# Méthode des éléments finis pour les équations de Navier et Stokes via leur approximation par les caractéristiques

GREBOT JÉRÉMY

29 janvier 2019

# Chapitre 1

# Avant propos physique et approximation en temps de Navier-Stokes

## Préambule

Le rapport commence véritablement avec la section 1.3. Dans les 1.1 à 1.2 il s'agit mécanique des fluides, ou comment deux principes de conservations se transposent en équation, ces deux sections ont donc leur véritable place en appendice.

## 1.1 Les deux principes fondamentaux : Conservation de la masse et de la quantité de mouvement

### 1.1.1 Premier principe : La masse

Introduisons les notations suivantes :

| | |
|---|---|
| $\rho(t,x)$ | la **densité** au point $x=(x_1,x_2,x_3)$ et au temps $t$. |
| $u(t,x)=(u_i(t,x))_{1\leq i\leq 2}$ | la **vitesse** au point $x$ et au temps $t$. |

Soit $\Omega \subset \mathbb{R}^2$ le domaine ouvert borné occupé par le fluide et $V \subset \Omega$ un sous-domaine ouvert régulier quelconque. Le principe de conservation de la masse du liquide se traduit ainsi : la variation de la masse totale du fluide contenue dans $V$ à l'instant $t$ est égale au flux de masse entrant à travers $\partial V$ :

$$\frac{d}{dt}\left(\int_V \rho(t,x)\mathrm{d}x\right) = -\int_{\partial V} \rho(t,s)\,\mathbf{u}(t,s)\cdot\mathbf{n}(s)\mathrm{d}s$$

Où $\mathbf{n}$ est la normale unitaire sortante à $V$ sur $\partial V$. Après permutation de la dérivation en temps et de l'intégrale dans le terme de gauche et après l'utilisation de la formule de Stokes à la fonction $\rho(t,\cdot)u(t,\cdot)$ sur le terme de droite. On obtient :

$$\int_V \left( \frac{d\rho}{dt} + \mathrm{div}(\rho \mathbf{u}) \right) = 0$$

$$\left( \text{Formule de Stokes} : \int_V \mathrm{div}(\mathbf{u}) = \int_{\partial V} \mathbf{u} \cdot \mathbf{n} \right) \tag{1.1}$$

Cette formule étant vraie pour tout voisinage $V$ d'un point quelconque $x$ d'un ouvert de $\Omega$, nous obtenons une expression globale de la conservation de la masse.

$$\frac{d\rho}{dt} + \mathrm{div}(\rho \mathbf{u}) = 0 \quad \text{dans } ]0,T[\times\Omega \tag{1.2}$$

Nous avons ainsi retrouvé l'équation de la chaleur à partir du principe de conservation de la masse.

### 1.1.2 Deuxième principe : la quantité de mouvement

Ce principe s'énonce ainsi : la masse multipliée par l'accélération est égale à l'ensemble des forces exercées, et ce dans tout volume $V \subset \Omega$.

***L'accélération :***

Soit une particule en $x$ au temps $t$, cette particule sera en $x + \delta t\, u(t,x) + \mathcal{O}(\delta t^2)$ à l'instant $t + \delta t$. Son accélération est donc :

$$\begin{aligned} \mathbf{a}(t,x) &= \lim_{\delta t \to 0} \frac{\mathbf{u}\left(t + \delta t, x + \delta t\, \mathbf{u}(t,x) + \mathcal{O}(\delta t^2)\right) - \mathbf{u}(t,x)}{\delta t} \\ &= \left( \frac{d\mathbf{u}}{dt} + \mathbf{u} \cdot \nabla \mathbf{u} \right)(t,x) \end{aligned}$$

où $\mathbf{u} \cdot \nabla \mathbf{u}$ est un vecteur :

$$(\mathbf{u} \cdot \nabla \mathbf{u})_{1 \le i \le 3} = \sum_{j=1}^{3} u_j \frac{\partial u_i}{\partial x_j}, \quad 1 \le i \le 3$$

***Les forces exercées :***

Elles sont deux types : *Les forces externes dues à la gravité* :

$$\int_V \rho(t,x) \mathbf{g}(x) \mathrm{d}x$$

où $\mathbf{g}$ désigne le vecteur gravité, supposé ici constant (nous négligerons la force de Coriolis et les forces dues aux effets magnétiques) et *les forces internes dues aux déformations du fluide* :

$$\int_{\partial V} \sigma_{tot}(t,s)\mathbf{n}(s)\mathrm{d}s$$

où $\sigma_{tot}$ désigne le tenseur symétrique des contraintes totales.

Le produit tenseur-vecteur $\sigma_{tot}\mathbf{n}$ est un champ de vecteur de composante :

$$(\sigma_{tot}\mathbf{n})_i = \sum_{j=1}^{3} \sigma_{ij} n_j \;\; 1 \leq i \leq 3$$

D'après le principe de la conservation de la quantité de mouvement nous obtenons l'égalité suivante :

$$\int_V \rho(t,x)\left(\frac{d\mathbf{u}}{dt} + \mathbf{u}\cdot\nabla\mathbf{u}\right)(t,x)\mathrm{d}x = \int_{\partial V} \sigma_{tot}(t,s)\mathbf{n}(s)\mathrm{d}s + \int_V \rho(t,x)\mathbf{g}(x)\mathrm{d}x$$

Introduisons la divergence d'un tenseur $\tau$, un vecteur à trois composante, notée en gras $\mathbf{div}(\tau)$ :

$$\mathbf{div}(\tau) = \left(\sum_{j=1}^{3} \frac{\tau_{ij}}{x_j}\right)_{1\leq i\leq 3}$$

En appliquant la formule de Stokes suivante $\int_{\partial V} \tau\mathbf{n} = \int_V \mathbf{div}(\tau) \;\; \forall \tau \in \left(\mathcal{H}^1(V)\right)^{2\times 2}$, la conservation de la quantité de mouvement devient :

$$\int_V \left(\rho\left(\frac{d\mathbf{u}}{dt} + \mathbf{u}\cdot\nabla\mathbf{u}\right) - \mathbf{div}(\sigma_{tot}) - \rho\mathbf{g}\right) = 0$$

Cette formule étant vraie pour tout voisinage $V$ d'un point quelconque $x$ d'un ouvert de $\Omega$, nous obtenons une expression globale de la conservation de la quantité de mouvement.

$$\rho\left(\frac{d\mathbf{u}}{dt} + \mathbf{u}\cdot\nabla\mathbf{u}\right) - \mathbf{div}(\sigma_{tot}) = \rho\mathbf{g} \;\; \text{dans } ]0,\mathrm{T}[\times\Omega \tag{1.3}$$

Nous avons ainsi deux équations de conservations (1.2) et (1.3) et trois inconnues $u$, $\sigma_{tot}$ et $\rho$. Ces deux équations sont complétées par la loi de comportement qui lie $u$ et $\sigma_{tot}$.

## 1.2 Loi de comportement

Nous commençons par introduire quelques définitions sur les tenseurs :

- La **trace** d'un tenseur est $\mathrm{tr}(\tau) = \sum_{i=1}^{3} \tau_{ii}$.
- La **partie sphérique** d'un tenseur est $\frac{1}{3}\mathrm{tr}(\tau)I$ , où $I$ est le tenseur identité : $I = (\delta_{ij})_{1\leq i,j\leq 3}$ et $\delta_{ij}$ le symbole de Kronecker.

— La **partie déviatrice** est $\tau - \frac{1}{3}\mathrm{tr}(\tau)I$, dans le cas du tenseur $\sigma_{tot}$ elle est notée $\sigma$.
— Le champs de **pression** d'un fluide est défini à partir du tenseur des contraintes totales : $\mathbf{p} = \frac{1}{3}\mathrm{tr}(\sigma_{tot})$

Nous avons la décomposition du tenseur des contraintes totales en partie sphérique et déviatrice, ce qui s'écrit à l'aide de la pression et de sa partie déviatrice ainsi :

$$\sigma_{tot} = -\mathbf{p}I + \sigma \tag{1.4}$$

La loi de comportement exprime une relation entre la partie déviatrice du tenseur des contraintes totales et le tenseur des gradients de vitesse, noté $\nabla\mathbf{u}$ et défini par :

$$(\nabla\mathbf{u})_{ij} = \frac{\partial u_j}{\partial x_j} \quad 1 \leq i, j \leq 3$$

Le tenseur des gradients de vitesse est formellement le jacobien du champs de vecteur $\mathbf{u}$.

Dans le cas d'un **fluide Newtonien** la loi de comportement est linéaire :

$$\sigma = 2\eta D(\mathbf{u}) - \frac{2\eta}{3}\mathrm{div}(\mathbf{u})I \tag{1.5}$$

où $\eta$ est une constante positive appelée la **viscosité** et $D(\mathbf{u})$ est la partie symétrique du tenseur de gradient des vitesses : $D(\mathbf{u}) = \frac{\nabla\mathbf{u}+\nabla^T\mathbf{u}}{2}$.

Un fluide est dit non-Newtonien si la loi de comportement entre $\sigma\, et\, \mathbf{u}$ est non linéaire. Le sang, les sables mouvants ou la mayonnaise sont des exemples de fluides non-Newtoniens.

## 1.3 Hypothèse d'incompressibilité

Résumons, nous avons 4 équations (1.2) - (1.5) et 5 inconnues : $\sigma_{tot}, \sigma, \mathbf{u}, \mathbf{p}\, et\, \rho$.
Dans le cas de fluide compressible, une seconde loi de comportement met en relation $\mathbf{p}$ et $\rho$. Cependant dans le cas de l'eau, la variation de la densité $\rho$ est extrêmement faible, et par la suite, nous allons supposer $\rho$ constante. Cette hypothèse se justifie également dans le cas de l'air à faible vitesse, et plus généralement dans tous les cas où nous sommes en présence d'un fluide incompressible.

Les conséquences de cette hypothèse pour nos équations sont multiples :
— La conservation de la masse (1.2) devient la **relation d'incompressibilité** :

$$\mathrm{div}\,\mathbf{u} = 0 \ \text{ dans } ]0, \mathrm{T}[\times\Omega \tag{1.6}$$

— La loi de comportement devient :

$$\sigma = 2\eta D(\mathbf{u})$$

— A l'aide de la décomposition du tenseur des contraintes totales (1.4), la conservation de la quantité de mouvement (1.3) devient :

$$\rho\left(\frac{d\mathbf{u}}{dt}+\mathbf{u}\cdot\nabla\mathbf{u}\right)-\mathbf{div}(2\eta D(\mathbf{u}))+\nabla p=\rho\mathbf{g}\ \text{ dans } ]0,\mathrm{T}[\times\Omega \tag{1.7}$$

Vu l'identité $\mathbf{div}(2\eta D(\mathbf{u}))=\eta\Delta\mathbf{u}+\eta\nabla\left(\mathrm{div}(\mathbf{u})\right)$ et (1.6) ; on obtient :

$$\rho\left(\frac{d\mathbf{u}}{dt}+\mathbf{u}\cdot\nabla\mathbf{u}\right)-\eta\Delta\mathbf{u}+\nabla p=\rho\mathbf{g}\ \text{ dans } ]0,\mathrm{T}[\times\Omega \tag{1.8}$$

Les équations (1.6) et (1.7) sont les équations de Navier-Stokes pour un fluide incompressible. Notre objectif est de calculer numériquement une solution dans le cas d'un domaine donné.

## 1.4 Le problème de Navier-Stokes bien posé

Un problème est dit bien posé s'il existe pour celui-ci une unique solution. Dans le cas des équations de Navier-Stokes, deux restrictions sur la solution sont requises.

### 1.4.1 Conditions

Il est nécessaire d'ajouter deux conditions :

— Une condition initiale :

$$\mathbf{u}(t=0,x)=\mathbf{u}_0(x)\quad\text{pour presque tout } x\in\Omega$$

— Une condition sur le bord :

$$\mathbf{u}(t,s)=\mathbf{u}_\Gamma(s)\ \text{ pour presque tout } s\in\partial\Omega,\ t\in]0,\mathrm{T}[$$

où $\mathbf{u}_0$ et $\mathbf{u}_\Gamma$ sont données.

Pour que $\mathrm{div}(\mathbf{u}(t,\cdot))$ soit continu en $t\to 0$, il est nécessaire que la donnée initiale $\mathbf{u}_0$ soit à divergence nulle, ce qui (par la formule de Stokes) implique sur la condition de bord :

$$\int_{\partial\Omega}\mathbf{u}_\Gamma\cdot n=0$$

On peut saisir cette condition sur $\mathbf{u}_\Gamma$ comme une conséquence de l'hypothèse d'incompressibilité : en présence d'un fluide incompressible, les flux entrants et les flux sortant doivent être opposés l'un l'autre.

**Remarque**

Afin de coïncider avec l'énoncé de notre projet, nous supposons que le liquide étudié est de l'eau, *ie* $\rho = 1$. Et nous négligeons les forces de la gravité.

