MASTER D'ASTRONOMIE ET D'ASTROPHYSIQUE DE L'OBSERVATOIRE DE PARIS

Rapport de stage :
Simulation des écoulements à la surface de Titan

Auteur
Pierre GOURBIN

Tuteur
Dr. Daniel CORDIER

EFFECTUÉ AU
GROUPE SPECTROMÉTRIE MOLÉCULAIRE ET ATMOSPHÉRIQUE, À L'UNIVERSITÉ DE REIMS

18 juin 2019

Table des matières

FIGURE *1.1 – Image SAR(RADAR) du site Sotra Patera, un des sites montrant des signes d'un potentiel cryovolcanisme*

Chapitre 1

Introduction

Avec un rayon moyen de 2574.73 km, Titan est le plus grand satellite naturel de Saturne est le deuxième plus grand satellite connu du système solaire, derrière Ganymède. C'est en 1655 que Christiaan Huygens découvrira Titan, néanmoins il ne pourra pas en dire beaucoup plus à ce sujet à ce moment.

On recommencera à s'y intéresser au début du 20^{e} siècle, et c'est en 1925 que James Hopwood Jeans montre qu'il est possible que Titan puisse retenir de manière durable une atmosphère dense, si celle-ci est composée de molécules plus lourdes que l'Hydrogène ou l'Hélium, comme l'Argon, le Néon, le diazote ou le méthane. La présence du méthane sera confirmée par des observations spectroscopiques faites par Gerard Kuiper durant l'hiver 1943-44. Kuiper précisera qu'une étude plus approfondie de cette atmosphère est compliquée au vu de la résolution angulaire de Titan ($\sim$ 1 arcseconde), voire impossible avec les moyens de l'époque.

La sonde *Pioneer 11*, en 1979, et la sonde *Voyager 1*, en 1980, ont permit d'en savoir plus sur Titan : *Pioneer 11* a permis de mieux contraindre le rayon de Titan, mais c'est vraiment avec *Voyager 1* que des avancées ont pu être faites dans l'étude de son atmosphère : on a notamment pu déterminer que celle-ci était composée en grande partie de diazote, avec seulement quelques pourcent de méthane. L'analyse des résultats ont pu également permettre de déduire une pression au sol de 1.6 bar et une température de 93 K, conditions notamment propices à la présence de certains hydrocarbures sous formes liquides. De nouvelles questions se poseront au fil des découvertes, notamment concernant le temps de vie du méthane dans l'atmosphère, estimé à quelques millions d'années, ce qui contredit sa présence dans l'atmosphère de nos jours, à moins d'avoir une source de méthane. L'une des possibilités est la présence d'océans de méthane qui permettraient un renouvellement du méthane dans l'atmosphère par évaporation. On commence donc à chercher des océans de méthane ou d'éthane sur la surface de Titan. Si MUHLEMAN et al. 1990 rejettent la possibilité d'un océan global, CAMPBELL et al. 2003 font une observation qui pourrait laisser entendre l'existence d'une étendue liquide.

L'étude de Titan restait compliquée malgré tout, notamment à cause de son atmosphère, opaque à une large gamme du spectre. La mission Cassini-Huygens, qui avait pour but une étude plus approfondie de Saturne et de ses satellites, en particulier Titan, comportait un orbiteur (Cassini) et un atterrisseur devant se poser sur Titan, ce qui permettrait de passer outre cette atmosphère opaque et d'examiner plus en détail la surface de Titan, et si possible confirmer la présence de lacs d'éthane ou de méthane. Sur ce dernier point, l'atterrisseur n'a malheureusement pas pu être d'une grande aide, atterrissant dans une zone dépourvue de lacs. Cependant, la présence de galets, indiquant des écoulements passés, et le sol meuble faisant penser à du sable humide laissait ouvert la possibilité de liquide présent en surface. L'orbiteur a continué sa mission, et a pu prendre des images d'étendues faisant penser à des lacs ou des mers, mais a également des structures qui pourraient être interprétées comme des possibles flots de cryolave, des zones ressemblant à des coulées solidifiées par exemple (voir 1.1). Les cryolaves sont des mixtures, en général composées d'eau, d'ammoniac ou de méthane (pour ne citer qu'eux), et qui se comportent, sur des astres froids (comme Titan par exemple), à la manière de la lave sur Terre. On appelle cette activité similaire à du volcanisme, du cryovolcanisme.

La présence de cryovolcanisme sur Titan ne serait pas anodine : il pourrait être, entre autres, à l'origine du réapprovisionnement en méthane de l'at-

mosphère. Cependant, sa présence est encore loin d'être confirmée, la seule "preuve" que nous possédions étant ses morphologies faisant penser à des flots de cryolave. Afin de se faire une meilleure idée de ce qu'il en est, une des possibilités est de simuler numériquement un écoulement de cryolave, et vérifier la forme et l'extension de l'écoulement, puis de le comparer aux observations de ces supposés flots gelés sur Titan. C'est ce que je tente d'accomplir pendant mon stage.

Modéliser la cryolave n'est pas une chose aisée, surtout dans le cas de Titan, où l'on ne dispose que de très peu d'informations sur la mixture étudiée. KARGEL 1994 avait déjà étudié différents types de cryolaves, et leur applications éventuelles sur différents satellites, cependant à l'époque nous n'avions aucune idée d'un éventuel cryovolcanisme à la surface de Titan, sa surface étant dissimulée sous son atmosphère opaque. À des fins pratiques, nous travaillerons plus en détail sur des mélanges eau-ammoniac, qui ont été plus étudiées que les autres types de cryolaves. Même ainsi, plusieurs questions se posent : quelle est la fraction d'ammoniac ? La cryolave est-elle purement liquide, ou est-elle en partie solide ? Y-a-t'il formation de cristaux dans la mixture ? Toutes ces questions peuvent influencer les propriétés rhéologiques et thermodynamiques de la cryolave de manière non négligeable, ce qui influera l'écoulement. De ce fait, la création d'un modèle pour la cryolave, et la simulation de l'écoulement de cette cryolave sur Titan, est quelque chose qui n'a pas encore été fait.

Plusieurs techniques existent pour modéliser et simuler la mécanique des fluides ; entre autres, on peut citer la technique du Smooth Particle Hydrodynamics (SPH) (MONAGHANN 1992), ou bien celle des automates cellulaires. Pour ma part, je vais utiliser le logiciel open source OpenFOAM, un outil de modélisation de mécanique des fluides qui possèdent de nombreuses fonctionnalités et est libre de droit.

Dans une première partie, j'expliquerai les bases du traitement numérique de la mécanique des fluides, en parlant notamment de la méthode des volumes finis, et je décrirai plus en détail le fonctionnement d'OpenFOAM. Puis, dans une deuxième partie, je décrirai plus en détail ma démarche pour créer la simulation d'écoulement de cryolave que je souhaite obtenir, en parlant plus en détail de la prise en main d'OpenFOAM, des modifications et ajouts que j'y ai apportés, mais également de la physique du problème, notamment des caractéristiques propres à Titan et à la cryolave, et qui sont nécessaires pour une simulation réussie.

Chapitre 2

Simulation numérique de la mécanique des fluides

2.1 Computational Fluid Dynamics ou CFD

La mécanique des fluides numériques (MFN, ou donc CFD), est une branche de la mécanique des fluides consistant à la résolution des équations régissant le ou les fluides pris en compte par des moyens numériques. La base de ce domaine repose sur la résolution des équations de Navier-Stokes. Ces équations, sous leur forme complète ou simplifiée, permettent de décrire un large éventail de situations impliquant la mécanique des fluides. Utiliser la simulation numérique présente plusieurs avantages :

— Il n'y a pas le problème de limitations des échelles spatiales et temporelles (possibilité de simuler le mouvement d'une imposante quantité de fluide sur des durées très longues).
— Il n'y a pas non plus le problème de devoir simuler des conditions physiques extrêmes (température, pression, etc)
— L'extraction et l'analyse de données est bien plus simple à faire depuis une simulation numérique que depuis une expérimentation.

Cependant, cela amène aussi son lot d'inconvénients, le premier étant que nous aurons dans tous les cas une approximation : une discrétisation est nécessaire afin de traiter informatiquement un problème. De plus, résoudre les équations de Navier-Stokes n'a rien d'une chose facile. La solution sera une approximation, et l'objectif de la simulation est de fournir l'approximation la plus précise. L'autre problème qui découle du premier est que le temps de la simulation dépendra de cette précision : Avoir un résultat précis implique d'utiliser des simulations gourmandes en ressources. On peut notamment parler des DNS (Direct Numerical Simulation), qui consistent à résoudre numériquement les équation de Navier-Stokes sans utiliser de modèle pour les turbulences. Cela implique de simuler les turbulences sur toutes leurs échelles spatiales et temporelles. Une telle simulation devient vite inutilisable, et est dans tous les

cas extrêmement gourmande en ressources.

2.1.1 Méthode des Volumes Finis ou FVM (Finite Volume Method)

La Méthode des Volumes Finis est la méthode de discrétisation employée par OpenFOAM afin de résoudre numériquement les équations différentielles.

Supposons Ω le domaine dans $\mathbb{R}^3$ sur lequel nous travaillons, et considérons une variable conservée intensive ϕ (par exemple, la densité ρ ou la vitesse $\vec{v}$). L'équation de conservation est la suivante :

$$\underbrace{\frac{\partial}{\partial t}\int_\Omega \rho\phi \mathrm{d}\Omega}_{\text{terme temporel}} + \underbrace{\int_S \rho\phi\vec{v}\cdot\vec{n}\mathrm{dS}}_{\text{terme convectif}} = \underbrace{\int_S \Gamma\vec{\nabla}\phi\cdot\vec{n}\mathrm{dS}}_{\text{terme diffusif}} + \underbrace{\int_\Omega q_\phi \mathrm{d}\Omega}_{\text{source/puit}} \quad , \tag{2.1}$$

où ρ est la densité dans Ω, $\vec{v}$ la vitesse, $\vec{n}$ le vecteur normal à la surface S délimitant Ω, Γ la diffusivité et q_ϕ la source (ou le puit) de ϕ. Pour le moment, je vais simplement traiter un cas stationnaire, i.e. indépendant du temps. L'équation ci-dessus devient alors :

$$\int_S \rho\phi\vec{v}\cdot\vec{n}\mathrm{dS} = \int_S \Gamma\vec{\nabla}\phi\cdot\vec{n}\mathrm{dS} + \int_\Omega q_\phi \mathrm{d}\Omega \tag{2.2}$$

Le principe de la FVM est de diviser le domaine de travail en un nombre finis de petits volumes de contrôles, que nous appellerons par la suite cellules. La particularité de la FVM est que les noeuds de calculs, i.e. les points où seront résolues les équations de conservation, sont situées dans les volumes (généralement au centre), et les conditions aux limites de chaque cellule seront définies.

Nous considérons pour l'instant un volume Ω cartésien, de même pour les cellules. Toutes les configurations que j'ai utilisées étaient dans un format cartésien, je n'aborderai donc pas les cas plus complexes. Dans cette configuration, chaque cellule a donc 4 faces en 2D et 6 faces en 3D. Je parlerai surtout du cas en 2D, étant donné qu'il peut simplement être considéré comme un cas en 3D où toutes les variables sont indépendantes de la troisième dimension ; le passage de l'un à l'autre se fait alors simplement. Dans cette configuration, le flux net à travers les limites d'une cellule sera alors la somme des intégrales au travers des 4 faces (ou 6 en 3D). Cela implique que pour garder la conservation sur tout le domaine, chaque face doit uniquement lier deux cellules (une de chaque côté) (fig.2.1), et ne doit pas dépasser sur d'autres cellules (fig.2.2).

Ce principe étant respecté, le flux net dans une cellule s'écrit donc ainsi :

$$\int_S f\mathrm{d}S = \sum_k \int_{S_k} f\mathrm{d}S \quad , \tag{2.3}$$

avec f la composante du flux (convectif ou diffusif dans notre cas) normale à la surface considérée, et k un indice variant pour chaque face. Pour les calculs

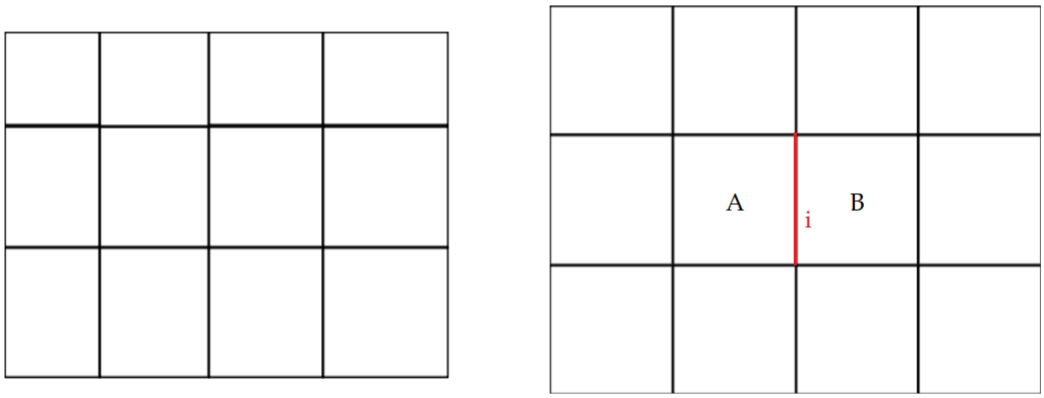

FIGURE 2.1 – *Deux configurations de cellules pouvant être utilisées pour la VOF, i.e. respectant la règle de non-dépassement*

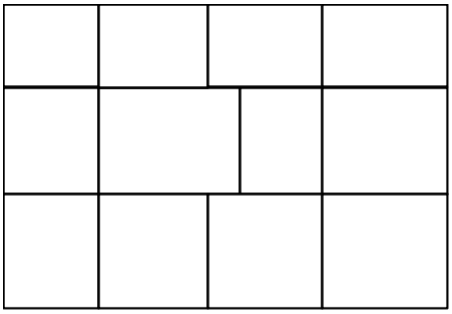

FIGURE 2.2 – *Configuration de cellules ne respectant pas la règle de non-dépassement, amenant ainsi un risque de perte de la conservation*

suivants, nous prendrons comme exemple la face i représentée sur la figure 2.1 séparant les cellules A et B. L'approximation la plus simple, et qui sera utilisée dans OpenFoam, et l'approximation du point-milieu : on approxime l'intégrande par la moyenne sur toute la surface, puis on approxime l'intégrale par le produit de l'intégrande sur la face considérée par l'aire de cette face. On aura donc :

$$\int_{V_A} \alpha \mathrm{d}V = \overline{\alpha} V_A \quad , \tag{2.4}$$

$$\int_{S_i} \vec{\beta} \cdot \mathrm{d}\vec{S} = \overline{\vec{\beta}} \cdot \vec{S_i} \quad , \tag{2.5}$$

avec α un scalaire générique, V_A le volume de la cellule A, β un vecteur générique et S_i la surface de la face i.

2.1.2 Discrétisation temporelle

La discrétisation temporelle peut se faire de plusieurs manières. Je vais ici en présenter deux : la méthode d'Euler et la méthode de Crank-Nicolson.

Méthode d'Euler

La méthode d'Euler est l'une des plus connues et des plus simples pour résoudre numériquement des équations différentielles. Le principe est simple : posons l'équation différentielle suivante :

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(t, y) \qquad y(t_0) = y_0 \tag{2.6}$$

On considère ensuite différents pas de temps $(t_0, t_1, t_2, ...)$, et que pour tout t_n, $y(t_n) = y_n$ Supposons que l'on connaît y_n, et que l'on souhaite avoir y_{n+1}. La méthode d'Euler consiste à approximer au premier ordre l'équation 2.6 de la manière suivante :

$$y_{n+1} = y_n + (t_{n+1} - t_n) f(t_n, y_n) \tag{2.7}$$

Méthode de Crank-Nicolson

La méthode de Crank-Nicolson est basée sur la règle des trapèzes, expliquée ici. Nous partons de nouveau de l'équation 2.6.

On suppose que l'on veut connaître la solution au pas de temps t_{n+1}, sachant que l'on connaît le pas de temps t_n. On va avoir :

$$y\left(t_{n+1}\right) - y\left(t_n\right) = \int_{t_n}^{t_{n+1}} f(t, y(t))\mathrm{d}t \tag{2.8}$$

La règle des trapèzes consiste à faire l'approximation suivante :

$$\int_{t_n}^{t_{n+1}} f(t, y(t))\mathrm{d}t \approx \frac{1}{2}\Delta t\Big(f(t_n, y(t_n)) + f(t_{n+1}, y(t_{n+1}))\Big) \tag{2.9}$$

En posant $y_n = y(t_n)$ et $y_{n+1} = y(t_{n+1})$, on obtient :

$$y_{n+1} = y_n + \frac{1}{2}\Delta t\Big(f(t_n, y_n) + f(t_{n+1}, y_{n+1})\Big) \tag{2.10}$$

L'appellation "méthode de Crank-Nicolson" réfère en général à l'application directe de la règle des trapèzes pour la discrétisation du temps dans une équation différentielle partielle.

Nombre de Courant

Dans les deux cas de discrétisation temporelle cités ci-dessus, le pas de temps peut être fixe ou variable, et la précision de ce résultat dépendra de ce pas de temps. Afin d'avoir des résultats à peu près fiable, il peut être judicieux de vérifier la valeur du nombre de Courant. Il s'agit d'un nombre sans dimension qui se pose ainsi :

$$Co = U * \frac{\delta t}{\delta x} \quad , \tag{2.11}$$

avec U la vitesse du fluide considéré, δx la taille d'une cellule et δt le pas de temps.

