n_{node}} N_1 \, d\Omega & \int_\Omega N_{n_{node}} N_2 & \dots & \int_\Omega N_{n_{node}} N_{n_{node}}
\end{pmatrix}
\begin{pmatrix}
(\boldsymbol{\Pi}_1)_{ij} \\ (\boldsymbol{\Pi}_2)_{ij} \\ \dots \\ (\boldsymbol{\Pi}_{n_{node}})_{ij}
\end{pmatrix}
$$
$$
= \begin{pmatrix}
\int_\Omega N_1 \frac{\partial N_1}{\partial x_j} & \int_\Omega N_1 \frac{\partial N_2}{\partial x_j} & \dots & \int_\Omega N_1 \frac{\partial N_{n_{node}}}{\partial x_j} \\
\int_\Omega N_2 \frac{\partial N_1}{\partial x_j} & \int_\Omega N_2 \frac{\partial N_2}{\partial x_j} & \dots & \int_\Omega N_2 \frac{\partial N_{n_{node}}}{\partial x_j} \\
\dots & \dots & \dots & \dots \\
\int_\Omega N_{n_{node}} \frac{\partial N_1}{\partial x_j} & \int_\Omega N_{n_{node}} \frac{\partial N_2}{\partial x_j} & \dots & \int_\Omega N_{n_{node}} \frac{\partial N_{n_{node}}}{\partial x_j}
\end{pmatrix}
\begin{pmatrix}
(\mathbf{u_1})_i \\ (\mathbf{u_2})_i \\ \dots \\ (\mathbf{u_{n_{node}}})_i
\end{pmatrix}
$$

$$
\boldsymbol{M_\Pi} = \begin{pmatrix}
\int_\Omega N_1 N_1 & \int_\Omega N_1 N_2 & \dots & \int_\Omega N_1 N_{n_{node}} \\
\int_\Omega N_2 N_1 & \int_\Omega N_2 N_2 & \dots & \int_\Omega N_2 N_{n_{node}} \\
\dots & \dots & \dots & \dots \\
\int_\Omega N_{n_{node}} N_1 \, d\Omega & \int_\Omega N_{n_{node}} N_2 & \dots & \int_\Omega N_{n_{node}} N_{n_{node}}
\end{pmatrix}
\qquad
\boldsymbol{\Pi}_{ij} = \begin{pmatrix}
(\boldsymbol{\Pi}_1)_{ij} \\ (\boldsymbol{\Pi}_2)_{ij} \\ \dots \\ (\boldsymbol{\Pi}_{n_{node}})_{ij}
\end{pmatrix} \tag{78}
$$

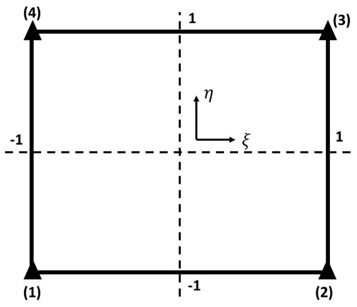

Figure 46: Reference element for $\mathbf{M_\Pi}$ and $\mathbf{A_j}$ matrices integration

$$\mathbf{A_j} = \begin{pmatrix} \int_\Omega N_1 \frac{\partial N_1}{\partial x_j} & \int_\Omega N_1 \frac{\partial N_2}{\partial x_j} & \dots & \int_\Omega N_1 \frac{\partial N_{n_{node}}}{\partial x_j} \\ \int_\Omega N_2 \frac{\partial N_1}{\partial x_j} & \int_\Omega N_2 \frac{\partial N_2}{\partial x_j} & \dots & \int_\Omega N_2 \frac{\partial N_{n_{node}}}{\partial x_j} \\ \dots & \dots & \dots & \dots \\ \int_\Omega N_{n_{node}} \frac{\partial N_1}{\partial x_j} & \int_\Omega N_{n_{node}} \frac{\partial N_2}{\partial x_j} & \dots & \int_\Omega N_{n_{node}} \frac{\partial N_{n_{node}}}{\partial x_j} \end{pmatrix} \qquad \mathbf{U_i} = \begin{pmatrix} (\mathbf{u_1})_i \\ (\mathbf{u_2})_i \\ \dots \\ (\mathbf{u_{n_{node}}})_i \end{pmatrix} \tag{79}$$

With these different notations, the linear system we have to solve becomes:

$$\boldsymbol{M}_{\boldsymbol{\Pi}} \boldsymbol{\Pi_{ij}} = \mathbf{A_j} \mathbf{U_i} \, , \; i \in \{1;2\}, \; j \in \{1;2\} \tag{80}$$

7.2.3 Integrals computation

The last thing to determine is the way to compute the integrals. As before, the matrices $\mathbf{M_\Pi}$ and $\mathbf{A_j}$ are computed for each element of the mesh and then assembled. However, we will not use the same reference element as for the computation of the stiffness and mass matrices because we are not computing the integrals over the four Gauss points as depicted in Figure 19. We will integrate the matrices $\boldsymbol{M_\pi}$ and $\mathbf{A_j}$ computing the sum over the four corner nodes as depicted in Figure 46, where ▲ represent the integration points.

This way, the matrix $\mathbf{M_\Pi}$ is diagonal[16], which simplifies the computation of the vector of unknowns $(\mathbf{\Pi}_k)_{ij}$. We have to solve four different systems because $i = 1$ or 2 and $j = 1$ or 2[17].

7.2.4 Apply boundary conditions

Let us sum up the results all we got until now. First of all, before beginning the time integration loop, we should look for the closest points which are on the original boundary of the geometry

[16]Let us remember that $N_k(\mathbf{x_k}) = 1$ and $N_k(\mathbf{x_i}) = 0$ for $i \neq k$

[17]In the case of a 3D problem, we would solve nine different linear systems of equations because the velocity $\mathbf{u}$ would have three components and there would be three space variables x, y and z

for each point of the structured mesh boundary. Still before beginning the time integration loom, the matrices $\boldsymbol{M_\Pi}$ and $\mathbf{A_j}$ ($j = 1; 2$) should be computed by computing the elemental ones and assembling them. These matrices remain constant all along time and should be computed once and for all.

Then we enter in the time integration loop using the techniques presented in Section 6. Within each time step, Algorithm 5 should be applied to impose the required boundary conditions.

Algorithm 5 Applying boundary conditions

```
Require: $\boldsymbol{M_\Pi}$, $\mathbf{A_1}$, $\mathbf{A_2}$, $\mathbf{U}^n$, $\Gamma_{mesh}$
  for $i = 1, 2$ do
    for $j = 1, 2$ do
      Solve $\boldsymbol{M_\Pi}\boldsymbol{\Pi}_{ij} = \mathbf{A_j}\boldsymbol{U_i^n} \to \boldsymbol{\Pi}_{ij}$
      Store $\boldsymbol{\Pi}_{ij}$ for each node of the mesh
    end for
  end for
  for node in $\Gamma_{mesh}$ do
    Get closest point P
    Compute $\mathbf{u_{node}} = \mathbf{u_P} - \boldsymbol{\Pi}_{node}\left(\mathbf{x_P} - \mathbf{x_I}\right)$
    Apply boundary condition to $\mathbf{U}^n$
  end for
```

Let us say that the Γ_{mesh} list contains all the nodes which are on the boundary of the structured mesh. Once we update U^n, we use it to compute U^{n+1} using a Forward Euler or Fractional Step schemes for instance.

8 Resolution of 2 calibration problems

From now on, we have all the elements to validate and/or calibrate the B-bar element for fluids. To do so, we'll study 2 very basic examples, for which analytical solution is "known". The comparison between analytical solution and the numerical solution will be a part of the simulation validation test. The 2 problems we consider are quite different because distinct phenomena are at stake in each one.

On one hand, Figure 47 represents the most basic viscosity-driven problem we can imagine. There the effects of viscosity are predominant and the compressibility (due to pressure) is negligible. There is no volume change because the displacement is tangential. This is a very well-known problem, already studied in the XIX century by Maurice Couette.

On the other hand, the problem presented in Figure 48 implies to consider the effect of compressibility. In this channel, the velocity propagates thanks to the volume change of the elements which transmit it to the following ones, such as in sound propagation.

A priory, we do not know if the B-bar element "reacts" the same way when dealing with viscosity effects or compressibility ones. Studying two different problems will bring us useful information.

8.1 Viscosity-driven problem / Couette flow

We impose a tangential velocity at the top of the domain, normal velocity being null. The bottom of the domain is supposed to be fix (null tangential and normal velocity). The velocity will "propagate" up to the bottom of the domain thanks to the viscosity of the fluid, which can be seen as different layers; viscosity acts as friction between each one.

In our case, several assumptions are to be made in order to find the analytical PDE:

- The problem is uni-dimensional in space, the variable x is not relevant. Everything only depends on y since $L_x >> L_y$.
- The vertical velocity in the whole domain is null: $v = 0$

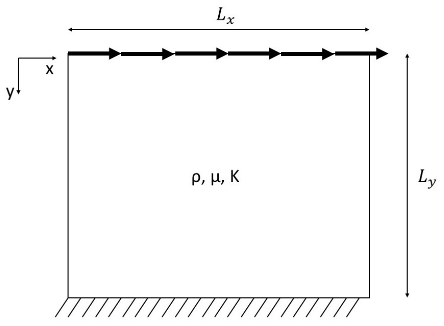

Figure 47: A viscosity-driven problem

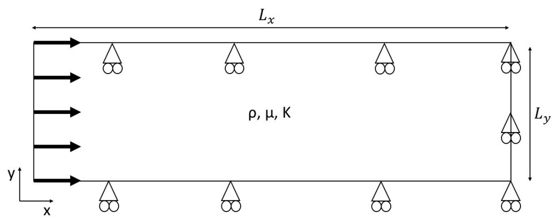

Figure 48: A compressibility-driven problem

- The fluid is quasi-incompressible.
- The parameters ρ, μ, and K are constant all over the domain
- We neglect the effect of gravity

As a consequence of all these hypotheses, the velocity field can be simplified and its gradient computed.

$$\mathbf{u} = \begin{pmatrix} u(x,y) \\ v(x,y) \end{pmatrix} = \begin{pmatrix} u(y) \\ 0 \end{pmatrix} \qquad \boldsymbol{\nabla u} = \begin{pmatrix} 0 & \frac{\partial u}{\partial y} \\ 0 & 0 \end{pmatrix} \tag{81}$$

We can easily see that $\mathbf{u} \cdot \boldsymbol{\nabla u} = 0$, which means that the convective term of the Navier-Stokes equation is null.

We compute the strain rate tensor.

$$\boldsymbol{\nabla^S u} = \begin{pmatrix} 0 & \frac{1}{2}\frac{\partial u}{\partial y} \\ \frac{1}{2}\frac{\partial u}{\partial y} & 0 \end{pmatrix}$$

The next step is to obtain the stress tensor using the constitutive equation of Newtonian fluid. One is able to compute the divergence of the stress tensor as well.

$$\boldsymbol{\sigma} = -p\mathbf{I} + 2\mu\boldsymbol{\nabla^S u} = \begin{pmatrix} -p & \mu\frac{\partial u}{\partial y} \\ \mu\frac{\partial u}{\partial y} & -p \end{pmatrix} \qquad \boldsymbol{\nabla} \cdot \boldsymbol{\sigma} = \begin{pmatrix} -\frac{\partial p}{\partial x} + \mu\frac{\partial^2 u}{\partial y^2} \\ \mu\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial y}\right) - \frac{\partial p}{\partial y} \end{pmatrix} = \begin{pmatrix} \mu\frac{\partial^2 u}{\partial y^2} \\ -\frac{\partial p}{\partial y} \end{pmatrix}$$

We finally apply the momentum balance equation, where we neglect the gravity term.

$$\boldsymbol{\nabla} \cdot \boldsymbol{\sigma} = \rho\frac{\partial \mathbf{u}}{\partial t}$$

By equalling the right and left-hand sides, we get two partial differential equations (PDE) which would enable us to determine the velocity $\mathbf{u}$ and the pressure p, taking into account the boundary conditions of the problem.

$$\begin{pmatrix} \mu\frac{\partial^2 u}{\partial y^2} \\ -\frac{\partial p}{\partial y} \end{pmatrix} = \begin{pmatrix} \rho\frac{\partial u}{\partial t} \\ 0 \end{pmatrix}$$

$$\frac{\partial u}{\partial t} - \frac{\mu}{\rho}\frac{\partial^2 u}{\partial x^2} = 0 \tag{82a}$$

$$\frac{\partial p}{\partial y} = 0 \tag{82b}$$

To finish, we use the mass conservation equation for quasi-incompressible fluid, assuming ρ is constant all over the domain.

$$\frac{\partial p}{\partial t} + K\,\nabla \cdot \mathbf{u} = \frac{\partial p}{\partial t} = 0$$

In the end, we have $^{\partial p}/_{\partial x} = 0$, $^{\partial p}/_{\partial y} = 0$ and $^{\partial p}/_{\partial t} = 0$ so we can conclude $p = 0$ in the whole domain. This is a more rigourous demonstration to show that the effects of pressure are negligible in this type of flow.

Equation 82a is a diffusion one. There is no analytical solution for the transient problem, even using the separation of variable. However, theory about this type of equation tells us we can define a diffusion coefficient $D = {}^{\mu}/_{\rho}\;(m^2/s) = {}^{L^2}/_{\tau}$, where L is a characteristic length of the problem and τ a characteristic time of the problem.

$$\tau = \frac{L^2\,\rho}{\mu}$$

We have to take care about the signification of τ. We are not saying that we reach the equilibrium state for $t = \tau$. We are only giving an order of magnitude of the duration of the process. Let us imagine the diffusion of a certain substance in a glass of water. One can relate the time needed by substance to diffuse in the whole domain and the size of the glass using the coefficient diffusion D. Of course this is only an order of magnitude. It is possible to apply the same reasoning to our particular problem. If one considers a particle of fluid at $y = L$, how long should he wait to see the particle move? We can get an order of magnitude using τ. In any case, there is not a clear delimitation between transient and stationary problem.

The stationary velocity field is very easy to compute since $^{\partial^2 u}/_{\partial x^2} = 0$.

$$u_{stationary} = v_0 \frac{L_y - y}{L_y}$$

8.2 Compressibility-driven problem

The second problem we want to solve is presented in Figure 48. We impose a given velocity at one side of the channel to study the propagation of this velocity all along the domain. The normal velocity on the up and down sides of the domain is null (there is no penetration of fluid into the wall of the channel). The prescribed displacement at the left side could be constant or sinusoidal depending on the type of problem we are solving.

As previously, we have to make some assumptions, in order to solve it analytically and then compare the results with the numerical simulation:

- We neglect gravity
- The vertical velocity is null in the whole domain: $v = 0$
- 1D problem, all variables only depend on x: $u(x, y) = u(x)$
- Quasi-incompressible fluid
- $||\,\mu\,\partial_{xx}u\,|| << ||\,\rho\,\partial_t u\,||$
- $||\,\rho u\,\partial_x u\,|| << ||\,\rho\,\partial_t u\,||$
- Infinite channel to avoid the rebound of waves.

We follow the same steps as before: compute the vector $\boldsymbol{\nabla^S u}$, then the tensions $\boldsymbol{\sigma}$ to finally apply the momentum balance equation and the mass conservation one.

$$\boldsymbol{\nabla^S u} = \begin{pmatrix} \frac{\partial u}{\partial x} & 0 \\ 0 & 0 \end{pmatrix} \qquad \boldsymbol{\nabla} \cdot \mathbf{u} = \frac{\partial u}{\partial x} \qquad \boldsymbol{u\nabla u} = \begin{pmatrix} u\frac{\partial u}{\partial x} \\ 0 \end{pmatrix}$$

In this particular case, we have $\boldsymbol{\nabla^S u} = \boldsymbol{\nabla u}$.

In this second problem, the convective term $\boldsymbol{u\nabla u}$ and $\boldsymbol{\nabla} \cdot \mathbf{u}$ are not null. The velocity divergence is not null because there are changes of volume.

We compute the stresses within the fluid. An interesting point to be remarked is that there is no shear stress in the fluid. The effects of the fluid viscosity are thus considerably reduced.

$$\boldsymbol{\sigma} = \begin{pmatrix} -p + 2\mu\frac{\partial u}{\partial x} & 0 \\ 0 & -p \end{pmatrix}$$

Afterwards, we apply the momentum balance (neglecting the gravity term) to obtain 2 PDEs.

$$\boldsymbol{\nabla} \cdot \boldsymbol{\sigma} = \begin{pmatrix} -\frac{\partial p}{\partial x} + 2\mu\frac{\partial^2 u}{\partial x^2} \\ -\frac{\partial p}{\partial y} \end{pmatrix} = \rho \begin{pmatrix} \frac{\partial u}{\partial t} \\ 0 \end{pmatrix} + \rho \begin{pmatrix} u\frac{\partial u}{\partial x} \\ 0 \end{pmatrix}$$

$$\partial_y p = 0 \tag{83a}$$

$$\rho\,\partial_t u = 2\mu\,\partial_{xx}u - \partial_x p - \rho u\,\partial_x u \tag{83b}$$

The last PDE of the problem is given by the mass conservation equation. $\partial_y p$ is still null because we did not take into account the gravity. In the case of a thin channel where $L_x >> L_y$, this is a reasonable assumption.

$$\partial_t p + K\,\partial_x u = 0 \tag{84}$$

It is time we applied all the hypotheses presented before, which gives the following simplified equations.

$$\partial_y p = 0 \tag{85a}$$

$$\rho\,\partial_t u = -\partial_x p \tag{85b}$$

$$\partial_t p + K\,\partial_x u = 0 \tag{85c}$$

The combination of Equations 85b and 85c will give a characteristic Alembert Equation which describes the wave propagation in our medium.

The demonstration of the Alembert equation is based on the expression of ∂_{tt} as a function of ∂_{xx}.

Proof.

$$\begin{aligned}\rho\,\partial_{tt} u &= -\partial_t(\partial_x p) \text{ using Equation 85b}\\ &= -\partial_x(\partial_t p)\end{aligned}$$

because we assume that t and x are independent variables

$$= K\,\partial_{xx} u \text{ using Equation 85c}$$

$\square$

We finally obtain the Alembert equation:

$$\partial_{xx} u - \frac{1}{c^2}\,\partial_{tt} u = 0 \tag{86}$$

Where $c = \sqrt{\frac{K}{\rho}}\ [m/s]$

As in the first problem, there is no analytical solution for a general transient problem. We will need to impose specific boundary conditions in order to get one. However, we can define a wave propagation velocity c, which the characterise the wave propagation in our medium. This will be very useful to validate our numerical simulation.

9 Structure of the Python script

We have all the elements to practically implement the solver before testing it on practical cases. The objective of this section is to present some important implementation without entering all its details. All the scripts I wrote can be found in Section 12. As for technical data, to run the scripts, I used my personal computer which has a processor Intel i5 (1.8 GHz) and a RAM of 4 Go.

9.1 Code structure

The application script has been written from scratch using Python 3.7 (see [VRD99] for the documentation of this version). This was very challenging to go from nothing up to writing the whole post-process of the solution. The goal is to study the behaviour of a Q1P0 finite elements mesh (over a rectangular domain) associated with the different time integration strategies we previously defined as well as the Shifted Boundary Conditions method and the BFECC method.

Figure 49 presents the general structure of the script. It is compound of 6 different modules which interact with the main one. As for now, this is not an object-oriented code, which would be a further great improvement. To make the code readable and facilitate the different tests we would operate, I tried to make it as flexible as possible, splitting it in many functions, within each module. Afterwards, each module and its interactions with the other ones are detailed. The aim is to highlight the most important elements of the numerical simulation without annoying the reader with all programming details.

Let us remember that the final objective of this work is the implementation of the new solver we are designing in Kratos. This means that, in this future implementation, we will not have to design neither the pre-process nor the post-process parts. The inputs of the solver in Kratos are directly the connectivity matrix gathering all the elements and their characteristics, the matrix of coordinates and the boundary conditions. However, in the Python script, a pre and post-process were needed so that the code works correctly.

9.2 I/O (Input/Output)

Thinking about the application relates itself with its environment is a really important part of the coding because we have to adapt the structure of the input parameters to the solver.