### 1.4.2 Le problème de Navier-Stokes :

*(N-S) Trouver* $\mathbf{u}$ *et* $p$ *tel que :*

$$\begin{aligned} \frac{d\mathbf{u}}{dt} + \mathbf{u}\cdot\nabla\mathbf{u} - \eta\Delta\mathbf{u} + \nabla p &= 0 && \text{sur } ]0,\mathrm{T}[\times\Omega \\ \mathrm{div}(\mathbf{u}) &= 0 && \text{sur } ]0,\mathrm{T}[\times\Omega \\ \mathbf{u}(0,\cdot) &= \mathbf{u}_0 && \text{sur } \Omega \\ \mathbf{u} &= \mathbf{u}_\Gamma && \text{sur } ]0,\mathrm{T}[\times\partial\Omega \end{aligned}$$

## 1.5 Méthode des caractéristiques pour le problème de Navier-Stokes

Soit $N$ un entier, partitionnons l'intervalle $[0,\mathrm{T}]$ en $N$ sous-intervalles $[t_n, t_{n+1}]$, où $t_n = n\Delta t$, $0 \le n \le N$ et où $\Delta t = \mathrm{T}/N$ est le pas temps.

La **dérivée totale** d'une fonction $\phi : [0,T]\times\mathbb{R}^2 \to \mathbb{R}^2$ est :

$$\frac{D\phi}{Dt} = \frac{\partial\phi}{\partial t} + (\mathbf{u}\cdot\nabla)\,\phi$$

Notons $\chi^{(n)}(x)$ la position de la particule au temps $t_n$ qui est en $x$ au temps $t_{n+1}$. Approchons la dérivée totale de $\phi$ au temps $t_{n+1}$ par cette expression de type différences finies :

$$\frac{D\phi}{Dt}(t_{n+1},x) = \frac{\phi(t_{n+1},x) - \phi(t_n,\chi^{(n)}(x))}{\Delta t} + \mathcal{O}(\Delta t)$$

Vu que la dérivée totale du champ des vitesses $\mathbf{u}$ apparaît dans l'équation (1.7), nous pouvons approximer ce terme par une expression de type différences finies. Utilisons un schéma d'Euler implicite d'ordre un pour approcher en temps notre problème :

$$\begin{aligned} \frac{\mathcal{Y}^{(n+1)} - \mathcal{Y}^{(n)}\circ\chi^{(n)}}{\Delta t} + \mathcal{F}\left(\mathcal{Y}^{(n+1)}\right) &= 0 \\ \mathcal{Y}^{(0)} &= \mathcal{Y}_0 \end{aligned}$$

avec

$$\begin{aligned} \mathcal{Y} &= \begin{pmatrix} \mathbf{u} \\ p \end{pmatrix} & \mathcal{Y}_0 &= \begin{pmatrix} \mathbf{u}_0 \\ 0 \end{pmatrix} \\ \mathcal{F}(\mathcal{Y}) &= \begin{pmatrix} -\eta\Delta\mathbf{u} + \nabla p \\ -\mathrm{div}(\mathbf{u}) \end{pmatrix} \end{aligned}$$

Nous pouvons ainsi construire par récurrence une suite $\left(\mathbf{u}^{(n)}\right)_{1\leq n\leq N}$ (où $\mathbf{u}^{(n)}(x) \approx \mathbf{u}(t_n, x)$ est une approximation du champ des vitesses) à l'aide de l'algorithme qui suit :

**Initialisation :**

$$n = 0 \ : \ \mathbf{u}^0 = \mathbf{u}_0 \ \textit{est donné}$$

**Hérédité :**

$$n \geq 0 \ : \ \mathbf{u}^{(n)} \ \textit{étant connu, trouver} \ \mathbf{u}^{(n+1)} \ \textit{et} \ p^{(n+1)} \ \textit{tels que :}$$

$$\begin{aligned} \mathbf{u}^{(n+1)} - \eta\Delta t\, \Delta\mathbf{u}^{(n+1)} + \Delta t\, \nabla p(n+1) &= \mathbf{u}^{(n)} \circ X^{(n)} && \textit{dans } \Omega \\ - \mathrm{div}\left(\mathbf{u}^{(n+1)}\right) &= 0 && \textit{dans } \Omega \\ \mathbf{u}^{(n+1)} &= \mathbf{u}_\Gamma\left(t_{n+1}\right) && \textit{dans } \partial\Omega \end{aligned}$$

**Remarquons** que $\chi^{(n)}(\mathbf{x}) = \mathbf{x} - \Delta t \times \mathbf{u}^{(n)}(\mathbf{x}) + \mathcal{O}(\Delta t^2)$. Puisque $\mathbf{u}^{(n)}$ est connu à l'étape $n$, on peut utiliser l'approximation $X^{(n)} \approx \mathbf{x} - \Delta t \times \mathbf{u}^{(n)}(\mathbf{x})$ dans le calcul du terme source.

# Chapitre 2

# Résolution numérique du problème de Stokes

## Préambule

Ecrire le problème de Stokes et pk on doit le résoudre.

$$(S) \qquad \begin{array}{rcll} \mathbf{u} - \eta\Delta t\ \Delta\mathbf{u} + \Delta t\ \nabla p & = & f & \text{dans } \Omega \\ -\operatorname{div}(\mathbf{u}) & = & 0 & \text{dans } \Omega \\ \mathbf{u} & = & \mathbf{u}_\Gamma & \text{sur } \partial\Omega \end{array}$$

Où on a noté $f(x) := \mathbf{u}^{(old)}\left(x - \Delta t \times \mathbf{u}^{(old)}(x)\right) \quad \forall x \in \Omega$.
$\Omega$ est ici un ouvert borné de $\mathbb{R}^2$

## 2.1 Formulation variationnelle

### 2.1.1 Remarque sur la pression

Remarquons de suite que si $(u, p)$ est solution de $(S)$, alors $(u, p + c)$ est également solution, pour tout $c \in \mathbb{R}$. En effet $p$, qui n'apparaît que sous la forme d'un gradient, est seulement déterminé à une constante près. Nous n'avons pas unicité de la solution pour $p$.

### 2.1.2 "Multiplication" par une fonction test

Multiplions la deuxième équation de $(S)$ par une fonction test scalaire $q$ quelconque, puis en intégrant sur $\Omega$ on obtient :

$$\int_\Omega q \, \mathrm{div}(\mathbf{u}) = 0 \tag{2.5}$$

Multiplions (sous la forme du produit scalaire usuel de $\mathbb{R}^2$) la première équation de $(S)$ par une fonction vectorielle $\mathbf{v}$ quelconque, puis en intégrant sur $\Omega$ on obtient :

$$\int_\Omega \mathbf{u} \cdot \mathbf{v} - \eta \int_\Omega \Delta \mathbf{u} \cdot \mathbf{v} + \int_\Omega \nabla p \cdot \mathbf{v} = \int_\Omega \mathbf{f} \cdot \mathbf{v} \tag{2.6}$$

où $\cdot$ désigne le produit scalaire usuel de $\mathbb{R}^2$.

### 2.1.3 Le terme $\int_\Omega \nabla p \cdot \mathbf{v}$ :

D'après la formule de stokes (1.1), en remplaçant $\mathbf{v}$ par $q\mathbf{v}$, ($\mathbf{v}$ et $q$ fonctions tests respectivement vectorielle et scalaire) on obtient en développant une formule bien connu de Stokes pour la divergence div :

$$\int_\Omega \nabla q \cdot \mathbf{v} = -\int_\Omega q \, \mathrm{div}(\mathbf{v}) + \int_{\partial\Omega} q \, \mathbf{v} \cdot \mathbf{n} \tag{2.7}$$

Avec $q = p$, on obtient :

$$\int_\Omega \nabla p \cdot \mathbf{v} = -\int_\Omega p \, \mathrm{div}(\mathbf{v}) + \int_{\partial\Omega} p \, \mathbf{v} \cdot \mathbf{n} \tag{2.8}$$

### 2.1.4 Le terme : $-\eta \int_\Omega \Delta \mathbf{u} \cdot \mathbf{v}$ :

Remarquons que $\Delta \mathbf{u} \cdot \mathbf{v} = \Delta u_1 \times v_1 + \Delta u_2 \times v_2$ et que $\Delta u_1 = \mathrm{div}(\nabla u_1)$. Utilisons donc deux fois la formule de Stokes pour la divergence (2.7) :

$$-\eta \int_\Omega \Delta \mathbf{u} \cdot \mathbf{v} = \eta \int_\Omega (\nabla u_1 \nabla v_1 + \nabla u_2 \nabla v_2) - \eta \int_{\partial\Omega} (v_1 + v_2)(\nabla u_1 + \nabla u_2) \cdot \mathbf{n} \tag{2.9}$$

$$= \eta \int_\Omega \nabla \mathbf{u} \cdot \nabla \mathbf{v} - \eta \int_{\partial\Omega} (v_1 + v_2)(\nabla u_1 + \nabla u_2) \cdot \mathbf{n} \tag{2.10}$$

$v_1$ et $v_2$ doivent être choisies nulles sur le bord afin que le terme de bord s'annule.

## 2.2 Espaces appropriés

Dans cette section nous fixons la régularité requise aux inconnues ($\mathbf{u}$,$p$), aux fonctions tests ($\mathbf{v}$,$q$), sur le terme source $f$ et la condition de bord $\mathbf{u}_\Gamma$, pour que les intégrales des formules (2.5) à (2.10) aient un sens.

### 2.2.1 p et q :

Introduisons l'espace des fonctions de carré sommables :

$$L^2(\Omega) = \left\{ q \ : \ \Omega \to \mathbb{R} \mid \int_\Omega q^2 < +\infty \right\}$$

Vu que les dérivées de la pression $p$ ou de la fonction test $q$ n'apparaissent pas, $p, q \in L^2(\Omega)$ suffit. Mais afin d'unifier la pression nous fixons $\int_\Omega p = 0$. Donc $p, q \in L^2_0(\Omega)$, où :

$$L^2_0(\Omega) = \left\{ q \in L^2(\Omega) \mid \int_\Omega q = 0 \right\}$$

### 2.2.2 u et v :

Les inconnues $\mathbf{u}$ et les fonctions test $\mathbf{v}$ apparaissent avec leurs dérivées premières : $\nabla u_i$ et $\nabla v_i$ $i = 1, 2$ . Nous avons donc besoin de l'espace $H^1(\Omega)$ pour $u_1$, $u_2$, $v_1$, $v_2$ :

$$H^1(\Omega) = \left\{ \varphi \in L^2(\Omega) \mid \nabla\varphi \in L^2(\Omega)^2 \right\}$$

Le champ de vecteurs des vitesses $\mathbf{u}$ doit appartenir à un sous espace de $H^1(\Omega)^2$ pour lequel la condition de bord est satisfaite :

$$\mathbf{u} \in \left\{ \mathbf{w} \in H^1(\Omega)^2 \mid \mathbf{w} = \mathbf{u}_\Gamma \text{ sur } (\partial\Omega^2) \right\} := V_\Gamma$$

et la fonction test associée $\mathbf{v}$ doit appartenir à un sous espace de $H^1(\Omega)^2$ pour lequel la condition de bord homogène est satisfaite (comme indiqué à la fin de la section **2.1**) :

$$\mathbf{v} \in \left\{ \varphi \in H^1(\Omega)^2 \mid \varphi = 0 \text{ sur } (\partial\Omega)^2 \right\} := V_0$$

Vu que le sous-espace de $H^1(\Omega)$ satisfaisant la condition de bord homogène est noté $H^1_0(\Omega)$, on a :

$$\mathbf{v} \in V_0 = H^1_0(\Omega)^2$$

Ainsi sous ces conditions l'ensemble des intégrales apparaissant dans notre formulation variationnelle existent : chacune des intégrales considérées s'écrit sous la forme d'un produit scalaire qui, à l'aide de la formule de Cauchy-Schwarz correspondante, est bornée par les normes de $\mathbf{u}$, $\mathbf{v}$, $p$ ou $q$.

## 2.3 Formulation variationnelle complète :

Introduisons $a$ et $b$, deux formes bilinéaires et $l$ une forme linéaire par :

$$a(\mathbf{u}, \mathbf{v}) = \int_\Omega (\mathbf{u} \cdot \mathbf{v} + \eta \nabla\mathbf{u} \cdot \nabla\mathbf{v})$$

$$b(\mathbf{v}, p) = -\int_\Omega p \, \mathrm{div}(\mathbf{v})$$

$$l(\mathbf{v}) = \int_\Omega \mathbf{f} \cdot \mathbf{v}$$

et voici (à l'aide de (2.5)-(2.11) et des formes définies ci-dessus) la formulation variationnelle complète :

$(FVS)$ : *trouver* $\mathbf{u} \in V_\Gamma$ *et* $p \in L^2_0(\Omega)$ *tel que* :

$$\begin{array}{rcccll} a(\mathbf{u},\mathbf{v}) & + & b(\mathbf{v},p) & = & l(\mathbf{v}), & \forall \mathbf{v} \in V_0 \\ & & b(\mathbf{u},q) & = & 0, & \forall q \in L^2_0(\Omega). \end{array}$$

Pour résultat d'existence et d'unicité on pourra consulter V. Girault and P. A. Raviart. *Finite element methods for the Navier-Stokes equations. Theory and algorithms.* Springer, 1986. (p80)

## 2.4 Choix des éléments finis

$(\mathbf{u},p)$ est approché par $(\mathbf{u}_h,p_h) \in X_h \times Q_h$. $X_h$ et $Q_h$ doivent être choisis de telle sorte que $(\mathbf{u}_h,p_h)$ converge vers $(\mathbf{u},p)$. Une condition nécessaire et suffisante pour que cette convergence ait lieu est que : voir F. Brezzi and M. Fortin. *Mixed and hybrid finite element methods.* Springer, 1991.(p205)

$$\exists \beta > 0 \text{ tel que } \forall h > 0, \inf_{q_h \in Q_h} \sup_{\mathbf{v}_h \in X_h} \frac{b(\mathbf{v}_h, q_h)}{\|\mathbf{v}_h\|_{H^1} \|q_h\|_{L^2}} \geq \beta$$

Cette condition s'appelle la **condition Brezzi-Babuska**. L'approximation la plus simple possible, affine par morceaux $(P_1)$ pour les vitesses et les pressions ne satisfait pas cette condition (pour un maillage ne dépendant pas des itérations) (voir Ref ci dessus p207- ).
Nous choisissons une combinaison satisfaisant cette condition, quadratique pour les vitesses et affine pour la pression, $P_2-P_1$, appelée éléments de Taylor-Hood. Pour la preuve que cette combinaison vérifie la condition inf-sup nous renvoyons au livre précédemment cité, p211.