Le nombre de Courant sera supérieur à 1 si, dans un intervalle de temps δt, il parcourt une distance supérieure à δx, donc parcourt plus d'une cellule. Ce cas peut mener à des imprécisions, voire des divergences, notamment si la trajectoire n'est pas rectiligne, parce qu'alors la direction du flux va changer d'une cellule à l'autre. Afin d'éviter ce problème, il faudra faire attention à maintenir le nombre de Courant inférieur à 1. La grille à des cellules de taille fixe, et la vitesse est calculée lors de la simulation à chaque pas de temps, il convient donc d'adapter le pas de temps en conséquence. Deux solutions s'offrent alors à nous :

— Nous pouvons choisir de fixer un pas de temps très petit de base, afin d'être sûr que le nombre de Courant ne dépasse pas 1. On se base sur la vitesse aux conditions initiales, et on peut éventuellement recalculer le nombre de Courant après la simulation, selon la vitesse maximale rencontrée, et recommencer la simulation avec un pas de temps plus petit si nécessaire. C'est la méthode la plus simple, qui se fait au prix d'un temps de simulation potentiellement accru dû au petit pas de temps.

— Une autre solution, qui est celle que nous allons utiliser, est d'adapter le pas de temps à chaque itération temporelle en recalculant le nombre de Courant avec les conditions du pas de temps actuel. Bien sûr, il serait bien trop laborieux d'arrêter la simulation à chaque pas de temps pour faire le changement manuellement. OpenFOAM, le logiciel que nous utilisons pour nos simulations, et dont j'explique le fonctionnement dans la section suivante, dispose d'une option qui recalcule le nombre de Courant à chaque itération, et qui prendra le pas de temps maximum satisfaisant les conditions rentrées par l'utilisateur, entre autre une valeur maximale du nombre de Courant (en général 1, mais certaines simulations, en fonction des options de résolution choisies, peuvent nécessiter un nombre de Courant plus petit), et éventuellement un pas de temps minimal.

2.2 Principe d'OpenFOAM

OpenFOAM (Open Field Operation And Manipulation) [1]est un logiciel open source qui a pour premier usage de traiter des problèmes d'hydrodynamiques.

Il s'agit en premier lieu d'une "boîte à outil" de résolution des équations de Navier-Stokes. C'est une imposante librairie en C++, comportant de nombreux outils, fonctions et modèles, pouvant être appelés via des applications, et qui a pour avantage non négligeable d'être relativement facile à modifier. Entre autres, les utilisateurs peuvent créer et modifier certaines parties du code déjà existantes sans pour autant recompiler OpenFOAM, ce qui se fait grâce à des fonctions et de compilateurs spécifiques à OpenFOAM. Le code d'OpenFOAM, profitant des spécificités du C++ et du langage orienté objet, est complètement structurée en classes, ce qui permet dans une certaine mesure de faciliter grandement la modification et la création de solveurs.

2.2.1 Structure du code

Bien qu'OpenFOAM soit aussi modulable qu'on le souhaite, certaines parties sont suffisamment basiques et générales à l'ensemble du code pour que l'on n'ait pas besoin d'y toucher, comme par exemple la discrétisation des opérateurs différentiels, ou bien encore l'itération du temps. Les parties que l'utilisateur va être le plus amené à modifier sont :

- **Les modèles :** ils définissent les champs, constantes et fonctions associées de la plupart des modèles physiques. Ils sont utilisés sous la forme de librairies appelées lors de la compilation des solveurs.
- **Les solveurs :** on les utilise à la manière d'une application. Ils contiennent les champs à calculer ainsi que les équations associées à résoudre (équations de Navier-Stokes, équation de la chaleur...).
- **Les problèmes, ou les cas à traiter :** ils comportent le maillage, les différents paramètres numériques (résolution des résultats, instant de départ/d'arrivée, pas de temps, intervalle d'écriture, etc...) et les conditions initiales et limites.

Bien que le code soit en C++, la structure et toutes les classes définies sont telles que hormis pour la modification des modèles, il n'y a pas besoin de connaissances très approfondies en C++ pour le modifier.

Dans toute la suite de ce rapport, je référerai à ces différentes parties par ces noms. Il est important de noter, afin de bien comprendre comment fonctionne OpenFOAM, qu'il n'y a pas d'interface pour utiliser cet outil : chaque modèle, solveur ou cas à traiter est sous la forme d'un répertoire, contenant différents fichiers et sous-répertoires. Par exemple, le cas à traiter "damBreak" réfère à l'ensemble des fichiers présents dans le répertoire nommé pour des aspects pratiques "damBreak", ainsi que dans ses sous-répertoires, et qui décrivent les aspects numériques et physiques du problème considéré. De même, le solveur "interFoam" réfère à l'ensemble des fichiers contenus dans le répertoire nommé

1. voir https ://www.openfoam.com

(toujours pour des aspects pratiques) "interFoam", et qui, lorsqu'on les compile avec la commande `wmake` d'OpenFOAM, crééent l'application interFoam.

Les modèles :

Les modèles sont en général sous la forme d'une classe, et sont composés de deux fichiers, un déclarant les objets et les fonctions membres de la classe (exemple : twoPhaseMixture.H) et un les définissant (twoPhaseMixture.C). Les méthodes de ces classes sont pour la plupart faites pour retourner et/ou lire les attributs de l'objet. Pour les classes plus complexes, il y aura également des méthodes pour calculer certains paramètres, tels que le nombre de Reynolds, ou le nombre de Prandtl. Afin d'exploiter au maximum la Programmation Orientée Objet (POO), beaucoup de modèles, notamment les plus spécifiques, utilisent des modèles déjà existants et plus généraux en tant que classe mère, et rajoutent les attributs et méthodes nécessaires. Par exemple, le modèle twoPhaseMixture est un modèle pour les cas traitant de deux phases (liquide et gazeuse par exemple). Cependant, elle ne contient en soi que deux variables par phase : le nom de la phase, et la fraction volumique relative des deux phases, égale à 0 pour une phase, 1 pour une autre, et entre 0 et 1 à l'interface (voir partie 3.1, les explications sur la méthode VOF). Cette classe est l'une des plus basiques, et elle est utilisée par exemple par incompressibleTwoPhaseMixture, qui rajoute à ses attributs pour chaque phase : la densité, un champ de vitesse, un de flux, un de viscosité, et deux modèles de viscosité (un pour chaque phase).

Cette structure en arborescence a pour avantage l'implémentation facile de nouveaux modèles spécifiques : il suffit d'aller piocher dans les modèles plus généraux et de rajouter le nécessaire. Cependant, cela peut devenir un inconvénient lorsqu'on souhaite modifier un modèle spécifique déjà existant, notamment si ce modèle est déjà lui-même assez complexe. Un des exemples est le modèle twoPhaseMixtureThermo, qui en plus d'être une classe fille de trois modèles, possède en attribut deux objets issus d'un quatrième modèle. Le problème de cette surdépendance est qu'il devient compliqué de rajouter ou de changer quoi que ce soit sans une connaissance approfondie de ce que fait chaque modèle, sans parler du fait que chacun de ces modèles héritent encore d'autres modèles.

Une fois un modèle créé ou modifié, il faut le compiler afin qu'il puisse être utilisé par les solveurs. Afin d'éviter de devoir recompiler tout OpenFOAM pour prendre en compte un nouveau modèle, ce qui, en plus de prendre un certain temps (environ 6h pour les ordinateurs mis à disposition), est impossible pour quelqu'un n'ayant pas les droits d'administrateur, il existe une application créée spécifiquement pour compiler une librairie ou un solveur d'OpenFOAM individuellement. Elle porte le nom de wmake, et s'utilise en appelant dans le terminal :

```
wmake -libso [rpertoire contenant le ou les nouveaux modle(s)]
```

ou plus simplement `wmake` si le répertoire courant et celui contenant le modèle.

Cette application utilise le contenu d'un répertoire Allwmake, servant pour plusieurs modèles en général, et contenant 2 fichiers :

— Un fichier "files", comprenant les noms des fichiers à compiler, ainsi que le nom de la ou des librairie(s) à créer et l'endroit ou la/les stocker
— Un fichier "options", qui liste les chemins vers les répertoires contenant les modèles utilisés, ainsi que le nom des librairies correspondantes

-libso est une option précisant que l'on compile une librairie. wmake s'utilise sans options pour compiler les solveurs, de la même manière que présentée précédemment, à ceci près que le dossier utilisé est appelé "Make", et que "files" contient le nom de l'application, et non celui d'une librairie.

Ces nouveaux modèles et nouvelles librairies peuvent être stockés dans n'importe quel répertoire, tant que l'utilisateur y a accès, cependant OpenFOAM comporte ses propres répertoires destinés aux utilisateurs, qui ont l'avantage d'être accessible via des alias.

Les solveurs :

Les solveurs définissent comment le problème va être résolu, quelles données vont être utilisées, etc. C'est notamment là que sont posées toutes les équations à résoudre, ainsi que les variables à lire et écrire. Il est composé principalement de deux fichiers en plus d'un dossier "Make" qui a un rôle similaire à celui de Allwmake évoqué précédemment :

— Un fichier .C ayant le nom du solveur (exemple : interFoam.C), dans lequel sont posées l'appel des modèles et classes nécessaires, ainsi que la boucle temporelle avec les équations à résoudre
— Un fichier createFields.H, dans lequel sont déclarés tous les champs de variables qui vont être utilisés par le solveur, ainsi que les détails de lecture et d'écriture. Il contient également les références de certaines constantes, ainsi que l'endroit où les lire (en général un dictionnaire dépendant du cas à traiter).

Si la boucle temporelle comprend beaucoup d'opérations à effectuer (ce qui est très vite le cas lorsque les solveurs se complexifient un peu), certaines parties peuvent être mises dans des fichiers header à part (par exemple, l'équation de Navier-Stokes sera stockée dans un fichier UEqn.H, l'équation de la chaleur dans un fichier TEqn.H...) et appelées au moment opportun.

Les solveurs les plus simples (une phase, incompressible) n'utilisent pratiquement aucun modèle, et n'appellent que les classes indispensables à OpenFOAM, comme celle gérant l'itération du temps ou celles définissant les objets d'OpenFOAM. De ce fait, on peut se permettre de rajouter des champs, des variables et de constantes directement dans le solveur. Pour les solveurs plus compliqués (deux phases, compressible, viscosité variable, prise en compte de la convection...), l'ajout de variables et paramètres se fait dans les modèles, et le solveur ne fait que les appeler.

Comme dit précédemment, afin de pouvoir utiliser le solveur, il faut le compiler en se servant de `wmake`. Une fois cela fait, il pourra être utilisé pour les cas à traiter comme une commande Linux, soit :

`solveur [repertoire contenant le cas à traiter] -options.`

Le choix vers le répertoire n'est pas nécessaire si l'on se trouve déjà dans ce répertoire (de la même manière que wmake)

Les cas à traiter :

Les cas à traiter sont des répertoires contenant tout ce qui est nécessaire au problème que l'on veut résoudre. Il est composé à la base de trois sous-répertoires (voir figure 2.3) :

- un répertoire ***system*** : Il contient les dictionnaires avec les informations pour la création de mesh, les paramètres temporels (début, fin, pas de temps,...) et d'écriture des données (pas de temps, format...), et il contient également les modèles numériques à utiliser pour les dérivées, ainsi que les tolérances et la précision demandée pour chaque variable. Il peut également contenir des paramètres supplémentaires pour la parallélisation et le traitement des données. C'est aussi là qu'on défini quel solveur utiliser.
- un répertoire ***constant*** : Il contient des dictionnaires avec les paramètres physiques constants (type de fluide, densité, viscosité...) ainsi que les modèles physiques utilisées, pour les turbulences par exemple (laminaire, RAS [2], LES [3]...). Il contient également un répertoire polyMesh contenant les données décrivant le mesh.
- un répertoire ***0*** : C'est le premier répertoire de temps. Il contient un fichier pour chaque variable du problème dans lesquels sont stockées les valeurs initiales ainsi que les conditions aux limites de ces variables.

2.2.2 Exécuter une simulation

Je vais ici expliquer succinctement comment traiter un cas OpenFoam, un exemple détaillé est fourni en Annexe A.

Pour traiter un problème, il faut tout d'abord créer le mesh, c'est à dire la grille qui nous servira de domaine de travail.

OpenFOAM dispose d'un large panel d'options pour créer ces grilles, cependant j'ai utilisé pour la plupart des cas l'application blockMesh. Cette application créé une grille à partir du fichier blockMeshDict (voir figure 2.3). On définit la grille via des blocs et via les conditions aux limites du mesh. Les blocs sont eux-mêmes définis par leurs sommets, et éventuellement par des paramètres de courbures des arêtes, celles-ci étant par défaut des droites. On défi-

2. Reynolds-averaged simulation
3. large-eddy simulation

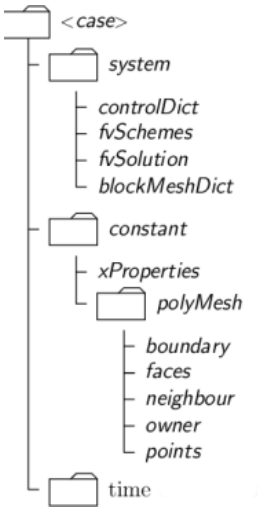

FIGURE 2.3 – *Arborescence du répertoire "cas à traiter"*

nit également les différents bords de la grille et éventuellement une contrainte pour les conditions limites (on peut simplement mettre `patch` pour laisser un choix total, ou par exemple préciser `wall`, ce qui limitera les choix de conditions limites à celles envisageables pour un mur, voir Annexe A). Une fois cela fait, il suffit d'utiliser la commande `blockMesh` dans le terminal pour créer le mesh. Cette commande va créer le dossier `polyMesh` présent dans le dossier `constant`.

Après avoir fait cela, il faut définir les conditions aux limites du domaine pour les différentes variables. Cela se fait dans le répertoire [0]. Pour chaque champ variable, il y a un fichier avec les dimensions de la variables, les valeurs initiales dans la grille, ainsi qu'un dictionnaire comprenant une condition limite pour chaque bord (voir Annexe A).

Il est à noter que le répertoire n'a pas à forcément s'appeler "0" : en effet, il est possible de lui donner le nom de n'importe quel instant t où l'on veut que la simulation commence, tant qu'on le précise dans `system/controlDict`. C'est notamment utile pour apporter une modification à la simulation à un instant t : on fait tourner la simulation de 0 à t, puis on effectue un changement dans les conditions limites à l'instant t, voire dans les propriétés de transport, et on fait repartir la simulation de t jusqu'au temps désiré.

Si la simulation comprend plusieurs phases, il peut être nécessaire de compléter le fichier `setFieldsDict` et d'éxécuter l'application setFields dans le terminal. Basiquement, cela est nécessaire si les conditions initiales comprennent au moins deux valeurs différentes pour un champ ou plus dans la partie interne de la grille (i.e. sans prendre en compte les bords).

Une fois tout cela fait, il convient de préciser les différents schémas numériques à utiliser pour les termes différentiels des équations utilisés, ainsi que les

paramètres de résolution (entre autres la tolérance) pour les différents champs calculés. Ces données se trouvent respectivement dans les fichiers `fvSchemes` et `fvSolution` du répertoire `system`.

Les paramètres physiques, telles que les constantes physiques ou les modèles de transport ou thermodynamiques à utiliser sont précisés dans les fichiers `turbulenceProperties`, `transportProperties` et/ou `thermodynamicalProperties` (certains solveurs utilisent différents types de fichiers).

Enfin, il faut rentrer dans `system/controlDict` les paramètres d'exécution de la simulation, tels que les instants de début et de fin, le pas de temps de la simulation, mais également le pas de temps et le format d'écriture, choisir si l'on veut un pas de temps variable ou fixe, etc.

Il reste une dernière chose à faire : si l'on veut faire tourner la simulation en parallèle, il faut utiliser l'application `decomposePar`, qui utilise le fichier `system/decomposeParDict`, et qui découpe le domaine en un nombre choisi de sous-domaines.

Une fois tout cela fait, on peut lancer la simulation avec une commande du type :

```
mpirun -np 2 myLavaInterFoam -parallel > log &
```

- `mpirun -np 2` : lance une application en parallèle sur 2 CPUs.
- `myLavaInterFoam -parallel` : nom du solveur que l'on utilise, et indique qu'il va être exécuté en parallèle.
- `> log` : écrit l'output de l'application dans un fichier nommé log.
- `&` : exécute la tâche en arrière plan.

Ici, on suppose qu'on se trouve dans le répertoire du cas à traiter, on a donc pas besoins de rajouter le chemin vers ce répertoire. De plus, l'appel est ici en parallèle. Si l'on souhaite exécuter simplement la simulation en séquentiel, il suffit simplement d'appeler le solveur.

Une fois l'application lancée, des dossiers contenant les données des différentes variables sur la grille seront crées pour chaque pas de temps d'écriture. À chaque pas de temps d'éxécution, le solver renverra une série d'informations sur le pas de temps en cours, comprenant entre autres : l'instant t auquel en est rendu la simulation, le pas de temps actuel, le nombre de Courant actuel, ainsi que les résidus et éventuelles erreurs pour les différentes variables calculées, et les temps d'exécution et d'horloge.

Chapitre 3

Simulation de l'écoulement de la cryolave

Je vais expliquer ici comment j'en suis arrivé à la simulation obtenue à la fin de mon stage, en décrivant la création d'un solveur prenant en compte les différents mécanismes thermodynamiques et mécaniques de l'écoulement, tout en cherchant des valeur réaliste, ou au moins crédible des paramètres du problème du problème, le but final étant d'avoir quelque chose ressemblant à un écoulement de cryolave sur la surface titanienne.

Le solveur dont j'ai besoin doit prendre en compte les différents aspects thermodynamiques et mécaniques de l'écoulement de la cryolave. Cela inclut :

- **Un écoulement à deux phases :** on modélise une phase liquide (la cryolave) et une phase gazeuse (l'atmosphère de Titan). la composition de l'atmosphère est suffisamment bien connue pour notre problème, néanmoins il n'y a pas de certitude sur la composition de la cryolave, si cryolave il y a.
- **Un rentrée de la cryolave dans le domaine de travail par un "trou" dans le sol :** C'est également quelque chose d'assez basique dans OpenFOAM, mais nécessaire si l'on veut se rapprocher d'un cas de cryovolcanisme
- **Une variation de la température de ces deux phases :** En effet, la lave sortira du sol relativement plus chaude que la température de surface. Il faudra introduire des équations de transfert de chaleur, et prendre en compte ces transferts autant entre les phases qu'au bord du domaine, notamment à l'interface cryolave-sol.
- **De la convection dans l'atmosphère :** Si il doit être possible de la négliger dans la cryolave, au vu de la densité et de la viscosité, il est fort probable qu'elle ait lieu dans l'atmosphère. D'après DAVIS et al. 2010, il y a de grandes chances pour que la convection, naturelle et forcée, soit le mécanisme prépondérant du refroidissement de la cryolave.
- **Un écoulement non Newtonien :** Peu de liquides sont en réalité newto-

nien, et la cryolave ne fait pas exception, son comportement étant rhéofluidifiant, autrement dit sa viscosité diminue lorsque le gradient de vitesse augmente.

