Rapport BEI

Complément de codes : Simulation de la resuspension d'une émulsion en écoulement de Couette

Chady KERYAKOS
Kévin LEPETIT
Arthur NANGO
Thiago VARELLA

Février-Mars

Table des matières

Chapitre 1

Rapport BEI

1.1 Introduction et Objectif

Aucun code n'entre en jeu dans cette partie.

1.2 Présentation physique et mathématique du problème considéré

Aucun code n'entre en jeu dans cette partie.

1.3 Implémentation numérique du problème

1.3.1 Choix de l'outil et des versions

Nous allons utiliser comme outil numérique la banque de projets **OpenFOAM**. Pour tout tutoriel, ou information sur le fonctionnement basique d'**OpenFOAM**, merci de consulter la *notice OpenFOAM* jointe à ce rapport.
Notre version d'**OpenFOAM** utilisée est la version 2206. Nous allons prendre comme point de départ le solver *icoFoam*. Il permet de résoudre numériquement les écoulements incompressibles, turbulents, laminaires et isothermes.

Nous allons programmer à partir d'une copie du code source d'icoFoam notre propre solver adapté aux modèles décris précédemment.

1.3.2 Programmation des équations

Attention, cette partie contiendra principalement du code. Nous préférons écrire le code morceau par morceau plutôt que de tout envoyer en un coup. Cela permet de mieux expliquer notre démarche, ainsi que de mieux faire comprendre nos méthodes utilisées.

Pour savoir comment copier un solver, et programmer, il faut voir la notice.

Nous allons organiser notre solver comme dans la figure qui suit :

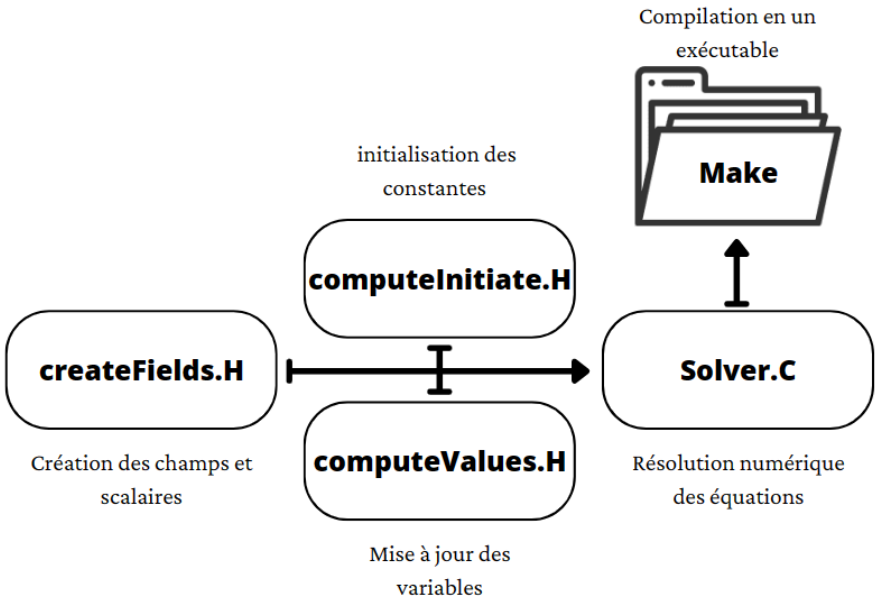

FIGURE 1.1 – Structure du solver

Par défaut, icoFoam résout les équations de Navier-Stokes :

$$\frac{DU}{Dt} = \nu \Delta U - \nabla p$$

Où p n'est pas la pression dynamique, mais la pression cinématique (divisée par la masse volumique).

```
[...] header
fvVectorMatrix UEqn
(
    fvm::ddt(U)
    + fvm::div(phi, U)
    - fvm::laplacian(nu, U)
);

if (piso.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));
}

[...]
```

Listing 1.1 – Extrait du code source icoFoam : équation initialement résolue.

Nous pouvons déjà ici rajouter le terme dû à la flottaison :

```
[...] header
fvVectorMatrix UEqn
(
    fvm::ddt(U)
```

```
    + fvm::div(phi, U)
    - fvm::laplacian(nu, U)
    - C*rhoi*g/rhoif
);

if (piso.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));
}

[...]
```

Listing 1.2 – Extrait du code source du solver personnalisé : ajout du terme de flottaison.