9.2.1 Input

In order to do the simulation of several different cases, the user is able to define various inputs:

- The fluid characteristics μ (viscosity), ρ (density) and K (bulk modulus) which are, in a first attempt, supposed to be constant all over the domain.
- The size of the square domain, i.e. L_x and L_y.

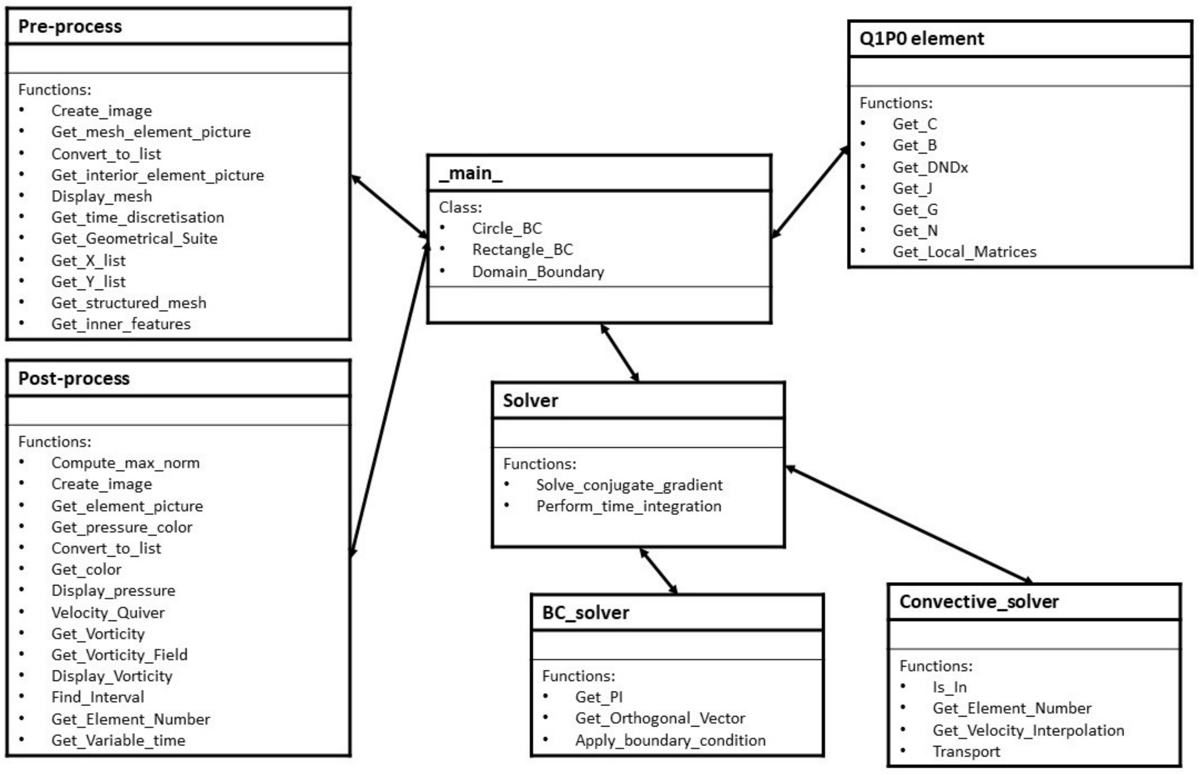

Figure 49: The Python script processes

- The number of elements in the horizontal and vertical directions, respectively n_x and n_y.
- The time step dt and the total simulation duration t_{max}.
- The time integration strategy. Up to now are implemented the Forward and Backward Euler schemes and the Fractional Step splitting.
- The boundary conditions.

Finding a flexible and efficient way to impose the boundary conditions was one of the most tricky part of the code. Up to now, we are able to simulate uniform, logarithmic, tangential and parabolic (on each border of the rectangular domain) velocity boundary conditions.

9.2.2 Output

The outputs of the simulation are a graphical representation (basically a picture) of the velocity and pressure fields in the domain. As we are dealing with a transient problem (which depends on time), we need to this post-process for various time steps and then animate it. The post-process of the pressure is greatly simplified because we are studying rectangular Q1P0 elements meshes, which means that the pressure is constant all over the element.

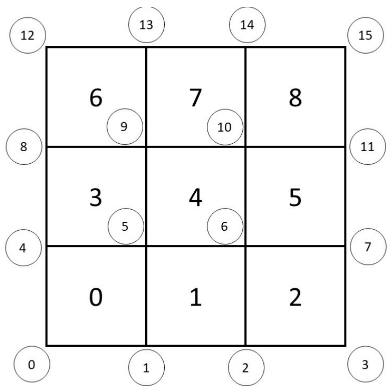

Figure 50: Global numbering of elements and nodes

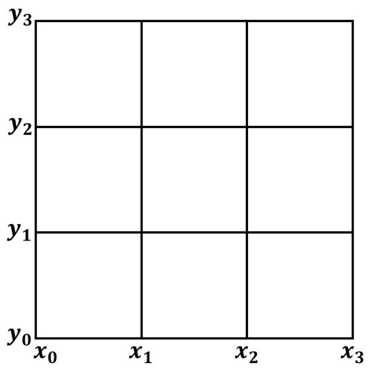

Figure 51: Space discretisation

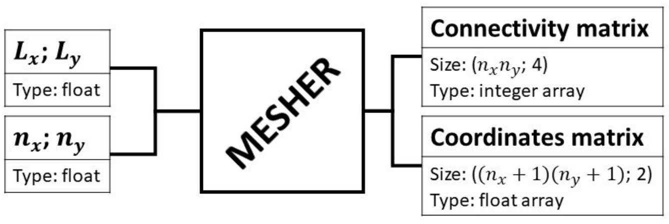

Figure 52: Mesher black box

9.3 Inputs pre-process

9.3.1 Mesh generation

Knowing the value of L_x, L_y, n_x and n_y, we are able to build a structured quadrilateral mesh of our domain[18]. We need to build a coordinates matrix $\mathbf{XY}$ to store the x and y coordinates of the mesh points and a connectivity matrix $\mathbf{M_{connectivity}}$. The total number of points in the mesh is $(n_x+1)(n_y+1)$, the total number of elements is $n_x n_y$; $\mathbf{XY}$ and $\mathbf{t}$ are thus a $\big((n_x+1)(n_y+1), 2\big)$ and $\big(n_x n_y, 4\big)$ Numpy arrays, respectively.

There we need to explain the global nodes and elements numbering. Figure 50 presents an example of a structured mesh we use in the numerical simulation. Each element and node has a global numbering (numbers within circles in Figure 50). We decided to label the nodes and elements from the bottom left up to the right and the top. For instance, the coordinates of node 6 are (x_2, y_1). As a consequence, it appears clearly that:

$$\mathbf{XY} = \begin{pmatrix} x_0 & x_1 & x_2 & \cdots & x_3 \\ y_0 & y_0 & y_0 & \cdots & y_3 \end{pmatrix}^T$$

[18]Let us recall that Q1P0 finite elements behave very poorly with unstructured meshes. It is really important to work with structured ones.

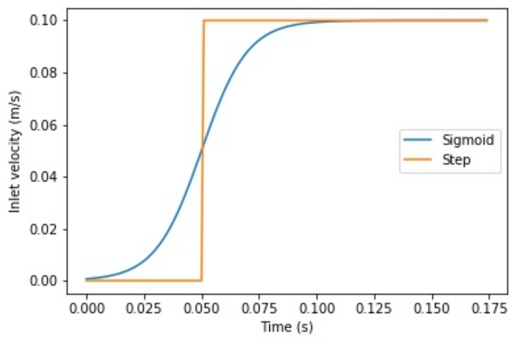

Figure 53: Time sigmoid inlet velocity

The i-th line of $\mathbf{XY}$ contains the x and y coordinates of point number i.

Then, we express the connectivity matrix $\mathbf{t}$. At the i-th line of $\mathbf{t}$ can be found the global numbers of the nodes of element i. For instance, considering the mesh of Figure 50, the nodes of the element 5 are 6, 7, 11 and 10. The fifth line of $\mathbf{t}$ will thus be $(6, 7, 11, 10)$. More generally:

$$\mathbf{M_{connectivity}} = \begin{pmatrix} 0 & 1 & 5 & 4 \\ 1 & 2 & 6 & 5 \\ 2 & 3 & 7 & 6 \\ \dots & \dots & \dots & \dots \\ 10 & 11 & 15 & 14 \end{pmatrix}$$

With the connectivity matrix $\mathbf{t}$ and the coordinates one $\mathbf{XY}$, we are able to completely describe the mesh of the domain.

9.3.2 Inlet velocity

Imposing to rapidly velocity boundary conditions is a source of numerical instability. In order to solve this issue, we choose to impose a sigmoid profile (in time) of inlet velocity (see Figure 53). Sigmoid functions are C^∞ ones, which means that all their derivatives are continuous. In comparison, neither "step" functions (see Figure 53) nor their derivatives are continuous, which can have severe consequences over the numerical simulation.

As for spatial features, we are able to impose uniform (Figure 54), parabolic (Figure 55) and logarithmic (Figure 56) inlet profiles. Depending on the type of problem we want to solve, it is necessary to choose between one of them. Parabolic profile is useful to model the fluid flow within a pipe with viscous fluids and the logarithmic one approximately describes the wind profile in the atmosphere.

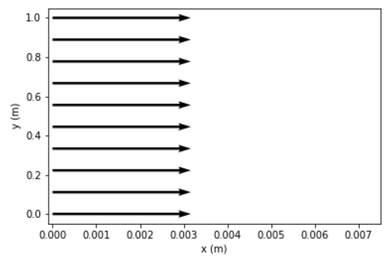

Figure 54: Example of uniform inlet

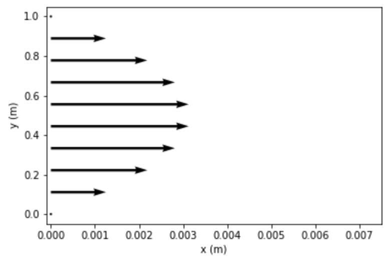

Figure 55: Example of parabolic inlet

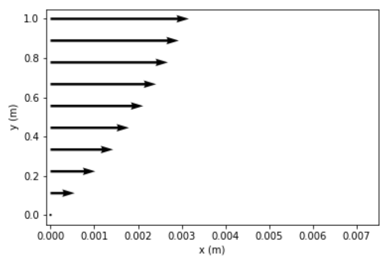

Figure 56: Example of logarithmic inlet

9.4 Implementation summary

Algorithm 6 very briefly sums up the main steps of the main module algorithm and how they should be organised. Getting inner nodes and elements is the first part of the Shifted Boundary Conditions method. Algorithm 7 presents the different stages performed within one time integration and how the different algorithm we detailed before are linked. Of course, the numerical scheme to pass from $\mathbf{U^n}$, $\mathbf{P^n}$ to $\mathbf{U^{n+1}}$, $\mathbf{P^{n+1}}$ depends on the method we previously defined (Forward Euler, Backward Euler or Fractional Step splitting).

Algorithm 6 Main module algorithm

```
Require: $v_{inlet}, \mu, \rho, K, L_x, L_y, n_x, n_y, \mathbf{U_0}, \mathbf{P_0}, dt, t_{max}$
  Get time discretisation
  Get connectivity and coordinates matrices
  Get inner nodes and elements
  Make the assembly of the global stiffness, mass and gradient matrices
  Lump mass matrices
  Integrate discretised equations in time
  Do solution post-process
```

Algorithm 7 Time integration algorithm

```
Require: $\mathbf{M_{UU}}, \mathbf{M_{PP}}, \mathbf{D}, \mathbf{K_{UU}}, \delta t, \mathbf{U_0}, \mathbf{P_0}$
  Apply boundary conditions $\rightarrow \mathbf{U^n}$
  Store $\mathbf{U^n}$
  Solve convective equation $\rightarrow \tilde{\mathbf{U}}^{\mathbf{n+1}}$
  Perform one step time integration $\rightarrow \mathbf{U^{n+1}}, \mathbf{P^{n+1}}$
```

9.5 Solution post-process

The results of the loop for time integration are 2 matrices **solU** and **solP** which contain the values of the velocity and pressure DOFs at each time step.

9.5.1 Pressure post-process

In a Q1P0 elements mesh, the pressure is uniform within the elements. The best way to post-process the outputs of the time integration loop is to represent coloured squares where the colour stands for the value of the pressure in the element. Blue stands for the minimum pressure, red for maximum one.

One advantage with the pressure post-process is that the matrix **solP** directly contains the pressure in every element; that is not the case with the matrix **solU**. Let us say that $P_i(t_n)$ is the pressure in element i at time t_n.

$$\mathbf{solP} = \begin{pmatrix} P_0(t_0) & P_0(t_1) & ... & P_0(t_{nbSteps}) \\ P_1(t_0) & P_1(t_1) & ... & P_1(t_{nbSteps}) \\ ... & ... & ... & ... \\ P_n(t_0) & P_n(t_1) & ... & P_n(t_{nbSteps}) \end{pmatrix}$$

For instance, we get all the pressures at the last time step by selecting the last column of **solP** and we post-process this column vector.

We use the Python library PIL which enables us to operate on JPEG images. Knowing the distances L_x and L_y and the total number of B-bar elements in the mesh, we are going to draw squares of ${}^{L_x}/_{n_x}$ pixels in the horizontal direction and ${}^{L_y}/_{n_y}$ pixels in the vertical one. Each square will be of one distinct colour to represent the pressure in the equivalent element. The blue colour in a square stands for the minimum pressure, the red for the maximum one.

10 Practical case studies

After having implemented every part of the solver, we will run several test cases in order to verify its convergence and derive some useful knowledge for the future implementation in a production-ready code.

The results presented in this section are organised in an increasing level of complexity:

- First of all, the Couette flow problem. This is the most basic and easy-to-solve one because it only involves viscosity which makes the numerical simulation stable, and it is quite easy to have a first intuition of what the velocity fluid should look like. We will support it with the theoretical studyt we realised in Section 8. This is the first stage of validation to show that the solver we designed is able to deal with viscous effects.

- The channel with uniform and constant inlet problem. It is more complicated than the Couette flow problem since it involves compressibility effects (and viscosity ones are almost null). This tends to make the simulation numerically unstable because of the bulk modulus high value. As previously, the theoretical study of Section 8 will be useful to consolidate the numerical results in comparison with the analytical ones.

- The fluid flow around a cylinder for different values of the Reynolds number. We will study the convergence of the numerical simulation when refining the mesh and using different time steps. This is a basic but quite complicated problem because it mixes the effects of viscosity and convection. At high values of the Reynolds number, vertices may appear and the solver should be able to reproduce this phenomenon. In addition, this problem will enable us to validate the implementation of the Shifted Boundary Conditions, which was absent in the 2 previous problems, because there were no obstacles.

- Finally, a more realistic problem of air circulation in a city will be implemented and studied in order to determine if our solver could be used in production-ready codes for solving future engineering problems.

10.1 Couette flow problem

The settings of the problem are identical to the ones of Section 8. Figure 57 presents the result of the simulation if we only consider two boundary conditions: a tangential speed at the top of the domain and a null one in the lower side, like depicted in Figure 47. It appears a vertical velocity next to both left and right sides just like small vortices, which is not consistent with the idea one can have of the flow a priori. To understand why, it is necessary to compute the expression of the tension $\boldsymbol{\sigma}$ in the domain taking into account that $u = 1 - y$, which is the analytical stationary solution of the problem. In this case, we have that $\partial_y u = -1$ and $p = 0$.

$$\boldsymbol{\sigma} = 2\mu \boldsymbol{\nabla}^{\boldsymbol{S}} \boldsymbol{u} = \begin{pmatrix} 0 & -\mu \\ -\mu & 0 \end{pmatrix}$$

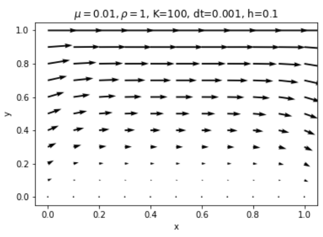

Figure 57: Free Couette flow

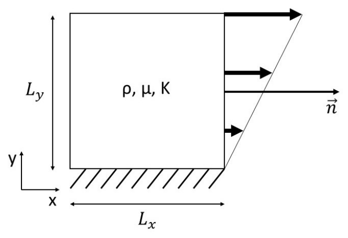

Figure 58: Stationary Couette solution

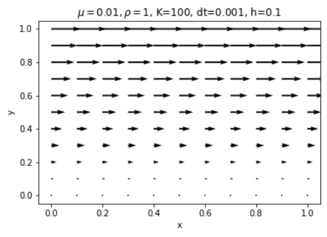

Figure 59: Constrained Couette flow

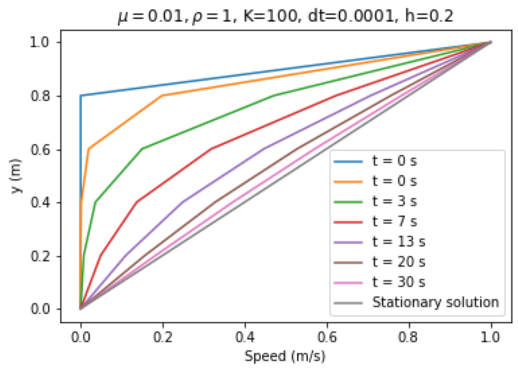

Figure 60: Couette flow results with FE and $h = 0.2$

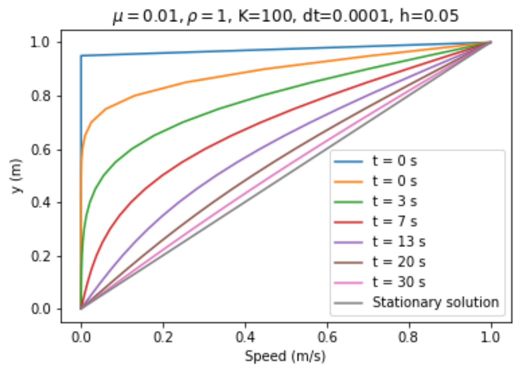

Figure 61: Couette flow results with FE and $h = 0.05$

We can compute the tension at the right side of the domain, considering the normal $\mathbf{n} = (1\ 0)^T$.

$$\mathbf{t} = \boldsymbol{\sigma} \cdot \mathbf{n} = \begin{pmatrix} 0 \\ -\mu \end{pmatrix}$$

This last equation means that the linear flow depicted in Figure 58 is not compatible with a free border at the left and right sides of the domain. If we want to have a perfectly horizontal flow, we need to impose two more boundary conditions. The vertical velocity should be null at the left and right sides of the domain, i.e. $v(x = 0, y) = 0$ and $v(x = L_x, y) = 0$.

Figure 59 presents the kind of results one gets when imposing these new boundary conditions: the small vortices disappeared because the flow is more constrained and the vertical component of the velocity is null in the whole domain.

Figures 60 and 61 are vertical cuts of the velocity field considering different element sizes h. The horizontal axis represents the transverse speed u whereas the vertical one is the vertical coordinate y. The solution at different time steps and the analytical stationary one are represented. These graphs were obtained using a Forward Euler time integration.

These results are consistent with the analytical study we previously did. We had defined a characteristic time $\tau = L^2 \cdot \rho/\mu$. In this case, $\tau = 25$ considering $L = 0.5$, which corresponds to the point in the middle of the domain. We ran the simulation up to $t = 30$ and the velocity

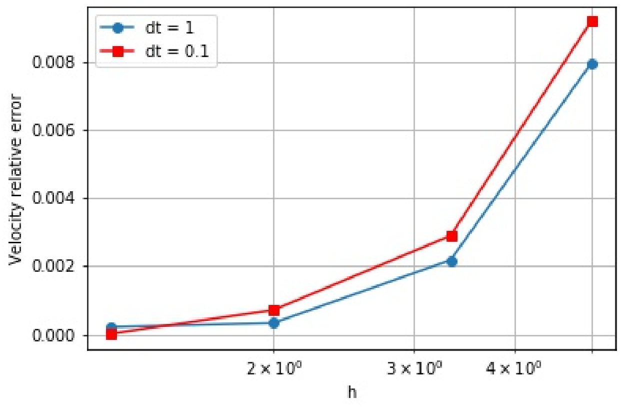

Figure 62: Mesh convergence for the Couette flow problem

field almost reached an equilibrium. Moreover, these results are consistent with the idea one can have of the problem. First the upper part of the domain starts moving, which, by the action of viscosity, makes the lower part move too.