— **Une viscosité variant avec la température :** La cryolave voient sa viscosité augmenter lorsque sa température diminue.
— **Une modélisation réaliste du terrain :** La dernière étape serait de créer une grille plus complexe, se rapprochant au possible de terrain où le cryovolcanisme a pu être détecté.

3.1 Solveur à plusieurs phases

Traitons d'abord le cas d'un modèle à plusieurs phases. C'est quelque chose de a priori très compliqué, cependant OpenFoam possède plusieurs solveurs traitant de cela. Les deux susceptibles de nous intéresser le plus sont **InterFoam**, un solveur à deux phases immiscibles, incompressibles, et isothermes, et **compressibleInterFoam**, un solveur pour deux fluides immiscibles, compressibles et non-isothermes. **compressibleInterFoam** se rapproche plus de ce que l'on veut faire, et comporte plusieurs fonctionnalités intéressantes, néanmoins il est bien plus complexe à utiliser et modifier. C'est pourquoi après avoir essayé les deux, j'ai choisi d'utiliser **InterFoam**, un solveur bien plus basique, mais ainsi bien facile à utiliser et moduler. Le solveur est basé sur la méthode du Volume de Fluide (ou VOF, Volume of Fluid). Cette méthode est basée sur une variable appelée fraction de phase, fraction de volume, ou encore fonction de fraction, et qui aura pour but d'indiquer la présence ou non de tel ou tel fluide. Plus précisément, pour notre cas à deux phases, elle prendra une valeur entre 0 et 1 : si sa valeur est 1 dans une cellule, cela signifie que la cellule est remplie de cryolave ; si sa valeur est 0 ,la cellule est remplie d'air ; une valeur entre 0 et 1 indique que l'interface entre l'atmosphère et la cryolave est présent dans la cellule. L'idée de la méthode VOF est de suivre cet interface. Pour cela, on considère l'ensemble des fluides lors de la résolution des équations, en supposant que les deux phases partagent les mêmes champs de variables. On résout ensuite une équation de conservation pour fraction de phase :

$$\frac{\partial \alpha}{\partial t} + \vec{\nabla} \cdot (\vec{U} \cdot \alpha) = 0 \quad , \tag{3.1}$$

où $\vec{U}$ est la vitesse et α la fraction de phase. Si la résolution de la grille a bien sûr un rôle important de toute manière, on peut voir ici qu'elle sera particulièrement cruciale, car une bonne simulation va demander une bonne résolution de l'interface.

Hormis cette équation, différentes équations sont déjà implémentées, telle que l'équation de Navier-Stokes, ainsi que diverses corrections pour la pression (détails en Annexe...).

Ce sera de ce solveur que je partirai pour aboutir à mon solveur final. Un cas de base est fourni avec OpenFOAM pour tester le solveur : il s'agit de "damBreak", où un volume de liquide (ayant les propriétés de l'eau) rencontre en

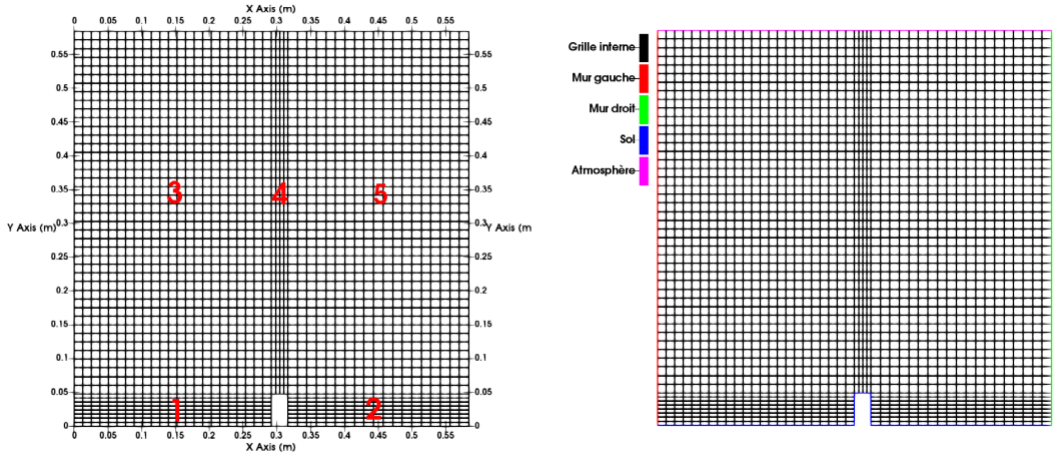

FIGURE 3.1 – *"Mesh", ou grille découpant l'espace de travail utilisé pour le modèle damBreak. À gauche sont numérotés les différents blocs définis qui composent la mesh, et les axes avec les dimensions de la grille sont données. À droite, les différents bords, ou champs limites, définis via blockMesh sont montrés.*

s'écoulant un obstacle. Les fichiers utilisés par OpenFOAM pour traiter ce cas sont retranscrits et expliqués en détail en annexe A.

On peut voir figure 3.1 la grille utilisée pour ce cas. On notera que la condition de non-dépassement évoquée partie 2.1.1 pour la méthode des volumes finis est respectée lors de la définition de plusieurs blocs, entre les cellules mais également entre les blocs. Pour ce qui est des bords, on peut remarquer qu'il aurait était possible de simplifier un peu le problème en incluant dans un même champ limite les murs droit et gauche : en effet, dans ce problème, les deux côtés ont les même conditions limites, il n'y a donc pas raison de les séparer. On peut également noter que des faces de différents blocs peuvent former un même champ limite (exemple : une face du bloc 1 et une du bloc 3 forment le mur gauche). Il n'est cependant pas possible de définir plusieurs champs limites sur une même face.

La commande setFields va permettre de placer le volume d'eau souhaité sur la mesh, et après avoir défini les paramètres physiques des deux phases et les conditions limites pour la pression, la vitesse et la fraction de phase (qui pour l'instant sont les seuls champs à calculer, voir Annexe A), on lance la simulation avec la commande **interFoam**. Le résultat est montré figure 3.2. Il faut noter qu'ici, nous n'avons pas pris en compte les turbulences (modèle laminaire). On peut interpréter les "nuages blancs" visibles après l'impact au barrage, comme des nuages de goutelettes, trop petites donc pour remplir entièrement une cellule.

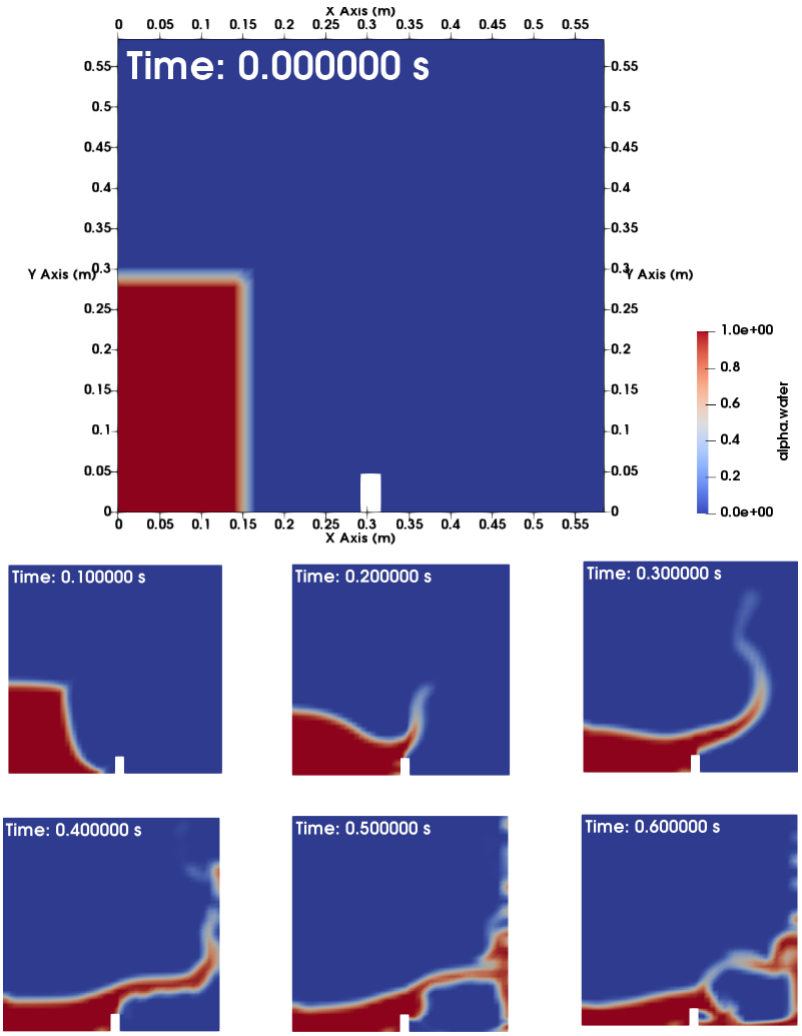

FIGURE 3.2 – *État initial du système à t=0s (grande figure en haut), et quelques visualisations à t=0.1s, 0.2s, 0.3s, 0.4s, 0.5s et 0.6s. Le champ représenté est celui de la fraction de phase de l'eau. La couleur rouge indique la présence de l'eau, la couleur bleue indique son absence, et ce qui se trouve entre ces deux extrêmes constituent l'interface. Sur la représentation montrée ici, le champ a été lissé (autrement, pour le cas initial, il n'y aurait que du rouge et du bleu, l'eau remplissant exactement les cellules).*

3.2 Rentrée du liquide depuis le sol

Le cas du barrage est certes bien fait, mais il est encore loin de répondre à nos attentes. Afin de créer notre cryovolcan, il va d'abord falloir changer la configuration du terrain. L'idée est de faire rentrer la cryolave dans l'espace de

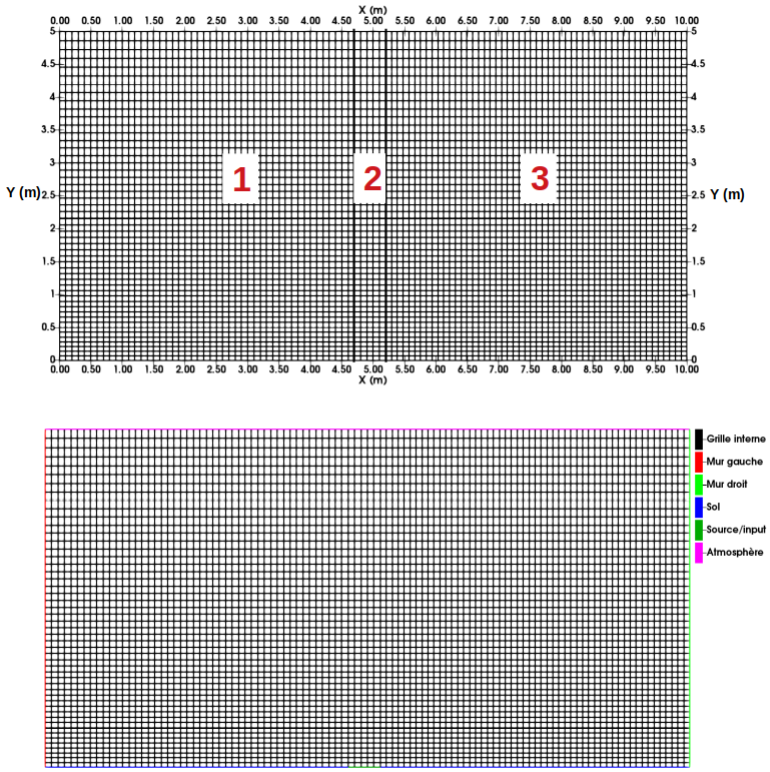

FIGURE 3.3 – *Mesh fonctionnelle pour la rentrée du liquide via une source dans le sol. En haut : Grille avec les différents blocs délimités et les dimensions. En bas : Grille avec les conditions limites*

travail depuis le sol. Il va donc falloir employer une condition d'input.

Je redéfinis tout d'abord la mesh, afin d'enlever le barrage, et de mettre à la place une "source", d'où le liquide émergera. Après plusieurs essais, je suis arrivé à la mesh visible figure 3.3. Je détaille en annexe **??** les différentes mesh que j'ai essayé et qui ne fonctionnaient pas.

Après avoir utilisé blockMesh et setFields, on lance interFoam. Le résultat est présenté figure 3.4. La légende est la même que dans la figure 3.2. L'écoulement semble avoir un comportement normal. Afin de rendre la simulation plus proche de ce que l'on cherche, j'enlèverai par la suite les murs sur les côtés, en les remplaçant par des conditions limites d'écoulement vers l'extérieur libre.

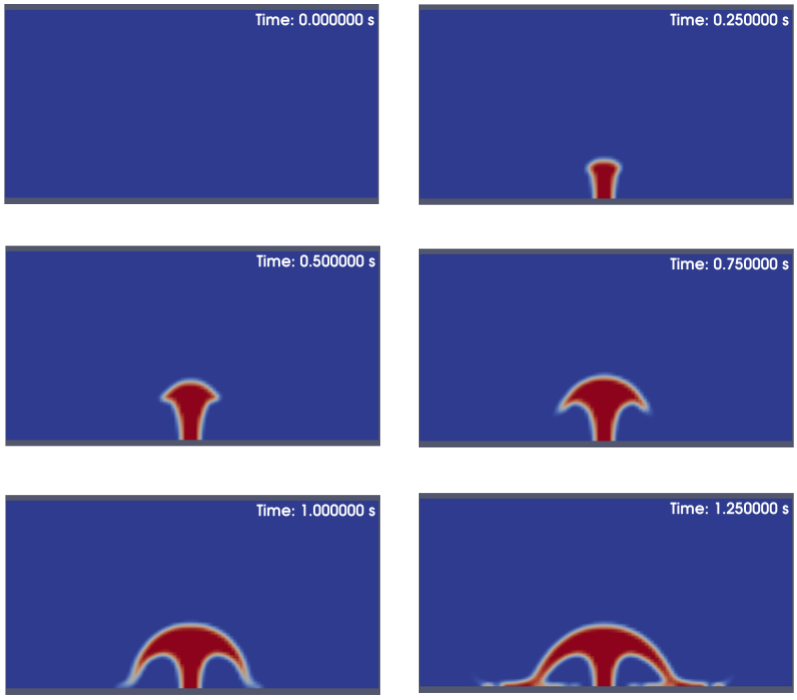

FIGURE 3.4 – *Simulation de l'entrée du liquide via une source dans le sol. Les dimensions sont les mêmes que celles données sur la figure 3.3.*

3.3 Prise en compte de la température

Maintenant que tout cela est fait, il va falloir commencer à modifier le solveur en lui-même. L'idée est de créer un nouveau solveur, basé sur interFoam, que j'appellerai myLavaInterFoam.

L'objectif ici est tout d'abord d'implémenter l'équation suivante :

$$\rho c_p * \left(\frac{\partial T}{\partial t} + \vec{u} \cdot \vec{\nabla} T \right) = \vec{\nabla} \cdot \left(k \vec{\nabla} T \right) \quad , \tag{3.2}$$

où ρ est la densité, c_p est la capacité thermique spécifique, T est la température, $\vec{u}$ est la vitesse, et k est la conductivité thermique.

La démarche que je m'apprête à décrire et qui à pour finalité l'implémentation d'une équation de la chaleur est basée sur le projet de Qingming Liu, un étudiant à l'Université de Technologie de Chalmers. Il a accompli son projet sur une ancienne version d'OpenFoam, il a donc fallu que je procède un peu autrement.

Modification du modèle

Avant de modifier le solveur en lui-même, il a fallu que j'aille modifier le modèle à deux phases utilisé par interFoam : **immiscibleIncompressibleTwoPhaseMixture**. Comme son nom l'indique, il s'agit d'un modèle utilisé pour une mixture à deux phases immiscibles et incompressibles. Ce modèle hérite d'un autre modèle : incompressibleTwoPhaseMixture. Pour des raisons que j'expliquerai dans la partie **3.4**, j'ai implémenté mes ajouts dans immiscibleIncompressibleTwoPhaseMixture.

Concernant l'équation 3.2, hormis la densité qui est déjà implémenté dans incompressibleTwoPhaseMixture, le reste de ces variables et paramètres est à ajouter. Afin de limiter le nombre de variables libres, nous allons calculer la conductivité thermique avec la formule suivante :

$$k = \rho\nu * \frac{c_p}{\mathrm{Pr}}, \tag{3.3}$$

où ν est la viscosité cinématique, et Pr est le nombre de Prandtl, un nombre adimensionné qui est définit ainsi :

$$\mathrm{Pr} = \frac{c_p\mu}{k} = \frac{\nu}{\alpha} = \frac{diffusivit\ mcanique}{diffusivit\ thermique}, \tag{3.4}$$

avec $\mu = \rho\nu$ la viscosité dynamique, et $\alpha = \frac{k}{\rho c_p}$ la diffusivité thermique. Il s'agit d'un nombre caractéristique du fluide à un état donné, et qui est souvent presque constant pour la plupart des gaz et certains liquides, il n'est donc pas rare de l'utiliser pour calculer la conductivité thermique. Sa valeur est également une indication sur le transfert thermique dominant : si $\mathrm{Pr} \ll 1$, cela signifie que la conduction thermique est dominante par rapport à la convection. Si $\mathrm{Pr} \gg 1$, la convection dominera.