Soit $\mathcal{T}_h$ une triangulation de $\Omega$. Les espaces $X_h$ et $Q_h$ sont alors définis par :

$$X_h = \left\{ \mathbf{v}_h \in \mathcal{C}^0(\Omega)^2 \mid \mathbf{v}_{h|K} \in (P_2)^2 \ , \ \forall K \in \mathcal{T}_h \right\} \tag{2.11}$$

$$Q_h = \left\{ q_h \in \mathcal{C}^0(\Omega) \mid q_{h|K} \in P_1 \ , \ \forall K \in \mathcal{T}_h \right\} \tag{2.12}$$

### Remarque sur la notation

Dans toutes la suite nous noterons indifféremment $\phi$ les éléments de $X_h$ ou de $Q_h$. Ainsi on pose $n_2 = \dim(X_h)$ et $(\phi_i)_{1 \leq i \leq n_2}$ désigne une base de $X_h$. De même on pose $n_1 = \dim(Q_h)$ et $(\phi_i)_{1 \leq i \leq n_1}$ désigne une base de $Q_h$. L'unique façon de distinguer les éléments des bases de $X_h$ et de $Q_h$ sera de se référer à l'ensemble sur lequel ils sont indexés.

## 2.5 Réécriture de notre problème sous la forme Ax=b

### 2.5.1 Le problème linéaire

Posons $\mathbf{X}_h = \mathrm{X}_h \times \mathrm{X}_h$ , $n_1 = \dim(\mathrm{Q}_h)$ , $n_2 = \dim(\mathrm{X}_h)$ et $n = n_2 + n_2 + n_1$.

La formulation variationnelle discrète de notre problème est :

*Trouver* $(\mathbf{u}_h, \mathrm{p}_h) \in \mathbf{X}_h \times \mathrm{Q}_h$ *tel que :*

$$\begin{aligned} a(\mathbf{u}_h, \mathbf{v}_h) + b(\mathbf{v}_h, \mathrm{p}_h) &= l(\mathbf{v}_h) && \forall \mathbf{v}_h \in \mathbf{X}_h \\ b(\mathbf{u}_h, \mathrm{q}_h) &= 0 && \forall \mathrm{q}_h \in \mathrm{Q}_h \end{aligned}$$

où $a$, $b$ et $l$ ont été définies dans la section **2.4**.

Posons :

$$A = (A_{ij})_{1 \leq i,j \leq n_2} \quad ; \quad A_{ij} = \int_\Omega \phi_i \phi_j + \eta \int_\Omega \nabla \phi_i \cdot \nabla \phi_j \quad ;$$

$$\boldsymbol{B} = (Bx_{ij}, By_{ij})_{1 \leq i \leq n_1; 1 \leq j \leq n_2} \quad ; \quad Bx_{ij} = -\int_\Omega \frac{\delta \phi_j}{\delta x} \phi_i \quad ; \quad By_{ij} = -\int_\Omega \frac{\delta \phi_j}{\delta y} \phi_i$$

$$\boldsymbol{B}^* = \begin{pmatrix} Bx^T \\ By^T \end{pmatrix} \quad ; \quad \boldsymbol{A} = \begin{pmatrix} A & 0 \\ 0 & A \end{pmatrix} \quad ; \quad \boldsymbol{U}_h = \begin{pmatrix} (u_{1,h}) \\ (u_{2,h}) \end{pmatrix} \quad ; \quad \boldsymbol{F}_h = \begin{pmatrix} l(\phi_i) \\ l(\phi_i) \end{pmatrix}_{1 \leq i \leq n_2}$$

Avec les notations précédentes la formulation variationnelle discrète est équivalente au problème linéaire suivant :

$$\begin{pmatrix} \boldsymbol{A} & \boldsymbol{B}^* \\ \boldsymbol{B} & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{U}_h \\ (p_h) \end{pmatrix} = \begin{pmatrix} \boldsymbol{F}_h \\ 0 \end{pmatrix}$$

Formellement,
- $\int_\Omega \phi_i \phi_j$ est la matrice de masse, présente ici deux fois, pour $u_x$ et $u_y$.
- $\int_\Omega \nabla \phi_i \nabla \phi_j$ est la matrice du laplacien, également présente deux fois, pour $u_x$ et $u_y$.

### 2.5.2 La méthode de pénalisation

Afin de rendre notre système linéaire assurément inversible, nous utilisons la méthode de pénalisation, ce qui revient à ajouter une matrice de masse pour la pression.

Le système linéaire que nous cherchons à résoudre à chaque itération du schéma d'Euler est donc :

$$\begin{pmatrix} \boldsymbol{A} & \boldsymbol{B}^* \\ \boldsymbol{B} & -\varepsilon I \end{pmatrix} \begin{pmatrix} \boldsymbol{U}_h \\ (p_h) \end{pmatrix} = \begin{pmatrix} \boldsymbol{F}_h \\ 0 \end{pmatrix}$$

# Chapitre 3

# Commentaires suivis de l'algorithme

## Préambule

En conséquence du chapitre 2 nous souhaitons résoudre à chaque itération en temps le système linéaire suivant :

$$\begin{pmatrix} M+\eta\Delta t M_{Lap} & 0 & \Delta t Bx^T \\ 0 & M+\eta\Delta t M_{Lap} & \Delta t By^T \\ Bx & By & -\varepsilon I \end{pmatrix} \begin{pmatrix} u_x^\varepsilon \\ u_y^\varepsilon \\ \{p_h\} \end{pmatrix} = \begin{pmatrix} \left(\int_\Omega \left(u_1^{old}(x-\Delta t\times \mathbf{u}^{old}(x))\varphi_i(x)\right)\mathrm{d}x\right)_{1\le i\le n_2} \\ \left(\int_\Omega \left(u_2^{old}(x-\Delta t\times \mathbf{u}^{old}(x))\varphi_i(x)\right)\mathrm{d}x\right)_{1\le i\le n_2} \\ 0t \end{pmatrix}$$

Où :
— $M$ est la matrice de masse pour les éléments finis $P_2$,
— $M_{Lap}$ la matrice du Laplacien pour les éléments finis $P_2$.
— $Bx = \left(-\int_\Omega \frac{\delta\phi_j}{\delta x}\phi_i\right)_{1\le i\le n_1\ ;\ 1\le j\le n_2}$ et $By = \left(-\int_\Omega \frac{\delta\phi_j}{\delta y}\phi_i\right)_{1\le i\le n_1\ ;\ 1\le j\le n_2}$
— $n_1 = \dim(\mathrm{Q}_h)$ , $n_2 = \dim(\mathrm{X}_h)$

Afin de nous repérer dans la matrice, nous la découpons en bloc carré, $n_2$+$n_2$+$n_1$, numéroté 0-1-2. Ainsi le bloc 0-1 est la matrice nulle $n_2 \times n_2$.

A chaque itération en temps de notre algorithme seul varie le terme source. La construction en C++ de la matrice de gauche est expliquée dans la section 3.1 ; la construction du vecteur source est expliquée dans la section 3.2 .

Nous approximons les intégrales avec une formule de quadrature. Nous avons choisi une formule à 5 points pour le calcul d"intégrale de polynôme de degré au plus 4. Cette formule est la même dans le calcul du terme source et le calcul des matrices de masses/du laplacien etc.

## 3.1 Construction de la matrice

### 3.1.1 Division en bloc

La matrice que nous souhaitons construire est carrée, symétrique et divisée en bloc de taille respectivement $n_2$, $n_2$ et $n_1$. Ainsi notre code est soigneusement organisées en bloc, comme on peut le voir aux lignes 82-85 du fichier mainStokesP2P1.cpp.

— LIGNE 82 : WPu est un vecteur de double qui contient qf.n*6*3 éléments, 3 informations pour chacune des fonctions de bases de $P_2$ (la valeur de la fonction, sa dérivée en $x$ et sa dérivée en $y$. Et ce pour chacun des points de quadratures (qf.n points au total). De même WPp a qf.n*3*3 éléments (3 fonctions de bases pour P1). Ces différentes valeurs des fonctions de bases sont dans le triangle de référence $\widehat{K}$, elles seront évalués en un triangle $K$ donné à l'aide de la fonction setWK. BWP permet donc d'accéder à partir d'un numéro de bloc aux valeurs des fonctions de bases correspondantes.

— LIGNE 83 : numu est un vecteur qui permet le passage du numéro locale des degrés de libertés à leur numéro globale. D'un triangle donné $K$ et d'un numéro local j de degré de liberté (de 0 à 5) numu[k*6+j] renvoie le numéro global de ce degré de liberté. La matrice que nous souhaitons construire respecte évidemment la numérotation globale. Bnum permet d'accéder à partir d'un numéro de bloc (0, 1 ou 2) à la numérotation des degrés de libertés correspondant.

— LIGNE 84 : offset renseigne sur la taille des blocs.

— LIGNE 85 : BDofK contient le nombre de fonction de base par bloc. Dans notre cas c'est {6,6,3} pour $P_2$, $P_2$ et $P_1$.

### 3.1.2 Dans mainStokesP2P1.cpp

C'est dans les lignes 64 à 87 que la matrice est créée.

Un vecteur de type ItemOP, nommé ici OPA, est modifié par les fonctions AddLap, AddMass, AddPdivV et AdddivUQ. Chacune de ces fonctions ajoutent un élément à OPA qui *in fine* contient les instructions pour que la fonction Add2MatLap2QF de la ligne 87 créée la matrice voulue.

Regardons de plus près les paramètres des fonctions modifiant OPA :
— LIGNE 69, "nu,0,0" indique que la matrice du Laplacien sera ajoutée avec le coefficient $\nu$ dans le bloc 0-0 de notre matrice.
— LIGNE 70, avec "nu,1,1" : de même que la ligne 69 mais pour le bloc 1-1.
— LIGNE 71, "1,0,0" indique que la matrice de masse sera ajoutée au bloc 0-0 avec le coefficient 1.

FIGURE 3.1 – Création de la matrice

```
64    double nu = 0.0025*dt, epsp = - 1e-7;
65    int ndofp,ndofu;
66    ndofu=SetWP2(qf,WPu,D);
67    ndofp=SetWP1(qf,WPp,D);
68    vector<ItemOP> OPA, OPM;
69    AddLap(OPA,nu,0,0);
70    AddLap(OPA,nu,1,1);
71    AddMass(OPA,1,0,0);
72    AddMass(OPA,1,1,1);
73    AddMass(OPA,epsp,2,2);
74    AddPdivV(OPA,-dt,2,0,1);
75    AdddivUQ(OPA,-1,0,1,2);
76    //cout << " OP A ="<< OPA << endl;
77
78    AddMass(OPM,1,0,0);
79    AddMass(OPM,1,1,1);
80
81    //  Data block
82    const vector<double> * BWP[] = { &WPu,&WPu,&WPp};
83    const vector<int> * Bnum[]  = { & numu,&numu,&nump};
84    int offset[]={ 0, n2, n2+n2, n2+n2+n1 };
85    int BDofK[]={ndofu,ndofu,ndofp};
86    hashMatrix  DataA(n), DataM(n);
87    Add2MatLap2QF(OPA,DataA,BWP,D, Bnum,offset, Th);
88    Add2MatLap2QF(OPM,DataM,BWP,D, Bnum,offset, Th);
```

— LIGNE 72, "1,1,1" : de même que la ligne 71 mais pour le bloc 1-1.
— LIGNE 73, "epsp,2,2" indique que la matrice de masse sera ajoutée au bloc 2-2 avec le coefficient $\varepsilon$.
— LIGNE 74 "-dt,2,0,1" indique que la matrice $\binom{\Delta t Bx^T}{\Delta t By^T}$ est ajoutée au bloc 0-2.
— LIGNE 75 "-1,0,1,2" indique que la matrice $(Bx, By)$ est ajoutée au bloc 2-0.

Ainsi en comparant avec la matrice du préambule on voit que tous les blocs sont remplis comme souhaité.

### 3.1.3 Les fonctions AddMass, AddLap, AddPdivV et AdddivUQ

Ces fonctions sont définies dans EF2D.cpp, ligne 172-205, voir l'extrait de code ci-dessous.

