



Grenoble INP – ENSIMAG
Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées

# Introduction à la Recherche en Laboratoire

Effectuée au Laboratoire Jean Kuntzmann dans l'équipe EDP

# Solutions de viscosité d'équations aux dérivées partielles et schémas numériques associés

Gaétan Bahl
Élève en 2A MMIS

2015-2016

**Laboratoire Jean Kuntzmann**
51, rue des Mathématiques
BP 53
38041 Grenoble Cedex 9

**Encadrant**
Emmanuel Maître
Équipe EDP

# Résumé

Ce document présente les travaux que j'ai effectués dans le cadre du module d'Introduction à la Recherche en Laboratoire, au cours du second semestre de ma deuxième année à l'Ensimag, en filière Modélisation Mathématique, Image, et Simulation.

Ces travaux on été réalisés au Laboratoire Jean Kuntzmann (LJK-IMAG), au sein de l'équipe Équations aux Dérivées Partielles (EDP) et sous la tutelle d'Emmanuel Maître, professeur à Grenoble INP et membre de cette équipe.

Les connaissances utilisées pour ces travaux ne s'inscrivent pas intégralement dans le cursus de l'Ensimag, la notion de solution de viscosité n'y étant pas étudiée. Les cours de MMIS permettent néanmoins de mieux les aborder, en particulier le module *Modèles EDP et Schémas Numériques en Sciences de l'Ingénieur*, enseigné par Emmanuel Maître au premier semestre de deuxième année.

Cette Introduction à la Recherche en Laboratoire s'intéresse à une nouvelle notion de solution d'équations aux dérivées partielles, plus générale que les définitions classiques. Ceci permet de considérer des cas que les définitions classiques ne permettent pas de résoudre. Nous nous sommes plus spécifiquement intéressés à ce que cette nouvelle notion peut nous apporter dans le cadre du transport optimal, en implémentant et optimisant un schéma numérique résolvant l'équation de Monge-Ampère.

# Remerciements

Je tiens à remercier Emmanuel Maître, non seulement pour m'avoir permis de réaliser cette Introduction à la Recherche en Laboratoire, mais aussi pour sa patience, sa pédagogie, et son aide précieuse.

Je tiens également à remercier l'équipe EDP du laboratoire Jean Kuntzmann, pour son accueil chaleureux et pour m'avoir permis d'assister à leurs nombreux et très intéressants séminaires, qui m'ont permis de découvrir des applications des équations aux dérivées partielles et de mieux comprendre leurs enjeux dans le monde contemporain.

# Table des matières

# 1 Introduction

La notion de solution de viscosité a été introduite au début des années 80 par Pierre-Louis Lions et Michael G. Crandall [1]. Il s'agit d'une généralisation de la notion de solution d'Équation aux Dérivées Partielles. Cette notion peut être utile dans la résolution d'EDP linéaires, telles que l'équation de la chaleur, mais elle est surtout utile dans le cadre d'équations non-linéaires. Elle n'est en général *pas* utilisée dans la résolution de systèmes d'équations (ex. Navier-Stokes) [2].

Le but de cette IRL a tout d'abord été de comprendre la notion de solution de viscosité, puis d'implémenter sous Scilab un schéma numérique convergeant vers une solution de viscosité de l'équation de Monge-Ampère pour le transport optimal, et enfin de ré-implémenter ce schéma sous OpenCL, afin de l'exécuter rapidement sur un processeur graphique.

# 2 Solutions de Viscosité

## 2.1 Rappels sur les Équations aux Dérivées Partielles

Commençons par rappeler quelques définitions, afin de mieux situer le contexte du sujet.

**Définition : Equation aux Dérivées Partielles** Une Équation aux Dérivées Partielles (EDP) d'ordre $k$ est une équation liant une fonction inconnue u et ses dérivées jusqu'à l'ordre $k$.

En particulier, nous nous intéresserons au cas $k = 1, 2$, c'est-à-dire :

$$k = 1,\ F(x, u, Du) = 0, x \in \Omega \subset \mathbb{R}^n, \tag{1}$$
$$k = 2,\ F(x, u, Du, D^2u) = 0, x \in \Omega \subset \mathbb{R}^n, \tag{2}$$

où $F : \Omega \times \mathbb{R} \times \mathbb{R}^n \times \mathcal{M}_{n,n} \to \mathbb{R}$.

**Définition : EDP linéaire** Une Équation aux Dérivées Partielles est dite *linéaire* si elle peut s'écrire sous la forme :

$$Lu = f(x)$$

où $L$ est un opérateur différentiel linéaire d'ordre $k$. Dans tous les autres cas, l'EDP est dite *non-linéaire*.

L'équation de la chaleur, par exemple, est linéaire :

$$\frac{\partial u}{\partial t} - \Delta u = 0 \tag{3}$$

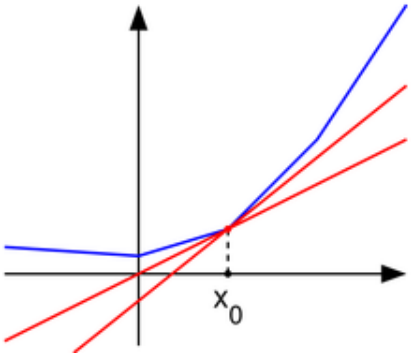


FIGURE 1 – Sous-gradient (tiré de Wikipedia.fr)

**Solution classique d'une EDP** Une fonction $u : \Omega \to \mathbb{R}^n$ est appelée *solution classique* de l'EDP 1 d'ordre $k$ si elle est $\mathcal{C}^k$ et satisfait l'EDP $\forall x \in \Omega$.

## 2.2 Justification de la nécessité des solutions de viscosité

Considérons l'équation eikonale (avec $f \equiv 1$) dans le problème de Dirichlet suivant :

$$\begin{cases} |Du| = 1, \ x \in [-1, 1] \\ u(x) = 0, \ |x| = 1 \end{cases} \tag{4}$$

Supposons qu'il exite $u$ solution classique de 4. $u$ est alors $\mathcal{C}^1$. On a $u(-1) = u(1) = 0$. Par le théorème des accroissements finis, il existe alors $x_0 \in ]-1, 1[$ tel que $u'(x) = 0$. Puisque $u$ est $\mathcal{C}^1$ il existe alors un voisinage de $x_0$ où $|u'(x)| < 1$. $u$ ne peut donc pas être solution de 4. C'est pourquoi nous avons besoin d'une notion de solution plus faible.

## 2.3 Définition d'une solution de viscosité

La définition d'une solution de viscosité et des explications plus détaillées peuvent être trouvées dans [3]. Nous en donnerons ici une explication intuitive dans un exemple en une dimension.

Afin de définir la notion de solution de viscosité, nous avons tout d'abord besoin de parler de *sous-différentielle* et de *sur-différentielle*.

**Définition : Sous-différentielle** Soit $u : \Omega \to \mathbb{R}$. Un vecteur $p$ est un sous-gradient (resp. sur-gradient) de $u$ en un point $x$ si la droite orientée par $p$ est tangente sous (resp. sur) la courbe de $u$ en $x$. La figure 1 montre un exemple de deux sous-gradients.

On note l'ensemble des sous-gradients (resp. sur-gradients) de $u$ en $x$ : $D^+u(x)$ (resp. $D^-u(x)$).

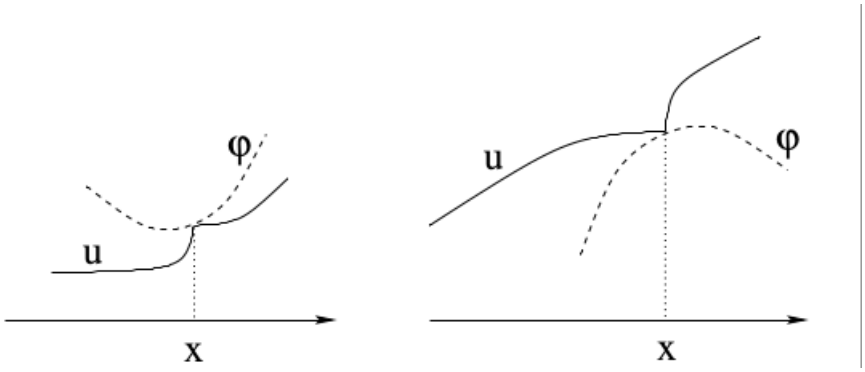


FIGURE 2 – Sous-différentielle (tiré de [3])

**Définition alternative : Sous-différentielle** Une définition alternative très utile de la notion de sous-différentielle d'une fonction $\mathcal{C}^k$ est la suivante :

On a $p \in D^+u(x)$ (resp. $D^-u(x)$) si et seulement si il existe une fonction $\phi \in \mathcal{C}^1(\Omega)$ telle que $\nabla\phi(x) = p$ et $u - \phi$ admet un minimum (resp. maximum) local en $x$.

Intuitivement, c'est le fait qu'il existe une fonction régulière dont la courbe touche celle de $u$ en $x$ et dont la dérivée est $p$. Un exemple est donné par la figure 2.

Ces définitions s'appliquent également aux dérivées d'ordres supérieurs.

**Définition : Solution de viscosité** Une fonction $u$ est une *sous-solution de viscosité* de 1 si

$$F(x, u(x), p) \leq 0, \forall x \in \Omega, p \in D^+u(x)$$

Une fonction $u$ est une *sur-solution de viscosité* de 1 si

$$F(x, u(x), p) \geq 0, \forall x \in \Omega, p \in D^-u(x)$$

Une fonction $u$ est une *solution de viscosité* si elle est à la fois une *sur-solution* et une *sous-solution* de viscosité.

## 2.4 Applications possibles des solutions de viscosité

Voici quelques exemples d'EDP non-linéaires pour lesquels les solutions de viscosité sont utiles [2]. Le contrôle optimal est une des applications principales des solutions de viscosité, en finance notamment. Une liste très fournie d'applications peut être trouvée dans [4].

**Équation de Hamilton-Jacobi**

$$H(x, u, Du) = 0$$

où $H$ est le *Hamiltonien*.

Cette équation est très importante en contrôle optimal et en économie, en particulier quand elle prend la forme *Hamilton-Jacobi-Bellman* [5].

**Mean Curvature Motion**

$$u_t - \Delta u + \langle D^2 u \frac{Du}{|Du|}, \frac{Du}{|Du|} \rangle$$

Cette équation de géométrie différentielle décrit l'évolution d'une hypersurface essayant de minimiser son aire, elle a des applications en physique.

**Equation de Monge-Ampère**

$$\det(D^2 u) = f(x)$$

Cette équation a des applications en transport optimal, et c'est sur celle-ci que nous avons choisi de nous concentrer.

# 3 Solutions de viscosité : Exemples

Nous allons maintenant voir quelques exemples de solutions de viscosité.

## 3.1 Exemple 1D : Équation eikonale

Nous revisitons ici l'équation eikonale

$$-|Du| + 1 = 0, \tag{5}$$

que nous avons vue plus haut.

Il s'agit de montrer que $u(x) = 1 - |x|$ en est solution de viscosité sur $[-1; 1]$.

