

Algorithmique numérique
Simulation de l'écoulement d'un fluide

HUMBERT Florent

2 juillet 2008

Table des matières

1	Avant-propos	3
1.1	Introduction	3
1.2	Motivations	3
2	Notations	4
2.1	Ensembles	4
2.2	Fonctions	4
3	Théorie	6
3.1	Les équations de Navier et Stokes	6
3.1.1	Notation de la partie	6
3.1.2	Les équations	6
3.1.3	Intèpretation	7
3.2	Technique de décomposition	8
3.2.1	Théorème de décomposition de Helmholtz-Hodge	8
3.2.2	Préliminaire	9
3.2.3	Méthode et utilisation du théorème	9
3.2.4	Récapitulation des étapes en discret	11
4	Modélisation	12
4.1	Introduction	12
4.2	Un type abstrait de donnée	13
4.3	Discrétisation des opérateurs	14
4.3.1	Opérateur Laplacien	14
4.3.2	Opérateur divergence	14
5	Réalisation	15
5.1	Les conditions aux bords	15
5.1.1	Théorie	15
5.1.2	Algorithme	16
5.2	Évolution : Viscosité	17
5.2.1	Théorie	17
5.2.2	Algorithme	18
5.3	Évolution : Advection	18
5.3.1	Théorie	18
5.3.2	Algorithme	20
5.4	Évolution forces	20
5.4.1	Théorie	21
5.4.2	Algorithme	21

5.5	Calcul de ϕ et de \mathbf{u}_d	21
5.5.1	Théorie	21
5.5.2	Algorithme	22
5.6	Détermination de \mathbf{u}	23
5.6.1	Attention :	23
5.7	Exemple de résultats	24
6	Problème de diffusion	26
6.1	Théorie	26
6.1.1	Notation	26
6.1.2	Équations	27
6.1.3	Interprétation	27
6.1.4	Résolution	27
6.2	Réalisation et algorithmique	28
6.2.1	Les conditions aux bords	28
6.2.2	Calcul de l'advection	28
6.2.3	Calcul de la viscosité	29
6.2.4	Calcul général	29
6.3	Exemple d'application	30
7	Ajout d'obstacles	32
7.1	Réalisation et algorithme	32
7.1.1	Bords pour un champs de scalaires	32
7.1.2	Champ de vecteurs	33
7.1.3	Cas général et critiques	34
7.2	Exemples et applications	34
8	Suppléments	36
8.1	Calcul de la pression	36
8.1.1	Théorie, rappel	36
8.1.2	Algorithme	36
8.1.3	Exemple d'application	37
8.2	Détection de tourbillons	37
8.2.1	Théorie	38
8.2.2	Algorithme	38
8.2.3	Exemple d'application	39
9	Annexe	41
9.1	Méthode de résolution de Gauss-Seidel	41
10	Remerciements	42

Chapitre 1

Avant-propos

1.1 Introduction

L'étude de la simulation des écoulements de fluide est présente dans de nombreux domaines, que ce soit en aérodynamique, en hydrodynamique ou encore en météorologie.

Ces études peuvent se faire directement en pratique (étude en soufflerie) ou bien numériquement. On connaît les équations régissant l'écoulement des fluides, mais à l'heure actuelle, l'étude de celles-ci d'un point de vue purement théorique est très difficile. C'est pourquoi, avec l'avènement de l'informatique, nous disposons d'une ressource supplémentaire nous permettant de simuler l'écoulement des fluides et nous permettant de déterminer des solutions fiables.

Je présenterai ici des algorithmes assez « simples » permettant de simuler les écoulements de fluides dans un domaine fermé (comme un rectangle ou une pièce). Dans l'industrie, on travaille souvent avec des algorithmes plus complexes prenant en compte les formes des objets que l'on souhaite étudier (par exemple une aile d'avion), cela faisant intervenir la notion de génération de maillage.

Mon approche donnera des résultats assez généraux qui ne seront pas aussi performants que les méthodes industrielles. Mais ces algorithmes auront l'avantage d'être rapides et facilement implémentables.

Ce cours peut avoir une utilité dans le domaine du jeux-vidéo (notamment grâce à la rapidité d'exécution des algorithmes). En intégrant une méthode de calcul d'écoulement dans un moteur de jeu, on peut rendre de plus en plus réel un monde purement virtuel. Avec les algorithmes proposés, on peut s'en servir pour modéliser un fluide ou encore de la fumée. Avec quelques ajouts, on pourrait également simuler la formation de nuage ou une explosion.

1.2 Motivations

L'écriture de ce cours a été motivé par le manque de documentation libre et en français dans ce domaine. La documentation française que l'on trouve implémente en général des méthodes de résolution très précises et ne permettant pas de faire de calcul en temps réel. De plus, pour la plupart, seule la théorie est développée et aucun algorithme n'est présenté.

Chapitre 2

Notations

Avant de commencer le cours, voici une petite liste de notations (assez classique) utilisées dans ce cours.

2.1 Ensembles

Notation 1 (Ensemble de fonctions) On notera $\mathcal{F}(E, F)$ l'ensemble des fonctions de l'ensemble E dans l'ensemble F .

Notation 2 (Ensembles classiques) Les ensembles \mathbb{C} , \mathbb{R} , \mathbb{N} et \mathbb{Z} désignent respectivement l'ensemble des complexes, des réels, des entiers naturels et des entiers relatifs.

L'ensemble \mathbb{K} désignera \mathbb{C} où \mathbb{R} .

Notation 3 (Ensemble d'entiers) On note $[[n, m]]$ l'ensemble des entiers compris entre n et m . (Donc $[[n, m]] = [n, m] \cap \mathbb{Z}$)

Notation 4 (Frontière) Soit F un ensemble borné. On note ∂F la frontière de F .

2.2 Fonctions

Notation 5 (Champs de vecteur) Si f est une application à valeurs dans \mathbb{R}^2 , on l'écrira en gras : \mathbf{f}

Notation 6 (Gradient) Soit $f \in C^1(\Omega, \mathbb{R})$ où Ω est un ouvert de \mathbb{R}^2 , on note : $\vec{\nabla}(f)$ le gradient de f défini par :

$$\forall (x, y) \in \Omega, \vec{\nabla}(f)(x, y) = \left(\frac{\partial f}{\partial x}(x, y), \frac{\partial f}{\partial y}(x, y) \right)$$

Notation 7 (Divergence) Soit $\mathbf{f} = (f_x, f_y) \in C^1(\Omega, \mathbb{R}^2)$ où Ω est un ouvert de \mathbb{R}^2 , on note : $\operatorname{div} \mathbf{f}$ la divergence de \mathbf{f} définie par :

$$\forall (x, y) \in \Omega, \operatorname{div} \mathbf{f}(x, y) = \frac{\partial f_x}{\partial x}(x, y) + \frac{\partial f_y}{\partial y}(x, y)$$

Notation 8 (Produit scalaire) On notera (\cdot) le produit scalaire usuel entre deux vecteurs de \mathbb{R}^n défini par :

$$\forall u \in \mathbb{R}^n, \forall v \in \mathbb{R}^n, (u \cdot v) = \sum_{i=1}^n u(i) \cdot v(i)$$

Notation 9 (Opérateur Laplacien) Soit f une application C^2 définie sur un ouvert Ω de \mathbb{R}^2 à valeur dans \mathbb{R} , on note Δ l'opérateur Laplacien défini par :

$$\forall (x, y) \in \Omega, \Delta(f)(x, y) = \frac{\partial^2 f}{\partial x^2}(x, y) + \frac{\partial^2 f}{\partial y^2}(x, y)$$

Notation 10 (Dérivée partielle) Soit \vec{u} un vecteur de \mathbb{R}^2 non nul, Soit $f \in C^1(\mathbb{R}^2, \mathbb{R})$.

On notera $\frac{\partial f}{\partial \vec{u}}$ l'application de \mathbb{R}^2 dans \mathbb{R} définie par :

$$\forall (x, y) \in \mathbb{R}^2, \frac{\partial f}{\partial \vec{u}}(x, y) = \left(\vec{\nabla} (f) \cdot \vec{u} \right)$$

Chapitre 3

Théorie

Pour permettre de simuler l'écoulement d'un fluide, nous allons utiliser les équations de Navier et Stokes pour un fluide incompressible. Ces équations peuvent-être utilisées pour déterminer les mouvements de l'air dans l'atmosphère (ou contre des objets) ou encore les mouvements de l'eau (par exemple pour les courants océaniques).

Actuellement de nombreuses recherches sont effectuées sur ces équations, mais les chercheurs n'ont pas encore réussi à déterminer de solution analytique (solution exacte) à celles-ci. On ne sait même pas prouver l'existence d'une solution dans le cas général.

Cela dit, avec la puissance de calcul des ordinateurs actuels, il est possible de déterminer des approximations raisonnablement bonnes.

3.1 Les équations de Navier et Stokes

3.1.1 Notation de la partie

Dans cette partie,

- Ω désigne un ouvert de \mathbb{R}^2 borné régulier
 - \mathbf{u}^0 une application de Ω à valeurs dans \mathbb{R}^2 de classe C^∞
 - $\mathbf{f} : (x, y, t) \in \Omega \times \mathbb{R}^+ \mapsto \mathbb{R}^2$ une application de classe C^∞ que l'on connaîtra.
 - $\mathbf{u} : (x, y, t) \in \Omega \times \mathbb{R}^+ \mapsto \mathbb{R}^2$ et $p : (x, y, t) \in \Omega \times \mathbb{R}^+ \mapsto \mathbb{R}$ des applications de classe C^∞ prolongeable par continuité sur $\partial\Omega$.
- On utilisera parfois la notation $\mathbf{u}(x, y, t) = (u_1(x, t), u_2(y, t))$.
- ν et ρ deux réels non nuls

3.1.2 Les équations

Les équations de Navier et Stokes s'écrivent comme suit :

Équations générales

$$\forall i \in \{1, 2\}, \frac{\partial u_i}{\partial t} + \sum_{j=1}^2 u_j \frac{\partial u_i}{\partial x_j} = \nu \Delta u_i - \frac{1}{\rho} \frac{\partial p}{\partial x_i} + f_i \quad (3.1)$$

$$\forall (x, y) \in \Omega, \operatorname{div} \mathbf{u} = \sum_{i=1}^2 \frac{\partial u_i}{\partial x_i}(x, y) = 0 \quad (3.2)$$

Il arrive que l'on note l'équation 3.1 avec une notation vectorielle :

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} = \nu \Delta \mathbf{u} - \frac{1}{\rho} \vec{\nabla} (p) + \mathbf{f}$$

Conditions aux bords

Les conditions sur le bord $\partial\Omega$ sont :

$$\forall (x, y) \in \partial\Omega, \forall t \geq 0, (\mathbf{u}(x, y, t) \cdot \mathbf{n}) = 0 \quad (3.3)$$

Où \mathbf{n} désigne le vecteur normal à $\partial\Omega$ au point (x, y) .

Conditions initiales

Les conditions initiales sont :

$$\forall (x, y) \in \Omega, \mathbf{u}(x, y, 0) = \mathbf{u}^0(x, y) \quad (3.4)$$

Ce que l'on sait et ce que l'on ne sait pas

Dans ces équations, les inconnus sont :

- le champ de vecteur \mathbf{u} ;
- le champ scalaire p .

Et on connaît tout le reste (\mathbf{f} , \mathbf{u}^0 et les constantes ν et ρ).

3.1.3 Intèpretation

Vitesse

Le terme \mathbf{u} correspond au vecteur vitesse du fluide en un certain point.

Les termes d'accélération

Pour déterminer ces équations, les scientifiques ont utilisés la relation fondamentale de la dynamique $\sum \vec{F} = m \vec{a}$. Le terme d'accélération correspond pour les équations de Navier et Stokes au terme :

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u}$$

On appelle le terme $(\mathbf{u} \cdot \nabla) \mathbf{u}$ l'advection. L'advection est définie comme étant le transport d'une quantité par un flux vectoriel (Par exemple le transport d'une plume sur une rivière).