10.1.1 Simulation convergence

Results obtained in the Couette flow case are consistent. However, we need to perform a more quantitative analysis of the results in order to prove the robustness of the solver we designed, including all the elements we previously described. To do that, we need to perform a given simulation (with the same fluid characteristics and the same boundary conditions) using finer meshes. If the solver is robust, the solution should converge when refining the mesh.

Let us say that we are to study the convergence of the numerical solution using a series of meshes $M_i = (V_i, E_i)$, where V_i is the set of nodes and E_i the set of elements, represented by the coordinates and connectivity matrices respectively, and characterised by its elements size h_i. We define the velocity relative error as

$$e_{u,i} = \frac{||\mathbf{U}(h_i) - \mathbf{U}(h_{i+1})||_2}{||\mathbf{U}(h_i)||_2} \tag{87}$$

where $||\cdot||_2$ refers to the L^2-norm as defined in Section 2 and $\mathbf{U}(h)$ represents the nodal values of the velocity field using a certain mesh size h.

The same way, it is possible to define the pressure relative error

$$e_{p,i} = \frac{||\mathbf{P}(h_i) - \mathbf{P}(h_{i+1})||_2}{||\mathbf{P}(h_i)||_2} \tag{88}$$

Figure 62 presents the results of the convergence analysis for the velocity field. The horizontal axis stands for the element size h, in logarithm scale. The velocity relative error $e_{u,i}$ is plotted

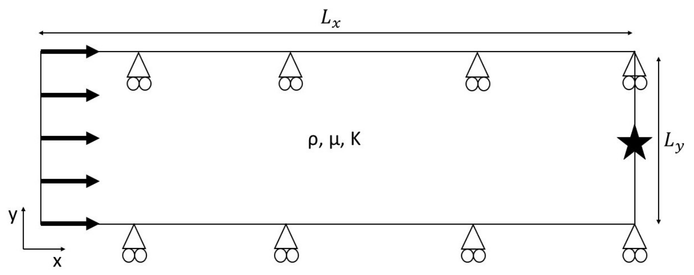

Figure 63: Channel configuration

in the vertical one. As for the pressure, since it is null in this type of flow, the results we get are not relevant[19] and that is why they are not presented.

The first conclusion we can draw from Figure 62 is that the solver is robust since the relative error tends to 0 when the element size tends to 0, and this does not depend on the time step dt. Very rapidly we reach very low values of the relative error, in the order of 0.2%, even with coarse meshes. The time step has a low influence over the convergence of the mesh.

The Couette flow problem is quite easy to solve because the main phenomena at stake are the viscous ones; the compressibility of the fluid being completely irrelevant. It is perfectly known that viscosity tends to stabilise the numerical simulation since it acts as an energy dissipater. The effects of viscosity are highly predictable. Moreover, the convective term, which tends to make the simulation unstable is null.

10.2 The incompressible limit

The solver we developed until now enables us to simulate flows involving quasi-incompressible fluid (with a very high bulk modulus, in comparison with the variation of pressure). We want to determine the behaviour of the simulation in the incompressible limit, i.e. when the bulk modulus K tends to infinite.

To do that, we consider the channel presented in Figure 63. The input flow on the left side of the domain is uniform along the y-axis. At the upper and lower boundaries, we impose that the normal component of the velocity is null but the tangential one is completely free. No boundary condition is applied at the right domain boundary.

Figures 64 and 65 presents the results of the simulations using different values of the bulk modulus K. Figure 64 presents the horizontal component u of the velocity as a function of time at the point represented by ★ in Figure 63. Figure 65 shows the pressure at the same point as a function of time.

Several conclusions can be drawn from these two figures. The analytical stationary solution of the problem are uniform and constant velocity field $u = 1$ (note that the inlet velocity is equal to 1 in each simulation) and pressure $p = 0$. For each value of the bulk modulus we

[19]The same type of analysis has been done, but only numerical noise was observed and it was impossible to interpret it correctly.

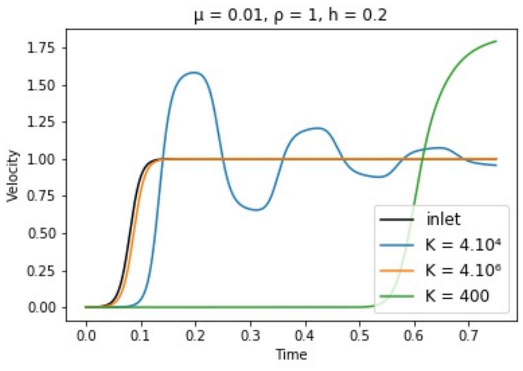

Figure 64: Velocity at the end of the channel as a function of time

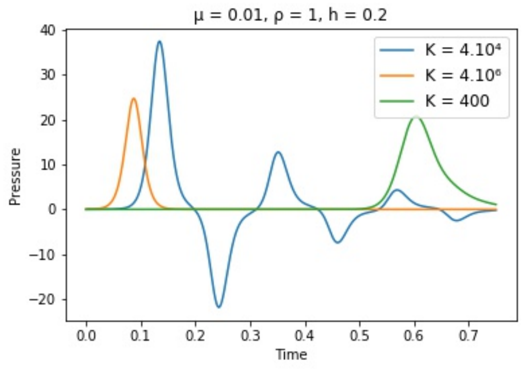

Figure 65: Pressure at the end of the channel as a function of time

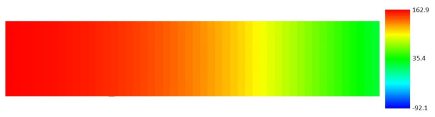

Figure 66: Pressure field in the channel at $t = 0.12$

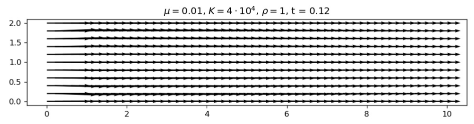

Figure 67: Velocity field in the channel at $t = 0.12$

tested in the three simulations, the velocity and pressure fields converge to the stationary one if we wait enough time. At first sight, we could think that the green line in Figure 64 (which stands for $K = 400$) does not tend to 1 but this is the case if we let the simulation run more time. The oscillations one can observe in the graphs are due to the compressibility of the fluid. In the case of the "incompressible fluid" (orange line with $K = 4 \cdot 10^6$), these oscillations are reduced and the convergence to 1 is much faster, which is consistent with the theory. If the bulk modulus is infinite, the fluid behaves itself as if it was a rigid body; the information propagates instantaneously from the left side to the right one. Let us remember that waves propagate in the fluid with a velocity of $c = \sqrt{K/\rho} = \sqrt{K}$ because, in our case, $\rho = 1$. If we increase the bulk modulus, the wave propagation speed increases. At $t = 0$, one imposes a uniform velocity at the left side of the domain. A fluid particle which is at the right side of the domain will get this information and start moving at time $t = L_x/c$.

If we consider Figure 65, it is very easy to numerically compute the wave propagation speed. For instance, in the simulation with $K = 400$ (green line), the pressure starts rising at approximately $t_0 = 0.5$, which means that the numerical wave propagation speed is $c_{numerical} = L_x/t_0 \approx 10/0.5 = 20$, which is consistent with the analytical wave propagation speed $c_{analytical} = \sqrt{K} = 20$. This process can be repeated with the two other simulations.

Figures 66, 67, 68, 69, 70, 71, 72, 73 describe the pressure and velocity fields at various times of

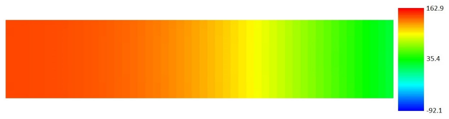

Figure 68: Pressure field in the channel at $t = 0.15$

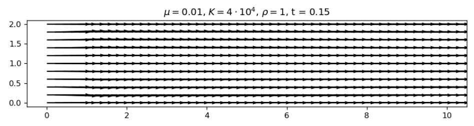

Figure 69: Velocity field in the channel at $t = 0.15$

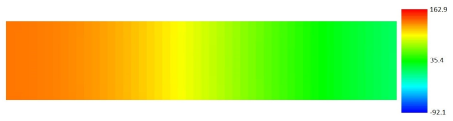

Figure 70: Pressure field in the channel at $t = 0.16$

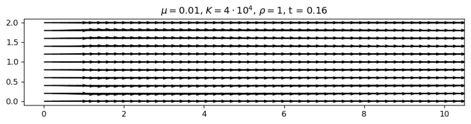

Figure 71: Velocity field in the channel at $t = 0.16$

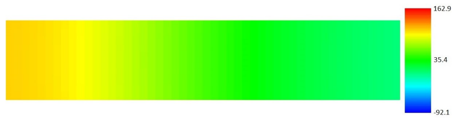

Figure 72: Pressure field in the channel at $t = 0.17$

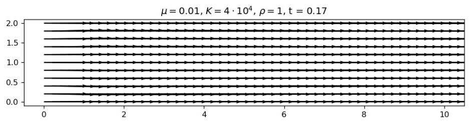

Figure 73: Velocity field in the channel at $t = 0.17$

the simulation in the case of $K = 4 \cdot 10^4$, for which some oscillations can be observed.

Little by little, the velocity we impose at the left side of the domain "propagates" from the left to the right, and the fluid starts moving. It is almost impossible to simulate an infinite channel (which would be the case in the real problem) because we have to mesh a finite domain and, as a consequence, when the front arrives at the end of the domain, it rebounds and goes back to the left. The oscillations of Figures 64 and 65 for $K = 4 \cdot 10^4$ and $K = 400$ are due to this phenomenon. In these two configurations, there are rebounds because the fluid does not comply with the quasi-incompressible approximation. The rate of change of the inlet velocity is too high compared to the bulk modulus of the fluid, the inlet velocity (the information to be propagated) increases too rapidly and the fluid is not able to propagate it correctly. Another way to see it is to compute the maximum pressure in the fluid and compare it with the bulk modulus of the fluid. We should take into account the density variations because it is not constant any more, the series of assumptions we made[20] are not valid any more and that is why we do not have physically-consistent results.

More generally, it is very complicated to impose coherent boundary conditions at the outlets of the finite domain. To dissipate the energy, we tried to implement shock absorbers, whose force is proportional to the derivative of the velocity $F = -\eta \, {}^{\partial \mathbf{u}}/_{\partial t}$ but this makes the simulation unstable because the numerical approximation of the velocity derivative is very imprecise.

10.3 Flow around a cylinder

Now that we have checked the ability of our solver to deal with the viscosity (Couette flow) and compressibility (in the channel of the previous section), it is possible to pass to the following stage, i.e. associate them in more complex flows. Moreover, by putting an obstacle in the flow, it will be possible to show the accuracy of the Shifted Boundary Conditions method and its association with the other employed techniques.

[20]Let us remember that we made the assumption that $\partial\rho << \rho$ in Section 3

Parameters	Value
μ	10
ρ	1
K	10^7
$D_{cylinder}$	20
t_{max}	15
Re	0.1

Figure 74: Parameters of the small Reynolds simulation

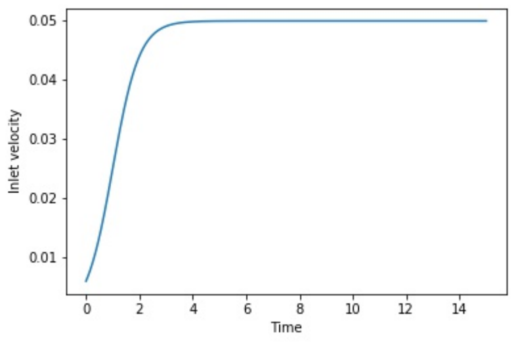

Figure 75: Inlet velocity for the low Re case

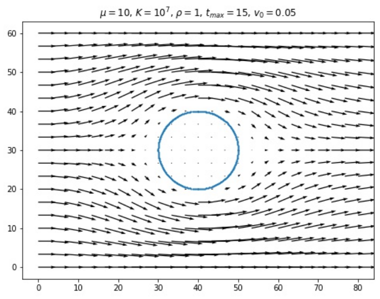

Figure 76: Velocity field around a cylinder for a low value of the Reynolds number

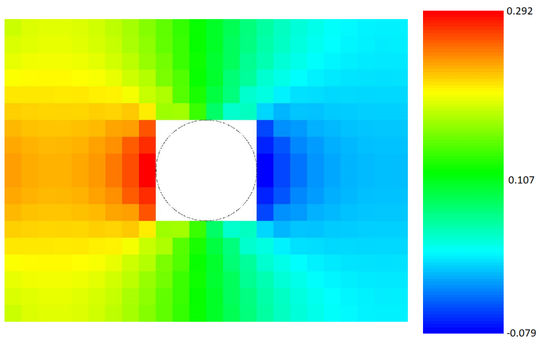

Figure 77: Pressure field around a cylinder for a low value of the Reynolds number

10.3.1 Low Reynolds flow

First, we consider the problem of the flow around a cylinder at a low value of the Reynolds number, for which the viscous effects are dominant and the numerical simulation should be stable.

The parameters used in the simulation are gathered in Table 74. $D_{cylinder}$ refers to the diameter of the cylinder and t_{max} the time that lasts the simulation. Let us just recall that the Reynolds number is computed as

$$Re = \frac{L\,\rho\,v_0}{\mu} \tag{89}$$

where L is the characteristic length of the problem, in our case the diameter of the cylinder.

As for the boundary conditions, the left side of the square domain is the inlet of the problem. The inlet velocity profile is presented in Figure 75. It increases little by little starting from 0 to avoid numerical instability. At the upper and lower sides of the domain are imposed "slip" boundary conditions, i.e. the normal component of the velocity is null.

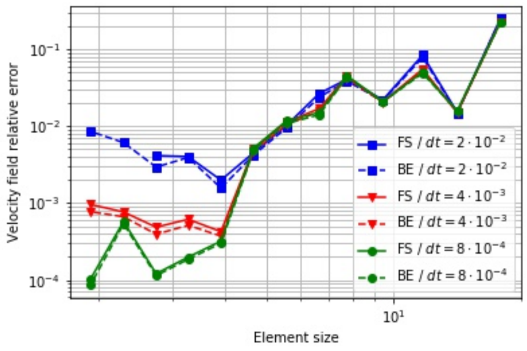

Figure 78: Velocity field relative error for the low Reynolds problem

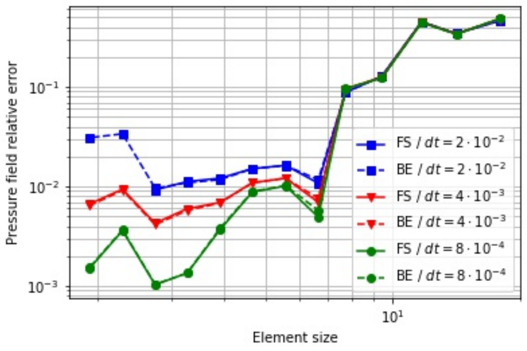

Figure 79: Pressure field relative error for the low Reynolds problem

Figures 76 and 77 present the velocity and pressure fields we obtain at the end of the simulation using a quite coarse mesh (it will be refined later on). First of all, what can be analysed from these figures is that the Shifted Boundary Conditions method is working since the fluid gets around the cylinder, which is what we expected. Moreover, the pressure field is consistent with the common sense since the pressure is higher upstream of the cylinder. There is a pressure gradient between the upstream and downstream sides of the cylinder.

We will use this problem type to validate the accuracy of the solver we previously designed. The FEM theory tells that the solutions we get using finer meshes should converge. To check that, several simulations using different mesh sizes and different time steps are run and the relative errors using the L^2-norm are computed. Figures 78 and 79 present the results for the velocity and pressure fields, respectively. In both graphs, the horizontal axis stands for the element size h and the vertical one for the corresponding relative error. In addition, we studied the convergence of the results for the Fractional Step (FS) and Backward Euler (BE) methods.

Several interesting conclusions can be drawn from these graphs. First of all, as foreseen, the Forward Euler method does not give results for the time steps we considered because, for every mesh size, the solution diverges and tends to infinite. It is mandatory to use either the BE method or the FS one. Secondly, these two numerical schemes present almost identical precision for a given time step and mesh size, which is even truer for the pressure field. Changing from BE to FS does not have a significant impact on the precision of the results we get, which suggests that these two numerical schemes are, in our case, quite equivalent. Lastly, the accuracy of the simulation strongly depends on the time step we use. For instance, if we consider $dt = 0.02$ (see the blue lines in Figures 78 and 79), when refining the mesh, the relative error decreases until $h = 4$. If we use even finer meshes, the relative error increases. The same pattern can be observed with $dt = 4 \cdot 10^{-3}$. All of this is consistent with the theoretical analysis we previously did since we had the restriction

$$dt << \rho h \tag{90}$$

for the FS scheme, which means that one should use smaller time steps for finer meshes. It is exactly the conclusions we get from Figures 78 and 79. When reducing the size of the elements in the mesh, the relative error does not decrease and, sometimes, it even increases. There must

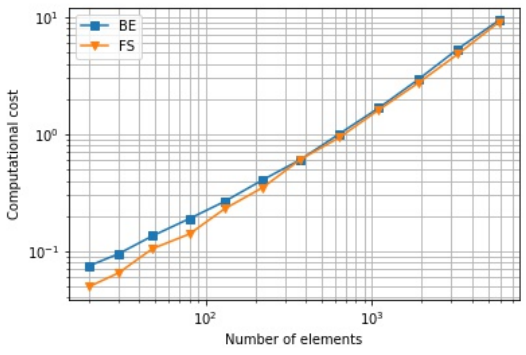

Figure 80: Comparison between the BE and FS computational costs (in seconds)

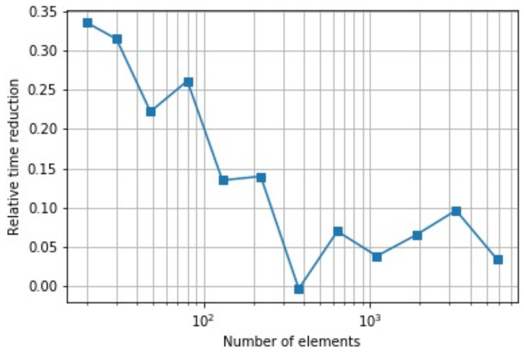

Figure 81: Relative reduction of computational cost between the BE and FS schemes

a be a balance between the time step one should use and the spatial precision he desires.

To show that the BE and FS methods are quite equivalent, we are to analyse the computational cost of the two methods in order to determine if one is more advantageous than the other. From now on, the Forward Euler method is discarded since it is quite hard to achieve convergence with reasonable time steps.

Figures 80 presents the simulation time (in the vertical axis) that the computer needs to do 10 time steps in the problem of the flow around the cylinder we previously described as a function of the number of elements there are in the mesh (horizontal axis). We compare the BE and FS methods. In Figure 81 is presented the relative computational cost reduction as a function of the number of elements in the mesh. This relative computational cost reduction is computed as

$$\tau(N_{element}) = \frac{C_{BE}(N_{element}) - C_{FS}(N_{element})}{C_{BE}(N_{element})} \tag{91}$$

where C_{BE} and C_{FS} refers to the computational cost of the BE and FS methods respectively. It describes how much time we could save using the FS method rather than the BE one.

For small meshes, the FS method is much more interesting because it has a much lower computational cost. The reduction is of approximately 30%. However, when increasing the number of elements in the mesh, this advantage almost disappears and there is no significant difference between the BE and FS methods. This result is a bit disappointing since we expected a lower computational cost for the FS method. I was not able to run the simulation for meshes with more than 6 000 elements because of the limited capacity of my PC. It is barely impossible to check if the order of magnitude of the computational cost is $O(N_{element}^2)$ since we would need to run the simulation with much larger meshes.

10.3.2 Higher Reynolds flow

The objective now is to increase the value of the Reynolds number in order to see if the solver is able to reproduce the behaviour of more complex flows, with vortices for instance.