Il faut cependant prendre en compte le fait que nous travaillons avec deux phases. Nous allons donc procéder ainsi :

— Nous allons implémenter un c_p et un Pr par phase, que l'on identifiera par les indice 1 et 2 (c'est ainsi que la densité est traitée). Nous allons les implémenter en tant que scalaires dimensionnées : c'est un type spécial créé par OpenFOAM pour les constantes (ou au moins ne dépendant pas de la position sur la grille, donc n'étant pas un champ), et qui a l'avantage de fournir une dimension au scalaire, ce qui permet à OpenFOAM de détecter les erreurs d'homogénéité.

— k va être défini en tant que champ scalaire, et sa formule va être modifié de cette façon :

$$k = x\rho_1\nu_1\frac{c_{p1}}{Pr_1} + (1-x)\rho_2\nu_2\frac{c_{p2}}{Pr_2}, \tag{3.5}$$

avec x la fraction de phase, l'indice indiquant la phase liquide et l'indice 2 indiquant la phase gazeuse. x étant un champ scalaire, cela permettra à la conductivité d'avoir la valeur correspondante à la phase présente dans chaque cellule, et une valeur intermédiaire à l'interface.

Nous déclarons les différents variables dans le fichier .H, et initialisons leurs valeurs dans le fichier .C. Il est à noter que l'on ne donne pas directement une valeur aux paramètres physiques dans le modèle (le modèle perdrait de sont intérêt) : on lui demande, via des fonctions d'OpenFOAM, d'aller chercher ses valeurs dans les fichiers du cas à traiter.

Une fois toutes ces modifications faites, il faut juste modifier le nom du modèle dans le fichier `Make/files` (appelons-le "myimmiscibleIncompressibleTwoPhaseMixture"), et le compiler avec `wmake libso`.

Modification du solveur

Une fois le modèle modifié et compilé, il faut modifier le solveur. Celui-ci commence par créer les différents champs qu'il va utiliser, puis créé un objet du type immiscibleIncompressibleTwoPhaseMixture (appelé `mixture`), et l'utilise pour initialiser les variables, voire les champs, à partir des données fournies dans le cas à traiter. Tout cela est implémenté dans le fichier textttcreateFields.H. On va tout d'abord rajouter à ce fichier la déclaration d'un champ scalaire T pour la température, ainsi que la déclaration de deux références vers c_{p1} et c_{p2} en allant les chercher dans `mixture` (objet qui je le rappelle contient la capacité thermique et le nombre de Prandtl de chaque phase, ainsi que le champ scalaire de conductivité thermique). On va ensuite déclarer un champ scalaire appelé rhoCp, qui ne sera pas écrit en output par le solveur, mais qui sera utilisé dans l'équation de la chaleur. On utilisera la formule suivante pour l'initialiser et la mettre à jour à chaque pas de temps :

$$x * \rho_1 * c_{p1} + (1 - x) * \rho_2 * c_{p2} \quad , \tag{3.6}$$

où x est la fraction de phase et ρ_1, ρ_2, C_{p1} et C_{p2} sont les densités et les capacités thermiques respectives des phases 1 et 2. Le but est encore une fois ici d'avoir les valeurs des paramètres physiques correspondant à la phase présente dans chaque cellule.

Pour le traitement numérique, le solveur a également besoin d'un champ surfacique du flux. Celui-ci va prendre le nom de rhoPhiCpf. On va le déclarer de la même manière que rhoCp, sauf qu'il s'agira d'un champ surfacique et non volumique. On initialisera sa valeur ainsi :

$$(\rho\phi c_p)_f = \phi_x * (\rho_1 * c_{p1} - \rho_2 * c_{p2}) + \phi * \rho_2 * c_{p2}$$

Je n'ai pas eu le temps pendant mon stage de me plonger en détail dans le fonctionnement profond d'OpenFOAM et de ses solveurs, aussi je ne suis pas certain de la fonction de cette variable, cependant il semble qu'OpenFOAM traite, en plus des équations usuelles, des équations similaires mais avec le flux en facteur, et sur des surfaces (que je suppose être les faces des cellules).

On notera que `createFields.H` ne sert qu'à déclarer et initialiser les champs et les variables. Afin de mettre à jour ces deux derniers termes (rhoCp et rhoPhiCp) au fil des pas de temps, il faut ajouter les formules pour les calculer

respectivement dans `alphaEqnSubCycle.H` et `alphaEqn.H`, qui sont appelés à chaque pas de temps par le solveur.

Une fois tout cela ajouté, j'implémente l'équation et sa résolution dans le solveur (voir annexe **??** pour les détails des équations dans OpenFOAM), puis je change le nom du solveur et sa localisation dans `Make/files` (si la localisation n'est pas changée, lors de la compilation OpenFOAM tentera d'écrire le solveur dans des répertoires protégés auxquels l'utilisateur n'a pas accès), et enfin on ajoute dans `Make/options` les répertoires et le nom de librairie liés à notre nouveau modèle, et on compile notre solveur.

Afin de vérifier que cela fonctionne, on prend le cas avec une source utilisé précédemment, et on rajoute des températures initiales : 100K pour l'air et le sol, et 200K pour le liquide à la source. La figure 3.5 montre le résultat. Comme on peut le voir, notamment au niveau du sol, la phase liquide a tendance à se refroidir, et d'un autre côté l'air en contact avec la lave se réchauffe.

3.4 Ajout de la convection

Si le résultat précédent commence à se rapprocher de ce que l'on souhaite, il reste encore plusieurs mécanismes à traiter, à commencer par la convection dans l'atmosphère. En effet, l'équation de la chaleur précédente ne traite que les transferts par conduction. Hors, d'après DAVIS et al. 2010, il est très probable que la convection dans l'atmosphère joue un rôle prépondérant dans le refroidissement de la lave. Je vais ici surtout parler de la convection naturelle (les mouvements dû aux gradients de température), la convection mécanique apparaissant simplement par la suite.

La convection naturelle va se traduire par une variation de densité locale liée au changement de température, et qui va mener à l'apparition d'un mouvement dans le fluide. Je vais utiliser ici l'approximation de Boussinesq, qui est déjà utilisée dans OpenFoam, dans un solveur nommé buoyantBoussinesqPimpleFoam, conçu pour traiter des cas à une phase non-isotherme avec convection. L'approximation de Boussinesq introduit une variation dans la densité :

$$\rho = \rho_0 * (1 - \beta(T - T_{ref})) \quad , \tag{3.7}$$

où ρ est la densité, ρ_0 est une densité de référence, où densité telle qu'elle serait sans convection, β est le coefficient d'expansion thermique, T est la température et T_{ref} est une température de référence, ou température initiale.

L'idée ici est de mettre à jour la valeur de ρ avec cette formule à chaque pas de temps. Il va cependant falloir modifier certaines choses avant, à commencer par l'ajout de β et T_{ref} pour chaque phase au modèle immiscibleIncompressibleTwoPhaseMixture. Les ajouter là permettra de les lire depuis le fichier `constant/transportProperties` du cas à traiter.

Remarque : *Au départ, j'avais tenté d'implémenter toutes les modifications concernant l'équation de la chaleur dans* ***incompressibleTwoPhaseMixture****. Si, lors de la*

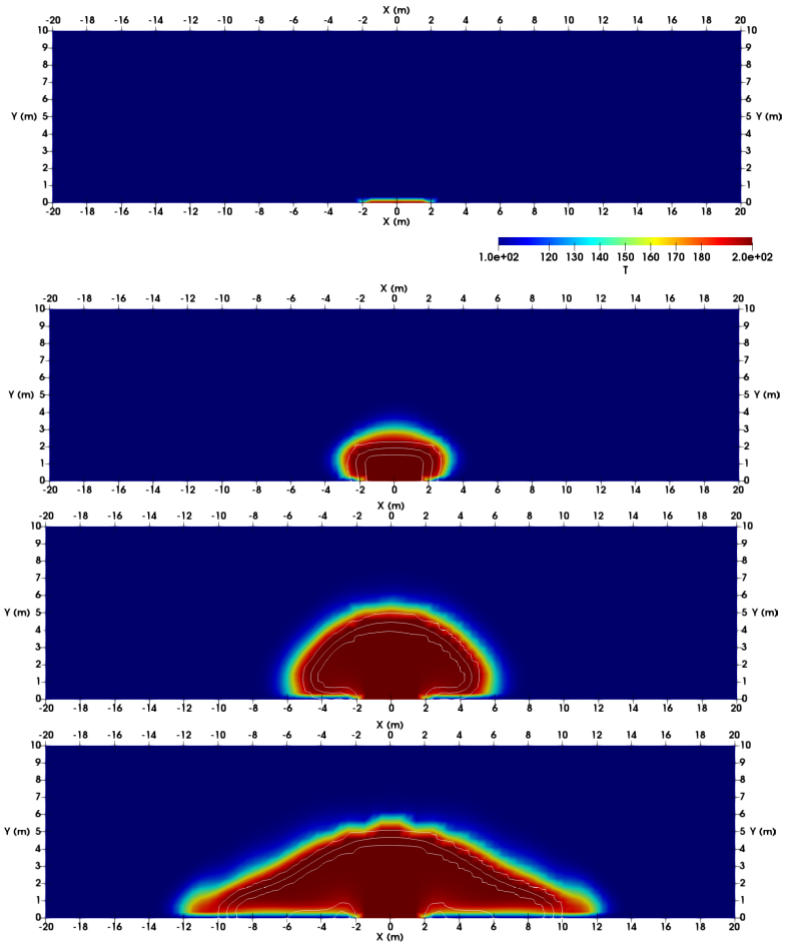

FIGURE 3.5 – *Simulation de l'écoulement après ajout de la température. Encore une fois, la représentation des champs est adoucie. Les lignes blanches représentent la valeur de la fraction de phase, respectivement 0.1, 0.5, 0.9 (de haut en bas). L'interface liquide-air se situe probablement près de 0.5, et dans tous les cas entre 0.1 et 0.9. La couleur représente la température, bleu étant 100K et rouge 200K.*

compilation du modèle et du solveur, je n'avais pas de problème particulier, je tombais systématiquement sur une erreur de segmentation. L'erreur avait lieu avant même que la boucle temporelle ne commence, et le message d'erreur été renvoyé par une fonction du code source d'OpenFOAM, si bien que je ne pouvais pas la modifier, même pour rajouter des "print" afin de mieux cerner l'erreur. Après plusieurs semaines, j'ai finalement réussi à trouver la provenance de l'erreur : si les variables c_p étaient Pr correctement lues depuis le cas à traiter, le modèle n'arrivait pas à lire les variables β et T_{ref}, qui étaient pourtant implémentées de la même manière. De plus, si les 4 variables (β_1, β_2, T_{ref1} et T_{ref2}) posaient problème, le problème était différent pour chaque variable : les variables de ma phase air (phase 2) étaient lues mais renvoyaient des valeurs aberrantes, et celles de la phase lave (phase 1) renvoyaient des erreurs, sachant que β_1 et T_{ref1} renvoyaient chacun une erreur différente. C'est finalement en déplaçant vers le modèle héritant de ***incompressibleTwoPhaseMixture***, ***immiscibleIncompressibleTwoPhaseMixture****, qui lui est directement appelé par le solveur, que le problème a pu être réglé.*

Une fois ces variables ajoutées et le modèle compilé, il suffit de les appeler dans le solveur depuis `createFields.H`. Ensuite, l'idée est de rajouter l'équation 3.7 juste après la résolution de l'équation de la chaleur. Cependant, cela est plus compliqué qu'il n'y parait. En effet, le solveur traite deux phases incompressibles, autrement dit les densités sont fixées. Il va donc falloir changer cela.

Les densités de chaque phase sont implémentées dans un modèle et appelés par le solveur, de la même manière que les variables que nous avons créées, cependant elles sont appelées en tant que scalaire dimensionné constant. Cela pose deux problèmes :

— Le type "scalaire dimensionné" n'est pas adapté à un paramètre étant amené à varier au cours du temps et en fonction de la position.

— On ne dispose pas d'une mais de deux densités.

L'idée va donc être de faire comme pour la conductivité thermique et "rhoCp" : créer un champ de densité. Notre champ de densité aura comme valeur :

$$x * \rho_1 + (1 - x) * \rho_2 \quad ,$$

avec x la fraction de phase, et se sont les valeurs de ρ_1 et ρ_2 qui seront recalculées à chaque pas de temps avec la formule de Boussinesq. Je créé donc deux champ nommés "rho1" et "rho2", puis, pour éviter un conflit entre les noms de variables, je change le nom des scalaires dimensionnés en "rho1c" et "rho2c" (ajout d'un c pour "constant"), et à partir de maintenant ces deux densités serviront de densité de référence pour chaque phase. Puis je créée un champ nommé "rho", dont la valeur sera initialisée et mise à jour à chaque pas de temps avec la formule précédente. L'initialisation se fait dans `createFields.H` et la mise à jour est faite, comme dit précédemment, juste après l'équation de T.

Une fois tout cela implémenté, on compile le solveur, et on peut le tester sur un problème. Afin de bien mettre en valeur la convection, j'ai pris un cas où les 2 phases sont au repos, avec la phase liquide plus chaude que la phase

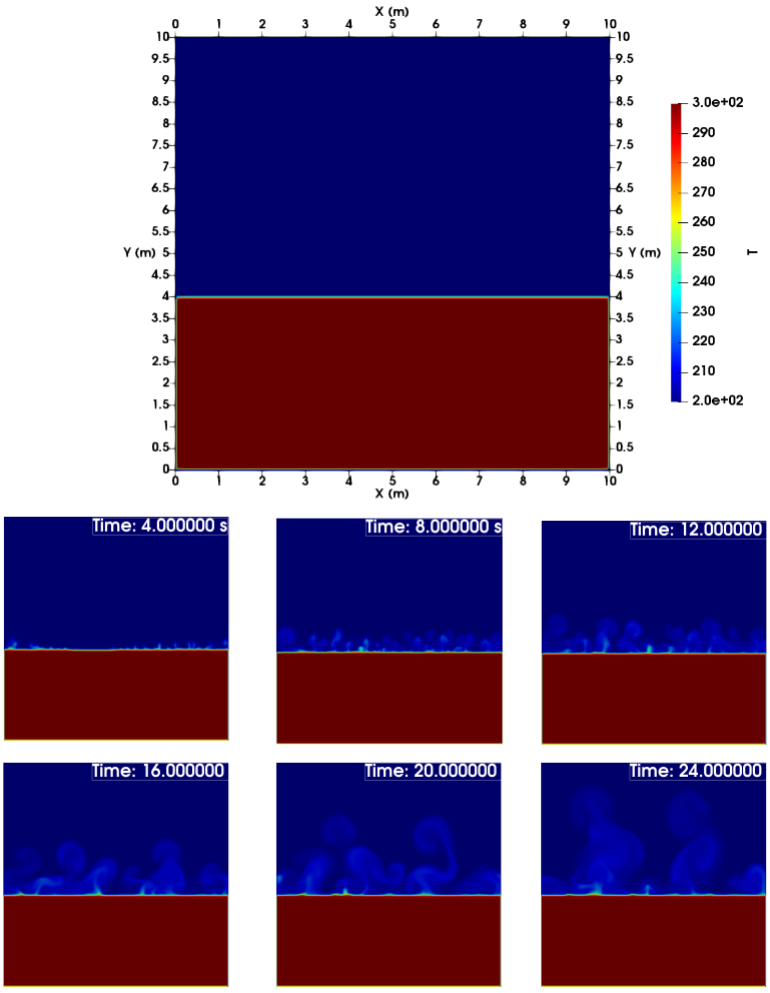

FIGURE 3.6 – *Simulation de la convection avec un phase liquide en bas à 300K et une phase gazeuse dans la moitié supérieure à 200K. La convection a été "désactivée" dans la phase liquide.*

gazeuse au-dessus, et de la convection seulement dans la phase gazeuse (pour supprimer la convection dans une phase, il suffit de mettre le coefficient β à 0). Le résultat est présenté figure 3.6. Le résultat ressemble à ce que l'on pourrait attendre de la convection.

Même si ce résultat parait correct, il ne faut pas oublier ce qu'est l'approximation de Boussinesq : une approximation. Il convient de vérifier si cette approximation est vraiment juste. Cette approximation n'est en faite valide que si les variations de densité restent négligeables devant la densité de départ, soit dans notre cas :

$$\frac{\beta(T - T_{ref})}{\rho_0} << 1 \tag{3.8}$$

Les paramètres physiques de notre problème seront discutés et précisés plus tard, cependant, on peut considérer une atmosphère à environ 100K. La cryolave étant mal déterminée, on peut supposer afin d'avoir de la marge une température extrême de 300K en sortie de croûte (on la prendra plutôt par la suite à 200K). On prendra de plus un coefficient d'expansion thermique de l'atmosphère d'environ 1e-2 K^{-1}, et une densité de 5.3 $kg \cdot m^{-3}$. On obtient alors $\frac{\beta(T-T_{ref})}{\rho_0} = 0.38$. Même si le résultat est inférieur à 1, on ne peut pas considérer ainsi la variation comme négligeable devant la densité de référence, même en réduisant l'écart de température à 100K. Il faut donc relativiser ces résultats. Néanmoins, la convection ne concerne pour l'instant que l'atmosphère, qui n'est pas notre sujet d'étude principal, on se permettra donc de continuer à utiliser cette approximation.

3.5 Variation de la viscosité

Maintenant que nous avons traité l'aspect de variation de la température dans les deux phases, il est temps d'aborder le problème de la viscosité. En effet, comme précisé au début de cette partie, plusieurs choses font varier la viscosité, et ce de manière non-négligeable. C'est un point important et un peu plus complexe que les points précédents, car la plupart des lois dont on dispose pour représenter ces variations de viscosité sont empiriques, autrement dit on les a créées de manière à fitter les données expérimentales. Par conséquent, elles ne sont en général valides que sur des intervalles limités de viscosité, et les constantes sont pour la plupart très dépendantes du fluide considéré, de ses propriétés physiques, et des conditions de l'expérience (pression, température, ...), entre autres. Les approximations faites dans cette section seront en conséquence plus grossières que précédemment.