Sur $[-1, 0[$ et $]0, 1]$, $u$ est différentiable et vérifie l'équation eikonale. La condition de sur-solution et de sous-solution est donc vérifiée sur ces intervalles.

Le problème se situe en 0, où $u$ n'est pas différentiable.

On a clairement $D^+u(0) = [-1; 1]$. On a donc bien $\forall p \in D^+u(0),\ -|p| + 1 \geq 0$, donc $u$ est une sous-solution de viscosité de l'équation 5.

On voit également que $D^-u(x) = \emptyset$, la propriété de sur-solution est donc trivialement vérifiée en 0.

$u$ est donc bien une solution de viscosité de 5.

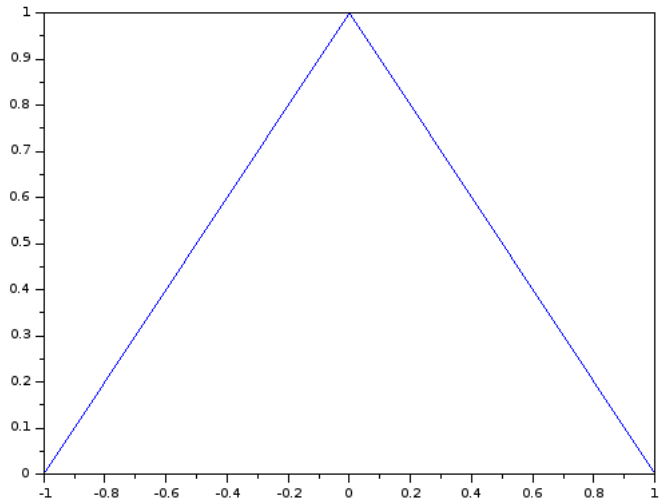


FIGURE 3 – Solution de viscosité de l'équation eikonale

## 3.2 Exemple 2D : Monge-Ampère

Le but est ici de prouver numériquement que $u(x) = \frac{1}{2}((|x| - 1)^+)^2$ est bien solution de viscosité de l'équation de Monge-Ampère : $\det(D^2u(x)) = f(x)$, quand $f(x) = (1 - \frac{1}{|x|})^+$.

### 3.2.1 Étude de la fonction u

On commence par représenter la fonction u sur $[-2, 2] \times [-2, 2]$ (figure 6).

On obtient une cuvette avec un fond plat. Cela se voit mieux lorsqu'on réalise une coupe de la courbe (figure 7).

La fonction est $C^2$ partout, sauf sur le cercle unité.

### 3.2.2 Vérification de l'équation

Pour montrer que $u$ vérifie l'équation de Monge-Ampère, on calcule le déterminant de sa Hessienne, et on le compare à $f$. Puisqu'on peut construire $u$ par révolution autour de l'axe des $z$, on se cantonne au plan $y = 0$.

On obtient la figure 8.

En soustrayant f à ce résultat, on obtient le résultat de la figure 9.

On voit que le résultat est très proche de $f$, à part sur le cercle unité, ce qui était attendu.

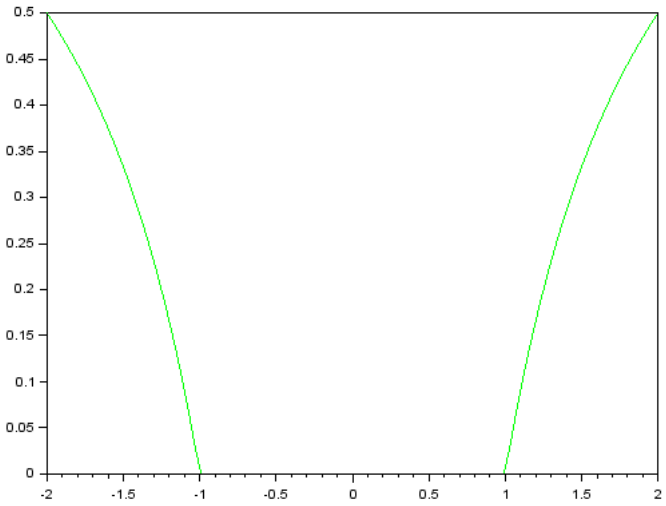


FIGURE 4 – fonction f

### 3.2.3 $u$ est $C^1$

On étudie maintenant la fonction $u$ en $(1,0)$. On sait que $u(1,0)=0$ et on veut étudier numériquement sa régularité en ce point.

On utilise le code Scilab suivant pour approximer $u'(1^+,0)$ par $\frac{u(1+dx)}{dx}$ en faisant tendre $dx>0$ vers 0 :

```
for i = 1:10
    dh = 10^(-i);
    disp(u(1+dh, 0)/dh);
end
```

Et on obtient :

```
    0.05
    0.005
    0.0005
    0.00005
    0.000005
    0.0000005
    5.000D-08
    5.000D-09
    5.000D-10
    5.000D-11
```

De plus, $u'(x,0)=0$ pour $x\in[-1,1]$, on a donc vérifié numériquement que $u$ est $C^1$.

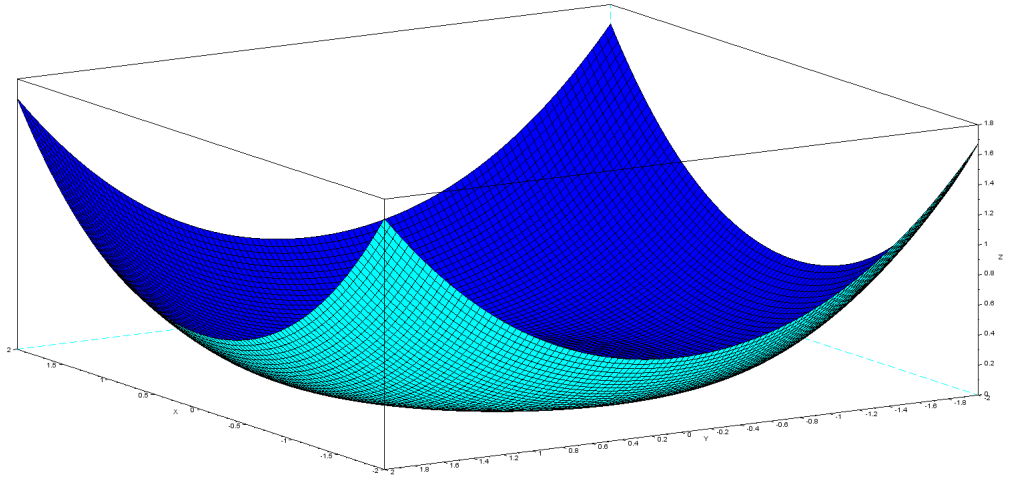

FIGURE 5 – fonction u vue d'au dessus

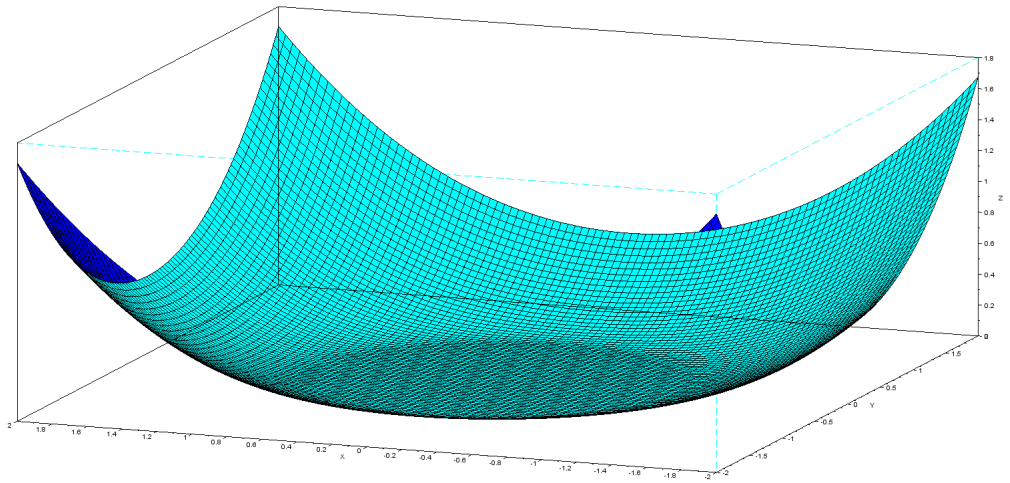

FIGURE 6 – fonction u vue d'en-dessous

### 3.2.4 Calcul des sous-gradients de $u$ en $(1, 0)$

Comme pour $u'$, on approxime $u''(1^+, 0)$ par $\frac{u(1+2dx,0)-2u(1+dx,0)}{dx^2}$, et on fait tendre $dx > 0$ vers 0.

On obtient :

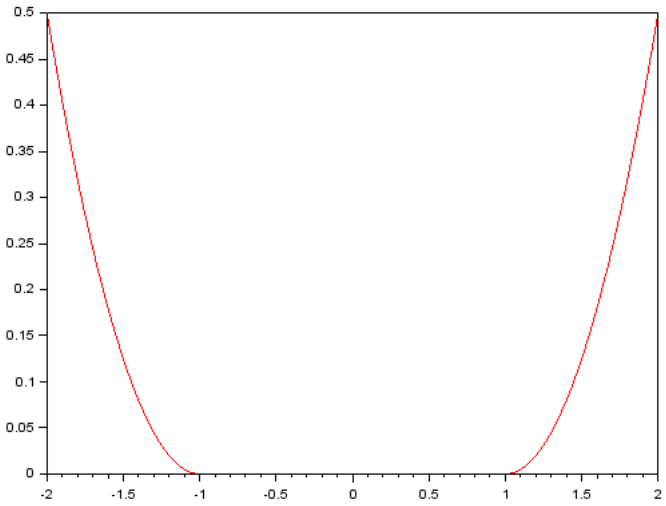


FIGURE 7 – fonction u dans le plan y = 0

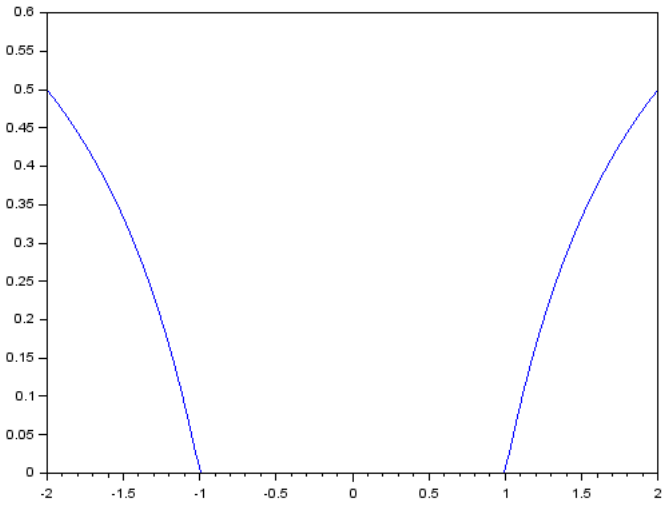


FIGURE 8 – $\det(D^2u)$ : membre de gauche de l'équation de Monge-Ampère

```
1.
1.
1.
```

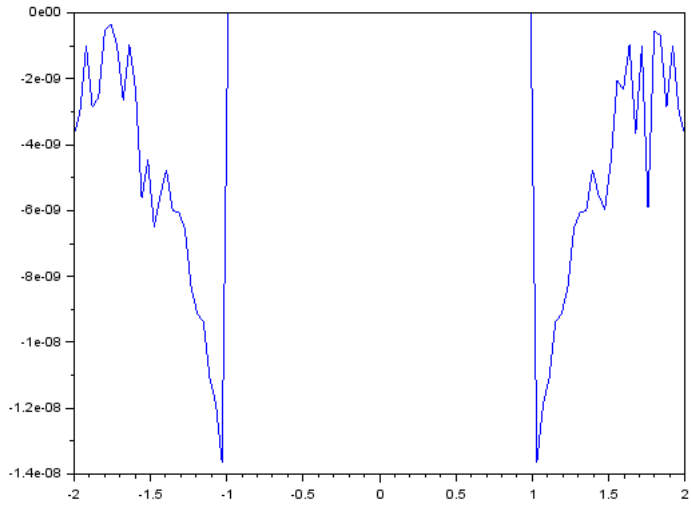


FIGURE 9 – Différence entre le membre de gauche et le membre de droite

```
4       1.
5       1.
6       1.
7       1.
8       1.
9       0.9999997
10      1.0000002
```

Ce qui nous donne que $D^{-}u'(1,0) = [0,1]$, car à gauche, $u''(1^{-},0) = 0$.

