BEI

Notice OpenFOAM

Chady KERYAKOS
Kévin LEPETIT
Arthur NANGO
Thiago VARELLA

Février-Mars

Table des matières

1 Installation et choix de la version

Il existe plusieurs versions maintenues à jour d'**OpenFOAM** dont deux principales : OpenFOAM 20xx (en particulier les versions 2206 et 2212) et OpenFOAM x (en particulier la version 10).

En pratique il n'existe pas de différence en ce qui concerne les utilitaires de base pour chacune des versions : un projet "basique" est donc compatible pour les deux versions d'**OpenFOAM**. Ce qui diffère est l'emploi d'utilitaires spécifiques, par exemple pour le post traitement.

Dans le cadre de ce BEI, nous avons décidé d'utiliser la version 2206. En effet, même s'il existe une version plus récente (à savoir, la version 2212) nous avons commencé ce bureau d'études quand cette dernière n'existait pas, et il ne nous parait pas judicieux de tout devoir adapter pour effectuer un changement de version.

L'installation d'**OpenFOAM** doit se faire une machine **Linux** (par exemple **Ubuntu**). Si l'on dispose seulement d'une machine **Windows** il est possible d'installer la couche de compatibilité **Windows-Linux** WSL 2. Lorsque l'on dispose d'un accès **Linux** sur sa machine, on peut installer **OpenFOAM** à partir du guide d'installation officiel.

2 Présentation des structures générales du projet

Pour chaque projet **OpenFOAM**, on est amené à éditer possiblement ou bien le répertoire principal du projet, ou bien le solver employé pour le projet ou bien enfin les conditions aux limites utilisées pour le projet.

On peut noter que l'on peut aussi programmer toute la partie post traitement mais on estime que l'utilisation du logiciel de visualisation **ParaView** est suffisante, donc dans le cadre de notre BEI, il n'est pas prévu de programmation en post traitement.

2.1 Répertoire du projet

Tout répertoire principal d'un projet **OpenFOAM** possède le même squelette :

- Un répertoire *0* correspondant aux conditions initiales. Il contient en général autant de fichiers qu'il y a de grandeurs à initialiser (par exemple *U*, *P*...).
- Un répertoire *constant* correspondant à toutes les informations du modèle utilisé (par exemple la viscosité) dans un fichier nommé en général *transportProperties* ainsi que les informations du maillage utilisé (une fois celui-ci construit) dans le sous répertoire *polyMesh*.
- Un répertoire *system* contenant dans plusieurs fichiers toutes les informations nécessaires à l'exécution du calcul (on peut citer par exemple le pas de temps, le temps de calcul, les schémas de discrétisation utilisés par exemple..).

2.1.1 Répertoire *0*

Dans ce dossier, doivent se trouver autant de fichier qu'il y a de grandeurs à initialiser. Chaque fichier doit porter le nom de la variable à initialiser. Prenons l'exemple du fichier *U* : comme tous les fichiers d'un projet **OpenFOAM**, il commence par un *header*, c'est une banderole contenant toutes les informations utiles à la fois pour le lecteur du fichier, que pour la machine.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2206                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

[...]
```

Listing 1 – Extrait du fichier *U* du dossier *0*.

Il contient entre autres la version d'**OpenFOAM** conseillée pour l'exploitation du fichier, la classe du fichier et son nom principalement. Cette partie constitue en quelque sorte l'identité du fichier, et puisqu'elle se retrouve à chaque fichier, il est choisi de ne plus la présenter par la suite car elle présente peu d'intérêt une fois vue pour la première fois.

Ensuite, la structure du fichier sera identique à chacun de ces fichiers du répertoire *0* ce qui est un cas particulier des fichiers du répertoire *0* : le code se poursuit par l'assignation d'une valeur de la grandeur dans le champ interne, c'est à dire hors frontières (où s'appliqueront les conditions aux limites), ainsi que par la spécification de la dimension physique de la grandeur :

```
[...]

dimensions      [0 1 -1 0 0 0 0];
```

```
internalField   uniform (0 0 0);

[...]
```

Listing 2 – Extrait du fichier *U* du dossier *0*. (2)

Dans notre cas, la vitesse s'exprime en mètre par seconde d'où le vecteur [0 1 -1 0 0 0 0] pour la dimension. De manière générale, dans l'ordre de lecture, les composantes du vecteur dimension s'identifient à : masse ; longueur ; temps ; température ; quantité de matière ; intensité électrique et intensité lumineuse.

Nous implémentons une vitesse initialement nulle dans tout le domaine, ce que l'on peut faire avec l'appel *uniform (0 0 0)*.

Enfin, il convient de spécifier les valeurs que peut prendre la vitesse (ou une de ses dérivées) aux frontières :

```
[...]

boundaryField
{
    movingWall
    {
        type            fixedValue;
        value           uniform (1 0 0);
    }

    floor
    {

        type            fixedValue;
        value           uniform (0 0 0);

    }

    Inlet
    {
        type            cyclic;
    }

    Outlet
    {
        type            cyclic;
    }

    frontAndBack
    {
        type            empty;
    }
}
```

Listing 3 – Extrait du fichier *U* du dossier *0*. (3)

Les conditions aux limites se spécifient dans le dictionnaire boundaryField. Pour chaque frontière, il faut l'identifier par un nom qui correspond au nom donné lors de la création du maillage ; la nature de la condition aux limites et enfin toute valeur requise par la condition si nécessaire. Voici un template d'à quoi doit ressembler l'implémentation d'UNE condition aux limites :

```
nomFrontiere
{

    type            typeDeLaCondition;
    value           valeurDeLaCondition;