Les forces de viscosité

Les forces de viscosité correspondent au terme :

$$\nu \Delta \mathbf{u}$$

La viscosité décrit comment un fluide s'écoule. Lorsque la viscosité est importante, le fluide a plus de difficulté à s'écouler (exemple : l'huile) tandis que lorsque la viscosité est faible, le fluide s'écoule bien (exemple : eau).

Le coefficient de viscosité (s'appelant également la viscosité cinématique du fluide) ν est différent selon la nature du fluide (ou du gaz) et de sa température. Par exemple (source *Wikipedia.fr*) :

- eau à $20,2^\circ$: $\nu = 10^{-3}$
- air à 0° : $\nu = 17,1 \times 10^{-6}$
- miel : $\nu = 10$
- huile d'olive : ν de 81×10^{-3} à 100×10^{-3}

Les forces de pression

Les forces de pression correspondent aux termes :

$$\frac{1}{\rho} \vec{\nabla} (p)$$

ρ désigne la masse volumique du fluide.

Une importante difficulté dans ces équations est que p est inconnu mais qu'on ne le connaît que par l'intermédiaire de son gradient.

Les forces extérieures

Les forces extérieures correspondent aux termes \mathbf{f} . Elles correspondent en général aux forces de gravité \vec{g} . Comme \vec{g} est orienté vers le bas et comme nous travaillons en deux dimensions, \mathbf{f} sera *a priori* nulle.

L'incompressibilité

Le fait que le fluide soit incompressible se traduit par la relation

$$\operatorname{div} \mathbf{u} = 0$$

On dit qu'un fluide est incompressible « si son volume est constant sous l'action d'une pression externe » (source *Wikipedia fr*). Cela signifie que si l'on place ce fluide dans une seringue (fermée), il sera impossible de le comprimer.

3.2 Technique de décomposition

Afin de déterminer une bonne approximation de la solution, nous allons utiliser une technique classique qui consiste à couper le problème en deux équations. La première permettra de déterminer d'un coup \mathbf{u} sans avoir de connaissance sur p et la seconde permettra de déterminer $\vec{\nabla} (p)$. Cette méthode permet de supprimer une difficulté majeure : on ne connaît la pression que par son gradient.

3.2.1 Théorème de décomposition de Helmholtz-Hodge

Nous allons voir un théorème qui permet de « simplifier » les équations de Navier et Stokes.

Je n'ai malheureusement pas trouvé une formulation sans faille de ce théorème, je vais en exposer une qui sera peut-être plus restrictive sur la nature des fonctions.

Théorème 1 (Décomposition de Helmholtz-Hodge) *Soit Ω un ouvert de \mathbb{R}^2 borné et régulier.*

Soit \mathbf{f} un champ vectoriel de classe C^∞ de Ω dans \mathbb{R}^2 prolongeable par continuité sur $\partial\Omega$.

Alors il existe une et une seule décomposition de \mathbf{f} de la forme :

$$\mathbf{f} = \mathbf{g} + \vec{\nabla} (h) \text{ sur } \Omega$$

Où :

- \mathbf{g} est de classe C^∞ et vérifie $\operatorname{div} \mathbf{g} = 0$ sur Ω et $(\mathbf{g}(\mathbf{x}, \mathbf{y}) \cdot \mathbf{n}) = 0$ sur $\partial\Omega$ où \mathbf{n} est le vecteur normal à $\partial\Omega$ en (x, y) .
De plus \mathbf{g} est prolongeable par continuité sur le bord de Ω
- h est de classe C^∞ de \mathbb{R}^2 dans \mathbb{R}
De plus, h est prolongeable par continuité sur le bord.

Lemme 1 (Projection) *En utilisant le théorème précédent, on montre qu'il existe une unique projection \mathcal{P} de l'espace des champs vectoriels de classe C^∞ dans l'espace des champs vectoriels de divergence nulle.*

Ainsi, pour tout \mathbf{f} de classe C^∞ , \mathbf{f} peut s'écrire de manière unique sous la forme :

$$\mathbf{f} = \mathbf{g} + \vec{\nabla}(h) \text{ (plus les conditions aux bords), on a donc :}$$

$$\mathcal{P}(\mathbf{f}) = \mathbf{g}$$

3.2.2 Préliminaire

Nous allons par la suite utiliser deux propriétés.

Propriété 1 *La projection du champs $\vec{\nabla}(p)$ est nulle.*

$$\mathcal{P}\left(\vec{\nabla}(p)\right) = 0$$

Cette propriété se montre facilement en utilisant le théorème précédent.

Propriété 2 *La projection du champs \mathbf{u} est à égale à \mathbf{u} .*

$$\mathcal{P}(\mathbf{u}) = \mathbf{u}$$

Cette propriété est évidente puisque \mathbf{u} est de divergence nulle et vérifie $(\mathbf{u} \cdot \mathbf{n}) = 0$
On peut en déduire l'équation suivante :

Propriété 3 (Projection de l'équation de Navier et Stokes)

$$\frac{\partial \mathbf{u}}{\partial t} = \mathcal{P}(-(\mathbf{u} \cdot \nabla) \mathbf{u} + \nu \Delta \mathbf{u} + \mathbf{f})$$

3.2.3 Méthode et utilisation du théorème

Fixons le réel $t_0 \geq 0$.

Nous allons décomposer l'équation en plusieurs petites équations.

1^{re} partie Nous allons chercher à résoudre localement l'équation suivante sur Ω :

$$\frac{\partial \tilde{\mathbf{u}}}{\partial t}(x, y, t) = -(\mathbf{u}(x, y, t) \cdot \nabla) \mathbf{u}(x, y, t) + \nu \Delta \mathbf{u}(x, y, t) + \mathbf{f}(x, y, t) \quad (3.5)$$

Avec comme conditions initiales :

$$\tilde{\mathbf{u}}(x, y, t_0) = \mathbf{u}(x, y, t_0) \quad (3.6)$$

Et comme conditions aux bords¹ :

$$\forall (x, y) \in \partial\Omega, (\tilde{\mathbf{u}}(x, y, t) \cdot \mathbf{n}) = 0 \quad (3.7)$$

¹À noter que le champ est continu sur le bord

NB : Il faut bien noter que $\tilde{\mathbf{u}}$ dépend de t_0 .

J'ai parlé de résoudre localement car on ne connaîtra à un instant t_0 , que le champ \mathbf{u} pour tout $t \leq t_0$. On déterminera (de manière approximative) $\tilde{\mathbf{u}}$ sur un petit intervalle $[t_0, t_0 + \epsilon]$.

Comme

$$\frac{\partial(\tilde{\mathbf{u}})}{\partial t} = -(\mathbf{u} \cdot \nabla) \mathbf{u} + \nu \Delta \mathbf{u} + \mathbf{f}$$

En utilisant les propriétés précédentes, en projetant on obtient :

$$\mathcal{P} \left(\frac{\partial \tilde{\mathbf{u}}}{\partial t} \right) = \frac{\partial \mathcal{P}(\tilde{\mathbf{u}})}{\partial t} = \mathcal{P}(-(\mathbf{u} \cdot \nabla) \mathbf{u} + \nu \Delta \mathbf{u} + \mathbf{f}) = \frac{\partial \mathbf{u}}{\partial t}$$

On doit ainsi avoir

$$\forall t \geq 0, \mathcal{P}(\tilde{\mathbf{u}}) = \mathbf{u} + \mathbf{k}$$

Où \mathbf{k} est un champ de vecteur constant par rapport à t et de divergence nulle (il suffit d'appliquer deux fois la projection et d'utiliser la propriété $\mathcal{P}^2 = \mathcal{P}$).

Les conditions initiales sur $\tilde{\mathbf{u}}$ font que le champ \mathbf{k} est nul.

On obtient donc la relation :

$$\mathcal{P}(\tilde{\mathbf{u}}) = \mathbf{u} \tag{3.8}$$

2^e partie Posons :

$$\tilde{\mathbf{u}} = \mathbf{u} + \vec{\nabla}(\phi) \tag{3.9}$$

Avec ϕ de classe C^∞ . Ce qui est possible grâce au théorème de Helmholtz-Hodge.

En prenant la divergence de l'expression $\tilde{\mathbf{u}} = \mathbf{u} + \vec{\nabla}(\phi)$, comme $\text{div } \mathbf{u} = 0$, nous obtenons :

$$\text{div } \tilde{\mathbf{u}} = \Delta \phi \text{ sur } \Omega$$

Ainsi, connaissant $\tilde{\mathbf{u}}$, on peut déterminer ϕ en résolvant l'équation de Poisson suivante :

$$\Delta \phi = \text{div } \tilde{\mathbf{u}} \text{ sur } \Omega$$

Pour pouvoir la résoudre correctement, il faut les conditions sur $\partial\Omega$.

Soit $(x, y) \in \partial\Omega$ et n le vecteur normal au point là. En utilisant certains prolongements par continuité et les conditions de \mathbf{u} sur le bord, on obtient :

$$\frac{\partial \phi}{\partial n}(x, y) = \left(\vec{\nabla}(\phi)(x, y) \cdot n \right) = (\tilde{\mathbf{u}}(x, y) - \mathbf{u}(x, y) \cdot n) = (\tilde{\mathbf{u}}(x, y) \cdot n) = 0$$

Une équation discrétisée de ce problème peut se résoudre de manière assez simple et rapide en utilisant la méthode de relaxation de Gauss-Seidel.

On peut ainsi déterminer $\tilde{\mathbf{u}}$ et ϕ , on peut ainsi obtenir \mathbf{u} par la relation :

$$\mathbf{u} = \tilde{\mathbf{u}} - \vec{\nabla}(\phi) \tag{3.10}$$

3^e partie On dispose à présent d'une méthode itérative permettant de déterminer le champ de vecteur \mathbf{u} sur un petit intervalle.

Mais nous n'avons pas déterminé la pression p (mais ce n'est pas obligatoire). Une méthode simple consiste à appliquer l'opérateur de divergence à l'équation de Navier et Stokes. Ce qui donne :

$$\frac{1}{\rho} \Delta p = \operatorname{div}((\mathbf{u} \cdot \nabla) \mathbf{u} - \mathbf{f})$$

Cette équation est encore une équation de Poisson, elle peut être résolue par la méthode de relaxation de Gauss-Seidel.

3.2.4 Récapitulation des étapes en discret

On discrétise le temps par un taux Δ_t assez petit.

On suppose qu'à un temps $t = i\Delta_t$, on connaisse \mathbf{u} sur Ω .

Recherche de $\tilde{\mathbf{u}}$

On pose : $\tilde{\mathbf{u}}(x, y, i\Delta_t) = \mathbf{u}(x, y, i\Delta_t)$.

On détermine $\tilde{\mathbf{u}}(x, y, (i+1)\Delta_t)$ en utilisant la relation :

$$\frac{\partial(\tilde{\mathbf{u}})}{\partial t} = -(\mathbf{u} \cdot \nabla) \mathbf{u} + \nu \Delta \mathbf{u} + \mathbf{f}$$

Et la condition sur les bords.

Calcul de ϕ

On détermine :

$$\operatorname{div} \tilde{\mathbf{u}}(x, y, (i+1)\Delta_t)$$

On résout (avec les conditions sur les bords) :

$$\Delta \phi(x, y, (i+1)\Delta_t) = \operatorname{div} \tilde{\mathbf{u}}(x, y, (i+1)\Delta_t)$$

Détermination de \mathbf{u}

On peut à présent déterminer $\mathbf{u}(x, y, (i+1)\Delta_t)$ par la relation :

$$\mathbf{u}(x, y, (i+1)\Delta_t) = \tilde{\mathbf{u}}(x, y, (i+1)\Delta_t) - \overrightarrow{\nabla}(\phi)(x, y, (i+1)\Delta_t)$$

Chapitre 4

Modélisation

Dans cette partie, nous allons modéliser notre problème afin de pouvoir l'implémenter sur un ordinateur. Comme nous l'avons vu dans les équations de Navier et Stokes, le fluide \mathbf{u} est défini de manière continue. Comme nous ne pouvons pas résoudre analytiquement ces équations, il nous faudra discrétiser le fluide.