# 4 Résolution numérique de l'équation de Monge-Ampère

## 4.1 Présentation

La suite du sujet consistait en l'implémentation d'une méthode de résolution numérique. Nous avons choisi de travailler sur l'équation de Monge-Ampère, appliquée au transport optimal, en implémentant une méthode proposée dans [6].

Le transport optimal est un problème connu ayant beaucoup d'applications, mais dont les méthodes de résolution restent sous-développées par rapport à la théorie qui, elle, est bien établie [7].

Le problème est le suivant :

Nous disposons de deux mesures de probabilité, $\rho_X$ sur $X$ et $\rho_Y$ sur $Y$, avec $X, Y \in \mathbb{R}^n$. Il s'agit de trouver une application $\mathbb{M}$ appartenant à l'ensemble des applications transformant la densité $\rho_Y$ en la densité $\rho_X$, $\{T : X \to Y, \rho_Y(T)\det(\nabla T) = \rho_X\}$.

Cette application doit être optimale au sens de la fonction de coût quadratique :

$$\frac{1}{2}\int_X ||x - T(x)||^2 \rho_X(x)dx$$

Cela nous donne donc l'équation de Monge-Ampère suivante :

$$\det(D^2u) = \frac{\rho_X}{\rho_Y(\nabla u)},\ x \in \Omega \tag{6}$$

Dans [6], la condition au bord est donnée par une équation de Hamilton-Jacobi, le but étant d'envoyer le bord de $X$ sur le bord de $Y$ :

$$H(\nabla u(x)) = dist(\nabla u(x), \partial Y) = 0,\ x \in \partial\Omega \tag{7}$$

## 4.2 Exemple analytique

Commençons par un exemple analytique de mappage d'une densité sur une autre à l'aide d'une application que l'on choisit à la main. De cette manière, on connaît le résultat que l'on doit trouver, ce qui nous permettra de vérifier plus tard que notre implémentation converge bien vers la bonne solution.

On choisit pour $\rho_X$ une gaussienne (fig. 10) :

$$\rho_Y = \exp(-\frac{||x||^2}{0.1}) \tag{8}$$

On choisit ensuite la solution $u(x, y) = \dfrac{x^4 + y^4}{4}$, qui nous donne, en prenant son gradient, la map suivante :

$$\nabla u(x, y) = \begin{pmatrix} x^3 \\ y^3 \end{pmatrix} \tag{9}$$

Cette map (fig. 11) sépare la gaussienne dans les 4 quadrants du repère. Pour obtenir le résultat du mapping, on calcule :

$$\rho_Y(x) = \det(D^2u(x)\rho_X(\nabla u(x)),\ \forall x \in X$$

et nous obtenons le résultat représenté sur la figure 12.

## 4.3 Méthode de résolution

La méthode de résolution que nous avons implémentée est celle qui est proposée dans [6].

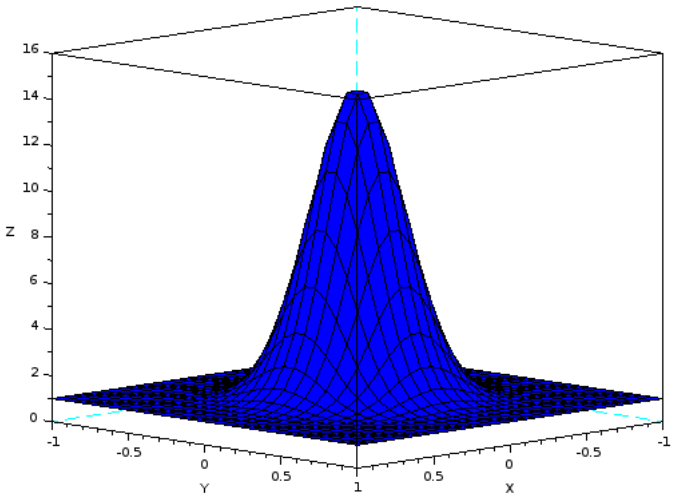


FIGURE 10 – La gaussienne

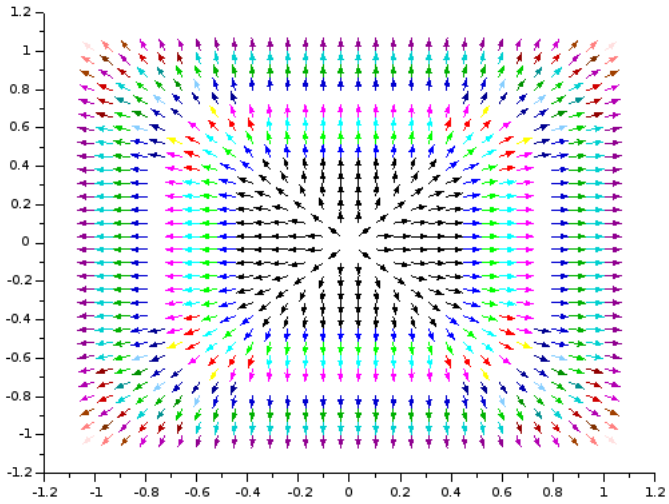


FIGURE 11 – Map choisie

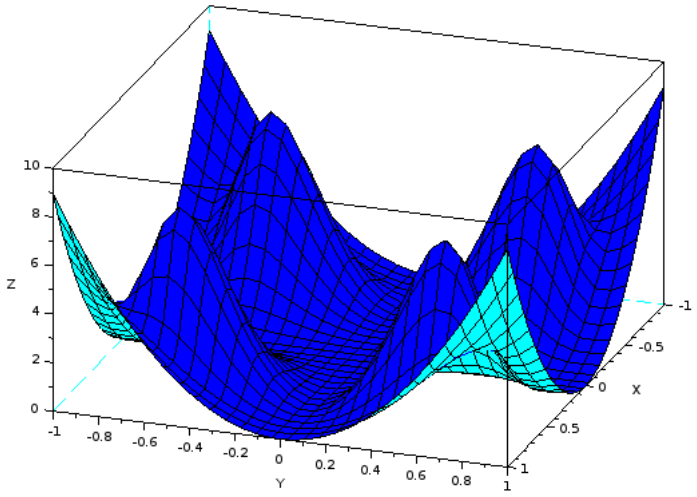


FIGURE 12 – Résultat du mappage

Il s'agit d'utiliser simultanément 3 schémas aux différences finies. Les deux premiers servant à résoudre l'équation de Monge-Ampère, le troisième servant à calculer la condition au bord.

Les opérateurs aux différences finies utilisées sont :

$$
\begin{aligned}
[D_{x_1x_1}u]_{i,j} &= \frac{1}{dx^2}(u_{i+1,j} + u_{i-1,j} - 2u_{i,j}) \\
[D_{x_2x_2}u]_{i,j} &= \frac{1}{dx^2}(u_{i,j+1} + u_{i,j-1} - 2u_{i,j}) \\
[D_{x_1x_2}u]_{i,j} &= \frac{1}{4dx^2}(u_{i+1,j+1} + u_{i-1,j-1} - u_{i-1,j+1} - u_{i+1,j-1}) \\
[D_{x_1}u]_{i,j} &= \frac{1}{2dx}(u_{i+1,j} - u_{i-1,j}) \\
[D_{x_2}u]_{i,j} &= \frac{1}{2dx}(u_{i,j+1} - u_{i,j-1})
\end{aligned}
$$

On utilise également des différences finies selon les directions $v = (\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$ et $v^{\perp} = (\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}})$ :

$$
\begin{aligned}
[D_{vv}u]_{i,j} &= \frac{1}{2dx^2}(u_{i+1,j+1} + u_{i-1,j-1} - 2u_{i,j}) \\
[D_{v^\perp v^\perp}u]_{i,j} &= \frac{1}{2dx^2}(u_{i+1,j-1} + u_{i-1,j+1} - 2u_{i,j}) \\
[D_{v}u]_{i,j} &= \frac{1}{2\sqrt{2}dx}(u_{i+1,j+1} - u_{i-1,j-1}) \\
[D_{v^\perp}u]_{i,j} &= \frac{1}{2\sqrt{2}dx}(u_{i+1,j-1} - u_{i-1,j+1})
\end{aligned}
$$

Les deux schémas pour l'équation de Monge-Ampère sont un schéma monotone, noté MAM, et un schéma plus précis, noté MAA :

$$MAA = D_{x_1x_1}uD_{x_2x_2}u - (D_{x_1x_2}u)^2 - \rho_X/\rho_Y(D_{x_1}u, D_{x_2}u) - u_0 \tag{10}$$

$$
\begin{aligned}
MA1 &= max\{D_{x_1x_1}, \delta\}max\{D_{x_2x_2}, \delta\} - min\{D_{x_1x_1}, \delta\} - min\{D_{x_2x_2}, \delta\} - \rho_X/\rho_Y(D_{x_1}u, D_{x_2}u) - u_0 \\
MA2 &= max\{D_{vv}, \delta\}max\{D_{v^\perp v^\perp}, \delta\} - min\{D_{vv}, \delta\} - min\{D_{v^\perp v^\perp}, \delta\} - \rho_X/\rho_Y(D_{v}u, D_{v^\perp}u) - u_0 \\
MAM &= min\{MA1, MA2\}
\end{aligned}
$$

Un filtrage permet, une fois les deux schémas calculés, de les mélanger correctement.

## 4.4 Implémentation Scilab

Nous avons commencé par implémenter le schéma précis présenté plus haut.

Par soucis de simplicité et de temps, mais aussi parce qu'elle sera plus facile à implémenter sous OpenCL par la suite, nous avons utilisé une itération explicite (Euler) au lieu de la méthode de Newton qui était conseillée dans [6]. Cette méthode est sujette à une condition CFL très restreinte qui nous force à utiliser un pas de descente très petit. De ce fait, la convergence se fait très lentement.