}
```

Listing 4 – Exemple d'implémentation d'un condition aux limites

Ainsi, si l'on reprend l'extrait de code qui précède l'exemple, on peut voir qu'en vertu d'un écoulement de couette plan nous avons imposé la vitesse aux parois supérieure *movingWall* et inférieure *floor* avec la condition *fixedValue*. La paroi supérieure étant mobile nous lui avons donné une vitesse de 1 mètre par seconde dans la direction x.

Les parois de gauche et et droite reçoivent quant à eux une condition *cyclic*, ou ce qui est équivalent, une condition périodique, c'est à dire que tout déplacement qui devrait traverser une paroi, se retrouve dans l'autre paroi.

Enfin, notre écoulement est en deux dimensions, mais **OpenFOAM** nous impose d'utiliser des maillages en trois dimensions. Pour parer cette difficulté, il existe la condition *empty* qui permet à **OpenFOAM** d'ignorer ces frontières et de créer un "faux deux dimensions".

Il n'y a en général rien de plus dans les fichiers du répertoires *0*. Dans un premier temps, comme nous ne disposons pas de notre propre solver, il faut se plier aux notations et aux fichiers imposés des solvers "par défaut". Ici, si l'on utilise le solver *icoFoam* nous devons aussi créer un autre fichier P correspondant à la pression cinétique (et non pression "classique" ! !).

2.1.2 Répertoire *constant*

Ce répertoire est en général assez simple et rapide à paramétrer. En l'absence de maillage généré, il ne contient que des fichiers textes, dont au moins un presque commun à tous les modèles : *transportProperties*. Il contient les principales constantes utilisées dans les équations de transports/conservations résolues numériquement par le solveur employé. Dans le cas où l'on utiliserait *icoFoam*, il est spécifié dans la documentation que seul le scalaire *nu* (viscosité cinématique) est à inscrire. De ce fait, le fichier *transportproperties* (en dehors de son header) sera tout simplement :

```
[...] (header)

nu              0.01;
```

Listing 5 – Fichier transportProperties

Il est à noter que *transportProperties* ne sera en général pas le seul fichier dictionnaire à créer dans les cas où on utilise des modèles thermodynamiques, ou bien turbulents (et dans cette situation il faudra écrire les fichiers correspondants).

En dehors de ces fichiers dictionnaires, il doit se trouver dans ce répertoire, une fois que le maillage a été généré, un sous répertoire *polyMesh*. Ce dossier contient toutes les informations du maillage (identifiant et position de chacune des mailles, des faces, points etc..). Étant généré automatiquement, et adapté à l'exploitation par les solvers, il n'est en général pas utile de le modifier sauf si l'on veut modifier le maillage à la main ce qui est périlleux ! Pour toute modification de maillage il est plutôt préférable de le faire avec l'éditeur de géométrie utilisé pour le projet considéré (par exemple *blockMesh*, *Salomé*, *GMSH* ...).

2.1.3 Répertoire *system*

C'est le répertoire le plus long à configurer. Il doit être composé d'au moins trois fichiers impérativement (et plus selon l'étude réalisée (par exemple il faudra un fichier de plus si l'on veut résoudre numériquement le problème en le parallélisant). Les trois fichiers obligatoires sont :

- *controlDict* : contient tous les paramètres de contrôle de la simulation (pas de temps, intervalle d'écriture des résultats ...).
- *fvSchemes* : contient toutes les informations sur les schémas numériques de discrétisation des opérateurs vectoriels (div, grad ...) utilisés.
- *fvSolution* : contient toutes les informations sur les solvers numériques utilisés pour les calculs directs (Gauss-Seidel, Gradient conjugué).

Dans notre cas, il va se rajouter un quatrième fichier, *blockMeshDict*. Ce dernier est un dictionnaire qui est nécessaire pour l'appel de la méthode *blockMesh*. La méthode *blockMesh* est une manière de générer un maillage. Il est pratique d'employer *blockMesh* lorsque le maillage est simple à concevoir. Dans le cas contraire, on préférera utiliser des mailleurs capables de générer plus simplement des géométries complexes, comme par exemple *Salomé* mais cela n'est pas nécessaire ici.

Tout fichier dictionnaire *controlDict* doit impérativement contenir le nom du solver utilisé, et une manière de paramétrer le temps : par exemple en donnant un instant initial, un pas de temps, et le nombre de pas souhaité. Il peux exister d'autres entrées, notamment sur le format de sortie des résultats.

```
[...] (header)

application     icoFoam;

startFrom       startTime;

startTime       0;

stopAt          endTime;

endTime         0.3;

deltaT          0.001;

writeControl    timeStep;

writeInterval   4;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;
```

```
writeCompression  off;

timeFormat        general;

timePrecision     6;

runTimeModifiable true;
```

Listing 6 – exemple de fichier *controlDict*

La spécification des paramètres du fichier *controlDict* se fait par l'inscription de mots clés qui sont à l'avance connus par **OpenFOAM**. Ici nous réalisons une simulation avec le solveur *icoFoam*, de l'instant initial à 0.3s, pour un pas de temps de 0.001s. D'autres informations sont précisées, mais elles ne sont pas nécessaires obligatoirement au fonctionnement de la résolution numérique.

Le fichier *fvSchemes* est lui aussi un dictionnaire, qui doit contenir une entrée pour une opération vectorielle réalisée par le modèle :

```
[...] (header)

ddtSchemes
{
    default          Euler;
}

gradSchemes
{
    default          Gauss linear;
    grad(p)          Gauss linear;
}

divSchemes
{
    default          none;
    div(phi,U)       Gauss linear;
}

laplacianSchemes
{
    default          Gauss linear orthogonal;
}

interpolationSchemes
{
    default          linear;
}

snGradSchemes
{
    default          orthogonal;
```

```
}
```

Listing 7 – exemple de fichier *fvSchemes*

En principe chaque opérateur se voit associer un schéma par défaut. Par exemple la discrétisation temporelle se fait par un schéma d'Euler par défaut. C'est à dire que si l'on rentre une opération de dérivée temporelle dans notre modèle, par défaut, le schéma employé sera celui d'Euler (le choix implicite ou explicite sera à faire lorsque l'on programmera DANS le solver, et non ici).

Il peut être judicieux de spécifier pour une opération particulière un schéma particulier si besoin est (par exemple, si après une étude à la main, on se rend compte que l'opérateur peut s'écrire sous la forme implicite d'une matrice symétrique, on a intérêt à employer des schémas optimisés pour cette configuration plutôt qu'un schéma général).

Le fichier dictionnaire *fvSolution* doit contenir autant d'entrées que de variables "à résoudre" sont employées dans le modèle utilisé :

```
[...] (header)

solvers
{
    p
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0.05;
    }

    pFinal
    {
        $p;
        relTol          0;
    }

    U
    {
        solver          smoothSolver;
        smoother        symGaussSeidel;
        tolerance       1e-05;
        relTol          0;
    }
}

PISO
{
    nCorrectors     2;
    nNonOrthogonalCorrectors 0;
    pRefCell        0;
    pRefValue       0;
}
```

Listing 8 – exemple de fichier *fvSolution*

Les entrées sont imposées par le modèle utilisé. Il faut là aussi réfléchir sur la meilleure sélection des solvers possible pour optimiser à la fois le temps de calcul et la qualité des résultats.

De plus, dans le cas où le modèle l'exige, il faut inscrire une méthode de projection (ce qui est en pratique presque toujours le cas lors d'un écoulement incompressible).

Enfin, le dernier fichier utilisé pour ce premier calcul numérique est le fichier *blockMeshDict*. Il est très important de savoir comment l'écrire car il permet de gagner énormément de temps chaque fois que l'on doit travailler avec une géométrie simple. Ce fichier est plus long que les autres alors il est nécessaire de l'étudier en le découpant.