3.5.1 Liquide non-Newtonien

La première cause de variation de viscosité est l'aspect non-Newtonien de la cryolave. Pour être exact, elle a un comportement pseudoplastique, c'est-à-dire que si les contraintes auxquelles est soumise la cryolave augmentent, sa viscosité diminuera. Cet aspect reste assez simple à traiter, car OpenFOAM dispose déjà de plusieurs modèles pour la viscosité. Dans le fichier `transportProperties`, il est possible de fournir un modèle de transport, qui correspond au modèle que va suivre la viscosité. de plus, pour le solveur InterFoam, nous avons la possibilité de fournir des modèles différents pour l'une ou l'autre des phases. Pour ma part, je vais utiliser un modèle Newtonien (viscosité constante) pour l'atmosphère (cela ne devrait pas influencer significativement l'écoulement), et

une loi de puissance pour la cryolave. Le solveur utilise la viscosité cinématique, l'équation de puissance aura donc cette forme :

$$\nu = k * \left(1.0 * \frac{d\epsilon}{dt}\right)^{n-1} \quad , \tag{3.9}$$

où ν (m^2/s) est la viscosité cinématique, k est une viscosité cinématique de référence, ϵ (rapport de deux longueurs, donc sans dimension) est la déformation, et n (sans dimension) est un nombre qui doit être inférieur à 1 pour modéliser le comportement pseudoplastique. Le 1.0 en facteur de la vitesse de déformation est une constante de dimension temporelle, afin d'adimensionner. Cette loi de puissance est fournie avec OpenFOAM comme modèle de transport, mais elle ne convient pas parfaitement avec ce que nous souhaitons faire. En effet, comme l'avons vu précédemment, la densité est modifiée par la prise en compte de la convection, et vu que $\nu = \frac{\mu}{\rho}$, avec μ la viscosité dynamique est ρ la densité, il devient nécessaire de prendre en compte la dépendance en densité. Pour cela, j'ai simplement donné au coefficient k une dimension de viscosité dynamique (pour le différencier de celui utilisé précédemment, nous l'appellerons k_{dyn}), et j'ai divisé par ρ. Cela donne :

$$\nu = \frac{k_{dyn}}{\rho} * \left(1.0 * \frac{d\epsilon}{dt}\right)^{n-1} \tag{3.10}$$

Une fois cela fait, on peut compiler le modèle, et l'utiliser directement dans le cas à traiter, sachant qu'il faudra fournir les paramètres physiques demandés, ici une viscosité maximale et minimale qu'on appelle ν_{min} et ν_{max}, la puissance n, ainsi que la viscosité de référence k_{dyn}. Il est important de préciser un intervalle de viscosité, afin d'éviter les risques de divergence. Ainsi, si la viscosité sort de l'intervalle, le solveur va garder une viscosité constante, correspondant à ν_{min} ou ν_{max}.

3.5.2 La dépendance en température de la viscosité

Un autre aspect plus complexe à traiter est celui de la dépendance en température. En effet, même sans prendre en compte le changement d'état de la cryolave, la viscosité aura tendance à diminuer si la température augmente, et inversement. Ce n'est pas une variation linéaire, est elle peut être influencée en fonction de où l'on se situe sur le diagramme de phase par rapport au liquidus. Pour rappel, le liquidus et une limite séparant deux "sous-états" d'un liquide : au-dessus du liquidus, le liquide est entièrement liquide, et en-dessous, la phase solide commence à se former, en général de manière dispersée dans le liquide, ce qui a tendance à changer ses propriétés, et notamment sa viscosité. Pour simplifier, nous ne prendrons qu'une seule loi pour la dépendance en température.

Selon KARGEL et al. 1991, qui étudie les propriétés rhéologiques des mélanges Eau-Ammoniac (l'une des mixtures probables pour la cryolave), le fit

le plus simple des données obtenues par expérimentation est une loi exponentielle :

$$\mu = \exp\left(A + \frac{B}{T}\right) \quad , \tag{3.11}$$

où $exp(A)$ est une viscosité dynamique de référence, B une "température d'activation", un paramètre lié à l'énergie d'activation, et T la température du liquide.

L'idée va être d'insérer cette loi dans notre loi de puissance. Pour faire cela, on va poser $\exp(A) = k_{dyn}$, ce qui permettra d'utiliser la formule suivante pour notre modèle de viscosité :

$$\nu = \frac{k_{dyn}}{\rho}\left(1.0 * \frac{d\epsilon}{dt}\right)^{n-1} \exp\left(\frac{B}{T}\right) \tag{3.12}$$

Dans l'article, les coefficients A et B sont estimés afin d'avoir le meilleur fit possible. Ils obtiennent les formules suivantes :

$$A = -13.8628 - 68.7617X + 230.083X^2 - 249.897X^3 \tag{3.13}$$

$$B = 2701.73 + 14973.3X - 46174.5X^2 + 45967.5X^3 \tag{3.14}$$

où X est la fraction d'Ammoniac dans la mixture Eau-Ammoniac. On prend la mixture Eau-Ammoniac décrite par KARGEL 1994, qui est composée à 32.6% d'Ammoniac, on a alors A=-20.4847 et B=4268.38K.

3.6 Choix des paramètres physiques des fluides

Maintenant que nous avons créé notre solveur et notre cas à traiter, il convient de définir les paramètres physiques à utiliser.

L'atmosphère de Titan est a priori la plus facile à définir, étant donné que l'on sait qu'elle ait composée à plus de 99% de diazote, et qu'à la surface la pression y est de 1.5 bar et la température est entre 90 et 100K. J'ai d'abord commencé par supposer une atmosphère de N_2 pur à 100K et une pression de 1 bar. On peut trouver sur le site *Engineering ToolBox* p.d. les propriétés physiques du N_2. On trouve :

- $\rho = 3.437$ kg/m^3
- $\nu = 2.024$e-6 m^2/s
- $c_p = 1039$ J.kg^{-1}.K^{-1}. On notera qu'ici, à défaut d'avoir trouvé mieux, il s'agit de la capacité thermique spécifique à 175K, et la pression ne nous étant pas spécifiée, je ne peux que supposer que le calcul est fait à pression atmosphérique (p=1bar).
- Pr = 0.7944
- β : On ne trouve pas la valeur du coefficient d'expansion thermique sur ce site, je vais donc la chercher autre part, sur le site suivant : p.d. Pour p=1 bar et T=200K, je trouve $\beta = 1.06487$e-2 K^{-1}. Cependant, j'ai

d'abord testé pour p=1.5 bar, et plusieurs des nombre physiques que renvoyait ce site (y compris β) avaient une valeur négative alors qu'ils sont censés être positifs. Cela semble être le cas pour des pressions supérieures à 1 bar. Je me permet donc de douter sérieusement de cette dernière valeur.

On peut voir que les données que l'on obtient pour l'atmosphère restent approximatives et incertaines, compte tenu des données que fournissent certains sites.

Pour ce qui est de la cryolave, c'est un peu plus compliqué que l'atmosphère, vu que l'on ne connaît pas vraiment la composition. Pour premier modèle, nous pouvons prendre une mixture Eau-Ammoniac, dont les propriétés et les applications planétologiques ont été plus étudiées que les autres mixtures envisageables. Dans KARGEL 1994, Il est mentionné une mixture Eau-Ammoniac qui pourrait avoir des applications en tant que cryolave. Celle-ci est composée à 32.6% d'ammoniac et à 67.4% d'eau. On donne dans cet article la densité et la viscosité dynamique de cette mixture : $\rho = 946$ kg/m^3 et $\mu = 4$ kg.m^{-1}.s^{-1}. Pour avoir la viscosité cinématique, il suffit alors de diviser μ par ρ. On a alors :

— $\rho = 946$ kg/m^3
— $\nu =$ 4.2e-3 m^2/s

Ces données ont été calculées en les considérant constantes pour toute la phase liquide. La température de fusion de cette mixture étant de 176K environ, on considérera par la suite un liquide à 200K qui sort du sol.

Pour ce qui est des propriétés thermodynamiques, je suis d'abord parti sur une approximation tirée de H MOHD RAZIF et al. 2015, dans laquelle ils calculent la capacité thermique du mélange avec la formule suivante :

$$c_p = \frac{x(\rho c_p)_A + (1-x)(\rho c_p)_W}{x * \rho_A + (1-x) * \rho_W} \quad , \tag{3.15}$$

avec x la fraction d'ammoniac dans le mélange, ρ la densité, c_p la capacité thermique, et les indices $_A$ et $_W$ référant respectivement aux propriétés de l'ammoniac et de l'eau. J'ai utilisé les propriétés de l'ammoniac à 200K (dans son état liquide), cependant l'eau est solide à cette température, quelle que soit sa pression, je me suis donc servi des propriétés de l'eau liquide à 290K. Les propriétés des deux composants sont présentés tableau 3.1. Avec cela, on obtient une capacité thermique pour le mélange Eau-Ammoniac de $c_p = 4191.8$ J.kg^{-1}.K^{-1}.

Il nous manque cependant toujours un nombre de Prandtl pour cette mixture, et ils y a quelques sérieux raccourcis pris dans cette analyse. Je continue mes recherches sur le sujet, et dans DAVIS et al. 2010, je trouve que les auteurs ont résumés les propriétés thermodynamiques de la cryolave ET de l'atmosphère de Titan afin d'étudier un modèle de refroidissement de la cryolave sur Titan. Cet article va me permettre de compléter mes données et de vérifier celles que j'avais déjà. Ils utilisent pour leurs modèles une cryolave à 32.1% d'ammoniac, ce qui est suffisamment proche de la nôtre pour que l'on puisse

	Paramètre		Unité	Valeur
Ammoniac	Densité	ρ	kg/m^3	730.94
	Capacité thermique	c_p	$J.kg^{-1}.K^{-1}$	4226.6
Eau	Densité	ρ	kg/m^3	1000.0
	apacité thermique	c_p	$J.kg^{-1}.K^{-1}$	4179.6

TABLE 3.1 – *Propriétés de l'ammoniac (200K, 1bar) et de l'eau (290K, 1bar)*

	Paramètre		Unité	Valeur
Cryolave	Densité	ρ [b]	kg/m^3	946
	Capacité thermique spécifique	c_p [a]	$J.kg^{-1}.K^{-1}$	3150
	Viscosité cinématique	ν [b]	m^2/s	4.2e-3
	Nombre de Prandtl	Pr [c]		12600
Atmosphère	Densité	ρ [a]	kg/m^3	5.2371
	Capacité thermique spécifique	c_p [a]	$J.kg^{-1}.K^{-1}$	1024.101
	Viscosité cinématique	ν [a]	m^2/s	1.291e-6
	Nombre de Prandtl	Pr [c]		7.698e-1
	Coefficient d'expansion thermique	β [a]	$1/K$	9.8522e-3

TABLE 3.2 – *Propriétés de la cryolave et de l'atmosphère sur Titan.*
[a] *: selon* DAVIS *et al. 2010.*
[b] *: selon* KARGEL *1994.*
[c] *: calculé*

réutiliser ces résultats. Les données pour la cryolave et pour l'atmosphère dans cet article sont indiquées dans le tableau 3.2.

On peut commencer par remarquer que les approximations faites sur l'atmosphère, bien que grossières sur certains points, sont au même ordre de grandeur que les valeurs utilisées par DAVIS et al. 2010. Pour ce qui est de la cryolave, j'ai dans les deux cas utilisées des données provenant de KARGEL 1994, donc hormis la capacité thermique spécifique et le nombre de Prandtl, peu de choses changent, et la capacité thermique, bien que différente, reste au même ordre de grandeur. Cette différence peut s'expliquer par plusieurs choses : dans la première approxiamtion, j'ai utilisé les capacités thermiques de l'eau et de l'ammoniac pour des cconditons p-T différentes. De plus, utiliser la formule 3.15 revient à faire une sorte de moyenne de la capacité thermique de chaque composant du mélange, mettant donc à part toute interaction entre les deux. L'application qui est faite dans H MOHD RAZIF et al. 2015 est une application industrielle, donc dans des condition p-T particulières, et avec un bien meilleur contrôle sur la composition du mélange. Sur Titan, le mélange se fait à l'état naturel, sous la surface, et d'autres composants (par exemple du méthane) peuvent se mêler à la mixture, sans parler de la potentielle formation de cristaux dans la cryolave.

Maintenant que l'on a bien défini les paramètres physiques de notre problème, nous pouvons faire tourner une simulation pour voir le résultat. Concernant la loi que doit suivre la viscosité, bien que la cryolave soit non-newtonienne, KARGEL et al. 1991 précise que, pour une fraction d'ammoniac dans le mélange supérieure à 29%, l'approximation newtonienne renvoie de bon résultats. Notre mélange eau-ammoniac contient 32% d'ammoniac, et comme nous ne savons pas quels sont le taux de déformation et la puissance n à utiliser, nous partirons donc sur un liquide newtonien, dont la viscosité dépend seulement de la température.

On se permettra également de négliger la tension de surface, étant donné l'étendue de l'écoulement et le nombre de Reynolds particulièrement bas pour la cryolave. L'atmosphère sera pris en tant que fluide newtonien. Pour la dépendance en température de la cryolave, on se servira de la viscosité donnée dans le tableau 3.2 comme viscosité de référence. Cependant notre modèle de viscosité demande une viscosité **dynamique** de référence, il faut donc celle à partir de laquelle ν a été calculée : $\mu = 4$ kg.m^{-1}.s^{-1}.

Je présente en figure 3.7 le résultat de la simulation avec les paramètres physiques ci-dessus. L'écoulement est étonnament assez peu visqueux, mais cela peut s'expliquer par le refroidissement lent (le liquide ne change pratiquement pas de température). Je parle de refroidissement lent relativement à la durée de l'écoulement. En effet, au vu des échelles utilisées ici, soit un écoulement observé sur une envergure de 80m (40m de chaque côté), pendant seulement deux minutes, n'est sans doute pas très représentatif d'un écoulement de l'envergure de Sotra Patera, soit une longueur de 30km, avec sûrement une durée bien plus longue pour arriver à l'état final.

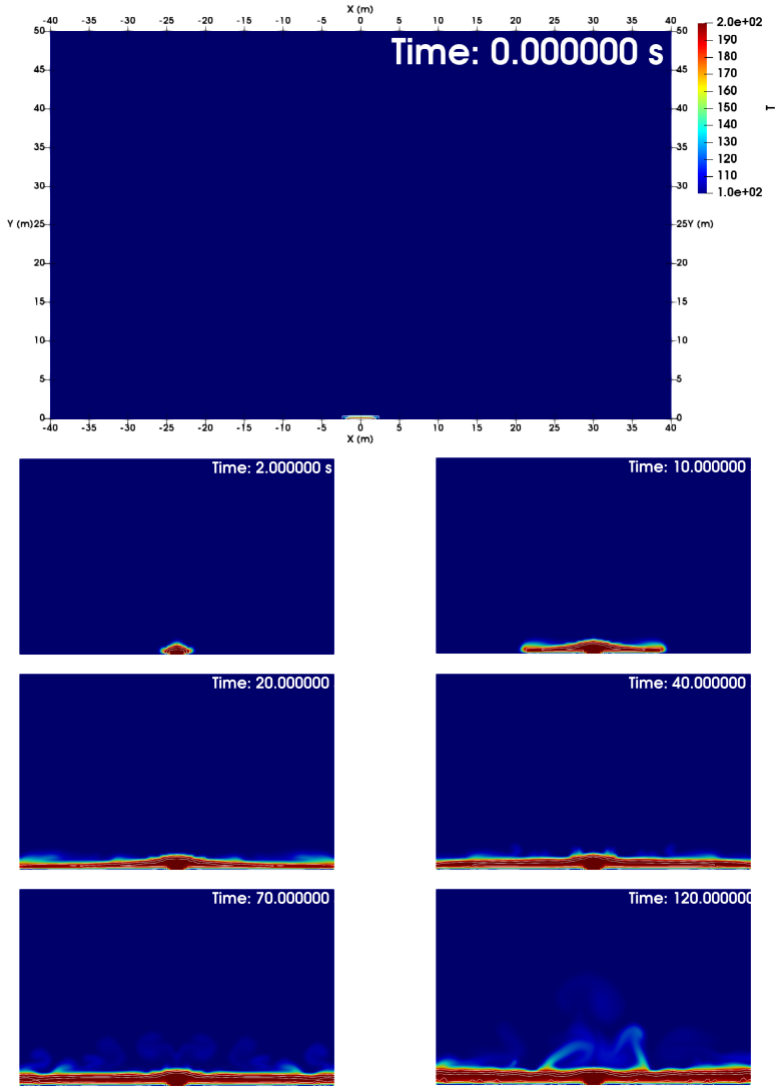

FIGURE 3.7 – *Simulation de l'écoulement d'un mélange eau-Ammoniac défini tel qu'expliqué dans la partie 3.6, sur une durée de 2 minutes. Les contours blancs suivent la valeur de la fraction de phase, de haut en bas : 0.1, 0.5 et 0.9. L'interface cryolave-atmosphère se trouve normalement proche de 0.5, et de toute manière entre 0.1 et 0.9.*

Chapitre 4

Conclusion

Notre modèle créé ici reste bien incomplet. Parmi les choses qu'il resterait à modéliser, la plus importante est sans doute le changement de phase, c'est-à-dire la solidification de la lave en refroidissant. Ce processus pourrait amener à des simulations plus poussés concernant la forme des coulées, mais également concernant un phénomène qui a lieu avec les laves terrestres : il peut arriver que de la lave solidifiée se reliquéfie lorsque de la lave chaude s'écoule à sa surface, épaississant considérablement la coulée de lave.

Il y a également le fait que la conduction dans le sol est encore mal modélisée. Le modèle utilisé ici fixe la température du sol, et les potentiels modèles utilisables demandent un coefficient de transfert thermiques, qu'à l'heure nous ne connaissons pas sur Titan pour ce cas. Il y a également la question du rayonnement : mon équation de la chaleur prend en compte la conduction, et l'approximation de Boussinesq permet de simuler la convection, cependant nous avons pour le moment ignoré le rayonnement. Ensuite, il est possible avec OpenFOAM de créer des mesh à géométrie complexe, notamment en important des fichiers topographiques avec certaines fonctions d'OpenFOAM. Cela pourrait être utilisé pour former un terrain similaire à celui de Sotra Patera (par exemple), et observer comment la cryolave s'écoule sur ce type de terrain. Malheureusement, je n'ai pas pu réussir cela pendant ce stage.