L'itération s'écrit :

$$\frac{du}{dt} = \det(D^2u) - \frac{\rho_X}{\rho_Y(\nabla u)},\ x \in X \tag{11}$$

Nous avons ensuite implémenté la condition au bord. Celle-ci est compliquée à implémenter, en particulier il est difficile de réaliser les calculs efficacement. Il est absolument nécessaire de précalculer la plupart des opérations pour n'avoir à les faire qu'une seule fois au début du programme. Pour les détails, voir [6].

Enfin, nous avons implémenté le schéma monotone. Celui-ci est compliqué, il subsiste des bugs que nous n'avons pas pu corriger par manque de temps, en particulier sur l'intéraction avec le bord. Il est cependant fonctionnel à l'intérieur du domaine. Je suspecte

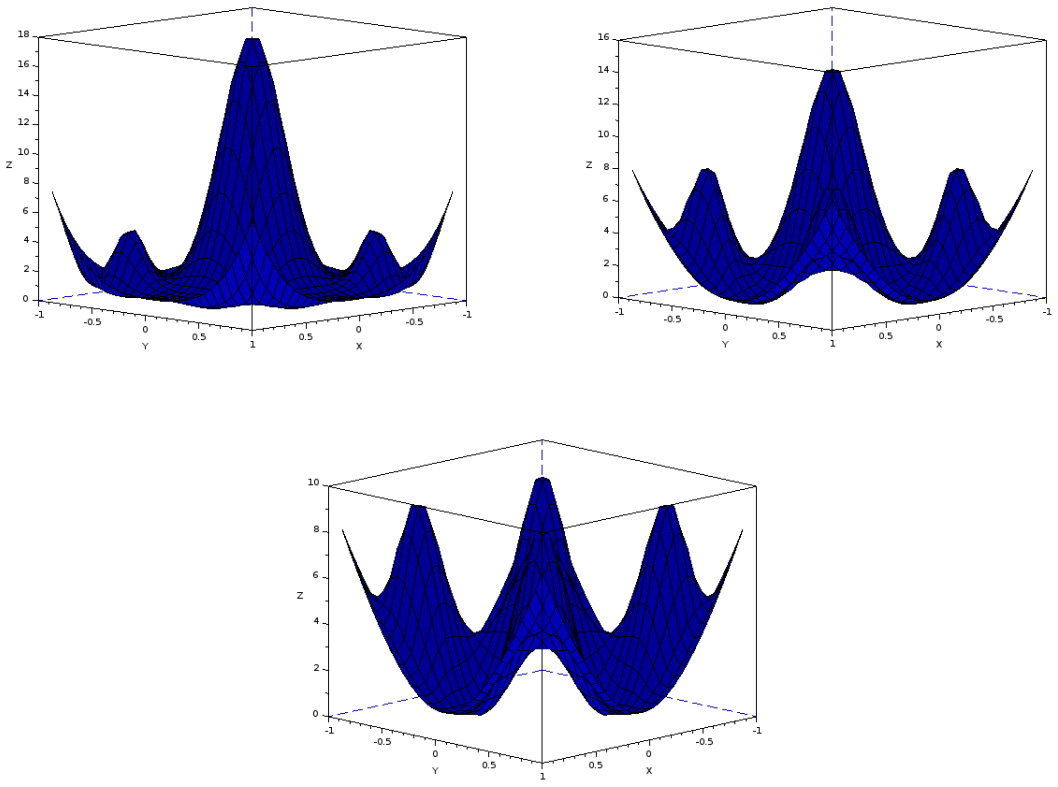


FIGURE 13 – Iterations 140, 200, et 700

l'existence de fautes de frappe dans [6], notamment dans les différences finies et le détail des schémas.

## 4.5 Résultats

En reprenant l'exemple vu plus haut et en calculant cette fois numériquement u, on voit que notre implémentation converge bien vers la solution attendue (figure 13).

Nous avons ensuite essayé de mapper la densité uniforme sur une gaussienne centrée en zéro. L'implémentation converge, lentement mais sûrement, vers la solution attendue (figure 14).

Idem, en essayant de déplacer une gaussienne sur une autre située en un autre endroit (figure 16).

Nous avons ensuite essayé un cas plus complexe avec des densités non continues. Il s'agit de déplacer un disque. L'algorithme converge vers une solution qui est proche de celle qui doit être trouvée.

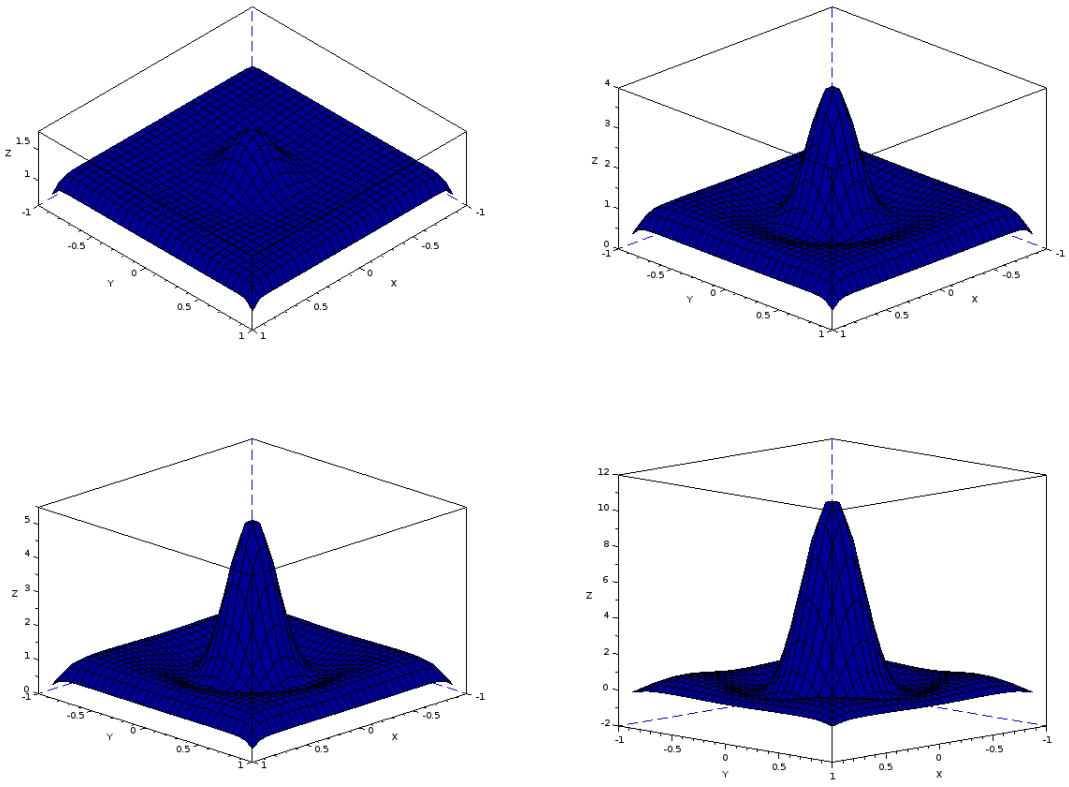


FIGURE 14 – Iterations 5, 50, 100 et 200 pour le mappage d'une gaussienne sur la densité uniforme