Ces fonctions ajoutent un ou deux éléments de type ItemOP au vecteur OPA. Elles se différencient par la valeur des opérations qu'elles stockent dans op.
— Pour AddMass les opérations sont opu = 0, opv = 0.
— Pour AddLap opu = 1, opv = 1 et opu = 2, opv = 2.
— Pour AddPdivV opu = 0, opv = 1 et opu = 0, opv = 2.
— Pour AdddivUQ opu = 1, opv = 0 et opu = 2, opv = 0.

Les trois valeurs pour les opérations, 0, 1 ou 2, désignent ce qui sera implémenté dans la matrice : $u$, $\frac{\delta u}{\delta x}$ ou $\frac{\delta u}{\delta y}$. Ce qui équivaut à choisir $\varphi$, $\frac{\delta \varphi}{\delta x}$ ou $\frac{\delta \varphi}{\delta y}$.
"opu" désigne le choix de l'opération sur l'inconnu, "opv" le choix de l'opération sur les fonctions tests.
Pour la matrice du Laplacien on reconnaît : $\nabla\varphi_i \cdot \nabla\varphi_j = \frac{\delta\varphi_i}{\delta x} \times \frac{\delta\varphi_j}{\delta x} + \frac{\delta\varphi_i}{\delta y} \times \frac{\delta\varphi_j}{\delta y}$ .

FIGURE 3.2 – Les fonctions Add :

```
172 void AddMass(vector<ItemOP> & vop,double alpha,int ccu,int ccv,double *vv)
173 {
174     if(ccv<0) ccv=ccu;
175     ItemOP op={alpha,0,0,ccu,ccv,vv};
176     vop.push_back(op);
177     if(debug)   cout << " AddMass " << ccu << " " << ccv << " ::: " << op<< endl;
178
179 }
180 void AddLap(vector<ItemOP> & vop,double beta,int ccu,int ccv,double *vv)
181 {
182     if(ccv<0) ccv=ccu;
183
184     ItemOP op1={beta,1,1,ccu,ccv,vv};
185     ItemOP op2={beta,2,2,ccu,ccv,vv};
186     vop.push_back(op1);
187     vop.push_back(op2);
188     if(debug)   cout << " AddLap " << ccu << " " << ccv << " ::: " << op1 << " " << op2 << endl;
189 }
190 void AddPdivV(vector<ItemOP> & vop,double beta,int cp,int cv1,int cv2,double *v0,double *v1)
191 {
192     ItemOP op1={beta,0,1,cp,cv1,v0};
193     ItemOP op2={beta,0,2,cp,cv2,v1};
194     vop.push_back(op1);
195     vop.push_back(op2);
196      if(debug)  cout << " AddPdivV " << cp << ";" << cv1 << "," << cv2 << " ::: " << op1 << " " << op2 << endl;
197 }
198 void AdddivUQ(vector<ItemOP> & vop,double beta,int cu1,int cu2,int cq,double *v0,double *v1)
199 {
200     ItemOP op1={beta,1,0,cu1,cq,v0};
201     ItemOP op2={beta,2,0,cu2,cq,v0};
202     vop.push_back(op1);
203     vop.push_back(op2);
204      if(debug)  cout << " AdddivUQ " << cu1 << "," << cu2 << ";" << cq << " ::: " << op1 << " " << op2 << endl;
205 }
```

Si on se rappelle que les fonctions de bases ont été évaluées avec trois informations par point de quadrature, il est possible de deviner que opu/opv désigneront un décalage dans le vecteur stockant les valeurs des fonctions de base.

### 3.1.4 Les fonctions Add2MatLap2QF et AddMatSymConst

Afin d'expliquer ces deux fonctions, nous décrirons leurs arguments et ce qu'elles s'effectuent.

#### Add2MatLap2QF

Cette fonction est définies dans EF2D.cpp, ligne 258-306.

Décrivons ses arguments :

- &vop est le vecteur de type ItemOP qui contient les instructions. Ce sera évidemment OPA.
- &A est la hash-matrice qui sera le résultat du calcul, et donc la matrice que nous souhaitons calculer depuis le début de cette section.
- **pWh ce sera *BWP, donc les valeurs des fonctions de bases par bloc, voir 3.1.1 pour plus de détails.

FIGURE 3.3 – Add2MatLap2QF - Partie 1

```
258 void Add2MatLap2QF( const vector<ItemOP> & vop,hashMatrix &A, const vector<double> ** pWh,const vector<double> Dh,const
    vector<int> ** pnum,int *offset,const Mesh2d & Th)
259 {
260     long npq= Dh.size();
261     typedef map< pair<int,int>, vector<ItemOP>  > MOPB;
262     typedef MOPB::const_iterator CI_MOPB;
263     MOPB opb;
264
265     for( int i=0;i<vop.size(); ++i)
266         opb[make_pair(vop[i].cu,vop[i].cv)].push_back(vop[i]);
267
268     for(CI_MOPB ib=opb.begin(); ib != opb.end(); ++ ib)
269     {
270         //  block
271         int cu = ib->first.first;
272         int cv = ib->first.second;
273         vector<ItemOP>  vopuv=ib->second;
274         // to store the adresse of pointeur of function quadrature
275         vector<double **> pvopuv;
276         for( int i=0;i<vopuv.size(); ++i)
277             if (vopuv[i].v) pvopuv.push_back(&(vopuv[i].v)); // on stoke les adr pointeur
278         long  nptr =pvopuv.size();
279         const vector<int> & numi=*pnum[cv];
280         const vector<int> & numj=*pnum[cu];
281         const vector<double> & Whi = *pWh[cv];
282         const vector<double> & Whj = *pWh[cu];
283
284         int oi=offset[cv];
285         int oj=offset[cu];
286         long ndofKi=numi.size()/Th.nt;
287         long ndofKj=numj.size()/Th.nt;
288
289         vector<double> WKi(Whi.size()),WKj(Whj.size()),DK(Dh.size());
290         cout << cu << " " << cv << "  ; "<< oi << " " << oj <<" || " <<  ndofKi << " " << ndofKj
291         << " :  " << vopuv <<endl;
```

FIGURE 3.4 – Add2MatLap2QF - Partie 2

```
292
293         for(int k=0; k< Th.nt; ++k)
294         {
295             const Simplex &K=Th[k];
296             SetWK(K,Whi,WKi);
297             SetWK(K,Whj,WKj);
298             SetDK(K,Dh,DK);
299             AddMatSymConst(A,npq,ndofKi,&numi[k*ndofKi],vopuv,&DK[0],&WKi[0],
300                            ndofKj,&numj[k*ndofKj],&WKj[0],oi,oj
301                            );
302         }
303         for(int ptr=0; ptr < nptr; ++ptr)
304             *(pvopuv[ptr]) += npq; // decalage de pointeur dans dans les items en fin de boucle  ...
305
306     }
```

— Dh ce sera D, soit le poids des points de quadratures dans l'élément de référence $\widehat{K}$.
— **pnum ce sera *Bnum, soit la numérotation des degrés de libertés organisés par bloc, voir 3.1.1 pour plus de détails.
— *offset, de même voir 3.1.1 pour plus de détails.
— $T_h$ est le maillage, créé à la ligne 53.

Afin de parcourir le vecteur OPA cette fonction crée un nouvel objet, opb, de type MOPB (une map constitué de la clef <cu,cv> et de l'item correspondant). Ensuite, après avoir rempli opb aux lignes 265-266, on le parcourt dans la boucle for ligne 268-

306.
A chaque itération sont définies les composantes concernés (cu et cv), les numérotations correspondantes (lignes 279-280), les valeurs des fonctions tests associées (lignes 281-282), les offsets (lignes 284-285) et le nombre de fonctions test (lignes 286-287).
Chacune des intégrales à calculer est découpée selon les triangles du maillage. C'est ce pourquoi une boucle for sur les triangles est appelé à la ligne 293. A chaque itération de celle-ci les fonctions de bases sont recalculées dans le triangle $K$ par la fonction SetWK. Les poids des points de quadrature sont également recalculés dans le triangle $K$ à l'aide de la fonction SetDK (multiplié par $K.mes$).
Notons que l'itération sur opb nous impose de décaler un pointeur afin de parcourir OPA.

**AddMatSymConst**

Décrivons les arguments de cette matrice :

- A est la hash-matrice que nous souhaitons calculer.
- npq c'est le nombre de point de quadrature.
- ndofKi c'est le nombre de fonction de base, soit 3 si la composante considérée est approximée en $P_1$, soit 6 si elle l'est en $P_2$.
- *iK c'est &numi[k*ndofKi] (numi c'est *pnum[cv] et pnum c'est Bnum). Donc *ik est le pointeur vers la numérotation globale des degrés de libertés de notre triangle k pour la composante cv.
- vop sera OPA.
- *D est le poids des points de quadrature dans le triangle $K$.
- *Wi c'est les valeurs des fonctions de bases pour la composante cv dans le triangle $K$.
- oi est l'offset pour la composante cv.
- ndofKj, *jK, *Wj et oj sont l'analogue pour la composante cu.

LIGNES 216-217 on parcourt à l'aide de deux boucles for les degrés de libertés selon leur numérotation locale.
LIGNE 222, ii permet de choisir une fonction de base, et pour celle-ci, soit sa valeur, soit sa dérivée en x, soit sa dérivée en y (on parcourt bien l'ensemble des fonctions de bases car ip parcourt l'ensemble de leur numérotation locale (0-5 pour $P_2$, 0-2 pour $P_1$)).
LIGNE 224, cp est le scalaire stocké dans l'item correspondant, par exemple $\eta$ pour la matrice du laplacien.
LIGNE 227 Dans la boucle for sur les points de quadrature, ki est indenté tous les ndofKi*3 car Wi est un vecteur qf.n*ndofKi*3. Pour chaque point de quadrature et pour chaque fonction de base nous avons trois informations (valeur de la fonction de base, valeur de sa dérivée en x et valeur de sa dérivée en y).
LIGNE 233, ici est utilisé la formule d'approximation d'une intégrale par une formule de quadrature.
LIGNE 237-238 Passage de la numérotation locale à la numérotation globale.
LIGNE 241 On vient de calculer le coefficient A(i,j) sur le triangle $K$ et on l'insère dans

FIGURE 3.5 – AddMatSymConst

```
207 void AddMatSymConst(hashMatrix &A,int npq,int ndofKi,const int *iK,
208                     const vector<ItemOP> & vop, const double *D,
209                     const double *Wi,int ndofKj,const int *jK,const double *Wj,int oi,int oj)
210 {
211     if(Wj==0) Wj=Wi;
212     if(jK==0) jK=iK;
213     if(ndofKj==0) ndofKj=ndofKi;
214     int ndofKi3=ndofKi*3;
215     int ndofKj3=ndofKj*3;
216     for(int ip=0; ip<ndofKi; ++ip)
217         for(int jp=0; jp<ndofKj; ++jp)
218         {
219             double aKij =0;
220             for(int op=0; op<vop.size(); ++op)
221             {
222                 int ii= ip*3+vop[op].opv;
223                 int jj= jp*3+vop[op].opu;
224                 double cp = vop[op].c;
225                 double *pc = vop[op].v;
226 
227                 for(int p=0,ki=ii,kj=jj  ;   p<npq;   ++p, ki+=ndofKi3, kj+=ndofKj3 )
228                 {
229                     if(pc) {
230                         cp = *pc++;
231                         if(pdebugF  && (ip==0) && (jp==0)  ) cout <<  p << " F[ " << pc-pdebugF-1 << "] "<< cp << endl;
232                     }
233                     aKij += D[p]*cp*Wi[ki]*Wj[kj];
234 
235                 }
236             }
237             int i = iK[ip]+oi;
238             int j = jK[jp]+oj;
239             if(debug>2)
240               cout << "Akij " << i << " " << j << " " << aKij << endl;
241             A(i,j) +=aKij;
242         }
```

la hash-matrice.

### 3.1.5 Affichage

Dans cette sous-section nous souhaitons commenter l'affichage du résultat pour une itération. Nous ne prêterons pas attention aux dernières lignes ( fh / ds / min u1 / min u2 et err ).

FIGURE 3.6 – La matrice affichée dans la console

```
root@pcmint:/home/fitz/Bureau/ProjetC/Nst# ./mainStokesP2P1 Th.msh
 End read 665 1152 mes =9 mesb = 22
0 0  ; 0 0 || 6 6 :   +0.0005 u1.dx v1.dx   +0.0005 u1.dy v1.dy   +1 u1 v1
0 2  ; 4962 0 || 3 6 :   -1 u1.dx v3
1 1  ; 2481 2481 || 6 6 :   +0.0005 u2.dx v2.dx   +0.0005 u2.dy v2.dy   +1 u2 v2
1 2  ; 4962 2481 || 3 6 :   -1 u2.dy v3
2 0  ; 0 4962 || 6 3 :   -0.2 u3 v1.dx
2 1  ; 2481 4962 || 6 3 :   -0.2 u3 v2.dy
2 2  ; 4962 4962 || 3 3 :   -1e-07 u3 v3
 fh: min 0 max 0
 ds : eps = 1e-08
 min u1:  -2.31198e-31 max 1
 min u2:  -0.586962 max 0.157707
 err =  0
Iteration2
root@pcmint:/home/fitz/Bureau/ProjetC/Nst#
```

Après End read s'affiche le nombre de point du maillage Th.msh (665) et son nombre de triangle (1152). mes est la mesure totale du domaine (9) et mesb est la mesure de la frontière (22). On pourra ainsi vérifier le calcul d'aire et de périmètre sachant que ce domaine est de hauteur 0.5 sur une longueur de 2 puis de hauteur de 1 sur une longueur de 8.