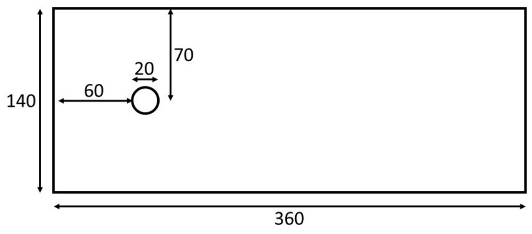

Figure 82: Dimensions of the domain for high values of the Reynolds number

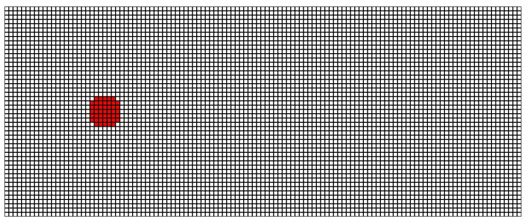

Figure 83: Mesh used for high Reynolds flows

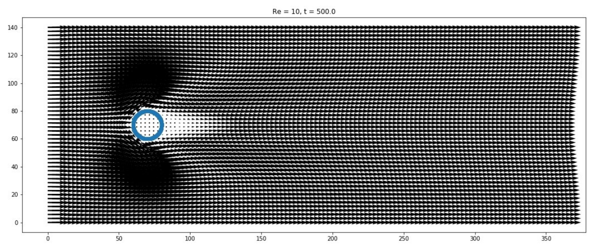

Figure 84: Velocity field at $Re = 10$

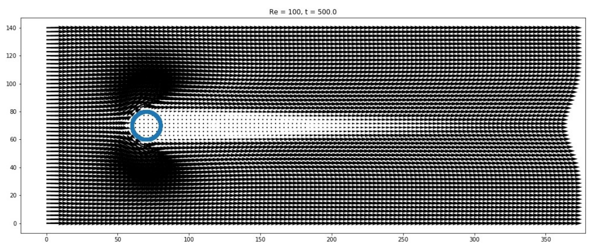

Figure 85: Velocity field at $Re = 100$

Figure 82 presents the dimensions of the domain of the simulation. The mesh we used is depicted in Figure 83; there are approximately 6 000 finite elements. I was not able to perform the simulation with finer meshes because I already reached the top capacity of my computer. I am aware that the mesh is a bit too coarse and it would be useful to refine it in some special areas.

As the Reynolds number increases, the size of the cylinder's trail increases too as it can be observed in Figures 84, 85, 86 and 87. Moreover, at high values of the Reynolds number, the trail starts waving with a constant time period, even if all boundary conditions are symmetrical. Moreover, for $Re = 10$, the inlet is approximately equivalent to the outlet. At high values of the Reynolds number, the obstacle has influence over a much larger area.

Figures 89, 91, 93 and 95 represent the vorticity index of the flows. The vorticity $\boldsymbol{\omega}$ is a first-order tensor defined as the cross product between the $\boldsymbol{\nabla}$ operator and the velocity field $\mathbf{u}$.

$$\boldsymbol{\omega} = \boldsymbol{\nabla} \times \mathbf{u} \tag{92}$$

As we are dealing with 2D problems, this expression can be simplified as

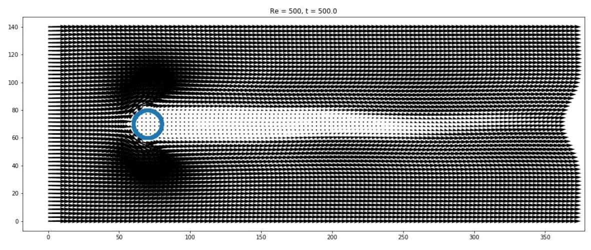

Figure 86: Velocity field at $Re = 500$

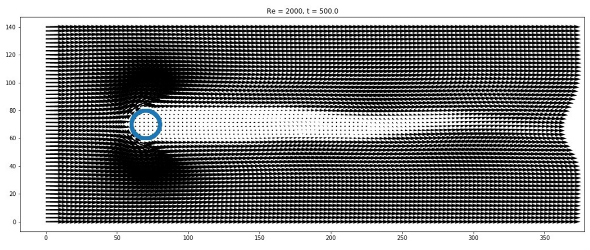

Figure 87: Velocity field at $Re = 2000$

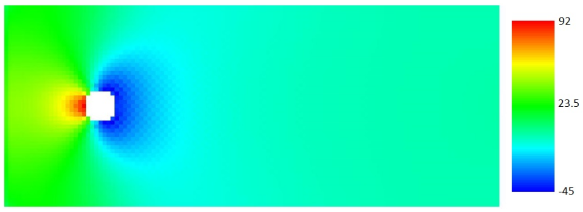

Figure 88: Pressure field at $Re = 10$

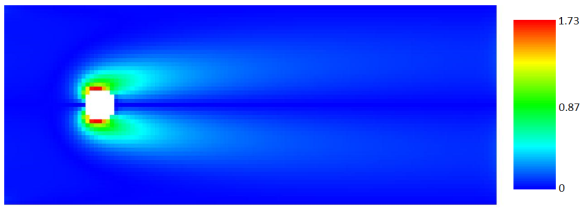

Figure 89: Vorticity field at $Re = 10$

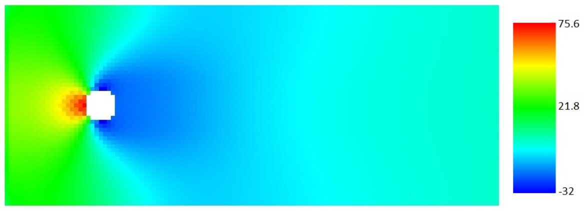

Figure 90: Pressure field at $Re = 100$

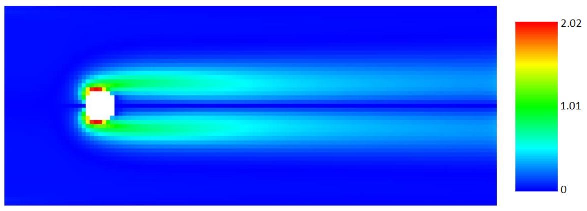

Figure 91: Vorticity field at $Re = 100$

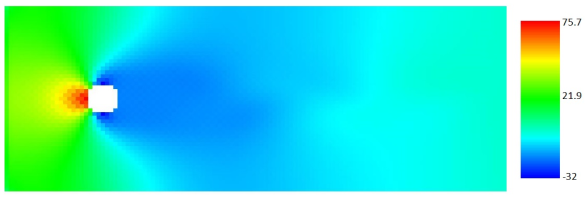

Figure 92: Pressure field at $Re = 500$

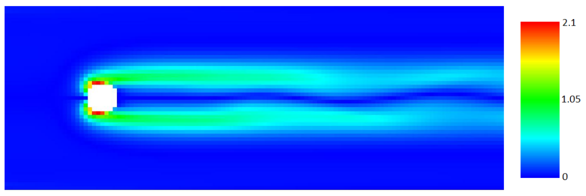

Figure 93: Vorticity field at $Re = 500$

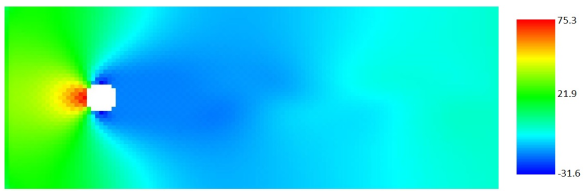

Figure 94: Pressure field at $Re = 2000$

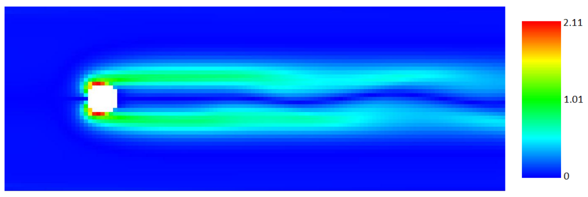

Figure 95: Vorticity field at $Re = 2000$

$$\boldsymbol{\omega} = \left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right)\mathbf{e_z} \tag{93}$$

To compute the vorticity index at one point of the domain, we just need to take the norm of the vorticity vector.

In Figures 89, 91, 93 and 95, we associated a certain colour to each element of the mesh, so that the final results can be easily interpreted. This colour stands for the vorticity index in the centre of the element, the maximum value of the vorticity index being red, and blue for the minimum.

The turbulence downstream of the cylinder increases as the Reynolds number increases. The depression area behind the cylinder is much larger when the Reynolds number of the flow is very high; the drag force applied on the cylinder would be more important. The vorticity is maximum at upper and lower part of the cylinder, which is consistent because they are the most critical zones of the flow. There is a competition between viscosity and convection in the boundary layers of the flow because we impose the velocity to be null on the cylinder.

What I expected before running these different simulations was to evidence the creation of Von Karman vortices at some given values of the Reynolds numbers. These characteristic vortices do not appear, and this is a failure of the simulations. I think the main explication is the mesh of the domain, which is too coarse[21]. To make Von Karman vortices appear, I should have used a much finer mesh. However, I already was at the top capacity of my computer (with around 6 000 elements) and simulations of around six hours. I really was limited by my machine.

10.4 Impact of a city over the wind field

In this last part of the section, we try to study a more realistic case thanks to the solver we developed in order to fully complete the tests and the master thesis.

As we said in Section 1, the solver would be used to solve large engineering problems, such as the modelisation of the wind field within a city. Of course, the object of this Proof of Concept we are doing is not to directly solve this kind of problem because the modelling process would very long and difficult and the computation power of a simple PC is not enough to run the numerical simulation.

However, by greatly simplifying the geometry and reducing it to its minimum, it is possible to get useful preliminary results and have a first idea of what could be at stake in the real engineering problem.

I chose to study the very simple problem of a "city" compound by four buildings and a unique street. The mesh I used in the numerical simulation is presented in Figure 96. In order to have more accurate results, finite elements are smaller around the buildings and larger in the zones which are less affected. The inlet velocity, on the left side of the square domain, is a logarithmic one[22]. In addition, we work at a Reynolds number of $Re = 500$

[21]We should particularly refine the mesh in the boundary layers of the cylinder
[22]This is a classical way to model the velocity profile in the atmosphere

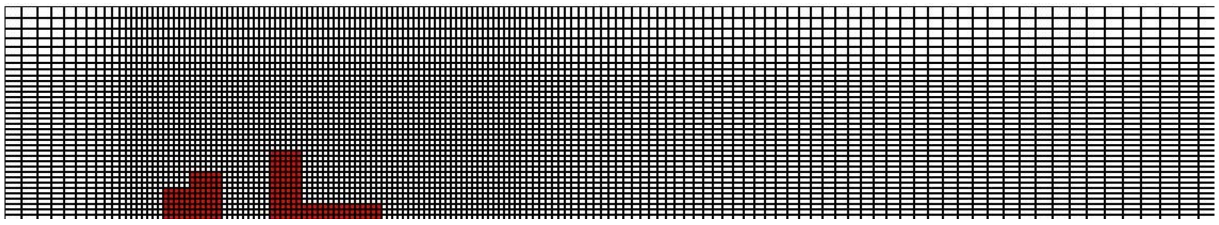
Figure 96: Mesh of the city

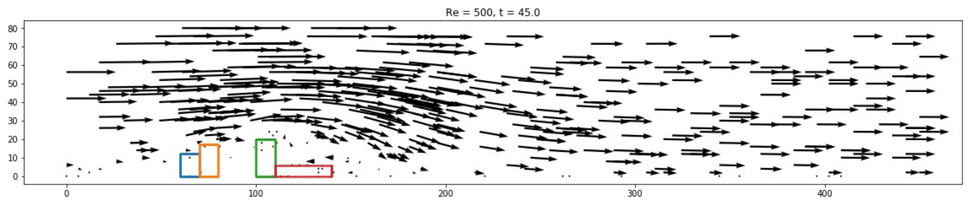

Figure 97: Velocity field at $t = 45$

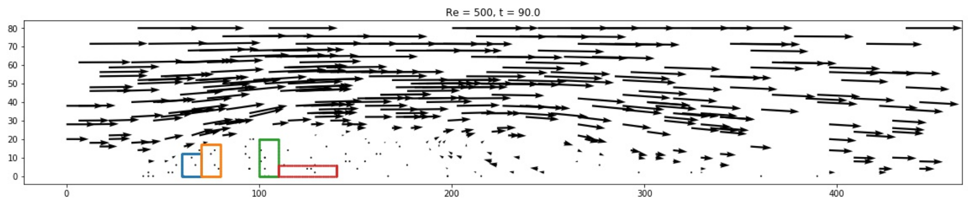

Figure 98: Velocity field at $t = 90$

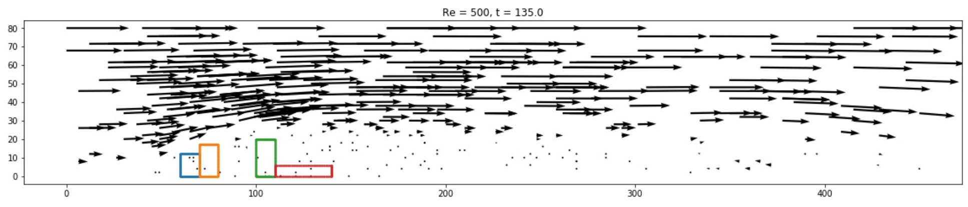

Figure 99: Velocity field at $t = 135$

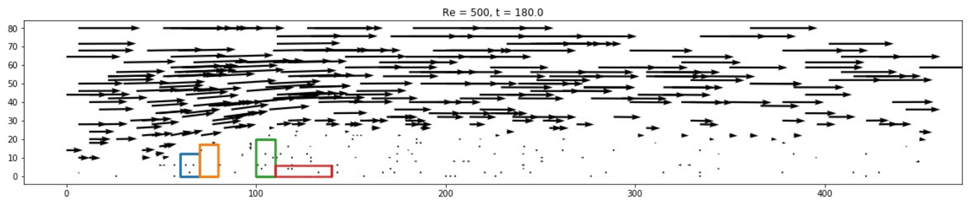

Figure 100: Velocity field at $t = 180$

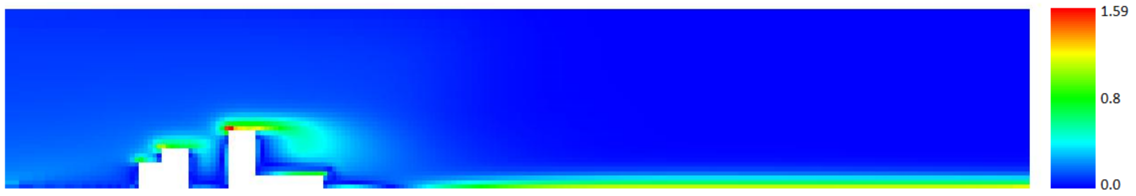

Figure 101: Vorticity field at $t = 45$

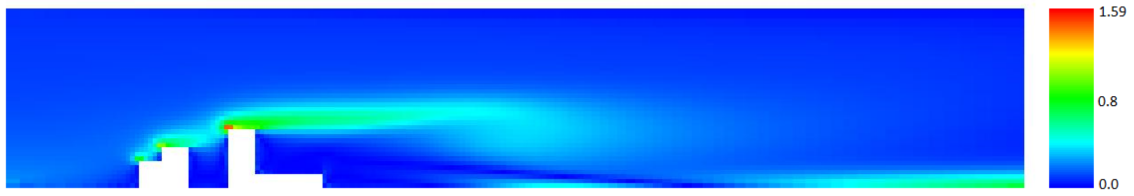

Figure 102: Vorticity field at $t = 90$

First of all is presented the velocity field in Figures 97, 98, 99 and 100. They are useful to confirm that the Shifted Boundary Condition method was well implemented for square elements because the velocity field is consistent with what we could expected before running the simulation: the fluid accelerates to pass over the buildings. Moreover, some phenomena of fluid re-circulation can be observed behind, for instance, the green building with the apparition of a small vortex which evacuates itself after a certain period of time (see Figures 97 and 98).

As we could expect it before running the simulation, we can that a city deeply modifies the wind field. The downstream fluid flow is much more turbulent than in the upstream part as depicted in Figures 101, 102, 103, 104 which present the vorticity field at different time steps. In addition, the city has an impact over a large distance. Of course, the numerical simulation should be bettered (refined mesh and smaller time step) and we have to take lots of care as for the representativity of the solution but this phenomenon is really important and should be taken into account in the resolution of real engineering problems. Let us imagine that an airport is to be built next to the city. The modifications of the wind field caused by the buildings could greatly affect the planes because more turbulence in the atmosphere means harder takes-off and landings. More generally, there are more and more attempts to model the impact of large structures over the wind field in a given region, let us just cite the example of wind turbine wake models.

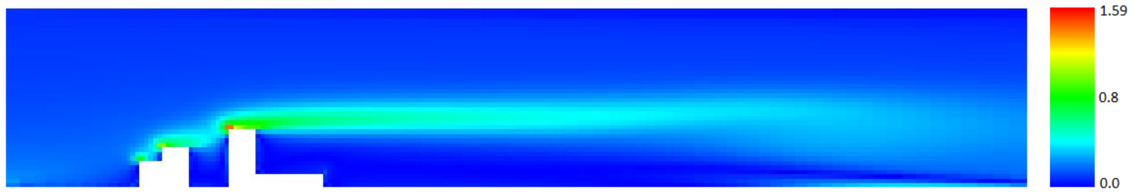

Figure 103: Vorticity field at $t = 135$

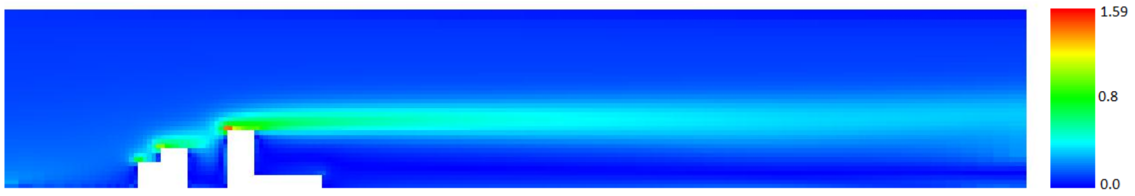

Figure 104: Vorticity field at $t = 180$

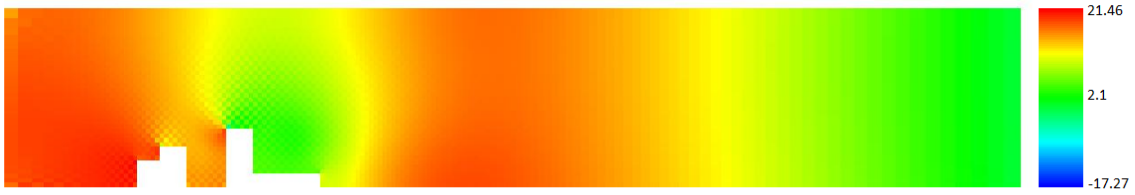

Figure 105: Pressure field at $t = 45$

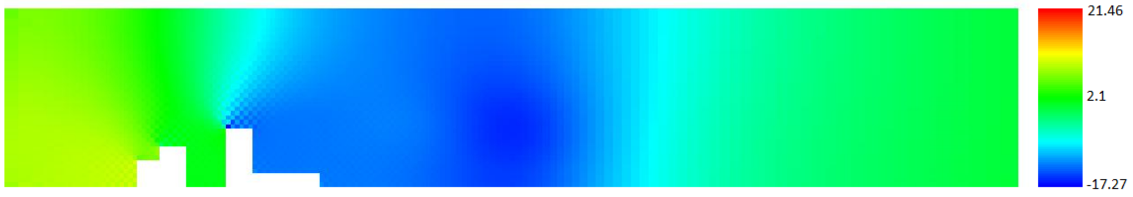

Figure 106: Pressure field at $t = 90$

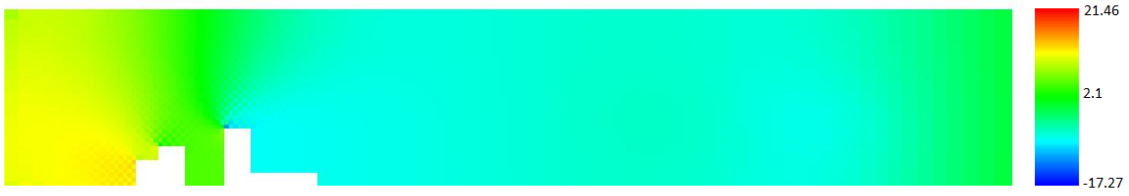

Figure 107: Pressure field at $t = 135$

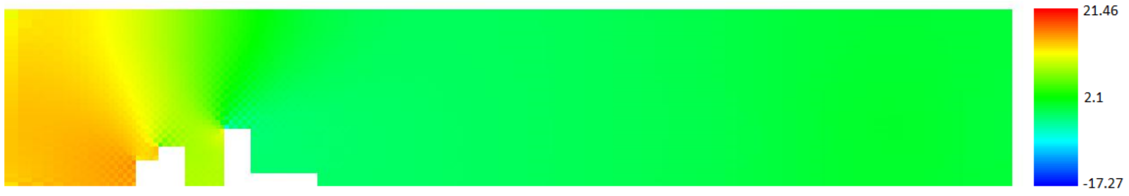

Figure 108: Pressure field at $t = 180$

11 Conclusion and perspectives

I chose this master thesis because it was a great opportunity for me to implement a finite element code from scratch and have more knowledge about what a FEM code does. In my professional life, one day or another, I may need to use one and it is very important for an engineer to know a bit how it internally works. If one uses this kind of software as a black box which displays results, some huge errors can be made. For instance, let us have a look at the results post-process. I chose not to do any post-process and display the raw results (for the velocity as well as the pressure), which means, for example, that the pressure field can appear a bit ugly because there may be sharp changes between some elements. However, this is very important not to forget that the pressure field is non-continuous when using Q1P0 elements; this is one of the fundamental characteristics of this element. In a commercial code, this raw pressure field would have certainly be post-processed to be softer and nobody would care about the continuity of the pressure field. This is not a very big deal but one just has to be aware of it.