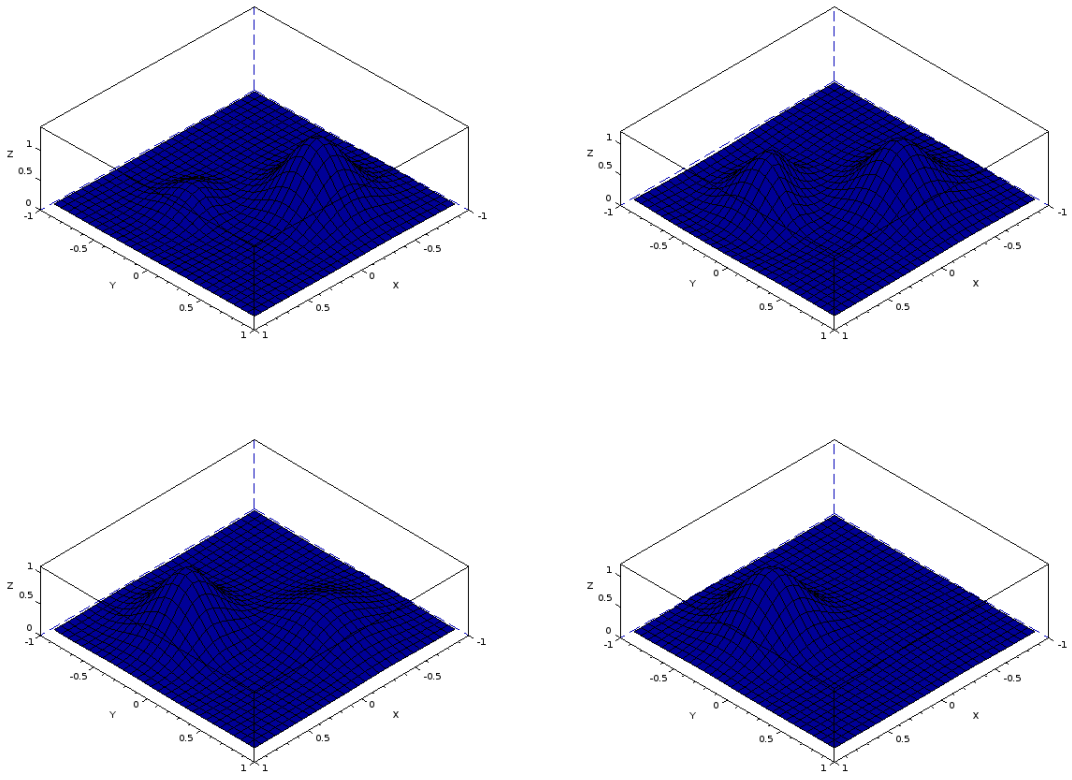


FIGURE 15 – Iterations pour le mappage d'une gaussienne sur une autre

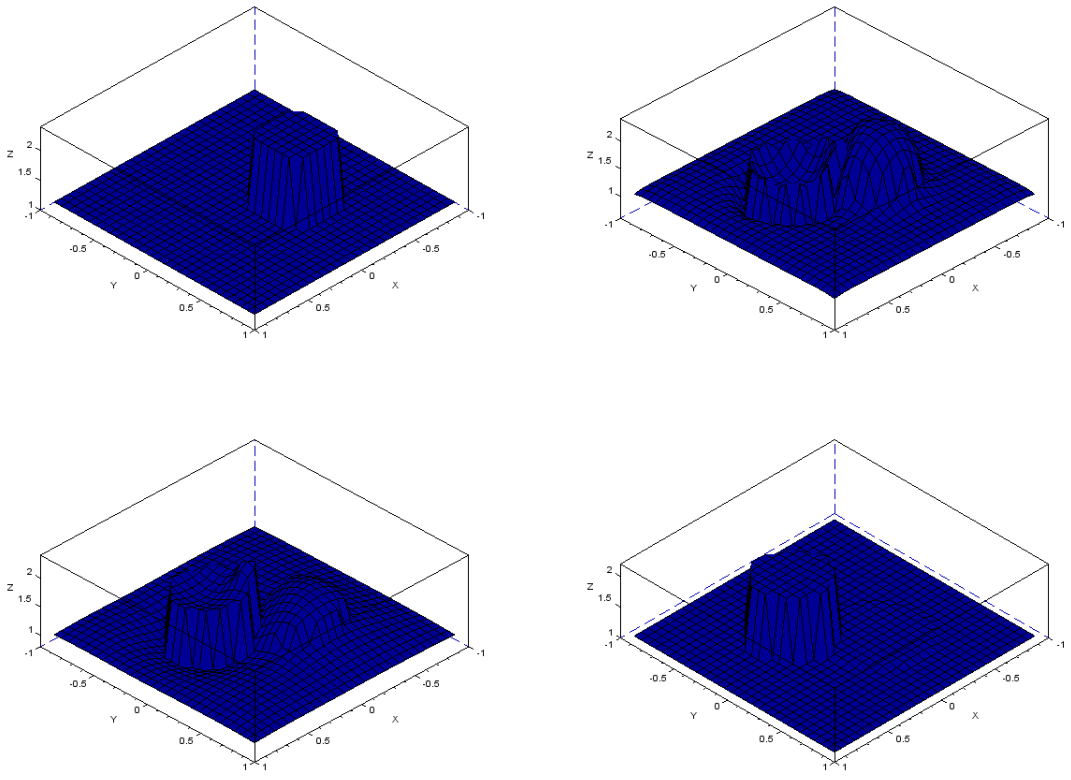


FIGURE 16 – Iterations 1, 15, 80, et 500 pour le déplacement d'un disque

## 4.6 Implémentation OpenCL

### 4.6.1 Qu'est-ce qu'OpenCL ?

OpenCL (pour *Open Compute Language*) et un langage proche du C permettant de faire facilement du calcul parallèle pour différentes plate-formes d'execution[8]. En particulier, les processeurs classiques (CPU) et les processeurs graphiques (GPU). Il est également possible d'utiliser d'autres types de puces, notamment les accélérateurs Xeon Phi d'Intel.

Il est similaire à CUDA dans son fonctionnement, mais je ne pouvais pas utiliser ce dernier car il ne fonctionne que sur les cartes graphiques Nvidia.

### 4.6.2 Qu'est-ce qu'un GPU ?

Un GPU, ou *Graphics Processing Unit*, est une puce étant à l'origine dédiée à l'affichage 2D et 3D. Elle est constituée de centaines, voire milliers de petits processeurs, qui vont tous exécuter la même tâche au même moment. Récemment, les GPU ont commencé à être utilisés pour du calcul scientifique, car leur architecture permet une haute parallélisation des tâches, ce qui est très pratique pour de la simulation, du calcul matriciel, etc.

### 4.6.3 Travail réalisé

Les différences finies dont nous avons besoin peuvent être calculées indépendamment pour chaque point de la grille. Notre problème peut donc très fortement bénéficier de la parallélisation.

Seul le schéma MAA a été implémenté en OpenCL, et exécuté sur le circuit graphique d'une puce Intel Ivy Bridge. Les gains en vitesse obtenus en passant sur GPU sont très importants, surtout en comparaison avec la vitesse que nous avions sur Scilab.

Le GPU nous permet également d'obtenir plus de précision, car il est possible d'augmenter significativement le nombre de points de la grille sur laquelle nous travaillons. En effet, la vitesse avec une grille de 30 par 30 sous Scilab est environ la même qu'avec une grille de 5000 par 5000 sur GPU (environ 2 itérations par seconde) !

### 4.6.4 Détails d'implémentation

Deux programmes ont été écrits à l'aide du SDK OpenCL d'Intel et de la documentation d'OpenCL [9] :

— Le premier s'exécute sur le processeur hôte (CPU). Il se charge d'initialiser et de lancer les calculs. En particulier, il alloue des *buffers* sur le GPU pour contenir les données après les avoir initialisées. Il récupère les résultats une fois que le GPU a terminé les calculs.

— Le deuxième, appelé *kernel*, est exécuté sur chaque processeur de flux du GPU. C'est celui qui va calculer les différences finies et mettre à jour la solution. Dans notre implémentation, le kernel réalise la mise à jour de la solution $u$ en un seul

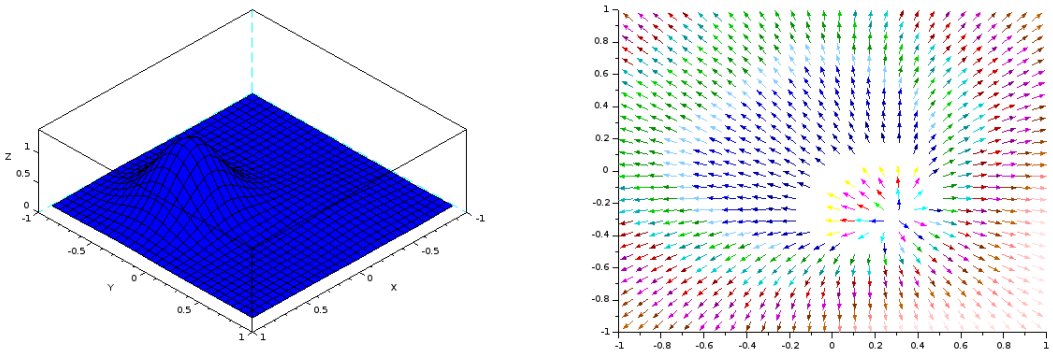


FIGURE 17 – Résultat obtenu après 1000 itérations sous OpenCL, avec une grille de 30x30

point de la grille. Les différents points de la grille sont divisés en tâches, qui vont être réparties entre les différents processeurs de flux.

Dans une implémentation itérative classique, la mise à jour de la solution se fait à l'aide d'une boucle *for*. Pour passer à OpenCL, il faut prendre l'intérieur de la boucle, pour le mettre dans le *kernel*.

Le programme hôte est composé de plusieurs étapes, qui doivent être réalisées avant le calcul. Elles entraînent un certain temps de mise en place, appelé *overhead*. Il faut : allouer les buffers sur l'hôte, les initialiser, allouer les buffers sur le GPU, copier les buffers de l'hôte au GPU, compiler le *kernel* pour l'achitecture spécifique de notre GPU, positionner les arguments du *kernel*, et enfin le lancer. Ces opérations sont chacune une source potentielle d'erreurs (manque de mémoire sur l'hôte, manque de mémoire sur le GPU, erreur de compilation du kernel, absence de GPU,...), il est donc important de vérifier la valeur de retour de chacune des fonctions afin d'être au courant de l'emplacement des erreurs que l'on rencontre, pour mieux déboguer le programme.

### 4.6.5 Résultats

Nous avons tout d'abord essayé de retrouver les résultats de la partie précédente. Pour le déplacement de la gaussienne, l'application calculée est la même que celle calculée par Scilab (cf fig. 17). Celle-ci a été calculée en 0,15s par le GPU, contre 1min10s sous Scilab.

Ensuite, nous avons décidé de calculer ce même déplacement, mais sur une grille de $256 \times 256$ (cf fig 18). Dans ce cas, la condition CFL est encore plus restrictive, et nous avons besoin d'un nombre très important d'itérations. Le GPU a calculé 200000 itérations en 45s. Une seule itération sous Scilab a pris 24s, il aurait donc fallu un peu moins de 8 semaines pour terminer le calcul sous Scilab.

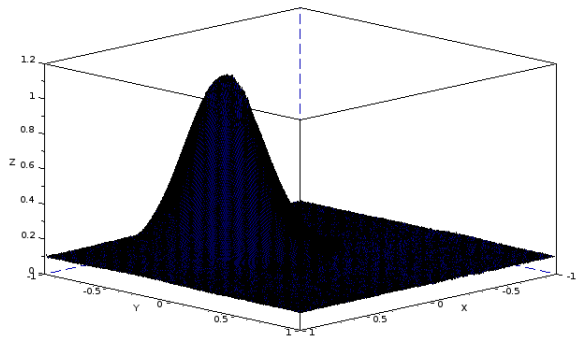


FIGURE 18 – Résultat obtenu après 200000 itérations sous OpenCL, avec une grille de 256x256

# 5 Conclusion

## 5.1 Résumé

Durant cette Introduction à la Recherche en Laboratoire, nous avons étudié une nouvelle notion de solution d'Équation aux Dérivées Partielles. Nous avons implémenté un schéma numérique résolvant l'équation de Monge-Ampère, tiré d'un article de recherche, et nous l'avons ensuite ré-implémenté en OpenCL pour être exécuté sur processeur graphique.

Nous avons obtenu des gains très importants avec le passage sur GPU, en particulier, le temps d'exécution de notre méthode d'itération (itération explicite) sur GPU est maintenant du même ordre de grandeur que le temps d'exécution de la méthode de Newton implémentée dans l'article duquel nous avons tiré notre schéma [6].

## 5.2 Perspectives

Les perspectives d'amélioration sont très claires. Puisqu'il subsiste des erreurs dans le schéma $MAM$, il faut les trouver et les résoudre, ce que je n'ai malheureusement pas eu le temps de faire. Une fois le problème corrigé, il sera aussi possible d'implémenter ce schéma sous OpenCL.

Il est également possible de paralléliser le schéma numérique utilisé pour résoudre l'équation d'Hamilton-Jacobi au bord du domaine et de le porter sous OpenCL.

La méthode de Newton pourrait également être implémentée. Cela n'a pas été fait ici car elle est plutôt complexe dans le cas de ce schéma numérique, et aurait donc été difficile à implémenter dans le temps imparti, autant en Scilab qu'en OpenCL.

## 5.3 Bilan de l'IRL

Ce travail de recherche a été pour moi très enrichissant. J'ai pu approfondir et mettre en pratique des connaissances étudiées en MMIS, ce qui m'a permis de prendre un peu de recul par rapport à ce que nous avons étudié au premier semestre.

Ayant fait un stage en laboratoire de recherche l'été dernier, je ne peux pas dire que je ai découvert l'ambiance d'un laboratoire uniquement grâce à l'IRL. J'ai cependant eu l'occasion de mieux comprendre les méthodes de recherche, en particulier parce que mon sujet n'était pas intégralement défini lorsque l'IRL a commencé. J'ai en effet eu l'opportunité de choisir moi-même sur quel problème j'allais travailler après avoir compris la notion de solution de viscosité.

J'ai pu, pendant ce semestre, assister à des séminaires de recherche au sein du Laboratoire Jean Kuntzmann, ce qui m'a permis de découvrir de nouvelles méthodes et de nouvelles applications des Équations aux Dérivées Partielles. Ceci a nettement renforcée l'idée que j'ai de vouloir continuer dans la voie de la simulation et de l'optimisation.

# Références

[1] M. G. Crandall and P.-L. Lions, "Viscosity solutions of hamilton-jacobi equations," *Transactions of the American Mathematical Society*, vol. 277, no. 1, pp. 1–42, 1983.

[2] F. Dragoni, "Introduction to viscosity solutions for nonlinear pdes."

[3] A. Bressan, "Viscosity solutions of hamilton-jacobi equations and optimal control problems," *Lecture notes*, 2010.