Les 7 lignes qui suivent nous intéressent tout particulièrement, elles sont écrites par la fonction Add2MatLap2QF (ligne 290-291 du fichier EF2D.cpp).

Les couples d'entiers 0 0 / 0 2 / 1 1 ... correspondent aux composantes cu cv stockées dans l'item correspondant. Ces chiffres correspondent également aux découpages en bloc de notre matrice.

Les deux entiers qui suivent sont les offsets correspondant aux composantes/blocs précédents. Pour ce maillage on voit que notre matrice est découpé par bloc 0/2481/4962/ 4962+$n_1$ ($n_1$ n'est pas affiché ici).

Ensuite, après les deux barres || s'affiche le nombre de fonctions de base pour les composantes correspondantes, 6 pour $P_2$ et 3 pour $P_1$.

Enfin tout à droite se trouve une prévisualisation des blocs calculés. Sachant que dans cet exemple $\eta = 0.0025$ et $\Delta t = 0.2$ et que les inconnues sont notées $u_1$, $u_2$ et $u_3$ et les fonctions tests $v_1$, $v_2$ et $v_3$ nous retrouvons exactement la matrice du préambule.

## 3.2 Calcul du terme source

Pour rappel nous devons calculer le vecteur

$$\begin{pmatrix} \left(\int_\Omega \left(u_1^{old}(x - \Delta t \times \mathbf{u}^{old}(x))\varphi_i(x)\right) \mathrm{d}x\right)_{1\leq i\leq n_2} \\ \left(\int_\Omega \left(u_2^{old}(x - \Delta t \times \mathbf{u}^{old}(x))\varphi_i(x)\right) \mathrm{d}x\right)_{1\leq i\leq n_2} \\ 0 \end{pmatrix}.$$

Pour la suite nous posons $\mathbf{g} \ : \ \mathbb{R}^2 \to \mathbb{R}^2 \mid \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x - \Delta t \times \mathbf{u}_x^{old}(x,y) \\ y - \Delta t \times \mathbf{u}_y^{old}(x,y) \end{pmatrix}$.

### 3.2.1 Description des différentes étapes

Premièrement voici le calcul de notre intégrale, pour $1 \leq i \leq n_2$ et pour $u_1^{old}$ :

$$\int_\Omega \left(u_1^{old}(x - \Delta t \times \mathbf{u}^{old}(x))\varphi_i(x)\right) \mathrm{d}x = \sum_{K\in\mathcal{T}_h} \int_K \left(u_1^{old}\left(\mathbf{g}(x)\right)\varphi_i(x)\right) \mathrm{d}x \tag{3.1}$$

$$= \sum_{K\in\mathcal{T}_h} \lambda(K) \sum_{l=1}^{N} \omega_l \, u_1^{old}\left(\mathbf{g}\left(\xi_K^l\right)\right) \varphi_i\left(\xi_K^l\right) \tag{3.2}$$

De (3.1) à (3.2) nous avons utilisé notre formule de quadrature. $N$ est le nombre de point de cette formule. $\omega_l$ est le poids du point de quadrature $l$. Dans notre code nous avons choisi $T_5$, soit sept points pour évaluer notre intégrale. $\lambda(K)$ est le mesure de Lebesgue du triangle $K$.

**Remarque sur les points de quadrature**

Les points de quadrature,$\xi_K^l$ d'un triangle $K$ sont définis à partir des points de quadrature $\widehat{\xi^l}$ de l'élément de référence $\widehat{K}$ par la formule qui suit :

$$\xi_K^l = \left(1 - \widehat{\xi^l}_x - \widehat{\xi^l}_y\right)\mathbf{K}_0 + \widehat{\xi^l}_x\mathbf{K}_1 + \widehat{\xi^l}_y\mathbf{K}_2$$

Les termes $1 - \widehat{\xi^l}_x - \widehat{\xi^l}_y$, $\widehat{\xi^l}_x$ et $\widehat{\xi^l}_y$ sont ainsi les coordonnées barycentriques du point $\xi_K^l$. Par un rapide calcul, remplaçant $\mathbf{K}_0$ par $(0,0)$, $\mathbf{K}_1$ par $(1,0)$ et $\mathbf{K}_2$ par $(0,1)$, c'est-à-dire si on remplace $K$ par $\widehat{K}$, on retrouve $\widehat{\xi^l}$. Ainsi $\widehat{\xi^l}$ et $\xi_K^l$ ont les mêmes coordonnées barycentriques.

Sachant que les fonctions de bases de $X_h$ ou de $Q_h$, *ie* les polynômes éléments finis, sont construits pour être invariant par translation affine et ainsi préserver les coordonnées barycentriques, (3.2) se réécrit :

$$\sum_{K\in\mathcal{T}_h} \lambda(K) \sum_{l=1}^{N} \omega_l \; u_1^{old}\left(\mathbf{g}\left(\xi_K^l\right)\right) \varphi_i\left(\widehat{\xi^l}\right) \tag{3.3}$$

Dans le sillage de la quantité précédente s'esquisse l'étape majeure de notre calcul : évaluer les quantités $u_1^{old}\left(\mathbf{g}\left(\xi_K^l\right)\right)$ et $u_2^{old}\left(\mathbf{g}\left(\xi_K^l\right)\right)$ pour tout triangle $K$ et tout $l \in [\![1, N]\!]$ de chacun des triangles. Ici $l$ dépends de $K$ : nous avons au total $N \times \#\mathcal{T}_h$ points de quadrature ($\#\mathcal{T}_h$ désigne le nombre de triangle du maillage).

La première étape consiste à calculer $\mathbf{g}\left(\xi_K^l\right)$ pour les $N \times \#\mathcal{T}_h$ points de quadrature, la seconde à évaluer en $\mathbf{g}\left(\xi_K^l\right)$ la fonction $\mathbf{u}^{old}$.

### 3.2.2 Calcul de $\mathbf{g}\left(\xi_K^l\right)$

Quatre fonctions C++ sont définies pour ce calcul : SetF de EF2D.cpp, FFx, FFy et CreatePp de mainStokesP2P1.cpp. Voici leur code :

Ensuite ces fonctions sont appelées dans les lignes 160 à 175 de mainStokesP2P1.cpp :

FIGURE 3.7 – SetF

```
488 void SetF(const Mesh2d &Th,const QF2 &qf,const vector<double> &WP2,const vector<int>& num,const vector<double> & u,
    vector<double> &F, double (*FF)(double x,double y,double v))
489 {
490
491     vector<double> WK(WP2.size());
492     int ndofK = num.size()/Th.nt;
493
494     for(int k=0,kf=0,nk=0; k<Th.nt; ++k, nk+=ndofK )
495     {
496         const Simplex &K=Th[k];
497         SetWK(K,WP2,WK);
498         for(int p=0,ii=0; p<qf.n; ++p,++kf )
499         {
500             R2 P=K(qf[p]);
501             double U =0;
502             for(int ip=0; ip<ndofK; ++ip,ii+=3)
503             {
504                 U+= u[num[nk+ip]]*WK[ii];
505
506             }
507
508             F[kf] = FF(P.x,P.y,U);
509             if(debug || pdebugF)
510                 cout << k << " : F " << p  << " kf " << kf << "  == " <<  F[kf]  << endl;
511         }
512     }
513 }
```

FIGURE 3.8 – FFx et FFy

```
28 double FFx(double x,double y,double ux) { return x - dt * ux;}
29 double FFy(double x,double y,double uy) { return y - dt * uy;}
```

FIGURE 3.9 – CreatePp

```
39 void CreatePp(vector<R2> &Pp, vector<double> Fx,vector<double> Fy){
40   for(int k=0; k<Fx.size(); k++){
41     R2 A(Fx[k],Fy[k]);
42     Pp[k]=A;}
43   }
```

FIGURE 3.10 – Le calcul dans main

```
160       // ------------------------------ CALCULATION OF P' ----------------------------
161
162
163       vector<double> F1(qf.n*Th.nt);
164       vector<double> F2(qf.n*Th.nt);
165       vector<R2> Pp(qf.n*Th.nt);
166       for (int k=0; k<n2; k++){
167         ux[k]=u[k];
168         uy[k]=u[n2+k];
169
170       }
171
172       SetF(Th,qf,WPu,numu,ux,F1,FFx);     // calcule les coordonnées en x de P'
173       SetF(Th,qf,WPu,numu,uy,F2,FFy);     // calcule les coordonnées en x de P'
174
175       CreatePp(Pp, F1, F2);
176
```

— COMMENTAIRE DE 3.10 : $\mathbf{g}\left(\xi_K^l\right)$ est dans $\mathbb{R}^2$ donc nous divisons le calcul en deux , un pour les coordonnées en $x$ et un pour les coordonnées en $y$. $F_1$ contiendra $g_1\left(\xi_K^l\right)$ et $F_2$ $g_2\left(\xi_K^l\right)$. $Pp$ est un vecteur de point de $\mathbb{R}^2$ et de taille $N \times \#\mathcal{T}_h$. C'est dans ce vecteur que nous stockerons les résultats. Comme le calcul a été

fait sur deux vecteurs de double, nous avons besoin de la fonction CreatePp qui agrège les résultats.

— COMMENTAIRE DE 3.9 : CreatePp est simplement une fonction qui remplie Pp vecteur de $\mathbb{R}^2$ à partir des coordonnées en $x$ et en $y$ stockées dans $F_1$ et $F_2$.

— COMMENTAIRE DE 3.8 : Ces fonctions sont extrêmement simples. Elles justifient leur existence dans un soucis de généralisation. Cette précaution trouvera sa pleine utilité dans la phase de vérification de notre code.

— COMMENTAIRE DE 3.7 : C'est ici que tout le calcul se fait. Nous souhaitons calculer $g_1$ (ou $g_2$ selon les arguments mis en SetF) en $N \times \#\mathcal{T}_h$ points de quadrature. On retrouve donc une boucle sur les points de quadrature (ligne 498) incluse dans une boucle sur les triangles (ligne 494). A l'intérieur de ces deux boucles on trouve ligne 508 le calcul souhaité. En effet $U$ sera $u_x\left(\xi_K^l\right)$ (ou $u_y\left(\xi_K^l\right)$ selon les arguments de SetF). Pour justifier ce point : $u_x$ s'approxime sur chaque triangle comme la somme des 6 fonctions de bases multipliées par la valeur de $u_x$ au degré de liberté correspondant (voir formule ci-dessous). Tout n'est plus qu'une question d'indice et de manipulation des données. L'usage de num est usuel, celui de WK également (modulo 3 car nous avons seulement besoin de la valeur des fonctions, WP2 contient déjà la valeur des fonctions de bases de $P_2$ en chacun des points de quadratures de $\widehat{K}$ (WP2 est de taille $qf.n * 6 * 3$)). Enfin notons que les points de quadratures de $\widehat{K}$ sont accessibles via la commande $qf[p]$. Il est donc nécessaire d'appeler l'opérateur $K(...)$, défini dans EF2d-base.hpp ligne 70-73, qui nous donne l'image d'un point de $\widehat{K}$ dans $K$ afin de calculer les coordonnées en $x$ et en $y$ de $\xi_K^l$.

La formule mentionnée plus haut :

$$Si\ x \in K \quad u_1(x) = \sum_{j \in JK} u_{1,j} \times \varphi_j(x) \tag{3.4}$$

où $JK$ désigne la numérotation globale des 6 degrés de libertés de $K$.