Moreover, I was very interested in computational mechanics and this first approach was very rewarding for me. This master thesis was not the easiest one because I had to take into account lots of parameters and implement lots of algorithms but I really liked it, even if I got blocked several times. For instance, at the first shoot, I made a huge error when coding my Python application in the part of the computation of the stiffness matrix because I had not really understood the mechanism and how this matrix and the other ones should be computed. I lost lots of time because of this but, finally, I managed to see where my error was and correct it. This is an example among many other ones. The best way to learn how to code well is to take a computer and start writing lines of code. One can have all the theoretical knowledge he wants, it is nothing without the practical one. I considerably bettered my knowledge of the Python language by coding during more than six months.

I realised that the way one codes can considerably affect the success or failure of a project. The same application is much easier to maintain if it is divided into many functions and/or subroutines because each of them can be separately tested, which makes the code debugging much faster. The testing and debugging of the functions are the most important parts of the job. I should have split much more my Python into functions at the beginning of the project to be more efficient.

Let us get to the overall conclusions of the project. We developed a fluid mechanics solver using a mixed u-p formulation for quasi-incompressible fluid and Q1P0 finite elements. This type of finite elements is widely used in computational mechanics to solve incompressible problems so the main idea was to determine if they could be adapted to CFD and the chances of success were quite high. However, it is well-known that the Q1P0 elements show really poor results when used in unstructured meshes and that is why structured ones had to be used, which raised the issue of boundary conditions since it is much more difficult to correctly impose them with structured them. The solution was to implement the Shifted Boundary Conditions method. Finally, a Back and Forth Error Compensation and Correction (BFECC) method was implemented to deal with the convective term of the Navier-Stokes equation.

All these elements combined constitute the new solver we were to design at the beginning of the project. Then, this solver was tested over different cases and compared with reference solutions in order to validate its robustness.

Globally, the solver is quite robust because it gives coherent results even when using quite coarse meshes, which is not always the case in the world of fluid dynamics. There is no significant difference between the Fractional Step time splitting and the Backward Euler scheme: this is due to the particular structure of the Q1P0 element. However, we were not able to model the Von Karman vortices that should have appeared in the case of the flow around a cylinder at given values of the Reynolds number. This does not question the efficiency and relevance of the solver since I think the main problem in these simulations was the mesh which was too coarse. Normally, the boundary layer (the most critical part of the flow because this is where there is a competition between convection and viscosity) should be meshed with much finer finite elements since correctly modelling this area is a key issue. However, as I only my PC at my disposition, I could not reasonably run simulations with more elements.

The solver we designed is meant to be used to solve "real life" engineering problems. The last case study is a first (and very simple) application of the solver to study the wind field in a city. It could be associated with, for instance, a numerical simulation of the contamination in order to study how it spreads. If one knows the wind field, it can introduce it in an advection/diffusion contamination model to compute the pollutant concentration in every part of the city. Knowing the wind field is just a first step towards the main interesting goals.

References

[BCH98] Jordi Blasco, Ramon Codina, and Antonio Huerta. A fractional-step method for the incompressible navier–stokes equations related to a predictor–multicorrector algorithm. *International Journal for Numerical Methods in Fluids*, 28(10):1391–1419, 1998.

[CCCdS03] Daniel Christ, M Cervera, M Chiumenti, and C Agelet de Saracibar. *A Mixed Finite Element Formulation for Incompressibility Using Linear Displacement and Presure Interpolations*. International Center for Numerical Methods in Engineering, 2003.

[Cod01a] Ramon Codina. Pressure stability in fractional step finite element methods for incompressible flows. *Journal of Computational Physics*, 170(1):112–140, 2001.

[Cod01b] Ramon Codina. A stabilized finite element method for generalized stationary incompressible flows. *Computer Methods in Applied Mechanics and Engineering*, 190(20-21):2681–2706, 2001.

[DL03] Todd F Dupont and Yingjie Liu. Back and forth error compensation and correction methods for removing errors induced by uneven gradients of the level set function. *Journal of Computational Physics*, 190(1):311–324, 2003.

[FP02] Joel H Ferziger and Milovan Perić. *Computational methods for fluid dynamics*, volume 3. Springer, 2002.

[Hug12] Thomas JR Hughes. *The finite element method: linear static and dynamic finite element analysis*. Courier Corporation, 2012.

[MR18] Marcel Montllor Ramoneda. Computational fluid dynamics: Fractional step method and its applications in internal and external flows. B.S. thesis, Universitat Politècnica de Catalunya, 2018.

[MS18] Alex Main and Guglielmo Scovazzi. The shifted boundary method for embedded domain computations. part i: Poisson and stokes problems. *Journal of Computational Physics*, 372:972–995, 2018.

[S$^+$94] Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[VdO05] GAH Van den Oord. Introduction to locking in finite element methods. *Materials Technology Institute, Eindhoven University of Technology*, 2005.

[VJP08] Bart Vossen, HR Javani Joni, and RHJ Peerlings. Volumetric locking in finite elements. *Bachelor Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands*, 2008.

[VRD99] Guido Van Rossum and Fred L Drake. *Python tutorial*. Open Documents Library, 1999.

12 Appendix

```
# -*- coding: utf-8 -*-
"""
Created on Thu Nov  8 18:35:50 2018

@author: clement.lemardele
"""


import numpy as np
import math
import os
from scipy.sparse import coo_matrix
from scipy.sparse import dia_matrix


import Element_Bbar_v2 as Bbar
import Solver_Navier_Stokes_v3 as solver
import Pre_process_v6 as pre_process
import Post_process_v7 as post_process


############################################################################
############################################################################
############################################################################

# Implementation of several classes to model obstacles in the fluid domain
# Each class has its own attributes depending on the features of the shape

# get_Level_Set(self, x) = return the value of the level set function
# associated to the obstacle at point x. The expression of the level set
# function depends on the shape features

# get_Closest_Point(self, x) = given a point with coordinates x, return the
# coordinates of the closest point which is on the border of the obstacle

# get_Shape_Coordinates(self) = return a list of points which are on the
# border of the obstacle ==> used for pre and post-process

############################################################################
############################################################################

class Circle_BC:

    # Defines a class to implement a round obstacle in the fluid domain
    # Attributes: centre and radius

    def __init__(self, centre_coordinates, radius):

        self.centre_coordinates = np.array(centre_coordinates)
        self.radius = radius

    ######################################################################

    def get_Level_Set(self, x):

        centre = self.centre_coordinates
        radius = self.radius

        return np.linalg.norm(x - centre, 2) - radius

    ##########################################################################

    def get_Closest_Point(self, x):

        x_centre = self.centre_coordinates
        radius = self.radius

        return x_centre + radius * (x-x_centre) / np.linalg.norm(x-x_centre, 2)

    ##########################################################################

    def get_Shape_Coordinates(self):

        radius = self.radius
        center = self.centre_coordinates
        x_C = center[0]
        y_C = center[1]

        x_list = np.linspace(x_C - radius, x_C + radius, 100)
        y_list = np.linspace(y_C - radius, y_C + radius, 100)
        shape_coordinates = np.zeros((400, 2), dtype = float)

        count = 0

        for x in x_list:

            y0 = y_C + math.sqrt(radius**2 - (x - x_C)**2)
            y1 = y_C - math.sqrt(radius**2 - (x - x_C)**2)

            shape_coordinates[count, 0] = x
            shape_coordinates[count, 1] = y0
            count += 1

            shape_coordinates[count, 0] = x
            shape_coordinates[count, 1] = y1
            count += 1

        for y in y_list:

            x0 = x_C + math.sqrt(radius**2 - (y - y_C)**2)
            x1 = x_C - math.sqrt(radius**2 - (y - y_C)**2)

            shape_coordinates[count, 0] = x0
            shape_coordinates[count, 1] = y
            count += 1

            shape_coordinates[count, 0] = x1
            shape_coordinates[count, 1] = y
            count += 1

        return shape_coordinates

    ##########################################################################

    def get_Normal(self, x):

        normal = np.zeros((2,1), dtype = float)

        normal[0,0] = self.centre_coordinates[0]
        normal[1,0] = self.centre_coordinates[1]

        normal = x - normal

        return 1 / np.linalg.norm(normal) * normal


################################################################################
################################################################################


class Rectangle_BC:

    # Defines a class to implement a square obstacle in the fluid domain
    # Attributes: x_left, x-coordinate of the rectangle left side
    #             x_right, x-coordinate of the rectangle right side
    #             y_bottom, y-coordinate of the rectangle bottom side
    #             y_up, y-coordinate of the rectangle up side

    def __init__(self, x_left, x_right, y_bottom, y_up):
        self.x_left = x_left
        self.x_right = x_right
        self.y_bottom = y_bottom
        self.y_up = y_up

    ############################################################################

    def get_Level_Set(self, x):

        x_point = x[0]
        y_point = x[1]

        x_left = self.x_left
        x_right = self.x_right
        y_bottom = self.y_bottom
        y_up = self.y_up

        level_set = 1

        if x_left < x_point and x_point < x_right\
        and y_bottom < y_point and y_point < y_up: level_set = -1

        if x_point == x_left and y_bottom <= y_point\
        and y_point <= y_up: level_set = 0

        if x_point == x_right and y_bottom <= y_point\
        and y_point <= y_up: level_set = 0

        if y_point == y_bottom and x_left <= x_point\
        and x_point <= x_right: level_set = 0

        if y_point == y_up and x_left <= x_point\
        and x_point <= x_right: level_set = 0

        return level_set

    ############################################################################

    def get_Closest_Point(self, x):

        x_point = x[0]
        y_point = x[1]

        x_left = self.x_left
        x_right = self.x_right
        y_bottom = self.y_bottom
        y_up = self.y_up

        x_closest = [x_point, y_point]

        if x_point <= x_left: x_closest[0] = x_left
        if x_point >= x_right: x_closest[0] = x_right

        if y_point <= y_bottom: x_closest[1] = y_bottom
        if y_point >= y_up: x_closest[1] = y_up

        return x_closest

    ##########################################################################

    def get_Shape_Coordinates(self):

        x_left = self.x_left
        x_right = self.x_right
        y_bottom = self.y_bottom
        y_up = self.y_up

        point_coordinates = np.zeros((400, 2), dtype = float)

        x_list = np.linspace(x_left, x_right, 100)
        y_list = np.linspace(y_bottom, y_up, 100)

        point_coordinates[0:100,0] = x_list
        point_coordinates[0:100,1] = [y_bottom for i in range(100)]

        point_coordinates[100:200,0] = x_list
        point_coordinates[100:200,1] = [y_up for i in range(100)]

        point_coordinates[200:300,1] = y_list
        point_coordinates[200:300,0] = [x_left for i in range(100)]

        point_coordinates[300:400,1] = y_list
        point_coordinates[300:400,0] = [x_right for i in range(100)]

        return point_coordinates


##############################################################################
##############################################################################


class Domain_Boundary:

    # Defines a class to implement the square domain boundary
    # Attributes: Lx, size of the domain in x-direction
    #             Ly, size of the domain in y-direction
    #             side: 'left', 'bottom', 'right' or 'up'
    def __init__(self, side, Lx, Ly):

        self.side = side
        self.Lx = Lx
        self.Ly = Ly

    ##########################################################################

    def get_Level_Set(self, x):

        if self.side == 'left': return x[0]
        if self.side == 'right': return self.Lx - x[0]
        if self.side == 'bottom': return x[1]
        if self.side == 'up': return self.Ly - x[1]

    ##########################################################################

    def get_Closest_Point(self, x):

        if self.side == "left": return [0, x[1]]
        if self.side == "right": return [self.Lx, x[1]]
        if self.side == "bottom": return [x[0], 0]
        if self.side == "up": return [x[0], self.Ly]

    ##########################################################################

    def get_Normal(self, x):

        if self.side == "left": return np.array([[-1.0], [0.0]])
        if self.side == "right": return np.array([[1.0], [0.0]])
        if self.side == "bottom": return np.array([[0.0], [-1.0]])
        if self.side == "up": return np.array([[0.0], [1.0]])



##############################################################################
##############################################################################
##############################################################################

# Input of simulation parameters (by user)

##############################################################################
# Inlet velocity

# Max inlet velocity
v_0 = 10

# Temporal variation of the inlet velocity
def get_u_inlet(time, v_0):

    # Time sigmoid inlet velocity

    t_0_1 = 2.5
    t_a_1 = 15

    sig_1 = ( 1 + np.exp(-(time - t_a_1)/t_0_1) )**(-1)

    return v_0 * sig_1

```

```
###########################################################################
# Fluid properties

mu = 0.4 # Viscosity
rho = 1 # Density
bulk = 1e7 # Compressibility / bulk modulus

###########################################################################
# Geometric properties

Lx = 450 # Horizontal size of the square domain
Ly = 80 # Vertical size of the square domain

# Mesh properties
nbElemHoriz = 145 # Number of finite elements in x-direction
nbElemVert = 35 # Number of finite elements in y-direction

x_concentrated = [[45, 175], 2]
y_concentrated = [[0, 50], 2]

###########################################################################
# Boundary conditions

# List of obstacles which are in the fluid domain
shapes = [Rectangle_BC(60, 70, 0, 12),
          Rectangle_BC(70, 80, 0, 17),
          Rectangle_BC(100, 110, 0, 20),
          Rectangle_BC(110, 140, 0, 6)]

# Possible boundary conditions: slip, no_slip
# Possible inlet: uniform_inlet, parabolic_inlet, logarithmic_inlet,
#                 tangential_inlet

boundaries = [[Domain_Boundary("left", Lx, Ly), "logarithmic_inlet"],
              [Domain_Boundary("bottom", Lx, Ly), "no_slip"],
              [Domain_Boundary("up", Lx, Ly), "slip"],
              [Rectangle_BC(60, 70, 0, 12), "no_slip"],
              [Rectangle_BC(70, 80, 0, 17), "no_slip"],
              [Rectangle_BC(100, 110, 0, 25), "no_slip"],
              [Rectangle_BC(110, 140, 0, 6), "no_slip"]]

###########################################################################
# Initial condition

nbPoints = (nbElemHoriz + 1) * (nbElemVert + 1)
nbElem = nbElemHoriz * nbElemVert

# Define initial conditions for the velocity and pressure fields
U0 = np.zeros((2*nbPoints, 1), dtype = float)
P0 = np.zeros((nbElem, 1), dtype = float)


###########################################################################
# Time integration parameters

# Numerical scheme to be used
integration_method = 'FS' # FE = Forward Euler
                          # BE = Backward Euler
                          # FS = Fractional Step
# Maximum time
tmax = 450

# Time step to be used
dt = 0.01

###########################################################################
# Post-process parameter

# Number of pictures to be post-processed
n_sample = 10

# Computation of the Reynolds number
Re = 20 * rho * v_0 / mu



###########################################################################
###########################################################################
###########################################################################

# Pre process of the simulation
print('Pre-process')

# Characteristic size of elements (in horizontal and vertical directions)
h = math.sqrt(Lx * Ly / (nbElemHoriz * nbElemVert))

# Time discretisation
nbSteps = int(tmax/dt) # Number of time steps of the simulation
time = pre_process.get_time_discretisation(nbSteps, dt)

u_inlet = get_u_inlet(time, v_0) # contains the values of the inlet velocity
                                 # at each time step

# Generation of the structured quadrilateral mesh over the whole (Lx, Ly)
# domain without taking into account the obstacles
connectivity, coordinates, neighbours = \
pre_process.get_structured_mesh(Lx, Ly, nbElemHoriz, nbElemVert,\
                                x_concentrated, y_concentrated)


###########################################################################

# Looking for inner nodes, boundary nodes and inner elements (see Shifted
# Boundary Condition)

# Initialisation of the lists
inner_nodes = []
boundary_nodes = []
inner_elements = []
nodes_boundaries = []

# Loop over all the boundary conditions contained in array "boundaries"
for boundary in boundaries:

    shape = boundary[0]
    BC_type = boundary[1]

    shape_inner_elements, shape_inner_nodes, shape_boundary_nodes = \
    pre_process.get_inner_features(shape, connectivity, coordinates)

    # We add the inner nodes, boundary nodes and inner elements associated to
    # each obstacle
    inner_nodes += shape_inner_nodes
    boundary_nodes += shape_boundary_nodes
    inner_elements += shape_inner_elements

    # To apply the Shifted Boundary Conditions method, we have to know which
    # obstacle is associated to each boundary node
    for node in shape_boundary_nodes: nodes_boundaries.append([node, shape,\
                                                              BC_type])

# Just to ensure that nodes appear only once in each list
inner_nodes = list(set(inner_nodes))
boundary_nodes = list(set(boundary_nodes))
inner_elements = list(set(inner_elements))

# We define the set of nodes which are not inner ones
active_nodes = list(range(nbPoints))
for node in inner_nodes: active_nodes.remove(node)

active_elements = list(range(nbElem))
for element_number in inner_elements: active_elements.remove(element_number)

# Generate a picture of the mesh and save it in the directory
pre_process.display_mesh(inner_elements, boundary_nodes, connectivity,\
                coordinates, shapes, nbElemVert, nbElemHoriz)



###############################################################################
###############################################################################
###############################################################################

# Assembly of the matrices
print('Beginning mesh assembly')


# Initialisation
Kuu = np.zeros([2*nbPoints, 2*nbPoints], dtype = float)
Muu = np.zeros([2*nbPoints, 2*nbPoints], dtype = float)
D = np.zeros([2*nbPoints, nbElem], dtype = float)
Mpp = np.zeros([1, nbElem], dtype = float)
Mpp_inv = np.zeros([1, nbElem], dtype = float)

M_PI = np.zeros([1, nbPoints], dtype = float)
N_DNDx = np.zeros([nbPoints, nbPoints], dtype = float)
N_DNDy = np.zeros([nbPoints, nbPoints], dtype = float)


# Loop over all the active elements of the mesh
# Inner elements which have been eliminated during the pre-process part are not
# taken into account in the assembly process, as if they did not exist any
# more

print('\tLooping over active elements')

for num_element in active_elements:

    # Get the four nodes of the given element
```

```
element = connectivity[num_element,:]

# Get the global numbers associated with each node of the element
numNode0 = element[0]
numNode1 = element[1]
numNode2 = element[2]
numNode3 = element[3]

# Get the coordinates of the four nodes
X0 = coordinates[numNode0]
X1 = coordinates[numNode1]
X2 = coordinates[numNode2]
X3 = coordinates[numNode3]

# Get local matrices (see Q1P0 finite element)
[MuuElem, KuuElem, DElem, N_DNDx_elem, N_DNDy_elem] =\
Bbar.get_Local_Matrices(X0, X1, X2, X3, rho, mu)

surface_element = (X1[0] - X0[0]) * (X3[1] - X0[1])

###########################################################################
# Global matrices assembly

for i in range(4):

    for j in range(4):

        numElemI = element[i]
        numElemJ = element[j]

        # Assembly of the stiffness matrix
        Knode = KuuElem[2*i : 2*i+2, 2*j : 2*j+2]
        Kuu[2*numElemI : 2*numElemI+2, 2*numElemJ : 2*numElemJ+2] += Knode


        # Assembly of the velocity mass matrix
        Mnode = MuuElem[2*i : 2*i+2, 2*j : 2*j+2]
        Muu[2*numElemI:2*numElemI+2, 2*numElemJ:2*numElemJ+2] += Mnode


        N_DNDx[numElemI, numElemJ] += N_DNDx_elem[i,j]
        N_DNDy[numElemI, numElemJ] += N_DNDy_elem[i,j]


# Assembly of the pressure mass matrix
Mpp[0, num_element] = surface_element / bulk
Mpp_inv[0, num_element] = bulk / surface_element


# Assembly of the gradient/divergence matrix
for i in range(4):

    numNodeI = element[i]

    vec = DElem[2*i:2*i+2]

    D[2*numNodeI : 2*numNodeI+2, num_element : num_element + 1] = vec

    M_PI[0, numNodeI] += surface_element / 4.0