```
[...] (header)

scale   0.1;

vertices
(
    (0 0 0)    # 0
    (5 0 0)    # 1
    (5 0.1 0)  # 2
    (0 0.1 0)  # 3
    (0 0 1)    # 4
    (5 0 1)    # 5
    (5 0.1 1)  # 6
    (0 0.1 1)  # 7
);

 [...]
```

Listing 9 – exemple de fichier *blockMeshDict*

La première étape consiste en la construction de la géométrie globale, c'est à dire du domaine qui contiendra chacune des mailles. Dans notre cas nous devons construire un pavé, donc huit points suffiront. Il est possible de construire la géométrie avec plus de points, mais l'ordre de création et de placement des points importe beaucoup, il peut être judicieux de commenter le nom des points et des groupes de la géométrie à laquelle ils appartiennent.

Ici nous construisons un pavé de dimensions $(x, y, z) = (0.5, 0.01, 0.1)$ (en m), il faut faire attention à l'entrée *scale* qui va se multiplier avec les coordonnées de tous les points du dictionnaire *vertices*.

Une fois cette étape réalisée, on procède au maillage :

```
[...]


blocks
(
    hex (0 1 2 3 4 5 6 7) (30 1 30) simpleGrading (1 1 1)
```

```
);

edges
(
);

  [...]
```

Listing 10 – exemple de fichier *blockMeshDict* (2)

Dans le dictionnaire *blocks*, on rajoute une entrée par région que nous voulons mailler. Il peut y avoir intérêt à découper le maillage par plusieurs régions afin d'y réaliser un raffinement ou un agencement des mailles plus adaptés à la géométrie locale (courbure, coin, O-grid ...) mais ici nous commençons par réaliser un maillage uniforme seulement. le mot clé *hex* indique que nous voulons mailler le domaine avec des hexaèdres. Il est nécessaire ensuite de spécifier le volume à mailler en indiquant les identifiants des points crées juste avant. L'entrée *(30 1 30)* indique que nous voulons découper la géométrie en 30 mailles selon les directions x et z, et une seule maille dans la direction y. Enfin, l'entrée *simpleGrading (1 1 1)* signifie que nous voulons un maillage uniforme. en pratique, la valeur N à la coordonnée i veut dire que la dernière maille dans la direction i doit être N fois plus grande que la première maille dans cette direction. Avec 1, cela correspond donc à un maillage uniforme.

Par défaut, chacun des points du domaine sont reliés par un segment droit. Si l'on veut que ce ne soit pas le cas, on peut spécifier dans la liste *edges* la nature des liaisons. Ici, notre maillage étant purement cartésien, la liste *edges* restera vide
Enfin, il reste à rentrer l'identification des frontières où s'appliqueront les conditions aux limites.

```
[...]


boundary
(
    movingWall
    {
        type wall;
        faces
        (
            (4 5 6 7)
        );
    }
    floor
    {
        type wall;
        faces
        (
            (0 3 2 1)
        );
    }

    Inlet
    {
        type patch;
        faces
```

```
        (
            (0 4 7 3)
        );
    }

    Outlet
    {
        type patch;
        faces
        (
            (1 2 6 5)
        );
    }

    frontAndBack
    {
        type empty;
        faces
        (
            (0 1 5 4)
            (3 7 6 2)
        );
    }
);
```

Listing 11 – exemple de fichier *blockMeshDict* (3)

En pratique, chaque entrée commence par le nom de la frontière, et il est OBLIGATOIRE de nommer la frontière de la même manière qu'elle est appelée dans les fichiers du répertoire **0** ! Ensuite, il faut indiquer le type de frontière dont il s'agit. De manière générale, trois grands types sont utilisés : *patch* qui correspond à une frontière générique (Inlet, Outlet, Dirichlet, Neumann, Robin...) ; *wall* et *empty* (à utiliser pour "forcer" des écoulements en deux dimensions). Enfin il faut indiquer dans la liste *faces* les faces où se définissent ces frontières en indiquant les points délimitant ces dernières.

Nous venons de finir de voir comment paramétrer un dossier projet. Nous sommes alors déjà en mesure de faire tourner des calculs et d'obtenir des résultats. Cependant, nous verrons par la suite que cela peut ne pas suffire. En effet, **OpenFOAM** propose de nombreux solvers par défaut, qui servent de template, mais il est possible que nous voulons faire usage d'un modèle qui n'est pas couvert par les solvers de base d'**OpenFOAM**. Il va alors falloir programmer nous même notre solver, souvent en partant du solver le plus proche. Le même constat peut se faire en ce qui concerne les conditions aux limites.

En attendant la section qui traite à proprement parler de programmation, nous allons montrer quelques résultats pour permettre de prendre en main le logiciel de visualisation, **ParaView**, et comment réaliser des simulations.

2.1.4 Première silumation

Une fois que les répertoires *0*, *constant*, *system* sont correctement paramétrés, on peut procéder au lancement du calcul numérique. Pour cela, à partir d'un terminal que nous plaçons dans le dossier qui contient ces trois répertoires, nous commençons par démarer **OpenFOAM** :

```
openfoam2206
```

Listing 12 – Instruction pour lancer des simulations.

Une fois **OpenFOAM** lancé, on va générer le maillage grâce au dictionnaire *blockMeshDict* que nous avons paramétré auparavant.

```
blockMesh
```

Listing 13 – Instruction pour lancer des simulations. (2)

Normalement, le dossier *polyMesh* a été crée dans le répertoire *constant*. Nous sommes maintenant prêts à faire appel au solver *icoFoam* :

```
icoFoam
```

Listing 14 – Instruction pour lancer des simulations. (3)

Si tout se passe bien, le listing de la simulation doit s'écrire dans le terminal. Une fois le calcul numérique terminé, et avant de lancer le logiciel **ParaView**, il est nécessaire de créer le fichier lisible par **ParaView** puisque par défaut, **OpenFOAM** ne renvoie pas de fichier au format *VTK*.