4.1 Introduction

Afin de simplifier le problème, nous supposons que le fluide n'est défini que dans un domaine rectangulaire. Dans ce cas, si le fluide est défini dans un domaine du type $\Omega =]0, 1[\times]0, d[$. (le fait que la borne supérieure soit 1 ne change pas les calculs à cause du pas de discrétisation).

Nous allons ainsi discrétiser ce domaine par un pas δ_x pour $]0, 1[$ et par un pas δ_y pour $]0, d[$. Notons \mathbf{u}_d le champ de vitesse discrétisé, nous aurons ainsi la relation suivante :

$$\forall (i, j) \in [[1, 1/\delta_x - 1]] \times [[1, 1/\delta_y - 1]], \mathbf{u}_d(i, j) = \mathbf{u}(i\delta_x, j\delta_y) \quad (4.1)$$

Ici, nous n'avons pas défini la valeur de \mathbf{u}_d aux bornes (0 ou $1/\delta_x - 1$ par exemple). Nous allons utiliser une astuce en prolongeant \mathbf{u}_d aux bornes, ces valeurs correspondront aux valeurs limites des bornes de \mathbf{u} .

Vous pouvez voir par exemple la figure 4.1. Je n'ai placé les vecteurs vitesses uniquement à gauche de la grille.

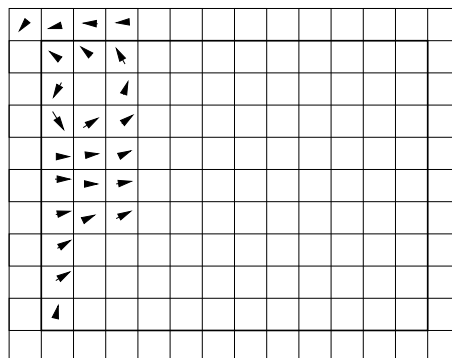


FIG. 4.1 – Grille de vecteurs vitesse avec les bords

Afin de simplifier les calculs et afin de correspondre à la figure 4.1, nous allons supposer $\delta_x = \delta_y = \delta$. Ainsi, le fluide sera discrétisé dans un rectangle de taille $N \times M$ où $N = 1/\delta_x$ et $M = 1/\delta_y$.

Comme il se posera des problèmes lors des calculs avec les conditions sur les bords, nous allons entourer notre tableau d'une ligne supplémentaire. Nous aurons donc un rectangle de taille $(N + 2) \times (M + 2)$.

Remarque : Ici, nous avons simplement découper l'espace en petits éléments de volume toujours de même taille. Dans l'industrie, pour effectuer une approximation de calculs plus précis, le découpage se fait relativement à l'objet que l'on veut tester (par exemple une aile d'avion). On réalise un découpage selon cet objet et on effectue ensuite les calculs par rapport à celui-ci. Cette méthode a l'inconvénient de dépendre de l'objet, mais a l'avantage d'être beaucoup plus précise.

4.2 Un type abstrait de donnée

Afin d'écrire nos algorithmes, nous allons définir un type abstrait de donnée correspondant à un champ de vecteurs discrétisé \mathbf{u}_d que l'on nommera **ChampVecteurs**.

Nous supposons l'existence des opérations de bases suivantes :

creer(Entier n, Entier m) → ChampsVecteurs Permettra la création d'un champ de vecteurs de taille $(n + 2) \times (m + 2)$.

tailleX(ChampVecteurs) → Entier Retournera la taille horizontale du champ.

tailleY(ChampVecteurs) → Entier Retournera la taille verticale du champ.

pas(ChampVecteurs) → Réel On pourra éventuellement définir un opérateur qui donne le pas de discrétisation spatiale du champ de vecteurs (sera en fait définie par $1/(\text{tailleX}(\mathbf{u}) - 2)$).

Lecture et écriture Afin de simplifier les écritures, nous n'allons pas définir des fonctions du type *lire* ou *ecrire*.

Nous écrirons directement :

$$\text{ud}(2,3) = \text{ud}(4,5) + \text{ud}(2,1)$$

Il faut bien noter que $\text{ud}(4,5)$ est un vecteur de dimension 2, donc on supposera que l'opération $+$ additionnera deux vecteurs.

ux et uy Il arrivera que l'on ne souhaite effectuer des opérations que sur une dimension de \mathbf{u} . Nous noterons alors ux le champ scalaires correspondant à l'axe des x et uy le champ correspondant à l'axe des y .

Remarque : À noter que lorsque l'on implémentera ce type abstrait de données, pour des raisons d'efficacité, il sera préférable de ne pas créer un tableau de vecteurs, mais deux tableaux de réels (correspondant chacun à une partie des vecteurs). Dans ce cas, les algorithmes seront légèrement différents.

4.3 Discrétisation des opérateurs

Par la suite, nous aurons besoin de discrétiser certains opérateurs. Nous allons les rappeler dans cette section.

Nous noterons $\Omega_d = [[1, n]] \times [[1, m]]$ et $\partial\Omega_d$ le bord du rectangle (soit $[[0, n+1]] \times [[0, m+1]] - \Omega_d$).

4.3.1 Opérateur Laplacien

Cet opérateur peut être discrétisé par la formule suivante :

$$\forall (i, j) \in \Omega_d, \Delta \mathbf{u}_d(i, j) = \frac{\mathbf{u}_d(i+1, j) + \mathbf{u}_d(i-1, j) - 2\mathbf{u}_d(i, j)}{\delta_x^2} + \frac{\mathbf{u}_d(i, j+1) + \mathbf{u}_d(i, j-1) - 2\mathbf{u}_d(i, j)}{\delta_y^2}$$

Comme nous avons supposé $\delta_x = \delta_y = \delta$.

Nous pouvons un peu simplifier l'expression par :

$$\forall (i, j) \in \Omega_d, \Delta \mathbf{u}_d(i, j) = \frac{\mathbf{u}_d(i+1, j) + \mathbf{u}_d(i-1, j) + \mathbf{u}_d(i, j+1) + \mathbf{u}_d(i, j-1) - 4\mathbf{u}_d(i, j)}{\delta^2}$$

4.3.2 Opérateur divergence

Cet opérateur peut être discrétisé par la formule suivante :

$$\forall (i, j) \in \Omega_d, \operatorname{div} \mathbf{u}_d(i, j) = \frac{\mathbf{u}_d(i+1, j) + \mathbf{u}_d(i-1, j)}{2\delta_x} + \frac{\mathbf{u}_d(i, j+1) + \mathbf{u}_d(i, j-1)}{2\delta_y}$$

Que l'on peut simplifier par :

$$\forall (i, j) \in \Omega_d, \operatorname{div} \mathbf{u}_d(i, j) = \frac{\mathbf{u}_d(i+1, j) + \mathbf{u}_d(i-1, j) + \mathbf{u}_d(i, j+1) + \mathbf{u}_d(i, j-1)}{2\delta}$$

Comme $\operatorname{div} \mathbf{u}_d$ est en général continue mais ne nécessite pas de condition particulière sur les bords, on peut définir sa valeur sur le bord $\partial\Omega_d$ comme valant la valeur la plus proche sur le rectangle.

Par exemple, $\operatorname{div} \mathbf{u}_d(i, 0) = \operatorname{div} \mathbf{u}_d(i, 1)$ où $\operatorname{div} \mathbf{u}_d(n+1, j) = \operatorname{div} \mathbf{u}_d(n, j)$.

Chapitre 5

Réalisation

Dans cette partie, nous allons implémenter les algorithmes nous permettant de simuler l'écoulement d'un fluide, puisque nous connaissons à présent les équations à résoudre.

Dans toute cette partie, nous noterons :

- t_0 un temps quelconque où l'on connaîtra \mathbf{u}_d .
- u_{xd} et u_{yd} les champs scalaires de \mathbf{u}_d projetés sur les axes x et y .
- Δ_t le taux de discrétisation temporel
- δ le taux de discrétisation spatial
- n et m la pseudo taille du rectangle où est défini le champ discrétisé \mathbf{u}_d .
- $\Omega_d = [[1, n]] \times [[1, m]]$ le rectangle.
- $\partial\Omega_d$ le bord du rectangle (soit $[[0, n + 1]] \times [[0, m + 1]] - \Omega_d$).
- \mathbf{u}_d sera donc défini sur $[[0, n + 1]] \times [[0, m + 1]]$. Le tour correspondra à la frontière.

But : Notre but ultime dans ce chapitre est de réussir à définir en tout point le champ de vecteurs \mathbf{u}_d à l'instant $t_0 + \Delta_t$. Nous allons résoudre dans cette partie les problèmes définis précédemment un par un. Chaque section définira et résoudra un de ces problèmes.

5.1 Les conditions aux bords

Lors des calculs que nous allons effectuer, nous allons connaître \mathbf{u}_d à un instant $t_0 + \Delta_t$ uniquement sur $[[1, n]] \times [[1, m]]$. En réalité, le contour de \mathbf{u}_d aura bien des valeurs mais ne respectera pas les conditions aux bords. C'est pourquoi nous effectuerons à ce moment là, une réévaluation des valeurs sur les bords.

5.1.1 Théorie

Heureusement pour nous, la pièce Ω_d où s'écoule le fluide est rectangulaire, les vecteurs \mathbf{n} orthogonaux à $\partial\Omega_d$ sont simples à déterminer.

Par exemple, sur le bord ouest, \mathbf{n} sera égal à $(1, 0)$.

Bord ouest

Nous souhaiterions avoir $\forall i \in [[1, m]], (\mathbf{u}_d(0, i) \cdot \mathbf{n}) = 0$. C'est à dire : $u_{xd}(0, i) = 0$

Pour rendre \mathbf{u} continue sur les bords, il faudra forcer la valeur u_{yd} à avoir la valeur : $u_{xd}(0, i) = u_{yd}(1, i)$

Bord est

Ici, $\mathbf{n} = (-1, 0)$. $u_{xd}(n+1, i) = 0$ et $u_{yd}(n+1, i) = u_{yd}(n, i)$

Bord nord

Ici, $\mathbf{n} = (0, -1)$. $u_{yd}(i, 0) = 0$ et $u_{xd}(i, 0) = u_{xd}(i, 0)$

Bord sud

Ici, $\mathbf{n} = (0, 1)$. $u_{yd}(i, m+1) = 0$ et $u_{xd}(i, m+1) = u_{xd}(i, m)$

Côtés

Pour les côtés, on fera simplement la moyenne des valeurs autour.

$$\begin{aligned} \mathbf{u}_d(0, 0) &= \frac{1}{2} (\mathbf{u}_d(0, 1) + \mathbf{u}_d(1, 0)) \\ \mathbf{u}_d(n+1, m+1) &= \frac{1}{2} (\mathbf{u}_d(n+1, m) + \mathbf{u}_d(n, m+1)) \\ \mathbf{u}_d(0, m+1) &= \frac{1}{2} (\mathbf{u}_d(0, m) + \mathbf{u}_d(1, m+1)) \\ \mathbf{u}_d(n+1, 0) &= \frac{1}{2} (\mathbf{u}_d(n, 0) + \mathbf{u}_d(n+1, 1)) \end{aligned}$$

5.1.2 Algorithme

On peut maintenant écrire un algorithme qui force les valeurs sur les bords d'un champ de vecteurs vitesse afin qu'il respecte les conditions sur les bords.