```

```

##########################################################################
# Lump the velocity mass matrix

print('\tLumping matrices')

# Initialisation
Muu_lumped = np.zeros((1, 2 * nbPoints), dtype = float)
Muu_lumped_inv = np.zeros((1, 2 * nbPoints), dtype = float)

# Lumping
for i in range(2 * nbPoints):

    Muu_lumped[0, i] = sum(Muu[i,:])

    if Muu_lumped[0, i] != 0: Muu_lumped_inv[0, i] = 1 / Muu_lumped[0, i]



##########################################################################
# Compute the inverse of M_PI matrix

print('\tComputing inverse matrices')

# Initialisation
M_PI_inv = np.zeros((1, nbPoints), dtype = float)

for i in range(nbPoints):

    if M_PI[0, i] != 0: M_PI_inv[0, i] = 1 / M_PI[0, i]



##########################################################################

# Definition as sparse matrices

print('\tCreating sparse matrices')

# coo_matrices
Kuu = coo_matrix(Kuu, dtype = float)
D = coo_matrix(D, dtype = float)
N_DNDx = coo_matrix(N_DNDx, dtype = float)
N_DNDy = coo_matrix(N_DNDy, dtype = float)

# Diagonal matrices
Mpp = dia_matrix((Mpp, 0), (nbElem, nbElem), dtype = float)
Mpp_inv = dia_matrix((Mpp_inv, 0), (nbElem, nbElem), dtype = float)
Muu_lumped = dia_matrix((Muu_lumped, 0), (2*nbPoints, 2*nbPoints),\
                        dtype = float)
Muu_lumped_inv = dia_matrix((Muu_lumped_inv, 0), (2*nbPoints, 2*nbPoints),\
                        dtype = float)
M_PI = dia_matrix((M_PI, 0), (nbPoints, nbPoints), dtype = float)
M_PI_inv = dia_matrix((M_PI_inv, 0), (nbPoints, nbPoints), dtype = float)



##########################################################################
##########################################################################
##########################################################################
```

```python
# Perform time integration depending on the numerical scheme the user has
# defined

solU, solP = \
solver.perform_time_integration(Muu_lumped, Muu_lumped_inv, Mpp, Mpp_inv, D, Kuu,\
                                dt, nbSteps, integration_method, U0, P0, M_PI_inv,\
                                N_DNDx, N_DNDy, nodes_boundaries, coordinates,\
                                u_inlet, connectivity, neighbours)



##############################################################################
##############################################################################
##############################################################################
# Saving the most important variables

print('Beginning post-process')

print('\tSaving matrices')

np.save('solU.npy', solU)
np.save('solP.npy', solP)
np.save('connectivity.npy', connectivity)
np.save('coordinates.npy', coordinates)
np.save('time.npy', time)
np.save('inner_elements.npy', inner_elements)

# Visualization of solution

increment = int(nbSteps / n_sample)

# Index of time steps for which we are going to post-process the solution
sample = [nb for nb in range(nbSteps) if not nb%increment]


sol_vorticity = np.zeros((nbElem, len(sample)), dtype = float)
count = 0

print('\tComputing vorticity field')

for i in sample:

    vorticity_field = post_process.get_Vorticity_Field(solU[:,i],\
                                         connectivity, coordinates)

    sol_vorticity[:,count] = vorticity_field

    count +=1

##############################################################################
# To build the pressure scale, we need to determine the minimum and maximum
# pressure in the whole simulation
# Same idea for the vorticity

max_pressure_list = [0 for i in range(len(sample))]
min_pressure_list = [0 for i in range(len(sample))]
max_vorticity_list = [0 for i in range(len(sample))]
min_vorticity_list = [0 for i in range(len(sample))]
count = 0
```

```python
print('\tDefining colour scale')

for i in sample:

    max_pressure_list[count] = max(solP[:,i])
    min_pressure_list[count] = min(solP[:,i])

    max_vorticity_list[count] = max(sol_vorticity[:,count])
    min_vorticity_list[count] = min(sol_vorticity[:,count])

    count += 1

min_pressure = min(min_pressure_list)
max_pressure = max(max_pressure_list)

min_vorticity = min(min_vorticity_list)
max_vorticity = max(max_vorticity_list)

print('\t\tMax pressure: ' + str(max_pressure))
print('\t\tMin pressure: ' + str(min_pressure))

print('\t\tMax vorticity: ' + str(max_vorticity))
print('\t\tMin vorticity: ' + str(min_vorticity))

print('\tCreating results directories')

# Creation of directories
os.mkdir("Pressure")
os.mkdir("Velocity")
os.mkdir("Vorticity")

count = 0

print('\tStoring velocity, pressure and vorticity')

for i in sample:

    post_process.velocity_Quiver(solU[:,i], coordinates, dt,\
                                 tmax, i, shapes, Re, Lx, Ly)

    post_process.display_pressure(inner_elements, connectivity, coordinates,\
                                  min_pressure, max_pressure, solP[:,i], i,\
                                  shapes, nbElemVert, nbElemHoriz)

    post_process.display_Vorticity(inner_elements, connectivity, coordinates,\
                                   min_vorticity, max_vorticity,\
                                   sol_vorticity[:,count], count, shapes,\
                                   nbElemVert, nbElemHoriz)

    count += 1
```

Scripts/Navier_Stokes_v3.py

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Apr 25 12:41:36 2019

@author: clement.lemardele
"""

```

```
# This module contains all the functions needed for the pre-process of the
# Navier problems using Q1P0 elements and a structured mesh

import numpy as np
from PIL import Image


def create_image(Lx, Ly):

    # Return an empty matrix of given size which wil be converted into a
    # picture later on. The matrix contains RGB tuples
    # Lx, Ly = horizontal and vertical size of the domain one wants to repre-
    # sent with the picture

    # The largest dimension of the domain picture will include nb_pixels pixels
    nb_pixels = 1200 # pixels

    if Lx == max(Lx, Ly): # the largest dimension is Lx

        # The image will have nb_pixels in the horizontal direction
        image_array = 255 * np.ones( (int(Ly/Lx*nb_pixels), nb_pixels, 3),\
                                    dtype = int )

    else: # the largest dimension is Ly

        # The image will have nb_pixels in the vertical direction
        image_array = 255 * np.ones( (nb_pixels, int(Lx/Ly*nb_pixels), 3),\
                                    dtype = int )

    return image_array


###############################################################################
###############################################################################

def get_mesh_element_picture(nx, ny):

    # Return a RGB matrix representing a mesh element
    # nx = number of pixels in the horizontal direction
    # ny = number of pixels in the vertical directions
    # White colour = [255, 255, 255] (RGB)
    # Black colour = [0, 0, 0] (RGB)

    # Creation of a "white" matrix with given dimensions nx and ny
    image_array = 255 * np.ones((ny, nx, 3), dtype = int)

    # The image will include a 1 pixel black border
    # Left black border
    image_array[:, 0:1] = np.zeros( (ny, 1, 3), dtype = int )
    # Right black border
    image_array[:, nx - 1 : nx + 1] = np.zeros( (ny, 1, 3), dtype = int )
    # Up black border
    image_array[0:1, :] = np.zeros( (1, nx, 3), dtype = int )
    # Bottom black border
    image_array[ny - 1 : ny + 1, :] = np.zeros( (1, nx, 3), dtype = int )

    return image_array


###############################################################################
```

```
##############################################################################


def convert_to_list(color_matrix):

    # Return a list of the RGB pixels which are contained in color_matrix
    # color_matrix = matrix of RGB pixels

    size = color_matrix.shape

    matrix_list = []

    # Loop over the entire matrix color_matrix
    for i in range(size[0]):

        for j in range(size[1]):

            # For each entry of color_matrix, we convert it into a tuple and
            # add to the list of tuple
            new_tuple = (int(color_matrix[i,j,0]), \
                         int(color_matrix[i,j,1]),\
                         int(color_matrix[i,j,2]))

            matrix_list.append(new_tuple)

    return matrix_list


##############################################################################
##############################################################################

def get_interior_element_picture(nx, ny):

    # Return the picture of an interior element, i.e. which has been
    # eliminated in the Shifted Boundary Condition method
    # nx = number of horizontal pixels
    # ny = number of vertical pixels
    # Red colour = [255, 0, 0]

    # Creating a new white image
    image_array = 255 * np.ones((ny, nx, 3), dtype = int)

    for i in range(ny):

        for j in range(nx):

            # We set the pixel to red
            image_array[i,j] = [255, 0, 0]

    # Creation of a 1 px black border
    image_array[:, 0:1] = np.zeros( (ny, 1, 3), dtype = int )
    image_array[:, nx - 1 : nx + 1] = np.zeros( (ny, 1, 3), dtype = int )
    image_array[0:1, :] = np.zeros( (1, nx, 3), dtype = int )
    image_array[ny - 1 : ny + 1, :] = np.zeros( (1, nx, 3), dtype = int )

    return image_array



def display_mesh(inner_elements, boundary_nodes, connectivity, coordinates,\
            shapes, nbElemVert, nbElemHoriz):

    # Create and save an image of a structured mesh of quadrilateral elements

    # inner_elements = list of elements which have been eliminated during
    # the Shifted Boundary conditions process. The inner elements will be
    # displayed in red
    # boundary_nodes = list of the mesh boundary nodes
    # connectivity = mesh connectivity matrix
    # coordinates = mesh coordinates matrix
    # nbElemHoriz = number of elements in the horizontal direction
    # nbElemVert = number of elements in the vertical direction
    # result_path = path where the mesh picture will be saved
    # shapes = list of shapes to be superimposed

    size_coordinates = coordinates.shape

    nbPoints = size_coordinates[0]

    Lx = coordinates[nbPoints - 1, 0]
    Ly = coordinates[nbPoints - 1, 1]

    image_array = create_image(Lx, Ly)

    size_image = image_array.shape

    # We take a margin so that to avoid errors
    # horizontal_pixel = number of pixels to represent the distance Lx
    # vertcial_pixel = number of pixels to represent the distance Ly
    horizontal_pixel = size_image[1]
    vertical_pixel = size_image[0]

    # In the mesh, there are 2 types of elements, each one represented by a
    # given image

    count = 0
    pixel_y = vertical_pixel

    for j in range(nbElemVert):

        pixel_x = 0

        for i in range(nbElemHoriz):

            element = connectivity[count,:]

            node_0 = element[0]
            node_2 = element[2]

            x0 = coordinates[node_0,:]
            x2 = coordinates[node_2,:]

            x_A = x0[0]
            y_A = x0[1]

            x_B = x2[0]
            y_B = x2[1]

            # Determine the dimensions of the element
            length_x = int((x_B - x_A) / Lx * horizontal_pixel)
            length_y = int((y_B − y_A) / Ly * vertical_pixel)

            # Once we determine the position of the element, we colocate it in
            # the RGB matrix

            if count in inner_elements:

                interior_element_picture =\
                get_interior_element_picture(length_x, length_y)

                image_array[pixel_y − length_y : pixel_y,\
                            pixel_x : pixel_x + length_x] \
                            = interior_element_picture

            else:

                mesh_element_picture =\
                get_mesh_element_picture(length_x, length_y)

                image_array[pixel_y − length_y : pixel_y,\
                            pixel_x : pixel_x + length_x] \
                            = mesh_element_picture

            pixel_x += length_x

            count += 1

        pixel_y = pixel_y − length_y


    # To use the module Image, we need to convert image_array, which is a
    # matrix, into a list of tuples
    image_tuple = convert_to_list(image_array)

    # Creation a new picture using the module Image
    img = Image.new("RGB", (size_image[1], size_image[0]))

    img.putdata(image_tuple)

    # We save in .jpg format the picture of the mesh in the indicated directory
    img.save("Mesh.jpg")


################################################################################
################################################################################

def get_time_discretisation(nbSteps, dt):

    # Return a vector containing the list of all the discretised times

    return np.linspace(0, dt*nbSteps, nbSteps + 1)


################################################################################
################################################################################

def get_Geometrical_Suite(x0, x1, alpha, num):

    # Return the values of a geometrical suite between x0 and x1 and reason
    # alpha
    x = x0
    suite = [float(x0)]
    length = x1 - x0

    # Determine initial interval length
    h = length * (1-alpha) / (1 - alpha**(num))

    for i in range(0, num-1):

        suite.append(x + h * alpha**i)
        x += h * alpha**i

    suite.append(float(x1))

    return suite


##############################################################################
##############################################################################

def get_X_list(x_concentrated, nbElemHoriz, Lx):

    # Return the list of the mesh X-coordinates
    # x_concentrated = area of the mesh where finite elements are concentrated
    #                  to have more precision
    # nbElemHoriz = number of finite elements in horizontal direction
    # Lx = size of the domain in horizontal direction

    alpha = 1.1

    X = []

    x0 = x_concentrated[0][0]
    x1 = x_concentrated[0][1]
    hx_c = x_concentrated[1]

    num = int((x1 - x0)/ hx_c)
    point_list_c = np.linspace(x0, x1, num + 1, dtype = float)
    point_list_c = list(point_list_c)

    X += point_list_c

    if x0 != 0:

        n_left = (nbElemHoriz - num) * (1 + (Lx - x1) / x0)**(-1)
        n_left = int(n_left)
        point_list_x_left = get_Geometrical_Suite(0, x0, 1/alpha, n_left)
        X += point_list_x_left

    else: n_left = 0

    n_right = nbElemHoriz - num - n_left
    point_list_x_right = get_Geometrical_Suite(x1, Lx, 1.02, n_right)

    X += point_list_x_right

    X = list(set(X))
    X.sort()
    return X


##############################################################################
##############################################################################

def get_Y_list(y_concentrated , nbElemVert , Ly):

    # See get_X_list

    alpha = 1.1

    Y = []

    y0 = y_concentrated[0][0]
    y1 = y_concentrated[0][1]
    hy_c = y_concentrated[1]

    num = int((y1 - y0)/ hy_c)
    point_list_c = np.linspace(y0, y1, num + 1, dtype = float)
    point_list_c = list(point_list_c)

    Y += point_list_c

    if y0 != 0:

        n_bottom = (nbElemVert - num) * (1 + (Ly - y1) / y0)**(-1)
        n_bottom = int(n_bottom)
        point_list_y_bottom = get_Geometrical_Suite(0, y0, 1/alpha , n_bottom)
        Y += point_list_y_bottom

    else: n_bottom = 0

    n_up = nbElemVert - num - n_bottom
    point_list_y_up = get_Geometrical_Suite(y1, Ly, alpha , n_up)

    Y += point_list_y_up

    Y = list(set(Y))
    Y.sort()

    return Y


##############################################################################
##############################################################################

def get_structured_mesh(Lx, Ly, nbElemHoriz , nbElemVert,\
                        x_concentrated , y_concentrated):

    # Return the connectivity and coordinates matrices of a structured quadri-
    # lateral mesh
    # Lx, Ly = horizontal and vertical size of the mesh
    # nbElemHoriz , nbElemVert = number of elements in the horizontal and
    # vertical directions

    # nbElem = total number of elements in the mesh
    nbElem = nbElemHoriz * nbElemVert

    # nbPoints = total number of points in the mesh
    nbPoints = (nbElemHoriz + 1) * (nbElemVert + 1)

    X = get_X_list(x_concentrated, nbElemHoriz, Lx)
    Y = get_Y_list(y_concentrated, nbElemVert, Ly)

    connectivity = np.zeros((nbElem, 4), dtype = int)
    coordinates = np.zeros((nbPoints, 2), dtype = float)

    i = 0

    print('\tGenerating coordinates matrix')

    for j in range(nbElemVert + 1):

        for k in range(nbElemHoriz + 1):

            coordinates[i, 0] = X[k]
            coordinates[i, 1] = Y[j]

            i = i + 1

    i = 0
    node = 0

    neighbour_matrix = [[] for i in range(nbPoints)]

    print('\tGenerating connectivity matrix')

    for j in range(nbElemVert):

        for k in range(nbElemHoriz):

            connectivity[i,:] = [node, node + 1, node + nbElemHoriz + 2,\
                                 node + nbElemHoriz + 1]

            neighbour_matrix[node].append(i)
            neighbour_matrix[node + 1].append(i)
            neighbour_matrix[node + nbElemHoriz + 2].append(i)
            neighbour_matrix[node + nbElemHoriz + 1].append(i)

            i += 1
            node += 1

        node += 1

    return connectivity, coordinates, neighbour_matrix


################################################################################
################################################################################

def get_inner_features(shape, connectivity, coordinates):

    # Return a list of inner_nodes, boundary_nodes and inner_elements for a
    # given shape
    # An inner_node is characterised by a negative value of the shape level
    # set function
    # One or more points of an inner element are inner nodes
    # connectivity = connectivity matrix of the mesh
    # coordinates = coordinates matrix of the mesh nodes
```

```

inner_nodes = []
boundary_nodes = []
inner_elements = []

nb_node = 0

# Loop over all the mesh nodes
# nb_node represents the number of the node we are looping on

for node in coordinates:

    # If the level set function is negative, this is an inner_node
    if shape.get_Level_Set(node) < 0: inner_nodes.append(nb_node)
    # If the level set function is null, this is a boundary node because
    # the node is exactly on the shape border
    elif shape.get_Level_Set(node) == 0: boundary_nodes.append(nb_node)

    nb_node += 1

nb_element = 0

# Loop over all the mesh elements
# nb_element represents the number of the element we are looping on

for element in connectivity:

    node_0 = element[0]
    x0 = coordinates[node_0,:]

    node_1 = element[1]
    x1 = coordinates[node_1,:]

    node_2 = element[2]
    x2 = coordinates[node_2,:]

    node_3 = element[3]
    x3 = coordinates[node_3,:]

    if node_0 in inner_nodes or node_1 in inner_nodes or \
    node_2 in inner_nodes or node_3 in inner_nodes:

        inner_elements.append(nb_element)

        # We need to set new boundary points
        # The points of the elements where the level set function is
        # strictly positive are the new boundary point of the mesh

        if shape.get_Level_Set(x0) > 0: boundary_nodes.append(node_0)
        if shape.get_Level_Set(x1) > 0: boundary_nodes.append(node_1)
        if shape.get_Level_Set(x2) > 0: boundary_nodes.append(node_2)
        if shape.get_Level_Set(x3) > 0: boundary_nodes.append(node_3)

    nb_element += 1

boundary_nodes = list(set(boundary_nodes))

return inner_elements, inner_nodes, boundary_nodes
```

Scripts/Pre_process_v6.py

```
# -*- coding: utf-8 -*-
"""
Created on Sat Mar  2 15:48:42 2019

@author: clement.lemardele
"""

import numpy as np
import math



def get_C(mu):

    C = np.array([[2*mu, 0, 0],
                  [0, 2*mu, 0],
                  [0,   0,  mu]])

    return C


##############################################################################
##############################################################################

def get_B(DNDx, J_inv):