[4] M. G. Crandall, H. Ishii, and P.-L. Lions, "User's guide to viscosity solutions of second order partial differential equations," *Bulletin of the American Mathematical Society*, vol. 27, no. 1, pp. 1–67, 1992.

[5] M. Bardi and I. Capuzzo-Dolcetta, *Optimal control and viscosity solutions of Hamilton-Jacobi-Bellman equations*. Springer Science & Business Media, 2008.

[6] J.-D. Benamou, B. D. Froese, and A. M. Oberman, "Numerical solution of the optimal transportation problem using the monge–ampere equation," *Journal of Computational Physics*, vol. 260, pp. 107–126, 2014.

[7] C. Villani, *Optimal transport : old and new*, vol. 338. Springer Science & Business Media, 2008.

[8] J. E. Stone, D. Gohara, and G. Shi, "Opencl : A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 1-3, pp. 66–73, 2010.

[9] K. GROUP *et al.*, "The khronos group," *Beaverton, 2011b. Disponível em*, 2009.

# 6 Annexe A : Code Scilab

```
//constantes


//delta = 1 + (K * h)/sqrt(2)
delta = 3; //en attendant
epsilon = 0.1
Niter = 100000;

Npoints = 30;

dt = 0.00001;

Nvectsbord = 201;

xmin=-1;
xmax=1;
ymin=-1;
ymax=1;

h = (xmax - xmin)/Npoints;

x = linspace(xmin, xmax, Npoints);
y = linspace(ymin, ymax, Npoints);

v = [1/sqrt(2) 1/sqrt(2)];
vorth = [1/sqrt(2) -1/sqrt(2)];
////////

//fonctions de densite
function z = rhoY(x,y)
//    z = 1
//    z =  exp(- norm([x+0.3,y-0.3]) ^2 / 0.1) +0.1;
    //z =  (Npoints^2)/66.051535  * exp(- norm([x,y]) ^2 / 0.1) +1;

    z =  0.1 + exp(- norm([x+0.3,y-0.3]) ^2 / 0.1);

//    if sqrt((x+0.2)^2 + (y-0.2)^2) < 0.3 then
//        z = 2
//    else
//        z = 1
//    end
//
endfunction
function z = rhoX(x,y)
    //z = 9 * rhoY(x^3,y^3) * x^2 * y^2 +1
    //z =  0.1 + exp(- norm([x,y]) ^2 / 0.1);
//    z =  (Npoints^2)/66.051535  * exp(- norm([x,y]) ^2 / 0.1) +0.1;
    z =  exp(- norm([x-0.3,y+0.3]) ^2 / 0.1) +0.1;
//
```

```
50 //     if sqrt((x-0.2)^2 + (y+0.2)^2) < 0.3 then
51 //         z = 2
52 //     else
53 //         z = 1
54 //     end
55 //
56 endfunction
57
58 truc = 0
59 for i = 1:Npoints
60     for j = 1:Npoints
61         truc(i,j) = rhoX(x(i),y(j))
62     end
63 end
64 clf();
65 plot3d(x,y,truc)
66 clf();
67 //c = 0
68 //for i = 1:Npoints
69 //     for j = 1:Npoints
70 //         c = c + rhoX(x(i),y(j))
71 //     end
72 //end
73
74 //disp(c)
75
76 set(gcf(),"color_map",[jetcolormap(64);hotcolormap(64)])
77 //Sfgrayplot(x,y,rhoX,strf="041",colminmax=[1,64])
78
79 ///////
80
81 //fonction pour le filtrage
82
83 function sx = S(x)
84     sx = zeros(Npoints-2,Npoints-2)
85     for i = 1:(Npoints-2)
86         for j = 1:(Npoints-2)
87
88     if x(i,j) <= 1 then
89         sx(i,j) = x(i,j)
90     elseif abs(x(i,j)) >= 2 then
91         sx(i,j) = 0
92     elseif x(i,j) > 1 then
93         sx(i,j) = 2 - x(i,j)
94     elseif x(i,j) < -1 then
95         sx(i,j) = - x(i,j) - 2
96     end
97 end
98 end
99
100 endfunction
```

```
101
102
103 // fonction pour calculer la constante de Lipschitz K
104
105
106 function K = lip(f,g, x, y, norme)
107
108     K = 0;
109
110     xmax = 0;
111     ymax = 0;
112
113     function z = unsurG(x)
114         z = 1/g(x(1),x(1));
115     endfunction
116     //a changer pour une autre fonction si jamais
117     [J,H] = numderivative(unsurG,[x(1),y(1)],[],[],"blockmat");
118
119     K = norm(J,norme);
120
121     for i = x
122         for j = y
123             [J,H] = numderivative(unsurG,[i,j],[],[],"blockmat");
124              tmp = norm(J,norme);
125
126              if(tmp > K) then
127                  K = tmp;
128              end
129         end
130     end
131
132
133
134 endfunction
135
136 K = lip(rhoX, rhoY,x,y,1);
137 delta = 0.5+ (K * h)/sqrt(2)
138 disp(K);
139 disp(delta)
140 //operateurs qui vont bien
141 function y = Dxx(u, i, j, h)
142     y = (1/h^2) * (u(i+1,j) + u(i-1,j) - 2*u(i,j));
143 endfunction
144
145 function y = Dyy(u, i, j, h)
146     y = (1/h^2) * (u(i,j+1) + u(i,j-1) - 2*u(i,j));
147 endfunction
148
149 function y = Dxy(u, i, j, h)
150     y = (1/(4 *h^2)) * (u(i+1,j+1) + u(i-1,j-1) - u(i-1,j+1) - u(i+1,j-1)
        );
```

```
151 endfunction
152
153 function y = Dx(u, i, j, h)
154     y = (1/(2*h)) * (u(i+1,j) - u(i-1,j));
155 endfunction
156
157 function y = Dy(u, i, j, h)
158     y = (1/(2*h)) * (u(i,j+1) - u(i,j-1));
159 endfunction
160
161 function y = Dvv(u, i, j, h)
162     y = (1/(2*h^2)) * (u(i+1,j+1) + u(i-1,j-1) - 2*u(i,j));
163 endfunction
164
165 function y = Dvvorth(u, i, j, h)
166     y = (1/(2*h^2)) * (u(i+1,j-1) + u(i-1,j+1) - 2*u(i,j));
167 endfunction
168
169 function y = Dv(u, i, j, h)
170     y = (1/(2*sqrt(2)*h)) * (u(i+1,j+1) - u(i-1,j-1));
171 endfunction
172
173 function y = Dvorth(u, i, j, h)
174     y = (1/(2*sqrt(2)*h)) * (u(i+1,j-1) - u(i-1,j+1));
175 endfunction
176 //////
177
178 // diff finie pour le bord
179
180 function r = DunBordGauche (u, i, j, h,n)
181     r = (1/h) * (n(1) * (u(i+1,j) - u(i,j)) + max(n(2),0) * (u(i,j) - u(i
        ,j-1)) + min(n(2),0) * (u(i,j+1) - u(i,j)))
182 endfunction
183
184 function r = DunBordDroite (u, i, j, h,n)
185     r = (1/h) * (n(1) * (u(i,j) - u(i-1,j)) + max(n(2),0) * (u(i,j) - u(i
        ,j-1)) + min(n(2),0) * (u(i,j+1) - u(i,j)))
186 endfunction
187
188 function r = DunBordHaut (u, i, j, h,n)
189     r = (1/h) * (n(2) * (u(i,j) - u(i,j-1)) + max(n(1),0) * (u(i,j) - u(i
        -1,j)) + min(n(1),0) * (u(i+1,j) - u(i,j)))
190 endfunction
191
192 function r = DunBordBas (u, i, j, h,n)
193     r = (1/h) * (n(2) * (u(i,j+1) - u(i,j)) + max(n(1),0) * (u(i,j) - u(i
        -1,j)) + min(n(1),0) * (u(i+1,j) - u(i,j)))
194 endfunction
195
196 function r = DunCoinHautGauche(u,i,j,h,n)
```

```
197     r = (1/h) * (n(1) * (u(i+1,j) - u(i,j)) + n(2) * (u(i,j) - u(i,j-1))
        )
198 endfunction
199
200 function r = DunCoinHautDroite(u,i,j,h,n)
201     r = (1/h) * (n(1) * (u(i,j) - u(i-1,j)) + n(2) * (u(i,j) - u(i,j-1))
        )
202 endfunction
203
204 function r = DunCoinBasGauche(u,i,j,h,n)
205     r = (1/h) * (n(1) * (u(i+1,j) - u(i,j)) + n(2) * (u(i,j+1) - u(i,j))
        )
206 endfunction
207
208 function r = DunCoinBasDroite(u,i,j,h,n)
209     r = (1/h) * (n(1) * (u(i,j) - u(i-1,j)) + n(2) * (u(i,j+1) - u(i,j))
        )
210 endfunction
211
212 ///// calcul de l'ensemble drond y
213
214 drondY = 0
215 //bord haut
216 for i = 1:Npoints
217     drondY(i,1) = x(i)
218     drondY(i,2) = ymax
219 end
220
221 //bord bas
222 for i = 1:Npoints
223     drondY(i+Npoints,:) = [x(i) ymin]
224 end
225
226 //bord gauche
227 for i = 1:Npoints
228     drondY(i+2*Npoints,:) = [xmin y(i)]
229 end
230
231 //bord droite
232 for i = 1:Npoints
233     drondY(i+3*Npoints,:) = [xmax y(i)]
234 end
235
236 // tous les vecteurs n
237
238 vectsgauche = [0 0]
239 vectsdroite = [0 0]
240 vectshaut = [0 0]
241 vectsbas = [0 0]
242
243
```

```
244 for i = 1:Nvectsbord
245
246     vectsgauche(i,:) = [cos(%pi/2 + ((i-1) * %pi)/(Nvectsbord-1)) sin(%pi
        /2 + ((i-1) * %pi )/(Nvectsbord-1))]
247     vectsdroite(i,:) = [cos(3*%pi/2 + ((i-1) * %pi )/(Nvectsbord-1)) sin
        (3*%pi/2 + ((i-1) * %pi )/(Nvectsbord-1))]
248     vectshaut(i,:) = [cos(((i-1) * %pi )/(Nvectsbord-1)) sin(((i-1) * %pi
         )/(Nvectsbord-1))]
249     vectsbas(i,:) = [cos(%pi + ((i-1) * %pi )/(Nvectsbord-1)) sin(%pi +
        ((i-1) * %pi )/(Nvectsbord-1))]
250
251 end
252
253 vectshautgauche = vectsgauche(1:(Nvectsbord/2),:)
254 vectshautdroite = vectshaut(1:(Nvectsbord/2),:)
255 vectsbasgauche = vectsbas(1:(Nvectsbord/2),:)
256 vectsbasdroite = vectsdroite(1:(Nvectsbord/2),:)
257
258 //fonction pour H*(n)
259
260 function r = Hetoile(n, drondY)
261
262     //je fais le calcul bebe maintenant, je changerai plus tard
263     r = sum(n .* drondY(1,:))
264     for i = 1:(4*Npoints)
265         tmp = sum(n .* drondY(i,:))
266         r = max(r,tmp)
267     end
268
269 endfunction
270
271 //pre-calcul des H*(n)
272
273 Hetoilegauche = 0
274 Hetoiledroite = 0
275 Hetoilehaut = 0
276 Hetoilebas = 0
277 Hetoilehautgauche = 0
278 Hetoilebasgauche = 0
279 Hetoilehautdroite = 0
280 Hetoilebasdroite = 0