De plus, après cette modification, il va être nécessaire de créer l'équation de transport, que nous allons nommer *CEqn* :

```
[...]
fvScalarMatrix CEqn
(
    fvm::ddt(C)
    + fvm::div(phi,C)
    - fvm::laplacian(DC,C)
);
CEqn.solve();
[...]
```

Listing 1.3 – Extrait du code source du solver personnalisé : équation de transport.

Pour l'instant, il s'agit de l'équation d'avection-diffusion, mais nous ne voulons pas résoudre cette équation exactement. Il manque les termes j_{tot}, $f(C)$, Σ. Nous allons présenter comment les construire un à un.

Construction de $f(C)$

Par définition, $f(C) = (1 - C/C_m)(1 - C)^2$. $f(C)$ ne dépend que de C et de C_m. On va donc déclarer ces variables dans le fichier createFields.H :

```
[...]
dimensionedScalar Cmax("Cmax",transportProperties);
[...]
volScalarField C
(
    IOobject
    (
        "C",
        runTime.name(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
[...]
```

Listing 1.4 – Fichier createFields.H : Création du champ C, et du scalaire Cmax.

Cmax est une constante, qui est fixée dans le dictionnaire *transportProperties*, elle s'identifie au scalaire C_m.

Le champ C est initialisé, avec la commande *IOobject : :MUST_READ* : on indique au compilateur, qu'il va falloir chercher des informations sur C dans le dossier *0*.

Il faut créer le dictionnaire *transportProperties* avec l'extrait de code suivant :

```
IOdictionary transportProperties
(
    IOobject
    (
        "transportProperties",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ_IF_MODIFIED,
        IOobject::NO_WRITE
    )
);
[...]
```

Listing 1.5 – Fichier createFields.H : Création du dictionnaire *transportProperties*.

Enfin, on peut implémenter $f(C)$ en créant un nouveau champ dépendant de tout ce qu'on vient de créer :

```
[...]
volScalarField fC
(
    IOobject
    (
        "fC",
        runTime.name(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    ((1-(C/Cmax)) * pow((1-C),2))
);
[...]
```

Listing 1.6 – Fichier createFields.H : Création de fC (ce qui correspond à $f(C)$).

Pour attribuer une valeur à fC, on peut, au lieu de mettre l'argument "mesh" comme dernier argument, inscrire directement l'expression. **Attention**, fC ne sera pas automatiquement mise à jour, pour pallier à ce problème il va falloir, dans le fichier *computeValues.H*, mettre à jour fC :

```
fC = (1-(C/Cmax)) * pow((1-C),2);
[...]
```

Listing 1.7 – Fichier computeValues.H : mise à jour de fC (ce qui correspond à $f(C)$).

Construction de Σ

Le tenseur des contraintes Σ peut se décomposer en deux parties : Σ_f associée au fluide, et Σ_p associée aux particules. Commençons par décrire l'obtention numérique de Σ_f.

Dans sa formule mathématique, Σ_f a pour expression :

$$\Sigma_f = 2\eta_0 \mathbf{E} - p\mathbf{I} \tag{1.1}$$

Attention, dans le solver icoFoam, le terme de $-\nabla p$ est déjà utilisé et donc, si l'on veut éviter un maximum de chantier, on peut calculer Σ_f sans le terme de pression, ainsi, nous allons coder dans notre solver :

$$\Sigma_f = 2\eta_0 \mathbf{E} \tag{1.2}$$

Σ_f ne dépend que de η_0 et de $\mathbf{E}$ qui lui même dépend du gradient de la vitesse U. Commençons par η_0 et U :

```
[...]
dimensionedScalar eta0("eta0",transportProperties);
[...]
volVectorField U
(
    IOobject
    (
        "U",
        runTime.name(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
[...]
```

Listing 1.8 – Fichier createFields.H : déclaration de η_0 et du champ de vitesse U.

η_0 est une constante physique, que l'utilisateur doit renseigner dans le dictionnaire *transportProperties*. Le champ des vitesses est un champ de vecteurs (**attention**, il faut le déclarer avec la classe *volVectorField* et non *volScalarField*), qui de la même manière que pour le champ scalaire C, devra être crée par l'utilisateur dans le répertoire *0*.

La création de $\mathbf{E}$ est plus technique : même s'il est simple de déclarer la matrice, les opérations algébriques pour la déclarer ne sont pas natives du langage informatique *C++*. Pour cela il va falloir piocher dans les utilitaires déjà codés d'**OpenFOAM**. Dans le guide *Programmers guide*, le lecteur trouvera en page **P-26** plus d'informations sur la transposition de matrices. Pour le calcul de gradients, il faudra lire la page **P-42**.