Procédure forcerBords(ChampVecteurs u)

```
Entier i
Entier j
Entier n = tailleX(u) -2
Entier m = tailleY(u)-2

Pour j=1 à m
    u(0,j) = (0, uy(1,j))
    u(n+1,j) = (0, uy(n+1,j))
Pour i=1 à n
    u(i,0) = (ux(i,1), 0)
    u(i,m+1) = (ux(i,m+1),0)
u(0,0) = 0.5 * (u(0,1)+u(1,0))
u(n+1,m+1) = 0.5 * (u(n+1,m)+ u(n,m+1))
u(0,m+1) = 0.5 * (u(0,m)+ u(1,m+1))
u(n+1,0) = 0.5 * (u(n,0)+ u(n+1,1))
```

À noter que l'on réalise un effet de bords sur le champ de vecteur u.

5.2 Évolution : Viscosité

Nous avons vu dans la partie théorique que nous devons dans un premier temps résoudre l'équation :

$$\frac{\partial(\tilde{\mathbf{u}})}{\partial t} = -(\mathbf{u} \cdot \nabla) \mathbf{u} + \nu \Delta \mathbf{u} + \mathbf{f}$$

Pour cela, nous allons diviser ce problème en 3 (ou 2) sous problèmes. Tout d'abord la résolution locale de :

$$\frac{\partial(\tilde{\mathbf{u}}_1)}{\partial t} = -(\mathbf{u} \cdot \nabla) \mathbf{u}$$

Et la résolution locale de :

$$\frac{\partial(\tilde{\mathbf{u}}_2)}{\partial t} = \nu \Delta \mathbf{u}$$

5.2.1 Théorie

Nous souhaitons résoudre localement l'équation :

$$\frac{\partial(\tilde{\mathbf{u}}_d)}{\partial t} = \nu \Delta \mathbf{u}_d$$

Avec $(\tilde{\mathbf{u}}_d(i, j, t) \cdot \mathbf{n}) = 0$ sur $\partial\Omega_d$.

Et comme condition initiale :

$$\tilde{\mathbf{u}}_d(i, j, t_0) = \mathbf{u}_d(i, j, t_0)$$

Une première méthode permettant de déterminer $\tilde{\mathbf{u}}_d$ en $t_0 + \Delta_t$ consisterait à écrire :

$$\forall (i, j) \in \Omega_d, \tilde{\mathbf{u}}_d(i, j, t_0 + \Delta_t) = \tilde{\mathbf{u}}_d(i, j, \Delta_t) + \Delta_t \nu \Delta \mathbf{u}_d(i, j)$$

Malheureusement, cette méthode n'est absolument pas stable au cours du temps. Nous allons utiliser la méthode de Stam qui consiste à résoudre le système suivant :

$$\forall (i, j) \in \Omega_d,$$

$$\mathbf{u}_d(i, j) = \tilde{\mathbf{u}}_d(i, j) - \frac{\Delta_t \cdot \nu}{\delta^2} (\tilde{\mathbf{u}}_d(i-1, j) + \tilde{\mathbf{u}}_d(i+1, j) + \tilde{\mathbf{u}}_d(i, j-1) + \tilde{\mathbf{u}}_d(i, j+1) - 4\tilde{\mathbf{u}}_d(i, j))$$

Ce système est un système d'équation linéaire du type $AX = b$. Ce système pourrait ainsi se résoudre par une méthode classique d'inversion de matrice, mais le calcul serait peu performant. En effet, la matrice A admet énormément de 0 (il y a 4 éléments non nuls par ligne). Il existe heureusement des méthodes itératives très rapides pour permettre la résolution de ce type de système, notamment la méthode de Gauss-Seidel¹ que nous allons implémenter.

¹Pour plus d'informations, section 9.1 page 41

5.2.2 Algorithme

Nous implémentons dans cet algorithme la méthode de relaxation de Gauss-Seidel.

```
Procédure calculViscosite(ChampVecteurs u, ChampVecteurs u2, Reel nu, Reel dt)
Entier i,j,k
```

```
Reel tempo = dt * nu * (u.tailleX()-2) * (u.tailleY()-2)
```

```
Pour k=1 à 20 /*itération pour la résolution*/
```

```
|Pour j=1 à tailleY(u)-1
```

```
| Pour i=1 à tailleX(u)-1
```

```
| u2(i,j) = (u(i,j) + tempo *
```

```
| (u2(i-1,j) + u2(i+1,j) + u2(i,j-1)+ u2(i,j+1) ))/(1+4 * tempo)
```

```
|forcerBords(u2)
```

Le champ de vecteur u2 est censé être déjà initialisé à la bonne taille. On réalise donc un effet de bords pour permettre de bonnes optimisations.

5.3 Évolution : Advection

Dans cette partie, nous allons tenter de résoudre localement l'équation :

$$\frac{\partial(\tilde{\mathbf{u}}_d)}{\partial t} = -(\mathbf{u}_d \cdot \nabla) \mathbf{u}_d$$

Avec comme conditions aux bords :

$$(\tilde{\mathbf{u}}_d \cdot \mathbf{n}) = 0$$

Et comme conditions initiales :

$$\tilde{\mathbf{u}}_d(x, y, t_0) = \mathbf{u}_d(x, y, t_0)$$

5.3.1 Théorie

De la même manière que précédemment, une première approche consisterait à écrire :

$$\tilde{\mathbf{u}}_d(i, j, t + \Delta_t) = \tilde{\mathbf{u}}_d(i, j, t) - \Delta_t \cdot (\mathbf{u}_d \cdot \nabla) \mathbf{u}_d$$

Malheureusement encore, cette approche n'admet pas de solution stable dans le temps. Nous utiliserons la méthode utilisant des propriétés physiques décrites dans l'article de [2]. Elle permet d'écrire la relation (dans l'espace continu) :

$$\begin{aligned} \forall (x, y) \in \Omega, \tilde{\mathbf{u}}(x, y, t + \Delta_t) &= \tilde{\mathbf{u}}(x - u_x(x, t)\Delta_t, y - u_y(y, t)\Delta_t, t) \\ &= \mathbf{u}(x - u_x(x, t)\Delta_t, y - u_y(y, t)\Delta_t, t) \end{aligned}$$

Que l'on peut éventuellement écrire en posant $\mathbf{v} = (x, y)$:

$$\tilde{\mathbf{u}}(\mathbf{v}, t + \Delta_t) = \mathbf{u}(\mathbf{v} - \mathbf{u}(\mathbf{v}, t)\Delta_t, t)$$

Cette relation est vraie en espace continu. Elle s'écrit dans un espace discret de cette manière :

$$\forall (i, j) \in \Omega_d, \tilde{\mathbf{u}}_d(i, j, t + \Delta_t) = \mathbf{u}_d(i - u_{xd}(i, t)\frac{\Delta_t}{\delta}, j - u_{yd}(j, t)\frac{\Delta_t}{\delta}, t) \quad (5.1)$$

Il se pose à présent deux problèmes :

- Les termes $i - u_{xd}(i, t)\Delta_t$ et $j - u_{yd}(j, t)\Delta_t$ peuvent ne pas être entiers.
- Ces mêmes termes peuvent sortir de Ω_d .

Problème du non entier

On peut en effet constater sur la figure 5.1 que $\mathbf{v} - \mathbf{u}(\mathbf{v})\Delta_t$ ne tombe pas sur un entier pile.

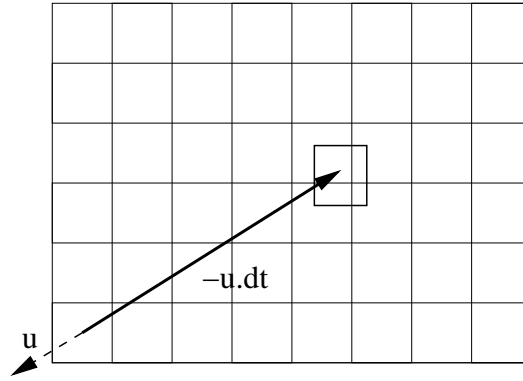


FIG. 5.1 – Calcul de l'advection

Pour parer le problème, il faut prendre en considération la contribution des 4 parties. Pour réaliser cela, nous allons calculer leurs aires.

Notons (i_d, j_d) les coordonnées de la case inférieure gauche. L'intersection des 4 cases a donc pour coordonnées $(i_d + 0.5, j_d + 0.5)$.

Notons $x_d = i - u_x(i, t)\frac{\Delta_t}{\delta}$ et $y_d = j - u_y(j, t)\frac{\Delta_t}{\delta}$.

L'aire de la contribution inférieure gauche est : $A_{SW} = (1 + i_d - x_d) \times (1 + j_d - y_d)$.

L'aire de la contribution inférieure droite est : $A_{SE} = (x_d - i_d) \times (1 + j_d - y_d)$.

L'aire de la contribution supérieure gauche est : $A_{NW} = (1 + i_d - x_d) \times (y_d - j_d)$.

L'aire de la contribution supérieure droite est : $A_{NE} = (x_d - i_d) \times (y_d - j_d)$.

On en déduit une bonne approximation :

$$\begin{aligned} \tilde{\mathbf{u}}_d(i, j, t + \Delta_t) = & A_{SW}\mathbf{u}_d(i_d, j_d, t) + \\ & A_{SE}\mathbf{u}_d(i_d + 1, j_d, t) + \\ & A_{NW}\mathbf{u}_d(i_d, j_d + 1, t) + \\ & A_{NE}\mathbf{u}_d(i_d + 1, j_d + 1, t) \end{aligned}$$

Problème du dépassement

Nous avons vu précédemment que pour une meilleure efficacité, il est plus performant de prendre en contribution le plus de parties possible. C'est ce que nous allons faire dans le cas d'un dépassement de bords.

Par exemple si $x_d < 1$, on pourrait le rebasculer à 1. Mais dans ce cas, il n'y aurait qu'une partie qui contribuerait. Nous préférons ainsi prendre le cas $x_d < 0.5$ et le rebasculer à 0.5, permettant ainsi la contribution de deux parties.

À noter que le vecteur vitesse près d'un bord sera en général assez parallèle à celui-ci (à cause des conditions sur les bords). C'est pourquoi si il y a dépassement de bords, ce dépassement sera très léger en théorie. Cela explique le fait que l'on rebascule simplement le vecteur en 0.5.

5.3.2 Algorithme

Nous pouvons à présent écrire l'algorithme de résolution de l'advection.

Procédure calculAdvection(ChampsVecteurs u, ChampsVecteurs u1, Reel dt)

```

Reel xd, yd
Entier n = tailleX(u)-2
Entier m = tailleY(u)-2
Entier dtspas = dt / pas(u)

Pour j=1 à m
  Pour i=1 à n
    |xd = i - dtspas * ux(i,j)
    |yd = j - dtspas * uv(i,j)
    /*rebasculer dans les bords*/
    |Si (xd < 0.5) alors xd = 0.5
    |Si (xd > n+0.5) alors xd = 0.5 + n
    |Si (yd < 0.5) alors yd = 0.5
    |Si (yd > m+0.5) alors yd = 0.5 + m
    /*calcul de id et jd*/
    |id = (Entier) xd
    |jd = (Entier) yd
    /*calcul des aires*/
    |Asw = (1+id-xd) * (1+jd-yd)
    |Ase = (xd - id) * (1+jd-yd)
    |Anw = (1+id-xd) * (yd - jd)
    |Ane = (xd - id) * (yd - jd)
    /*calcul de la valeurs*/
    |u1(i,j) =
    | Asw * u(id, jd) +
    | Ase * u(id+1,jd) +
    | Anw * u(id, jd+1) +
    | Ane * u(id+1, jd+1)
  forcerBords(u1)
Fin

```

5.4 Évolution forces

Nous allons déterminer la résolution locale de l'équation :

$$\frac{\partial \tilde{\mathbf{u}}_d}{\partial t} = \mathbf{f}$$

Avec comme conditions initiales :

$$\frac{\partial \tilde{\mathbf{u}}_d}{\partial t}(x, y, t_0) = \mathbf{f}(x, y, t_0)$$

5.4.1 Théorie

Comme nous l'avons vu, en général, cette force est nulle ou constante.
Le calcul de $\tilde{\mathbf{u}}_d$ peut se faire de manière exacte par intégration.