```
touch results.foam
```

Listing 15 – Instruction pour lancer des simulations. (4)

Il est à noter qu'il est possible de donner n'importe quel nom au fichier, pourvu que son extension soit *.foam*. Il faut alors maintenant lancer le logiciel de visualisation **ParaView** et charger le fichier *results.foam* ce qui donne cette vue :

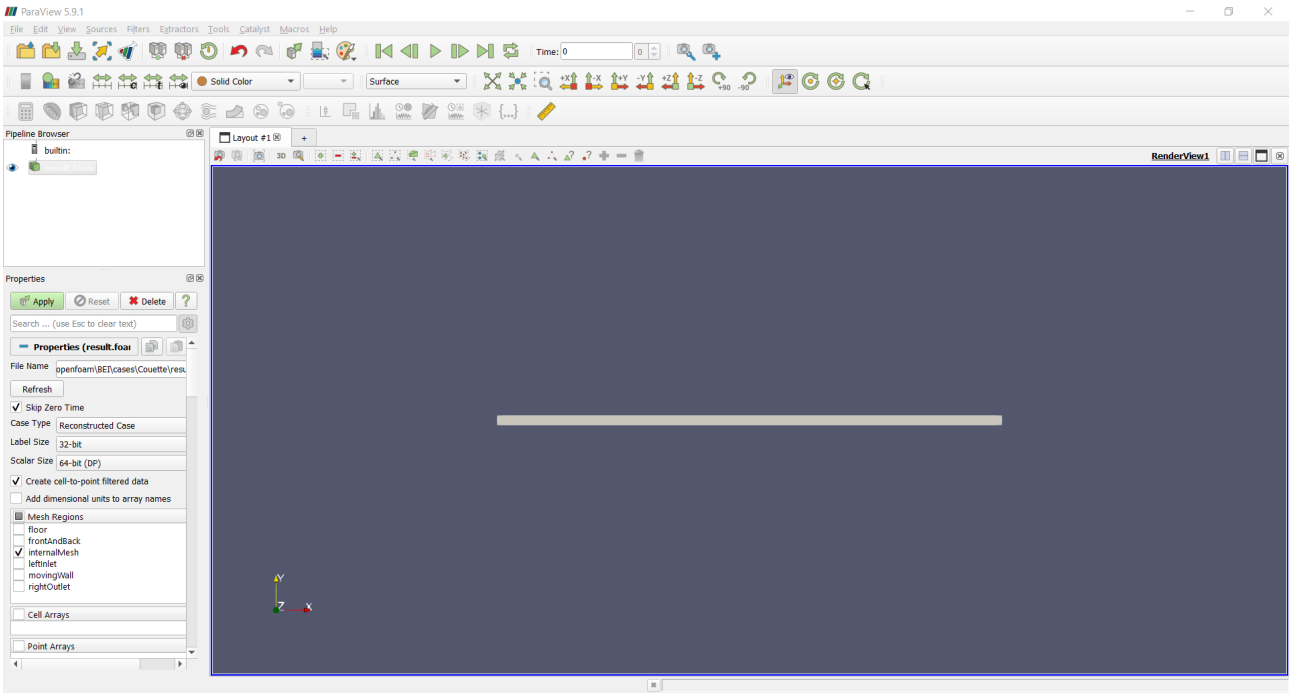

FIGURE 1 – Vue principale **Paraview**

Il est possible de choisir la grandeur à observer, par exemple la vitesse selon la direction x, et on peut faire jouer l'animation pour observer l'évolution de cette dernière.

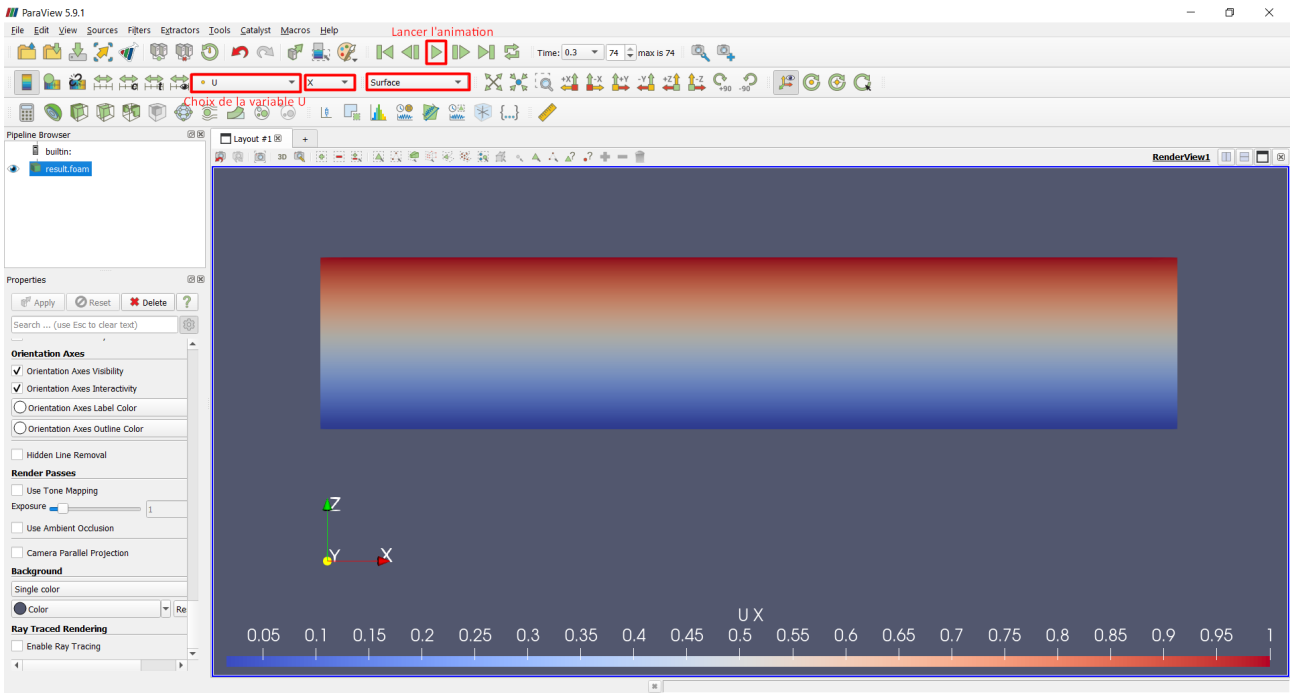

FIGURE 2 – Visualisation de l'écoulement de Couette

On peut aussi décider de n'observer qu'un seul profil de vitesse selon x vertical au centre, ce qui permet d'observer que le profil est bien linéaire.

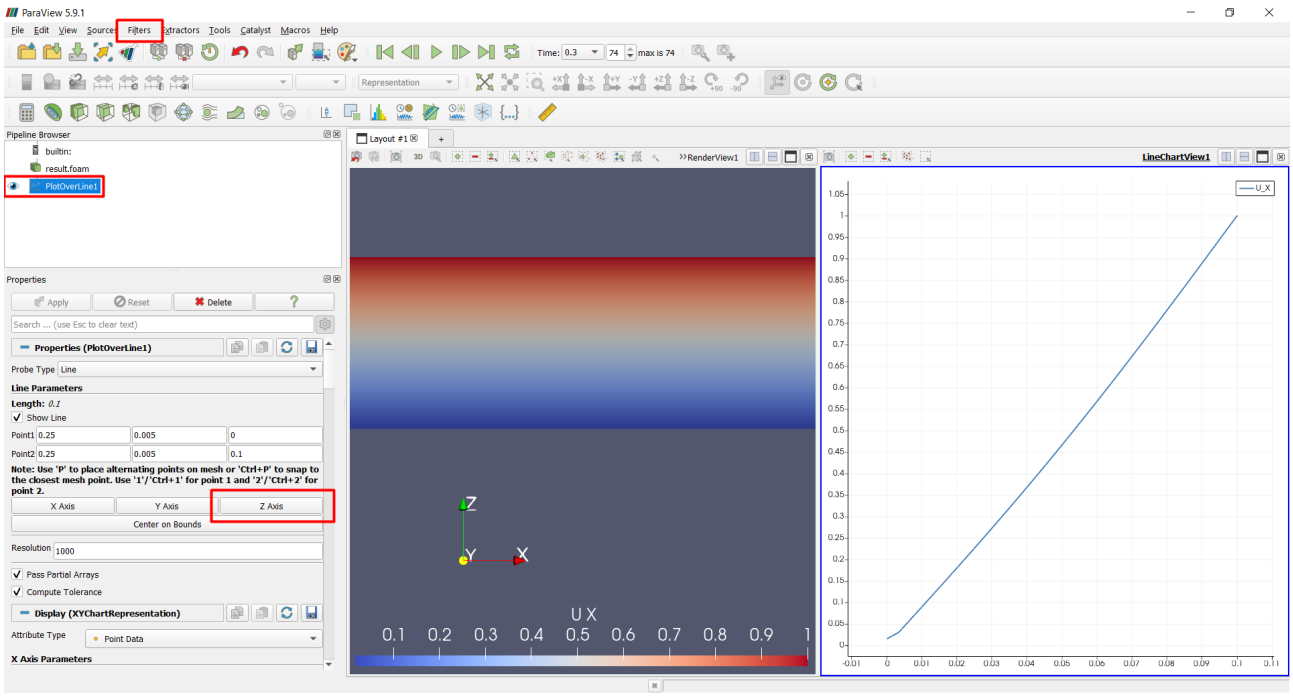

FIGURE 3 – Profil vertical

Enfin, il est possible de consigner les données dans un *spreadsheet* pour réaliser des études de post traitement avec le langage informatique *Python* ce qui peut s'avérer plus pratique si l'on dispose déjà de codes et ou de bibliothèques pour.

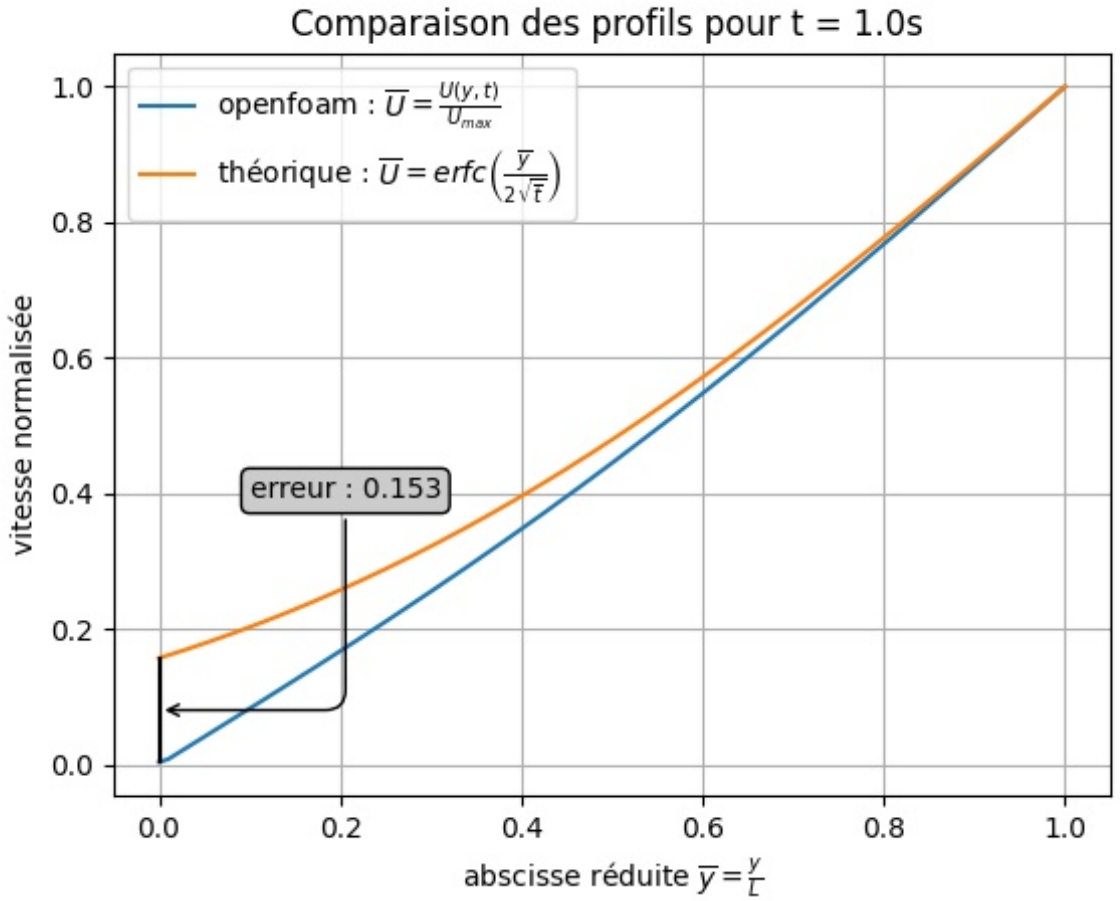

FIGURE 4 – Post traitement Python