Une fois les connaissances nécessaires acquises, la déclaration du tenseur $\mathbf{E}$ a pour code :

```
[...]
volTensorField gradU
(
    IOobject
    (
        "gradU",
        runTime.name(),
```

```
            mesh,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        (fvc::grad(U))
);

volTensorField E
(
    IOobject
    (
        "E",
        runTime.name(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    (gradU + gradU.T())
);
[...]
```

Listing 1.9 – Fichier createFields.H : déclaration du tenseur **E** et de ses dépendances.

D'une part nous construisons le gradient du champ des vitesses (il s'agit d'un champ de tenseurs, et non de vecteurs !). D'autre part nous créons le tenseur **E** avec son expression.

En indiquant "*NO_READ*" nous indiquons au compilateur que nous devrons mettre à jour les valeurs numérique de **E** et du champ de gradient des vitesses pendant le calcul, ce qui fait qu'il n'y a pas besoin de les lire dans le dossier *0* comme c'est le cas pour les champs U, p et C.

En indiquant "*AUTO_WRITE*" nous indiquons au compilateur d'écrire la valeur numérique des deux grandeurs dans les dossiers de résolution, au même titre que pour les champs U, p et C. Cela permettra de simplifier le post traitement.

Il ne reste plus qu'à mettre à jour les valeurs numérique du gradient, et du tenseur **E** dans le fichier computeValues.H :

```
[...]
gradU = fvc::grad(U);
E = gradU + gradU.T();
[...]
```

Listing 1.10 – Fichier computeValues.H : mise à jour du tenseur **E** et de ses dépendances.

Le tenseur des contraintes lié au fluide peut alors simplement être déclaré :

```
[...]
volTensorField sigmaF
(
    IOobject
    (
        "sigmaF",
```

```
            runTime.name(),
            mesh,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        (2.*eta0*E)
);
[...]
```

Listing 1.11 – Fichier createFields.H : déclaration du tenseur des contraintes.

Qu'il ne reste plus qu'à mettre à jour :

```
[...]
sigmaF = 2.*eta0*E;
[...]
```

Listing 1.12 – Fichier computeValues.H : mise à jour du tenseur des contraintes.

Nous en avons fini avec le tenseur des contraintes liées au fluide. Nous allons construire celui associé aux particules. Son expression mathématique est :

$$\Sigma_p = 2\eta_0\eta_p(C)\mathbf{E} - \eta_0\eta_N(C)\dot{\gamma}\mathbf{Q} \tag{1.3}$$

Pour se remémorer les expressions et significations des termes présents dans cette équation, il faut reconsulter les modèles décris précédemment.

Nous allons commencer par déclarer toutes les dépendances scalaires, à savoir : $\eta_p(C)$, $\eta_N(C)$ et $\dot{\gamma}$:

```
[...]
volScalarField etaS
(
    IOobject
    (
        "etaS",
        runTime.name(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    (pow(1.-C/Cmax,-2))
);

volScalarField etaN
(
    IOobject
    (
        "etaN",
        runTime.name(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
```

```
    (0.75 * pow(C/Cmax,2) * pow(1.-C/Cmax,-2))
);

volScalarField etaP
(
    IOobject
    (
        "etaP",
        runTime.name(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    (etaS - 1.)
);

volScalarField gammaShear
(
    IOobject
    (
        "gammaShear",
        runTime.name(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    (sqrt(2*(E && E)))
);
[...]
```

Listing 1.13 – Fichier createFields.H : déclaration des dépendances scalaire.

Pour être conforme aux notations de l'article de T.Dbouk, nous créons la viscosité η_S pour obtenir celle de η_P. La mise à jour se fait par :

```
[...]
etaN = 0.75 * pow(C/Cmax,2) * pow(1-C/Cmax,-2);
etaS = pow(1-C/Cmax,-2);
etaP = etaS - 1.;

gammaShear = sqrt(2*(E && E));
[...]
```

Listing 1.14 – Fichier computeValues.H : mise à jour des dépendances scalaire.

En ce qui concerne le tenseur $\mathbf{Q}$, il va aussi falloir construire ses dépendances. Nous commençons à considérer que l'écoulement n'est pas anisotropique, ce qui va faire que nous allons écrire $\mathbf{Q}$ comme la matrice identité proportionnelle à la moyenne de chacune de ses composantes :

```
[...]
```

```
volScalarField lambda2
(
    IOobject
    (
        "lambda2",
        runTime.name(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    (0.81 * (C/Cmax) + 0.66)
);

volScalarField lambda3
(
    IOobject
    (
        "lambda3",
        runTime.name(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    (-0.0088 * (C/Cmax) + 0.54)
);

volTensorField Q
(
    IOobject
    (
        "Q",
        runTime.name(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    (1.+lambda2 + lambda3)/3.*tensor(1.,0,0,0,1.,0,0,0,1.)
);
[...]
```

Listing 1.15 – Fichier createFields.H : déclaration du tenseur **Q** et de ses dépendances.