    # Return matrix B
    # DNDx = matrix of shape functions derivatives
    # J_inv = inverse of the Jacobian matrix

    B = np.zeros([3, 8])

    for i in range(4):

        B[0, 2*i] = DNDx[i,0] * J_inv[0, 0] + DNDx[i, 1] * J_inv[1, 0]
        B[1, 2*i+1] = DNDx[i,0] * J_inv[0, 1] + DNDx[i, 1] * J_inv[1, 1]
        B[2, 2*i] = B[1, 2*i+1]
        B[2, 2*i+1] = B[0, 2*i]

    return B


##############################################################################
##############################################################################

def get_DNDx(X):

    # Return matrix of shape functions derivatives
    # X = coordinates of a given point

    x = X[0]
    y = X[1]

    DNDx = np.array([[0.25*(y-1), 0.25*(x-1)],
                     [-0.25*(y-1), -0.25*(x+1)],
                     [0.25*(y+1), 0.25*(x+1)],
                     [-0.25*(y+1), -0.25*(x-1)]])

    return DNDx
```

```


################################################################################
################################################################################

def get_J(X0, X1, X2, X3, X):

    # Return the jacobian of the isoparametric transformation
    # X0, X1, X2, X3 = coordinates of the 4 nodes of the element
    # X = coordinates of a given point

    x0 = X0[0]
    y0 = X0[1]

    x1 = X1[0]
    y1 = X1[1]

    x2 = X2[0]
    y2 = X2[1]

    x3 = X3[0]
    y3 = X3[1]

    x = X[0]
    y = X[1]

    J = np.array([ [.25*(y*(x0-x1+x2-x3)-x0+x1+x2-x3),\
                   .25*(x*(x0-x1+x2-x3)-x0-x1+x2+x3) ],
                   [.25*(y*(y0-y1+y2-y3)-y0+y1+y2-y3),\
                   .25*(x*(y0-y1+y2-y3)-y0-y1+y2+y3)] ])

    # Compute the inverse of the Jacobian matrix

    J_inv = np.zeros([2, 2])

    J_inv[0, 0] = 1.0 / J[0, 0]

    J_inv[1, 1] = 1.0 / J[1, 1]

    det_J = J[0,0] * J[1,1] - J[1,0] * J[0,1]

    return det_J, J, J_inv


################################################################################
################################################################################

def get_G(DNDx, J_inv):

    # Return G matrix
    # DNDx = matrix of shape functions derivatives
    # J_inv = inverse of the jacobian matrix

    G = np.zeros([8, 1])

    for i in range(0, 4):

        G[2*i, 0] = DNDx[i, 0] * J_inv[0, 0] + DNDx[i, 1] * J_inv[1, 0]
        G[2*i+1, 0] = DNDx[i, 0] * J_inv[0, 1] + DNDx[i, 1] * J_inv[1, 1]

    return G


##############################################################################
##############################################################################

def get_N(X):

    # Return the matrix of shape functions N
    # X = coordinates of a given point

    x = X[0]
    y = X[1]

    N = np.array([  [0.25*(x-1)*(y-1), 0, -0.25*(x+1)*(y-1), 0,\
                    0.25*(x+1)*(y+1), 0, -0.25*(x-1)*(y+1), 0 ],
                    [0, 0.25*(x-1)*(y-1), 0, -0.25*(x+1)*(y-1),\
                     0, 0.25*(x+1)*(y+1), 0, -0.25*(x-1)*(y+1)] ])

    return N


##############################################################################
##############################################################################

def get_Local_Matrices(X0, X1, X2, X3, rho, mu):

    # Return the stiffness, mass, gradient matrices of a given element
    # X0, X1, X2, X3 = coordinates of the 4 nodes
    # rho, mu = density, dynamic viscosity

    # Gauss integration points and weights for the quadrature of the stiffness
    # and mass matrices
    Xg = np.array([[-1/math.sqrt(3), 1/math.sqrt(3), 1],
                    [1/math.sqrt(3), 1/math.sqrt(3), 1],
                    [1/math.sqrt(3), -1/math.sqrt(3), 1],
                    [-1/math.sqrt(3), -1/math.sqrt(3), 1]])

    # Integrations points used for the quadrature of the N_DNDx and N_DNDy
    # matrices (see Shifted Boundary Conditions)
    Xref = np.array([[-1.0, -1.0],
                     [1.0, -1.0],
                     [1.0, 1.0],
                     [-1.0, 1.0]])

    MuuElem = np.zeros( (8,8), dtype = float )
    KuuElem = np.zeros( (8,8), dtype = float )
    DElem = np.zeros( (8,1), dtype = float )

    N_DNDx_elem = np.zeros( (4, 4), dtype = float )
    N_DNDy_elem = np.zeros( (4, 4), dtype = float )

    # Quadrature of the stiffness, mass and gradient matrices
    # Loop over the 4 Gauss points
    for X in Xg:

        DNDx = get_DNDx(X)

        det_J, J, J_inv = get_J(X0, X1, X2, X3, X[:2])

        B = get_B(DNDx, J_inv)
        N = get_N(X[:2])
        C = get_C(mu)

        matrix1 = np.dot(B.transpose(), C)
        KuuElem += np.dot(matrix1, B) * det_J * X[2]
        MuuElem += rho * np.dot(N.transpose(), N) * det_J * X[2]

        DElem += get_G(DNDx, J_inv) * det_J * X[2]

    # Quadrature of the N_DNDx and N_DNDy matrices
    # Loop over the Xref points
    for i in range(4):

        for k in range(4):

            DNDx = get_DNDx(Xref[i,:])

            det_J, J, J_inv = get_J(X0, X1, X2, X3, Xref[i,:])

            DNk_D_Xi = DNDx[k, 0]
            DNk_D_eta = DNDx[k, 1]

            DNk_Dx = DNk_D_Xi * J_inv[0,0] + DNk_D_eta * J_inv[1,0]
            DNk_Dy = DNk_D_Xi * J_inv[0,1] + DNk_D_eta * J_inv[1,1]

            N_DNDx_elem[i,k] = DNk_Dx * det_J
            N_DNDy_elem[i,k] = DNk_Dy * det_J


    return MuuElem, KuuElem, DElem, N_DNDx_elem, N_DNDy_elem
```

Scripts/Element_Bbar_v2.py

```
# -*- coding: utf-8 -*-
"""
Created on Thu Apr 25 12:57:01 2019

@author: clement.lemardele
"""

import numpy as np
import time


import Convective_solver_v2 as convective_solver
import Boundary_condition_solver as BC_solver


def solve_conjugate_gradient(A, b, ini):

    # Solve the linear system of equations Ax = b using the Conjugate Gradient
    # method
    # ini = initial point of the iterative method

    # Maximum number of iterations in the Conjugate Gradient method
    maxiter = 100
    # Tolerance over the residual Ax - b
    tol = 5e-3

    count = 1

    x_0 = ini
    d_0 = b - A.dot(x_0)
    r_0 = d_0

    ref = np.linalg.norm(r_0)

    while count <= maxiter and np.linalg.norm(r_0) >= tol * ref:

        vec = A.dot(d_0)
        alpha = np.dot(r_0.transpose(), r_0) / np.dot(d_0.transpose(), vec)

        x_1 = x_0 + alpha * d_0

        r_1 = r_0 - alpha * vec

        beta = np.dot(r_1.transpose(), r_1) / np.dot(r_0.transpose(), r_0)

        d_1 = r_1 + beta * d_0

        r_0 = r_1
        d_0 = d_1
        x_0 = x_1

        count +=1

    # If the Conjugate Gradient has not converged in maxiter iterations
    if count == maxiter + 1 and np.linalg.norm(r_0) >= tol * ref:
        print('Conjugate gradient convergence: not achieved')

    return x_0



################################################################################
################################################################################

def perform_time_integration(Muu_lumped, Muu_lumped_inv, Mpp, Mpp_inv, D, Kuu,\
                            dt, nbSteps, method, U_ini, P_ini, M_PI_inv,\
                            N_DNDx, N_DNDy, nodes_boundaries, coordinates,\
                            u_inlet, connectivity, neighbours):

    # Perform the time integration of the problem
    # Returns matrices solU and solP, which contains the nodal values of the
    # velocity and pressure at each time step

    size_problem_U = U_ini.shape
    size_problem_P = P_ini.shape

    # Initialisation of the integration
    U0 = U_ini
    P0 = P_ini

    # Time initialisation
    t = 0

    solU = np.zeros([size_problem_U[0], nbSteps + 1])
    solP = np.zeros([size_problem_P[0], nbSteps + 1])
```

```
count_iteration = 0

###########################################################################

# Forward Euler numerical scheme
if method == "FE":

    print("Beginning Forward Euler time integration")
    t0 = time.time()

    for i in range(nbSteps):

        # We compute the nodal values of the approximation
        # of the gradient of U
        PI = BC_solver.get_PI(M_PI_inv, N_DNDx, N_DNDy, U0)

        # We apply boundary conditions
        DOF = BC_solver.apply_boundary_condition(U0, PI, nodes_boundaries,\
                                                 coordinates, u_inlet, i)

        # We store U0 for post-process
        solU[:,i] = U0[:,0]
        solP[:,i] = P0[:,0]

        # Computation of the convective term
        U_transport = convective_solver.transport(U0, coordinates, dt,\
                                                  connectivity, DOF, neighbours)

        # Forward Euler numerical scheme
        vec1 = - np.dot(Kuu, U0) + np.dot(D, P0)
        vec1 = Muu_lumped_inv.dot(vec1)
        U1 = dt * vec1 + U_transport

        vec2 = - D.transpose().dot(U0)
        vec2 = Mpp_inv.dot(vec2)
        P1 = P0 + dt * vec2

        t += dt

        if count_iteration == 10000:

            # Indicate how the simulation is going on
            print("Time integration: %.2f" % t)
            count_iteration = 0

        U0 = U1
        P0 = P1

        count_iteration += 1

    PI = BC_solver.get_PI(M_PI_inv, N_DNDx, N_DNDy, U0)
    DOF = BC_solver.apply_boundary_condition(U0, PI, nodes_boundaries,\
                                             coordinates, u_inlet, nbSteps)

    solU[:, nbSteps] = U0[:,0]
    solP[:, nbSteps] = P0[:,0]

    t1 = time.time()
    simulation_time = t1 - t0
```

```
        print("Simulation time: %.3f" % simulation_time)

    ###########################################################################

    # Fractional Step method
    if method == "FS":

        print("Beginning Fractional Step time integration")
        t0 = time.time()

        U_hat_0 = np.zeros((size_problem_U[0], 1), dtype = float)
        dP = np.zeros((size_problem_P[0],1), dtype = float)
        P1 = np.zeros((size_problem_P[0], 1), dtype = float)

        # Previous computation of useful matrices
        L = Muu_lumped_inv.dot(D)
        L = dt * D.transpose().dot(L)
        L = 1/dt * Mpp + L

        matrix = Muu_lumped_inv.dot(D)

        for i in range(nbSteps):

            PI = BC_solver.get_PI(M_PI_inv, N_DNDx, N_DNDy, U0)
            DOF = BC_solver.apply_boundary_condition(U0, PI, nodes_boundaries,\
                                               coordinates, u_inlet, i)

            solU[:, i] = U0[:,0]
            solP[:, i] = P0[:,0]

            # Computation of the convective term
            U_transport = convective_solver.transport(U0, coordinates, dt,\
                                                    connectivity, DOF, neighbours)

            # First equation
            vec1 = - Kuu.dot(U_hat_0) + D.dot(P0)
            vec1 = Muu_lumped_inv.dot(vec1)
            U_hat_1 = dt * vec1 + U_transport

            # Second equation
            vec2 = - D.transpose().dot(U_hat_1)
            dP = solve_conjugate_gradient(L, vec2, dP)

            # Pressure update
            P1 = P0 + dP

            # Third equation
            U1 = U_hat_1 + dt * matrix.dot(dP)

            t += dt

            if count_iteration == 1000:

                print("\tTime integration: %.4f" % t)
                count_iteration = 0

            U0 = U1
            U_hat_0 = U_hat_1
            P0 = P1
        count_iteration += 1

    PI = BC_solver.get_PI(M_PI_inv, N_DNDx, N_DNDy, U0)
    DOF = BC_solver.apply_boundary_condition(U0, PI, nodes_boundaries,\
                                     coordinates, u_inlet, nbSteps)

    solU[:, nbSteps] = U0[:,0]
    solP[:, nbSteps] = P0[:,0]

    t1 = time.time()
    simulation_time = t1 - t0
    print('\t\tSimulation_time: %.3f' % simulation_time)


##########################################################################

# Backward Euler method
if method == "BE":

    print("Beginning Backward Euler time integration")
    t0 = time.time()

    # Computation of useful matrices
    D_Mpp_inv = D.dot(Mpp_inv)
    L = D_Mpp_inv.dot(D.transpose())

    for i in range(nbSteps):

        PI = BC_solver.get_PI(M_PI_inv, N_DNDx, N_DNDy, U0)
        DOF = BC_solver.apply_boundary_condition(U0, PI, nodes_boundaries,\
                                         coordinates, u_inlet, i)

        solU[:, i] = U0[:,0]
        solP[:, i] = P0[:,0]

        U_transport = convective_solver.transport(U0, coordinates, dt,\
                                          connectivity, DOF)

        # Update of velocity
        vec1 = D.dot(P0) + 1/dt * Muu_lumped.dot(U_transport)
        matrix1 = 1/dt * Muu_lumped + Kuu + dt * L
        U1 = solve_conjugate_gradient(matrix1, vec1, U0)

        # Update of pressure
        vec2 = D.transpose().dot(U1)
        vec2 = Mpp_inv.dot(vec2)
        P1 = P0 - dt * vec2

        t += dt

        if count_iteration == 100:

            print("\tTime integration: %.3f" % t)
            count_iteration = 0

        U0 = U1
        P0 = P1

        count_iteration += 1
        PI = BC_solver.get_PI(M_PI_inv, N_DNDx, N_DNDy, U0)
        DOF = BC_solver.apply_boundary_condition(U0, PI, nodes_boundaries,\
                                                    coordinates, u_inlet, nbSteps)

        solU[:, nbSteps] = U0[:,0]
        solP[:, nbSteps] = P0[:,0]

        t1 = time.time()
        simulation_time = t1 - t0
        print('\t\tSimulation time: %.3f' % simulation_time)


    return solU, solP
```

Scripts/Solver_Navier_Stokes_v3.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun Aug 11 16:40:29 2019

@author: clement lemardele
"""

import numpy as np
import math


def get_PI(M_PI_inv, N_DNDx, N_DNDy, U):

    # Compute the value of the PI matrix at each point of the mesh
    # See Shifted Boundary Conditions method
    # Return the nodal values of the velocity gradient approximation

    size_U = U.shape
    nbNodes = int(size_U[0] / 2.0)
    PI = np.zeros((nbNodes, 4), dtype = float)

    even_number = [nb for nb in range(2*nbNodes) if not nb % 2]
    odd_number = [nb for nb in range(2*nbNodes) if nb % 2]

    Ux = U[even_number,:]
    Uy = U[odd_number,:]

    PI[:,0] = M_PI_inv.dot(N_DNDx.dot(Ux))[:,0]
    PI[:,1] = M_PI_inv.dot(N_DNDy.dot(Ux))[:,0]
    PI[:,2] = M_PI_inv.dot(N_DNDx.dot(Uy))[:,0]
    PI[:,3] = M_PI_inv.dot(N_DNDy.dot(Uy))[:,0]

    return PI

##############################################################################
##############################################################################


def get_Orthogonal_Vector(vector):

    a = vector[0, 0]
    b = vector[1, 0]

    return 1/math.sqrt(a**2 + b**2) * np.array([[b], [-a]])
```

```
###########################################################################
###########################################################################

def apply_boundary_condition(U, PI, nodes_boundaries, coordinates, u_inlet,\
                             index):

    # Apply the Shifted Boundary Conditions
    # U = value of velocity DOFS at current time step
    # PI = approximation of the gradient of U
    # nodes_boundaries = list of boundary nodes
    # coordinates = coordinates matrix of the mesh nodes
    # u_inlet = value of inlet flow at current time step
    # index = index of the current time step

    size_coordinates = coordinates.shape
    nbPoints = size_coordinates[0]

    DOF = [i for i in range(nbPoints)]

    Lx = coordinates[nbPoints - 1, 0]
    Ly = coordinates[nbPoints - 1, 1]

    for node_boundary in nodes_boundaries:

        x_I = np.zeros((2,1), dtype = float)
        x_P = np.zeros((2,1), dtype = float)

        u_I = np.zeros((2,1), dtype = float)
        u_P = np.zeros((2,1), dtype = float)

        numNode = node_boundary[0]
        geometry = node_boundary[1]
        BC_type = node_boundary[2]

        node_coordinate = coordinates[numNode,:]

        # We get the closest point which is on the geometry
        closest_point = geometry.get_Closest_Point(node_coordinate)

        x_I[:,0] = node_coordinate
        x_P[:,0] = closest_point

        # We reconstruct the gradient of the velocity at point I
        nabla_U = np.zeros((2,2), dtype = float)
        nabla_U[0,0] = PI[numNode,0]
        nabla_U[0,1] = PI[numNode,1]
        nabla_U[1,0] = PI[numNode,2]
        nabla_U[1,1] = PI[numNode,3]

        # We apply the boundary conditions depending on the type of boundary
        # conditions we imposed at the given geometry
        #################################################################
        if BC_type == 'no_slip':

            u_P[:,0] = [0.0, 0.0]

            u_I = u_P - np.dot(nabla_U, x_P - x_I)

        U[2*numNode : 2*numNode + 2, 0] = u_I[:,0]

        if numNode in DOF: DOF.remove(numNode)

    ########################################################################

    elif BC_type == 'uniform_inlet':

        normal = geometry.get_Normal(x_P)
        u_P[:,0] = - u_inlet[index] * normal[:,0]

        u_I = u_P - np.dot(nabla_U, x_P - x_I)

        U[2*numNode : 2*numNode + 2, 0] = u_I[:,0]

        if numNode in DOF: DOF.remove(numNode)

    ########################################################################

    elif BC_type == 'parabolic_inlet':

        if numNode in DOF: DOF.remove(numNode)

        if geometry.side == 'left' or geometry.side == 'right':

            normal = geometry.get_Normal(x_P)
            y_closest = closest_point[1]

            u_P[:,0] = - 4 * u_inlet[index] / Ly**2 *\
                        y_closest * (Ly - y_closest) * normal[:,0]

            u_I = u_P - np.dot(nabla_U, x_P - x_I)

            U[2*numNode : 2*numNode + 2, 0] = u_I[:,0]

        if geometry.side == 'bottom' or geometry.side == 'up':

            normal = geometry.get_Normal(x_P)
            x_closest = closest_point[0]

            u_P[:,0] = - 4 * u_inlet[index] / Lx**2 \
                        * x_closest * (Lx - x_closest) * normal[:,0]

            u_I = u_P - np.dot(nabla_U, x_P - x_I)

            U[2*numNode : 2*numNode + 2, 0] = u_I[:,0]

    ########################################################################

    elif BC_type == 'logarithmic_inlet':

        if numNode in DOF: DOF.remove(numNode)

        if geometry.side == 'left' or geometry.side == 'right':

            normal = geometry.get_Normal(x_P)
            y_closest = closest_point[1]

            y_0 = Ly / (math.exp(1) - 1)
            u_P[:,0] = - u_inlet[index] * math.log(1 + y_closest/y_0) \
                            * normal[:,0]

                u_I = u_P - np.dot(nabla_U, x_P - x_I)

                U[2*numNode : 2*numNode + 2, 0] = u_I[:,0]

            if geometry.side == 'bottom' or geometry.side == 'up':

                normal = geometry.get_Normal(x_P)
                x_closest = closest_point[0]

                x_0 = Lx / (math.exp(1) - 1)
                u_P[:,0] = - u_inlet[index] * math.log(1 + x_closest/x_0) \
                            * normal[:,0]

                u_I = u_P - np.dot(nabla_U, x_P - x_I)

                U[2*numNode : 2*numNode + 2, 0] = u_I[:,0]

        ######################################################################

        elif BC_type == 'tangential_inlet':

            if numNode in DOF: DOF.remove(numNode)

            normal = geometry.get_Normal(x_P)
            orth_normal = get_Orthogonal_Vector(normal)

            u_P[:,0] = 0.1 * orth_normal[:,0]

            u_I = u_P - np.dot(nabla_U, x_P - x_I)

            U[2*numNode : 2*numNode + 2, 0] = u_I[:,0]

        ######################################################################

        elif BC_type == 'slip':

            if geometry.side == 'left' or geometry.side == 'right':
                U[2*numNode, 0] = 0.0

            elif geometry.side == 'bottom' or geometry.side == 'up':
                U[2*numNode + 1, 0] = 0.0

        ######################################################################

    return DOF
```