3 Perfectionnement du projet

3.1 Édition d'un solver

Nous allons voir comment créer et éditer un solver **OpenFOAM**. De manière générale il faut partir d'un solver par défaut qui est proche du modèle que l'on souhaite résoudre numériquement. Dans notre cas nous souhaitons implémenter une équation de conservation de la matière :

$$\frac{\partial C}{\partial t} + \vec{U}\text{div}(C) = -\text{div}(\vec{j})$$

Avec $\vec{j} = \vec{j}_{diff} + \vec{j_g}$ où $\vec{j_g} = (1 - \frac{C}{C_{max}})(1-C)^{\alpha}\vec{v}_{stokes}C$ lié à la resuspension due à la poussée d'Archimède.

3.1.1 Copie d'un solver existant

Nous allons nous baser sur le solver icoFoam. Il se trouve dans le dossier de référence *$WM_PROJECT_DIR/applications/solvers/incompressible*. Il faudra le copier dans le dossier de votre choix (souvent on le copie dans le dossier *$WM_PROJECT_USER_DIR/*). Dans cet exemple, nous allons l'appeler *myIcoFoam*.

La première chose à faire une fois la copie réalisée est de renommer le fichier *icoFoam.C* en *myIcoFoam.C*. De plus, il faut créer un dossier *Make* (attention le nom du dossier doit commencer par un M et non un m!!!), qui va devoir contenir deux fichiers. Le premier s'appelle *files* et il devra avoir ce contenu :