"*tensor(1.,0,0,0,1.,0,0,0,1.)*" dénote la matrice identité. Pour en savoir plus sur les outils algébriques propre aux tenseurs, il faut consulter la page **P-23/P-24** du guide *Programmers guide*. Comme on commence à en prendre l'habitude, voici comment mettre à jour **Q** :

```
[...]
lambda2 = 0.81 * (C/Cmax) + 0.66;
lambda3 = -0.0088 * (C/Cmax) + 0.54;

tensor I(1.,0,0,0,1.,0,0,0,1.);
Q = (1.+lambda2 + lambda3)/3.*I;
```

```
[...]
```

Listing 1.16 – Fichier computeValues.H : mise à jour du tenseur **Q** et de ses dépendances.

Nous venons maintenant de rassembler tous les éléments nécessaires pour construire le tenseur Σ_p, que ce soit pour la déclaration :

```
[...]
volTensorField sigmaP
(
    IOobject
    (
        "sigmaP",
        runTime.name(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    (2*eta0*etaP*E - eta0*etaN*gammaShear*Q)
);
[...]
```

Listing 1.17 – Fichier createFields.H : déclaration du tenseur des contraintes associé aux particules.

Ou que ce soit pour la mise à jour :

```
[...]
sigmaP = 2*eta0*etaP*E - eta0*etaN*gammaShear*Q;
[...]
```

Listing 1.18 – Fichier computeValues.H : mise à jour du tenseur des contraintes associé aux particules.

Construction de j_{tot}

Par définition, l'expression du flux total est :

$$j_{tot} = f(C)C\vec{v_{st}} + \frac{2a^2}{9\eta_0} f(C)\vec{\text{div}}(\mathbf{\Sigma_p}) \tag{1.4}$$

Que l'on peut décomposer en deux flux comme :

$$\begin{cases} j_g &= f(C)\vec{v_{st}}C \\ j &= \frac{2a^2}{9\eta_0} f(C)\vec{\text{div}}(\mathbf{\Sigma_p}) \end{cases} \tag{1.5}$$

Pour construire j_g il faut créer la vitesse $\vec{v_{st}}$. Nous allons la créer au centre des cellules par commodité, mais nous verrons dans la prochaine partie que cela peut jouer des mauvais tours si l'on a mal choisi les discretisations spatiales.

```
[...]
volScalarField rhoi
(
    IOobject
    (
```

```
            "rhoi",
            runTime.name(),
            mesh,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        mesh,
        dimensionedScalar(rhoip - rhoif)
);
volVectorField vst
(
        IOobject
        (
            "vst",
            runTime.name(),
            mesh,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        ((2./9.*a*a*rhoi/eta0)*g)
);
[...]
```

Listing 1.19 – Fichier createFields.H : mise à jour du tenseur des contraintes associé aux particules.

Cette fois-ci, comme la vitesse est constante, il ne faut pas la mettre à jour, à la place, il faut l'initier dans le fichier computeInitiate.H :

```
[...]
rhoi = rhoip - rhoif;
vst = (2./9.*a*a*rhoi/eta0)*g;
[...]
```

Listing 1.20 – Fichier computeInitiate.H : calcul de la vitesse.

Ce qui nous permet de construire le flux :

```
[...]



volVectorField jflux
(
        IOobject
        (
            "jflux",
            runTime.name(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
```

```
    mesh
);
[...]
```

Listing 1.21 – Fichier createFields.H : déclaration du flux.

Il est important de noter que le flux est déclaré en "*MUST_READ*", en effet, nous verrons que c'est une astuce qui va énormément aider dans le calcul des conditions aux limites.

```
[...]
volVectorField jflux
(
    IOobject
    (
        "jflux",
        runTime.name(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
[...]
```

Listing 1.22 – Fichier createFields.H : déclaration du flux.