Enfin, l'idéal aurait été d'avoir un super-calculateur afin de faire tourner les modèles en 3D (ils mettaient bien trop de temps à tourner sur l'ordinateur fourni). Il y en avait un, malheureusement OpenFOAM était mal installé dessus, et une panne a empêché le problème d'être réglé.

Arrivé à ce point, un tel modèle pourrait avoir une large gamme d'application, que ce soit pour l'étude du cryovolcanisme sur Titan, mais également sur Pluton ou Encélade par exemple. Dans ces cas là en particulier, où l'atmosphère est ténue voire négligeable, implémenter le rayonnement prend tout son sens, étant donné qu'hormis l'interaction avec le sol, la cryolave n'aura plus d'atmosphère pour refroidir.

La cryolave mise à part, OpenFOAM s'avère être un outil d'une grande

polyvalence concernant la simulation numérique de mécanique des fluides. Je n'ai pas pu explorer toutes ses fonctionnalités, noamment en ce qui concerne la diversité des algorithmes pour les termes différentiels. Il serait judicieux de comparer les résultats, et voir lesquels sont les plus adaptés pour notre simulation.

Pour ne se limiter qu'à Titan, on pourrait se servir d'OpenFOAM pour simuler des océans et/ou des rivières de méthane. Bien qu'ici je me sois concentré sur la création de la simulation, OpenFOAM et l'outil de visualisation Paraview, utilisé pour visualiser mes simulations, possèdent tous deux de nombreux outils de post-processing et d'analyses des résultats, ce qui permettrait d'étudier plus en détail la physique derrière un modèle.

Annexe A

Exemple d'un cas OpenFOAM à deux phases : damBreak

Dans cette annexe, je présente en détail les fichiers utilisés par OpenFOAM pour le cas du damBreak. Il s'agit d'un cas en deux dimensions. Tous ce qui se trouve après "//" sur une ligne ou entre "/*" et "texttt*/" sont considérés comme des commentaires et donc pas pris en compte par OpenFOAM (format de commentaire C++).

La création de mesh avec blockMesh

Nous avons tout d'abord le fichier permettant la création de mesh : blockMeshDict. Le contenu du fichier est visible ci-dessous :

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version: v1812                                  |
|   \\  /    A nd           | Web:     www.OpenFOAM.com                       |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

scale   0.146;

vertices
(
    (0 0 0)             //0
    (2 0 0)             //1
```

```
    (2.16438 0 0)         //2
    (4 0 0)               //3
    (0 0.32876 0)         //4
    (2 0.32876 0)         //5
    (2.16438 0.32876 0)   //6
    (4 0.32876 0)         //7
    (0 4 0)               //8
    (2 4 0)               //9
    (2.16438 4 0)         //10
    (4 4 0)               //11

    (0 0 0.1)             //12
    (2 0 0.1)             //13
    (2.16438 0 0.1)       //14
    (4 0 0.1)             //15
    (0 0.32876 0.1)       //16
    (2 0.32876 0.1)       //17
    (2.16438 0.32876 0.1) //18
    (4 0.32876 0.1)       //19
    (0 4 0.1)             //20
    (2 4 0.1)             //21
    (2.16438 4 0.1)       //22
    (4 4 0.1)             //23
);

blocks
(
    hex (0 1 5 4 12 13 17 16) (23 8 1) simpleGrading (1 1 1)  //1
    hex (2 3 7 6 14 15 19 18) (19 8 1) simpleGrading (1 1 1)  //2
    hex (4 5 9 8 16 17 21 20) (23 42 1) simpleGrading (1 1 1) //3
    hex (5 6 10 9 17 18 22 21) (4 42 1) simpleGrading (1 1 1) //4
    hex (6 7 11 10 18 19 23 22) (19 42 1) simpleGrading (1 1 1) //5
);

edges
(
);

boundary
(
    leftWall
    {
        type wall;
        faces
        (
            (0 12 16 4)
            (4 16 20 8)
        );
    }
    rightWall
    {
        type wall;
        faces
        (
            (7 19 15 3)
            (11 23 19 7)
        );
```

```
    }
    lowerWall
    {
        type wall;
        faces
        (
            (0 1 13 12)
            (1 5 17 13)
            (5 6 18 17)
            (2 14 18 6)
            (2 3 15 14)
        );
    }
    atmosphere
    {
        type patch;
        faces
        (
            (8 20 21 9)
            (9 21 22 10)
            (10 22 23 11)
        );
    }
);

mergePatchPairs
(
);

// ************************************************************************* //
```

Regardons ce qui compose ce fichier :

— Chaque fichier du cas à traiter possède un header OpenFOAM renseignant notamment la version, ainsi qu'un dictionnaire FoamFile donnant quelques infos sur le fichier. Ils sont juste là à titre informatif.

— Les sommets, qui sont la base de la grille, sont donnés en unités arbitraires. Afin de savoir quelles dimensions aura notre domaine de travail, on multiplie chaque composante x, y et z des sommets par le coefficient `scale`, qui est typiquement en mètres. On peut voir également dans les anciennes versions la variable `convertToMeter`.

— La liste `vertices` liste les points qui pourront éventuellement servir de sommets à un ou plusieurs blocs. Ils sont définis par leurs coordonées (x y z), et sont utilisés par la suite en appelant leur numéro, dépendant de leur ordre dans la liste, et commençant à 0.

— La liste `blocks` contient la définition des blocs du domaine. `hex` indique que les blocs seront hexahédraux ; la liste qui suit définit les sommets délimitant les blocs ; la liste à 3 nombres indique le nombre de cellules que l'on souhaite dans chaque direction x, y et z. Mettre 1 cellule pour l'une de ses dimensions permet de la supprimer, se ramenant ici à un

cas à deux dimensions. Le paramètre `simpleGrading` suivi d'un liste à 3 chiffres permet de définir un gradient de la taille des arêtes vers chaque dimension. le nombre dans la liste correspond au rapport de taille entre la première et la dernière arête sur l'axe correspondant. Un ratio de 1 donne donc des arêtes de taille égale sur tout l'axe.

— La liste `edges` sert à définir les options de courbures des arêtes si nécessaires. Par défaut, les arêtes sont des droites.

— La liste `boundary` sert à définir quelles vont être les limites du domaine. C'est une liste de dictionnaires où chaque dictionnaire définit une condition limite. Chaque condition limite à un nom que vous lui choisissez, un type, et une liste de faces correspondant à cette limite. Le type patch est le type par défaut, laissant un choix libre à l'utilisateur. Le type wall est restreint aux conditions limites que l'on rencontre à un mur. Il était possible ici, sans plus de précision, de rassembler "leftWall", "rightWall" et "lowerWall" sous une même condition limite (appelée "walls" par exemple). Cependant, si l'on souhaite des murs avec différentes conditions limites, il faut les séparer. Ces conditions aux limites seront définies pour chaque variable du problème dans le répertoire `0/` du cas à traiter.

— La liste `mergePatchPairs` est utilisée pour fusionner les blocs entre eux, si ceux-ci n'utilisent pas des sommets identiques entre eux.

Constantes physiques dans transportProperties

Le fichier dans lequel est stocké les paramètres physiques des fluides considérés est appelé `transportProperties` :

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website: https://openfoam.org
    \\  /    A nd           | Version: 6
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

phases (water air);
```

```
water
{
    transportModel Newtonian;
    nu              1e-06;
    powerLawCoeffs
    {
        nuMax   [ 0 2 -1 0 0 0 0 ] 1e-03;
        nuMin   [ 0 2 -1 0 0 0 0 ] 1e-05;
        k       [ 0 2 -1 0 0 0 0 ] 1e-05;
        n       [ 0 0 0 0 0 0 0 ] 1;
    }
    rho             1000;
}

air
{
    transportModel Newtonian;
    nu              1.48e-05;
    rho             1;
}

sigma           0.07;

// ************************************************************************* //
```

— L'en-tête (limitée par "/*" et "*/"), ainsi que le dictionnaire `FoamFile` sont juste là à titre indicatif
— `phases (water air)` : indique le nom des différentes phases utilisées.
— `water {···}` et `air {···}` : dictionnaires indiquant les différentes propriétés respectivement de la phase `water` et de la phase `air`, ici la viscosité cinématique `nu` et la densité `rho`, ainsi que le modèle rhéologique utilisé, ici `Newtonian` pour les deux phases.
— Si le modèle rhéologique avait été `powerLaw` pour l'eau, le solveur aurait eu besoin du dictionnaire `powerLawCoeffs`, contenant ici les viscosités minimales et maximales `nuMin` et `nuMax`, une viscosité de référence `k`, ainsi que l'indice `n`.

Création du cas initial avec setFields

C'est dans le fichier setFieldsDict que l'on définit des régioons avec différentes propriétés (tout du moins si l'on ne souhaite pas définir les valeurs de chaque champ dans chaque cellule) :

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website: https://openfoam.org
    \\  /    A nd           | Version: 6
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version    2.0;
```

```
    format      ascii;
    class       dictionary;
    location    "system";
    object      setFieldsDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

defaultFieldValues
(
    volScalarFieldValue alpha.water 0
);

regions
(
    boxToCell
    {
        box (0 0 -1) (0.1461 0.292 1);
        fieldValues
        (
            volScalarFieldValue alpha.water 1
        );
    }
);


// ************************************************************************* //
```

— `defaultFieldsValues` : il s'agit d'une liste où on donne une valeur par défaut aux champs spécifiés (ici, la fraction de phase de l'eau), autrement dit, la valeur que prendra le champ si pas spécifié autrement dans la liste `regions` en dessous.

— `regions` : C'est une liste où on définit les différentes régions de notre mesh. Le dictionnaire `boxToCell` indique que l'on va former une "boîte" (un parallélépipède rectangle), limitée par les deux sommets `box`. Tout champ à modifier prendra la valeur définie dans `fieldValues`. Il existe bien sûr d'autres types de régions, par exemple `sphereToCell` définit une région sphérique via le centre et le rayon fourni.

On peut remarquer que la boîte est définie en z de -1 à 1, alors que dans `blockMeshDict`, on peut voir que les sommets ne vont que de 0 à 0.1. On peut supposer que c'est pour éviter les effets de bord : "**box To Cell**" laisse penser que le programme va transcrire du mieux qu'il peut la région donnée dans la grille crée précédemment. Aussi, faire dépasser la boîte permet d'éviter toute anomalie en bord de figure, surtout si le programme utilise des flottants, dont la valeur est toujours précise à ϵ près en informatique.

États initiaux et conditions limites des variables

Ces fichiers sont situés dans le répertoire inital de l'exécution, en général le répertoire `0/` :

— `dimensions` indique la dimension du champ de variable considéré. C'est une fonctionnalité d'OpenFOAM, qui permet entre autres d'éviter les

erreurs d'homogénéité dans les équations. Les 7 dimensions avec l'unité utilisée sont, de gauche à droite :
  - La masse (kg)
  - La distance (m)
  - Le temps (s)
  - La température (K)
  - La quantité de matière (mol)
  - Le courant (A)
  - L'intensité lumineuse (cd)
- `internalField` fournit une valeur par défaut à l'ensemble des cellules, ainsi qu'aux faces de cellules n'appartenant pas aux bords, c'est à dire aux limites de la mesh.
- `boundaryField` : C'est ici que l'on définit les conditions limites pour tous les bords définis dans `blockMeshDict`.

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website: https://openfoam.org
    \\  /    A nd           | Version: 6
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    location    "0";
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

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

internalField  uniform (0 0 0);

boundaryField
{
    leftWall
    {
        type           noSlip;
    }
    rightWall
    {
        type           noSlip;
    }
    lowerWall
    {
        type           noSlip;
    }
    atmosphere
    {
        type           pressureInletOutletVelocity;
        value          uniform (0 0 0);
```

```
    }
    defaultFaces
    {
        type            empty;
    }
}


// ************************************************************************* //
```

— `noSlip` : Applique une vélocité nulle aux limites.
— `pressureInletOuteltVelocity` : Applique la vitesse `value` si le fluide à une vitesse dirigée vers l'intérieur du domaine, applique un gradient zéro sinon. Cette vitesse sera utilisée pour calculer la pression si spécifié par les conditions de pression.

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website: https://openfoam.org
    \\  /    A nd           | Version: 6
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p_rgh;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [1 -1 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    leftWall
    {
        type            fixedFluxPressure;
        value           uniform 0;
    }

    rightWall
    {
        type            fixedFluxPressure;
        value           uniform 0;
    }

    lowerWall
    {
        type            fixedFluxPressure;
        value           uniform 0;
    }
```

```
    atmosphere
    {
        type            totalPressure;
        p0              uniform 0;
    }

    defaultFaces
    {
        type            empty;
    }
}

// ************************************************************************* //
```

— `fixedFluxPressure` : Fixe le gradient de pression à la valeur spécifiée par `value`.

— `totalPressure` : La pression sera calculée avec la pression totale p_0 et la vitesse U selon la formule suivante : $p = p_0 - 0.5|U|$, avec les pression en $m^2.s^{-2}$. Il s'agit de la formule dans le cas d'un fluide incompressible et subsonique. La formule est amenée à changer en fonction de la physique considérée.

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website: https://openfoam.org
    \\  /    A nd           | Version: 6
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      alpha.water;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    leftWall
    {
        type            zeroGradient;
    }

    rightWall
    {
        type            zeroGradient;
    }

    lowerWall
```

```
    {
        type            zeroGradient;
    }

    atmosphere
    {
        type            inletOutlet;
        inletValue      uniform 0;
        value           uniform 0;
    }

    defaultFaces
    {
        type            empty;
    }
}

// ************************************************************************* //
```

— `zeroGradient` : applique un gradient zéro au bord.
— `inletOutlet` : applique un gradient zéro si le fluide sort du domaine et la valeur `inletValue` si le fluide rentre. La valeur `value` n'est pas utilisée mais doit tout de même être rentrée. Il s'agit d'une condition que l'on met en général lorsqu'on veut que le fluide puisse sortir librement du domaine.

Ici, mettre une condition `zeroGradient` à la place de `inletOutlet` avec l'inlet 0 comporte un risque majeur (auquel j'ai été confronté) : si, pendant un pas de temps, la vitesse est tournée vers le centre du domaine et, à cause d'un artefact de la simulation par exemple, la valeur de alpha passe à 1 au bord, on se retrouve avec une quantité indéfinie de liquide qui se déverse dans notre espace de travail, ce qui, en plus de "casser" les paramètres de notre simulation, a tendance à s'accompagner de divergence du champ de vitesse.

Paramètres de simulation dans controlDict

Dans ce fichier, on défini les paramètres globaux de la simulation.

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website: https://openfoam.org
    \\  /    A nd           | Version: 6
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

```
application     interFoam;

startFrom       startTime;

startTime       0;

stopAt          endTime;

endTime         1;

deltaT          0.001;

adjustTimeStep yes;

maxCo           1;
maxAlphaCo      1;

maxDeltaT       1;

writeControl    adjustableRunTime;

writeInterval 0.05;

purgeWrite      0;

writeFormat     binary;

writePrecision 6;

writeCompression off;

timeFormat      general;

timePrecision 6;

runTimeModifiable yes;


// ************************************************************************* //
```

— `application` définit l'application qui sera utilisée dans le problème. On l'appelle explicitement de toute manière lors du lancement, mais je suppose que c'est pour éviter d'utiliser le mauvais solveur, notamment pour ce qui est des cas tutoriels fournis avec OpenFOAM.

— `startFrom` définit où est-ce que l'on commence. Ici, on commence au temps spécifié par `startTime`, mais il est également possible de commencé au pas de temps le plus ancien (`firstTime`) ou le plus récent (`latestTime`).

— `stopAt` donne la conditon d'arrêt. Ici, on s'arrête au temps désigné par `endTime`, mais on peut également choisir de n'éxécuter qu'un pas de temps (et écrire ou non le résultat après ce pas de temps), voire de faire tourner la simulation jusqu'au prochain pas de temps d'écriture, définit par `writeControl`.

— `deltaT` donne le pas de temps.
— `adjustTimeStep` précise si le pas de temps doit être changé en cours de simulation pour maintenir un nombre de Courant inférieur à `maxCo` pour les différents champs. `maxAlphaCo` est la limite pour le champs de la fraction de phase en particulier.
— `maxDeltaT` définit la valeur maximale que le pas de temps peut prendre. Si l'option `adjustTimeStep` est activée, le solveur choisira toujours le plus grand pas de temps qui satisfait la condition Co < MaxCo, quand bien même ce pas de temps serait supérieur à la durée de la simulation. `maxDeltaT` permet d'éviter ce genre de problème.
— `writeControl` définit le pas de temps d'écriture en utilisant `writeInterval`. On peut choisir d'écrire tous les n pas de temps (`timestep`), ou bien toutes les n secondes de temps simulé (`runTime`). Ici, `adjustableRunTime` décrit le même comportement que `runTime`, en précisant en plus que le pas de temps pouvant changer, il faut qu'il s'adapte au pas de temps d'écriture.
— `purgeWrite` donne le nombre de répertoire de pas de temps (`0/, 0.1/, 0.2/,` etc) que le solveur doit écrire avant de commencer à écraser les plus anciens. 0 équivaut à pas de limite.
— `writeFormat` donne le format d'écriture (ASCII ou binaire). Binaire prend moins d'espace, mais ASCII permet de rendre le document directement lisible, sans utiliser de logiciels de post-processing.
— `writePrecision` donne la précsision dans laquelle doit être donnée les flottants si les fichiers sont écrits en ASCII.
— `writeCompression` indique si les données écrites aux pas de temps d'écriture doivent être compressées ou non (si oui, c'est au format gzip).
— `timeFormat` donne le format que doit prendre le nom des répertoires de temps : `fixed` écrit normalement ($\pm$`m.dddddd`), `scientific` écrit en écriture scientifique ($\pm$`m.dddddde`$\pm$`xx`), et `general` écrit scientifique seulement si l'exposant est inférieur à -4 ou supérieur ou égal à `timePrecision`. Dans tous les cas, le nombre de décimales dans le nom est le plus petit possible, et est limité par `timePrecision`.
— `runTimeModifiable` indique si oui ou non le fichier **controDict** est relu à chaque pas de temps par le solveur. Indiquer oui permet d'effectuer des mdofications sans arrêter la simulation, par exemple si l'on souhaite modifier le temps final, ou bien qu'on souhaite changer le pas de temps d'écriture, entre autres.