```
myIcoFoam.C

EXE = $(FOAM_USER_APPBIN)/myIcoFoam
```

Listing 16 – fichier *files*

La première ligne correspond au fichier que le compilateur c++ spécifique à **OpenFOAM** doit compiler. La deuxième ligne contient le chemin où notre solver sera écrit, de plus cette ligne permet de donner un nom à notre solver. Ainsi, lorsque nous voudrons lancer une simulation avec ce solver, il faudra l'appeler myIcoFoam et non icoFoam.

Pour en comprendre plus sur la nécessité de ce fichier, du suivant, et de la structure du solver en général, il convient de consulter le **Programmer's Guide** de la documentation **OpenFOAM**. Le deuxième fichier s'appelle *options* et doit avoir ce contenu :

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude

EXE_LIBS = \
    -lfiniteVolume \
    -lmeshTools
```

Listing 17 – fichier *options*

Il s'agit simplement des imports nécessaires au fonctionnnement du code.

Nous en avons fini avec la partie de copiage du solver *icoFoam*, nous allons maintenant passer à la partie programmation.

3.1.2 Édition des fichiers sources

Commençons par le fichier *createFields.H*, il s'agit d'un fichier header. En nomenclature C++, c'est dans ce fichier que nous rangeons toutes nos instantiations des variables et classes. Nous allons commencer créer les constantes à renseigner dans le dictionnaire transportProperties :

```
IOdictionary transportProperties
(
    IOobject
    (
        "transportProperties",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ_IF_MODIFIED,
```

```
            IOobject::NO_WRITE
        )
);

dimensionedScalar nu("nu",transportProperties);
dimensionedScalar DC("DC",transportProperties);
dimensionedScalar Cmax("Cmax",transportProperties);
dimensionedVector rhog("rhog",transportProperties);
dimensionedVector vst("vst",transportProperties);
```

Listing 18 – fichier *createFields.H* (transportProperties)

L'explication de la nature des lignes se trouve aussi dans le **Programmer's guide**. Pour faire simple, disons que les première lignes permettent à **OpenFOAM** de comprendre qu'il va devoir chercher des constantes dans le fichier *transportProperties*, et que les lignes *dimensionedType name("name", transportProperties);* servent à indiquer qu'il faut chercher le type *name* au nom de *name* dans ce dictionnaire.

Toujours dans le même fichier *createFields.H* nous allons déclarer les champs scalaires P et C ainsi que vectoriel U. Pour ce faire, il faut écrire à la suite dans le fichier :

```
Info<< "Reading field p\n" << endl;
volScalarField p
(
    IOobject
    (
        "p",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

Info<<"Reading field C\n"<<endl;
volScalarField C
(
    IOobject
    (
        "C",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

volScalarField fC((1-(C/Cmax))*pow((1-C),3));
```

```
Info<< "Reading field U\n" << endl;
volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

#include "createPhi.H"

label pRefCell = 0;
scalar pRefValue = 0.0;
setRefCell(p, mesh.solutionDict().subDict("PISO"), pRefCell, pRefValue);
mesh.setFluxRequired(p.name());
```

Listing 19 – fichier *createFields.H* (déclaration des champs)

Là aussi nous ne rentrons pas trop dans les détails puisqu'ils ne sont pas nécessaires pour comprendre comment créer un nouveau champs étant donné qu'on peut s'inspirer des champs déjà précédents pour créer le nôtre. Il faut toutefois faire attention à la notation *volScalarField* et *volVectorField* (ou éventuellement *volTensorField*) selon la nature du champ que l'on veut créer. On peut noter, après la déclaration du champ de C l'implémentation du champ fC rendant l'écriture du terme $\vec{j}_{stokes}$ plus aisée.

ATENTION, une fois ces champs déclarés, il ne faut pas oublier d'écrire les fichiers correspondants dans le répertoire *0* !

Le fichier *createFields.H* est pour l'instant complet. Nous pouvons alors éditer le fichier *myIcoFoam.C*. On peut observer qu'il parait assez complexe de prime abord, mais il n'est pas si difficile de deviner le rôle de chaque ligne quand à la résolution numérique des équations de Navier-Stokes incompressibles. La seule modification à apporter se fait après l'étape de projection PISO, il suffit de rajouter entre la fin de cette étape et les appels de runTime le code suivant :

```
//Ajout equation concentration
fvScalarMatrix CEqn
(
    fvm::ddt(C)
    + fvm::div(phi,C)
    - fvm::laplacian(DC,C)
    + fvc::div(vst*C)
);
CEqn.solve();
```

Listing 20 – fichier *myIcoFoam.C* (ajout de l'équation de conservation de matière)

C'est la seule chose à rajouter. Une fois cela fait, il suffit de compiler le solver : pour cela il faut se placer avec le terminal dans le répertoire qui contient le dossier Make, et les fichiers .C et .H et rentrer la commande *Make*.
En pratique il est déjà possible de lancer un calcul, mais il va se poser un problème physique quant aux conditions aux limites que doit vérifier C sur les parois *movingWall* et *floor*. En effet, nous souhaitons que la densité totale $\vec{j}$ soit nulle pour que l'on puisse représenter ces frontières comme des murs. Or par défaut la condition *zeroGradient* d'**OpenFOAM** ne concerne que le flux diffusif et donc en l'état il n'y aura pas conservation de matière puis que le flux de stokes fera que la matière traversera le mur ! De ce fait, nous allons coder la condition aux qui impose une densité totale nulle, c'est à dire $\vec{j}_{diff} + \vec{j}_{stokes} = \vec{0}$