281
282 for k = 1:Nvectsbord
283     Hetoilegauche(k) = Hetoile(vectsgauche(k,:),drondY)
284     Hetoiledroite(k) = Hetoile(vectsdroite(k,:),drondY)
285     Hetoilehaut(k) = Hetoile(vectshaut(k,:),drondY)
286     Hetoilebas(k) = Hetoile(vectsbas(k,:),drondY)
287 end
288
289 for k = 1:(Nvectsbord/2)
290     Hetoilehautgauche(k) = Hetoile(vectshautgauche(k,:),drondY)
```

```
291     Hetoilehautdroite(k) = Hetoile(vectshautdroite(k,:),drondY)
292     Hetoilebasgauche(k) = Hetoile(vectsbasgauche(k,:),drondY)
293     Hetoilebasdroite(k) = Hetoile(vectsbasdroite(k,:),drondY)
294 end
295 
296 
297 // fonctions pour Hamilton Jacobi au bord
298 
299 function r = Hdeltaugauche(u,i,j,h)
300 
301     r = -1000;
302     for k = 1:Nvectsbord
303 
304         n = vectsgauche(k,:);
305         Hstar = Hetoilegauche(k);
306         deltauxn = DunBordGauche(u,i,j,h,n);
307 
308         res = deltauxn - Hstar;
309         r = max(r,res);
310     end
311 
312 endfunction
313 
314 function r = Hdeltaudroite(u,i,j,h)
315 
316     r = -1000;
317     for k = 1:Nvectsbord
318 
319         n = vectsdroite(k,:);
320         Hstar = Hetoiledroite(k);
321         deltauxn = DunBordDroite(u,i,j,h,n);
322 
323         res = deltauxn - Hstar;
324         r = max(r,res);
325     end
326 
327 endfunction
328 
329 function r = Hdeltauhaut(u,i,j,h)
330 
331     r = -1000;
332     for k = 1:Nvectsbord
333 
334         n = vectshaut(k,:);
335         Hstar = Hetoilehaut(k);
336         deltauxn = DunBordHaut(u,i,j,h,n);
337 
338         res = deltauxn - Hstar;
339         r = max(r,res);
340     end
341 
```

```
342 endfunction
343
344 function r = Hdeltaubas(u,i,j,h)
345
346     r = -1000;
347     for k = 1:Nvectsbord
348
349         n = vectsbas(k,:);
350         Hstar = Hetoilebas(k);
351         deltauxn = DunBordBas(u,i,j,h,n);
352
353
354         res = deltauxn - Hstar;
355         r = max(r,res);
356     end
357
358 endfunction
359
360 function r = HdeltauCoinHautGauche(u,i,j,h)
361
362     r = -1000;
363     for k = 1:(Nvectsbord/2)
364
365         n = vectshautgauche(k,:);
366         Hstar = Hetoilehautgauche(k);
367         deltauxn = DunCoinHautGauche(u,i,j,h,n);
368
369         res = deltauxn - Hstar;
370         r = max(r,res);
371     end
372
373 endfunction
374
375 function r = HdeltauCoinHautDroite(u,i,j,h)
376
377     r = -1000;
378     for k = 1:(Nvectsbord/2)
379
380         n = vectshautdroite(k,:);
381         Hstar = Hetoilehautdroite(k);
382         deltauxn = DunCoinHautDroite(u,i,j,h,n);
383
384         res = deltauxn - Hstar;
385         r = max(r,res);
386     end
387
388 endfunction
389
390 function r = HdeltauCoinBasGauche(u,i,j,h)
391
392     r = -1000;
```

```
393     for k = 1:(Nvectsbord/2)
394 
395         n = vectsbasgauche(k,:);
396         Hstar = Hetoilebasgauche(k);
397         deltauxn = DunCoinBasGauche(u,i,j,h,n);
398 
399         res = deltauxn - Hstar;
400         r = max(r,res);
401     end
402 
403 endfunction
404 
405 function r = HdeltauCoinBasDroite(u,i,j,h)
406 
407     r = -1000;
408     for k = 1:(Nvectsbord/2)
409 
410         n = vectsbasdroite(k,:);
411         Hstar = Hetoilebasdroite(k);
412         deltauxn = DunCoinBasDroite(u,i,j,h,n);
413 
414         res = deltauxn - Hstar;
415         r = max(r,res);
416     end
417 
418 endfunction
419 //////
420 
421 u=0;
422 matDxx = 0;
423 matDyy = 0;
424 matDxy = 0;
425 matDx = 0;
426 matDy = 0;
427 matDvv = 0;
428 matDvvorth = 0;
429 matDv = 0;
430 matDvorth = 0;
431 rhoXmat = 0;
432 rhoYmat = 0;
433 detmat = 0;
434 matGradient = 0;
435 matZeros = 0;
436 //initialisations
437 
438 for i = 1:Npoints
439     for j = 1:Npoints
440         u(i,j) = norm([x(i), y(j)])^2 / 2;
441         //u(i,j) = (x(i)^4 + y(j) ^4)/ 4;
442         matZeros(i,j) = 0
443         matDxx(i,j) = 0;
```

```
444         matDyy(i,j) = 0;
445         matDxy(i,j) = 0;
446         matDx(i,j) = 0;
447         matDy(i,j) = 0;
448         matDvv(i,j) = 0;
449         matDvvorth(i,j) = 0;
450         matDv(i,j) = 0;
451         matDvorth(i,j) = 0;
452         rhoXmat(i,j) = rhoX(x(i),y(j));
453         rhoYmat(i,j) = rhoY(x(i),y(j));
454         detmat(i,j) = 0;
455         matGradient(i,j) = 0;
456         rhoYmatOrth(i,j) = rhoY(x(i),y(j));
457     end
458 end
459
460 u0 = u;
461
462 for k = 1:Niter
463     disp(k);
464
465     matDxx(2:(Npoints-1), 2:(Npoints-1))  = (1/h^2) * (u(3:Npoints,2:(
        Npoints-1)) + u(1:(Npoints-2),2:(Npoints-1)) - 2*u(2:(Npoints-1)
        ,2:(Npoints-1)));
466     matDyy(2:(Npoints-1), 2:(Npoints-1)) = (1/h^2) * (u(2:(Npoints-1),3:
        Npoints) + u(2:(Npoints-1),1:(Npoints-2)) - 2*u(2:(Npoints-1),2:(
        Npoints-1)));
467     matDxy(2:(Npoints-1), 2:(Npoints-1)) = (1/(4 *h^2)) * (u(3:Npoints,3:
        Npoints) + u(1:(Npoints-2),1:(Npoints-2)) - u(1:(Npoints-2),3:
        Npoints) - u(3:Npoints,1:(Npoints-2)));
468     matDx(2:(Npoints-1), 2:(Npoints-1)) = (1/(2*h)) * (u(3:Npoints,2:(
        Npoints-1)) - u(1:(Npoints-2),2:(Npoints-1)));
469     matDy(2:(Npoints-1), 2:(Npoints-1)) = (1/(2*h)) * (u(2:(Npoints-1),3:
        Npoints) - u(2:(Npoints-1),1:(Npoints-2)));
470
471
472     for i = 2:(Npoints-1)
473         for j = 2:(Npoints -1)
474 //             matDxx(i,j) = Dxx(u,i,j,h);
475 //             matDyy(i,j) = Dyy(u,i,j,h);
476 //             matDxy(i,j) = Dxy(u,i,j,h);
477 //             matDx(i,j) = Dx(u,i,j,h);
478 //             matDy(i,j) = Dy(u,i,j,h);
479             matDvv(i,j) = Dvv(u,i,j,h);
480             matDvvorth(i,j) = Dvvorth(u,i,j,h);
481             matDv(i,j) = Dv(u,i,j,h);
482             matDvorth(i,j) = Dvorth(u,i,j,h);
483             //matGradient(i,j) = norm([matDx(i,j) matDy(i,j)],2);
484             rhoYmat(i,j) = rhoY(matDx(i,j), matDy(i,j)) ;
485             rhoYmatOrth(i,j) = rhoY((1/sqrt(2)) * (matDv(i,j) + matDvorth
                (i,j)), (1/sqrt(2)) * (matDv(i,j) - matDvorth(i,j)))
```

```
486             detmat(i,j) = det([Dxx(u,i,j,h) Dxy(u,i,j,h) ; Dxy(u,i,j,h)
                    Dyy(u,i,j,h)]);
487         end
488     end
489
490
491
492     res = rhoYmat .* detmat;
493
494     //update MAA
495     MAA = matDxx .* matDyy - matDxy .* matDxy - rhoXmat ./ rhoYmat - u(
            Npoints/2,Npoints/2) ;
496
497     MA1 = max(matDxx, delta) .* max(matDyy, delta) - min (matDxx, delta)
            - min(matDyy, delta) - rhoXmat ./ rhoYmat - u(Npoints/2,Npoints
            /2);
498
499     MA2 = max(matDvv, delta) .* max(matDvvorth, delta) - min(matDvv,
            delta) - min(matDvvorth, delta) - rhoXmat ./ rhoYmatOrth - u(
            Npoints/2,Npoints/2);
500
501     MAM = min(MA1, MA2)
502     MA = matZeros
503     MA(2:(Npoints-1), 2:(Npoints-1)) = MAA(2:(Npoints-1), 2:(Npoints-1))
            //+ epsilon * S((MAA - MAM)/epsilon)
504
505 //     for i = 2:(Npoints-1)
506 //        MA(1,i) = Hdeltaugauche(u,1,i,h)
507 //        MA(Npoints,i) = Hdeltaudroite(u,Npoints,i,h)
508 //        MA(i,Npoints) = Hdeltauhaut(u,i,Npoints,h)
509 //        MA(i,1) = Hdeltaubas(u,i,1,h)
510 //     end
511 //
512 //     MA(1,Npoints) = HdeltauCoinHautGauche(u,1,Npoints,h)
513 //     MA(Npoints,Npoints) = HdeltauCoinHautDroite(u,Npoints,Npoints,h)
514 //     MA(1,1) = HdeltauCoinBasGauche(u,1,1,h)
515 //     MA(Npoints,1) = HdeltauCoinBasDroite(u,Npoints,1,h)
516 //
517     //update u
518     prevu = u
519     u = u + dt * MA;
520
521     //plot things
522     drawlater
523     clf();
524     plot3d(x(2:(Npoints-1)),y(2:(Npoints-1)),res((2:(Npoints-1)),(2:(
            Npoints-1))));
525
526     //Sgrayplot(x,y,rhoYmat,strf="041",colminmax=[1,64]);
527     //champ1(x,y,matDx,matDy)
528     //Sgrayplot(x,y,rhoYmat .* detmat,strf="041",colminmax=[1,64]);
```

```
529 //     colorbar(0, 2, [1,64]);
530 //     Sgrayplot(x(2:(Npoints-1)),y(2:(Npoints-1)),res((2:(Npoints-1))
        ,(2:(Npoints-1))),strf="041",colminmax=[1,64]);
531     //Sgrayplot(x,y,prevu - u,strf="041",colminmax=[1,64]);
532     drawnow
533     disp(max(prevu-u))
534
535
536     //plot3d(x,y,res);
537     //plot3d(x,y,res);
538 end
```