Calcul de la densité

Au vu des hypothèses du modèle physique, il faut créer une densité qui tiens en compte le taux de particules dans un volume donné. Ainsi, la densité ρ aura pour expression

$$\rho = C\rho_f^i + (1 - C)\rho_p^i \tag{1.6}$$

La création se fait elle aussi dans le fichier createField.H :

```
[...]
dimensionedScalar rhoip("rhoip",transportProperties);
dimensionedScalar rhoif("rhoif",transportProperties);
[...]
volScalarField rhoi
(
    IOobject
    (
        "rhoi",
        runTime.name(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedScalar(rhoip - rhoif)
);
```

```
[...]
volScalarField rho
(
    IOobject
    (
        "rho",
        runTime.name(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    (C * rhoif + ( 1. - C ) * rhoif)
);
[...]
```

Listing 1.23 – Fichier createFields.H :Création de la densité.

```
[...]
rho = C * rhoif + ( 1 - C ) * rhoif;
[...]
```

Listing 1.24 – Fichier computeValues.H : mise à jour de la densité.

Programmation de l'équation

L'équation de quantité de mouvement s'écrit :

```
[...]
fvVectorMatrix UEqn
(
      rho/rhoif*(fvm::ddt(U)
    + fvm::div(phi, U))
    - fvc::div(sigmaF+sigmaP)/rhoif
    - C*rhoi*g/rhoif
);



if (piso.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));
}
[...]
```

Listing 1.25 – Fichier solver.C : équation de la quantité de mouvement.

L'équation de transport s'écrit :

```
[...]
fvScalarMatrix CEqn
(
```

```
      fvm::ddt(C)
    + fvm::div(phi,C)
    + fvc::div(jflux)
);
CEqn.solve();
[...]
```

Listing 1.26 – Fichier solver.C : équation de la quantité de mouvement.

1.3.3 Présentation détaillée des cas modélisés

Scripts du cas test utilisé

Le fichier *blockMeshDict* a pour contenu :

```
[...]
scale   0.1;

vertices
(
    (0 0 0) // 0
    (1 0 0) // 1
    (1 1 0) // 2
    (0 1 0) // 3
    (0 0 0.05) // 4
    (1 0 0.05) // 5
    (1 1 0.05) // 6
    (0 1 0.05) // 7
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (5 50 1) simpleGrading (1 1 1)
);

edges
(
);

boundary
(
    movingWall
    {
        type patch;
        faces
        (
            (7 6 2 3)
        );
    }
    floor
    {
        type patch;
```

```
        faces
        (
            (5 4 0 1)
        );
    }

    Inlet
    {
        type patch;
        faces
        (
            (0 4 7 3)
        );
    }

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

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

Listing 1.27 – Fichier *system/blockMeshDict*.

Le fichier *controlDict* a pour contenu :

```
[...]
application     sbmFoam;

startFrom       startTime;

startTime       0;

stopAt          endTime;

endTime         50.;

deltaT          0.001;
```

```
writeControl     adjustableRunTime;

writeInterval    1.;

purgeWrite       0;

writeFormat      ascii;

writePrecision   6;

writeCompression off;

timeFormat       general;

timePrecision    6;

runTimeModifiable true;

adjustTimeStep     yes;

maxCo              0.7;
[...]
```

Listing 1.28 – Fichier *system/controlDict*.

Le fichier *fvSchemes* a pour contenu :

```
[...]
ddtSchemes
{
    default          CrankNicolson 0.5;
}

gradSchemes
{
    default          Gauss linear;

}

divSchemes
{
    default           Gauss linear;
    div(phi,U)        Gauss vanLeer;
    div(SigmaP)       Gauss midPoint corrected;
    div(SigmaL)       Gauss midPoint corrected;
    div(phi,et)       Gauss vanLeer;
    div(jflux)        Gauss midPoint corrected;


}
```

```
laplacianSchemes
{
    default         Gauss linear corrected;

}

interpolationSchemes
{
    default         linear corrected;
}

snGradSchemes
{
    default         orthogonal;
}
[...]
```

Listing 1.29 – Fichier *system/fvSchemes*.

Le fichier *fvSolutions* a pour contenu :

```
[...]
solvers
{
    p
    {
        solver          smoothSolver;
        smoother        GaussSeidel;
        tolerance       1e-06;
        relTol          0.05;
    }



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

   pFinal
    {
        $p;
        relTol          0;
    }

    C
    {
        solver          smoothSolver;
            smoother        symGaussSeidel;
```

```
        tolerance       1e-05;
        relTol          0.05;
    }

}

PISO
{
    nCorrectors     2;
    nNonOrthogonalCorrectors 0;
    pRefCell        0;
    pRefValue       0;
}
[...]
```

Listing 1.30 – Fichier *system/fvSolution*.

Bibliographie