5.4.2 Algorithme

```
fonction calculForces(ChampsVecteurs u, Vecteur force, Reel dt)
  Pour j=1 à tailleY(u)-2
    Pour i=1 à tailleX(u)-2
      u(x,y) = u(x,y) + force * dt
    forcerBords(u)
```

5.5 Calcul de ϕ et de \mathbf{u}_d

En appliquant successivement les algorithmes précédents, nous avons un algorithme permettant de déterminer localement la solution de l'équation :

$$\frac{\partial(\tilde{\mathbf{u}}_d)}{\partial t} = -(\mathbf{u}_d \cdot \nabla) \mathbf{u}_d + \nu \Delta \mathbf{u}_d + \mathbf{f}$$

Mais il faut pouvoir résoudre l'équation :

$$\Delta \phi(x, y) = \text{div } \tilde{\mathbf{u}}_d(x, y, t_0 + \Delta t)$$

pour déterminer \mathbf{u}_d au temps $t_0 + \Delta t$.

5.5.1 Théorie

Nous noterons

$$g(x, y) = \text{div } \tilde{\mathbf{u}}_d(x, y, t_0 + \Delta t)$$

À une position (i, j) , nous disposons dans l'espace discret de l'équation suivante :

$$\frac{\phi(i+1, j) + \phi(i-1, j) + \phi(i, j+1) + \phi(i, j-1) - 4\phi(i, j)}{\delta^2} = g(x, y)$$

Avec : ϕ continue et $(\phi(i, j) \cdot \mathbf{n}) = 0$ sur $\partial\Omega_d$.

Afin de limiter les erreurs d'arrondis, nous préférons résoudre le système :

$$4\phi(i, j) - \phi(i+1, j) - \phi(i-1, j) - \phi(i, j+1) - \phi(i, j-1) = -g(x, y) \cdot \delta^2$$

Ceci est un système d'équations linéaires du type $AX = b$ dont beaucoup de coefficients de A sont nuls.

Nous allons encore une fois utiliser la méthode de relaxation de Gauss pour déterminer ϕ .

Ensuite, il suffira de déterminer $\vec{\nabla}(\phi)$ et d'appliquer la formule :

$$\mathbf{u}_d(x, y, t_0 + \Delta t) = \tilde{\mathbf{u}}_d(x, y, t_0 + \Delta t) - \vec{\nabla}(\phi)(x, y)$$

5.5.2 Algorithme

On écrit tout d'abord un algorithme permettant de calculer la divergence multiplié par le pas au carré d'un champ de vecteurs.

```

Procédure div_pascarreChamps(ChampsVecteurs u, ChampsScalaires divpout)
Entier n = tailleX(u)-2
Entier m = tailleY(u)-2
Reel pas = pas(u)

/*calcul hors frontiere*/
Pour j=1 à m
  Pour i=1 à n
    divpout(i,j) = 0.5 * (ux(i+1,j) - ux(i-1,j)+uy(i,j+1) - uy(i, j-1)) * pas

/*calcul sur la frontiere*/
Pour i=1 à n
  divpout(i,0) = div(i,1)
  divpout(i, m+1) = div(i, m)
Pour j=1 à m
  divpout(0,j) = divpout(1,j)
  divpout(n+1,j) = divpout(n,j)

divpout(0,0) = 0.5 * (divpout(1,0) + divpout(0,1))
divpout(n+1,0) = 0.5 * (divpout(n+1,1) + divpout(n,0))
divpout(0,m+1) = 0.5 * (divpout(1,m+1) + divpout(0,m))
divpout(n+1,m+1) = 0.5 * (divpout(n,m+1) + divpout(n+1,m))

```

Maintenant, nous pouvons écrire l'algorithme déterminant u .

```

Procédure calculProjection(ChampsVecteurs u)
ChampsScalaire divpas (mémoire initialisée une seule fois pour gain de temps
                        et de même taille que u)
ChampsScalaire phi (même taille que u)
Entier n = tailleX(u)-2
Entier m = tailleY(u)-2
phi = 0
div_pascarreChamps(u, divpas)

/*resolution*/
Reel tempo = dt * nu * n * m

Pour k=1 à 20 /*itération pour la résolution*/
  |Pour j=1 à m
  |  Pour i=1 à n
  |    phi(i,j) = (-div(i,j) +
  |              (phi(i-1,j) + phi(i+1,j)
  |              + phi(i,j-1)+ phi(i,j+1))) / 4)
  |forcerBordsScalaires(phi)

```

```

/*on soustrait directement la divergence de phi à u*/
Pour j=1 à m
  Pour i=1 à n
    ux(i,j) = ux(i,j) - 0.5*(phi(i+1,j) - phi(i-1,j)) / pas
    uy(i,j) = uy(i,j) - 0.5*(phi(i,j+1) - phi(i,j-1)) / pas
  forcerBords(u)

```

5.6 Détermination de u

À présent, nous disposons de toutes les fonctions nécessaires à la détermination de \mathbf{u}_d à l'instant $t_0 + \Delta_t$.

```

fonction evoluerVitesse(ChampsVecteurs u, Reel nu, Reel dt)
  ChampsVecteurs u, u1, u2 : de même taille que u

  calculViscosite(u, u1, nu, dt)
  calculAdvection(u1, u2, dt)
  calculForces(u2, u, force, dt)
  calculProjection(u)

```

L'algorithme peut être un peu modifié suivant le fait que l'on prenne en compte les forces ou non.

5.6.1 Attention :

Ici, on suppose que la divergence de \mathbf{u} est nulle à chaque étape. Ce n'est pas forcément le cas si l'utilisateur ajoute une force par lui-même à n'importe quel moment. Dans un tel cas, il faudra éventuellement calculer la projection de \mathbf{u} avant le reste. L'algorithme deviendrait alors (mais il y a une grande perte de temps) :

```

fonction evoluerVitesse(ChampsVecteurs u, Reel nu, Reel dt)
  ChampsVecteurs u, u1, u2 : de même taille que u

  calculProjection(u)
  calculViscosite(u, u1, nu, dt)
  calculAdvection(u1, u2, dt)
  calculForces(u2, u, force, dt)
  calculProjection(u)

```

Une astuce pour éviter cela serait de placer sur une pile la liste des modifications à réaliser sur le champ de vecteur vitesse et de les faire juste avant la projection. Ainsi, la projection ne serait calculée qu'une seule fois par tour. C'est cette approche que j'ai utilisée dans le code source.

5.7 Exemple de résultats

Dans mon algorithme, j'ai permis à l'utilisateur de modifier par un simple clique le vecteur vitesse en un point (ce qui revient à peu près à ajouter une force pendant un court instant). Cela permet de rendre plus attractive l'application.

Exemple1 Voici le type de résultat que l'on peut obtenir figure 5.2. Nous avons ici utiliser des obstacles rectangulaires, nous verrons par la suite comment gérer ce cas.

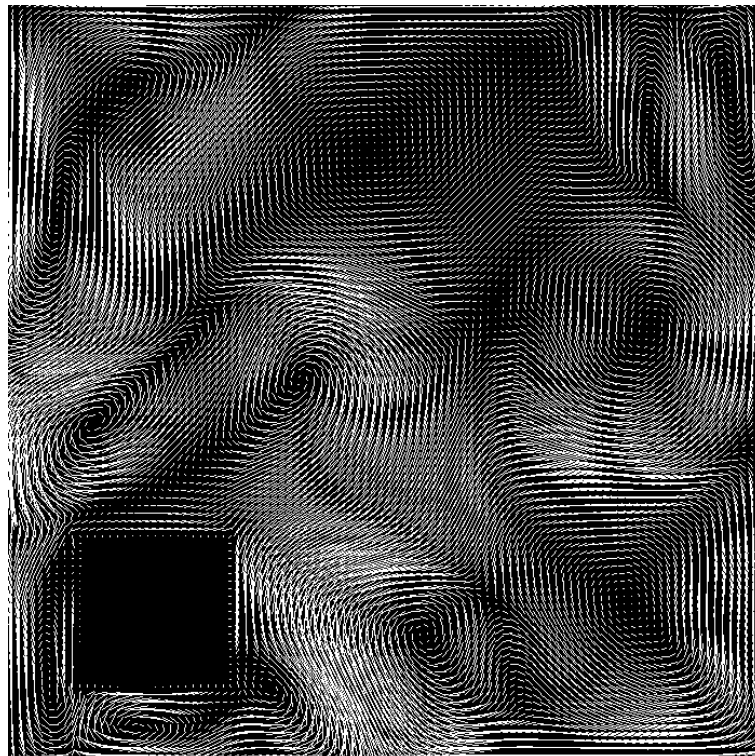


FIG. 5.2 – Visualisation du champ de vecteurs vitesse de taille 100×100 avec obstacle

Exemple 2 Figure 5.3.

Exemple 3 Un exemple un peu plus coloré, figure 5.4.

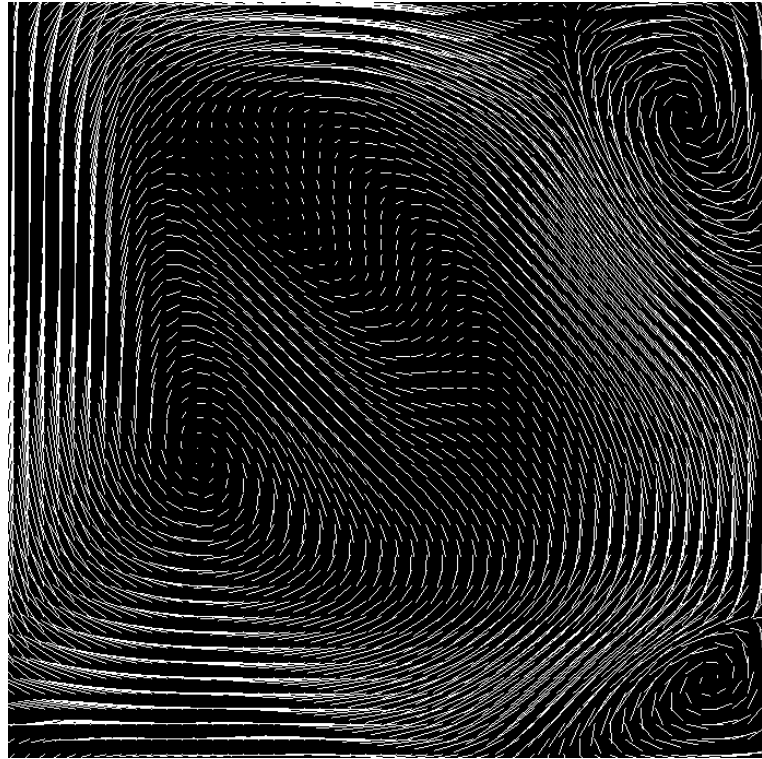


FIG. 5.3 – Visualisation du champ de vecteurs vitesse de taille 50×50 sans obstacle

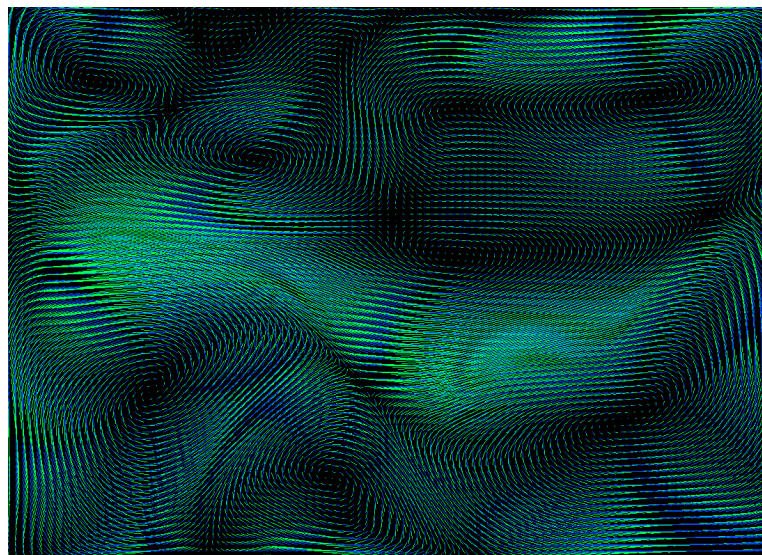


FIG. 5.4 – Visualisation du champ de vecteurs vitesse tronquée

Chapitre 6

Problème de diffusion

À présent, nous disposons d'un algorithme nous permettant de faire évoluer un champ de vecteurs vitesse dans une partie rectangulaire.