### 3.2.3 Calcul de u $\left(\mathbf{g}\left(\xi_K^l\right)\right)$

Nous avons besoin de deux fonctions, CalculUp et SetWPP2, définies dans EF2d.cpp :

**COMMENTAIRE DE SetWPP2**

Cette fonction calcule les valeurs des fonctions de base de $P_2$ en un point donné $P$. Elle prend en argument le point de $\mathbb{R}^2$, $P$ ; le triangle dans lequel se trouve ce point (dénoté par son numéro dans le maillage $kk$) ; un pointeur de double, $*WP2$, qui stockera les valeurs ; et pour finir le maillage $\mathcal{T}_h$. LIGNES 56-57 nous récupérons le triangle $kk$ et ses trois sommets sous la forme d'un Simplex et trois Vertex. Ces objets sont définis dans EF2d-base.hpp, ligne 13 à 75. LIGNES 59-66 nous stockons dans le tableau de réel (double), les coordonnées barycentriques de $P$ dans le triangle $kk$. LIGNES 67-83 nous

FIGURE 3.11 – SetWPP2

```
54 int  SetWPP2(const R2 &P,int kk,double *WP2,const Mesh2d & Th)
55 {
56   const Simplex & T(Th[kk]);
57   const R2 Q[3]={R2(T[0]),R2( T[1]),R2(T[2])};
58
59     R l[3];
60     l[0] = det(P  ,Q[1],Q[2]);
61     l[1] = det(Q[0],P  ,Q[2]);
62     l[2] = det(Q[0],Q[1],P  );
63     R Det = l[0]+l[1]+l[2];
64     l[0] /= Det;
65     l[1] /= Det;
66     l[2] /= Det;
67     int k=0;
68     R2 Dl[3]= {R2(-1,-1),R2(1,0),R2(0,1)};
69     for(int s=0; s<3; ++s )
70     {
71         WP2[k++]= l[s]*(2*l[s]-1);
72         R2 D2 = Dl[s]*((2*l[s]-1)+ 2*l[s]) ;
73         WP2[k++]= D2.x;
74         WP2[k++]= D2.y;
75     }
76     for(int s=0; s<3; ++s )
77     {
78         int i=(s+1)%3, j= (s+2)%3;
79         WP2[k++]= 4.*l[i]*l[j];
80         R2 D2 = (4*l[i])*Dl[j] + (4*l[j])*Dl[i];
81         WP2[k++]= D2.x;
82         WP2[k++]= D2.y;
83     }
84     return k;
85 }
```

FIGURE 3.12 – CalculUp

```
463 double CalcUp(R2 Pp,int Kp,vector<double> u,const  vector<int> &num ,int ndofu,  int c,const Mesh2d & Th){
464
465   vector<double>  WPprime(6*3);
466   double uprime=0.0;
467   int sizeWP;
468   sizeWP=SetWPP2(Pp,Kp ,&WPprime[0],Th);
469
470   for(int i=0; i<ndofu; i++){
471
472     uprime+=u[num[ndofu*Kp+i]+c]*WPprime[3*i];
473     //std::cout << "WPprime[]" << WPprime[3*i]<< '\n';
474
475     //std::cout << "u =  " <<  u[num[ndofu*Kp+i]+c] << '\n' << endl;
476     //std::cout << "WP' =  " <<  WPprime[3*i] << '\n' << endl;
477
478   }
479
480   return uprime;
481 }
```

évaluons les valeurs des fonctions de bases des polynômes de Lagrange $P_2$ en $P$. Ces fonctions de bases sont définis à partir des coordonnées barycentriques, ce qui justifie l'étape précédente.

### COMMENTAIRE DE CalculUp

A l'aide de cette fonction nous souhaitons évaluer les fonctions $u_1$ et $u_2$ en un point $\xi_K^l$ ($Pp$ dans notre fonction). Formellement nous utilisons la formule (3.4) mentionnée dans la section précédente. Nous avons donc besoin des coefficient de $u_1$ au degrés de libertés

du triangle $Kp$ contenant le point $Pp$. Le terme ligne 472 $u[num[ndofu * Kp + i] + c]$ accomplit cela : on se situe dans $u_1$ ou $u_2$ avec l'offset ; $num$ sera la numérotation des degrés de liberté de $P_2$ ($numu$) ; $ndofu * Kp + i$ est le numéro local du degré de liberté $i$ du triangle $Kp$.
Ensuite nous avons calculé à l'aide de SetWPP2 les fonctions de bases en $Pp$. Nous n'avons pas besoin ici de SetWK car seules les valeurs des fonctions de bases sont nécessaires, et non les valeurs de leurs dérivées.
Enfin $uprime$ contient exactement $u_1(Pp)$, pour $Pp$ un point de quadrature $\xi_K^l$

### 3.2.4 Calcul du terme source

Pour rappel nous devons calculer la quantité suivante pour $j = 1, 2$ et pour tout $1 \leq i \leq n_2$

$$\sum_{K \in \mathcal{T}_h} \lambda(K) \sum_{l=1}^{N} \omega_l \; u_j^{old} \left( \mathbf{g} \left( \xi_K^l \right) \right) \varphi_i \left( \widehat{\xi}^l \right)$$

Ce calcul est réalisé lignes 187 à 231.

- LIGNE 210 : Afin d'appeler la foncion CalculUp nous avons besoin du triangle contenant le point $Pp$. Plus précisément nous avons besoin du numéro de ce triangle dans le maillage Th. Ce numéro est noté dans le programme $it$ et il est modifié par la fonction Find.
- LIGNES 212-214 : Dans le cas où le point $Pp$ est situé en dehors du domaine $\Omega$, la fonction Find calcule son projeté sur la frontière. Pour cela elle utilise $Phat$ qui contient les coordonnées barycentrique du projeté dans le triangle $it$. Comment à partir de cela retrouver les coordonnées cartésiennes du projeté ? Avec $Th[it]$ nous obtenons le triangle, objet de classe Simplex ; et avec $Th[it](Phat)$ nous utilisons l'opérateur défini lignes 70-73 de EF2d-base.hpp, qui d'un triangle et de coordonnées barycentrique nous donne les coordonnées cartésiennes du point concerné.
- LIGNE 191 et LIGNES 226-228 : Afin de calculer le terme source en tout $1 \leq i \leq n_2$ (pour $u_1$ et $u_2$), nous utilisons la stratégie suivante : au lieu de parcourir les degrés de libertés globalement, par exemple avec une boucle for de 1 à $n_2 + n_2$, nous parcourons l'ensemble des triangles du maillage (ligne 187) et les six degrés de libertés de chacun des triangles (ligne 198). Afin de repasser du local au global, nous avons besoin de la numérotation globale des degrés de liberté pour un triangle fixé, c'est ce qui est fait ligne 191. Enfin une fois calculé notre intégrale sur un triangle donné et pour une fonction de base donnée le résultat est inséré dans $unew$ à l'aide des lignes 226 à 228.
- LIGNES 219-220 : Après le calcul des valeurs des fonctions de bases $P2$ dans $WPKfin$ (à l'aide des fonctions SetWP2 et SetWK), le poids du point de quadrature est multiplié. On retrouve dans ces lignes exactement $\omega_l \; u_j^{old} \left( \mathbf{g} \left( \xi_K^l \right) \right) \varphi_i \left( \widehat{\xi}^l \right)$.

FIGURE 3.13 – Calcul du terme source

```
187       for(int k=0; k<Th.nt; k++){
188
189          const  Simplex & K= Th[k];
190
191          int *jK= &numu[BDofK[0]*k];
192          // int *jKy= &numu[BDofK[1]*k];
193
194          SetWK(K,WPfin,WPKfin);
195          double mes = K.mes;
196
197
198          for(int j=0; j<6; j++){
199            double uxj=0;
200            double uyj=0;
201
202            for(int p=0; p<qf.n; p++){
203              double upx;
204              double upy;
205              int it;
206              R2 Phat;
207              const Simplex * Khat;
208              bool outside;
209              const Simplex * tstart = & Th[k];
210              Khat=Find(Th,Pp[k*qf.n+p],Phat,outside,tstart,it);
211
212              if(outside==1){
213                Pp[k*qf.n+p]=Th[it](Phat);
214              }
215
216              upx=CalcUp(Pp[k*qf.n+p],it, u, *Bnum[0] ,BDofK[0] , offset[0],Th);
217              upy=CalcUp(Pp[k*qf.n+p],it, u, *Bnum[1] ,BDofK[1] , offset[1],Th);
218
219              uxj+= D[p] * upx * WPKfin[6*3*p+3*j];
220              uyj+= D[p] * upy * WPKfin[6*3*p+3*j];
221
222            }
223            uxj*=mes;
224            uyj*=mes;
225
226            int jj=jK[j];
227              unew[jj+offset[0]]+=uxj;
228              unew[jj+offset[1]]+=uyj;
229          }
230
231        }
```

— LIGNES 223-224 : nous retrouvons la mesure de lebesgue du triangle $K$.

## 3.3 Commentaires divers

### 3.3.1 Architecture globale

Après avoir défini la matrice du problème linéaire à résoudre et la structure en bloc la boucle en temps commence. Les étapes sont les suivantes (dans mainStokesP2P1.cpp) :

— LIGNES 98-135 : Fixages des conditions au bords sur $u$ future solution du système linéaire et application de la méthode de Très Grande Valeur (voir section suivante).
— LIGNES 139-150 : Résolution du système linéaire.
— LIGNES 163-231 : Calcul du terme source pour l'itération suivante.
— LIGNE 233 : On insère le terme source calculé dans b.
— LIGNES 235-243 : Calcul et affichage de l'erreur récursive en norme $L^2$.

— LIGNEs 248-271 : Graphique de la solution u à l'aide de Freefem++, conditionné par le paramètre *kkk* afin de ne pas afficher la solution pour toutes les itérations, mais seulement selon un modulo à choisir.

### 3.3.2 La méthode TGV

Dans notre programme nous avons utilisé aux lignes 98-135 la méthode TGV afin d'indiquer au Solver de ne pas calculer, pour chaque itération, les termes de la solution $u$ situé sur les bords soumis à une condition de Dirichlet (soit le bord du flux entrant, le haut et le bas de notre domaine). Le bord de droite, celui du flux sortant n'est pas concerné (car condition de Neumann) comme l'indique la restriction de la ligne 103 (label!=2).

### 3.3.3 Condition sur le bord

Ligne 131 les conditions au bords sont insérées dans le vecteur $u$. Celles-ci sont définis au préalable avec une fonction g située en tête de notre programme. On retrouve ainsi la condition nulle sur les bords hauts et bas et le flux entrant sur le mode parabolique. Les lignes en commentaire dans la définition de g correspondent à la condition d'entrée pour un maillage spécifié.

### 3.3.4 Condition CFL

Dans notre programme nous devons veiller à respecter la condition CFL : $\frac{|u_x|\Delta t}{\Delta x} + \frac{|u_y|\Delta t}{\Delta x} \leq 1$. Si nous avons 5 degrés de libertés ($nn$ dans marche.edp) sur la largeur notre domaine qui est de hauteur 1, $\Delta x = 0.2$. Le maximum de la vitesse étant 1, on prend $\Delta t = 0.2$. Si $nn = 10$, on prendra $\Delta t = 0.1$

### 3.3.5 Calcul de l'erreur récursive en norme L2

Déterminer si notre solution a convergé n'est pas aisé. Un des indicateurs de cette convergence est l'erreur récursive en norme $L^2$. Son calcul est effectué ligne 235-243. Notons tout de même qu'un nombre, même infime, n'est pas synonyme de raisonnable, solide ou précis.

### 3.3.6 Affichage des résultats pour Reynolds = 200, 400, 800

# Chapitre 4

# Procédure de vérification

## Préambule

Pour chacune des vérifications le nom du fichier .cpp effectuant la vérification contient le numéro de la section : par exemple pour la vérification de la résolution du problème de Stokes (section 4.1), le fichier correspondant s'appelle check41.cpp . Ce fichier affiche des résultats, comme la norme $L^2$ et des graphiques. Pour ce faire il utilise le fichier Freefem check41.edp .

Ces vérifications sont effectués pour le maillage produit par marche.edp, avec le paramètre $nn = 10$ (le nombre de degrés de libertés sur la largeur totale de notre domaine est 10). Il est possible d'effectuer la vérification pour des maillage plus ou moins fin. Il sera alors nécessaire de changer $nn$ dans les fichiers Freefem++ correspondant. Par exemple pour la vérification du problème de Stokes, on peut lancer marche.edp avec $nn = 15$ et mettre également $nn = 15$ dans check41.edp .

De même la viscosité est fixé à $0,0025$.

## 4.1 Stokes

Dans cette section nous testons la résolution du problème de Stokes uniquement. Le test est effectué avec check41.cpp . Formellement nous résolvons Stokes avec notre code, puis avec Freefem++, ensuite nous calculons la différence des deux solutions, en norme $L^2$, $L^1$ et $L^\infty$ ; et pour finir nous affichons les graphiques des vitesses et des pressions des deux solutions, d'abords notre solution puis celle de Freefem++.

Nous devons donc résoudre le système linéaire suivant :

$$\begin{pmatrix} \eta M_{Lap} & 0 & Bx^T \\ 0 & \eta M_{Lap} & By^T \\ Bx & By & -\varepsilon I \end{pmatrix} \begin{pmatrix} u_x^\varepsilon \\ u_y^\varepsilon \\ \{p_h\} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{4.1}$$

Voici le résultat dans la console :

FIGURE 4.1 – ./check41 Th.msh dans la console

```
root@pcmint:/home/fitz/Bureau/ProjetC/Navier Stokes# ./check41 Th.msh
 lecture de Th.msh
 End read 1011 1800 mes =9 mesb = 22
 n1 = 1011
 n2 = 3821
 ndof = 8653
 OP A = +0.0025 u1.dx v1.dx   +0.0025 u1.dy v1.dy   +0.0025 u2.dx v2.dx   +0.0025 u2.dy v2.dy   -1e-07
 u3 v3   -1 u3 v1.dx   -1 u3 v2.dy   -1 u1.dx v3   -1 u2.dy v3
0 0  ; 0 0 || 6 6 :   +0.0025 u1.dx v1.dx   +0.0025 u1.dy v1.dy
0 2  ; 7642 0 || 3 6 :   -1 u1.dx v3
1 1  ; 3821 3821 || 6 6 :   +0.0025 u2.dx v2.dx   +0.0025 u2.dy v2.dy
1 2  ; 7642 3821 || 3 6 :   -1 u2.dy v3
2 0  ; 0 7642 || 6 3 :   -1 u3 v1.dx
2 1  ; 3821 7642 || 6 3 :   -1 u3 v2.dy
2 2  ; 7642 7642 || 3 3 :   -1e-07 u3 v3
 fh: min 0 max 0
 ds : eps = 1e-08
 A: min -0.0333333 max 1e+30
 min u1:  -0.000804034 max 1.00055
 min u2:  -0.35308 max 0.00547391
 min p:  -1.38074e-13 max 0.256513
-- FreeFem++ v  3.610001 (date Jeu 12 jul 2018 12:05:08 CEST)
 Load: lg_fem lg_mesh lg_mesh3
 sizestack + 1024 =3160  ( 2136 )

  -- Square mesh : nb vertices  =1111 ,  nb triangles = 2000 ,  nb boundary edges 220
  -- Solve :
          min -0.000804034  max 1.00055
          min -0.35308  max 0.00547391
          min -1.38061e-13  max 0.256513
1.00055 0.35308 0.256513
L2 err =2.2766e-07
Linf err =5.02096e-06
L1 err =1.25042e-07


times: compile 0.005069s, execution 0.168783s,  mpirank:0
 CodeAlloc : nb ptr  3391,  size :398528 mpirank: 0
Ok: Normal End
root@pcmint:/home/fitz/Bureau/ProjetC/Navier Stokes#
```

Avec $OPA$ on peut observer que la matrice du système linéaire résolu est identique à (4.1). On voit que les erreurs sont de l'ordre $10^6$, donc satisfaisantes. Les graphiques ne sont pas ici nécessaires.