3.2 Écriture d'une condition aux limites personnalisée

Une étude à la main permet de montrer que pour que la densité totale soit nulle, il faut que la concentration de la face Cf de la frontière vérifie

$$Cf = Cc\left(1 + \Delta\frac{f(Cc)}{DC}\vec{v}_{stokes}\cdot\vec{n}\right)$$

Où Cc est la concentration au centre de la cellule associée à Cf, Δ est la distance entre le point d'évaluation de Cc (donc le centre de la cellule) et celui de Cf (donc le milieu de la face concernée). DC est le coefficient de diffusion associé au flux de diffusion, $\vec{n}$ est le vecteur normal de la face concernée.

Remarquons que si la vitesse de resuspension s'annule, nous obtenons une condition de *zeroGradient* plus "traditionnelle". Nous avons envisagé deux manières d'implémenter cette condition aux limites : la première consiste à utiliser la condition *codedFixedValue* qui permet de coder la condition dans le fichier lui-même. La deuxième consiste, à l'image de la création d'un solver, à créer nous même notre condition aux limites.

3.2.1 Condition aux limites construite dans le fichier *0/C*

La condition aux limites *codedFixedValue* permet à l'utilisateur de coder sa propre condition aux limites. Elle laisse à l'utilisateur tous les outils C++ pour permettre d'évaluer C aux frontières comme il le veut. Voici comment débute l'écriture de la condition :

```
    [...] (header)

type            codedFixedValue;
value           uniform 0.2;
name            codedTopBC;
code            #{

    // dictionnaires
    const dictionary& transportProperties = db().lookupObject
    <IOdictionary> ("transportProperties");
    dimensionedScalar DC("DC",transportProperties);
    dimensionedScalar Cmax("Cmax",transportProperties);
    dimensionedVector vst("vst",transportProperties);

    // Grandeurs "frontieres"
    scalarField& Cf = *this; // concentration

    // Grandeurs "centres"
    const tmp<vectorField>& tvn = patch().nf();
    const vectorField& vn = tvn();
```

```
    const tmp<scalarField>& tdelta = this->patch().deltaCoeffs();
    const scalarField& delta = tdelta();
    const tmp<scalarField>& Cc = patchInternalField(); // concentration

    [...]
```

Listing 21 – fichier *createFields.H* (déclaration des champs)

Avant d'implémenter la formule pour évaluer Cf, il faut importer les constantes et champs nécessaires. C'est ce qui est fait dans les premières lignes et cela est analogue au code écrit dans le fichier *createFields.H*.

Pour obtenir les coordonnées des faces où doit être évalué Cf, on peut simplement importer Cf en s'aidant du pointeur **this* puisque nous nous trouvons DANS le fichier C. *vn* fait référence au vecteur normal sortant $\vec{n}$. *delta* correspond à l'inverse de la distance Δ (cf documentation) (c'est confus ! Mais ce problème de notations ambiguës est pathologique au projet **OpenFOAM**). Enfin, les concentrations au centre des cellules s'obtiennent grâce à la méthode *patchInternalField*.

Une fois les champs déclarés, on peut maintenant écrire la condition aux limites :

```
    [...]

    forAll(Cf, faceID)
    {
        Cf[faceID] = Cc.ref()[faceID]*(1.+(vst.value()&vn[faceID]*(1-
        Cc.ref()[faceID]/Cmax.value())*(pow((1-Cc.ref()[faceID]),2))
        /(DC.value()*delta[faceID])));
    }
#};
```

Listing 22 – fichier *createFields.H* (déclaration des champs)

Ce qui correspond bien à la condition que nous voulons implémenter.

3.2.2 Condition aux limites construite dans une librairie

Au début, la condition aux limites était écrite comme étant de type *codedFixedValue*, c'est-à-dire que le code qui calcule la condition de flux massique nul à la frontière était dans le fichier C. Ainsi, à chaque lancement de la simulation, le code est compilé et une liste de dépendances est créée. Cette étape augmente le temps de calcul de la simulation et n'est pas la manière la plus appropriée pour mettre en œuvre la condition aux limites souhaitée. De plus, le fichier C contient beaucoup d'informations, ce qui le rend difficile à lire.

Pour créer une librairie (et donc une condition aux limites), il faut, à l'image de la programmation d'un solver, écrire un fichier *condition.C* ainsi que son header associé appelé code source et un dossier Make/. Ces fichiers sont importés via une méthode spécifique appelée *#include* qui signale au compilateur que le code contenu dans l'en-tête doit être inséré dans le code source.

Le fichier .H ne présente pas d'intérêt en terme de programmation en ceci qu'il est très indigeste et sert fondamentalement à indiquer à **OpenFOAM** comment lire les fichiers du répertoire *0*.
Le fichier .C est le code source où sont écrites les implémentations des méthodes et dans ce cas le code qui calcule la concentration à la surface d'une cellule.

L'un des problèmes rencontrés lors de l'implémentation du code source était un bogue qui empêchait la compilation de la bibliothèque à cause de la méthode *printMessage*. Pour l'instant, le moyen de contourner le problème était de commenter les lignes où cette méthode est utilisée, car elle n'affiche qu'un message et n'a aucune influence sur le calcul.