Schémas numériques et paramètres de résolution

C'est dans ces deux fichiers que seront définis les schémas numériques à appliquer à chaque terme différentiels (fvSchemes), ainsi que les solveurs, algorithmes et tolérances à utiliser pour chaque champ calculé (fvSolution).

Je n'ai malheureusement pas eu le temps de m'intéresser en détail aux différentes options proposées pour ces paramètres, aussi la description que je vais fournir sera au mieux succinte concernant les paramètres utilisés dans le fichier

présenté.

fvSchemes

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website: https://openfoam.org
    \\  /    A nd           | Version: 6
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

ddtSchemes
{
    default         Euler;
}

gradSchemes
{
    default         Gauss linear;
}

divSchemes
{
    div(rhoPhi,U) Gauss linearUpwind grad(U);
    div(phi,alpha) Gauss vanLeer;
    div(phirb,alpha) Gauss linear;
    div(((rho*nuEff)*dev2(T(grad(U))))) Gauss linear;
}

laplacianSchemes
{
    default         Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}


// ************************************************************************* //
```

— `ddtSchemes` va définir comment les dérivées temporelles vont être traitées. Ici, elles seront traitées par défaut avec une méthode d'Euler. Il est possible si souhaité de spécifier un schéma numérique pour un terme en particulier en plus du schéma par défaut. Parmi les options proposées pour la dérivée temporelle, il y a les méthode d'Euler, Euler implicite, Crank-Nicolson, et *steady state*, qui met simplement la dérivée à 0 (état stationnaire).
— `gradSchemes`, `divSchemes` et `laplacianSchemes` décrivent les schémas numériques à appliquer respectivement pour les termes de gradient, de divergence, et les termes laplaciens. On notera juste ici que tous les schémas sont basés sur une méthode de quadrature de Gauss (`Gauss` précisé devant tous les termes), et que le terme venant après *Gauss* correspond à un limiteur de flux (ou limiteur de pente), une fonction ayant pour but de limiter les oscillations (`linearUpwind, vanLeer`, etc). Ces derniers sont nécessaire à cause de la discrétisation qui, notamment lors d'un choc, d'une discontinuité où simplement de forts gradient, peuvent faire apparaître des oscillations.
— Comme expliqué dans la partie sur la Méthode des Volumes Finis, des interpolations ont lieu, et le schéma numérique à appliquer est donné dans `interpolationSchemes`.
— `snGradSchemes` correspond au schéma numérique à appliquer pour calculer le gradient normal de surface. C'est un gradient qui est calculé non pas au centroïde de la cellule, mais sur une face entre 2 cellules : il correspond à la composante du gradient entre les deux centroïdes des cellules reliées par la face, et qui est normale à cette face.

fvSolution

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website: https://openfoam.org
    \\  /    A nd           | Version: 6
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
```

```
    "alpha.water.*"
    {
        nAlphaCorr      2;
        nAlphaSubCycles 1;
        cAlpha          1;

        MULESCorr       yes;
        nLimiterIter    5;

        solver          smoothSolver;
        smoother        symGaussSeidel;
        tolerance       1e-8;
        relTol          0;
    }

    "pcorr.*"
    {
        solver          PCG;
        preconditioner DIC;
        tolerance       1e-5;
        relTol          0;
    }

    p_rgh
    {
        solver          PCG;
        preconditioner DIC;
        tolerance       1e-07;
        relTol          0.05;
    }

    p_rghFinal
    {
        $p_rgh;
        relTol          0;
    }

    U
    {
        solver          smoothSolver;
        smoother        symGaussSeidel;
        tolerance       1e-06;
        relTol          0;
    }
}

PIMPLE
{
    momentumPredictor no;
    nOuterCorrectors  1;
    nCorrectors       3;
    nNonOrthogonalCorrectors 0;
}

relaxationFactors
{
    equations
```

```
    {
        ".*" 1;
    }
}

// ************************************************************************* //
```

— le dictionnaire `solvers` indique quels sont les solveurs linéaires qui vont être utilisés pour chaque équation. Ce dictionnaire contient des mots clés représentant les différentes variables du problème (`p_rgh, U, alpha.water`). Chaque variable réfère à l'équation où est calculée cette variable, et est ici le nom d'un sous-dictionnaire dans lequel vont être précisés le solveur à utiliser pour cette équation et les paramètres de ce solveur. Il est à noter que les solveurs diffèrent en fonction de la symétrie des matrices de l'équation. Par exemple, les dérivées temporelles forment des matrices symétriques, mais les termes advectifs sont en général asymétriques.

— Les systèmes d'équations non-linéaires sont résolus avec des solveurs qui laissent un résidu, qui est en gros une mesure de l'erreur. Il s'agit de la différence entre les membres de gauche et de droite de l'équation avec la solution actuelle La plupart des solveurs sont itératifs : ils vont réduire petit à petit le résidu, jusqu'à ce qu'il arrive en-dessous de la tolérance, précisée par `tolerance`, ou bien si le rapport entre le résidu actuel et le résidu initial tombe en-dessous de `relTol`. On peut aussi fixer un nombre maximal d'itérations.

— Le conditionnement représente la dépendance d'une variable d'un problème aux paramètres du problème. Plus cette dépendance sera haute, plus le calcul numérique pour résoudre le problème sera compliqué et long. Aussi, il est possible d'appliquer un préconditionneur (via le paramètre `preconditioner`), qui est basiquement une matrice que l'on va multiplier au système d'équations, et qui a pour but de réduire le conditionnement, ce qui réduira le temps de calcul.

— Certains solveurs utilisent un `smoother`, autrement dit...

— `relaxationFactors` est un dictionnaire qui contient des facteurs non pas de relaxation (ce qui peut faire référence aux méthodes numériques de relaxation), mais de sous-relaxation ("*under-relaxation*"). Ce sont des facteurs affectant la convergence du problème. On peut donner un facteur par équation (en les désignant par leur nom de variable, comme `p_rgh` ou `U`). Plus ces facteurs sont grands, plus le système aura tendance à converger. Ils ont cependant tendance à augmenter significativement la durée de calcul, il est donc préférable de ne les utiliser que si le problème a du mal à converger, et de ne pas les prendre trop grand.

— Un dernier dictionnaire est présent dans ce fichier, qui portera le nom de l'algorithme utilisé pour résoudre l'équation pression-vitesse, soit

PISO [1], SIMPLE [2] ou PIMPLE [3]. Ils ont pour rôle notamment de coupler les équations entre elles, et de s'assurer de la conservation de la masse et de la quantité de mouvement. SIMPLE est utilisé pour les problème à état stationnaire, les deux autres sur les problèmes à état transitif.

- `nCorrectors` : utilisé par PISO et PIMPLE, il indique combien de fois l'algorithme doit résoudre l'équation de la pression et la correction de la quantité de mouvement à chaque pas.
- `nNonOrthogonalCorrectors` : utilisé par tous les algorithmes, il indique le nombre de corrections non-orthogonales à appliquer aux dérivées. En théorie, pour une grille orthogonale, il peut être mis à 0. Dans le cas d'un problème transitif, il est en général mis à 1.
- `nOuterCorrectors` : utilisé par PIMPLE, il donne le nombre de fois que l'algorithme se répète sur tout le pas de temps (donc sur tout le système d'équations), typiquement mis à 1.
- `momentumPredictor` indique si oui ou non l'algorithme doit résoudre l'équation de quantité de mouvement. Il est typiquement mis sur `off` pour les problèmes à petit nombre de Reynolds et les problèmes à plusieurs phases.

— Pour certains problèmes, typiquement les cas incompressibles et fermés, la pression est relative, il faut donc rentrer une pression de référence dans le sous-dictionnaire SIMPLE/PISO/PIMPLE.

1. Pressure-Implicit Split-Operator
2. Semi-Implicit Method for Pressure-Linked Equations
3. une combinaison de SIMPLE et PISO

Annexe B

Les solveurs dans OpenFOAM : exemple de mon solveur myLavaInterFoam

Je présente ici la structure d'un solveur d'OpenFOAM, en prenant pour exemple le solveur que j'ai créé en partant du solveur interFoam : **myLavaInterFoam**. Je vais ici présenter quelques aspects de la structure du code, sans pour autant m'attarder sur tous les détails, notamment car je n'ai pas cherché à comprendre en détail ce qui n'était pas nécessaire sur le moment pour la création de mon solveur, par manque de temps.

B.0.1 Initialisation des variables dans createFields

Le fichier suivant est celui où les différents champs et les différentes variables sont initialisés.

```
#include "createRDeltaT.H"

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

```
);

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

//- Adding thermophysical properties ----------------------

Info<< "Reading field T\n" << endl;

volScalarField T
(
    IOobject
    (
        "T",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

//- ------------------------------------------------------

#include "createPhi.H"


Info<< "Reading transportProperties\n" << endl;
immiscibleIncompressibleTwoPhaseMixture mixture(U, phi);


volScalarField& alpha1(mixture.alpha1());
volScalarField& alpha2(mixture.alpha2());

const dimensionedScalar& rho1c = mixture.rho1();
Info<< "mixture rho1 = " << rho1c << "\n" << endl ;
const dimensionedScalar& rho2c = mixture.rho2();
Info<< "mixture rho2 = " << rho2c << "\n" << endl ;

//- Adding thermophysical properties ----------------------

const dimensionedScalar& cp1 = mixture.cp1();
Info<< "mixture cp1 = " << cp1 << "\n" << endl ;

const dimensionedScalar& cp2 = mixture.cp2();
Info<< "mixture cp2 = " << cp2 << "\n" << endl ;
```

```
const dimensionedScalar& beta2 = mixture.beta2();
Info<< "mixture beta2 = " << beta2 << "\n" << endl ;

const dimensionedScalar& Tref2 = mixture.Tref2();
Info<< "mixture Tref2 = " << Tref2 << "\n" << endl ;

const dimensionedScalar& beta1 = mixture.beta1();
Info<< "mixture beta1 = " << beta1 << "\n" << endl ;

const dimensionedScalar& Tref1 = mixture.Tref1();
Info<< "mixture Tref1 = " << Tref1 << "\n" << endl ;



volScalarField rho1
(
    IOobject
    (
        "rho1",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    alpha1*rho1c
);

volScalarField rho2
(
    IOobject
    (
        "rho2",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    alpha2*rho2c
);

//- ------------------------------------------------------

// Need to store rho for ddt(rho, U)
volScalarField rho
(
    IOobject
    (
        "rho",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    // alpha1*rho1 + alpha2*rho2
    rho1 + rho2
```

```
);
//rho.oldTime();


// Mass flux
surfaceScalarField rhoPhi
(
    IOobject
    (
        "rhoPhi",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    fvc::interpolate(rho)*phi
);


// Construct incompressible turbulence model
autoPtr<incompressible::turbulenceModel> turbulence
(
    incompressible::turbulenceModel::New(U, phi, mixture)
);


#include "readGravitationalAcceleration.H"
#include "readhRef.H"
#include "gh.H"


volScalarField p
(
    IOobject
    (
        "p",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    p_rgh + rho*gh
);

label pRefCell = 0;
scalar pRefValue = 0.0;
setRefCell
(
    p,
    p_rgh,
    pimple.dict(),
    pRefCell,
    pRefValue
);

if (p_rgh.needReference())
{
```

```
    p += dimensionedScalar
    (
        "p",
        p.dimensions(),
        pRefValue - getRefCellValue(p, pRefCell)
    );
    p_rgh = p - rho*gh;
}

mesh.setFluxRequired(p_rgh.name());
mesh.setFluxRequired(alpha1.name());

//- Adding thermophysical properties --------------------------

Info<< "Reading/calculating rho*cp\n" << endl;
volScalarField rhoCp
(
    IOobject
    (
        "rho*Cp",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    alpha1*rho1c*cp1 + alpha2*rho2c*cp2, //Boussinesq : rho*cp constant
    alpha1.boundaryField().types()
);
rhoCp.oldTime();

Info<<"Reading / calculating rho*phi*cp\n" << endl;
surfaceScalarField rhoPhiCpf
(
    IOobject
    (
        "rho*phi*cpf",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    rhoPhi*cp1
);

//- ------------------------------------------------------------


#include "createMRF.H"
#include "createFvOptions.H"
```

— Les différents fichiers inclus dans createFields.H par une commande `include` sont simplement des déclarations supplémentaires, posées autre part dans l'arborescence car pouvant être utilisées par d'autres solveurs (que ce soit le champ ϕ, la fraction de phase ou bien encore les variables liées au termes gravitationnel).

— Il y a différents types de champs : scalaire ou vectoriels, volumiques

ou surfaciques... Par exemple, le champ de vitesse est un champ vectoriel volumique (`volScalarField`), alors que le champ du flux de masse est un champ scalaire surfacique (`surfaceScalarField`). Ces champs sont des `IOobject`, autrement ils peuvent être lus et/ou écrits lors de la simulation. Après avoir donné le nom qui sera donné au fichier (éventuel) dans lequel l'objet sera lu/écrit, le paramètre `runTime.timeName()` indique le répertoire où il sera créé (ici le répertoire du pas de temps courant), il indique que c'est un champ (avec `mesh`, je suppose que c'est pour gérer le format d'écriture), et enfin on peut préciser si l'on souhaite qu'il l'écrive et/ou le lise au cours de la simulation. Après cela, on peut donner une valeur initiale au champ (si non, on met `mesh` à la place).

— L'objet `mixture` est un objet tiré du modèle créé en amont de ce solveur. Il va contenir différents paramètres relatifs aux deux fluides utilisés, et déclarés via références dans les lignes suivantes.

— Les commandes `Info << ...` sont l'équivalent de la commande `cout << ...` en C++.

On peut reconnaître ici certains détails que j'ai décrit dans 3.3 et 3.4 concernant la déclaration de certaines variables, comme les champs de densité `rho1`, `rho2` et `rho`, ou les références vers les paramètres thermodynamiques (`T, beta`, etc) que je déclare en amont dans le modèle.

B.0.2 Implémentation des équations

Les différentes classes implémentées dans OpenFOAM permettent d'écrire des équations dans une forme assez simple. Par exemple, l'équation suivante :

$$\frac{\partial}{\partial t}(\rho\vec{U}) + \vec{\nabla}\cdot(\phi\vec{U}) - \mu\nabla^2\vec{U} = -\vec{\nabla}p$$

sera représentée par par le code suivant :

```
solve
(
    fvm::ddt(rho,U)
  + fvm::div(phi,U)
  - fvm::laplacian(mu,U)
 ==
  - fvc::grad(p)
);
```

Il s'agit ici du cas le plus basique d'implémentation de Navier-Stokes. Au fur et à mesure de la complexification des solveurs (problèmes transitifs, ajout d'une ou plusieurs phases, multiplication des champs de variables, fonctionnalités supplémentaires), d'autres termes vont être rajoutés à l'équation, qui ont pour but en général d'aider à la convergence, d'augmenter la précision de la solution, etc.

Equation de Navier-Stokes

```
MRF.correctBoundaryVelocity(U);

fvVectorMatrix UEqn
(
    fvm::ddt(rho, U)
  + fvm::div(rhoPhi, U)
  - fvm::Sp(fvc::ddt(rho) + fvc::div(rhoPhi), U) //ligne rajoute en
      s'inspirant de compressibleInterFoam
  + MRF.DDt(rho, U)
  + turbulence->divDevRhoReff(rho, U)
 ==
    fvOptions(rho, U)
);

UEqn.relax();

fvOptions.constrain(UEqn);

if (pimple.momentumPredictor())
{
    solve
    (
        UEqn
     ==
        fvc::reconstruct
        (
            (
                mixture.surfaceTensionForce()
              - ghf*fvc::snGrad(rho)
              - fvc::snGrad(p_rgh)
            ) * mesh.magSf()
        )
    );

    fvOptions.correct(U);
}
```

On peut noter plusieurs choses :

— Il y a deux types de termes différentiels : `fvm` et `fvc`. Les termes en `fvm` sont implicites, autrement dit, une matrice va être créée pour les décrire. Les termes en `fvc` sont explicites, c'est-à-dire qu'ils vont simplement renvoyer une valeur. L'idée est d'écrire les termes que l'on ne connait pas en implicite, et ceux qu'on connait en explicite. Par exemple, dans l'équation de Navier-Stokes, on cherche à connaître U, on mettra donc les termes différentiels avec U en implicite. Pour résoudre l'équation, on utilisera une dérivée explicite de p, entre autre.

— En plus des termes classiques des équation (`ddt` pour $\frac{\partial}{\partial t}$, `div` pour divergence, etc), on peut noter que d'autres termes apparaissent : `fvm::Sp`, `MRF.DDT`, ou encore `turbulence->divDevRhoReff` et `fvOptions`. Ce sont des termes qui je suppose fournissent une correction numérique à la résolution de l'équation, afin d'éviter des divergences ou des croissances d'erreurs. Je ne peux malheureusement pas dire en détail à quoi ces

termes servent, néanmoins certains de ces termes sont nécessaires au bon fonctionnement du solveur. En effet, prenons par exemple le terme `- fvm::Sp(...)` dans l'équation de Navier-Stokes, que j'ai rajouté manuellement en le recopiant à partir d'un autre solveur plus complexe, qui notamment prenait en compte les variations de température. Sans ce terme, si mon solveur fonctionnait sans problème (autrement dit, transfert de chaleur et convection) pour deux fluides au repos (voir figure 3.6), les problèmes commençait dès que le liquide présentait le moindre mouvement (même une petite vague) : le champ de vitesse dans l'air se mettait soudainement à diverger après 2 ou 3 pas de temps, sans raison apparente, et le résultat était une simulation extrêmement longue (pas de temps réduit à la microseconde pour maintenir le nombre de Courant inférieur à 1) en plus d'être incorrecte, et cela si le solveur ne me renvoyait pas une erreur `Floating Point Exception` avant la fin. C'est seulement en rajoutant ce terme que que les divergences ont disparu. Il s'est avéré que ce terme servait à rendre implicite le terme source de l'équation, afin qu'il contribue plus à la diagonale de la matrice, ce qui a pour effet d'aider à la convergence.

Equation de la chaleur

L'équation de la chaleur reste assez simple) lire. Elle est posée dans le foirmat que l'équation de Navier-Stokes, et comprend moins de termes. On notera l'ajout de la correction de la densité en fonction de la température comme décrit dans la partie 3.4, qui se ffait après résolution de l'équation de la chaleur.