Il faut avouer que graphiquement, la visualisation unique du champ de vecteurs n'est pas très intéressante. On souhaiterait pouvoir utiliser un phénomène de diffusion comme sur la figure 6.1.

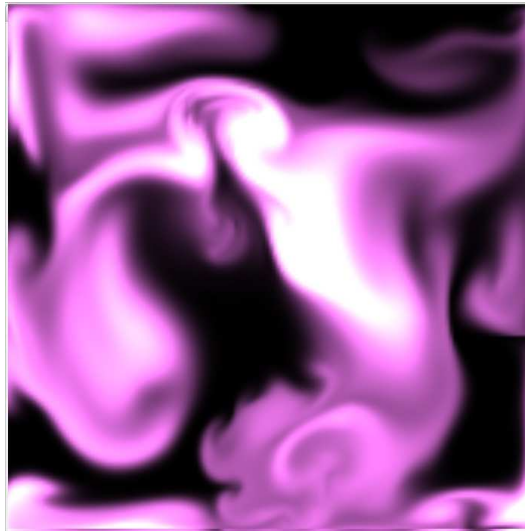


FIG. 6.1 – Visualisation du champ de diffusion, taille 150×150

Pour réaliser ce type de graphisme, nous allons utiliser les équations de densité appliquées à un fluide.

6.1 Théorie

6.1.1 Notation

Dans cette partie, nous utiliserons les mêmes notations que dans les chapitres précédents.

– $d : (x, y, t) \mapsto \mathbb{R}$ désignera une application de classe C^∞ .

- $S : (x, y, t) \mapsto \mathbb{R}$ une application de classe C^∞
- D un réel positif

6.1.2 Équations

Les équations de densité s'écrivent ainsi :

$$\forall (x, y) \in \Omega, \frac{\partial d}{\partial t}(x, y, t) = -(\mathbf{u}(x, y, t) \cdot \nabla) d(x, y, t) + D\Delta d(x, y, t) + S(x, y, t) \quad (6.1)$$

Soit en notation abrégée :

$$\frac{\partial d}{\partial t} = -(\mathbf{u} \cdot \nabla) d + D\Delta d + S \quad (6.2)$$

Et avec comme condition sur les bords, d est continue sur $\partial\Omega$.

6.1.3 Intèrprétation

Densité Notre problème ici peut être imaginé comme suit. On dispose d'une marre d'eau et on y introduit un autre liquide (de couleur violette par exemple). La densité du liquide violet pourra être déterminée par ces équations et correspondra à d .

Coefficient de densité Le réel D correspondant au coefficient de densité, on pourra le prendre nul pour alléger les calculs.

Source La fonction S correspond à une source d'introduction d'un autre liquide (le liquide violet). On pourra laisser l'utilisateur gérer lui même l'introduction ou bien la forcer.

6.1.4 Résolution

On peut constater que l'équation est assez similaire à l'équation de Navier et Stokes à l'exception que d est un champ de scalaires et non un champ de vecteurs et le champ \mathbf{u} est connu à tout instant.

On pourra décomposer le problème de la même manière.

Premier problème On connaît d à un instant t_0 . On résout localement l'équation

$$\frac{\partial d_1}{\partial t} = -(\mathbf{u} \cdot \nabla) d$$

Avec d_1 continue sur les bords et $d_1(x, y, t_0) = d(x, y, t_0)$.

Deuxième problème On résout l'équation

$$\frac{\partial d_2}{\partial t} = D\Delta d$$

Avec $d_2(x, y, t_0) = d(x, y, t_0)$ et d_2 continue sur les bords.
on sommera finalement les résultats (ou on combinera successivement les deux applications).

6.2 Réalisation et algorithmique

6.2.1 Les conditions aux bords

Nous allons dans un premier temps écrire l'algorithme permettant de forcer un champ de scalaires à être continu sur les bords.

```

fonction forcerBordsScalaire(ChampScalaires d)
  Entier n = tailleX(d)-2
  Entier m = tailleY(d)-2

  Pour j=1 à m
    d(0,j) = d(1,j)
    d(n+1,j) = d(n,j)

  Pour i=1 à n
    d(i, 0) = d(i,1)
    d(i, m+1) = d(i,m)

  d(0,0) = 0.5* (d(1,0) + d(0,1))
  d(n+1,0) = 0.5* (d(n,0) + d(n+1,1))
  d(n+1,m+1) = 0.5* (d(n,m+1) + d(n+1,m))
  d(0,m+1) = 0.5* (d(1,m+1) + d(0,m))

```

6.2.2 Calcul de l'advection

Le calcul de l'advection se fait à peu près de la même manière que dans l'ancien algorithme à l'exception que d est de dimension 1 et qu'il faut forcer les bords avec la fonction précédente.

```

fonction calculAdvectionS(ChampScalaires in, ChampScalaires out, ChampVecteurs u, Reel dt)
  Entier id, jd
  Réel Asw, Ase, Ane, Anw
  Entier n = tailleX(d)-2
  Entier m = tailleY(d)-2

  Réel dtspas = dt / pas;

  Pour j=1 à m
    Pour i=1 à n
      xd = i -dtspas * ux(i,j);
      yd = j - dtspas * uy(i,j);
      /*rebascule dans les bords*/
      Si(xd < 0.5) Alors xd = 0.5;
      Si(yd < 0.5) Alors yd = 0.5;
      Si(xd > n+0.5) Alors xd = n+0.5;
      Si(yd > m+0.5) Alors yd = m+0.5;
      id = (Entier) xd;
      jd = (Entier) yd;
      Asw = (1.0+id-xd) * (1.0+jd-yd);
      Ase = (xd - id) * (1.0+jd-yd);
      Anw = (1.0+id-xd) * (yd - jd);
      Ane = (xd - id) * (yd - jd);

```

```

        out(i,j) =
            Asw * in(id, jd) +
            Ase * in(id+1,jd) +
            Anw * in(id, jd+1) +
            Ane * in(id+1, jd+1));
        out(i,j) =
            Asw * in(id, jd) +
            Ase * in(id+1,jd) +
            Anw * in(id, jd+1) +
            Ane * in(id+1, jd+1));
    }
}

forcerBordsScalaires(out);
}

```

6.2.3 Calcul de la viscosité

Cet algorithme n'est pas obligatoire. Il ne change pas beaucoup le résultat.

```

fonction calculViscositeS(ChampScalaires in, ChampScalaires out, Reel D, Reel dt)
Entier k
Entier i, j
Entier n = tailleX(in)-2
Entier m = tailleY(in)-2

Reel mult = dt * nu * n *n;

Pour k=1 à 20
|Pour j=1 à m
| Pour i=1 à n
|     out(i, j) =
|         (in(i,j) + mult*
|           ( out(i+1,j) + out(i-1,j) + out(i,j+1) + out(i,j-1)))/(1+4*mult));
|
|
|forcerBordsScalaires(out);

```

6.2.4 Calcul général

À présent, nous pouvons écrire facilement l'évolution de d .

```

fonction calculDiffusion(ChampScalaires diffusion, ChampScalaires u, Reel D, Reel dt)
calculViscositeS(diffusion, tempo, D, dt)

diffusion = tempo

calculAdvectionS(tempo, diffusion, u, dt)

```

Le symbole = dans l'algorithme correspond à un recopiage de `tempo` dans `diffusion`.

Remarque 1 Il faut noter qu'à chaque étape, il vous faudra faire évoluer le champ de vitesse \mathbf{u} par l'intermédiaire de la fonction `evoluerVitesse`.

Remarque 2 La source S n'apparaît nulle part dans les calculs. En fait, on laissera l'utilisateur (ou le programmeur) ajouter la force comme il le souhaite.

Si il veut ajouter un « liquide violet » pendant un court laps de temps à une position, il suffira de forcer la variable `diffusion` durant un court laps de temps à une valeur assez grande (par exemple 100) :

```
diffusion(i,j) = 100
```

6.3 Exemple d'application

Voici quelques exemples.

Exemple 1 Figure 6.2

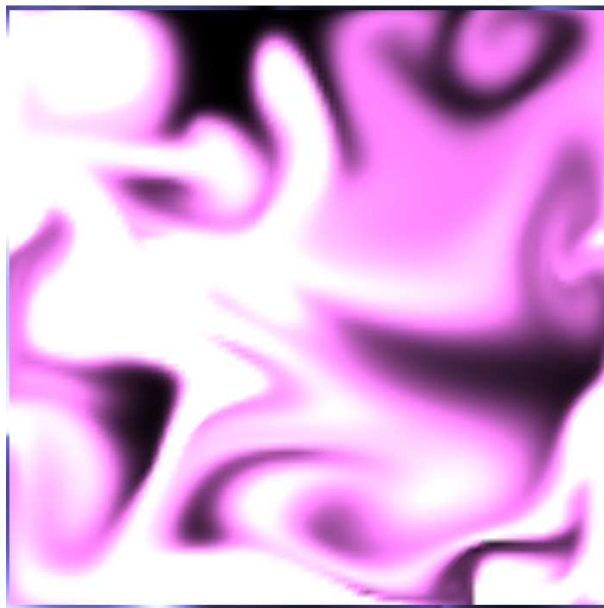


FIG. 6.2 – Visualisation du champ de scalaires diffusion

Exemple 2 Figure 6.2

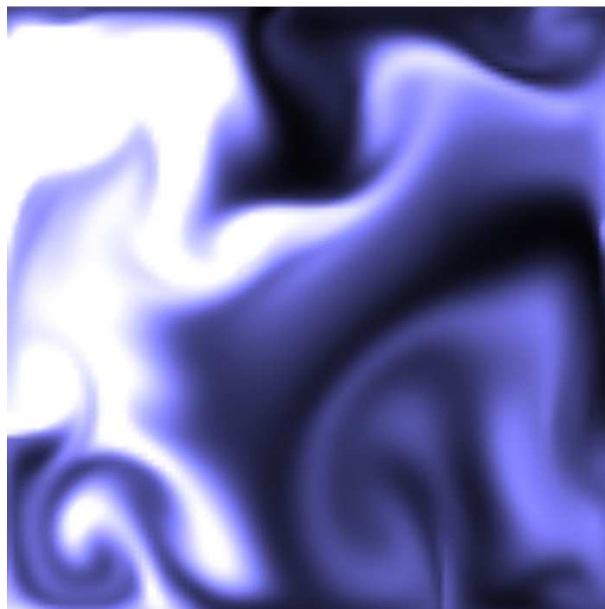


FIG. 6.3 – Visualisation du champ de scalaires diffusion

Chapitre 7

Ajout d'obstacles

Un problème intéressant que l'on peut se poser à présent est la question de la gestion d'obstacle. Nous allons pour cela utiliser les conditions sur les bords que l'on a pu voir dans la partie théorique.