Ci-dessous, nous pouvons voir un l'extrait du code source utilisé pour la condition aux limites de flux nul, la méthode pour mettre à jour les valeurs calculées et un exemple où la méthode *printMessage* est commentée et donc ignorée par le compilateur.

```
    [...]

    void
Foam::
zeroMassFluxFvPatchScalarField::updateCoeffs()
{
    if (this->updated())
    {
        return;
    }

    if (false)
    {
        //printMessage("updateCoeffs zeroMassFlux");
    }
// dictionnaires
            const dictionary& transportProperties = db().lookupObject<IOdictionary> ("transpo
            dimensionedScalar DC("DC",dimViscosity,transportProperties);
            dimensionedScalar Cmax("Cmax",transportProperties);
            dimensionedVector vst("vst",transportProperties);

            // Grandeurs "frontieres"
            scalarField& Cf = *this; // concentration

            // Grandeurs "centres"
            const tmp<vectorField>& tvn = patch().nf();
            const vectorField& vn = tvn();

            const tmp<scalarField>& tdelta = this->patch().deltaCoeffs();
            const scalarField& delta = tdelta();
            const tmp<scalarField>& Cc = patchInternalField(); // concentration


            forAll(Cf, faceID)
            {
                Cf[faceID] = Cc.ref()[faceID]*(1.+
            (vst.value()&vn[faceID]*(1-Cc.ref()
                [faceID]/Cmax.value())*
                (pow((1-Cc.ref()[faceID]),2))/(DC.value()*delta[faceID])));
            }
```

```
    this->parent_bctype::updateCoeffs();
}
```

Listing 23 – fichier *zeroMassFluxFvPatchFieldTemplate.C* (méthode et code updateCoeffs() pour calculer la concentration à la surface d'une cellule)

Remarquons que le code est en fait exactement le même que pour la méthode précédente.
Enfin, nous avons à configurer le dossier *Make* : à l'instant du précédent dossier *Make* que nous avions eu à configurer, deux fichiers sont nécessaires : le fichier *files* :

```
zeroMassFluxFvPatchFieldTemplate.C

LIB = $(FOAM_USER_LIBBIN)/libZeroMassFlux
```

Listing 24 – fichier *files*

C'est presque identique au précédent fichier *files*. Il faut tout de même remarquer que la variable utilisée est le *FOAM_USER_APPBIN* au lieu du *FOAM_USER_LIBBIN*.

Le fichier *options* a lui pour écriture :

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude

LIB_LIBS = \
    -lfiniteVolume \
    -lmeshTools
```

Listing 25 – fichier *options*

Une fois les deux fichiers du répertoire *Make* bien écris, notre condition aux limites est prête à être utilisée.

3.3 Validation de la condition aux limites

La validation de cette condition aux limites est nécessaire pour la suite du projet. Un test simple est conçu pour valider cette condition ; un cas sera initialisé avec un champ de concentration uniforme, et la quantité suivie dans le temps est l'intégrale sur le domaine entier de la fraction volumique. Des tests sur la sensibilité par rapport au maillage ont été effectués, et nous n'observons pas de grande différences à ce niveau. Cependant, en fonction du pas de temps choisi, les résultats obtenus sont différents. La figure ci-dessous représente la variation de la masse en fonction du temps dans le domaine pour 3 pas de temps différents :

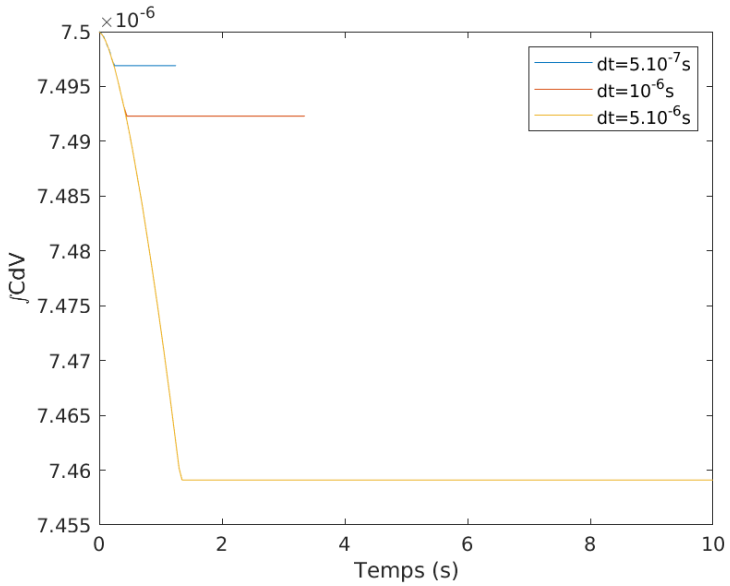

FIGURE 5 – Profils temporels de la quantité totale de matière

Nous observons que plus le pas de temps est petit, plus la conservation de la masse est obtenue rapidement et les pertes numériques sont moins importantes. La perte de masse s'arrête dès que le régime stationnaire est atteint ; ceci indique que le temps d'atteinte du régime stationnaire est différent pour chaque pas de temps utilisé.

De plus, l'observation des profils de concentration à l'état stationnaire montre que ces derniers sont différents pour chaque pas de temps.

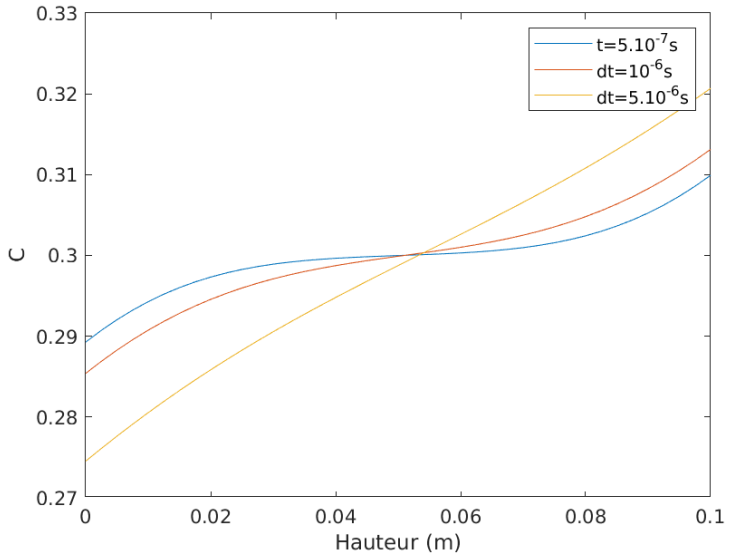

FIGURE 6 – Profils d'axe vertical de concentration à l'état stationnaire en fonction du pas de temps

On constate qu'à conditions initiales égales, avec un maillage identique, mais pour un pas de temps différent, que la répartition spatiale "finale" de concentration n'est pas la même.

En fait, si l'on s'intéresse au calcul du champ stationnaire (c'est à dire résoudre l'équation de conservation de matière, en annulant la dérivée temporelle), on se rend compte que l'on doit résoudre une équation différentielle scalaire.

C étant continue, il existe forcément au moins une solution, mais pour qu'elle soit unique, on doit imposer une condition de Dirichlet quelque part dans notre domaine. Ce n'est pas le cas dans notre étude alors la solution obtenue risque de ne pas avoir le sens physique attendu.

4 Listing complet du code utilisé

Le code présenté au dessus peut se consulter sur ce repository github.