Scripts/Boundary_condition_solver.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun Aug 11 16:34:25 2019

@author: clement lemardele
"""

def is_In(x_P, element, coordinates):

    # Return True if the point x_P is within a given element
    # Otherwise return False

    x = x_P[0]
    y = x_P[1]

    node_0 = element[0]
    node_2 = element[2]

    x0 = coordinates[node_0,:]
    x2 = coordinates[node_2,:]

    x_A = x0[0]
    y_A = x0[1]

    x_B = x2[0]
    y_B = x2[1]

    test = False

    if x_A <= x and x <= x_B and y_A <= y and y <= y_B: test = True

    return test


################################################################################
################################################################################


def get_Element_Number(x_P, num_node, coordinates, connectivity, neighbours):

    # Return the number of the element where the point x_P is situated

    element_list = neighbours[num_node]
    found_element = element_list[0]

    for num_element in element_list:

        element = connectivity[num_element,:]

        if is_In(x_P, element, coordinates) == True: found_element = num_element

    return found_element


################################################################################
################################################################################


def get_Velocity_Interpolation(U, x_point, connectivity, num_element,\
                                                coordinates):

    # Return the value of the velocity at point x_point, by doing the
    # interpolation in the element num_element
    # U = nodal values of the velocity field
    # x_point = coordinates of a given point
    # num_element = number of the element where we are doing the interpolation
    # connectivity = connectivity matrix of the mesh
    # coordinates coordinates matrix of the mesh

    x = x_point[0]
    y = x_point[1]

    # We get the nodes of the element
    element = connectivity[num_element,:]

    node_0 = element[0]
    node_1 = element[1]
    node_2 = element[2]
    node_3 = element[3]

    x0 = coordinates[node_0,:]
    x1 = coordinates[node_1,:]
    x3 = coordinates[node_3,:]

    x_A = x0[0]
    x_B = x1[0]
    y_A = x0[1]
    y_B = x3[1]

    u_0 = U[2*node_0 : 2*node_0 + 2,:]
    u_1 = U[2*node_1 : 2*node_1 + 2,:]
    u_2 = U[2*node_2 : 2*node_2 + 2,:]
    u_3 = U[2*node_3 : 2*node_3 + 2,:]

    N0 = (x - x_B) * (y - y_B) / (x_A - x_B) / (y_A - y_B)
    N1 = (x - x_A) * (y - y_B) / (x_B - x_A) / (y_A - y_B)
    N2 = (x - x_A) * (y - y_A) / (x_B - x_A) / (y_B - y_A)
    N3 = (x - x_B) * (y - y_A) / (x_A - x_B) / (y_B - y_A)

    # Interpolation using shape functions

    return N0 * u_0 + N1 * u_1 + N2 * u_2 + N3 * u_3


###############################################################################
###############################################################################


def transport(U, coordinates, dt, connectivity, DOF, neighbours):

    # Return the nodal values of the purely-convective problem solution at
    # following time step

    # U = nodal values of the current velocity field
    # coordinates = coordinates matrix of the mesh
    # dt = time step
    # connectivity = connectivity matrix of the mesh
    # DOF = the problem degrees of freedom
    # neighbours = matrix containing all the elements which contain a given
    #              node

    # Initialisation
    U_hat_1 = U

    # First step forward
    for i in DOF:

        x_node = coordinates[i,:]
        u_node = U[2*i : 2*i+2, 0]

        x_former = x_node - dt * u_node

        # Get the number of the element where the fluid particle was
        former_element = get_Element_Number(x_former, i, coordinates,\
                                            connectivity, neighbours)

        # Interpolate velocity
        u_hat_node_1 = get_Velocity_Interpolation(U, x_former, connectivity,\
                                former_element, coordinates)

        U_hat_1[2*i : 2*i + 2] = u_hat_node_1


    U_hat_0 = U

    # Backward step
    for i in DOF:

        x_node = coordinates[i,:]

        x_future = x_node + dt * U_hat_1[2*i : 2*i+2, 0]

        future_element = get_Element_Number(x_future, i, coordinates,\
                                            connectivity, neighbours)

        u_hat_node_0 = get_Velocity_Interpolation(U_hat_1, x_future,\
                                    connectivity, future_element, coordinates)

        U_hat_0[2*i : 2*i + 2] = u_hat_node_0

    # Correction and compensation
    U_hat_hat = U + 0.5 * (U - U_hat_0)

    U_1 = U

    # Second forward step
    for i in DOF:

        x_node = coordinates[i,:]

        x_former = x_node - dt * U_hat_hat[2*i : 2*i + 2, 0]

        former_element = get_Element_Number(x_former, i, coordinates,\
                                            connectivity, neighbours)

        u_node_1 = get_Velocity_Interpolation(U_hat_hat, x_former,\
                                connectivity, former_element, coordinates)

        U_1[2*i : 2*i + 2] = u_node_1

    return U_1
```

Scripts/Convective_solver_v2.py

```
# -*- coding: utf-8 -*-
"""
Created on Thu Apr 25 12:44:04 2019

@author: clement.lemardele
"""
```

```
# This module contains all the functions needed for the post−process of the
# solution

from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import math



def compute_max_norm(U):

    # U = (u_1 v_1 u_2 v_2 ... u_N v_N)
    # (u_1 v_1) for instance is the velocity of mesh node 1
    # Return the maximum velocity norm for all mesh nodes

    nbPoints = int(len(U) / 2)
    Unorm = [0 for i in range(nbPoints)]

    for i in range(nbPoints):

        Unorm[i] = math.sqrt(U[2*i]**2 + U[2*i + 1]**2)

    return max(Unorm)

###############################################################################
###############################################################################

def create_image(Lx, Ly):

    # See the Pre−process module for more commentary

    nb_pixels = 1200

    if Lx == max(Lx, Ly):

        image_array = 255 * np.ones((int(Ly/Lx*nb_pixels), nb_pixels, 3))

    else:

        image_array = 255 * np.ones((nb_pixels, int(Lx/Ly*nb_pixels), 3))


    return image_array

###############################################################################
###############################################################################

def get_element_picture(nx, ny, colour):

    # Return a RGB matrix corresponding to a rectangular element of the
    # given colour
    # nx, ny = number of horizontal and vertical pixels

    image_array = np.ones((ny, nx, 3), dtype = int)

    for i in range(ny):
```

```python

        for j in range(nx):

            image_array[i,j] = colour

    return image_array


################################################################################
################################################################################

def get_pressure_color(min_pressure, max_pressure, element_pressure):

    # Return the colour of given element_pressure
    # Convert a pressure into a colour using the following code
    # Blue colour = min_pressure
    # Red colour = max_pressure

    scale = (element_pressure - min_pressure) / (max_pressure - min_pressure)

    if scale <= 0.25:
        RGB = np.array([0, int(scale/0.25 * 255), 255])

    elif 0.25 < scale <= 0.5:
        RGB = np.array([0., 255, int(255-(scale - 0.25)/0.25 * 255)])

    elif 0.5 < scale <= 0.75:
        RGB = np.array([int((scale - 0.5)/0.25 * 255), 255, 0.])

    else:
        RGB = np.array([255, int(255 - (scale - 0.75)/0.25 * 255), 0.])

    return RGB


################################################################################
################################################################################

def convert_to_list(color_matrix):

    size = color_matrix.shape

    matrix_list = []

    for i in range(size[0]):

        for j in range(size[1]):

            new_tuple = (int(color_matrix[i,j,0]), \
                         int(color_matrix[i,j,1]),\
                         int(color_matrix[i,j,2]))

            matrix_list.append(new_tuple)

    return matrix_list


################################################################################
################################################################################

```

```
def get_color(RGB):

    # Return the corresponding colour of a RGB pixel
    # White = [255, 255, 255]
    # Black = [0, 0, 0]
    # Red = [255, 0, 0]

    if RGB[0] == 255 and RGB[1] == 255 and RGB[2] == 255:
        return "white"
    elif RGB[0] == 0 and RGB[1] == 0 and RGB[2] == 0:
        return "black"
    elif RGB[0] == 255 and RGB[1] == 0 and RGB[2] == 0:
        return "red"
    else:
        return "undefined_colour"

###############################################################################
###############################################################################

def display_pressure(inner_elements, connectivity, coordinates, min_pressure,\
              max_pressure, pressure, index, shapes, nbElemVert, nbElemHoriz):

    size_coordinates = coordinates.shape

    nbPoints = size_coordinates[0]

    Lx = coordinates[nbPoints - 1, 0]
    Ly = coordinates[nbPoints - 1, 1]

    image_array = create_image(Lx, Ly)

    size_image = image_array.shape

    horizontal_pixel = size_image[1]
    vertical_pixel = size_image[0]

    count = 0
    pixel_y = vertical_pixel

    for j in range(nbElemVert):

        pixel_x = 0

        for i in range(nbElemHoriz):

            element = connectivity[count,:]

            node_0 = element[0]
            node_2 = element[2]

            x0 = coordinates[node_0,:]
            x2 = coordinates[node_2,:]

            x_A = x0[0]
            y_A = x0[1]

            x_B = x2[0]
            y_B = x2[1]

            length_x = int((x_B - x_A) / Lx * horizontal_pixel)
            length_y = int((y_B - y_A) / Ly * vertical_pixel)

            if count not in inner_elements:

                # We convert the value of the pressure into a colour
                element_color = get_pressure_color(min_pressure, max_pressure,\
                                                   pressure[count])

                # We get the corresponding coloured rectangle
                element_picture =\
                get_element_picture(length_x, length_y, element_color)

                # We put it in the picture matrix
                image_array[pixel_y - length_y : pixel_y,\
                            pixel_x : pixel_x + length_x] \
                            = element_picture

            count += 1
            pixel_x += length_x

        pixel_y = pixel_y - length_y

    image_tuple = convert_to_list(image_array)

    img = Image.new("RGB", (size_image[1], size_image[0]))

    img.putdata(image_tuple)
    img.save("Pressure/Pressure_t" + str(index) + ".jpg")
    img.close()


################################################################################
################################################################################


def velocity_Quiver(U, coordinates, dt, tmax, index, shapes, Re, Lx, Ly):

    # Save the quiver plot of the velocity field U

    nb_DOF = len(U)

    even = [nb for nb in range(nb_DOF) if not nb%2]
    odd = [nb for nb in range(nb_DOF) if nb%2]

    Ux = U[even]
    Uy = U[odd]

    plt.figure(figsize = (20, Ly / Lx * 20))
    ax = plt.gca()
    ax.quiver(coordinates[:,0], coordinates[:,1], Ux, Uy, scale = 300)

    for shape in shapes:

        shape_points = shape.get_Shape_Coordinates()

        X = shape_points[:,0]
        Y = shape_points[:,1]

        plt.scatter(X, Y, 1)
    plt.title('Re = ' + str(Re) + ', t = ' + str(round(dt*index, 1)))

    plt.savefig('Velocity/Velocity_t' + str(index) + '.jpg')

    plt.close()


##############################################################################
##############################################################################


def get_Vorticity(x, x0, x1, x2, x3, u0, u1, u2, u3):

    # Return the vorticity at point x
    # x0, x1, x2, x3: coordinates of the element nodes
    # u0, u1, u2, u3: nodal values of the velocity field at point x0, x1, x2
    #                 and x3

    x_P = x[0]
    y_P = x[1]

    x_A = x0[0]
    x_B = x1[0]
    y_A = x0[1]
    y_B = x3[1]

    # Compute the shape functions derivatives
    dN0_dx = (y_P - y_B) / (x_A - x_B) / (y_A - y_B)
    dN0_dy = (x_P - x_B) / (x_A - x_B) / (y_A - y_B)

    dN1_dx = (y_P - y_B) / (x_B - x_A) / (y_A - y_B)
    dN1_dy = (x_P - x_A) / (x_B - x_A) / (y_A - y_B)

    dN2_dx = (y_P - y_A) / (x_B - x_A) / (y_B - y_A)
    dN2_dy = (x_P - x_A) / (x_B - x_A) / (y_B - y_A)

    dN3_dx = (y_P - y_A) / (x_A - x_B) / (y_B - y_A)
    dN3_dy = (x_P - x_B) / (x_A - x_B) / (y_B - y_A)

    dv_dx = u0[1] * dN0_dx + u1[1] * dN1_dx + u2[1] * dN2_dx + u3[1] * dN3_dx
    du_dy = u0[0] * dN0_dy + u1[0] * dN1_dy + u2[0] * dN2_dy + u3[0] * dN3_dy

    return dv_dx - du_dy


##############################################################################
##############################################################################


def get_Vorticity_Field(U, connectivity, coordinates):

    # Return the vorticity field computed after a given velocity field U

    size_connectivity = connectivity.shape
    nbElem = size_connectivity[0]

    vorticity_field = [0 for i in range(nbElem)]

    for i in range(nbElem):

        element = connectivity[i,:]

        node_0 = element[0]
        node_1 = element[1]
        node_2 = element[2]
        node_3 = element[3]

        x0 = coordinates[node_0,:]
        x1 = coordinates[node_1,:]
        x2 = coordinates[node_2,:]
        x3 = coordinates[node_3,:]

        u0 = U[2*node_0 : 2*node_0 + 2]
        u1 = U[2*node_1 : 2*node_1 + 2]
        u2 = U[2*node_2 : 2*node_2 + 2]
        u3 = U[2*node_3 : 2*node_3 + 2]

        # Coordinates of the element center
        x_center = 0.25 * x0 + 0.25 * x1 + 0.25 * x2 + 0.25 * x3

        # Get the value of the vorticity in the center of the element
        vorticity = get_Vorticity(x_center, x0, x1, x2, x3, u0, u1, u2, u3)

        vorticity_field[i] = abs(vorticity)

    return vorticity_field


##############################################################################
##############################################################################


def display_Vorticity(inner_elements, connectivity, coordinates, min_vorticity,\
                      max_vorticity, vorticity, index, shapes, nbElemVert,
    nbElemHoriz):

    size_coordinates = coordinates.shape

    nbPoints = size_coordinates[0]

    Lx = coordinates[nbPoints - 1, 0]
    Ly = coordinates[nbPoints - 1, 1]

    image_array = create_image(Lx, Ly)

    size_image = image_array.shape

    horizontal_pixel = size_image[1]
    vertical_pixel = size_image[0]

    count = 0
    pixel_y = vertical_pixel

    for j in range(nbElemVert):

        pixel_x = 0

        for i in range(nbElemHoriz):

            element = connectivity[count,:]

            node_0 = element[0]
            node_2 = element[2]

            x0 = coordinates[node_0,:]
            x2 = coordinates[node_2,:]

            x_A = x0[0]
            y_A = x0[1]

            x_B = x2[0]
            y_B = x2[1]

            length_x = int((x_B - x_A) / Lx * horizontal_pixel)
            length_y = int((y_B - y_A) / Ly * vertical_pixel)

            if count not in inner_elements:

                # We convert the value of the pressure into a colour
                element_color =\
                get_pressure_color(min_vorticity, max_vorticity,\
                                   vorticity[count])

                # We get the corresponding coloured rectangle
                element_picture =\
                get_element_picture(length_x, length_y, element_color)

                # We put it in the picture matrix
                image_array[pixel_y - length_y : pixel_y,\
                            pixel_x : pixel_x + length_x] \
                            = element_picture

            count += 1
            pixel_x += length_x

        pixel_y = pixel_y - length_y

    image_tuple = convert_to_list(image_array)

    img = Image.new("RGB", (size_image[1], size_image[0]))

    img.putdata(image_tuple)
    img.save("Vorticity/Vorticity_t" + str(index) + ".jpg")
    img.close()


################################################################################
################################################################################


def find_Interval(x, data):

    n_data = len(data)
    test = False
    index = 0

    while test == False and index < n_data - 1:

        if data[index] <= x and x <= data[index + 1]:

            test = True

        index += 1

    index = index - 1

    if test == False: index += 1

    if index == n_data - 1:

        if x < data[0]: return -1
        if x > data[-1]: return index

    else: return index


###############################################################################
###############################################################################


def get_Element_Number(x_node, coordinates):

    # Return the number of the element where the point x_node is situated

    x = x_node[0]
    y = x_node[1]

    X_list = coordinates[:,0]
    X_list = list(set(X_list))
    X_list.sort()

    Y_list = coordinates[:,1]
    Y_list = list(set(Y_list))
    Y_list.sort()

    nbElemHoriz = len(X_list) - 1
    nbElemVert = len(Y_list) - 1

    num_column = find_Interval(x, X_list)
    num_line = find_Interval(y, Y_list)

    if num_column == -1: num_column = 0
    if num_line == -1: num_line = 0

    if num_column == nbElemHoriz: num_column = num_column - 1
    if num_line == nbElemVert: num_line = num_line - 1

    element_number = num_line * nbElemHoriz + num_column

    return element_number


###############################################################################
###############################################################################


def get_Variable_time(variable, solU, solP, connectivity, coordinates,\
                      x, y):

    # Return the value of a given variable at point (x,y) as a function of
    # time
    # variable = pressure, x_velocity (horizontal component of the velocity),
    #            y_velocity (vertical component of the velocity)

    num_element = get_Element_Number([x,y], coordinates)

    if variable == 'pressure':

        pressure_point = solP[num_element,:]

        return pressure_point

    if variable == 'x_velocity':

        element = connectivity[num_element,:]

        node_0 = element[0]
        node_1 = element[1]
        node_2 = element[2]
        node_3 = element[3]

        x0 = coordinates[node_0,:]
        x1 = coordinates[node_1,:]
        x3 = coordinates[node_3,:]

        x_A = x0[0]
        x_B = x1[0]
        y_A = x0[1]
        y_B = x3[1]

        u_0 = solU[2*node_0,:]
        u_1 = solU[2*node_1,:]
        u_2 = solU[2*node_2,:]
        u_3 = solU[2*node_3,:]

        N0 = (x - x_B) * (y - y_B) / (x_A - x_B) / (y_A - y_B)
        N1 = (x - x_A) * (y - y_B) / (x_B - x_A) / (y_A - y_B)
        N2 = (x - x_A) * (y - y_A) / (x_B - x_A) / (y_B - y_A)
        N3 = (x - x_B) * (y - y_A) / (x_A - x_B) / (y_B - y_A)

        return N0 * u_0 + N1 * u_1 + N2 * u_2 + N3 * u_3

    if variable == 'y_velocity':

        element = connectivity[num_element,:]

        node_0 = element[0]
        node_1 = element[1]
        node_2 = element[2]
        node_3 = element[3]

        x0 = coordinates[node_0,:]
        x1 = coordinates[node_1,:]
        x3 = coordinates[node_3,:]

        x_A = x0[0]
        x_B = x1[0]
        y_A = x0[1]
        y_B = x3[1]
    v_0 = solU[2*node_0 + 1,:]
    v_1 = solU[2*node_1 + 1,:]
    v_2 = solU[2*node_2 + 1,:]
    v_3 = solU[2*node_3 + 1,:]

    N0 = (x - x_B) * (y - y_B) / (x_A - x_B) / (y_A - y_B)
    N1 = (x - x_A) * (y - y_B) / (x_B - x_A) / (y_A - y_B)
    N2 = (x - x_A) * (y - y_A) / (x_B - x_A) / (y_B - y_A)
    N3 = (x - x_B) * (y - y_A) / (x_A - x_B) / (y_B - y_A)

    return N0 * v_0 + N1 * v_1 + N2 * v_2 + N3 * v_3
```

Scripts/Post_process_v7.py