Ainsi, la gestion d'obstacle sera assez similaire à la gestion des bords à quelques différences près.

Je présenterai ici une méthode assez simple de gestion d'obstacle, elle est malheureusement loin d'être parfaite d'un point de vue théorique, mais fonctionne bien d'un point de vue pratique. La partie théorie reprend les hypothèses énoncées dans la section 5.1.

7.1 Réalisation et algorithme

Nous supposons qu'un obstacle est rectangulaire (et compris dans le domaine du fluide), le sud ouest positionné aux coordonnées (x, y) , de largeur w et de hauteur h .

Nous allons écrire deux fonctions, l'une gérant les conditions aux bords pour un champ de vecteurs, l'autre gérant les conditions aux bords pour un champ de scalaires.

7.1.1 Bords pour un champs de scalaires

L'algorithme pour forcer les bords d'un champ de scalaire s'écrit comme suit :

```
Procédure forcerBordsScalairesObstacles(ChampScalaires in, Entier squarex,  
                                       Entier squarey, Entier squarew, Entier squareh)
```

```
Entier i;  
Entier j;  
Entier n,m;
```

```
n = tailleX(in)-2;  
m = tailleY(in)-2;
```

```
Pour j=squarey à squarey+squareh  
  in(squarex-1, j) = in(squarex-2,j);  
  in(squarex+squarew+1,j) = in(squarex+squarew+2, j);  
  Pour i=squarex à squarex+squarew  
    in(i,j) = 0.0f; /*on annule le centre*/
```

```
Pour i=squarex à squarex+squarew
```

```

in(i, squarey-1) = in(i,squarey-2);
in(i,squarey+squareh+1) = in(i, squarey+squareh+2));

in(squarex-1,squarey-1) = 0.5 * (in(squarex, squarey-1) + in(squarex-1,squarey));
in(squarex+squarew+1,squarey-1) = 0.5 * (in(squarex+squarew+1,squarey) +
in(squarex+squarew,squarey-1));
in(squarex-1,squarey+squareh+1) = 0.5 * (in(squarex,squarey+squareh+1) +
in(squarex-1,squarey+squareh));
in(squarex+squarew+1,squarey+squareh+1) = 0.5 * (in(squarex+squarew,squarey+squareh+1) +

```

7.1.2 Champ de vecteurs

Nous allons décomposer l'algorithme en plusieurs parties. Nous traiterons dans un premier temps le calcul des champ de scalaires u_x et u_y séparément.

```

Procédure forcerBordsXObstacle(ChampScalaires ux, Entier squarex,
Entier squarey, Entier squarew, Entier squareh)

```

```

Entier i;
Entier j;
Entier n,m;

```

```

n = tailleX(ux)-2;
m = tailleY(ux)-2;

```

```

Pour j=squarey à (squarey + squareh)
ux(squarex-1, j) = 0
ux(squarex + squarew+1, j) = 0
Pour i=squarex à (squarex+squarew)
ux(i, j) = 0 /*on annule l'intérieur*/

```

```

Pour i=squarex à squarex+squarew
ux(i, squarey-1) = ux(i, squarey-2);
ux(i, squarey+squareh+1) = ux(i, squarey+squareh+2)

```

```

ux(squarex-1,squarey-1) =
0.5 * (ux(squarex, squarey-1) + ux(squarex-1,squarey));
ux(squarex+squarew+1,squarey-1) =
0.5 * (ux(squarex+squarew+1,squarey) + ux(squarex+squarew,squarey-1))
ux(squarex-1,squarey+squareh+1) =
0.5 * (ux(squarex,squarey+squareh+1) + ux(squarex-1,squarey+squareh))
ux(squarex+squarew+1,squarey+squareh+1) =
0.5 * (ux(squarex+squarew,squarey+squareh+1) +
ux(squarex+squarew+1,squarey+squareh))

```

À présent, nous allons traiter le champ u_y .

```

Procédure forcerBordsXObstacle(ChampScalaires uy, Entier squarex,
Entier squarey, Entier squarew, Entier squareh)

```

```

Entier i;
Entier j;
Entier n,m;

```

```

n = tailleX(uy)-2;
m = tailleY(uy)-2;

Pour i=squarex à squarex+squarew
  uy(i, squarey-1) = 0
  uy(i, squarey+squarew+1) = 0
  Pour j=squarey à (squarey+squareh)
    uy(i, j) = 0

Pour j=squarey à squarey+squareh
  uy(squarex-1,j) = uy(squarex-2,j)
  uy(squarex+squarew+1,j) = uy(squarex+squarew+2, j)

uy(squarex-1,squarey-1) =
  0.5 * (uy(squarex, squarey-1) + uy(squarex-1,squarey));
uy(squarex+squarew+1,squarey-1) =
  0.5 * (uy(squarex+squarew+1,squarey) + uy(squarex+squarew,squarey-1))
uy(squarex-1,squarey+squareh+1) =
  0.5 * (uy(squarex,squarey+squareh+1) + uy(squarex-1,squarey+squareh))
uy(squarex+squarew+1,squarey+squareh+1) =
  0.5 * (uy(squarex+squarew,squarey+squareh+1) +
  uy(squarex+squarew+1,squarey+squareh))

```

Nous pouvons écrire l'algorithme général de la manière suivante :

```

Procédure forcerBordsVecteursObstacle(ChampVecteurs u = (ux, uy), Entier squarex,
  Entier squarey, Entier squarew, Entier squareh)
  forcerBordsXObstacle(ux, squarex, squarey, squarew, squareh)
  forcerBordsYObstacle(uy, squarex, squarey, squarew, squareh)

```

7.1.3 Cas général et critiques

Pour que cela fonctionne correctement, il faudra ajouter ces procédures à chaque appel de `forcerBords` et `forcerBordsScalaire`s. Si il y a plusieurs obstacles, il faut alors appeler ces procédures pour tous les obstacles (éventuellement par l'intermédiaire d'une liste).

Pour que ceci soit théoriquement correct, il faudrait également modifier les fonctions de calcul d'advection. En effet, avec les appels du type `xd = i - dtspas * ux(i,j)`, il peut arriver que l'on tombe sur un obstacle. Mais gérer ce cas complique les algorithmes. En ne le prenant pas en compte, on peut constater en pratique que cela donne des résultats cohérents et assez corrects.

7.2 Exemples et applications

Voici quelques exemples de ce que l'on peut obtenir.

Exemple 1 : Exemple avec deux obstacles (figure 7.1).

Exemple 2 : Figure 7.2

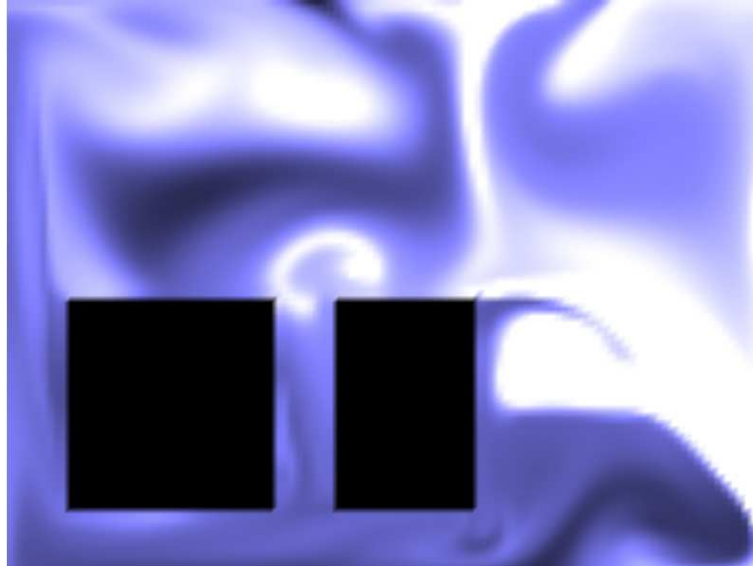


FIG. 7.1 – Visualisation du champ diffusion avec obstacles

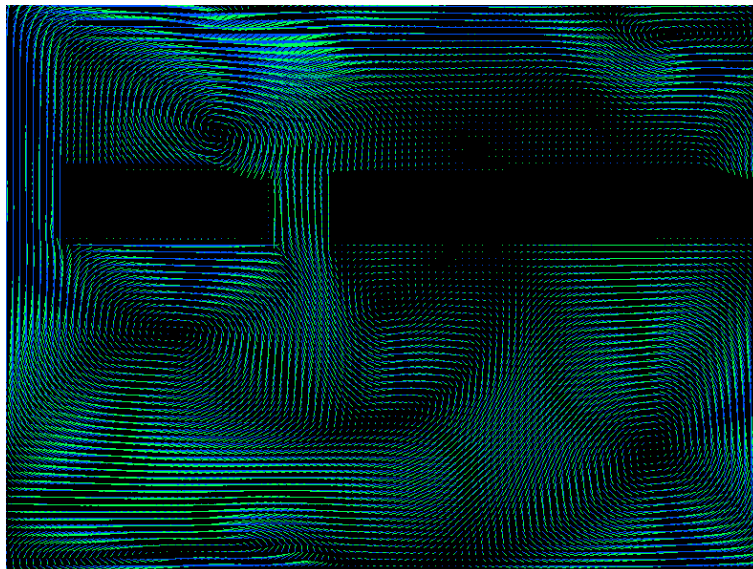


FIG. 7.2 – Visualisation du champ vitesse avec obstacles

Chapitre 8

Suppléments

Dans ce chapitre, nous allons voir quelques algorithmes supplémentaires pouvant être utiles, notamment sur le calcul de la pression et sur la détection des tourbillons.

8.1 Calcul de la pression

Les équations originales utilisaient la notion de pression, mais nous avons réussi à trouver une astuce pour supprimer le calcul de celle-ci. Mais il peut tout de même être intéressant de déterminer cette grandeur.

8.1.1 Théorie, rappel

Nous rappelons qu'à un instant t_0 , la pression peut être obtenue par la résolution du système¹ :

$$\Delta p = \rho \operatorname{div}((\mathbf{u} \cdot \nabla) \mathbf{u})$$

Avec comme conditions au bords, p continue.

Nous avons déjà déterminé des algorithmes permettant la résolution de ce type d'équation.

Il suffira de calculer $(\mathbf{u} \cdot \nabla) \mathbf{u}$ ce qui peut être fait de manière approximative (exacte pour un petit pas spatial) car nous avons à chaque étape de nouvelles valeurs de \mathbf{u} *qui ne dépendent pas de p* .

8.1.2 Algorithme

Nous allons réaliser un premier algorithme qui détermine $(\mathbf{u} \cdot \nabla) \mathbf{u}$.

in correspondra à \mathbf{u} et out vaudra $(\mathbf{u} \cdot \nabla) \mathbf{u}$.

```
fonction calculUNablaU(ChampsVecteurs in = (ux, uy), ChampsVecteurs out = (outx, outy))
  Entier n = tailleX(in)-2;
  Entier m = tailleY(in)-2;
  Entier i, j;

  Reel h = pas(in)

  Pour j=1 à m
```

¹Nous avons pris f nulle

```

Pour i=1 à n
  outx(i,j)=
    ux(i,j) * (ux(i+1,j) - ux(i-1,j)) / (2*h)
    +uy(i,j) * (ux(i,j+1) - ux(i,j-1)) / (2*h));
  outy(i,j)=
    ux(i,j) * (uy(i+1,j) - uy(i-1,j)) / (2*h)
    +uy(i,j) * (uy(i,j+1) - uy(i,j-1)) / (2*h));
}

forcerBords(out);

```

Nous allons utiliser la fonction `div_pascalreChamps` définie lors du calcul de ϕ .