## 4.2 Le terme source

### 4.2.1 CalculUp donne les mêmes résultats que SetF

Pour le fichier check421.cpp, nous avons pris le fichier final mainStokesP2P1.cpp pour une seule itération.

Un observateur pointilleux aura remarqué que nous avons à notre disposition deux fonctions pour calculer $u_i\left(\xi_K^l\right)$. A la condition de paramétrer FFx et FFy correctement, SetF calcule cela directement pour tous les points de quadrature ; alors que CalculUp peut calculer $u_i\left(\xi_K^l\right)$ pour un point de quadrature donné en argument.

Nous paramétrons donc FFx et FFy, ligne 53 et 54 de check421.cpp .

FIGURE 4.2 – Paramétrage des fonctions FF

```
53 double FFx(double x,double y,double ux) { return ux ;}
54 double FFy(double x,double y,double uy) { return uy ;}
```

Enfin aux lignes 185-206 nous affichons le test : nous parcourons l'ensemble des points de quadrature du triangle 64 et affichons les résultats de SetF et de CalculUp, pour $u_1$ et $u_2$. Le triangle peut être choisir librement entre 0 et 1799 à la ligne 191 (le maillage Th.msh contient 1800 triangles (marche.edp avec $nn = 10$)).

FIGURE 4.3 – SetF et CalculUp aux points de quadrature du triangle k1=64

```
185     SetF(Th,qf,WPu,numu,ux,F1,FFx);     // calcualates the x coordinates of P'
186     SetF(Th,qf,WPu,numu,uy,F2,FFy);     // calculates the y coordinates of P'
187
188
189
190
191     int k1=64;
192     const Simplex &K=Th[k1];
193     std::cout << "Triangle K =" <<k1 << '\n'<<endl;
194     for(int p=0; p<qf.n; p++){
195       //int p=0;
196       R2 P=K(qf[p]);
197       double ttt, tttt;
198
199       ttt=CalcUp(P,k1, u, *Bnum[0] ,BDofK[0] , offset[0],Th);
200       tttt=CalcUp(P,k1, u, *Bnum[1] ,BDofK[1] , offset[1],Th);
201       std::cout << "Quadrature point number " << p<<'\n';
202       std::cout << " Calcupx = " << ttt << " Calcupy = " << tttt <<'\n';
203       std::cout << " F1 = " << F1[p+qf.n*k1] << " F2 = " << F2[p+qf.n*k1] <<'\n'<<endl;
204
205     }
206
```

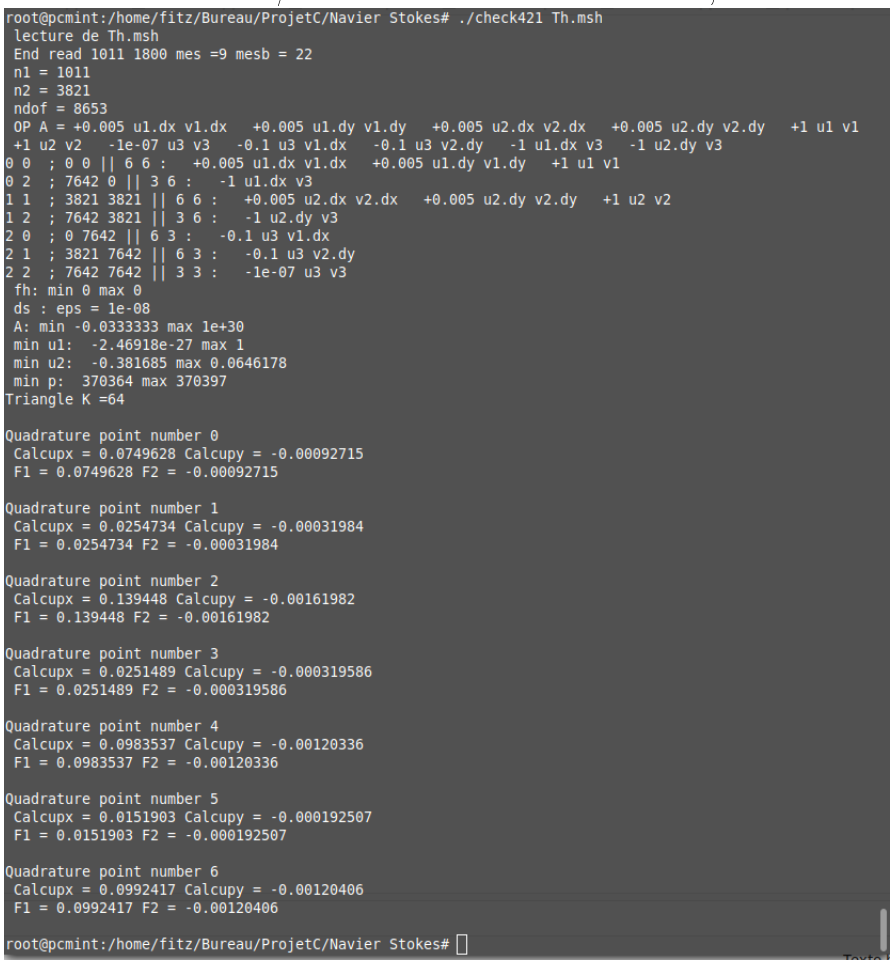
La figure 4.4 ci-dessous est le résultat dans la console de ./check421 Th.msh pour le triangle 64. On observe que les valeurs de $u_i\left(\xi_K^l\right)$ sont identiques selon les deux méthodes de calculs.

### 4.2.2 Stokes en temps

Dans check422.cpp nous testons Stokes en temps. Le système linéaire à résoudre est le suivant (nous posons $\Delta t = 1$) :

$$\begin{pmatrix} M+\eta M_{Lap} & 0 & Bx^T \\ 0 & M+\eta M_{Lap} & By^T \\ Bx & By & -\varepsilon I \end{pmatrix} \begin{pmatrix} u_x^\varepsilon \\ u_y^\varepsilon \\ \{p_h\} \end{pmatrix} = \begin{pmatrix} \left(\int_\Omega \left(u_1^{old}(x-\Delta t \times \mathbf{u}^{old}(x))\varphi_i(x)\right) \mathrm{d}x\right)_{1\leq i\leq n_2} \\ \left(\int_\Omega \left(u_2^{old}(x-\Delta t \times \mathbf{u}^{old}(x))\varphi_i(x)\right) \mathrm{d}x\right)_{1\leq i\leq n_2} \\ 0 \end{pmatrix} \tag{4.2}$$

FIGURE 4.4 – ./check421 Th.msh dans la console, k1=64




Sachant que la solution du problème de Stokes est la limite stationnaire de Stokes en temps, à la première itération nous résolvons le problème de Stokes, puis nous injectons la solution dans Stokes en temps, sous la forme de $\mathbf{u}^{old}$. Le résultat attendu est une erreur entre $\mathbf{u}$ et $\mathbf{u}$Stokes proche de 0 ($\mathbf{u}$Stokes est la solution du problème de Stokes, calculée en première itération et préservée après).

Si tel est le cas le calcul du terme source sera confirmé, c'est en effet le seul calcul non trivial qui est opéré ici. L'ajout de la matrice de Masse $M$ est usuel, ce n'est qu'une modification de $OPA$.

Comme précédemment, le fichier de base de check422.cpp est mainStokesP2P1.cpp.

— LIGNES 251-252 : Pour chaque itération nous calculons le vecteur $e$ qui contient la différence de la solution $u$ et $uStokes$.

FIGURE 4.5 – check422.cpp lignes 248-275

```
248 //-----------------------------Stokes in time verification --------------------
249
250
251   for(int i=0; i<n; ++i  )
252       e[i] = u[i]-uStokes[i];
253
254
255 //--------  ------ L2 ERROR --------------------
256    ProduitMatVec(DataM,&e[0],&Me[0]);
257   double errL2 = sqrt(mysdot(n,&e[0],&Me[0]));
258
259 //-------------- Linf  ERROR -------------------
260
261 double errLinf=normsum(&u[0],&uStokes[0],n2+n2);
262
263
264   cout << " err L2= "<<  " " << errL2  << endl;
265   cout << " err Linf= "<<  " " << errLinf << endl;
266
267
268   if(mcount==0){
269     uStokes=u;
270      AddMass(OPA,1,0,0);
271      AddMass(OPA,1,1,1);
272      Add2MatLap2QF(OPA,DataA,BWP,D, Bnum,offset, Th);
273      mcount++;
274   }
275
```

— LIGNES 255-265 : Nous calculons les erreurs $L2$ et $L^\infty$ et les affichons pour chaque itération.
— LIGNES 268-274 : mcount est un compteur extérieur qui vaut 0 initialement et qui ainsi est modifié une seule fois. Il permet de poser le problème de Stokes initialement puis d'ajouter la matrice de Masse afin de résoudre Stokes en temps. Le terme source étant initialisé à 0, on retrouve bien Stokes à la première itération.

Voici avec la figure 4.6 les résultats dans la console de check422.cpp :

(Commentaire linéaire de la figure 4.6) On voit par la première valeur de OPA que c'est bien le problème de Stokes qui est résolu à la première itération. 'err L2' et 'err Linf' sont alors les normes $L^2$ et $L^\infty$ de $uStokes$.
Ensuite OPA indique que c'est Stokes en temps qui est désormais résolu. Comme prévu c'est dès la première itération que les erreurs $L^2$ et $L^\infty$ sont proches de 0, *ie* de l'ordre de $10^{-16}$. Nous laissons le lecteur modifier le paramètre $T$ dans check422.cpp afin de tester Stokes en temps sur le nombre d'itération souhaité.

## 4.3 Navier-Stokes

Dans cette section nous comparons notre résolution des équations de Navier-Stokes celle de Freefem++.

Nous faisons tourner notre code sur 250 itérations, puis nous résolvons le même problème avec Freefem++. Afin de comparer notre solution et la solution Freefem++, les

FIGURE 4.6 – ./check422 Th.msh

```
root@pcmint:/home/fitz/Bureau/ProjetC/Navier Stokes# ./check422 Th.msh
 End read 1011 1800 mes =9 mesb = 22
0 0  ; 0 0 || 6 6 :   +0.0025 u1.dx v1.dx   +0.0025 u1.dy v1.dy
0 2  ; 7642 0 || 3 6 :   -1 u1.dx v3
1 1  ; 3821 3821 || 6 6 :   +0.0025 u2.dx v2.dx   +0.0025 u2.dy v2.dy
1 2  ; 7642 3821 || 3 6 :   -1 u2.dy v3
2 0  ; 0 7642 || 6 3 :   -1 u3 v1.dx
2 1  ; 3821 7642 || 6 3 :   -1 u3 v2.dy
2 2  ; 7642 7642 || 3 3 :   -1e-07 u3 v3
0 0  ; 0 0 || 6 6 :   +1 u1 v1
1 1  ; 3821 3821 || 6 6 :   +1 u2 v2
 fh: min 0 max 0
 ds : eps = 1e-08
 min u1:  -0.000804034 max 1.00055
 min u2:  -0.35308 max 0.00547391
 err L2=  1.27978
 err Linf=  1.00055
0 0  ; 0 0 || 6 6 :   +0.0025 u1.dx v1.dx   +0.0025 u1.dy v1.dy   +1 u1 v1
0 2  ; 7642 0 || 3 6 :   -1 u1.dx v3
1 1  ; 3821 3821 || 6 6 :   +0.0025 u2.dx v2.dx   +0.0025 u2.dy v2.dy   +1 u2 v2
1 2  ; 7642 3821 || 3 6 :   -1 u2.dy v3
2 0  ; 0 7642 || 6 3 :   -1 u3 v1.dx
2 1  ; 3821 7642 || 6 3 :   -1 u3 v2.dy
2 2  ; 7642 7642 || 3 3 :   -1e-07 u3 v3
Iteration = 1
TIME = 0.528554
 fh: min 0 max 0
 ds : eps = 1e-08
 min u1:  -0.000804034 max 1.00055
 min u2:  -0.35308 max 0.00547391
 err L2=  6.8936e-16
 err Linf=  7.77156e-16
Iteration = 2
TIME = 0.897686
```

différences en norme $L^2$, $L^1$ et $L^\infty$ sont calculées. Afin de juger la convergence de notre solution nous calculons la norme $L^2$ et $L^\infty$ de l'erreur récursive. Enfin seul les graphiques de la dernière itération sont tracés automatiquement, pour la solution C++ et la solution Freefem++.