```
surfaceScalarField kappaf = mixture.kappaf();

fvScalarMatrix TEqn
(
     fvm::ddt(rhoCp,T)
   + fvm::div(rhoPhiCpf,T)
   - fvm::laplacian(kappaf,T)
   ==
     fvOptions(rhoCp, T)
);

TEqn.relax();

fvOptions.constrain(TEqn);

TEqn.solve();

fvOptions.correct(T);

rho1 = alpha1*rho1c*(1. - beta1*(T - Tref1));
rho2 = alpha2*rho2c*(1. - beta2*(T - Tref2));

rho = rho1 + rho2;
```

Corrections des différents champs et calcul de la pression

Dans ce fichier, diverses corrections des champs sont effectuées, et la pression y est mise à jour.Vu que je n'ai pas eu à modifier le fichier de base déjà présent dans interFoam, je n'ai pas étudié en détail son contenu.

```
{
    if (correctPhi)
    {
        rAU.ref() = 1.0/UEqn.A();
    }
    else
    {
        rAU = 1.0/UEqn.A();
    }

    surfaceScalarField rAUf("rAUf", fvc::interpolate(rAU()));
    volVectorField HbyA(constrainHbyA(rAU()*UEqn.H(), U, p_rgh));
    surfaceScalarField phiHbyA
    (
        "phiHbyA",
        fvc::flux(HbyA)
      + MRF.zeroFilter(fvc::interpolate(rho*rAU())*fvc::ddtCorr(U, phi, Uf))
    );
    MRF.makeRelative(phiHbyA);

    if (p_rgh.needReference())
    {
        fvc::makeRelative(phiHbyA, U);
        adjustPhi(phiHbyA, U, p_rgh);
        fvc::makeAbsolute(phiHbyA, U);
    }

    surfaceScalarField phig
    (
        (
            mixture.surfaceTensionForce()
          - ghf*fvc::snGrad(rho)
        )*rAUf*mesh.magSf()
    );

    phiHbyA += phig;

    // Update the pressure BCs to ensure flux consistency
    constrainPressure(p_rgh, U, phiHbyA, rAUf, MRF);

    while (pimple.correctNonOrthogonal())
    {
        fvScalarMatrix p_rghEqn
        (
            fvm::laplacian(rAUf, p_rgh) == fvc::div(phiHbyA)
        );

        p_rghEqn.setReference(pRefCell, getRefCellValue(p_rgh, pRefCell));

        p_rghEqn.solve(mesh.solver(p_rgh.select(pimple.finalInnerIter())));
```

```
        if (pimple.finalNonOrthogonalIter())
        {
            phi = phiHbyA - p_rghEqn.flux();

            p_rgh.relax();

            U = HbyA + rAU()*fvc::reconstruct((phig - p_rghEqn.flux())/rAUf);
            U.correctBoundaryConditions();
            fvOptions.correct(U);
        }
    }

    #include "continuityErrs.H"

    // Correct Uf if the mesh is moving
    fvc::correctUf(Uf, U, phi);

    // Make the fluxes relative to the mesh motion
    fvc::makeRelative(phi, U);

    p == p_rgh + rho*gh;

    if (p_rgh.needReference())
    {
        p += dimensionedScalar
        (
            "p",
            p.dimensions(),
            pRefValue - getRefCellValue(p, pRefCell)
        );
        p_rgh = p - rho*gh;
    }

    if (!correctPhi)
    {
        rAU.clear();
    }
}
```

B.0.3 Corps principal du solveur

C'est dans ce fichier que l'on appelle toutes les librairies nécessaires, ainsi que les autres fichiers présentés précédemment, et que la boucle sur le temps va être effectuée.

```
#include "fvCFD.H"
#include "dynamicFvMesh.H"
#include "CMULES.H"
#include "EulerDdtScheme.H"
#include "localEulerDdtScheme.H"
#include "CrankNicolsonDdtScheme.H"
#include "subCycle.H"
#include "immiscibleIncompressibleTwoPhaseMixture.H"
#include "turbulentTransportModel.H"
#include "pimpleControl.H"
```

```
#include "fvOptions.H"
#include "CorrectPhi.H"
#include "fvcSmooth.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

int main(int argc, char *argv[])
{
    #include "postProcess.H"

    #include "setRootCaseLists.H"
    #include "createTime.H"
    #include "createDynamicFvMesh.H"
    #include "initContinuityErrs.H"
    #include "createDyMControls.H"

    #include "createFields.H"
    #include "createAlphaFluxes.H"
    #include "initCorrectPhi.H"
    #include "createUfIfPresent.H"

    turbulence->validate();

    if (!LTS)
    {
        #include "CourantNo.H"
        #include "setInitialDeltaT.H"
    }
    Info<< "\nArriving to time loop\n" << endl;
    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
    Info<< "\nStarting time loop\n" << endl;

    while (runTime.run())
    {
        Info<< "\nRead readDyMControls.H\n" << endl;
        #include "readDyMControls.H"

        if (LTS)
        {
            Info<< "\nRead setRDeltaT.H\n" << endl;
            #include "setRDeltaT.H"
        }
        else
        {
            Info<< "\nRead CourantNo.H\n" << endl;
            #include "CourantNo.H"
            Info<< "\nRead alphaCourantNo.H\n" << endl;
            #include "alphaCourantNo.H"
            Info<< "\nRead setDeltaT.H\n" << endl;
            #include "setDeltaT.H"
        }

        Info<< "\nincrement runTime\n" << endl;
        runTime++;

        Info<< "Time = " << runTime.timeName() << nl << endl;
```

```
// --- Pressure-velocity PIMPLE corrector loop
while (pimple.loop())
{
    if (pimple.firstIter() || moveMeshOuterCorrectors)
    {
        mesh.update();

        if (mesh.changing())
        {
            // Do not apply previous time-step mesh compression flux
            // if the mesh topology changed
            if (mesh.topoChanging())
            {
                talphaPhi1Corr0.clear();
            }

            gh = (g & mesh.C()) - ghRef;
            ghf = (g & mesh.Cf()) - ghRef;

            MRF.update();

            if (correctPhi)
            {
                // Calculate absolute flux
                // from the mapped surface velocity
                phi = mesh.Sf() & Uf();

                #include "correctPhi.H"

                // Make the flux relative to the mesh motion
                fvc::makeRelative(phi, U);

                mixture.correct();
            }

            if (checkMeshCourantNo)
            {
                #include "meshCourantNo.H"
            }
        }
    }

    #include "alphaControls.H"
    #include "alphaEqnSubCycle.H"

    mixture.correct();

    #include "UEqn.H"

    //- Adding thermophysical properties
    #include "TEqn.H"
    //- --------------------------------

    // --- Pressure corrector loop
    while (pimple.correct())
    {
        #include "pEqn.H"
```

```
            }

            if (pimple.turbCorr())
            {
                turbulence->correct();
            }
        }

        runTime.write();

        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }

    Info<< "End\n" << endl;

    return 0;
}
// ************************************************************************* //
```

— Tout ce qui se situe avant la fonction `main` correspond à des librairies qui vont être utilisées pour la résolution, que ce soit des librairies de discrétisation du pas de temps (*EulerDdtScheme.H, localEulerDdtScheme.H* ou bien *CrankNicolsonDdtScheme.H*), ou bien différents modèles pour les aspects physiques, comme la turbulence ou le multiphase. On notera que plusieurs schémas numériques pour le temps sont inclus par défaut ici, car le choix du schéma à appliquer se fait a posteriori dans le cas à traiter.

— Tous les `#include` situées dans la fonction `main` sont des morceaux de codes, mis à part car utilisés par plusieurs solveurs ou modèles, et qui vont servir pour certains à initialiser des constantes, des champs, des modèles, des classes, des objets, etc., qui vont être utilisés plus tard pendant la simulation. D'autres, comme UEqn.H, sont des équations qui vont être résolues, ou des calculs à effectuer avant, pendant ou après la simulation.

— `runTime.run()` est une fonction qui regarde si le pas de temps courant est supérieur ou non au pas de temps final, et qui renverra True tant que ça n'est pas le cas. Plus bas dans le code, `runTime++` permet d'incrémenter le temps. On notera l'existence de `runTime.loop()`, qui fait la même chose que `run()`, mais qui en plus incrémente le temps. Ici, on ne peut pas l'utiliser à cause de fonctions à appliquer avant l'incrémentation du temps, entre autres la mise à jour du pas de temps en fonction du nombre de Courant.

— Comme dit dans l'annexe A, un alogrithme va être chargé de résoudre l'équation pression/vitesse, et de corriger la pression. Ici, il s'agit de PIMPLE, et des corrections sur différents termes du problème sont appliquées avant (`while(pimple.loop()){...}`) et après (`while(pimple.correct()){...}`).

Annexe C

Erreurs commises lors de la création de mesh

Dans cette partie, je vais exposer deux différentes mesh que j'ai essayé avant d'en arriver à une qui fonctionne pour la source de liquide depuis le sol.

On peut voir figure C.1 le premier cas d'erreur à éviter : j'avais ici défini deux blocs avec des cellules assez petites, car c'est là que le liquide évoluerait principalement (blocs 1 et 2), et deux autres blocs au dessus un peu plus grossiers. Le cinquième bloc est défini afin d'avoir une face faisant office de source sur le sol, cependant on peut voir que je n'avais pas fait attention sur le moment à l'alignement des cellules, aussi je me suis retrouvé avec un problème de dépassement. Comme expliqué dans la partie 2.1.1, le non-dépassement des celules entre elles est l'une des bases fondamentales de la méthode des volumes finis sur laquelle se base OpenFOAM, aussi il était donc pratiquement impossible que cela fonctionne correctement.

Le solveur n'a pas renvoyé d'erreur, mais lors de la simulation, bien que liquide rentrait bien par la source définie dans le sol, il est resté dans la colonne centrale, sans s'étaler sur les côtés. Apparemment, afin de palier au problème de dépassement, le solveur à simplement décidé de considérer la colonne centrale et le reste des blocs comme indépendant l'un de l'autre.

La figure C.2 présente ma deuxième tentative de créer une mesh avec rentrée du liquide. Cette fois-ci, j'ai fait attention à ne pas avoir de dépassement entre les cellules. Cela devait à priori marcher, cependant il s'est avéré que lors de la simulation, le liquide était vite dispersé à cause de divergences dans le champ de vitesse. De plus, Ces anomalies n'apparaissait que dans les trois blocs inférieurs (1, 2 et 3). Ma théorie est que cette fois-ci c'est le dépassement des faces des blocs qui pose problème : on peut voir que le bloc 4 partage une de ses faces avec les 3 autres blocs. Bien que cela ne va pas en contradiction avec l'aspect théorique et la méthode des volumes finis, il est possible qu'OpenFOAM n'arrivve pas à gérer ce dépassement. Cela a jusque là été vérifié par

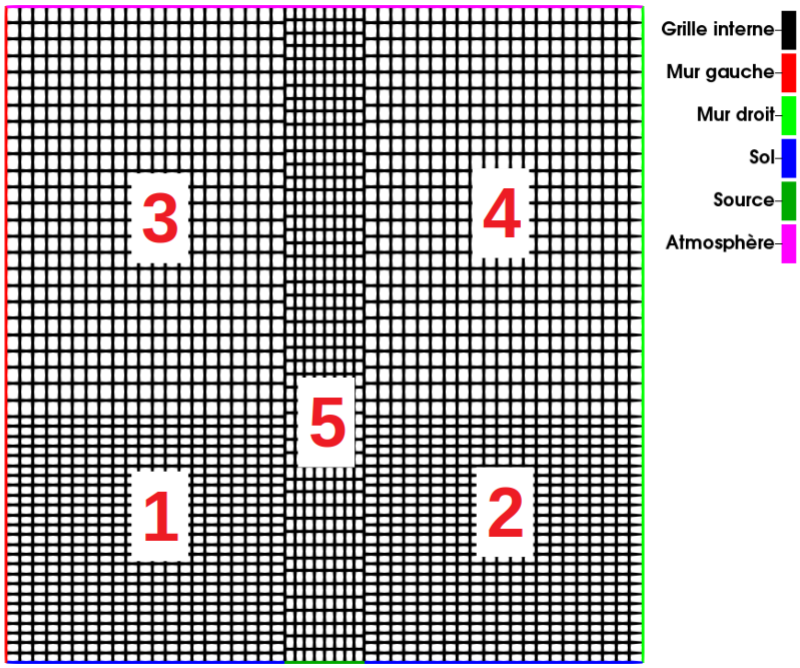

FIGURE C.1 – *Première mesh incorrecte créée avec une source de liquide. Elle est constituée de 5 blocs différents. Il y a dépassement des cellules du bloc 5 par rapport aux autres blocs.*

d'autres tests que j'ai faits pour d'autres problèmes.

Ces deux cas permettent de présenter quelques problèmes que j'ai pu rencontrer lors de la création de la mesh pour ma simulation.

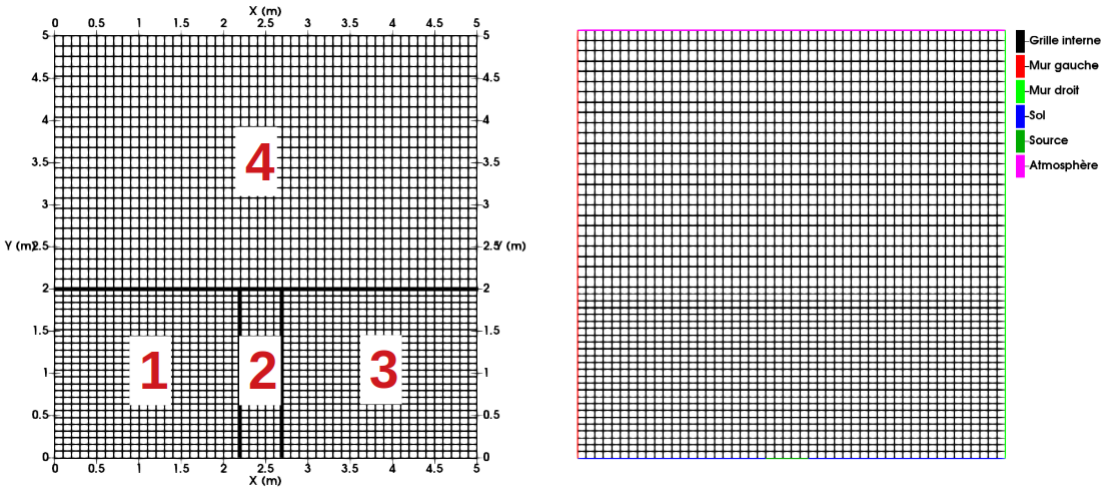

FIGURE C.2 – *Deuxième tentative de créer une mesh d'input. Ici, 4 blocs sont créés : 3 pour le sol et la source, qui sont assez affinés, car c'est là qu'évoluera principalement le liquide ; et 1 au-dessus des 3 autres, avec une mesh plus grossière, étant donné que seul de l'air y circulera (qui à ce niveau d'avancement dans les travaux ne nous intéressait que très peu)*

Bibliographie

(p.d.). URL : http://www.peacesoftware.de/einigewerte/stickstoff_e.html. (Dernière consultation : 14/06/2019).

CAMPBELL, Donald B. et al. (2003). « Radar Evidence for Liquid Surfaces on Titan ». In : *Science* 302.5644, p. 431–434. DOI : 10.1126/science.1088969.

DAVIS, Ashley Gerard et al. (2010). « Atmospheric control of the cooling rate of impact melts and cryolavas on Titan's surface ». In : *Icarus* 208, p. 889–895. DOI : https://doi.org/10.1016/j.icarus.2010.02.025.

Engineering ToolBox (p.d.). URL : https://www.engineeringtoolbox.com.

FERZIGER, Joel H. et Milovan PERIĆ (2002). *Computational Methods for Fluid Dynamics*. ISBN : 3-540-42074-6.

H MOHD RAZIF, N et al. (2015). « Thermophysical properties analysis for ammonia-water mixture of an organic Rankine cycle ». In : *Jurnal Teknologi* 75, p. 13–17. DOI : 10.11113/jt.v75.5203.

KARGEL, Jeffrey S. (1994). « Cryovolcanism On The Icy Satellites ». In : *Earth, Moon, and Planets* 67, p. 101–113. DOI : https://doi.org/10.1007/BF00613296.

KARGEL, Jeffrey S. et al. (1991). « Rheological properties of ammonia-water liquids and crystal-liquid slurries: Planetological applications ». In : *Icarus* 89, p. 93–112. DOI : https://doi.org/10.1016/0019-1035(91)90090-G.

MÁRQUEZ DAMIÁN, Santiago (2013). « An Extended Mixture Model for the Simultaneous Treatment of Short and Long Scale Interfaces ». Thèse de doct. Universidad Nacional Del Litoral.

MONAGHANN, Joseph J. (1992). « Smooth Particle Hydrodynamics ». In : *Annual Review of Astronomy and Astrophysics* 30, p. 543–574. DOI : https://doi.org/10.1146/annurev.aa.30.090192.002551.

MUHLEMAN, D. O. et al. (1990). « Radar Reflectivity of Titan ». In : *Science* 248.4958, p. 975–980. DOI : 10.1126/science.248.4958.975.