# 7 Annexe B : Code OpenCL

```
 1
 2 __kernel void MongeAmpere(__global float* u, __global float* xaxis,
       __global float* yaxis) {
 3
 4     const int i = get_global_id(0);
 5     const int j = get_global_id(1);
 6
 7     const int N= get_global_size(0);
 8     const int iOffset = j * N;
 9
10     const float dx = xaxis[1] - xaxis[0];
11
12     const float dt = 0.00001;
13
14     if((i ==0) || (j == 0) || (i == N-1) || (j == N-1)) {
15         return;
16     } else {
17
18         float Dx, Dy, Dxx, Dyy, Dxy = 0.0;
19         float rhoY, rhoX = 0.0;
20
21         float milieu = 0.0;
22
23         Dxx = 1/(dx*dx) * (u[i-1 + (j)*N] + u[i+1 + (j)*N] - 2* u[i + (j)
               *N]);
24         Dyy = 1/(dx*dx) * (u[i + (j+1)*N] + u[i + (j-1)*N] - 2* u[i + (j)
               *N]);
25         Dy = 1/(2*dx) * (u[i + (j+1)*N] - u[i + (j-1)*N] );
26         Dx = 1/(2*dx) * (u[i+1 + (j)*N] - u[i-1 + (j)*N] );
27         Dxy = 1/(4* dx *dx) * (u[i+1 + (j+1)*N] + u[i-1 + (j-1)*N] - u[i
               -1 + (j+1)*N] - u[i+1 + (j-1)*N]);
28
29         milieu = u[N/2 + N * N/2];
30
```

```
31          rhoY =((float) exp((float)( - ((Dx-0.3)*(Dx-0.3) + (Dy+0.3)*(Dy
               +0.3))/0.1))) + 0.1;
32          rhoX =((float)  exp((float)( - ((xaxis[i]+0.3)* (xaxis[i]+0.3)+ (
               yaxis[j]-0.3)*(yaxis[j]-0.3))/0.1))) + 0.1;
33
34          u[i+j*N] = u[i+j*N] + dt * (Dxx * Dyy - Dxy * Dxy  - rhoX / rhoY
               - milieu);
35
36      }
37
38  }
```

```
 1  #ifndef __linux__
 2  #include "stdafx.h"
 3  #else
 4  #include <string.h>
 5  #endif
 6
 7  #include <iostream>
 8  #include "basic.hpp"
 9  #include "cmdparser.hpp"
10  #include "oclobject.hpp"
11  #include "utils.h"
12  #include <math.h>
13
14  using namespace std;
15
16  const int NPOINTS = 256;
17  const float XMIN = -1;
18  const float YMIN = -1;
19  const float XMAX = 1;
20  const float YMAX = 1;
21  const int NITER = 200000;
22  //const float DT = 0.0001;
23
24
25  void generateaxis(cl_float * axis) {
26      for (int i = 0; i < NPOINTS; i++) {
27         axis[i] = i * (XMAX - XMIN)/(NPOINTS-1) + XMIN;
28      }
29  }
30
31  void initubuffer(cl_float * ubuffer, float * xaxis, float * yaxis) {
32
33      int iOffset;
34      for (int i = 0; i < NPOINTS; i++) {
35          for (int j = 0; j < NPOINTS; j++) {
36              iOffset = j * NPOINTS;
37
38              ubuffer[iOffset + i] = (pow(xaxis[i],2) + pow(yaxis[j],2))/2;
39          }
```

```
40      }
41  }
42
43  float ExecuteMongeAmpereKernel(cl_float* xaxis, cl_float* yaxis, cl_float
        * ubuffer, OpenCLBasic& ocl, OpenCLProgramOneKernel& executable)
44  {
45      float perf_start;
46      float perf_stop;
47
48      cl_int err = CL_SUCCESS;
49
50      cl_mem cl_xaxis =
51          clCreateBuffer
52          (
53              ocl.context,
54              CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
55              NPOINTS * sizeof(cl_float),
56              xaxis,
57              &err
58          );
59      SAMPLE_CHECK_ERRORS(err);
60      if (cl_xaxis == (cl_mem)0)
61          throw Error("Failed␣to␣create␣XAXIS␣Buffer!");
62      cl_mem cl_yaxis =
63          clCreateBuffer
64          (
65              ocl.context,
66              CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
67              NPOINTS * sizeof(cl_float),
68              yaxis,
69              &err
70          );
71      SAMPLE_CHECK_ERRORS(err);
72      if (cl_yaxis == (cl_mem)0)
73          throw Error("Failed␣to␣create␣YAXIS␣Buffer!");
74
75      cl_mem cl_ubuffer=
76          clCreateBuffer
77          (
78              ocl.context,
79              CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
80              NPOINTS * NPOINTS * sizeof(cl_float),
81              ubuffer,
82              &err
83          );
84      SAMPLE_CHECK_ERRORS(err);
85      if (cl_yaxis == (cl_mem)0)
86          throw Error("Failed␣to␣create␣U␣Buffer!");
87
88
89
```

```
 90      err = clSetKernelArg(executable.kernel, 0, sizeof(cl_mem), (void *) &
             cl_ubuffer);
 91      SAMPLE_CHECK_ERRORS(err);
 92      err = clSetKernelArg(executable.kernel, 1, sizeof(cl_mem), (void *) &
             cl_xaxis);
 93      SAMPLE_CHECK_ERRORS(err);
 94      err = clSetKernelArg(executable.kernel, 2, sizeof(cl_mem), (void *) &
             cl_yaxis);
 95      SAMPLE_CHECK_ERRORS(err);
 96      //err = clSetKernelArg(executable.kernel, 3, sizeof(float), DT);
 97      //SAMPLE_CHECK_ERRORS(err);
 98
 99      size_t global_work_size[2] = { (size_t)NPOINTS, (size_t)NPOINTS};
100
101      // execute kernel
102      perf_start=time_stamp();
103
104      cl_event event;
105      for (int t = 0; t < NITER; t ++) {
106      err = clEnqueueNDRangeKernel(ocl.queue, executable.kernel, 2, NULL,
             global_work_size, NULL, 0, NULL, &event);
107      SAMPLE_CHECK_ERRORS(err);
108      //cout << "iteration : " << t << "\n";
109      clWaitForEvents(1, &event);
110
111      }
112
113      err = clFinish(ocl.queue);
114      SAMPLE_CHECK_ERRORS(err);
115      perf_stop=time_stamp();
116
117      void* tmp_ptr = NULL;
118      tmp_ptr = clEnqueueMapBuffer(ocl.queue, cl_ubuffer, true, CL_MAP_READ
             , 0, sizeof(cl_float) * NPOINTS * NPOINTS, 0, NULL, NULL, NULL);
119      if(tmp_ptr!=ubuffer)
120      {
121          throw Error("clEnqueueMapBuffer␣failed␣to␣return␣original␣pointer
                 ");
122      }
123
124      err = clFinish(ocl.queue);
125      SAMPLE_CHECK_ERRORS(err);
126
127      err = clEnqueueUnmapMemObject(ocl.queue, cl_ubuffer, tmp_ptr, 0, NULL
             , NULL);
128      SAMPLE_CHECK_ERRORS(err);
129
130      err = clReleaseMemObject(cl_xaxis);
131      SAMPLE_CHECK_ERRORS(err);
132      err = clReleaseMemObject(cl_yaxis);
133      SAMPLE_CHECK_ERRORS(err);
```

```
134     err = clReleaseMemObject(cl_ubuffer);
135     SAMPLE_CHECK_ERRORS(err);
136 
137     for (int i = 0; i < NPOINTS; i++) {
138         for (int j = 0; j < NPOINTS; j++) {
139             cout << ubuffer[i*NPOINTS+j] << "␣";
140         }
141         cout << "\n";
142     }
143 
144     // retrieve perf. counter frequency
145     return (float)(perf_stop - perf_start);
146 }
147 
148 int main (int argc, const char** argv)
149 {
150     //return code
151     int ret = EXIT_SUCCESS;
152     // pointer to the HOST buffers
153     cl_float* xaxis= NULL;
154     cl_float* yaxis= NULL;
155     cl_float* ubuffer= NULL;
156 
157     try
158     {
159         // Define and parse command-line arguments.
160         CmdParserCommon cmdparser(argc, argv);
161 
162         cmdparser.parse();
163 
164         // Immediatly exit if user wanted to see the usage information
                only.
165         if(cmdparser.help.isSet())
166         {
167             return EXIT_SUCCESS;
168         }
169 
170         int width = NPOINTS;
171         int height = NPOINTS;
172 
173         // Create the necessary OpenCL objects up to device queue.
174         OpenCLBasic oclobjects(
175             cmdparser.platform.getValue(),
176             cmdparser.device_type.getValue(),
177             cmdparser.device.getValue()
178         );
179 
180         // Build kernel
181         OpenCLProgramOneKernel executable(oclobjects,L"
                MongeAmpere_Kernels.cl","","MongeAmpere");
182 
```

```
183
184         xaxis = (cl_float*)malloc(NPOINTS * sizeof(float));
185         yaxis = (cl_float*)malloc(NPOINTS * sizeof(float));
186         ubuffer = (cl_float*)malloc(sizeof(cl_float) * width * height);
187
188         if(!(xaxis && yaxis&& ubuffer))
189         {
190             throw Error("Could␣not␣allocate␣buffers␣on␣the␣HOST!");
191         }
192
193         generateaxis(xaxis);
194         generateaxis(yaxis);
195         initubuffer(ubuffer, xaxis, yaxis);
196
197         //printf("Executing OpenCL kernel...\n");
198         float ocl_time = ExecuteMongeAmpereKernel(xaxis, yaxis, ubuffer,
               oclobjects, executable);
199         //printf("NDRange perf. counter time %f ms.\n", ocl_time*1000);
200
201     }
202     catch(const CmdParser::Error& error)
203     {
204         cerr
205             << "[␣ERROR␣]␣In␣command␣line:␣" << error.what() << "\n"
206             << "Run␣" << argv[0] << "␣-h␣for␣usage␣info.\n";
207         ret = EXIT_FAILURE;
208     }
209     catch(const Error& error)
210     {
211         cerr << "[␣ERROR␣]␣Sample␣application␣specific␣error:␣" << error.
               what() << "\n";
212         ret = EXIT_FAILURE;
213     }
214     catch(const exception& error)
215     {
216         cerr << "[␣ERROR␣]␣" << error.what() << "\n";
217         ret = EXIT_FAILURE;
218     }
219     catch(...)
220     {
221         cerr << "[␣ERROR␣]␣Unknown/internal␣error␣happened.\n";
222         ret = EXIT_FAILURE;
223     }
224
225     free( ubuffer);
226     free( xaxis);
227     free( yaxis);
228
229     return ret;
230 }
```