Procédure `calculPression(ChampsVecteurs vitesse, ChampsScalaires pression)`

```

Entier n = tailleX(in)-2;
Entier m = tailleY(in)-2;
Entier i, j;

ChampsScalaires vect_tempo1 /*de même taille que vitesse*/

calculUNablaU(vitesse, vect_tempo1, &lobs);
div_pascalreChamps(vect_tempo1, divpp, &lobs);

/*on peut multiplier divpp par rho pour avoir exactement la formule
mais en pratique ce n'est pas utile*/

pression = 0 /*on annule le champs*/

Pour k=1 à 20 /*itération pour la résolution*/
|Pour j=1 à m
| Pour i=1 à n
|   pression(i,j) = (-divpp(i,j) +
|     (pression(i-1,j) + pression(i+1,j)
|     + pression(i,j-1)+ pression(i,j+1))) / 4)
|forcerBordsScalaires(pression)

```

À noter que l'on aurait pu écrire une fonction commune avec le calcul de ϕ .

8.1.3 Exemple d'application

On pourra remarquer que la pression s'annule relativement vite, dans ces exemples, on a augmenté ces valeurs pour que l'image soit plus visible.

Exemple 1 Sur la figure 8.1, le rouge correspond à une pression positive tandis que le bleu correspond à une pression négative.

8.2 Détection de tourbillons

Nous allons voir dans cette partie comment détecter les tourbillons, c'est à dire comment détecter un ensemble de point qui semble tourner autour d'un point central.

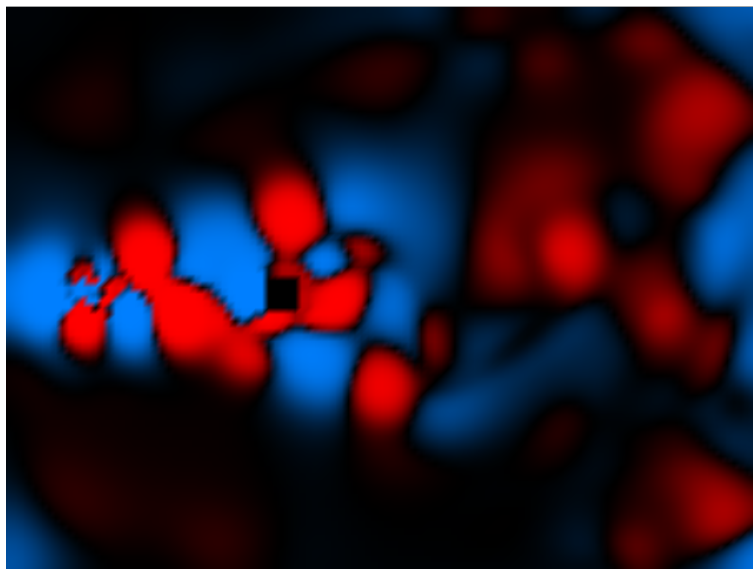


FIG. 8.1 – Visualisation de la pression avec obstacle

8.2.1 Théorie

La définition peu formelle que l'on vient de donner correspond en fait à la définition physique de la notion de rotationnel.

Nous rappelons que le rotationnel est défini pour une application \mathbf{u} à valeurs dans \mathbb{R}^3 de classe C^1 par :

$$\operatorname{rot}\mathbf{u}(x, y, z) = \begin{pmatrix} \frac{\partial u_z}{\partial y} - \frac{\partial u_y}{\partial z} \\ \frac{\partial u_x}{\partial z} - \frac{\partial u_z}{\partial x} \\ \frac{\partial u_y}{\partial x} - \frac{\partial u_x}{\partial y} \end{pmatrix}$$

Le champs de vitesse n'est défini que sur \mathbb{R}^2 . Nous allons lui ajouter une composante sur l'axe z nulle. La formule devient donc :

$$\operatorname{rot}\mathbf{u}(x, y, z) = \begin{pmatrix} 0 \\ 0 \\ \frac{\partial u_y}{\partial x} - \frac{\partial u_x}{\partial y} \end{pmatrix}$$

Comme les deux premiers champs sont nuls, on peut simplement écrire :

$$\operatorname{rot}\mathbf{u}(x, y, z) = \frac{\partial u_y}{\partial x} - \frac{\partial u_x}{\partial y}$$

C'est cette valeur qui nous servira à la détection des tourbillons. Il faudra uniquement lui ajouter une condition de continuité sur les bords.

8.2.2 Algorithme

Nous appliquons simplement la formule précédente pour l'algorithme.

```

Procédure calculVortex(ChampVecteurs u = (ux, uy), ChampScalaires rot)
  Entier n = tailleX(u)-2;
  Entier m = tailleY(u)-2;
  Entier i, j;
  Reel h = pas(u)

  Pour j=1 à m
    Pour i=1 à n
      rot(i,j) =
        (uy(i+1,j) - uy(i-1,j)) / (2*h)
        - (ux(i,j+1) - ux(i,j-1)) / (2*h));

/*pour les conditions sur les bords*/
forcerBordsScalaires(rot)

```

8.2.3 Exemple d'application

Exemple 1 Sur la figure 8.2, la couleur rouge correspond aux valeurs positives et la couleur verte aux valeurs négatives. En fait, la couleur correspond au sens de rotation des tourbillons.

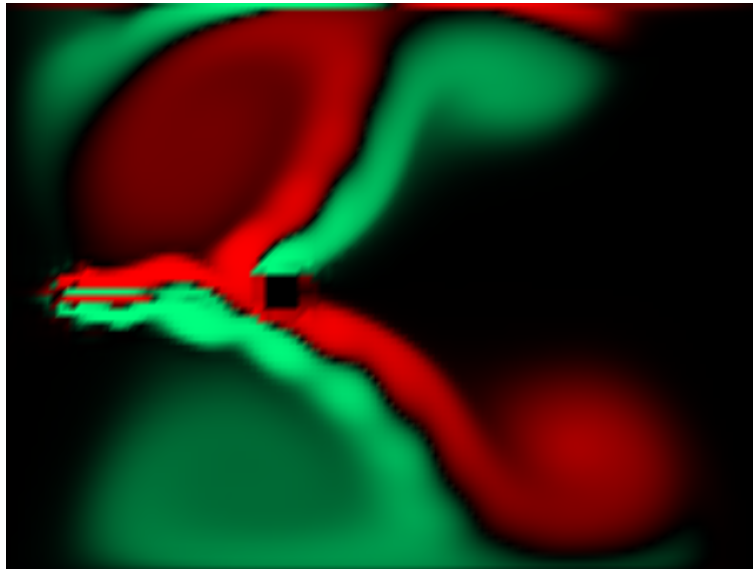


FIG. 8.2 – Détection de tourbillons

Exemple 2 Voici un exemple illustrant le phénomène du tourbillon de Karman. Figure 8.3

Exemple 3 Figure 8.4, état assez chaotique.

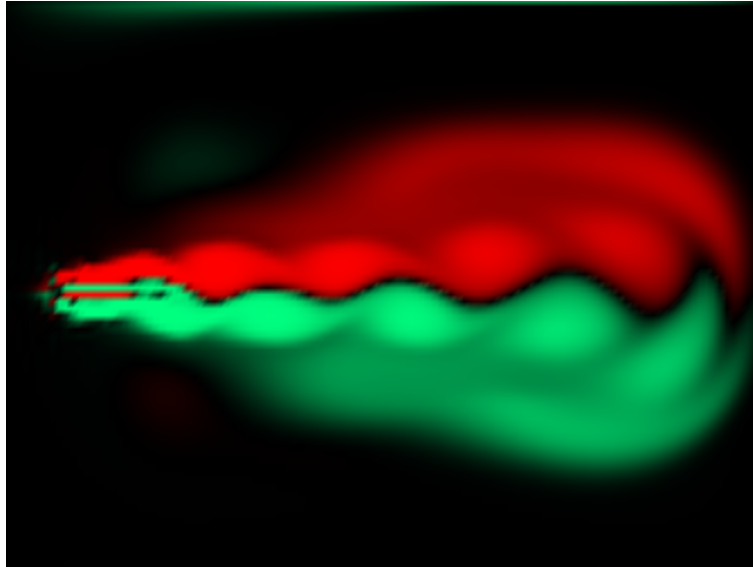


FIG. 8.3 – Phénomène de Karman

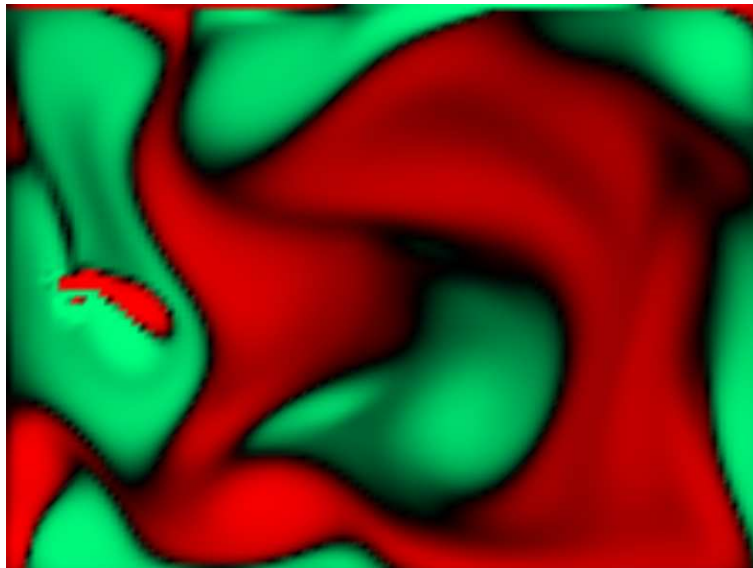


FIG. 8.4 – Détection de tourbillons

Chapitre 9

Annexe

9.1 Méthode de résolution de Gauss-Seidel

La méthode de résolution de Gauss-Seidel permet de résoudre un système du type $Ax = b$ où A est une matrice carrée de taille $n \times n$ et x et b , des matrices colonnes de taille n .

On définit la suite des $x^{(k)}$ par :

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j < i} a_{ij} x_j^{(k)} - \sum_{j > i} a_{ij} x_j^{(k-1)} \right)$$

Cette suite converge rapidement vers la solution x . Cette méthode a plusieurs avantages. Si le nombre de coefficients nuls de la matrice A est très important, les sommes peuvent être courtes. De plus, l'implémentation de cette méthode ne nécessite qu'une seule matrice colonne x (on écrase à chaque passage les anciennes valeurs).

Chapitre 10

Remerciements

Je tiens à remercier Jean Ballat (jeannot45) et Patrick Gonord (diogene) pour les corrections apportées.

Bibliographie

- [1] G. Medić et B. Mohammadi, *NSIKE - an incompressible Navier-Stokes solver for unstructured meshes*, INRIA, 1999
- [2] Mark J. Harris, *Fast Fluid Dynamics Simulation on the GPU*
- [3] Joel H.Ferziger, Milovan Perić, *Computational Methods for Fluid Dynamics, second edition*, SPRINGER, 1999
- [4] Jos Stam, *Real-Time Fluid Dynamics for Games*

Table des figures

4.1	Grille de vecteurs vitesse avec les bords	12
5.1	Calcul de l'advection	19
5.2	Visualisation du champ de vecteurs vitesse de taille 100×100 avec obstacle . . .	24
5.3	Visualisation du champ de vecteurs vitesse de taille 50×50 sans obstacle	25
5.4	Visualisation du champ de vecteurs vitesse tronquée	25
6.1	Visualisation du champ de diffusion, taille 150×150	26
6.2	Visualisation du champ de scalaires diffusion	30
6.3	Visualisation du champ de scalaires diffusion	31
7.1	Visualisation du champ diffusion avec obstacles	35
7.2	Visualisation du champ vitesse avec obstacles	35
8.1	Visualisation de la pression avec obstacle	38
8.2	Détection de tourbillons	39
8.3	Phénomène de Karman	40
8.4	Détection de tourbillons	40