Ce test est effectué par check43.cpp. Afin de faire varier le nombre d'itération du test il est convenu de changer la valeur de *lastit* dans les fichiers check43.cpp et check43.edp.

Pour commencer les figures 4.7 à 4.11 sont les graphiques des solutions C++ et Freefem++ au bout de 250 itérations :

Ensuite avec la figure 4.13 on peut trouver les dernières lignes écrites dans la console à la suite de ./check43 Th.msh : on voit que l'erreur récursive est très faible tandis que la différence avec Freefem++ n'est pas infime, de l'ordre du millième.

## 4.4 l'intégrale sur le bord doit être nulle

C'est une conséquence de l'hypothèse d'incompressibilité, comme indiqué section 1.4.1 . On peut voir que tel est le cas pour Freefem++ et notre code à l'aide de la figure 4.13 avec les quantités Différence flux C++ et Différence flux Freefem++.

FIGURE 4.7 – Vitesse C++

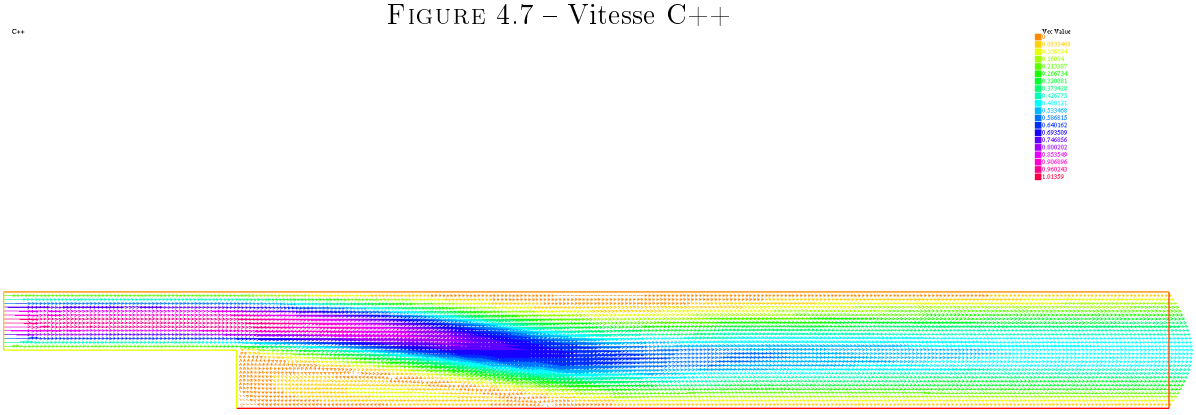



FIGURE 4.8 – Vitesse Freefem++

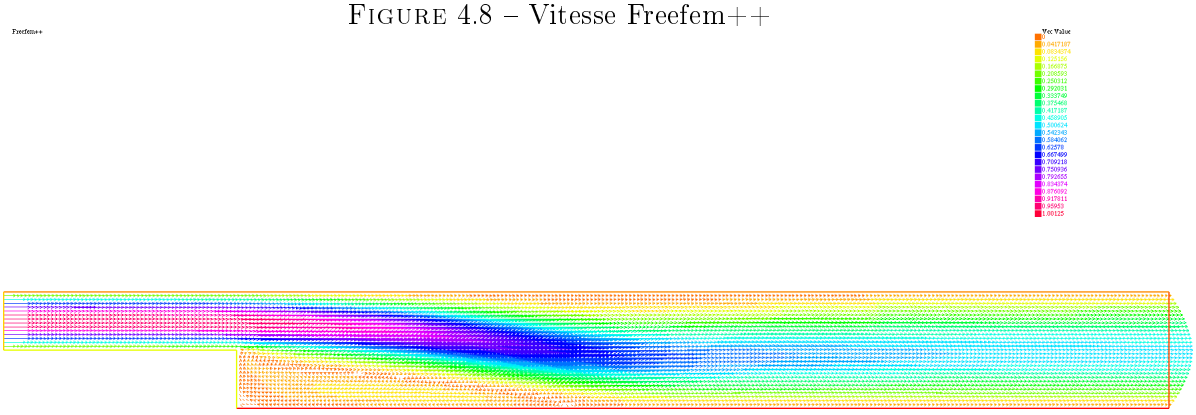



FIGURE 4.9 – Pression C++, 2D

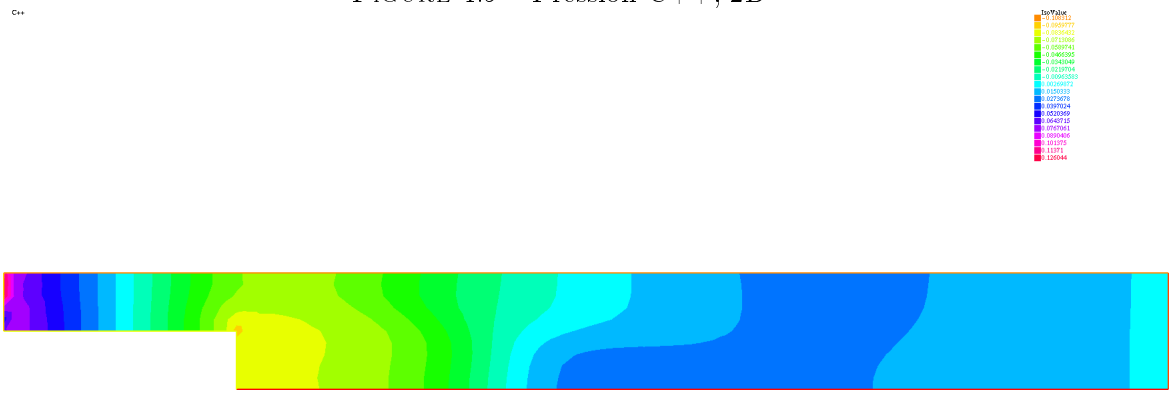


FIGURE 4.10 – Pression Freefem++, 2D

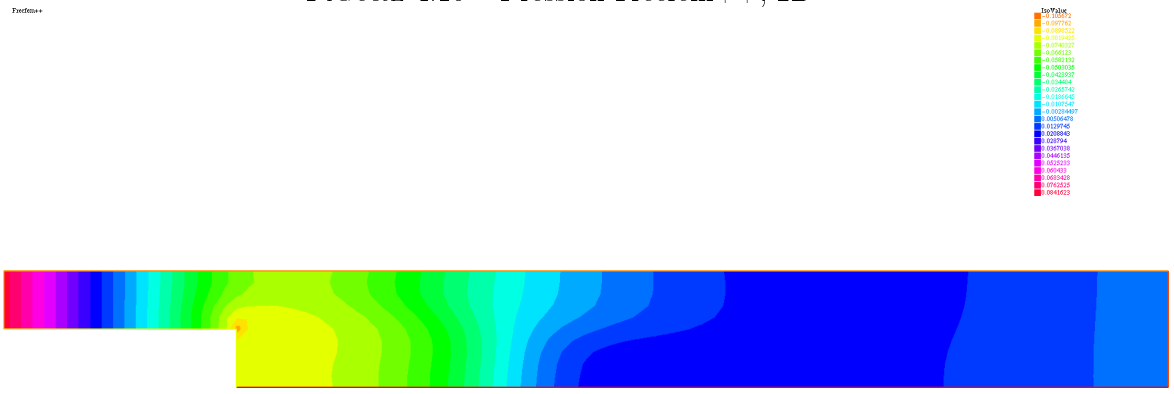


FIGURE 4.11 – Pression C++, 3D

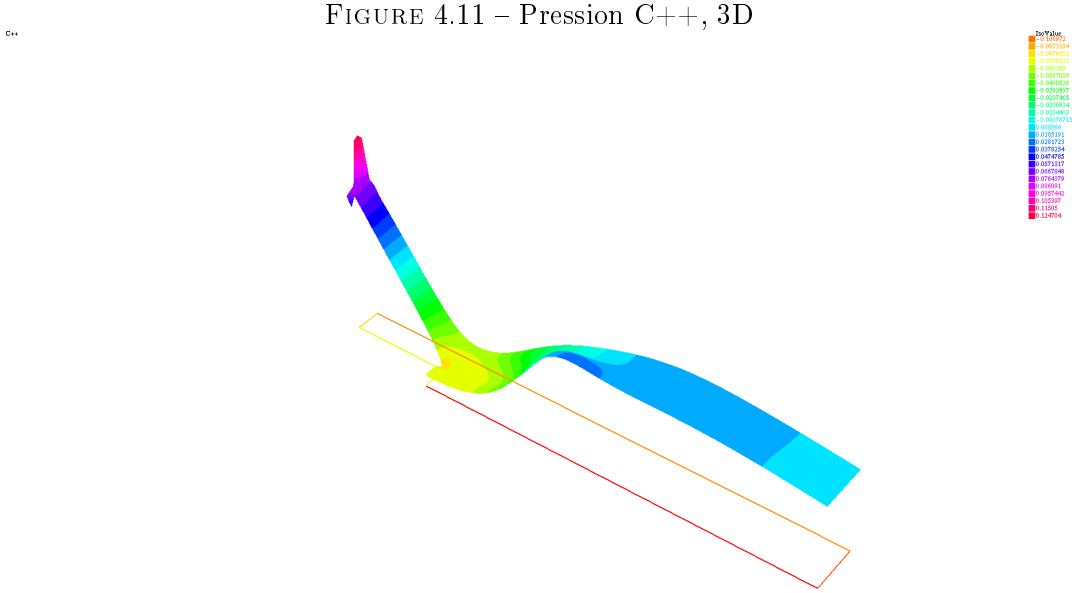


FIGURE 4.12 – Pression Freefem++, 3D

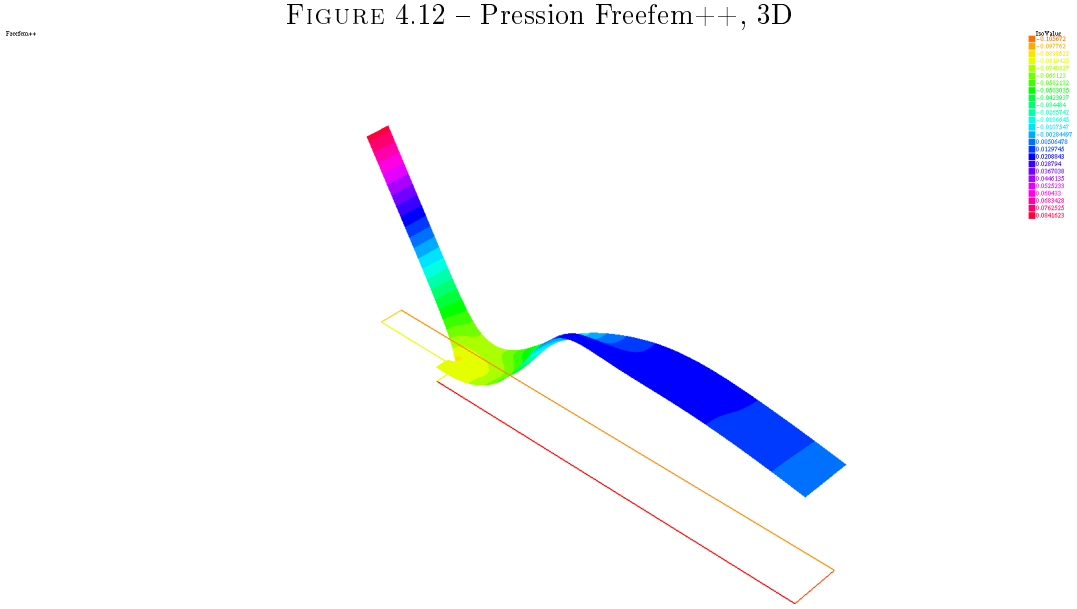


FIGURE 4.13 – Affichage dans la console de ./check43 Th.msh

```
 iter 249
  -- Solve :
          min -0.115593  max 1
          min -0.101175  max 0.0458649
          min -0.101717  max 0.0802074
 iter 250
 Difference flux freefem = 9.8195e-09
 Difference flux c++ = -4e-08
Difference Freefem++ C++ L2 =0.0221809
Difference Freefem++ C++ Linf =0.110358
Difference Freefem++ C++ L1 =0.0154023
times: compile 0.016823s, execution 4.43566s,  mpirank:0
 CodeAlloc : nb ptr  3491,  size :402896 mpirank: 0
Ok: Normal End
Iteration C++250
 Erreur Récursive L2 =  0.000607999
 Erreur Récursive Linf =  0.000636475
TIME = 80.9163
root@pcmint:/home/fitz/Bureau/ProjetC/Navier Stokes#
```

# Table des matières